An Open Architecture for Transport-level Coordination in Distributed Multimedia Applications

by
David Edward Ott

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2005

Approved by:

Ketan Mayer-Patel, Advisor

F. Donelson Smith, Reader

Kevin Jeffay, Reader

Prasun Dewan, Reader

Amin Vahdat, Reader

ABSTRACT
DAVID EDWARD OTT: An Open Architecture for Transport-level Coordination in Distributed Multimedia Applications.
(Under the direction of Ketan Mayer-Patel.)

Complex multimedia applications of the future will employ clusters of computing hosts and devices where single endpoint hosts once sufficed. Communication between clusters will require an increasing number of data flows as new media types and sophisticated modes of interactivity continue to be developed.

With an increase in the number of data flows sharing the same forwarding path comes a need for coordinated use of network resources. Unfortunately, modern transport protocols like TCP, UDP, SCTP, TFRC, DCCP, etc. are designed to operate independently and lack mechanisms for sharing information with peer flows and coordinating data transport within the same application.

In this dissertation, we propose an open architecture for data transport that supports the exchange of network state information, peer flow information, and application-defined information among flows sharing the same forwarding path between clusters. Called simply the *Coordination Protocol (CP)*, the scheme facilitates coordination of network resource usage across flows belonging to the same application, as well as aiding other types of coordination.

We demonstrate the effectiveness of our approach by applying CP to the problem of multi-streaming in *3D Tele-immersion (3DTI)*. Laboratory results show that CP can be used to significantly increase transport synchrony among video streams while, at the same time, minimizing buffering delay, maintaining good network utilization, and exhibiting fairness to TCP traffic competing on the same forwarding path. Experiments on the Abilene backbone network verify these results in a scaled, uncontrolled environment.

ACKNOWLEDGMENTS

I would like to express my gratitude to many individuals who assisted me in various ways throughout this work.

First is my advisor, Ketan Mayer-Patel, whose instruction and guidance was instrumental throughout. Thank you. My thanks also to my committee members, Don Smith, Kevin Jeffay, Amin Vadhat, and Prasun Dewan, who gave of their time to assist me at various points during the process and whose suggestions were always valuable.

With respect to equipment and lab support, I thank Don Smith and Kevin Jeffay for graciously providing me with access to the Distributed and Real-time Systems laboratory in Sitterson Hall. Herman Towles and Henry Fuchs provided additional equipment support from the Office of the Future. Kostas Daniilidis from the University of Pennsylvania generously provided yet additional equipment support from the General Robotics, Automation, Sensing and Perception (GRASP) Lab.

My thanks to the technical support staff in the Department of Computer Science, especially Bil Hays, Murray Anderegg, and Mike Stone, for their expertise and timely assistance with issues as they arose.

My thanks to fellow students in the Distributed and Real-time Systems research group whose input and assistance helped me a great deal at various points. Some of these individuals include Felix Hernandez Campos, Long Le, Jay Aikat, Michele Clark Weigle, Mikkel Christiansen, Vivek Sawant, and Mark Parris. Thanks also to Travis Sparks who assisted with the original CP-RUDP prototype.

On a more personal note, I am grateful to my family, especially sisters Jeanne Marie and Rebecca, for their friendship and support. Reg Pendergraph provided a much-needed outsider's perspective and many stimulating technical conversations. Finally, yet perhaps most important of all, I am grateful to my wife, Malai, whose patience and encouragement played an essential role in my successful completion of this challenge.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Its been more than a decade since application developers and researchers made teleconferencing a reality. Early pioneering efforts by researchers like K.A. Lantz [Lan86] and S.R. Ahuja [AEH86] set the stage for a flurry of innovative systems and products in the early 1990s. IBM's *PicTel* was released in 1991, followed by Cornell's *CU-SeeMe* for MacIntosh in 1992 and Novell's *VocalChat* for IPX networks in 1993. In 1996, Microsoft introduced *NetMeeting*, originally from PictureTel's *Liveshare Plus*, followed by VocalTec's *Internet Phone* for Windows in the same year. [Wil04] Each of these systems sought to provide a desktop-to-desktop conferencing experience in real-time that was easy to use, ran on widely used platforms, and required little by way of custom network support.

The late 1990s and early 2000s saw systems of increasing complexity that extended the notion of desktop teleconferencing to new, more sophisticated group environments. One particular system of note in this context is the *Access Grid* introduced by Argonne National Laboratory in 1998. [Gri, CDO+00] The original Access Grid configuration [Ols03] required each participant to have four computing hosts: a display host, a video capture host, an audio capture host, and a control host. With some additional equipment (audio processing hardware, microphones, video cameras, projectors) and middleware software developed by ANL, Access Grid provided a group-to-group collaboration environment with advanced display and interaction modes.

Distributed multimedia applications of the future promise even greater sophistication as *tele-immersion* becomes a driving vision for bringing remote users together. Tele-immersion improves upon collaborative environments like the Access Grid by tracking the presence and movement of individuals and objects, and reconstructing them in a shared 3D display environment. Such systems, which combine both virtual

reality (VR) and digital video into the same environment [LDE+97], are bandwidth intensive and involve a large number of computing hosts and media devices. A prototype of one such system, the *Office of the Future* [RWC+98], is described in Section 1.1.

The evolution of desktop teleconferencing into more involved multimedia environments, and then complex tele-immersion environments, illustrates several important trends in cutting-edge distributed multimedia applications. First is an *increase in computing hosts and application devices.* While a single host was sufficient for a video teleconferencing endpoint with CU-SeeMe, Access Grid required four, and Office of the Future requires many more. Similarly, the number of digital cameras, audio devices, and display devices involved in the application has increased. While an application endpoint was once a single host and a couple of associated devices, it is now a whole *environment* with numerous hosts and a large number of devices.

Second is an *increase in the number of application data flows.* Simple video teleconferencing with CU-SeeMe generated incoming and outgoing audio and video streams and relatively little else. Meanwhile, the Access Grid generated four outgoing and four (or more) incoming video streams, at least one full duplex audio stream, a secondary communication channel for debugging, and possibly other types of interactive data flows. [CDO+00]. Office of the Future requires even more streams as the number of capture, reconstruction, and display hosts and devices is scaled to accommodate larger shared environments and greater freedom of user movement.

Application streams, furthermore, *share complex semantic relationships* with one another. Media data may be layered across streams [MJV96], flows may share temporal relationships and possess synchronization requirements, video streams may share geometric relationships based on camera positioning in the capture environment, text streams may annotate the changing content of media streams, control flows may give instructions for handling stream data based on current application state, etc.

Finally, complex distributed multimedia applications *require substantial network bandwidth.* This follows naturally from the number of application streams employed, and the data types involved (e.g., digital video). Ideally, an application's end-to-end path would be provisioned to comfortably support these demanding requirements. In practice, however, such provisioning may be prohibitively expensive or impossible due to the number of service providers involved in a given end-to-end path. The result is that application designers need to be aware of network resource limitations and design communication schemes that accommodate them in intelligent ways.

Figure 1.1: The Office of the Future.

1.1 Office of the Future

A good illustration of a future distributed multimedia applications is *Office of the Future*, conceived of by Fuchs et al. [RWC+98] at the University of North Carolina at Chapel Hill. In this application, tens of digital light projectors are used to make almost every surface of an office (walls, desktops, etc.) a display surface. Similarly, tens of video cameras are used to capture the office environment from a number of different angles. At real-time rates, the video streams are used as input to stereo correlation algorithms that extract 3D geometry information. Audio is also captured from a set of microphones. The video streams, geometry information, and audio streams are all transmitted to a remote Office of the Future environment. At the remote environment, the video and audio streams are warped using both local and remote geometry information and stereo views are mapped to light projectors. Audio is spatialized and sent to a set of speakers. Users within each Office of the Future environment wear shutter glasses that are coordinated with the light projectors.

The result is an immersive 3D experience in which the walls of one office environment essentially disappear to reveal the remote environment and provide a tele-immersive collaborative space for the participants. Synthetic 3D models may additionally be rendered, incorporating them into both display environments as part of the shared,

collaborative experience. Figure 1.1 is an artistic illustration of the application. A prototype of the application is described in [RWC+98].

From a networking standpoint, the Office of the Future is a challenging application because the endpoints are collections of devices rather than single hosts. Two similarly equipped offices must exchange myriad data streams, involving both heterogeneous data types and complex semantic relationships among streams. While few streams (if any) share a complete end-to-end communication path, all of the data streams span a common shared path between participant Office of the Future environments.

The local network environment of each Office of the Future instance is not likely to be a significant source of congestion, loss, or other dynamic network conditions because it can be provisioned to support the Office of the Future application. The shared Internet path between two Office of the Future environments, however, is not under local control and thus will be the source of dynamic network conditions.

1.2 The Problem of Flow Coordination

A fundamental problem in distributed multimedia applications like Office of the Future is that of **flow coordination**. Such applications employ a large number of flows that share a common forwarding path between remote computing environments. This path, since it typically cannot be provisioned end-to-end, is a dynamic source of network latency, packet loss, and changes in available bandwidth.

Ideally, an application would be aware of changing network conditions and make controlled adjustments to some or all flows to compensate for them. If, for example, more network bandwidth becomes available, then it might choose to apportion this bandwidth to particular flows based on their role in the application at the time. If network delay and loss increase, then the application may choose to adjust the sending rate of less essential data flows until conditions improve. In general, an application will have specific priorities and objectives that it can use to make adjustments for changing network conditions. This strategy includes exploiting inter-stream tradeoffs to use limited network resources as effectively as possible.

Consider once again the Office of the Future. While media streams share a common forwarding path between Office environments, not every stream is of equal priority. The orientation and position of the user's head, for instance, indicates a region of interest within the environment. Media streams that are displayed within that region of interest

may require higher resolutions and frame rates than media streams that are outside the region of interest. Ideally, the application should apportion available bandwidth among streams dynamically as a user's region of interest changes. This is particularly important during periods when bandwidth resources become scarce. Without careful resource allocation, the application as a whole can fail to achieve its desired objective as lower priority streams compromise the performance of higher priority streams within the application.

While application control over network resource allocation in response to changing network conditions is desirable, in practice it is *hardly ever realized*. This is because application flows use transport protocols that:

- Operate in isolation from one another,
- Share no consistent view of network conditions, and
- Fail to respond to network delay and congestion in application-defined ways.

Consider, for example, an application where endpoints use UDP to transmit media data of varying priority levels and TCP to transmit high-priority control messages. During periods of network congestion, media streams may be unaware of network of conditions and continue to send at constant data rates. This lack of responsiveness increases latency and loss for all media streams and, in particular, makes the performance of high priority streams unacceptable. Meanwhile, TCP control flows back off considerably in response to packet loss. Yet high loss rates continue to impact the transmission of control message when, in fact, control messages are drastically needed to make application adjustments on the receiving cluster and to inform the user.

Fundamentally, lack of flow coordination stems from the inherent limitations of today's widely used transport-level protocols. Among these limitations include:

1. Many protocols lack mechanisms to measure and respond to network conditions. *Non-responsive flows*, as they are referred to in [BCC+98], are most often UDP-based and frequently associated with media streaming applications. Such protocols provide no way for an application to respond adaptively to network conditions as they change dynamically.

2. Protocols with congestion response mechanisms achieve inconsistent results across flows. This effect may follow from the fact that not all flows are using the same congestion response algorithm. For example, one flow may be using

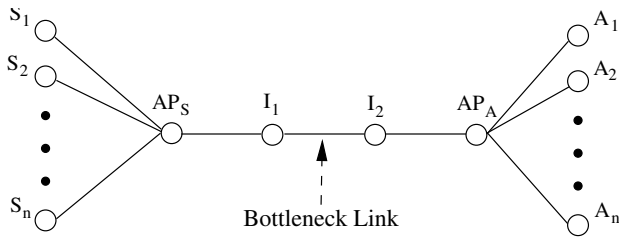


Figure 1.2: A simple distributed application.

TCP Reno while another uses DCCP [KHF04]. Or, two flows may use TCP but with different implementations (e.g., Windows XP and Redhat Enterprise LINUX) or with different options (e.g., SACK, window scaling). Equally as important, roughly identical flows may experience significantly different sets of packet loss events despite sharing the same network forwarding path. This effect is illustrated in Section 1.2.1.

3. Protocols operate without awareness of peer flows. Transport-level protocols typically provide no mechanisms for recognizing peer flows sharing the same bottleneck link, the same intermediary forwarding path, the same end-to-end path, or even the same source or destination hosts. As such, they are unable to exchange information or coordinate data transport in any significant way.

4. Protocols lack mechanisms for configuring adaptive behavior. Protocols with congestion response mechanisms (e.g., TCP [Pos81], SCTP [XMS+00]) typically respond to network conditions in a transparent manner, leaving no higher order control for an application that wishes to modify response behavior in some way.

1.2.1 An Illustration: Peer TCP Flows

To illustrate the problem of coordination, consider a distributed application like the one modeled in Figure 1.2. In this experiment, all flows use TCP to transport exactly the same data in a reliable, congestion responsive manner. Each host, furthermore, uses the same base operating system with the same protocol implementation and the same configuration.

While the network topology consists of only a few intermediary hops between computing clusters, these hops are shared by all flows in the system. In particular, all flows share the same bottleneck link which has been congested using Web background traffic. (Full details of our laboratory testbed and the Web traffic generated are provided in Appendix B.)

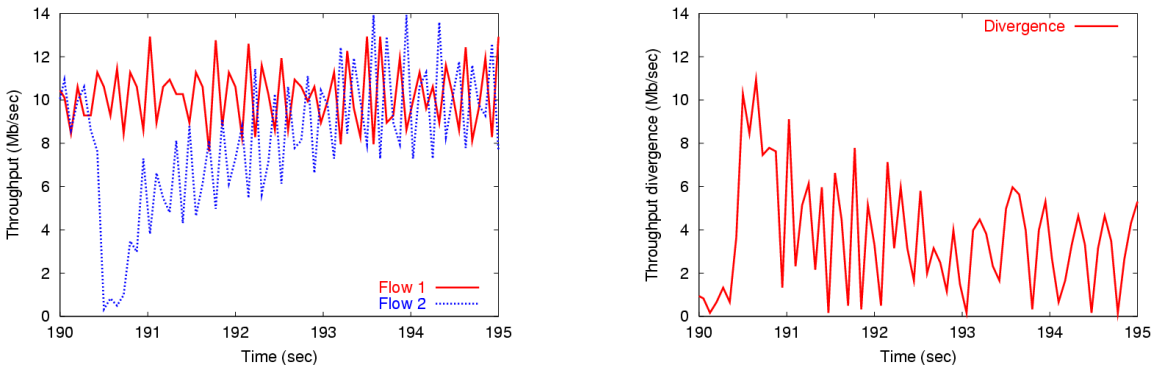


Figure 1.3: (a) Throughput and (b) throughput divergence for two TCP flows.

One might expect flow behavior in this scenario to be markedly similar across flows since:

- The data to be transported is the same for all flows.
- The transport protocol (i.e., TCP) employed is the same for all flows.
- Windows size and other protocol configuration parameters are the same for all flows.
- Network conditions on the shared data path between clusters are the same for all flows.

Given this homogeneity, it would seem reasonable to hypothesize that flow behavior will *naturally* exhibit a high degree of coordination. This is because flows use the same congestion control algorithm to respond to the same conditions on the same bottleneck link.

Figure 1.3 and Figure 1.4 demonstrate that this is not at all the case. In Figure 1.3 (a), we see the difference in throughput for an application consisting of two flows. First, we observe that throughput for each TCP flow generally oscillates over time. Presumably, this follows from additive increases in congestion window size and multiplicative decreases when losses are encountered and congestion avoidance behavior results. What is important to note here, however, is that this oscillation behavior is not synchronized. That is, one flow may be increasing its throughput while another is decreasing it, and vice versa. Likewise, the period between oscillation events may vary both within the same flow and between flows. *All of this implies that the difference in how each flow experiences loss events over the shared data path between clusters is significant.*

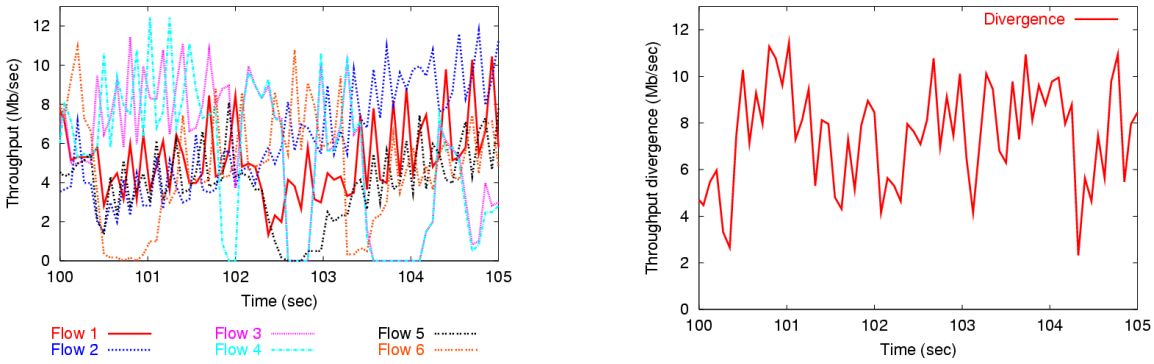


Figure 1.4: (a) Throughput and (b) throughput divergence for six TCP flows.

This conclusion is underscored by the period during which flow 2 receives significantly less bandwidth than flow 1. This period, between seconds 190 and 191, suggests that flow 2 experienced a congestion event that caused it to begin slow start. Meanwhile, flow 1 experienced no such event and thus behaved very differently during the same interval.

This lack of coordination between two identical flows is further illustrated in Figure 1.3 (b) where the difference in bandwidth between the two flows, referred to as *throughput divergence*, is shown. Divergence values can commonly be seen to spike at 6 Mb/sec or more, with a long spike over 8 Mb/sec between seconds 190 and 191. Coordinated flows would ideally show values near zero throughout indicating a high degree of adaptive synchrony.

All of these conclusions are seen even more starkly in Figure 1.4 plots where the number of identical flows in the application has been increased to six. Plot (a) once again shows the considerable differences that exist in throughput for various flows over a 10-second period. For any given interval, one or more flows is likely to receive significantly more bandwidth than peer flows within the same application, while another flow receives significantly less.

This difference is once again underscored by the throughput divergence plot in Figure 1.4 (b). Values in this plot give the maximum throughput difference across all flow pairs within the application for the given point in time. In general, values remain significantly high throughout, suggesting that lack of coordination increases with the number of identical flows within the system.

In summary, Figure 1.3 and Figure 1.4 illustrate the problem of flow coordination for even the most rudimentary of scenarios, i.e., when flows transport artificially homo-

geneous data and use identical transport-level protocols. These flows share the same forwarding path and are thus subject to the same dynamic path characteristics, yet they still exhibit a striking lack of coordination. This is due to the lack of consistency of network information across flows as each flow encounters loss events within the same system somewhat differently.

1.3 Thesis Statement

The thesis of this dissertation may be stated as follows:

> *Using strategically placed information sharing mechanisms within the network, a distributed application can coordinate flows to significantly improve application performance.*

Furthermore, this can be done:

- *Without the use of buffering, scheduling, or shaping mechanisms within the network,*
- *Without disrupting the semantics of end-to-end transport-level protocols, and*
- *While maintaining correct aggregate congestion response behavior over the shared data path.*

In this dissertation, we will describe how information sharing mechanisms can be implemented within each locally administrated cluster and how an application can use such elements to implement flow coordination. Our scheme represents an *open architecture* that puts the application in control over the details of coordination, thus freeing it to respond to network conditions in any way that best meets its objectives.

Since performance improvements are naturally application-specific, we will apply our scheme to a proof-of-concept application in Chapter 6. Our results demonstrate the dramatic improvements that follow from even simple coordination schemes that exploit information consistency to coordinate application flows.

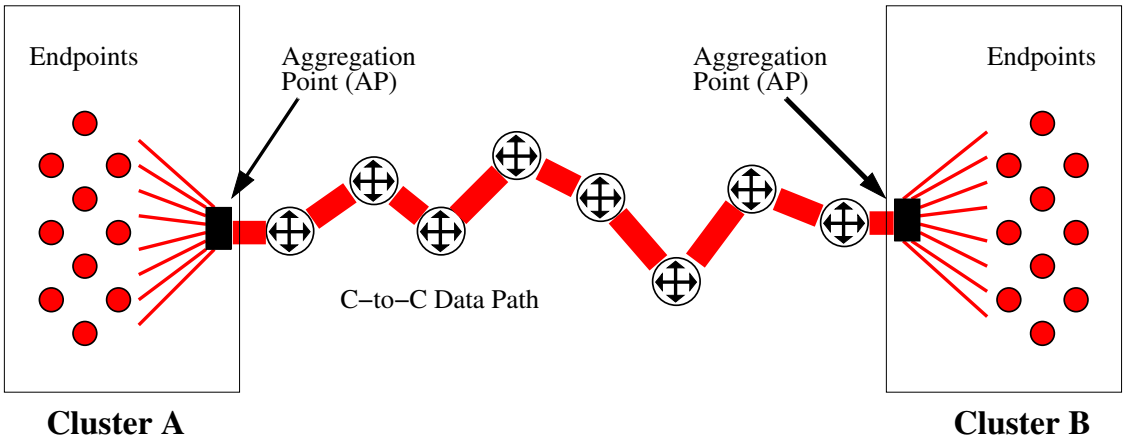


Figure 1.5: Cluster-to-cluster application model.

1.4 The Cluster-to-Cluster Application Model

To better focus the problem of flow coordination in distributed multimedia applications, we define the notion of a *cluster-to-cluster application* in this section. We then go on to discuss various properties of this model architecture that are important to flow coordination.

We define a *cluster* as a set of *endpoints* distributed over a set of *endpoint hosts* (computers or communication devices) and a single *aggregation point* or *AP*. Each endpoint is a process that sends and/or receives data from another endpoint belonging to a remote cluster. The AP, typically a first-hop router, functions as a gateway node. The common traversal path between aggregation points is known as the *cluster-to-cluster data path*. Figure 1.5 illustrates this model.

Two important characteristics of cluster-to-cluster applications are as follows:

- *Minimal communication overhead within a cluster.* Each cluster typically represents a local area network (LAN) and is under local administrative control. As such, it can be provisioned to comfortably support the communication needs of the application and does not represent a significant source of network delay or packet loss. [1]

- *Dynamic conditions on the cluster-to-cluster data path.* In contrast, the cluster-to-cluster data path is shared with other Internet flows and typically cannot be

[1] Wireless clusters have somewhat different assumptions and are not treated here.

provisioned end-to-end. Hence, it represents a significant source of network delay and packet loss for application packets.

While few application flows share the exact same end-to-end path in a cluster-to-cluster application, all flows are subject to the same network delay, delay jitter, packet loss, and other performance characteristics. This is because *intra-cluster* network effects are largely insignificant compared to *inter-cluster* network effects. This is especially true as the number of forwarding nodes along the shared path grows, the physical distance traversed between clusters increases, bottleneck links are encountered on the path, and the number of Internet flows competing with application flows increases.

Two more properties of the cluster-to-cluster application model are worth noting. These include:

- *Flow convergence.* Each AP represents a natural point at which application flows originating from various endpoints within a cluster converge to the same forwarding node.

- *Shared network resources.* Since all flows share the same intermediary path between clusters, network resources are naturally shared among flows. Thus, an increase in bandwidth usage by some flows under conditions of limited bandwidth will result in a decrease in bandwidth available to other flows sharing the path.

Each of these characteristics play a significant role in the problem solution presented in Chapter 3. In particular, they highlight the need for coordinating bandwidth usage among flows and point out an important architectural feature that can be exploited in accomplishing this objective; namely, that of flow convergence within the local environment at the aggregation point.

1.5 Characterizing Cluster-to-Cluster Flows

Flows within a cluster-to-cluster application share several properties significant to this dissertation. Here we divide them into several categories: flow heterogeneity, peer flow relationships, and network resource usage.

1.5.1 Flow Heterogeneity

Flows within a cluster-to-cluster application exhibit heterogeneity on at least two levels:

1. Data heterogeneity. Complex multimedia applications may employ flows with a variety of data types. Some examples include images, text, audio, video, haptic data, geometry data, and control data. Each data type presents a unique set of transport requirements. For example, video streaming often requires significant bandwidth while haptic data is less bandwidth intensive but very sensitive to delay and delay jitter. In contrast, control data may be far less bandwidth- or delay-sensitive, but it places a high demand on data integrity and reliable delivery.

2. Transport-level protocol heterogeneity. Each data flow must be matched with a transport-level protocol appropriate to its requirements. Continuous media types like audio and video, for example, may require a streaming protocol like RTSP [SLR98] with specially designed session control and synchronization features. Control data, on the other hand, may use TCP [Pos81] which provides reliable, in-order delivery semantics.

Data and transport-level heterogeneity imply the need for cluster-to-cluster application transport services to be managed *independently*, on a per-flow basis. This is because different flows have different transport requirements, and no one transport-level protocol can satisfy all sets of requirements.

1.5.2 Peer Flow Relationships

Flows within a cluster-to-cluster application, despite their heterogeneity, share peering relationships in various ways. In this section, we identify several types of relationships important to the problem of flow coordination.

1. Semantic relationships among flows. Flows in a cluster-to-cluster application may share complex semantic relationships with other flows. An application, for example, may divide complex media objects into multiple streams with specific encoding relationships. Different media streams may share temporal relationships and require synchronization mechanisms for an orchestrated delivery at the destination cluster. Flows may transport data with geometric relationships based on device positioning within the media capture environment. Control or annotation data may describe the content of other data flows. Priority relationships may exist as some flows transport pri-

mary data while other flows transport secondary or predictive data. Such relationships can be complex and change dynamically to reflect changing application state. Furthermore, they are known only to the application. This implies the need for application control over flow coordination mechanisms, as well as considerable flexibility.

2. Common application goals. While flows in a cluster-to-cluster application can be both numerous and diverse, each belongs to the same application. As such, each shares the same set of global application goals which take precedence over any single flow's individual requirements. For example, an application may be designed to present a complex information visualization environment that is responsive to certain types of user feedback. Particular flows within the application each have data transport needs, but the overall quality of presentation and user responsiveness is the primary consideration. For this reason, all flows share a vested interest in cooperating to achieve the application's global objective.

3. Shared intermediary path. Flows in a cluster-to-cluster application share a common intermediary path between clusters. Whenever resources are limited, the behavior of any individual flow within the application impacts directly peer flows within the same application. A flow that is unresponsive to network congestion events, for example, may negatively impact all other flows by taking a disproportionate share of bandwidth and making conditions persist. Meanwhile, responsive flows may continue to reduce their sending rate, but with little effect. In a different scenario, a flow that takes less than its fair share of bandwidth may naturally make more bandwidth available to other flows in the same application.

While data and transport-level heterogeneity imply the need for cluster-to-cluster application transport services to be treated on an *individual* flow basis, peer flow relationships imply the need for these services to be treated on an *aggregated* basis. That is, flows sharing a common global objective, a common intermediary forwarding path, and particular semantic relationships would best be managed using a coordination scheme that can consider the aggregate effect of various resource allocation policies. This is because performance depends not on the transport success of any particular flow, but upon the overall success of the right flows in concert with one another.

1.5.3 Network Resource Usage

Finally, cluster-to-cluster application flows share two common properties with respect to network resource usage.

1. Transport requirements exceed available resources. While the precise number of flows and bandwidth requirements depend entirely on the application, it is not unreasonable to expect most cluster-to-cluster applications to be naturally *high-bandwidth* in character. Specifically, such applications are likely to have bandwidth requirements that, at least to some extent, exceed the resources available to them on the shared cluster-to-cluster data path. Recall, as mentioned above, the shared cluster-to-cluster data path typically cannot be provisioned end-to-end and is shared with an indeterminate number of public Internet flows. Furthermore, the nature of multimedia data types (e.g., digital video) naturally lends itself to high bandwidth requirements.

2. Complex adaptation requirements. With limited resources comes the need for adaptive behavior that takes into account the global objectives of the application. In part, this is a problem of dynamic resource allocation among flows. An application must be able to apportion limited network resources (i.e., bandwidth) to individual flows in a way that best serves its global objectives at the time. But this is also a problem of enabling individual flow adaptation. Given appropriate information about network conditions, flows may exhibit adaptive behavior individually on various levels. For example, a flow may be able to modify the amount of data to be sent by altering encoding strategies, modifying compression techniques, changing sampling rates, etc.

These properties imply the need for mechanisms that inform application flows of changing network conditions on the cluster-to-cluster data path, and facilitate prioritized bandwidth allocation on a global level.

1.6 Design Goals

In this section, we identify a number of important design goals for our solution to the flow coordination problem described in Section 1.2. These goals help to motivate our approach described briefly in Section 1.7, and at length in Chapter 3 and Chapter 4.

1. Information sharing among flows. The problems described in Section 1.2 imply the need for mechanisms to exchange information among flows. Such information must, among other things, include information about network conditions on the shared cluster-to-cluster data path, and information about peer flows in the same application. In addition, these mechanism might allow an application to designate other types of information to be shared as well. *Consistency of information* is an essential property for enabling coordinated flow behavior.

2. Preserving transport-level protocol semantics. Data heterogeneity within a cluster-to-cluster application implies that different transport-level protocols will be employed, each with somewhat different semantics. A solution to the flow coordination problem should address the cluster-to-cluster concerns of information sharing and resource allocation among flows while still preserving the end-to-end concerns of transport-level protocol semantics. That is, coordination mechanisms may enhance, but never obstruct, end-to-end transport-level protocol semantics. For example, reliable, in-order delivery semantics (as provided by TCP) should still be achievable after coordination mechanisms have been added to the cluster-to-cluster application architecture.

3. Leveraging application-level adaptation. One approach to solving the flow coordination problem is to use in-network mechanisms that are transparent to application endpoints. While this approach has the benefit of not requiring changes at each endpoint, it fails to leverage application-level adaptation fully. In other words, by sharing information with each endpoint about network conditions, an application can assist with adaptation by adjusting the amount of data it produces for transmission. This can be done, for example, by modifying its encoding techniques, compression strategies, media sampling rates, etc. Additionally, endpoint-centered adaptation avoids the need for complex buffering and scheduling schemes within the network which unnecessarily inhibit forwarding performance.

4. Aggregate measurement of network throughput, delay, loss, and available bandwidth. An important component of flow coordination is obtaining information on network delay, packet loss, and aggregate bandwidth available on the shared data path between clusters. Rather than letting individual flows measure these conditions independently, it would seem advantageous to measure them once with a single mechanism. The information could then be disseminated to all flows in a consistent and timely manner. This not only relieves individual transport-level protocols of the measurement burden, it insures that measurement results are consistent across all flows.

5. Aggregate congestion responsiveness. It is important for aggregate application traffic to be responsive to congestion within the network for three reasons. First, it prevents unfairness to competing flows sharing the same network path. Second, it prevents the possibility of congestion collapse [FF99]. Third, it minimizes unnecessary loss and retransmissions for flows within the application. *TCP-compatibility* [FHPW00] is one way to measure congestion responsiveness and will be used throughout this dis-

sertation.

6. Flexibility for the application. A flow coordination architecture needs to provide flexibility for the application for several reasons. First, the requirements for coordination depend entirely on the flows involved and the nature of the application itself. One application may require some form of token-passing to implement flow coordination in a video conferencing application, while another requires a class-based hierarchical scheme in a gaming application. Second, a coordination scheme may change dynamically over time as application state changes and user input modifies an application's operational mode. Thus, an application should be free to exploit trade-offs without constraint. That is, a coordination mechanism should not preclude dynamic changes in bandwidth usage among flows, or enforce any particular scheme for establishing bandwidth usage relationships between flows. The application should be free to implement whatever adaptation policy is most appropriate using whatever means is most appropriate.

1.7 The Coordination Protocol (CP)

Our solution to the problem of flow coordination in cluster-to-cluster applications is called the *Coordination Protocol*, or *CP*. CP operates between the network layer and transport layer, making it transparent to IP routers on the cluster-to-cluster forwarding path while preserving end-to-end semantics for CP-based transport protocols.

Data packets transmitted by an endpoint include a CP header that identifies the cluster application and the source flow. Additionally, it may contain state information to be exchanged with peer flows in the same cluster. The local AP provides special handling for CP packets, in part, by storing the state information to be shared with other flows in a table for future reference.

The AP keeps a table of bandwidth usage statistics on flows in the same cluster-to-cluster application, also tracking the number of flows and aggregate bandwidth usage by the cluster-to-cluster application as a whole. It also conspires with the remote AP to measure network conditions (network delay, packet loss, and available bandwidth) across the shared data path in the following way.

When an AP receives a data packet from an endpoint participating in the cluster-to-cluster application, it processes the incoming information and then overwrites the CP header with network probe information. This piggybacked information is received

by the remote AP and used to collect network statistics. Since packets are available for the same purpose on the return path, the two APs can establish a probe exchange protocol that monitors network conditions on the shared path in considerable detail as the application runs.

To estimate bandwidth available to the application, we use a TCP modeling equation that takes as input current network loss events, round trip time, average packet size, and several other values. In particular, we make use of the TFRC [HFPW03] equation, giving a throughput estimate that is both gradually responsive to network congestion and TCP-compatible. In addition, this dissertation presents techniques to extend single-flow modeling equations to multiple-flow scenarios. This allows cluster-to-cluster application flows to receive an aggregate bandwidth equivalent to the number of independent flows in the application, rather than attempting to multiplex a single flow.

A packet originating from a remote endpoint and traversing the path toward a particular local endpoint will be processed by a receiver's local AP as the last hop on the forwarding path. The AP will place information in the CP header, and then forward the packet to the destination endpoint. A consistent view of network conditions across flows follows from the fact that the same information is shared among all endpoints.

Transport-level protocols at application endpoints are built on top of CP. Using information on aggregate bandwidth availability, loss rate, round trip time, number of application flows, etc., as well as various application configuration parameters, the transport protocol can choose an appropriate sending rate that reflects an application's global coordination strategies. Information on peer flows and network conditions is also made available to the application layer directly, allowing it to modify data encoding parameters or perform others types of adaptive behavior.

While CP provides the essential building blocks to enable cluster-to-cluster flow coordination, the implementation of a particular *flow coordination scheme* is left to the application. This is necessarily the case since it alone knows the nature and function of various data flows, the semantic relationships between them, and how best to use limited bandwidth during any given time interval.

1.8 Contributions

The contributions of this dissertation to the field of Computer Science are as follows:

- We identify an important class of forward-looking distributed multimedia applications known as *cluster-to-cluster applications* and describe their generalized characteristics.

- We define the *flow coordination problem*. This problem is fundamental to cluster-to-cluster applications, but also of interest to multiflow Internet applications generally.

- We propose a novel open architecture, called the *Coordination Protocol (CP)*, that solves the flow coordination problem in cluster-to-cluster applications.

- We identify the *multiple flowshare problem* and solve it using a novel technique called *bandwidth filtered loss detection (BFLD)*. Using this technique, almost any single-flow congestion control algorithm can be scaled to a multiple-flow cluster-to-cluster application context.

- We implement an experimental prototype of CP and evaluate various aspects of its performance. Our implementation includes both kernel extensions for FreeBSD software routers and a reliable transport protocol called CP-RUDP that uses CP information to perform coordinated adaptation.

- We demonstrate how CP can be applied to the problem of multi-streaming in *3D Tele-immersion (3DTI)*, a complex cluster-to-cluster application developed at UNC, for dramatically improved communication performance.

1.9 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes background and related work. Chapter 3 presents the Coordination Protocol in detail, with a focus on state table design and how coordination schemes can be constructed. Chapter 4 presents CP congestion control mechanisms. This includes our scheme for applying single-flow congestion control algorithms to the multiple-flow context. In Chapter 5, we discuss aggregation point (AP) implementation and performance. In Chapter 6, we describe how CP can be applied to solve the problem of synchronized multi-streaming in 3D Tele-immersion (3DTI). Chapter 7 presents some conclusions and future work.

Chapter 2

Related Work

In this chapter, we discuss related work. First, we describe some well-known *in-network* approaches for apportioning bandwidth among flows and consider whether they can be applied to solve the problem of flow coordination in cluster-to-cluster applications. Next, we consider the issue of aggregate congestion control. This includes several schemes that have been proposed for applying congestion control to flow bundles, most notably the *Congestion Manager* by Balakrishnan *et al.* [BRS99]. Next, we discuss the issue of bandwidth estimation. Our focus here is on TCP-based modeling equations and how they may be used to estimate a sending rate given particular input parameters describing network path conditions. *TCP-compatibility* is an important property of the calculated rate, one which ensures both stability and fairness to competing flows on the Internet path. Finally, we briefly look at programmable network architectures, including *active networks* [TW96] and *ephemeral state processing* [CGW02]. These ideas provide useful background to our approach described in Chapter 3.

2.1 In-network Bandwidth Sharing Approaches

The problem of flow coordination in distributed multimedia applications is principally concerned with how to apportion aggregate bandwidth among application flows sharing the same network path. Several approaches to this problem have been suggested over the years that rely on mechanisms within the network. These approaches are often desirable because they require few, if any, modifications to end systems hosting flow endpoints. Here, we consider two such approaches: *differentiated services* and *traffic shaping*.

2.1.1 Differentiated Services

From a high level viewpoint, one may wonder whether the problem of flow coordination in distributed multimedia applications hasn't already been solved by previous work in *Quality of Service (QoS)*. Some examples of such work include [FJ95, Zha95, GGPS96, PJS99]. Broadly stated, QoS provides a framework for provisioning network resources (e.g., buffering, bandwidth) along the forwarding path between endpoints to insure minimal throughput and bounded delay.

In particular, consider *differentiated services* [BCD+98]. *Diffserv*, as it is commonly referred to, provides a framework for offering tiered service levels at forwarding nodes based upon pre-arranged customer agreements. In general, a customer and provider may negotiate a service level (e.g., leased line emulation, better than best-effort) for their particular traffic profile and pay accordingly for the service. Traffic falling within the profile is guaranteed that service level while traffic exceeding that profile may be treated in a variety ways depending upon the provider's infrastructure and policy (e.g., best effort forwarding, dropping).

Diffserv works by associating packets with a particular pre-established Service Level Agreement (SLA). Packets are marked using a subset of bits in the Type of Service (TOS) field of the IP header known as the DS field. A Service Level Specification (SLS), which includes a Traffic Conditioning Specification (TCS), gives a set of detailed service parameters that include expected throughput, drop probability, latency, service scope (ingress and egress points), etc. An SLA can take many forms, but generally provides some sort of bandwidth allocation characterized by the parameters of a leaky token bucket [Tan96].

Diffserv, and QoS more generally, might ideally be used to provision aggregate C-to-C application traffic and avoid the need for congestion response and resource allocation mechanisms entirely. Purchasing a service agreement to serve this purpose for high-bandwidth applications is likely to be prohibitively expensive, however. In part, this is because a complex cluster-to-cluster application will employ many flows and require considerable network resources. But it is also because service agreements need to be negotiated with providers spanning the entire forwarding path. For clusters separated by large geographic distances, there can be many service providers involved.

A more economical solution would be to use diffserv to provision for the minimum bandwidth required for the lowest acceptable application performance level, and then make use of best-effort, shared bandwidth whenever possible. In this case, how to

provide coordinated congestion response and resource allocation among flows remain important open problems since application demands exceed provisioning. Neither diffserv, nor QoS generally, provide a framework for solving these problems.

Another issue is that of flexibility. As mentioned above, using a single SLA to specify a minimum service level for cluster-to-cluster application traffic does little to eliminate the need for flow coordination. In an alternative use of diffserv, an application might employ multiple SLAs to provide different service levels for different application flows depending on their role within the application. For example, high priority streaming flows might be associated with an SLA that provides considerable guaranteed bandwidth and minimal delay while low priority bulk data flows are associated with an SLA that provides merely best-effort forwarding. By associating flows with different service levels, an application effectively provisions flows differently and thus provides a naturally tiered (and to some extent coordinated) use of network resources.

This approach is not without its problems, however. Provisioning each flow with its own SLA might not be cost effective. Furthermore, changes in application architecture (i.e., number of devices in either cluster, flow endpoints, flow profiles, etc.) would require a new set of SLAs. To avoid this problem, an application might employ a set of SLAs on an *aggregate* rather than individual flow basis. That is, an application's many flows would be associated with a smaller number of SLAs such that many or all SLAs service more than one flow. How to coordinate changes in SLA association dynamically, and how to handle SLAs that provision inadequate bandwidth for the number of flows associated with it are both open questions. Once again, neither diffserv nor QoS more generally provide a framework for solving these problems.

Finally, a practical consideration should be noted. At the date of this writing, diffserv has yet to be widely deployed on today's production Internet. Nor is there a movement to realize widespread deployment in the near future. Without such deployment, cluster-to-cluster path provisioning using diffserv remains an academic exercise. This dissertation, in contrast, offers a real-world solution that is readily deployable in the present.

2.1.2 Traffic Shaping

Another approach that might be considered is that of *traffic shaping*. In this approach, traffic entering the network through a particular forwarding node is modified to conform to a specification or profile. This profile can, as with a Traffic Conditioning

Specification (TCS) in diffserv, be expressed as a static set of parameters like those of a leaky token bucket [Tan96]. These include an average output rate (the rate at which tokens are generated) and a maximum burst size (the number of credits or tokens that can be stored in the bucket). In fact, diffserv mechanisms may include traffic shapers. Diffserv, however, is distinct from traffic shaping in that it deals with service level agreements while traffic shapers are merely mechanisms that modify the profile of incoming traffic to a particular output specification.

Traffic shaping in a cluster-to-cluster application would logically be done at each AP. Unlike simple leaky bucket rate regulation which provides a constant average output rate, however, each traffic shaper is charged with estimating dynamically an output rate that reflects current network path conditions. It then buffers packets from individual flows and transmits them at the target rate in a way that is responsive to conditions on the cluster-to-cluster data path. This need for congestion control follows partly from the fact that no provisioning is employed in this approach, and partly from the fact that application traffic will share the cluster-to-cluster data path with an indeterminate number of other flows on the public Internet. Without dynamic adjustments in output rate, an application may cause excessive packet loss when network congestion occurs, fail to utilize bandwidth when it becomes available, or cause unfairness to competing flows on the same forwarding path.

The important open problem in this approach, then, is that of determining a congestion responsive output rate that reflects current conditions along the shared forwarding path between clusters. Traffic shaping, in and of itself, provides no solution to this problem. In contrast, this dissertation describes mechanisms in Chapter 3 and Chapter 4 for accomplishing this task in an efficient and non-intrusive way. (Non-intrusive refers to the fact that no additional packets are introduced into the network in the process.)

Assuming a solution to the problem of output rate adjustment, another serious problem is that of coordinating bandwidth distribution across application flows. Given an output rate that is less, either temporarily or on an average, than the aggregate flow input rate, how can bandwidth be apportioned among flows in a way that reflects changing application priorities and inter-stream trade-offs?

What is needed are additional mechanisms within the shaper. One such approach is to employ a hierarchical scheduling scheme (hierarchical round robin [KK90], weighted fair queuing [PG93], weighted round robin [KSC91], class-based queuing [FJ95], etc.).

In class-based queuing (CBQ), for example, traffic is divided into classes (flat or hierarchical) and given separate queues at the output link. Each queue is assigned a bandwidth share that is enforced by a link-scheduler when limitations exist on available bandwidth due to network congestion. The scheme also requires a packet classifier that identifies a packet's class and queues it appropriately upon arrival.

While this approach addresses the problem of flow coordination to some extent, we believe the solution is less than ideal for several reasons. First, the approach does not lend itself to dynamic reconfiguration. As flows within a complex cluster-to-cluster application change in the amount and priority of data to be sent, classifier, queuing, and scheduling mechanisms operate according to a static configuration. Even if this configuration could be changed dynamically, such changes might cause interruptions in service and would generally be problematic to implement.

An equally serious problem is that of transparency. A traffic shaper is intended to operate in a transparent manner with respect to individual flows. While potentially a desirable feature when flows are unrelated and relative priorities static, cluster-to-cluster application flows require information on network performance (i.e., available bandwidth, loss rates, etc.) to make dynamic adaptation decisions at the source. These decisions take into account semantic relationships among flows, changing priority levels, and salient aspects of application state, all of which cannot be known by the scheduler. For example, a flow may adjust its media encoding strategy at key points given changes in available bandwidth and a particular user event.

Finally, the approach relies on buffering at the shaping node to adjust bandwidth usage among flows. This necessarily results in additional network delay as data packets for some flows wait in particular queues to be scheduled. Clearly this is a disadvantage for real-time interactive applications where network delay and jitter are critical performance parameters. Much like the authors of the Congestion Manager (discussed in Section 2.2.2), we believe that application control over data transmission events represents a better design strategy than the extensive use of buffering within the network.

2.2 Aggregate Congestion Control

The need for congestion control in networked applications is discussed at length in [FF99]. In general, the problem with *unresponsive flows* (flows that do not reduce their level of bandwidth usage when subjected to packet drops) on the Internet is first, the

danger of congestion collapse, and second, unfairness to competing flows.

Congestion collapse occurs when an increase in network load results in a decrease of useful work done by the network. In *classical* congestion collapse, for example, flows continually retransmit packets but fail to achieve goodput levels (bandwidth delivered to the receiver excluding duplicate packets) in keeping with the bandwidth used at the sender. Floyd and Fall [FF99] identify several other types of congestion collapse, including collapse from undelivered packets, fragmentation-based collapse, collapse from increased control traffic, and collapse from stale or unwanted packets.

The notion of *fairness* pertains to bandwidth usage relative to competing flows sharing the same network link. A flow is unfair if it uses a disproportionate amount of bandwidth compared to peer flows. This relativistic definition naturally begs the question, "How much bandwidth do peer flows use?" Given the widespread use of the *Transmission Control Protocol (TCP)* [Pos81] by Internet applications, including the World Wide Web [CMT98, SCJO01], the prevailing standard for determining fairness is *TCP-compatibility* or *TCP-friendliness*. A flow that is TCP-compatible sends at a rate that does not exceed the rate of a conformant TCP flow under the same network conditions. [FF99].

The challenge in cluster-to-cluster applications is to regulate an application's traffic so that it is congestion responsive on an *aggregate* level. This not only prevents unnecessary packet loss for the application's flows, it reduces the potential for congestion collapse and provides fairness to flows from independent applications sharing various portions of the cluster-to-cluster data path. Addressing this problem on the aggregate (as opposed to individual) flow level serves two purposes. First, it allows us to provide a single congestion control mechanism for all flows. Second, it allows flexibility in apportioning bandwidth among flows without affecting overall aggregate traffic behavior. (This is addressed more fully in Chapter 3.)

One further consideration is that of *multiple flowshares*. A multiple-flow cluster-to-cluster application must not only be responsive to network congestion on an aggregate level, it should receive the bandwidth equivalent to the number of participating flows. That is, an m-flow application should, as an aggregate, receive bandwidth equivalent to m congestion-responsive flows or simply m flowshares. While receiving less has little adverse effect on the network, it results in significant unfairness to the application. (This will be addressed more fully in Chapter 4.)

2.2.1 Flow Segmentation and Bundling

One possible approach to providing congestion control for flow aggregates is to multiplex the flows of a cluster-to-cluster application together into a single congestion controlled flow between aggregation points, and then to demultiplex them at the remote cluster. This could be done explicitly at the application-level or transparently by a mechanism implemented at the AP. This approach is taken by [KW99] in their work on *TCP trunking* for connections that traverse a common backbone path.

Kung and Wang define a TCP trunk as "an aggregate traffic stream whose data packets are transported at a rate dynamically determined by TCP's congestion control." Individual flows sending data along the same intermediary path do so using whatever transport-level protocol is appropriate for their purposes. When the packets reach the endpoint of the trunk, they are buffered until they can be sent by a single *management connection* to the remote endpoint of the trunk and then forwarded to their destination. What makes the trunk congestion responsive is the fact that the management connection employed is a TCP connection.

Another variation of this approach, known as *aggregated TCP (ATCP)*, is presented in [PCN00]. In this approach, an end-to-end connection is divided into a *local subconnection* with the portal router and a shared *remote subconnection* between this router and a commonly accessed remote host. (In the context of a cluster-to-cluster application, the remote subconnection might further be divided into an intermediary subconnection between APs, and a remote subconnection between the remote AP and endpoint host.) While the authors' original intention is to improve the performance of TCP connections by growing congestion windows more quickly and using persistent connections, the approach might be applied to cluster-to-cluster traffic as a way of introducing aggregate congestion control.

Each of these approaches, when applied to the cluster-to-cluster application context, present significant difficulties. First, the approach reduces aggregate application traffic to a single flowshare as multiple flows utilize a single management or remote subconnection. We argue in Chapter 4 that limiting aggregate cluster-to-cluster 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, the approach fails to inform cluster-to-cluster application endpoints of current network performance. Much like traffic shaping, the congestion controlled man-

agement connection between clusters operates in a manner transparent to each flow source. As such, endpoints receive no information about network conditions (available bandwidth, loss rates, etc.) that is crucial in order to provide any kind of adaptive response. In this way, application endpoints cannot fully exploit specific inter-stream adaptation schemes of the type described in Chapter 1.

Third, this approach may result in substantial end-to-end delay as application packets are buffered at the trunk node waiting to be forwarded in a congestion controlled manner. As mentioned in Section 2.1.2, this is a clear a disadvantage for real-time interactive applications where network delay and delay jitter are critical performance parameters.

Fourth, the end-to-end semantics of the individual flows are not preserved if the communication is segmented into subconnections (i.e., endpoint to AP, AP to AP, AP to endpoint). For example, reliability semantics dictate that an acknowledgment received by a sender indicates that a receiver has successfully received the transmitted data. In the segmentation approach, however, an acknowledgment may inform an endpoint only that the data was successfully transmitted to the next multiplexing agent. There is no way to know whether the data was actually received by the destination endpoint on the remote cluster.

Finally, the approach once again fails to provide a framework for coordinated bandwidth allocation across flows. This problem can be solved to some degree by using the same scheduling mechanisms described in Section 2.1.2 at the multiplexing point, but with the same drawbacks also described.

2.2.2 Congestion Manager

The *congestion manager (CM)* architecture, proposed by Balakrishnan et al. in [BRS99], 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.

The CM architecture consists of a sender and a receiver. At the sender, a *congestion controller* adjusts the aggregate transmission rate based on its estimate of network congestion, a *prober* sends periodic probes to the receiver, and a *flow scheduler* divides available bandwidth among flows and notifies applications when they are permitted to send data. At the receiver, a *loss detector* maintains loss statistics, a *responder*

maintains statistics on bytes received and responds to CM probes, and a *hints dispatcher* sends information to the sender informing them of congestion conditions and available bandwidth. An API is presented that allows an application to request information on round trip time and current sending rate, and to set up a callback mechanism to regulate send events according to its apportioned bandwidth.

It should be noted that CM is solving the problem of providing aggregate congestion control for flows that share the entire end-to-end path. That is, all flows share the same source and destination hosts. In contrast, flows in the cluster-to-cluster context share the same *intermediary path* but not the same end-to-end path. That is, flows share a significant portion of the forwarding path, but not the entire path end-to-end. Another important contrast is that flows in CM are not necessarily part of the same application, while in cluster-to-cluster applications they are.

To some extent, the work presented in this dissertation represents our proposal for applying CM concepts to the cluster-to-cluster application model. We agree with CM's philosophy of putting the application in control, though for CM this means allowing unrelated flows to know the individual bandwidth that is available to them, while for cluster-to-cluster applications it means allowing endpoints to know the aggregate bandwidth available to the application. Furthermore, we believe CM's notion of using additional packet headers for detecting loss and identifying flows is a good one, and this is reflected in our own architecture as described in Chapter 3 and Chapter 4.

On the other hand, applying the CM architecture to the cluster-to-cluster application context is not without its problems and issues. First, CM's use of a flow scheduler to apportion bandwidth among flows is problematic in the cluster-to-cluster context for many of the same reasons given in our discussion of traffic shaping. Because cluster-to-cluster applications can have complex schemes for adding and deleting flows, 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. As a result, characterizing each flow's rate requirements is difficult to do *a priori*. This kind of characterization is required with CM because individual flow 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 well-characterized flows and static priorities and build a hierarchical schedule for bandwidth allocation, this approach is less suitable in

the more dynamic cluster-to-cluster context.

Likewise, CM's callback structure for handling application send events is difficult to implement in the cluster-to-cluster application context. This is because in CM, flows share the same sending host. For senders on the same host, a callback architecture is reasonably implemented as a simple system call provided by the OS. In contrast, individual flow endpoints of a cluster-to-cluster application commonly reside on different computing and communication devices. A callback scheme using send notification messages from the AP to various application endpoints would result in significant communication overhead, making it impractical. Our approach, as will be seen in Chapter 3, is to instead provide individual endpoints with global information about aggregate network performance of the larger application and then allow each flow to independently determine an appropriate share of the aggregate bandwidth using an application-specific configuration. This allows for complex adaptation schemes that would be difficult to reconcile in a centralized scheduler. The onus is now on the application to ensure that individual endpoint decisions result in the appropriate aggregate behavior.

Finally, CM is designed to multiplex a single congestion responsive flow among application flows sharing the same end-to-end path. As argued with the multiplexing approach, it may be undesirable to constrain a cluster-to-cluster application which commonly employs a large number of flows to a single flow share. Our solution allows aggregate cluster-to-cluster traffic to use multiple flow shares while remaining congestion responsive.

2.3 Bandwidth Estimation

Coordinating flows in a cluster-to-cluster application involves knowing about the network resources currently available to the application and apportioning them among flows in a way that best meets problem-specific objectives. In this section, we consider background work on the issue of estimating the amount of network resources available to the application. Specifically, we look at recent work in equation-based congestion control, including both a TCP modeling equation and a transport-level protocol (TFRC) that applies that model to the problem of smoothed, congestion responsive data transport.

2.3.1 Equation-based Congestion Control

Given a particular cluster-to-cluster data path, including information on its current round trip time, average packet size, and the rate of packet loss, how can we choose a sending rate that is both congestion responsive and fair to other Internet flows sharing portions of the same forwarding path?

It was mentioned in Section 2.2 that the *Transmission Control Protocol (TCP)* [Pos81] has emerged as the prevailing standard for determining a fair sending rate. This is partly because of its widespread deployment in today's Internet (e.g., Web applications [CMT98, SCJO01]) and partly because of its stability properties [CJ89]. A flow is said to be *TCP-compatible* or *TCP-friendly* if it sends at a rate that does not exceed the rate of a conformant TCP connection under the same network conditions. [FF99].

To achieve a conformant send rate, one could always replicate the window-based mechanisms of TCP in full detail. Recent work in equation-based congestion control [FF99, PFTK98, FHPW00], however, provides an alternative. Work in this area began with Floyd and Fall [FF99] who recognized the need for a TCP-friendliness test that could be used by routers to identify misbehaving flows. Toward this end, they develop an analytic expression that models TCP steady state behavior characterized by the familiar additive increase multiplicative decrease (AIMD) rate control policy. The equation is as follows:

$$T = \frac{1.5\sqrt{\frac{2}{3}}B}{R\sqrt{p}} \tag{2.1}$$

Here, T is the maximum TCP-friendly send rate (bytes/sec), B is packet size (bytes), R is round trip time (sec), and p is the packet loss rate. The model assumes that each data packet arrival generates an acknowledgment packet in return, and doesn't take into account retransmission timeouts or consecutive packet drops.

Padhye *et al.* expand on this approach in [PFTK98] by handling:

- Triple duplicate ACK loss indicators (four ACKs received with the same acknowledgment number),
- Retransmission timeout loss indicators, and
- Limitations in receiver window size.

The approach is based on TCP Reno (a widely deployed version of TCP) and models congestion avoidance behavior in the form of rounds where each round is the transmission of a window of data over the course of one round trip time. The analytic expression is as follows:

$$X = \frac{s}{R\sqrt{\frac{2bp}{3}} + t_{RTO}(3\sqrt{\frac{3bp}{8}})p(1+32p^2)} \tag{2.2}$$

Here, X is the TCP-conformant transmission rate (bytes/sec) to be calculated, s is the packet size (bytes), R is the round trip time (sec), p is the loss rate on the interval $[0, 1.0]$, t_{RTO} is the TCP retransmission timeout (sec), and b is the number of packets acknowledged by a single TCP acknowledgment.

Some additional assumptions in this model include *loss independence between rounds* and *correlated packet loss within a round*. The former means that a packet loss within a round of data transmission does not change the probability that a packet will be lost in a subsequent round. The latter assumption means that if a packet is lost, then all remaining packets transmitted until the end of the round are also lost. Clearly this is a somewhat crude approximation of the real world where correlations among packet loss events are a complex affair. [Pax97]

Equation 2.2 allows one to calculate a TCP-friendly send rate dynamically as conditions on the cluster-to-cluster data path change. Furthermore, using a rate-based approach allows us to encapsulate information about the bandwidth available to an application flow explicitly in the form of a numerical value. This value can be passed to application endpoints to enable both send rate modification at the transport level and various kinds of adaptation at the application level.

2.3.2 TCP-friendly Rate Control (TFRC)

In [FHPW00] and [HFPW03], Floyd *et al.* show how equation 2.2 can be used to construct a transport-level protocol that is less abrupt in its rate changes than TCP yet still TCP-compatible. Known as *TCP-Friendly Rate Control*, or *TFRC* [FHPW00, HFPW03], the protocol provides a more moderate response to transient changes in congestion. As such, it represents an attractive alternative for multimedia streaming in a best effort, unicast networking context.

TFRC avoids abrupt changes in sending rate in part by smoothing round trip time

values, an important input parameter to equation 2.2. To accomplish this purpose, an exponentially weighted moving average is used as follows:

$$R_{avg} = \beta R_{sample} + (1 - \beta) R_{avg} \tag{2.3}$$

[HFPW03] suggests a value of 0.1 for the sample weighting factor β. A more complex scheme for preventing oscillation behavior may optionally be used, but at the expense of a square root computation.

Smoothing is furthermore incorporated into the handling of loss rate, another crucial input parameter for equation 2.2. Rather than using a simple packet loss rate, TFRC uses a somewhat more sophisticated value called the *loss event rate* designed to better model TCP behavior in the face of multiple losses. Let d_0 be the sequence number of the first lost packet in a sequence of data packets transmitted by a flow. After a dampening interval of one round trip time, the next lost packet is noted and its sequence number becomes d_1. A *loss interval* s_0 is then defined as the difference between these two values or $d_1 - d_0$. Ignoring losses following an initial loss within the dampening interval is intended to model TCP which reduces its congestion window only once under such circumstances. This is because additional losses will be hidden by the duplicate ACK mechanism signaling a congestion event.

To apply smoothing, TFRC uses a *loss history* to calculate an *average loss interval*. This averaging is intended to reduce the impact of sudden, transitory values that are unrepresentative of prevailing values on the whole. A loss history includes simply the last n consecutive loss event values. (RFC 3448 [HFPW03] suggests a value of $n = 8$.) An average loss interval is then calculated as a weighted moving average using the formula:

$$s_{avg} = \frac{\sum_{i=1}^{n} w_i * s_i}{\sum_{i=1}^{n} w_i} \tag{2.4}$$

where

$$w_i = 1 \quad for \quad 1 \leq i \leq \frac{n}{2}$$
$$w_i = 1 - \frac{i - \frac{n}{2}}{\frac{n}{2} + 1} \quad for \quad \frac{n}{2} \leq i \leq n$$

An additional technique called *history discounting* is used to reduce the weighting of

older history values when the current loss event becomes larger than twice the current value of s_{avg}.

Note that both loss event and history discounting calculations require knowing the approximate arrival time of packets that were lost in transit. This can be done through interpolation using the equation below:

$$T_{lost} = T_{before} + ((T_{after} - T_{before}) * (S_{lost} - S_{before})/(S_{after} - S_{before})) \quad (2.5)$$

Here, S_{lost} is the sequence number of a lost packet, S_{before} is the sequence number of the last packet to arrive prior to the lost packet, S_{after} is the sequence number of the first packet to arrive after the lost packet, T_{lost} is the arrival time of the lost packet (to be calculated), T_{before} is the arrival time of the last packet to arrive prior to the lost packet, and T_{after} is the arrival time of the first packet to arrive after the lost packet.

TFRC is important to this dissertation because it provides a reasonably tested set of congestion control techniques that can be applied to the cluster-to-cluster problem scenario. Its smoothing features make it an appealing choice for complex multimedia applications that are sensitive to abrupt changes in bandwidth. Its rate-based approach, which encapsulates estimation results into a single value, lends itself well to information sharing among endpoints. Finally, its congestion response behavior is compatible with TCP flows sharing the cluster-to-cluster data path.

It should be noted, however, that the solution architecture presented in Chapter 3 and Chapter 4 does not *depend* on TFRC. As we will argue later, almost any equation-based congestion control scheme could be used with suitable result, provided that the scheme demonstrates the right set of properties (e.g., TCP-compatibility).

2.3.3 Rate Adaptation Protocol (RAP)

While bandwidth estimation in our implementation of CP makes use of equation 2.2 and TFRC's method for loss handling, work on aggregate congestion control in Chapter 4 also considers the *Rate Adaptation Protocol (RAP)* proposed by R. Rejaie *et al.* in [RHE99].

RAP is a transport protocol designed for real-time media streams over the Internet (e.g., digital video or audio playback applications). While it is designed to be TCP-compatible, its novel contribution lies in the way its rate-based approach separates

congestion control mechanisms from error control mechanisms. As the authors argue, this is because congestion control depends solely upon the network while error control is an application-level concern.

RAP achieves TCP-compatibility by employing TCP's familiar *additive increase, multiplicative decrease (AIMD)* algorithm. Rather than applying this scheme to a window of transmission data, however, it makes adjustments to an inter-packet gap, or IPG, defined as the current packet size divided by the current sending rate.

Additive increase contains both *step length* and *step height* dimensions. Step length refers to the frequency with which the increase is made. Since feedback delay in the system is one round trip time (RTT), RAP adopts the smoothed round trip time ($SRTT$) as its IPG update interval. (Less frequent updates may result in unresponsive behavior while more frequent updates may result in oscillations.) Step height refers to the increase in transmission rate at each update point. Translated into IPG terms, the update is defined as

$$IPG_{i+1} = \frac{IPG_i * C}{IPG_i + C} \tag{2.6}$$

where C has the dimension of time and controls the rate of transmission increase. To maintain the TCP invariant of exactly one additional packet transmission per step, a value of SRTT is chosen for C. (See [RHE99] for a more detailed discussion.)

Multiplicative decrease is handled using a β parameter that adjusts the current IPG as follows:

$$IPG_{i+1} = \frac{IPG_i}{\beta} \tag{2.7}$$

In keeping with TCP, β is defined to be the constant 0.5. Also like TCP, a multiplicative decrease update is applied immediately upon detecting network congestion.

Clustered losses in RAP are handled using a *cluster-loss-mode*. A lost packet with sequence number $Seq_{FirstLoss}$ will cause a back-off event. After that, subsequent losses with sequence numbers in the range $Seq_{LastLoss} \geq Seq > Seq_{FirstLoss}$, where $Seq_{LastLoss}$ is the last packet that has already been transmitted, will be ignored. This mechanism, as the authors point out, is similar to that of TCP-SACK [MMFR96, FMMP00, FF96].

While RAP's rate-based AIMD implementation provides a coarse-grained level of adaptation, additional fine-grained mechanisms are also proposed for increasing stability and responsiveness to transient congestion. This includes, among other things, a

feedback signal $Feedback_i$ defined as $Feedback_i = \frac{FRTT_i}{XRTT_i}$ where $FRTT_i$ and $XRTT_i$ are short- and long-term exponential moving averages of RTT. This *dimension-less, zero-mean signal* provides fine adjustments to the inter-packet gap using the update expression $IPG'_i = IPG_i * Feedback_i$.

The RAP module described in the authors' design architecture is charged with detecting packet loss and continuously modifying transmission rate at the sender. How the application layer responds to error events is left open to the application. For example, an application may adjust its encoding scheme, provide complete or partial error recovery using data retransmissions, or implement forward error correction with a chosen level of robustness. Regardless of the approach taken, data transmission cannot exceed that of the rate-governed RAP module.

RAP is important to this dissertation because it represents another TCP-conformant, rate-based scheme for congestion control. As such, it provides an interesting alternative to TFRC for implementing aggregate congestion control within the cluster-to-cluster application context. While TFRC uses a modeling equation to calculate an instantaneous TCP-friendly send rate given current network conditions, RAP calculates a sequence of relative rate changes in a way that closely mimics the behavior of TCP. Each of these approaches will be discussed further in Chapter 4.

2.4 Open Network Architectures

State sharing among flows in a cluster-to-cluster application could be handled in a variety of ways, including application-level control connections, network-level multicast, or even physical layer broadcast using a suitable technology like ethernet. One attractive option for a variety of reasons, however, is that of extensible router services that interact with application endpoints via pre-defined directives carried by application data packets. Such schemes are often referred to as *programmable networks* or *open network architectures*.

Open network architectures provide a superior alternative to control connections, multicast schemes, or broadcast channels in that they provide more than just a communication technology. Information in such architectures can be deposited into persistent storage, exchanged among flows traversing the same forwarding node, and modified using pre-defined programming directives. The result is a powerful and flexible way to solve complex communication problems [Wet99b, CMK+99, CGW02] by leveraging

resources within the network.

In this section, we first discuss the classic end-to-end argument for system design proposed by Saltzer, Reed, and Clark in [SRC84]. Then, we consider *active networking* [TW96, Wet99b] which is seen as *the* seminal work in programmable networks. Finally, we consider a more recent contribution to open network architectures called *ephemeral state processing (ESP)* [CGW02] which focuses more directly on the issue of state exchange among flows.

2.4.1 The End-to-End Argument

The *end-to-end argument* for system design was first proposed by Saltzer, Reed, and Clark in their classic 1984 paper "End-to-end Arguments In System Design" [SRC84]. Quoting from the paper, the argument is given as follows:

> *Functions placed at low levels of the system may be redundant or of little value when compared with the cost of providing them at that low level.*

To paraphrase for the networking context, designers should avoid placing functionality in the low-level forwarding infrastructure of the middle of the network when it can be implemented at the end systems.

The canonical example for this design principle is *transport reliability.* Building reliability mechanisms directly into the network at a low level (e.g., between each forwarding node) provides the service for all flows, but not without paying a price in forwarding performance.

The obvious problem, of course, is that some applications (e.g., certain kinds of media streaming) may not require reliable transport. Such applications end up paying a performance price for an unnecessary service when, in fact, network forwarding performance (reflected in end-to-end delay and throughput performance metrics) may be critical to the application.

An even more serious problem is that low-level reliability mechanisms in the network still fail to solve the problem of transport reliability. While the reliability of data exchange between hops in the network is important, transport reliability is fundamentally an end-to-end notion and must be handled at the endpoints of communication. Thus, mechanisms in the network that govern segments of the end-to-end forwarding path do little to prevent the need for end-to-end mechanisms that provide the same

functionality in a more comprehensive way. Some other examples of similar end-to-end functionality include packet sequencing, encryption, duplicate packet detection, and delivery acknowledgments.

While the end-to-end argument is persuasive when considering services like transport reliability, it is important to note that not all services are fundamentally end-to-end in nature. Bhattacharjee, Calvert, and Zegura argue in [BCZ98] that some network services are either possible or greatly enhanced by leveraging information available only within the network. Some examples are determining the location of network congestion, identifying global access patterns in the World Wide Web, and constructing multicast distribution tree topologies.

For this reason, the end-to-end argument as a design philosophy has been disputed in recent years, especially among researchers in the programmable networks community [Wet99b, CMK+99, CGW02]. Such researchers have proposed in-network solutions for such diverse problem areas as

- Topology discovery.
- Overlay, virtual, and peer-to-peer networks,
- Multicast feedback thinning,
- Pricing, accounting, and billing services,
- Proxies, middleboxes, and mediation devices, and
- Network security.

As we will see in Chapter 3, limited use of edge-routers as information exchange points by cluster-to-cluster application flows provides an elegant and deployable solution to the problem of flow coordination. It also represents only a minor re-interpretation of the end-to-end principle: rather than end hosts representing the domain boundaries of the end-to-end principle, cluster gateways or aggregation points (APs) do. The nodes between each AP remain unmodified in keeping with the end-to-end principle, thus avoiding redundant services and unnecessary forwarding overhead within the network core.

2.4.2 Active Networking

Active networking was first proposed in 1996 by Tennenhouse and Wetherall [TW96]. The main idea of the approach is to allow computation to occur within the network where previously only simple forwarding was provided. While the idea appears to fly in the face of the well-established end-to-end design argument, the authors argue that in-network computation is by no means new. Firewalls, Web proxies, multi-point communication, and certain types of mobile/nomadic computing have all done exactly that.

In Tennenhouse and Wetherall's original vision of active networking, end systems were given the ability to program the behavior of forwarding nodes as application data packets are received. This can be done in one of two ways. First, a packet, or *capsule*, might contain a header that references a particular program or operation on the router that should be applied to the application data contained within. Second, a packet might contain both program instructions and data to be operated on. A router would then execute the instructions using a standardized run-time environment deployed throughout the network.

Active network nodes have both a transient run-time environment that is reclaimed when packet has completed its execution, and a persistent environment that may be associated with the application flow. The latter might include *foundation components* that provide controlled access to router resources like routing tables and transmission links, *active storage* for data to be exchanged among packets or updated over time, and *extensible components* that can be built by an application flow and later referenced concisely using brief program instructions.

Since their original conception, a variety of active networking systems have built [Wet99b, A+97, D+98, vdM+98, YdS96, H+99, N+99, S+99]. They demonstrate the power and flexibility of the approach for creating new and customized network services. Such services often make direct use of network topology information, loss and load information, or the ability to make more efficient services that are widely dispersed across the network. Some examples include reliable multicast [LGT98], anycast [PMM93], explicit congestion notification (ECN) [RF99], and Web cache routing [Wet99a].

While the benefits of active networking are clear, the challenges posed by the approach are considerable. First is that of code mobility. For the approach to be enabled, a computation model, including both instruction set and available resources, must be standardized across all forwarding nodes. [TW96] Second, resources must be carefully

managed. This includes both properly restricting access to only those resources owned by the flow, and preventing a flow from consuming too many resources by injecting a large number of packets into the system. [Wet99b]

A third challenge is maintaining adequate forwarding performance. Capsule execution must be suitably bound and must not starve non-active network flows sharing the network. Finally, active nodes must be widely deployed for active services to be effective. Unfortunately, it is unlikely that this will be the case in the near future.

While active networking may be difficult to realize in its original vision, it represents a significant step in developing a framework for open network architectures. Subsequent work (including that presented here) builds upon many of these ideas but in ways that are less ambitious and far more deployable.

2.4.3 Ephemeral State Processing

[CGW02] presents a lightweight active networking scheme called *ephemeral state processing (ESP)* that allows IP packets to manipulate small amounts of state at routers using a set of pre-defined operations. State in this scheme is stored and retrieved using $(tag, value)$ pairs where tags are randomly chosen from a very large tag space. The size of the space minimizes the chance of collisions within the *ephemeral state store (ESS)* which is organized as an associative lookup table using a hash function.

ESP addresses the issue of resource management by using the *space-time product* as a natural bound for resource consumption. That is, conventional *soft state* is created by a user and then maintained as long as a user can continue to refresh it. The holding time of such state is thus unbounded. In contrast, state in ESP has a *bounded lifetime* of approximately 10 seconds before it is forcibly reclaimed. Thus, the resource requirements for a given flow (as well as all flows collectively) are naturally bounded by the space-time product as old state is reclaimed as fast as new state can be created.

ESP addresses the issue of performance in at least two ways. First, ESS access operations are designed to be simple and efficient. Memory access requires a simple hash which can be done in constant time and in hardware. Collisions are very unlikely given a tag size of 64 bits, and the entire ESS is naturally bounded in size by the space-time product. Second, the ESS instructions are likewise very simple and efficient. Each packet is allowed only one instruction, thus providing a bounded execution time. The authors envision an instruction set of not more than a few dozen instructions, with some examples being *COUNT* (counts the number of packets before reaching a threshold),

COMPARE (compares a new value with a value in the ESS), and *COLLECT* (applies an operation to a value in the ESS and then returns a value when a counter reaches zero).

While ESP is comparatively lightweight in its design compared to Tennenhouse and Wetherall's original active networking vision, it is nonetheless powerful in solving otherwise difficult problems. The authors demonstrate this by applying the approach to the problem of topology discovery, multicast feedback thinning, and data aggregation across participant flows.

The scheme we present in Chapter 3 shares much in common with the ESP approach. Both present open architectures and support the exchange of soft state between flows with arbitrary, application-defined semantics. Furthermore, both provide operations that allow state to be aggregated across flows in various ways.

Unlike ESP, however, the CP approach relies on enhanced forwarding services only at first- and last-hop routers (APs) and, in this way, is more deployable than ESP. While the ESP authors mention the possibility of incremental deployment and deployment only at network edges, they acknowledge that the approach is as effective as the proportion of nodes participating in it.

CP's approach is also distinct from that of ESP in the way it handles persistent state information. First, it treats state information as *soft state* that can be refreshed rather than *ephemeral state* that will be reclaimed. Resource management is not a problem in this approach because the state table itself is bounded in size. Second, while operations on ESS state are handled using packet-carried instructions, the CP approach relies solely on simple read and write operations. This is possible because many values held in the state table are the result of various pre-defined operations (computed on-demand for efficiency). Finally, CP extends ESP's approach by *providing* state information in addition to *storing/retrieving* deposited state. This information includes both shared network path state (obtained through probing mechanisms) and application flow state.

Overall, ESP presents a generalized infrastructure spread throughout the network for solving many types of problems that rely on topology information and the exchange of state between flows. While the approach is powerful, it still presents a considerable deployment challenge. In contrast, the CP approach is more tightly coupled with the cluster-to-cluster application architecture and more deployable in that cluster aggregation points are already under local administrative control and no additional support is

required from routers on the cluster-to-cluster data path between APs.

Chapter 3

Coordination Protocol

In this chapter, we describe our solution to the problem of flow coordination in cluster-to-cluster applications. Our focus here will be on *information sharing* mechanisms: how information can be stored and shared within the network, how application endpoints can read and write state information, how transport-level protocols can incorporate shared information into their operation, and how information sharing can be used to implement coordination among flows.

Issues and mechanisms related to aggregate congestion control, an important part of the Coordination Protocol, are deferred to Chapter 4 where they will be discussed in greater detail.

3.1 Overview

The Coordination Protocol (CP) architecture was designed with several 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,
- To provide an infrastructure for exchanging state among flows and allowing an application to implement its own flow coordination scheme, and
- To avoid the problems of centralized adaptation by relying on individual endpoints rather than scheduling or policing mechanisms at aggregation points.

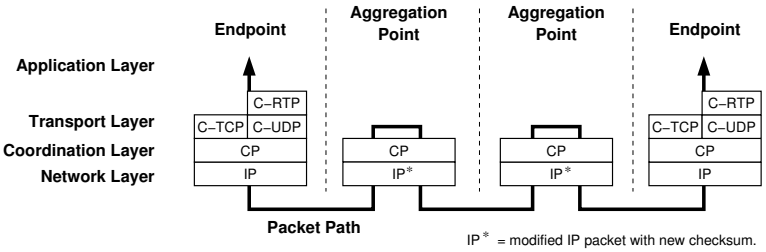


Figure 3.1: CP network architecture.

To realize these goals, CP makes use of a shim header inserted by application endpoints into each data packet. Ideally, this header is positioned between the network-layer header and the transport-layer header. The network stacks of each cluster endpoint and their associated AP are modified to process CP packet headers, while all other nodes along the cluster-to-cluster data path require no special modifications (i.e, CP is transparent to forwarding agents between each application cluster).

CP mechanisms are largely implemented at each aggregation point (AP) where there is *a natural convergence of flow data to the same forwarding host*. This may be the cluster's first hop router, or a forwarding agent in front of the first hop router. As mentioned in Section 1.4, an AP is part of each cluster's local computing environment and, as such, is under local administrative control.

Figure 3.2 summarizes CP operation by tracing a packet traversing the path between source and destination endpoints. The CP header is processed by the AP during packet forwarding. Essentially, the AP uses information in the CP header to maintain a per-application state table. Flows deposit information (e.g., their current priority) into the state table of their local AP as packets traverse the outbound path from an endpoint to the local AP, and then onward toward the remote cluster. Packets traversing the inbound path in the reverse direction pick up entries from the AP state table (e.g., the priority of peer flows, estimated bandwidth available) and report them to each endpoint.

In addition, the two APs conspire to measure characteristics of the cluster-to-cluster data path such as round trip time, packet loss rate, available bandwidth, etc. These measurements are made by exchanging probe information via the CP headers available from application packets traversing the data path in each direction. Measurements use all packets from all flows belonging to the same cluster-to-cluster application and thus monitor network conditions in a fine-grained manner. Resulting values are inserted into

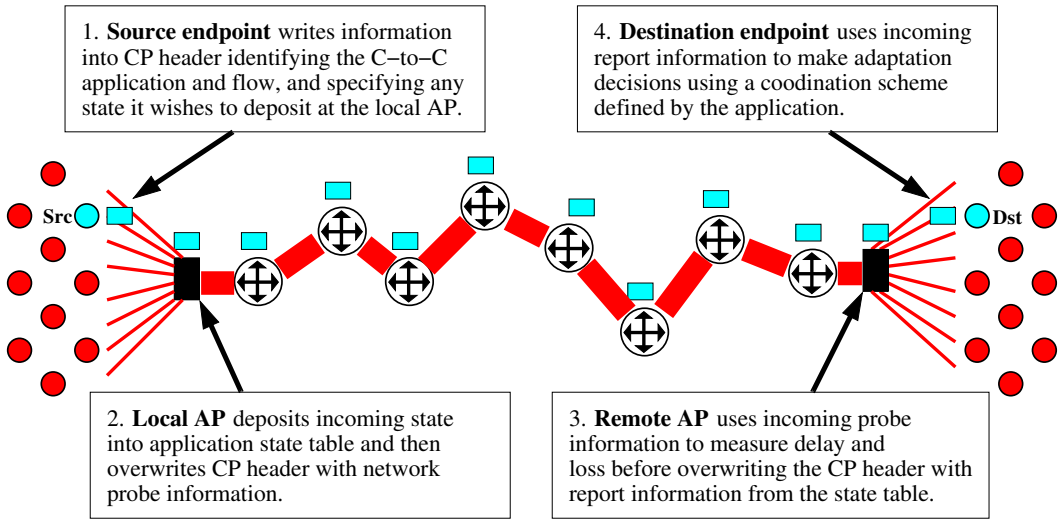


Figure 3.2: CP operation.

the state table.

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 cluster-to-cluster 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 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.

It is important to emphasize that CP is an *open architecture*. It's role is to provide information "hints" 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 already 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 and other adjustments that reflect the higher objectives of the application.

3.1.1 Why a new protocol layer?

The decision to create a new protocol layer between the network and transport layers rather than handling information sharing and flow coordination at the application layer requires some justification.

Consider, for example, a scheme in which simple socket connections are used between endpoints on the same cluster to allow information sharing among hosts and coordinated send rate adjustments. Such a scheme would have the advantage of requiring no fundamental modifications to participating network components, including network stacks and transport-level protocols on each endpoint and forwarding routines on the AP.

We argue that application-level schemes of this type, while perhaps easier to deploy, fall short from a variety of standpoints. First and foremost, they fail to exploit the problem topology effectively. Here we again note that a cluster's AP provides a natural point of convergence on the outgoing data path. Likewise, data on the incoming path must first pass this final forwarding hop before fanning out to application endpoints within the cluster. As such, a natural point for information exchange and dissemination exists without relying on extra connectivity between hosts.

Furthermore, piggybacking information sharing services on existing data flows effectively solves a host of problems associated with the endpoint-to-endpoint based approach, including

- Communication failures due to host failures or frequent join/leave events,
- Information inconsistencies due to propagation delay among endpoints,
- The need for peer directory services, and
- Group protocols for joining or leaving the cluster application.

Nor do application-level approaches solve the problem of providing centralized cluster-to-cluster network path measurement, bandwidth estimation, or aggregate statistics collection (application throughput, number of participating flows, etc). Such services can easily, however, be implemented at each AP due to its position within the application topology. This is because the AP handles the forwarding of all data packets from all flows in both directions, and thus is in an easy position to collect aggregate statistics, observe network performance, and estimate bandwidth available to the application.

Another objection might take the form of CP header placement. Even if we accept CP's solution architecture, couldn't headers be implemented at the application layer rather than between the network and transport layers?

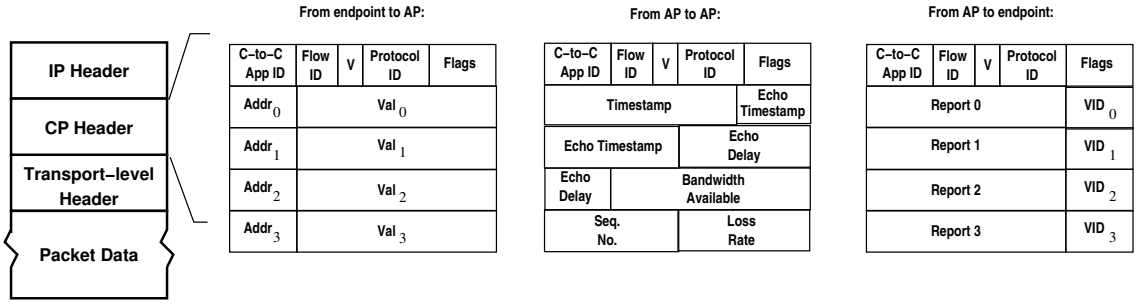


Figure 3.3: CP packet header format.

First we note that by placing CP below the transport-layer, we preserve the end-to-end semantics of individual transport-level protocols. This is a key requirement mentioned in Section 1.3 and Section 1.6. Second, we argue that CP logically belongs in this position. While the network layer handles next-hop forwarding on a packet-by-packet basis without regard to the notion of a flow, and the transport layer handles end-to-end semantics for individual flows, CP is concerned with the behavior of aggregate flows that may share a significant number of hops along the forwarding path but not the entire end-to-end path.

Nevertheless, we acknowledge that it is certainly possible to implement CP headers and mechanisms at the application layer (with some loss of efficiency). In fact, our implementation (described in Chapter 4) places the CP header in the first 20 bytes of UDP data. This implementation has the benefit of requiring no network stack changes on cluster endpoints. To make packet forwarding at the AP more efficient, software routers use kernel-based "deep processing" (header processing that goes below the IP header level) as an alternative to application-level handling. This approach demonstrates that implementation options exist, and that hybrid approaches are possible.

Finally, an important point to emphasize here is that the issue of header placement is merely one of implementation, and that the merits and disadvantages of a particular scheme should not obscure the core ideas proposed by CP. These ideas include, among other things, measuring network conditions across all application flows, maintaining shared state information at each AP, applying existing congestion control algorithms to the aggregate flow context, and providing the application with a framework for implementing sophisticated, inter-stream adaptation strategies.

3.1.2 CP Packet Headers

Figure 3.3 illustrates the CP header and its contents at different points on the forwarding path between source and destination endpoints. The header is exactly 20 bytes in length. Each of the three formats use the same prefix consisting of the following five fields:

Version (4 bits)
Coordination Protocol (CP) version number.

Cluster ID (5 bits)
Cluster-to-cluster application identifier.

Flow ID (7 bits)
Flow identifier.

Protocol (8 bits)
Transport protocol employed by this flow.

Flags (8 bits)
Flags directing AP to handle this packet in special ways.

A source endpoint will initially assign all of these fields when a data packet is created and a CP header inserted into the appropriate position. How an endpoint obtains a cluster ID and a flow ID may be handled a variety of ways. For example, a cluster ID may be assigned offline by network administrators and flow IDs by application designers. Alternatively, an ID server may be deployed on the local cluster.

The current version of CP allows up to 31 cluster-to-cluster applications per AP, and up to 128 flows per application. (See Section 3.2.) In addition, up to 255 different transport protocols may potentially be employed that use CP information in specialized, application-specific ways.

Flag fields currently include DATAPATH, LSCOPE, and REFLECT. DATAPATH is used to indicate that a packet is traversing the cluster-to-cluster portion of the end-to-end path. If set, a receiving AP will interpret header fields accordingly. LSCOPE refers to *locally scoped* and prompts the AP to process a packet received from a local endpoint and then drop it. This allows an endpoint to deposit or refresh state at the AP during periods when it has no data to send. REFLECT refers to a *reflected packet*. A

packet of this type, after being received by an AP from an endpoint on the local cluster, will be sent back to the source host along with report information. That is, it will never traverse the cluster-to-cluster data path. This can be useful when an endpoint requires state reports from its local AP, but no incoming data packets are available.

A key point to note is that *the content and role of the CP header change as the packet traverses the end-to-end path* between a source endpoint on one cluster and a destination endpoint on another. As shown in Figure 3.2, this includes the following path segments and their corresponding header contents:

Source endpoint to local AP. Header contents identify the packet and contain information to be deposited in the state table at the local AP.

Local AP to remote AP. Header contents include probe information being exchanged by APs. This information will be used to measure cluster-to-cluster path conditions and estimate available bandwidth.

Remote AP to destination endpoint. Header contents contain state information requested by the destination endpoint. This information will be used by the endpoint to make coordinated adaptation decisions.

Header contents are modified as a packet is forwarded by each AP. In addition to re-writing the header with a new format and constituent values, an IP checksum is recomputed. To be more precise, the current IP checksum is altered based on modifications to the five 32-bit words comprising the header. (Recomputing the checksum in its entirety turns out to be unnecessary.) More will be said about the IP checksum in Section 5.2.3.

Finally, the details of endpoint-to-AP operation fields are discussed in Section 3.2.1, AP-to-AP probe fields in Chapter 4, and AP-to-endpoint report fields in Section 3.2.2. CP header specifications are also provided in Appendix A.

3.2 AP State Tables

An AP creates a *state table* for each cluster-to-cluster 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 cluster-to-cluster application.

The organization of a state table is as follows:

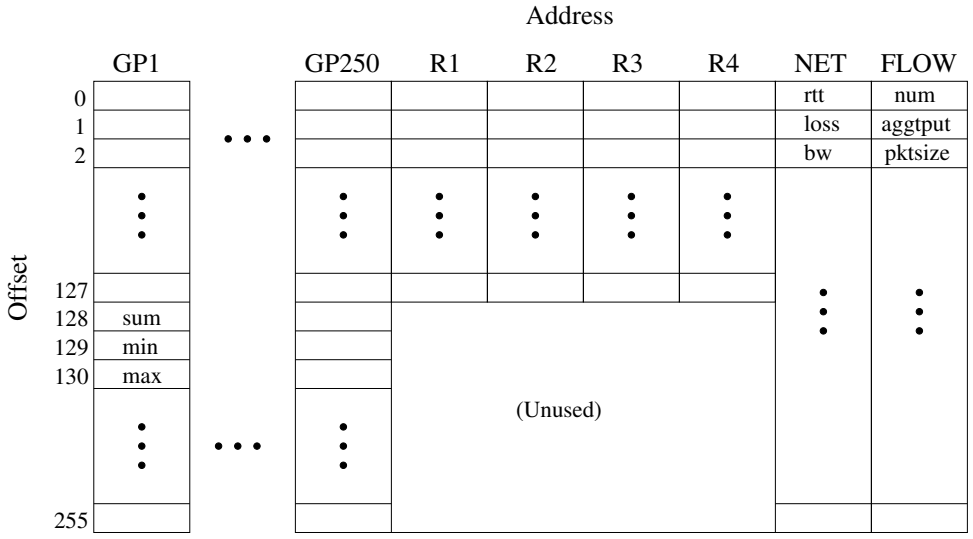


Figure 3.4: CP state table maintained at each AP.

- 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. (See Section 3.2.1 for details.) 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 cluster-to-cluster 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 general purpose addresses; network and flow statistic addresses are assigned by the AP and are read-only. The state table is illustrated in Figure 3.4.

3.2.1 Assigning Cells of the State Table

An endpoint can use the CP header of outgoing packets to assign the value of up to four cells in the state table per packet sent. When an *outbound* packet (i.e., a

packet leaving the local 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.
- Four "operation" fields which are used to assign 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 four operation fields in the header. Figure 3.3 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 exactly one of the first 128 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 writable (i.e., some of the addresses are read-only). Similarly, since a flow id is restricted to the range $[0, 127]$, in fact only 128 of the offsets associated with a particular writable address can be assigned. As mentioned, the address space is divided into four address types. The mapping between address range and type is illustrated in Figure 3.4. The semantics of a cell value at a particular offset depends on the specific address type and is described in the following subsections.

3.2.2 Report Pointers

Four of the writable addresses in the state table are known as *report pointers*. Values held in these address act as pointers to other locations in the state table. An application endpoint can select report information by assigning each of these report fields. The AP will then use the assigned values to select information to be passed back to the endpoint via the CP header as shown in Figure 3.3.

Specifically, each flow uses the mechanisms described above to write a 24-bit value into $R_j.fid$ where R_j is one of the four report pointers (i.e., $R1$, $R2$, $R3$, and $R4$ in Figure 3.4) and fid is the flow id. The value of these four cells 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 ($address$). The second 8 bits are interpreted as an offset ($offset$) 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 ($address.offset$) concatenated with the 8-bit validation token vid. This is done for each of the four report fields. In this way, any cell in the state table may be read.

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 endpoint to disambiguate between different reports. This is important because report pointer values can change over time as they are assigned by the endpoint. (See Section 3.2.6 for an illustration.) Reports following an assignment event would be ambiguous without being associated with a validation token chosen by the endpoint.

3.2.3 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 cluster-to-cluster data path as measured by APs across the aggregate of all flows in the cluster-to-cluster application. The most important offsets of this address include:

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

In fact, each of these values is associated with a whole family of related offsets, including the most recent sample value, mean and median values, a smoothed average, sample variance, minimum and maximum values, etc. Up to 256 network-related statistics are potentially available using offsets in the NET address.

$NET.bw$ 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 Chapter 4.

3.2.4 Flow Statistics

While statistics characterizing the cluster-to-cluster data path are available through the NET address, statistics characterizing application flows are provided by offsets of the flow statistics address $FLOW$. Offsets of this address are similarly read-only and include, among other things, information about:

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

Once again, each value is associated with a whole family of related offsets giving the most recent sample value, mean and median values, a smoothed average, sample variance, minimum and maximum values, etc. Up to 256 different statistics can be provided.

Each AP collects these statistics by measuring various features of application traffic as it is forwarded. Maintaining these statistics requires relatively simple per-packet accounting and periodic averaging.

3.2.5 General Purpose Addresses

The general purpose addresses (i.e., $GP1$ through $GP250$) in Figure 3.4 give a cluster-to-cluster application a set of tools for sharing information among flows in an application-defined way that facilitates coordination. 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 128 offsets of each

of general purpose address, the remaining 128 offsets are used to report aggregate functions of these first 128 flow 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.

Function offset values are computed using lazy evaluation for efficiency. (See Section 5.2.5 for further details.) Flow offsets are treated as soft state and time out if not refreshed.

3.2.6 An Illustration

Figure 3.5 illustrates state table operation as a cluster endpoint ($cid = 1$, $fid = 2$) sends and receives packets. Both outbound and inbound packets are received by the endpoint's local AP which performs read and write operations on the state table before forwarding the packet. CP packet headers are used for exchanging information between the endpoint and its local AP. The contents of these headers are shown to the left of the figure. To the right, state table contents are shown as the endpoint makes new assignments to various cells.

Packet A assigns to $GP1.fid$ the value 250, and to report addresses $R1.fid$ and $R2.fid$ the values $NET.rtt$ and $NET.bw$, respectively. The latter assignments act as pointers that will be used by the AP to assign CP reports when an inbound packet becomes available. This is shown in Figure 3.5 when packet B arrives and is assigned report values 17.2 and 872.

Note the use of validation tokens ("00" and "01") to identify report values. These tokens serve two purposes. First, they reduce the number of bits required to identify the state table cell associated with a report value. Second, they allow the endpoint

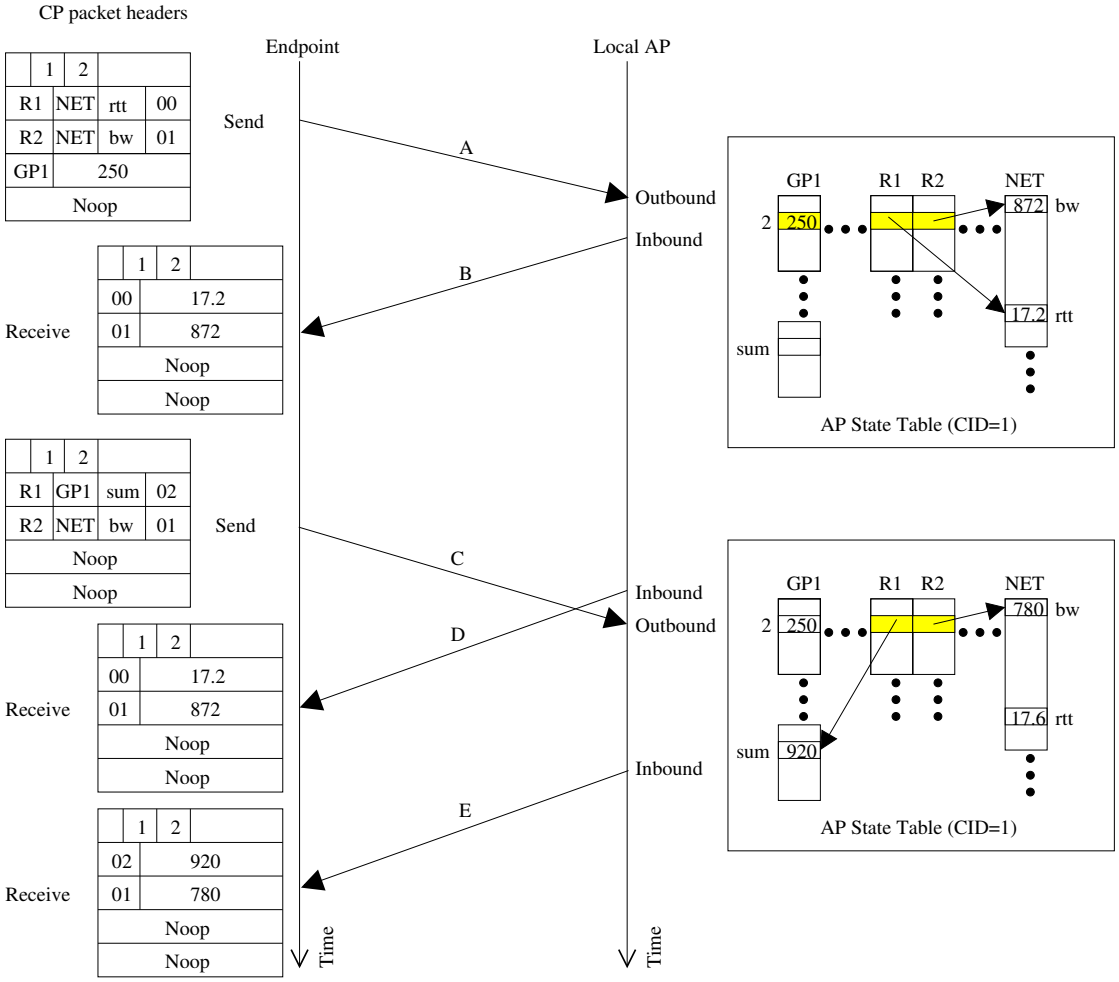


Figure 3.5: Illustrating state table operation. CP headers and AP state table contents as a cluster endpoint sends and receives packets.

to disambiguate between CP reports when new report assignments are issued. This is illustrated in Figure 3.5 when the endpoint sends packet C with a new set of report pointer assignments. While packet D is received by the endpoint after this transmission, the reports in the CP header follow the previous assignments made by packet A. (This is because packet D is forwarded by the AP *before* packet C has been received.) Not until packet E has been received do CP reports now reflect the new report address assignments. By observing validation tokens "00", "01", and "02", however, the endpoint is able to interpret reports without ambiguity throughout the transition period.

The use of validation tokens allow an endpoint to cycle reports through a large number of state table cells. This is important because the CP header allows only four reports per packet. By using the validation token space of up to 256 values and the mechanisms illustrated in Figure 3.5, an endpoint may change report content often without generating ambiguity in the values received.

3.3 Implementing Flow Coordination

While CP provides network and flow information, as well as facilities for exchanging information, it is up to the cluster-to-cluster application to exploit these services to achieve coordination among flows. The details of how an application goes about this may vary widely since much depends on the specifics of the problem an application is trying to solve. Most, however, will want to employ some type of *CP-enabled transport protocol* that can be configured to participate in one or more application-specific *coordination schemes*.

3.3.1 CP-enabled Transport Protocols

A CP-enabled transport protocol provides data transport services to an endpoint application using both information provided by CP and configuration provided by the application. Its behavior takes into account not only the data transport needs of the application endpoint it is serving, but the wider concerns of the cluster-to-cluster application as a whole. It is at the same time network-aware, peer flow-aware, and a participant in a larger context of cooperative resource sharing and information exchange.

Figure 3.6 shows a generic CP-enabled transport-level protocol in schematic form.

In essence, it is charged with managing at least three functions:

- Send rate adaptation.
- State sharing with peer flows.
- Disseminating information to the application layer.

Send rate adaptation involves using CP information and configuration parameters to maintain a flow send rate that is both responsive to network congestion and reflective of application priorities. The endpoint, for example, might be configured to send at CP's suggested available bandwidth rate unless the number of peer flows reach a particular threshold. Or, it may send at a constant rate which is announced to peer flows in the same application using the state table mechanisms described above.

CP-enabled transport protocols may furthermore participate in coordination schemes that require extensive state sharing with peer flows. For example, general purpose addresses in the AP state table may be used to implement floor control or dynamic priority assignments. The transport protocol may be configured to participate in such algorithms by reading and writing particular table items and modifying its send rate when certain conditions are met.

Another function is responding to application-layer requests for CP information. Like the *Congestion Manager* [BRS99] described in Chapter 2, an important goal in CP design is to put the application in control over the data transmission process. This means sharing with the application layer information about path conditions and peer flows, and allowing the application to make decisions about what data should be sent given the resources available.

Furthermore, the application layer on a given endpoint may wish to coordinate with other application endpoints on matters not directly related to data transport. For example, changes in a camera's physical position may be significant to all the endpoints within the same cluster in a video capture application. Using a special API, the application may instruct the transport layer to read and write particular cells in the AP state table in order to share information on the change with peer endpoints.

Whether a CP-enabled transport-level protocol is implemented as an application-level library or an operating system service depends on the implementation details of the CP header. In Section 3.1.1, we noted that the CP header logically fits between the

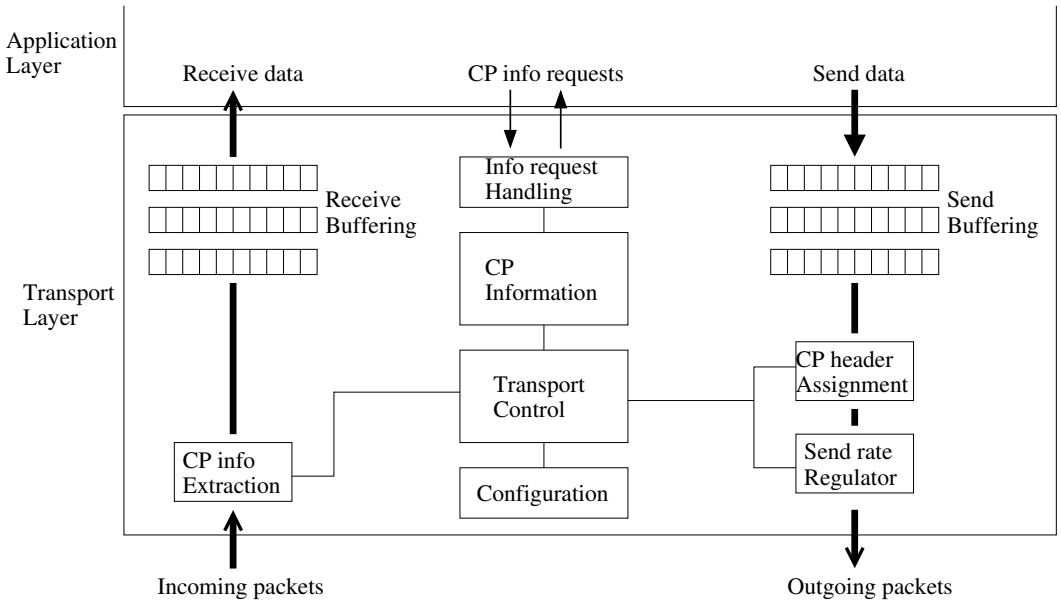


Figure 3.6: CP-enabled transport-level protocol schematic.

network and transport layers, but that an application-level implementation is likewise possible.

A transport-level service API could provide a fairly seamless substitute for the current TCP/IP socket interface, providing additional options for assigning cluster and flow id values. Or, it could be designed to pass various types of state table information directly to the application, for example, to help regulate media encoding adaptation. Still other transport-level protocols might simply provide a thin layer of mediation for an application to both read and write values from a local AP's state table; for example, using the information to coordinate media capture events across endpoints.

The principal function of a CP-enabled transport-level protocol is to use both CP information and application configuration to regulate a flow's sending rate as network conditions change on the shared cluster-to-cluster data path. How this configuration is accomplished and the degree of transparency to the application are both left to the protocol designer. In general, we believe that the expanded operational and informational context of transport-level protocols in the CP problem domain represent a rich frontier for future research.

3.3.2 Coordination Schemes

A flow *coordination scheme* is used by cluster-to-cluster application designers to define global application objectives and specify individual flow behavior in realizing those objectives. While CP provides the raw tools needed for coordination (network information and state sharing mechanisms), it is up to the application to define its objectives and use these services to achieve coordination in the manner desired. In this sense, CP is an *open architecture* for creating flow coordination solutions and not a solution in and of itself.

The focus in this dissertation is on coordination schemes that *apportion bandwidth among flows*. To illustrate, we provide two brief examples below.

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 $GP1.fid(A)$, the latter of 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 examples are "miniature" in that realistic cluster-to-cluster applications are likely to have many more flows and networking requirements that are more complex and change dynamically. Nonetheless, they serve to illustrate how CP information and state sharing can be used to coordinate bandwidth usage among flows in a manner that requires no centralized control.

An important point to note here is that aggregate bandwidth available to the application as a whole (equal to CP's bandwidth 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 ($n * NET.bw$) is not exceeded. We believe this to be a powerful feature of our protocol architecture with the potential to dramatically enhance overall application performance in a wide variety of circumstances.

In addition to bandwidth distribution, an application may use CP mechanisms to perform one or more types of *context-specific coordination*. That is, an application may use CP state exchange mechanisms to achieve coordination for any arbitrary problem. Some examples include leader election, fault detection and repair, media capture synchronization, coordinated streaming of multiple data types, distributed floor control, dynamic priority assignment, and various types of group consensus.

3.4 Summary

In this chapter, we have described the *Coordination Protocol (CP)*, our solution to the problem of flow coordination in cluster-to-cluster applications. Our discussion has focused on AP state table design and its use by cluster endpoints, as well as CP header mechanisms that carry information to and from application endpoints. In Chapter 4, we discuss CP mechanisms related to aggregate congestion control. This includes available bandwidth estimation and a method for scaling estimation results to multiple flowshares.

Chapter 4

Aggregate Congestion Control

An important issue for cluster-to-cluster 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. Failure to exhibit such responsiveness, as described in Section 2.2, will result in unfairness to other Internet flows sharing the same bottleneck link and, even worse, the potential for congestion collapse. [FF99] Within the application itself, a lack of responsiveness by some flows may affect the performance of other flows, as well as the application's ability to maintain coordinated control of available network resources. Without this control, application performance on an aggregate level will suffer.

In this section, we describe CP mechanisms for achieving aggregate congestion control. Our scheme provides the following benefits:

- Almost any rate-based, single-flow congestion control algorithm may be applied to make aggregate cluster-to-cluster traffic congestion responsive.
- Cluster-to-cluster applications may receive multiple flowshares and still exhibit correct aggregate congestion responsiveness.
- Individual flow behavior is decoupled from aggregate congestion response behavior, thus freeing the application to realize aggregate responsiveness using any self-designed bandwidth allocation scheme.

Define a *bandwidth flowshare*, or simply *flowshare*, for transport-level protocol P to be the bandwidth used by a conformant P-flow under comparable network conditions.

For example, a TCP flowshare is simply the bandwidth taken by a TCP-conformant flow with unlimited data to send under a particular set of network path conditions (round trip time, bottleneck bandwidth, packet loss rate, number of competing flows, etc.).

While a flowshare may be defined using any number of transport-level protocols, the widespread deployment of TCP implies a *de facto* standard that will be observed in this work. As will be seen, however, the use of TCP as a standard for defining flowshares does not imply that TCP must be employed as a transport-level protocol. Nor does it imply that each individual flow within a cluster-to-cluster application must employ the same TCP-equivalent protocol. The requirement, instead, is to realize TCP flowshare *equivalence* using any assortment of heterogeneous transport-level protocols and a bandwidth coordination mechanism (i.e., CP).

In this dissertation, we assert that *a cluster-to-cluster application employing m flows should receive the equivalent of m flowshares*. To see why, consider a cluster-to-cluster application with m flows sharing a bottleneck link with n additional flows. Let the total bottleneck bandwidth available be B. If each of m application flows uses a single flowshare, then any given flow will receive a bandwidth share of $\frac{B}{m+n}$. If, however, m flows share a single flowshare, then any given application flow will receive a bandwidth share of only $\frac{B}{m(n+1)}$. As m or n grow, this difference becomes significant.

Less formally, an application employing m independent flows will naturally receive m flowshares using current transport protocols like TCP. In order to avoid penalizing a complex cluster-to-cluster application for using CP, we must provide an equivalent number of flowshares to emulate (in *aggregate*) the bandwidth realized by an independent-flow approach. For this reason, we believe each of the single flowshare approaches to congestion control described in Chapter 2 ([BRS99], [PCN00], [KW99]) to be unduly restrictive for a multi-flow cluster-to-cluster application.

The organization of this chapter is as follows. First, we first discuss CP mechanisms for measuring path conditions and estimating the bandwidth available for a single application flowshare. Included in this discussion are both *TCP-Friendly Rate Control (TFRC)* [FHPW00, HFPW03] and *Rate Adaptation Protocol (RAP)* [RHE99], two representative rate-based schemes that provide TCP-friendly bandwidth estimation. We then describe simulation results that demonstrate the success of our approach when aggregate application traffic conforms to a single flowshare. Next, we discuss the problem of scaling aggregate application traffic to multiple flowshares. *Bandwidth filtered*

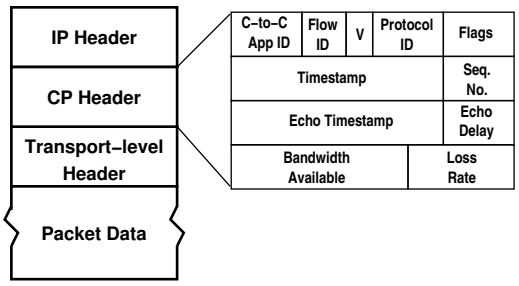


Figure 4.1: CP header contents as packet is forwarded between APs.

loss detection (BFLD) is presented as a technique for making single-flow bandwidth estimation algorithms work when aggregate traffic uses multiple flowshares. We then demonstrate the effectiveness of our approach by presenting TFRC- and RAP-based simulation results, followed by a more extensive set of TFRC-based implementation results taken from our laboratory testbed. (See Appendix B.) Our results demonstrate the viability of our approach for a wide range of network conditions.

4.1 Measuring Network Conditions

As described in Section 3, all packets from all flows in a cluster-to-cluster application are used by CP to measure network conditions on the shared data path between APs. As each outbound packet is sent by an endpoint to its local AP, it will have network probe information written into its CP header. The packet will then be forwarded over the cluster-to-cluster data path toward the remote cluster. When the packet is received by the remote AP, its probe information will be processed before forwarding the packet to its destination endpoint. Application packets on the reverse path allow probe information to be exchanged in both directions. In general, the number of packets available for exchanging probe information is large since cluster-to-cluster applications typically employ many flows and are high bandwidth in character. As a result, network probing on the cluster-to-cluster data path can take place on a very fine-grained level.

Figure 4.1 shows the CP header contents for a packet that this being forwarded between APs. *Timestamp*, *echo timestamp*, and *echo delay* fields are used for measuring round trip time and are explained in Section 4.1.1. *Sequence number* and *loss rate* fields are used for measuring packet loss and are explained in Section 4.1.2. The *bandwidth available* field is used to pass the results of bandwidth estimation between APs and is covered in Section 4.2. To review information on the first four bytes of the header

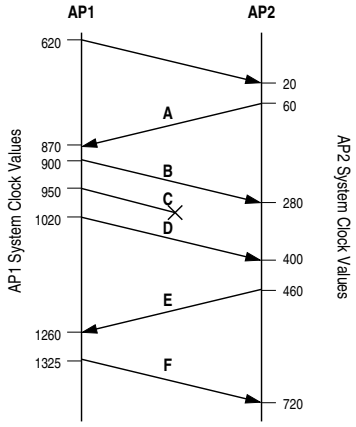


Figure 4.2: Timeline of AP packet exchanges.

Packet	Sequence Number	Time-stamp	Time-stamp Echo	Echo Delay
B	14	900	60	30
C	15	950	60	80
D	16	1020	60	150
F	17	1325	460	65
A	76	60	620	40
E	77	460	1020	60

Figure 4.3: CP header contents for various packets in Figure 4.2.

(*C-to-C application id*, *flow id*, etc.), refer to Section 3.1.2 of the previous chapter.

4.1.1 Network Delay

To measure RTT, the APs use a timestamp-based mechanism. An AP inserts a 24-bit timestamp into the *timestamp* field of the CP header (refer to Figure 3.3) of each packet on the forwarding path. (See Appendix A for details on how this timestamp is formatted.) This timestamp represents the time at which the packet begins its traversal of the cluster-to-cluster data path.

The remote AP will then echo the timestamp of the last packet received from the remote AP by placing the value in the 24-bit *timestamp echo* field of the CP header of the next packet traveling on the reverse path back to the sending AP. Along with this timestamp, a delay value will be placed in the 24-bit *echo delay* field of the CP header indicating the length of time between the arrival of the original packet timestamp and the time when a packet became available in the reverse direction for carrying the timestamp echo.

By noting the time when a timestamp echo packet is received ($T_{arrival}$), the AP can calculate the round trip time as $(T_{arrival} - T_{echo}) - T_{delay}$. In our example of Figure 4.2 and Figure 4.3, AP2 receives packet B at time 280. The CP header contains the timestamp echo 60 and an echo delay value of 30. Thus, the round trip time is calculated as $280 - 60 - 30 = 190$. This value, along with a smoothed average of recent samples and a sample variance, are placed in the AP's state table at various offsets in

the *NET* address.

Note that there is no one-to-one correspondence between timestamps sent and timestamps echoed between APs. It may be the case that more than one packet is received by a remote AP before a packet traversing the reverse path is available to echo the most current timestamp. The AP simply makes use of available packets in a best effort manner. In Figure 4.2 this can be seen as AP2 receives both packets B and D before packet E is available to send on the return path. Likewise, an AP may echo the same timestamp more than once if no new CP packet arrives with a new timestamp. In our example, this occurs when AP1 sends packets B, C, and D with a timestamp echo value of 60, which it received from packet A, and increasing values of echo delay.

4.1.2 Packet Loss

To detect loss, the APs employ a sequence number mechanism. Each AP inserts a monotonically increasing number into the 16-bit *sequence number* field of the CP header. It is important to note that this sequence number bears no relationship to additional sequence numbers appearing in the end-to-end transport-level protocol header nested within the packet. As with all CP probe mechanisms, the underlying transport-level protocol remains unaffected as CP operates in a manner that is transparent to end-to-end concerns.

At the receiving AP, losses are detected by observing gaps in the sequence number space. In our example of Figure 4.2 and Figure 4.3, AP2 detects the loss of packet C when the sequence number received skips from 14 (packet A) to 16 (packet D). To increase robustness in the face of packet re-ordering, CP follows the TCP practice of recognizing a packet as "lost" only after three subsequent packets have arrived. In other words, only after three subsequent packets fail to "fill in" the sequence number gap detected does CP conclude that a packet loss (or losses) has occurred.

As with RTT, the current loss sample, along with a smoothed average of recent samples and a sample variance, are placed in the AP's state table at various offsets in the *NET* address. In addition, a simple packet loss rate is reported back to the remote AP using the 16-bit *loss fraction* field in the CP header. This is necessary because losses are detected at the *receiving* cluster's AP, after application packets have already traversed the cluster-to-cluster data path. Measured values, however, need to inform endpoint *senders* who may wish to make encoding, compression, forward error correction, or rate adjustments directly based on packet loss statistics.

4.2 Estimating Available Bandwidth

An important function of each AP is estimating the congestion-responsive level of bandwidth available to the application given current conditions on the cluster-to-cluster data path. To accomplish this task, we leverage current work on single-flow, rate-based congestion control algorithms. In particular, we look at both *TCP-Friendly Rate Control (TFRC)* [FHPW00, HFPW03] and *Rate Adaptation Protocol (RAP)* [RHE99] which we have implemented using simulation. Later, we will also present results for a TFRC-based implementation using FreeBSD software routers and a laboratory testbed.

We emphasize, however, that the CP architecture and its related mechanisms are not tied to any particular bandwidth estimation algorithm. In fact, almost *any* single-flow, rate-based congestion control algorithm could be adopted for use. This is because nearly all such algorithms make use of information readily available at each AP: round trip time and round trip time variance, average packet size, packet loss events, packet loss rate, current clock time, etc. As work in equation-based congestion control [FHPW00, PFTK98] continues, for example, modifications to Equation 4.1 can readily be applied to CP without mandating changes in CP header format or basic probe exchange mechanisms.

An important clarification point to note is that the estimated bandwidth available, calculated by each AP, is for a *single flowshare*. This value (calculated as X in Equation 4.1), is stored in an offset of the network statistics address ($NET.bw$) at each AP. To scale this value to multiple flowshares and obtain the bandwidth available to the entire application as a whole, an endpoint may simply multiply the value by the number of active flows within the application. As described in Section 3.2.4, this value is stored in an offset of the flow statistics address ($FLOW.num$).

So, while an m-flow application may use up to m flowshares, the estimated bandwidth availability is maintained for a single flowshare. This is done for several reasons. First, the value is easier to represent than its scaled counterpart in the 24-bit field within each AP's state table. Storing the scaled version would mean a loss of precision as a much larger number still must be represented within the same 24-bit field. Second, storing the scaled version would require all endpoints to obtain both $NET.bw$ and $FLOW.num$ values from AP state tables in almost all circumstances. This is because $NET.bw$ cannot properly be interpreted without understanding the number of flows that have scaled the value. In contrast, many applications may wish to simply have a

subset of their flows send at a single flowshare. For these flows, it is sufficient to obtain only the *NET.bw* value and nothing else. Hence, maintaining a single flowshare value in *NET.bw* provides a useful component value that can either be used by an endpoint directly, or scaled at the endpoint host using additional information.

Finally, we emphasize that the role of the congestion control algorithm at each AP is merely to estimate the congestion-responsive level of bandwidth available to a single flow in the cluster-to-cluster application. This value is placed in the state table and passed back to individual endpoints on demand using the CP header and related mechanisms described in Section 3.2. The algorithm is *not* used by the AP to perform traffic shaping, policing, or packet scheduling in any manner whatsoever. Instead, CP's approach is to inform application endpoints of bandwidth available to them and allow them to make their own coordinated rate adjustments. Justification for this approach is provided in Section 2.1 where we discuss the shortcomings of transparent, in-network traffic shaping schemes.

4.2.1 TCP-Friendly Rate Control (TFRC)

TCP-Friendly Rate Control [HFPW03], or simply TFRC, is based upon fairly recent work on *equation-based congestion control* [FHPW00, PFTK98]. In this approach, an analytic model for TCP Reno congestion avoidance behavior (in steady state) is derived that can be used to estimate an instantaneous TCP-friendly sending rate given various channel properties like round trip time, loss rate, and average packet size. In particular, the analytic expression used is given as follows:

$$X = \frac{s}{R\sqrt{\frac{2bp}{3}} + t_{RTO}(3\sqrt{\frac{3bp}{8}})p(1+32p^2)} \tag{4.1}$$

Here, X is the TCP-conformant transmission rate (bytes/sec) to be calculated, s is the packet size (bytes), R is the round trip time (sec), p is the loss rate on the interval $[0, 1.0]$, t_{RTO} is the TCP retransmission timeout (sec), and b is the number of packets acknowledged by a single TCP acknowledgment. Since Section 2.3.1 already describes background related to Equation 4.1 in some detail, we will not repeat such exposition here. Instead, we consider how this approach is actually applied in the CP context. To do this, we consider how an AP handles each input value into the equation, and then the calculation itself.

First we note that b, the number of packets acknowledged by a single acknowledgment, is simply a constant value. [HFPW03] recommends that this value be set to 1 on the grounds that many TCP Reno implementations do not use delayed acknowledgments. The value could be set to 2 in the future if, empirically, the situation were to change.

R, the round trip time (sec), is already obtained using the mechanisms described in Section 4.1.1. To implement the smoothing described in Section 2.3.2, per-packet round trip time updates are passed into a TFRC handler that performs the weighted averaging given in Equation 2.3 with relatively few calculations. Furthermore, t_{RTO}, the TCP retransmission timeout (sec), can be approximated using $t_{RTO} = 4 * R$. The authors in [HFPW03] refer to [Wid00] in support of the claim that more precise calculations for t_{RTO} provide relatively little increase in fairness with existing TCP implementations.

The value s, packet size (bytes), is already obtained by each AP as part of the flow ($FLOW$) statistics described in Section 3.2.4. Like many other statistics in the AP state table, the most recent measured value, along with a smoothed average of recent samples and a sample variance, are all kept at various offsets in the $FLOW$ address. Here, a smoothed average is an appropriate value to adopt for input into Equation 4.1.

This leaves p, the loss rate on the interval $[0, 1.0]$. As described in Section 2.3.2, loss rate handling in TFRC is somewhat intricate and represents by far the most complex input component into Equation 4.1. Rather than use a simple packet loss rate, a somewhat more sophisticated value called the *loss event rate* is employed that better models TCP behavior in the face of multiple losses. Essentially, round trip time is used to dampen the effect of subsequent losses once an initial loss has been encountered. This is intended to model TCP's round trip time delay before acknowledgments can establish the missing segment. Weighted averaging of a loss event history, as well as history discounting, are then used to further smooth resulting statistics. (See Section 2.3.2 for additional details.)

The important thing to note here is that CP *already provides the required raw information to handle loss event rate calculations.* That is, the mechanisms described in Section 4.1.2 already identify lost packets and provide a round trip time estimate. Calculating TFRC's *loss event rate*, then, requires merely an additional lost packet handler that calculates loss event intervals, maintains a history of such intervals and a weighted average, and applies history discounting.

Calculation of X, the TCP-conformant transmission rate, is done on a per packet

basis using Equation 4.1. For efficiency, a lookup table can be employed to approximate square root values in the denominator. Fixed point techniques can furthermore be used to reduce the number of cycles required for multiplication and division operations. Chapter 5 describes these techniques in some detail.

Available bandwidth, like packet loss rate, is calculated by the receiving AP and reported back to the sending AP using the 24-bit *available bandwidth* field in the CP header. For example, in Figure 1.5, 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 traversing the reverse path. In the same manner, Cluster A maintains an estimate of available bandwidth from Cluster B to Cluster A.

In general, it is advantageous to do bandwidth estimation at the receiving AP because this is where loss packet events are handled. In the case of TFRC, it would be difficult to pass loss event size and history information back to the sending AP using the CP header. It would be even more difficult to implement history discounting at the sending AP. In contrast, it is relatively easy to handle such complexity at the receiving AP and pass a single result value (i.e., available bandwidth) back to the sending AP using the CP header.

4.2.2 Rate Adaptation Protocol (RAP)

While TFRC uses a modeling equation to calculate an instantaneous TCP-friendly send rate given current path conditions, RAP [RHE99] calculates a sequence of relative rate changes in a way that closely mimics the behavior of TCP congestion control. In general, adapting RAP to the CP context is a good deal easier than that of TFRC. This is because there are fewer input parameters, and treatment of lost packets is much simpler.

Two types of rate changes must be handled in RAP: *additive increase* and *multiplicative decrease*. The former, as described in Section 2.3.3, uses a *step height* parameter α defined as $\frac{Packetsize}{RTT}$. As described in Chapter 3, both average packet size ($Packetsize$) and round trip time (RTT) are maintained in the AP state table as $FLOW.pktsize$ and $NET.rtt$ respectively. Updating α, then, is a relatively simple affair requiring a short per-packet calculation that can be done using the fixed point methods described in Chapter 5. Multiplicative decrease is handled using a β rate parameter defined to be simply 0.5. As a constant, the value requires no updating whatsoever.

Step length, or the time between two additive increases is set to be one $SRTT$, or smoothed round trip time, a value defined by Jacobson in [Jac88]. As mentioned in Section 4.1.1, a smoothed round trip time value is already kept in the state table at each AP. If, however, the smoothing factor requires a different sample weight or specialized handling for whatever reason, then it can easily be implemented with a handling routine that takes as input new round trip time samples obtained using the mechanisms described in Section 4.1.1.

To update the current available bandwidth estimate, each AP maintains a timer that expires after the current step length. If no losses occur before the timer expires, then the current rate is updated using the α parameter. If, on the other hand, a packet loss is detected, then the β parameter is immediately applied to update the current sending rate.

Cluster-loss-mode (see Section 2.3.3) can be implemented simply by applying backoff parameter β only once for an entire cluster of contiguous lost packets rather than once for each lost packet individually. Feedback signal $Feedback_i$, used to increase stability and responsiveness to transient congestion, can be used to adjust the current rate in a fine-grained manner. This value, defined as $Feedback_i = \frac{FRTT_i}{XRTT_i}$, can easily be calculated by maintaining a short-term exponential moving average $FRTT_i$ and long-term exponential moving average $XRTT_i$. The value can be computed with each packet arrival or calculated periodically (e.g., every one millisecond) and applied to the current rate accordingly.

What makes RAP a particularly interesting candidate for a TCP-friendly bandwidth estimator in CP is both its rate-based approach, and its separation of congestion control from error control. That is, while rate adjustments and loss detection are fundamental to the algorithm, packet retransmissions and error recovery are an orthogonal concern. (This is in stark contrast to TCP which incorporates data retransmissions directly into its congestion detection and response mechanisms.) The separation of concerns works well for CP which requires congestion control functionality alone. Error control, when it is needed, may be handled on an individual flow basis using an appropriately selected transport-level protocol.

4.3 Single Flowshare Evaluation

In this section, we evaluate the effectiveness of CP congestion control mechanisms for cluster-to-cluster applications configured to send at an aggregate rate equivalent to a single flowshare.

Our evaluation makes use of the *ns2* [BEF+00] network simulator. *ns2* is a widely used discrete event simulator that originated in 1989 as a variant of the REAL network simulator [KMJ+]. Over the past 15 years, it has received substantial support from DARPA, and code contributions from LBL, Xerox PARC, UCB, USC/ISI, ACIRI, CMU, and Sun Microsystems. Additional contributions have been made by countless researchers (and graduate students) in the network community as *ns2* has been extended repeatedly to support new work in diverse areas of the field. At the date of this writing, it is one of the most comprehensive (if not unwieldy) simulators of its type.

It is worth noting that *ns2* was chosen to evaluate work on CP aggregate congestion control for at least two reasons. First, it provided a manageable environment for working with several different congestion control algorithms. In particular, both TFRC and RAP were implemented for study as TCP-friendly bandwidth estimators at each AP. By studying more than one algorithm, we gained valuable perspective on CP mechanisms in their generality.

Second, *ns2* provided implementations of both TFRC and RAP single-flow transport protocols. This was important in validating our implementation. The point to be noted here is an important separation of concerns. While TFRC and RAP both claim to be TCP-compatible, how well each competes with peer TCP flows, while important, is not our first concern. Our first concern is how well a TFRC- and RAP-based CP implementation competes with peer TFRC and RAP flows. *ns2* allows us to make such comparisons by providing standard implementations of TFRC and RAP, written by the algorithm authors themselves. Such implementations, to our knowledge, were not available to the FreeBSD or LINUX world at the time this work was undertaken.

Later, Section 4.5 will present results from a TFRC-based implementation run in our laboratory testbed at UNC. Among other things, these results serve to demonstrate the effectiveness of our approach in the context of competing TCP flows.

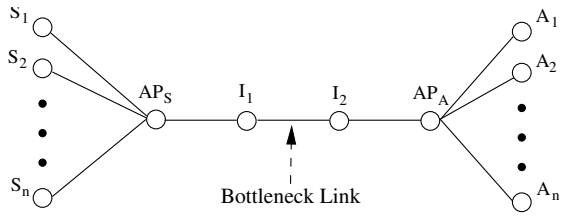


Figure 4.4: Simulation testbed in *ns2*.

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	DropTail
Simulation duration	180 sec

Figure 4.5: Configuration parameters.

4.3.1 Configuration

We refer to our *ns2* [BEF+00] implementations of TFRC and RAP congestion control algorithms in CP as *CP-TFRC* and *CP-RAP*, respectively. Each uses the same CP mechanisms for measuring network path conditions and exchanging state between application endpoints and AP. The only difference is in the algorithm used to estimate bandwidth availability. With either method, the value is placed in each AP state table at the location $NET.bw$. Application endpoints can then request the value using the mechanisms described in Chapter 3.

Figure 4.4 shows our 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 set of sending and ACK agents is partitioned into two subsets, one for CP agents and the other for competing TFRC or RAP agents.

In general, links in this topology are configured to provide 2 ms delay, have a bandwidth of 200 Mb/s, and use DropTail queuing with a queue length of 200. There are two exceptions. First, links between ACK agents A_1 through A_n and AP_A are assigned delay values that vary but do not exceed 2.0 ms. This allows some variation in RTT for different end-to-end flows. Second, the bottleneck link is parameterized with respect to link speed, delay, queue type, and queue length. Table 4.5 summarizes these and other parameters.

CP flows in our simulated cluster-to-cluster application are configured to take an equal fraction of the current bandwidth available to the application. That is, if m cluster-to-cluster endpoints share a flowshare bandwidth estimate of B, then each endpoint sends at a rate of B/m. More complex configurations are possible, and the reader is referred to [OMP02], [OSMP04], and [OMP04] for further illustrations.

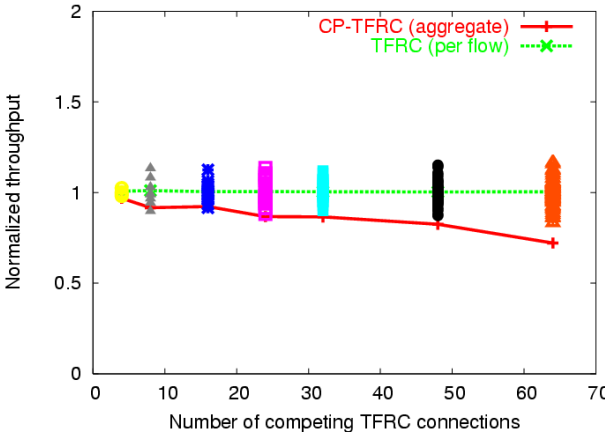


Figure 4.6: CP-TFRC: Number of competing TFRC flows.

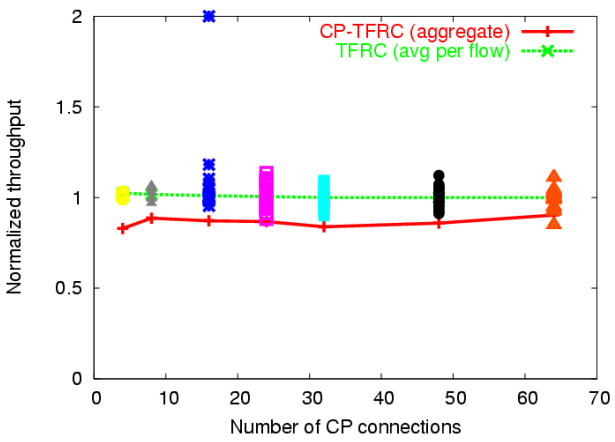


Figure 4.7: CP-TFRC: Number of constituent CP flows.

4.3.2 Comparing TFRC and CP-TFRC

Our goal in this section is to compare aggregate CP-TFRC traffic using a single flowshare with competing TFRC flows sharing the same cluster-to-cluster data path. While it would be of interest to know how well CP-TFRC performs with respect to competing TCP flows, or other flow types, our principle concern is that of *TFRC-conformancy*. That is, here we are not concerned with evaluating the properties (eg., TCP-compatibility) of a particular congestion control scheme, but rather examining how close we conform to known TFRC flow behavior.

In Figure 4.6, a cluster-to-cluster application consisting of 24 flows competes with a varying number of TFRC flows sharing the same cluster-to-cluster data path. Normalized throughput for protocol X ($F_{norm.X}$) is obtained by dividing the throughput for a single flowshare (F_x) by the mean flowshare in the simulation (F_{mean}).

$$F_{norm,X} = \frac{F_x}{F_{mean}} \tag{4.2}$$

F_x for TFRC (or F_{TFRC}) is equal to the mean TFRC flow throughput across all flows. (The mean throughput for each TFRC flow is averaged across all flows.) This is because each TFRC flow takes the bandwidth equivalent to one flowshare. F_x for CP-TFRC (or $F_{CP-TFRC}$) is equal to the mean throughput for all CP-TFRC flows taken as an aggregate. This is because all CP-TFRC flows *combined* comprise a single flowshare. The value of F_{mean} can be calculated by dividing the aggregate throughput for all flows by the number of flowshares in the system. Here, the number of flowshares is equal to

the number of TFRC flows plus one. A normalized flowshare value of 1.0 represents an average throughtput level for a single flow that is ideally fair. Values greater than 1.0 indicate an average throughput level that exceeds fair, while values less than 1.0 indicate an average level that falls short of fair.

The performance of TFRC flows is presented two ways. First, normalized throughput values for a single run at each given configuration is presented as a series of points. Each point represents the normalized throughput received by a single flow, and illustrates the range in values naturally occurring in a single run. Second, a line connects points representing the *average* (mean) flowshare received by a TFRC flow, additionally averaging this value across 20 different trials of the same configuration. Since the number of TFRC flows completely dominates the total number of flowshares used within the simulation, this value is very close to 1.0 for each configuration.

The CP-TFRC line connects points representing the *aggregate* normalized throughput received by 24 CP flows averaged over 20 trials. For each trial, this aggregate flow competed as only a single flowshare within the simulation. We see from this plot that as the number of competing TFRC flows increases, cluster-to-cluster flows receive only slightly less than their fair share, dropping somewhat when 64 TFRC flows compete for bandwidth along the same data path. These values are within (or very close to) the normal range of variance seen for competing TFRC flows.

Figure 4.7 shows per-flow normalized throughput when the number of competing TFRC flows is held constant at 24, and the number of CP-TFRC flows is increased. Once again, the aggregate CP-TFRC traffic competed with TFRC flows on the same cluster-to-cluster data path for a single flowshare. All other details are similar to the previous plot. We see from this plot that once again aggregate CP-TFRC traffic received very close to its fair share of available bandwidth, with normalize values of greater than .8 throughout. Values are once again within (or very close to) the normal range of variance seen for competing TFRC flows.

We conclude from these experiments overall that CP-TFRC is reasonably successful in achieving the aggregate band with equivalent to a single TFRC flowshare. This is true both as the number of competing TFRC flows is increased, and as the number of application flows comprising aggregate CP-TFRC traffic is increased.

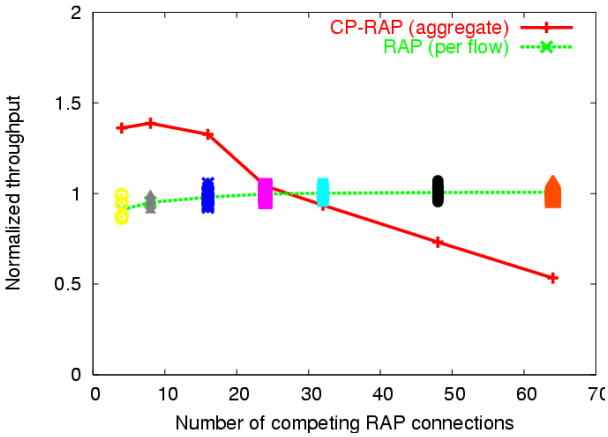


Figure 4.8: CP-RAP: Number of competing RAP flows.

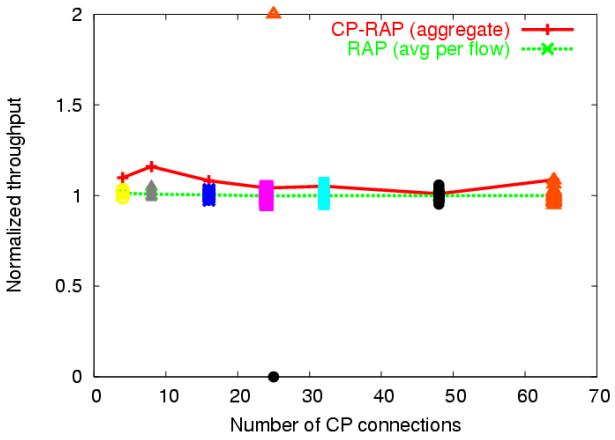


Figure 4.9: CP-RAP: Number of constituent CP flows.

4.3.3 Comparing RAP and CP-RAP

In this section we consider the performance of CP-RAP using the same simulation scenarios described in the previous section. In general, we find that CP-RAP performs far less consistently than CP-TFRC, but reasonably well for a subset of configuration parameters.

Figure 4.8 shows the bandwidth share received by aggregate CP-RAP traffic as the number of competing RAP flows is varied. Once again, CP-RAP flows are configured to send an aggregate of one single flowshare as calculated by the CP-RAP implementation. We see immediately that CP-RAP flows receive somewhat more than a single flowshare when the number of competing connections is 16 or less, and somewhat less than a single flowshare when the number of competing connections is greater than 32. For competing connections within the range $24 \leq n \leq 32$, CP-RAP flows are very successful in obtaining an accurate fair share of available bandwidth.

Clearly our implementation of CP-RAP requires some additional tuning to increase fairness and consistency at both low and high extremes. We note, however, that the variations seen at the extremes were not enough to affect competing RAP flows drastically. Even when aggregate CP-RAP traffic received 1.4 times its bandwidth share, the competing 8 RAP connections received better than .9. Similarly, 16 competing connections seem almost entirely unaffected.

Figure 4.9 shows per-flow normalized throughput when the number of competing RAP flows is held constant at 24, and the number of CP-RAP flows is increased.

As usual, the aggregate CP-RAP traffic competed with other RAP flows for a single flowshare. We see somewhat more variation than in the case of CP-TFRC, but in general normalized throughput for aggregate CP-RAP traffic remained fairly close to a single flowshare with values for most configurations less than or equal to 1.1. These values are within (or very close to) the normal range of variance seen for competing RAP flows.

4.4 Multiple Flowshares

In this section, we consider the problem of scaling aggregate cluster-to-cluster application traffic to support multiple flowshares. As explained in the introduction to this chapter, an application employing m independent flows will naturally receive m flowshares if each flow operates independently using a congestion responsive transport protocol like TCP. In order to avoid penalizing a complex cluster-to-cluster application for using CP, we must provide an equivalent number of flowshares to emulate (in *aggregate*) the bandwidth realized by an independent-flow approach.

Several approaches ([BRS99], [PCN00], [KW99]) were reviewed in Chapter 2 that show how one may implement aggregate congestion control using a single flowshare. To our knowledge, however, we are unaware of any approach that considers the multiple flowshare problem. The reason for this, we believe, 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 and CP-RAP. After discussing the issue in some detail, we present a new technique, *bandwidth filtered loss detection*, or *BFLD*, that solves the problem. Our approach works by reducing the packet arrival and loss events that will be considered by a given congestion control algorithm. With an appropriate level of loss feedback, the algorithm will continue to correctly estimate the bandwidth available to a single flow under a particular set of network path conditions.

4.4.1 Naive Approach

A *naive approach* to achieving multiple flowshares is simply to have each cluster-to-cluster application endpoint multiply the estimated bandwidth availability value B

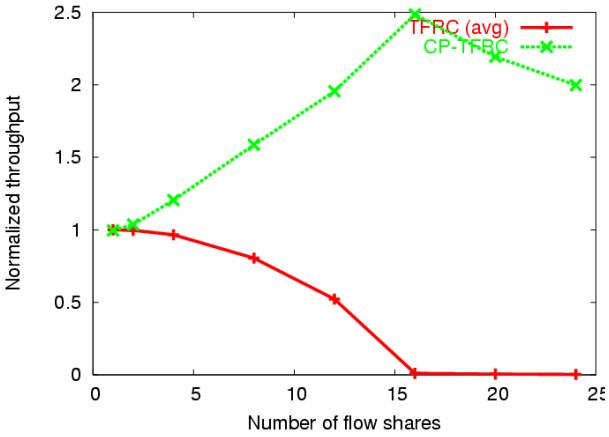


Figure 4.10: CP-TFRC: Multiple flow-shares using the naive approach.

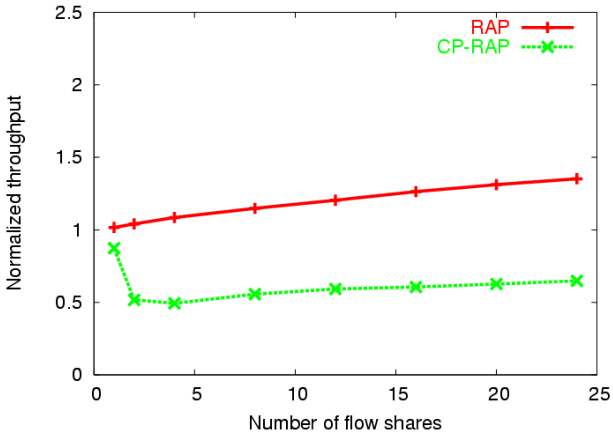


Figure 4.11: CP-RAP: Multiple flow-shares using the naive approach.

by a scaling factor m, where m is the number of flows in the application. (Recall from Section 4.2 that B is estimated for a single flowshare and stored in the AP state table at the location $NET.bw$. m is stored at the location $FLOW.num$.) Thus, each endpoint behaves as if the bandwidth available to the application as a whole is mB instead of B. No other mechanisms or techniques are employed.

One might justify this 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 the change. Ideally, the system will settle on a new value of B which, when multiplied by m, results in the appropriate congestion-controlled level that would have otherwise been achieved by m independent flows.

Figure 4.10 and Figure 4.11 show that this is not the case. For each of these simulations, the number of flows competing with the cluster-to-cluster application's aggregate flow is held constant at 24. The number of flowshares used by cluster-to-cluster flows is then increased using the naive approach. In other words, the aggregate send rate of the cluster-to-cluster flows is set to $k \leq m$ times the estimated available rate reported by CP. The factor k is given by the x-axis. The normalized throughput ratio (with 1.0 representing perfect fairness) is given by the y-axis.

In Figure 4.10, increases in the number of flowshares cause the average bandwidth received by an average competing TFRC flow to drop unacceptably low. By $k = 16$, TFRC flows receive virtually *no* bandwidth, and beyond $k = 16$, growing loss rates

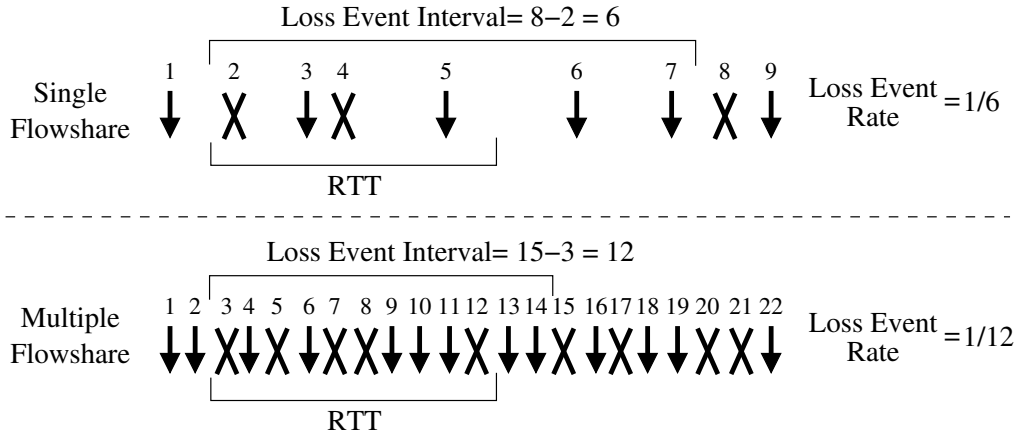


Figure 4.12: Loss event rate calculation for TFRC.

eventually trigger the onset of congestion collapse.

The results shown in Figure 4.11 are less dramatic, but nevertheless, distinctive. Even when $k = 2$, cluster-to-cluster flows receive nearly half the bandwidth they should receive, a pattern that continues as values of k are increased up to 24.

From these two examples, we conclude that *single-flow congestion control algorithms cannot be scaled to multiple flowshares using the naive approach.* Furthermore, the manner in which bandwidth availability calculations fail depends entirely on the algorithm. Different algorithms will fail in different ways. A congestion control algorithm that does not fail may exist, but in our work we are trying to leverage established algorithms.

4.4.2 Handling Packet Loss

In the case of CP-TFRC, recall that RTT and loss event rates are the primary inputs to Equation 2.2. We note that increasing the cluster-to-cluster 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 cluster-to-cluster 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 number of 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. This is true because, 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 *longer* interval, or a smaller loss event rate, and hence an inflated available bandwidth calculation.

This situation is depicted in Figure 4.12. In the single flowshare case, a loss event begins with packet number 2 and ends with packet number 8, the first packet lost after the round trip time dampening period has expired. This gives a loss event interval if $I = 6$ and a loss event rate of $R = 1/6$. In the multiple flowshare case, a loss event begins with packet number 3 and ends with packet number 15. While the raw packet loss rate is roughly the same as the single flow case, lost packet number 15 occurs much *earlier* than lost packet 8 in the single flowshare case. Instead of a smaller loss event interval value, however, the value $I = 12$ is much *larger* and the resulting loss event rate is much smaller $R = 1/12$. This is because of the dense arrival stream during the round trip time dampening period.

The situation for CP-RAP is somewhat different. When a packet loss is detected in RAP, a response will be made which reduces the available bandwidth value by $\beta = 0.5$, in keeping with its AIMD control algorithm. Like TFRC, a damping period of one RTT must then elapse before another packet loss may trigger a second backoff event. In this case, drastically increasing the number of packets in the cluster-to-cluster aggregate traffic has the effect of sharply decreasing the time elapsed beyond the damping interval before another lost packet is encountered. The result is that the algorithm applies backoff behavior too frequently, and available bandwidth values remain below what they should be.

In a sense, each 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, while in CP-RAP, too many packets artificially increases the frequency of multiplicative decrease events. In both cases, the algorithms have been tuned for the *appropriate* amount of feedback which would be generated by a flow source that is conformant to a single flowshare sending rate.

4.4.3 Bandwidth Filtered Loss Detection

Our solution to the problem of loss handling 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*, estimate B_{avail} (stored in the state table at $NET.bw$) which is calculated by the congestion control algorithm employed at the AP. This represents the congestion responsive sending rate for a single flowshare as defined by a particular congestion control scheme like TFRC or RAP. Second is the *aggregate arrival bandwidth*, or B_{arriv}. The value B_{arriv} is the total bandwidth currently being used by the cluster-to-cluster application. The value of B_{arriv} is maintained as a weighted moving average.

An important point to note is that the value of B_{arriv} includes not only actual packet arrivals, but an estimate of lost packets as well. B_{arriv} is calculated from the complete arrival stream where the size of lost packets is estimated from long-term moving averages and the time of arrival, were the packet not lost, is interpolated. (See Section 2.3.2 for how this may be calculated.)

Using 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*. (The term "packet event stream" refers to both packet arrivals and losses over a given time interval.)

To determine whether a packet arrival or packet 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 $[0, 1.0]$. If r is within the subinterval $0 \leq r \leq F$, then an arrival or loss 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 4.13 illustrates the effect of this process for TFRC. In this figure, we see that a subset of the multiple flowshare packet

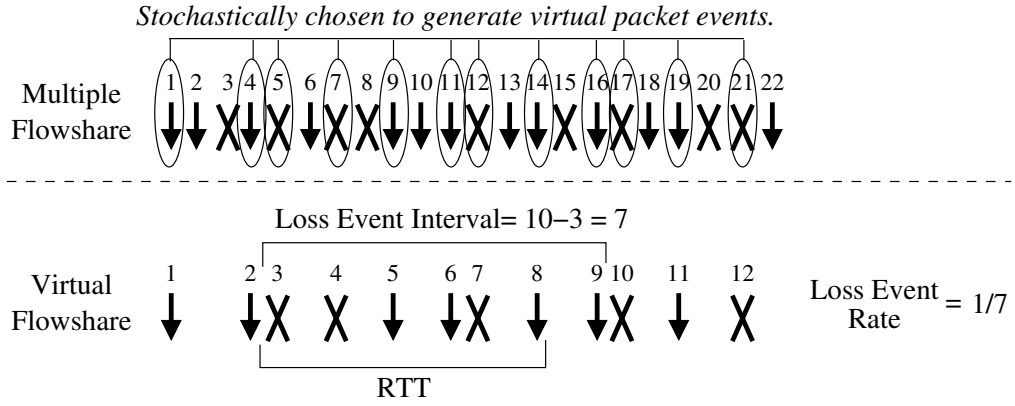


Figure 4.13: Virtual packet event stream construction by BFLD.

event stream is stochastically chosen to generate a virtual packet event stream. Virtual sequence numbers are assigned to packet events within this stream. As a result, the TFRC calculation for the loss event interval decreases from 12 to 7, remedying the problem illustrated in Figure 4.12. An interesting feature of this techniques is that it can be applied *regardless of the number of flowshares used* by the cluster-to-cluster application. This is because the factor F adjusts with the amount of aggregate bandwidth used.

4.4.4 Evaluation

Figure 4.14 and Figure 4.15 show the results of applying BFLD to the simulations of Figure 4.10 and Figure 4.11 in Section 4.4.1. As before, the number of CP flows and competing TFRC or RAP flows are both held constant at 24, while the number of flowshares taken by CP as an aggregate is increased from $k = 1$ to m.

The results show a dramatic improvement. Normalized throughput for CP-TFRC flowshare configurations are close to .95 as the number of flowshares increases from 12 to 24. Throughput levels achieved by competing TFRC flows are consistently close to 1.0. The reason for this improvement can be seen in Figure 4.16 which shows the mean loss event interval over time. Loss event interval values without BFLD are approximately 17 times more than with BFLD.

For CP-RAP, normalized throughput values increase somewhat at 4 flowshares, but then converge to exactly 1.0 as the number of flowshares increase to 24. Throughput levels achieved by competing TFRC flows are consistently close to 1.0. The reason for this improvement can be seen clearly in Figure 4.17 which shows the number of lost

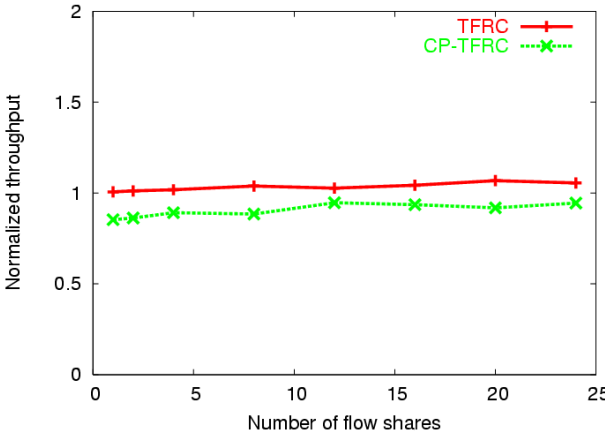


Figure 4.14: CP-TFRC: Multiple flow-shares using BFLD.

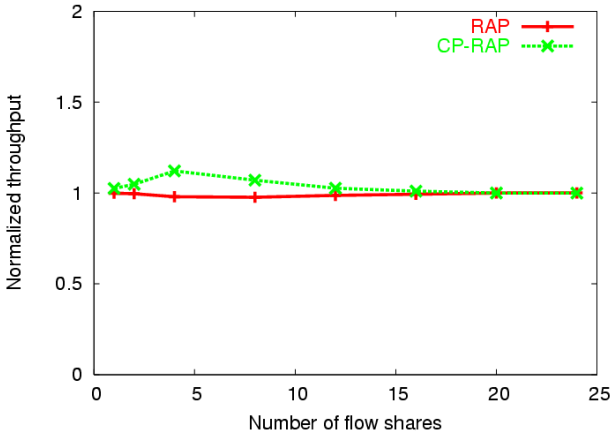


Figure 4.15: CP-RAP: Multiple flow-shares using BFLD.

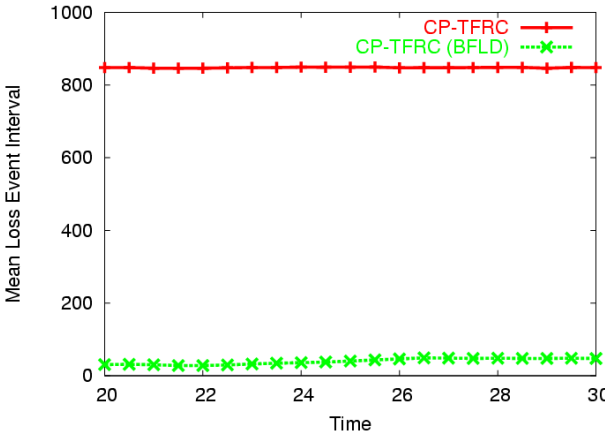


Figure 4.16: CP-TFRC: Mean loss event interval.

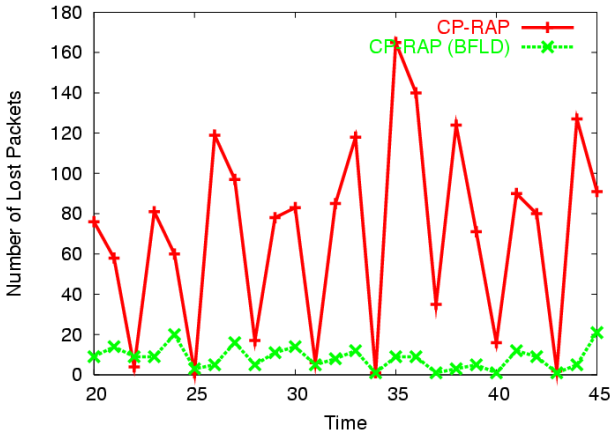


Figure 4.17: CP-RAP: Number of lost packets.

packet events considered by the RAP congestion control algorithm. Without BFLD, the total number of losses is often 10 to 16 times more than the number of virtual losses considered using BFLD.

4.5 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. [1] We then go on to present results showing how BFLD performs in our laboratory testbed with competing TCP connections and various levels of network delay, bottleneck bandwidth, random loss, and background traffic loads.

Our implementation makes use exclusively of the TFRC congestion control algorithm in estimating the bandwidth available to a single application flowshare. There are several reasons for this. First, a real-world implementation was a substantial undertaking and limitations in time and resources did not permit the development of multiple algorithms. (As mentioned above, simulation provided an easier context to explore alternative congestion control schemes.) Second, TFRC is more widely recognized and accepted by the networking research community. This is evident from work surrounding RFC 3448 ("TCP Friendly Rate Control (TFRC): Protocol Specification" [HFPW03]) and the very large number of [FHPW00] citations in the networking literature compared to [RHE99]. Finally, Figure 4.8 seems to suggest that RAP may be less stable than TFRC under some circumstances.

Unlike the simulations in the previous section where we compare CP-TFRC throughput to that of competing TFRC flows, in this section we consider the performance of CP-TFRC (now referred to simply as "CP") with that of competing TCP flows. In part, this is because a real-world implementation of TFRC was difficult to obtain at the time that this work was undertaken. More importantly, however, we wished to examine the performance of CP in the context of real-world traffic. Such traffic is dominated by TCP flows, in part due to the World Wide Web which has emerged as a substantial fraction of Internet traffic as a whole. [SCJO01]

In general, all of our experiments make use of multiple flowshares and BFLD. The number of cluster-to-cluster application endpoints and the number of flowshares have

[1] See Appendix B for a more complete description of our laboratory testbed.

been matched. In addition, an equal number of competing TCP flows have been employed. We find overall that CP does quite well in maintaining TCP-compatibility under a wide variety of network conditions.

4.5.1 Implementation

Our implementation of the CP architecture is a compromise between the approach described in Chapter 3 and an application-level approach. The implementation uses UDP 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. This makes CP easier to deploy.

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. (See Chapter 5 for additional information on AP implementation and performance.) 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" (header processing beyond the IP layer) 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 library handles packetization and inserts a CP header at the beginning of each send buffer. For receive calls, the library removes and processes the CP header, and then passes data to the application level. API calls are provided that allow the application to query network and flow information.

To drive the system, we constructed a test application comprised of two endpoint clusters exchanging data as infinite data sources. Each of m endpoints acts essentially as a rate-based traffic generator, sending mock data to a remote endpoint at a rate equal to B, where B is the available bandwidth estimate for a single flowshare reported by CP. The total aggregate traffic produced by the cluster-to-cluster application is mB. 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 9 hosts (kernel version 2.4) and FreeBSD 4.5 hosts.

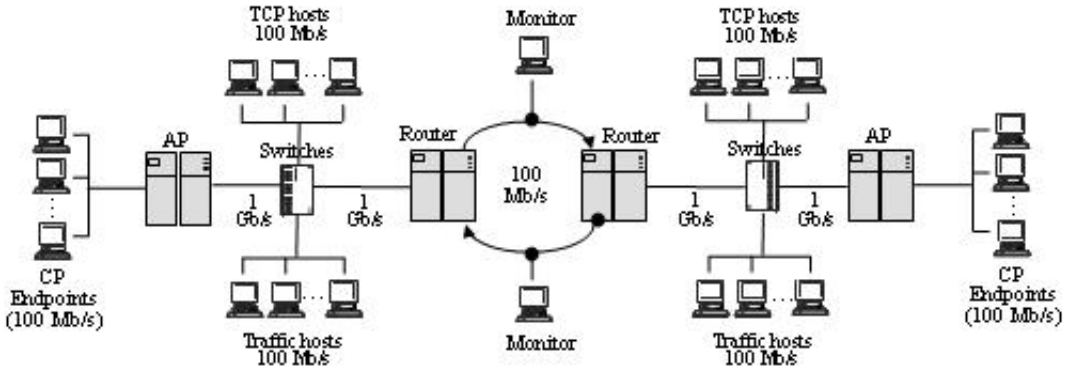


Figure 4.18: Experimental network setup.

4.5.2 Experimental Setup

Our laboratory testbed configuration is shown in Figure 6.6. CP hosts and their local AP on each side of the network represent two clusters that are part of the same cluster-to-cluster 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. Aggregate cluster-to-cluster 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 two clusters of TCP hosts on opposite sides of the network. These hosts use the well-known utility *iperf* [Ipe] 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. These flows allow us to measure the fairness of CP to other application flows sharing the bottleneck link on the cluster-to-cluster data path.

Also sharing the bottleneck link are background flows between traffic hosts on each end of the network. These flows are used in several experiment groupings where the effect of background traffic workload on CP performance is considered. More will be said about the purpose and configuration of these flows in Section 4.5.7.

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* [Ken98] extensions to FreeBSD to monitor queue size and packet drop events on the outbound interface of the bottleneck routers. The resulting log information provides packet loss rates with great accuracy.

Our description here has been brief. See Appendix B for a more complete presentation of our laboratory testbed setup.

4.5.3 Performance Metrics

Overall, our goal is to compare aggregate CP flow performance to that of TCP under various network conditions. In particular, we're looking to see whether CP probing, bandwidth estimation, and state sharing mechanisms result in a TCP-compatible aggregate traffic that successfully realizes the equivalent of m flowshares. To measure our success, we make use of two comparative metrics closely related to those described in [FHPW00].

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.

$$R_{TCP,CP} = \frac{F_{TCP}}{F_{CP}} \tag{4.3}$$

Here F_{TCP} and F_{CP} are normalized flowshares as defined in Section 4.3.2 and represent the average throughput for a single TCP flow or CP flowshare, normalized so 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:

$$C.O.V._{TCP,CP} = \frac{C.O.V._{TCP}}{C.O.V._{CP}} \tag{4.4}$$

C.O.V. [Jai91] 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.

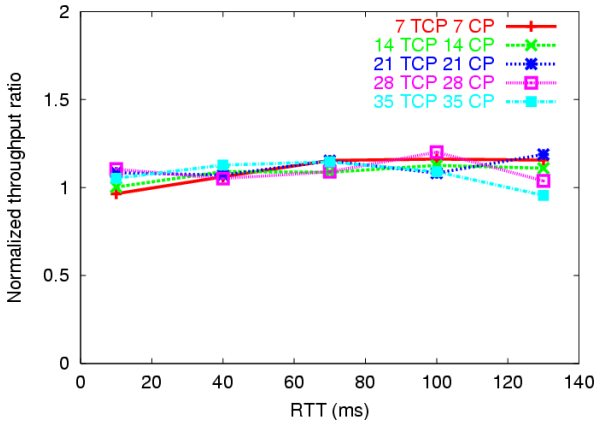


Figure 4.19: Normalized throughput ratio as RTT varies.

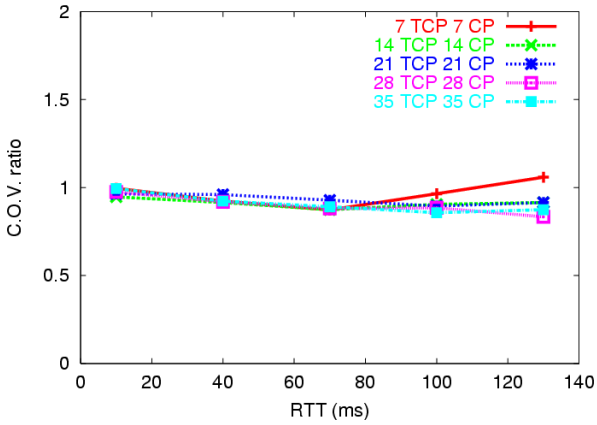


Figure 4.20: C.O.V. ratio as RTT varies.

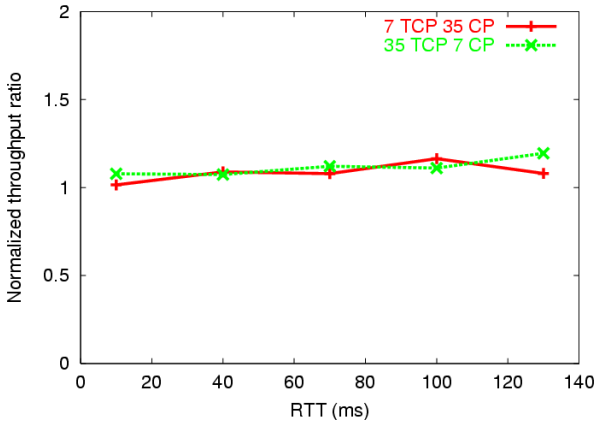


Figure 4.21: Normalized throughput ratio as RTT varies.

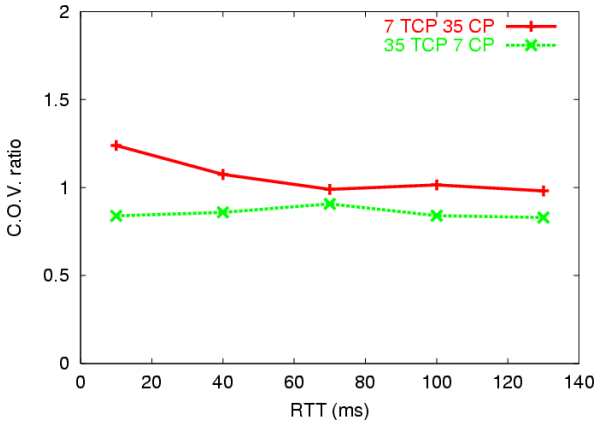


Figure 4.22: C.O.V. ratio as RTT varies.

4.5.4 Delay Experiments

To test CP under various network delay conditions, we made use of the *dummynet* [Riz97] 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 propagation delays between 10 and 130 ms. Various combinations of CP and TCP flows are run to explore the effects of scaling and unequal flow distributions on CP performance. Expressed using the format "$l - m$", where l

is the number of CP flows and m is the number of TCP flows within the same run, these combinations include 7-7, 14-14, 21-21, 28-28, 35-35, 7-35, and 35-7. For each combination, each of m CP flows sends at the reported bandwidth availability rate, for a total of m flowshares of aggregate cluster-to-cluster 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 4.19 shows normalized throughput ratio results for runs with an equal number of TCP and CP flows. In general, values remain very close to 1.0 for all trials, with average TCP flows receiving only slightly more bandwidth. C.O.V. ratios in Figure 4.20 likewise remain fairly close to 1.0 but show somewhat more variance in TCP for the 7-7 configuration at a round trip time of 130 milliseconds.

Figure 4.21 shows normalized throughput ratio results for runs with an unequal number of TCP and CP flows. Like the previous configurations, values generally remain very close to 1.0 with TCP receiving only slightly more bandwidth. Interestingly, Figure 4.22 results seem to indicate that some difference in throughput variation exists when the number of competing flows is unequal. In the 7-35 configuration, 7 TCP flows exhibit much more throughput variation than 35 CP flows. In the 35-7 configuration, the situation is reversed. although the difference is not nearly as extreme. This would seem to suggest that fewer flows generally results in more throughput variation during the run, especially for TCP.

4.5.5 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 4.23 are fairly close to 1.0 for nearly all configurations with an equal number of TCP and CP flows. Figure 4.24 shows a very balanced throughput variance for the same configurations as seen by values that are strikingly close to 1.0 throughout.

There is somewhat less fairness seen in the unequal flow number configurations. While Figure 4.25 shows the 7-35 configuration to be very close to 1.0 in the normalized throughput ratio results for nearly all bottleneck bandwidths, the 35-7 configuration re-

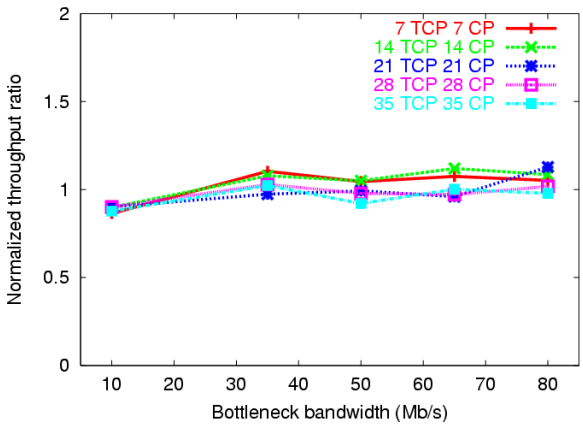


Figure 4.23: Normalized tput ratio as bottleneck bandwidth varies.

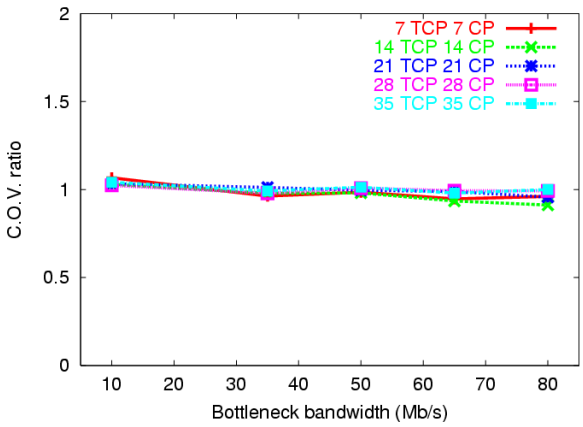


Figure 4.24: C.O.V. ratio as bottleneck bandwidth varies.

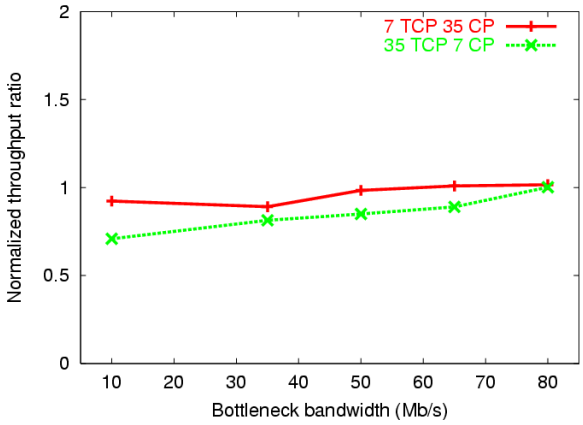


Figure 4.25: Normalized tput ratio as bottleneck bandwidth varies.

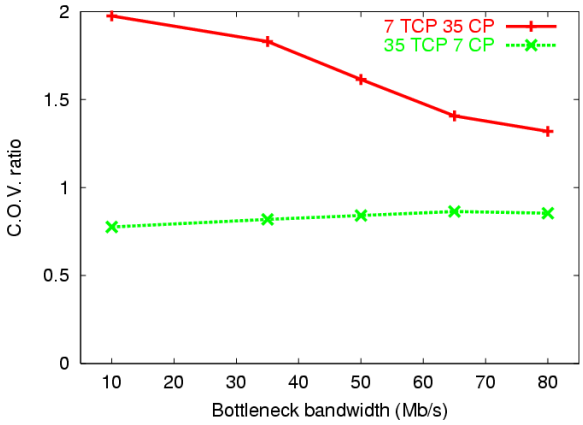


Figure 4.26: C.O.V. ratio as bottleneck bandwidth varies.

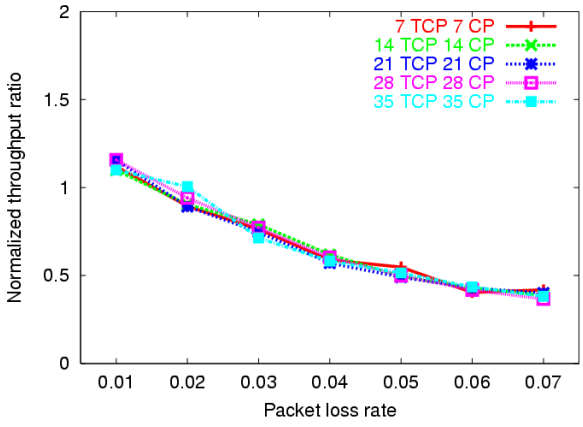


Figure 4.27: Normalized throughput ratio as loss varies.

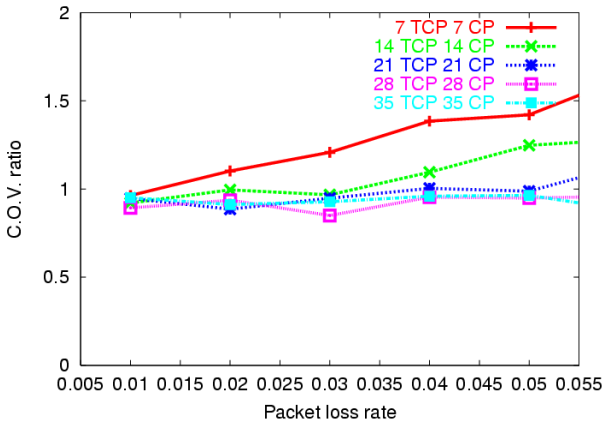


Figure 4.28: C.O.V. ratio as loss varies.

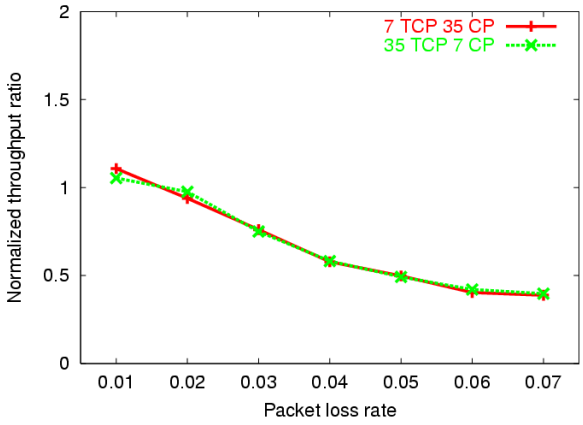


Figure 4.29: Normalized throughput ratio as loss varies.

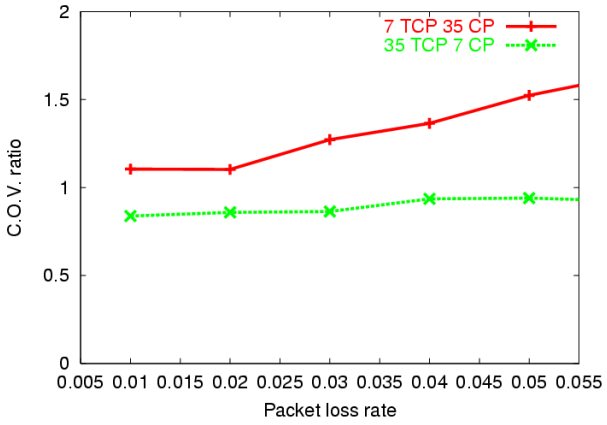


Figure 4.30: C.O.V. ratio as loss varies.

sult in values that are significantly less than 1.0. Values are the lowest when bottleneck bandwidth is at its lowest, and improve steadily as bottleneck bandwidth improves.

Differences in throughput variation are even more stark for unequal flow number configurations. As seen in Figure 4.26, the 7-35 configuration results in a strikingly large amount of throughput variation for the 7 TCP flows compared to the 35 CP flows. The situation is most extreme at very low bottleneck bandwidth levels and improves steadily as bottleneck bandwidth increases. The 35-7 configuration once again results in slightly more throughput variation for the 7 CP flows compared to the 35 TCP flows. This improves only slightly as bottleneck bandwidth increases.

We note here that most laboratory testbed experiments in this dissertation use a 100 Mb/s bottleneck bandwidth configuration. Hence, the fairness issue observed for

very low bottleneck bandwidth configurations in this section (e.g., 10 Mb/s) do not re-surface in our subsequent experiment sets.

4.5.6 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 packet loss levels from 1 to 5%, meanwhile maintaining a constant 40 ms round trip time.

Normalized throughput results in Figure 4.27 show a marked drop in ratio values for equal flow configurations as loss levels are increased. This indicates that TCP flows are increasingly losing bandwidth to CP flows. This is a known problem with TFRC that has been described in [Wid00]. 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 in this context, but it is not supported by FreeBSD version 4.5 and hence the issue could not be explored.

Figure 4.28 indicates that TCP and CP throughput variation is largely similar for most configurations, evident by the fact that most values remain close to 1.0. This is not true, however, for flow configuration 7-7 and, to a lesser degree, 14-14, both of which increasingly show more throughput variation in TCP as packet loss rates increase.

Figure 4.29 shows the same drop in normalized throughput ratio values as packet loss rates increase for unequal flow configurations. Similarly, configuration 7-35 in Figure 4.30 shows the same increase in throughput variation for TCP as loss rates increase. In contrast, the 35-7 configuration once again results in slightly more throughput variation for the 7 CP flows compared to the 35 TCP flows. This improves only slightly as loss rates increase.

Overall, we conclude that CP, using the TFRC congestion control algorithm to perform bandwidth estimation, achieves a high degree of fairness when random packet loss levels do not exceed 2%. Beyond these levels, Equation 2.2 begins to overestimate available bandwidth somewhat, a problem documented by the original authors of TFRC in [Wid00].

4.5.7 Traffic Load Experiments

While testing CP performance under various dummynet loss conditions is instructive, a random loss model is 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 [SCJO01] 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, response sizes, 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 6.6. 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 (see Appendix B) and ran experiments using the same flow configurations used above. Resulting loss rates are shown in Figure 4.35 and Figure 4.36 as measured on bottleneck router queues.

Figure 4.31 shows normalized throughput ratios for equal numbers of TCP and CP flows. Results look improved over *dummynet* random loss trials shown in Figure 4.27, perhaps due to fewer timeouts in TCP as losses are encountered in bursts rather than randomly distributed. As the number of browsers increases to 6,000, throughput ratios are very close to 1.0 for all configurations. At smaller browser numbers, however, 7-7 and 35-35 configurations are less and more than 1.0, respectively. This indicates that CP flows receive somewhat more bandwidth for the former, and somewhat less bandwidth for the latter.

C.O.V. ratio results in Figure 4.32 show very similar levels of throughput variation in TCP and CP. Only the 7-7 configuration shows some divergence from 1.0 with CP exhibiting somewhat more throughput variation for runs with fewer browsers.

The situation with an unequal number of TCP and CP flows is somewhat different for normalized throughput ratios shown in Figure 4.33. Configuration 7-35 show CP flows receiving somewhat more throughput for several configurations, though values

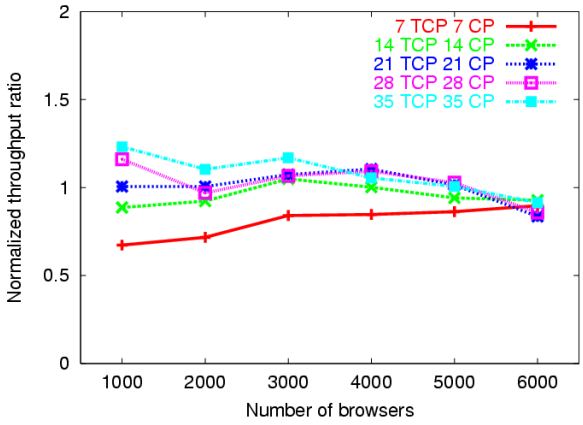


Figure 4.31: Normalized throughput ratio as load varies.

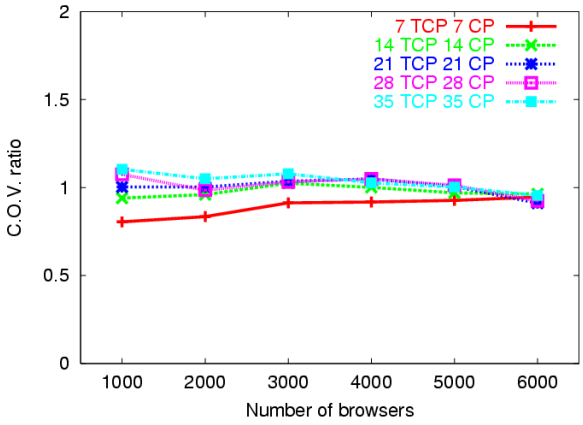


Figure 4.32: C.O.V. ratio as load varies.

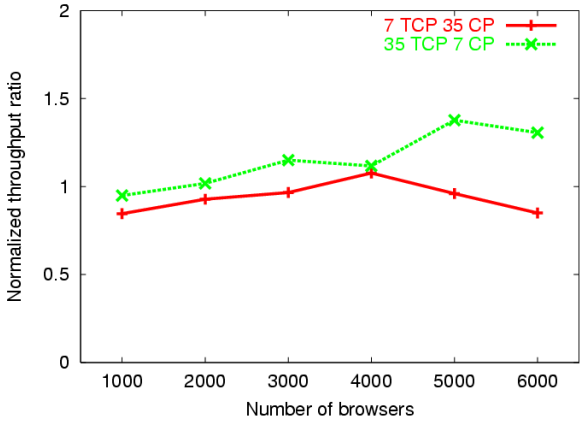


Figure 4.33: Normalized throughput ratio as load varies.

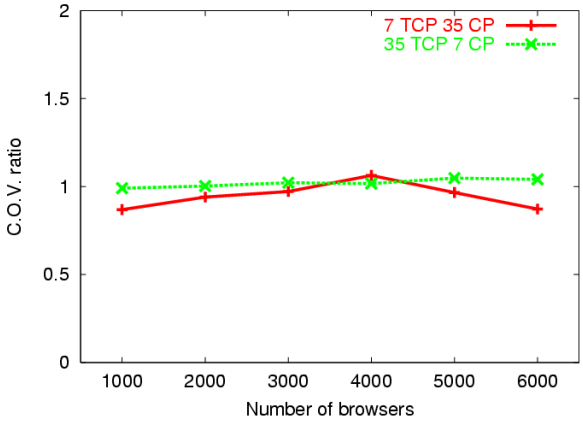


Figure 4.34: C.O.V. ratio as load varies.

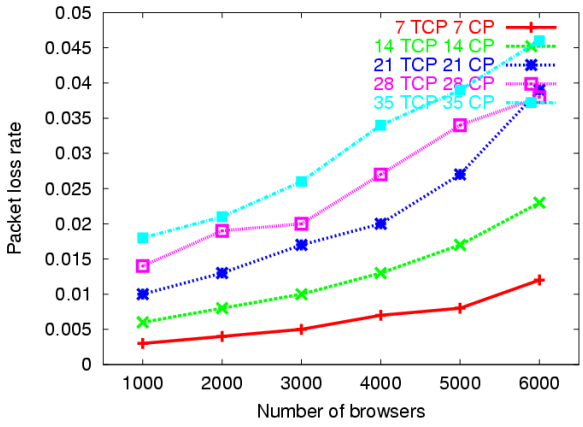


Figure 4.35: Loss rates generated by background web traffic.

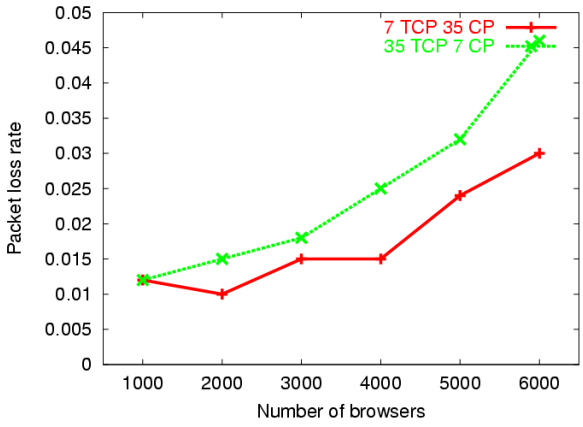


Figure 4.36: Loss rates generated by background web traffic.

are reasonably close to 1.0 throughout. In contrast, configuration 35-7 shows TCP receiving more bandwidth for 5,000 and 6,000 browser levels. Note in Figure 4.36 that configuration 35-7 also experienced substantial loss rates for those browser configurations.

C.O.V. ratio results in Figure 4.34 show very similar levels of throughput variation in TCP and CP. Only the 7-35 configuration shows some divergence from 1.0 with CP exhibiting somewhat more throughput variation for runs with small and large (but not intermediate) numbers of browsers.

4.5.8 Summary

In this section, we described our implementation of CP in a laboratory testbed and looked at various experimental results related to aggregate congestion control. Specifically, we looked at the performance of CP using multiple flowshares under various network conditions (round trip time, bottleneck bandwidth, random loss rates, and Web traffic loads). BFLD was used to enable correct bandwidth estimation as the cluster-to-cluster application uses multiple flowshares.

Our performance study made use of competing TCP flows as a standard for congestion responsiveness and network-friendly resource usage. The metrics used, *normalized bandwidth ratio* and *C.O.V. ratio*, compare the throughput and throughput variance between TCP and CP flows. Our overall goal was to establish the TCP-compatibility of CP under a wide range of network conditions.

Our results show that CP exhibits TCP-compatibility over a wide range of round trip times. Results for different bottleneck bandwidths were likewise solid, although runs with a small number of CP flows versus a large number of TCP flows resulted in somewhat higher throughput variability. For random loss levels, CP performed well when values did not exceed 2%. Beyond this point, CP increasingly receives more bandwidth than competing TCP. These conditions appear to re-create a TFRC modeling flaw that is described in [Wid00]. Loss produced in a more realistic environment where traffic load periodically causes drop tail queue overflow produced much better results. In general, CP behavior was sound for equal flow configurations with greater than 1,000 Web browsers and acceptable for unequal flow configurations with less than 5,000 Web browsers.

Chapter 5

Aggregation Point Implementation and Performance

In this chapter, we discuss aggregation point (AP) implementation and performance within the Coordination Protocol architecture. First, we provide an overview of AP deployment within our laboratory testbed, describing both hardware and software configuration in some detail. Then, we describe a number of kernel module implementation techniques that were used to improve CP packet handling performance. Finally, we present some performance results that quantify the level of performance we achieved with our prototype.

We note here that our approach to AP implementation (i.e., enhanced FreeBSD software routers) is by no means the only one. LINUX might have been used instead, or network processors like Intel's IXP product family. The Coordination Protocol does not presuppose any particular implementation platform any more than it presupposes a particular congestion control algorithm for estimating available bandwidth. Hence, the implementation described in this chapter represents merely a case study; one that may provide useful information for subsequent implementations.

5.1 Overview

Each AP within our laboratory testbed (see Figure B.2) was implemented using an Intel-based PC workstation running FreeBSD 4.9.

The machine is a Dell Precision 650 MiniTower workstation purchased in October of 2003. It has a single 3.20 GHz Intel XEON processor with a 533 MHz front side bus,

8 KB level 1 data cache, 512 KB advanced transfer level 2 cache, and 1 MB full-speed level 3 cache. For main memory, it uses 2 GB of 266 MHz, dual channel DDR SDRAM equipped with ECC in the form of four 512 MB modules. Two 73 GB hard drives are SCSI Ultra 320 controlled and spin at 15K RPM. Network adapters are Intel PRO/1000 MF Dual Port Server Adapters for 1000BASE-SX multi-mode fiber. The adapter has 128 KB of onboard memory and uses DMA to transfer data on a 133 MHz PCI-X bus.

On this PC workstation, we ran FreeBSD version 4.9 Release. The kernel for this system was compiled with optimization options "-02" and "-funroll-loops", and uses a configuration trick to increase clock granularity from 100 Hz to 1000 Hz. Some additional configuration changes include increasing network mbufs to 65536 and enabling *ipfirewall* and *dummynet* options.

CP enhancements to FreeBSD are implemented as a dynamically linked kernel module. Beginning with FreeBSD 3.1, dynamic kernel linker utilities became available that can add and remove kernel extensions from a running FreeBSD system without the need to reboot. This feature was originally designed for developing device drivers, but it quickly became the tool of choice for kernel development in other spheres as well (e.g., Kenjiro's ALTQ [Ken98]). In its compiled form, the CP kernel module is about 45 KB and takes only a few seconds to load using *kldload*.

Adding CP packet handling capability required creating a point of indirection within FreeBSD's native IP handling routine. This routine is called `ip_input()` and is located in the source tree at `src/sys/netinet/ip_input.c`. To create the indirection, a function pointer is declared within the same file with the same arguments and return type as `ip_forward()` (also in `ip_input.c`), and then assigned to point at `ip_forward()` as per the original configuration.

```
static void (*ip_forward_ptr)(struct mbuf *m, int srcrt,
                              struct sockaddr_in *next_hop) = ip_forward;
```

Within `ip_input()`, subsequent calls to `ip_forward()` like the following:

```
ip_forward(m, 0, args.next_hop); /* original */
```

are then replaced with calls to the function pointer

```
(*ip_forward_ptr)(m,0,args.next_hop); /* new version */
```

Upon loading the CP kernel module, the current value of this function pointer is saved, and the pointer is re-assigned to point to a dynamically linked CP routine (`hook_ip_forward()`) that does CP packet classification and calls additional handling routines for recognized CP packets. At the close of the routine, the original function pointer value is used to call `ip_forward()` and return control to FreeBSD's native IP forwarding code.

To identify CP packets, a system call was created that allows an administrator to add IP address and port ranges to a table. Incoming UDP packets with source and destination addresses contained within the table are flagged for CP handling. There are, of course, other ways to handle the packet classification issue, including methods that are less static in character. This one was chosen merely because it met our needs at the time.

Another system call is used to configure the AP to recognize which interface is on its local cluster and which is on the cluster-to-cluster data path. CP packets coming from the local cluster (i.e., outbound packets) will be parsed for state table assignment directives before overwriting their headers with network probe information and forwarding them on toward the remote cluster. CP packets coming from the remote cluster (i.e., inbound packets) will be parsed for network probe information before overwriting their headers with CP report information and forwarding them to their destination endpoint on the local cluster.

A cluster-to-cluster *application table* is maintained and a new entry added each time a new cluster id (cid) is encountered. Associated with each application is also a *flow table* which tracks a small number of per-flow statistics (e.g., throughput) as an application runs. When a new flow id (fid) is encountered, it is added to this table. All operations involving application or flow table access can be performed in constant time. This is because the former is implemented as a direct index table while the latter uses a hash-based approach.

It should be emphasized that CP handling routines at each AP *perform a bounded amount of work*. That is, CP packet handling costs cannot exceed certain fixed overheads that depend upon the packet's position on the forwarding path. For an outbound packet, the following operations will be performed:

- **State table assignments.** An AP will make up to four state table assignments based upon the contents of the four *operation* fields in the CP header. (Some of these fields may contain a *NOOP* flag value indicating that no assignment is

requested.)

- **CP header assignments.** The CP header will be overwritten with six network probe fields, including *timestamp*, *echo timestamp*, *echo delay*, *bandwidth available*, *loss fraction*, and *sequence number*.
- **IP checksum update.** Five IP checksum modifications (see Section 5.2.3) will be made, one for each 32-bit word in the CP header that has been modified.
- **Flow accounting updates.** Several flow-related accounting updates are made, including outbound packet size, flow throughput, and aggregate flow throughput.

The situation for inbound packets is only slightly more complex:

- **Probe information processing.** An AP will process incoming probe information by using *timestamp*, *echo timestamp*, and *echo delay* values to calculate a round trip time sample, and using the *sequence number* to identify any lost packets.
- **Bandwidth estimation.** Using the configured algorithm and updated round trip time and loss rate values, a new bandwidth estimation will be made. For TFRC, this means handling loss event rate calculations and then applying Equation 4.1.
- **State table updates.** Probe and bandwidth estimation values will be used to update several offsets in the NET portion of the state table.
- **CP header assignments.** The CP header will be overwritten with four *report* values taken from the AP state table. While many values can simply be read from the table, a worst case scenario involves four GP addresses with aggregate operation offsets. Even in this case, however, the number of flow offsets involved is bounded.
- **IP checksum update.** Five IP checksum modifications (see Section 5.2.3) will be made, one for each 32-bit word in the CP header that has been modified.
- **Flow accounting updates.** Several flow-related accounting updates are made, including inbound loss rate, packet size, flow throughput, and aggregate flow throughput.

To summarize, bounded handling overhead stems from the small number of fields to be handled and written within the CP header, the limited size of AP state tables, and the fact that all complex aggregation operations in general purpose addresses (e.g., $GP3.mean$) are bounded in input size and updated only on demand.

Finally, it is also important to recognize the *scope* of AP function within the network. While backbone routers in the middle of the network provide forwarding service for tens of thousands of flows, an AP is charged with handling only the flows belonging to a single cluster. Hence, performance requirements are not nearly as stringent. Even if a number of cluster-to-cluster applications are operating simultaneously, the total number of flows and aggregate bandwidth should be manageable using a commodity processor like the one described in this chapter.

5.2 Implementation Notes

In this section, we describe a number of implementation techniques used in the kernel module code to improve efficiency and reduce packet handling overhead. These include avoiding system time calls, use of fixed point calculations, using IP checksum modifications, implementing square root calculations with a lookup table, and employing lazy evaluation for general purpose aggregation operations.

5.2.1 Avoiding System Time Calls

Getting the current clock time is a frequent operation within each AP. Clock time is needed to calculate throughput rates for aggregate application traffic and individual flows, to calculate loss rates at regular intervals, to create timestamps for CP probe headers, to calculate round trip time when receiving a CP probe header, and to time out soft state.

While FreeBSD offers a set of kernel-based system calls for getting the time (`getmicrotime()` and `getnanotime()`) more efficiently than application-level `gettimeofday()`, the *most* efficient way to get the system time is to use the CPU cycle count register. Within `/usr/include/machine/cpufunc.h` is the following assembly routine:

```
static __inline u_int64_t
rdtsc(void)
{
```

```
        u_int64_t rv;

        __asm __volatile(".byte 0x0f, 0x31" : "=A" (rv));
        return (rv);
}
```

This machine-level routine provides a very efficient way to obtain the CPU cycle count as an unsigned, 64-bit value. This value, which is read directly from a system register, increases monotonically without cycling. Using a calibration routine at module load time, one can easily establish the relationship between cycle count and clock time:

```
start_cycles = rdtsc();
getmicrotime(&end_ts);
do_dummy_task(); /* time-intensive dummy task */
end_cycles = rdtsc();
getmicrotime(&end_ts);
diff_usec = (u_int64_t) getDiffUsec(&end_ts,&start_ts);
diff_cycles = end_cycles - start_cycles;
cycles_per_usec = diff_cycles / diff_usec;
```

Now, getting the clock time is easy. Simply get the cycle count and then convert to microseconds (if needed) using `cycles_per_usec` and a single integer multiply.

5.2.2 Fixed Point Calculations

Calculating available bandwidth using Equation 4.1, interpolating lost packet arrival time, and maintaining packet loss statistics all require floating point calculations. Unfortunately, the floating point arithmetic unit (FPU) is not available in kernel code and any attempts at a floating point operation will promptly result in a system exception. It is just as well, however, since floating point calculations are overly time-consuming for the kernel context and a more efficient approach is needed. Fortunately, *fixed point* is an established alternative for floating point computations; one that relies exclusively on unsigned integer data types.

The idea in fixed point math is to divide an unsigned integer into an *integer* component consisting of some number of leftmost bits and a *fractional* component consisting of some number of rightmost bits. For example, a 32-bit unsigned integer may be

divided into 20 leftmost integer bits and 12 rightmost fractional bits. To convert an integer into fixed point, use the bit shift operator to "scale" the original representation by shifting it left by 12 bits. (Care must be taken in fixed point to make sure values can be represented correctly with fewer digits.) To convert a fixed point number to an integer, use the bit shift operator to "unscale" the value by shifting it right by 12 bits. (Rounding it first may be desirable.) A convenient set of C macros for performing these operations are as follows:

```
#define FP32_SHIFT 12
#define FP32_UNIT (((u_int32_t)1) << (FP32_SHIFT))
#define FP32_HALF (((u_int32_t)1) << (FP32_SHIFT-1))
#define FP32_SCALE(x) ((x) << FP32_SHIFT)
#define FP32_UNSCALE(x) ((x) >> FP32_SHIFT)
#define FP32_UNSCALE_ROUNDED(x) (((x) + FP32_HALF) >> FP32_SHIFT)
```

The benefit obtained by using fixed point representations is that numeric calculations can be performed with bitwise shift and integer math operations. To see this, consider the rules for fixed point arithmetic using unsigned integers:

- **Adding two numbers.** Make sure both numbers have been scaled, and then simply add. (Fractional carry works correctly without any additional steps.)
- **Subtracting two numbers.** Make sure both numbers have been scaled, and then simply subtract them. (Fractional borrow works correctly without any additional steps.)
- **Multiplying two numbers.** For a fixed point number and a whole number, simply multiply them. For two fixed point numbers, multiply them and then unscale the result by shifting the product 12 bits to the right. The result is still a fixed point number.
- **Dividing two numbers.** For a fixed point dividend and a whole number divisor, simply divide them. For two fixed point numbers, scale the dividend by shifting it 12 bits to the left, then divide. The result is still a fixed point number.

Rules for signed fixed point are only slightly more complex. Interestingly, most Intel machines will hold the sign of a number that is scaled or unscaled using the bit shift operator (<< and >>).

CP uses both 32-bit and 64-bit fixed point depending upon the range of values involved and the need for precision. Floating point values are kept in fixed point representation at all times. 24-bit state table fields also use fixed point representations (e.g., *NET.loss*) to inform endpoints of fractional values.

5.2.3 IP Checksum Modifications

Each application packet arriving at an AP will have its CP header overwritten. For outbound packets, state table assignment operators will be overwritten with network probe information. For inbound packets, network probe information will be overwritten by CP report information meant for the destination host.

An unavoidable complication that results from modifying the CP header during forwarding is that the IP checksum associated with the packet becomes invalid. One way to handle the situation, of course, is to recompute the checksum in its entirety. This fairly complex operation is expensive, not only because of the header manipulations required, but because it must read every last byte of the packet in the process.

Fortunately, a better option exists. Instead of recomputing the entire checksum, the current checksum can simply be modified based on the bytes that have changed. (RFC 1071 [BBP88] mentions the notion of incremental updates and provides some discussion of the mathematical basis for it.) The code excerpt below was authored by Darrell Anderson from Duke University (with some very slight modifications) and pays tribute to code in Darren Reed's *IPFilter* implementation. The routine, which was used in our CP kernel module implementation, modifies the current IP checksum based upon changes to 32-bit word units.

```
#define LONG_SUM(in) (((in) & 0xffff) + ((in) >> 16))
#define CALC_SUMD(s1, s2, sd) { \
    (s1) = ((s1) & 0xffff) + ((s1) >> 16); \
    (s1) = ((s1) & 0xffff) + ((s1) >> 16); \
    (s2) = ((s2) & 0xffff) + ((s2) >> 16); \
    (s2) = ((s2) & 0xffff) + ((s2) >> 16); \
    if ((s1) > (s2)) (s2)--; \
    (sd) = (s2) - (s1); \
    (sd) = ((sd) & 0xffff) + ((sd) >> 16); \
}
```

```
void udp_rewrite_word(struct mbuf *m, int off, u_int32_t new_val)
{
    struct udpiphdr *uihdr = mtod(m, struct udpiphdr *);
    u_int32_t old, sum1, sum2, sumd, len;

    if (m->m_pkthdr.csum_flags != CSUM_UDP) {
old = *((u_int32_t *)(mtod(m, caddr_t) + off));
sum1 = LONG_SUM(ntohl(new_val));
sum2 = LONG_SUM(ntohl(old));
CALC_SUMD(sum1, sum2, sumd);
if ((sumd = (sumd & 0xffff) + (sumd >> 16)) != 0) {
    sum1 = (~ntohs(uihdr->ui_sum)) & 0xffff;
    sum1 += ~(sumd) & 0xffff;
    sum1 = (sum1 >> 16) + (sum1 & 0xffff);
    sum1 = (sum1 >> 16) + (sum1 & 0xffff); /* again */
    if ((uihdr->ui_sum = htons(~(u_short)sum1)) == 0) {
uihdr->ui_sum = 0xffff;
            }
}
*((u_int32_t *)(mtod(m, caddr_t) + off)) = new_val;
    }

    *((u_int32_t *)(mtod(m, caddr_t) + off)) = new_val;

    if (m->m_pkthdr.csum_flags == CSUM_UDP) {
len = m->m_pkthdr.len - sizeof(struct udpiphdr);
uihdr->ui_sum = in_pseudo(uihdr->ui_src.s_addr, uihdr->ui_dst.s_addr,
  htons((u_short)len + sizeof(struct udphdr) + IPPROTO_UDP));
    }
}
```

5.2.4 Square Root Calculations

Square roots in the TFRC equation (Equation 4.1) pose a difficult problem. They are expensive to compute yet must be used on a per-packet basis to update the current bandwidth availability estimate for a single flowshare ($NET.bw$ in the state table). To get around this problem, we employ a pre-calculated lookup table.

To understand the approach, we first observe that each square root expression in Equation 4.1 contains only two variables: b, the number of packets acknowledged by a single TCP ACK packet, and p, the loss rate on the interval $[0, 1.0]$. As explained in Section 4.2.1, b is merely a constant with a value of 1 or 2. Constructing a lookup table, then, requires merely anticipating possible values for p that might be encountered as an application runs over a particular network path.

Recall from Section 2.3.2 that loss rate is handled in TFRC as a *loss event rate* which employs a RTT dampening mechanism to avoid responding to packet loss more than once during the same round trip time interval. Furthermore, it uses a *loss history* to compute a weighted average of the last eight loss events. Both of these techniques work to smooth bursts in packet loss behavior and make values of p resilient to spikes. Our approach, then, is provide lookup table coverage for a reasonable range of values and then use brute force calculations when a truly anomalous value for p comes along.

The square root lookup table is wrapped in a function called `fpsqrt32()`. This function takes a 32-bit fixed point input parameter and treats it as an integer index into the lookup table contained within. This table has 2732 values which cover all possible fixed point values in the range $[0, 0.6666]$. (Remember that each fractional unit value for a 12-bit fractional representation space is $1/2^{12}$.) At four bytes per value, the table occupies approximately 10 KB of kernel memory.

```
u_int32_t fpsqrt32(u_int32_t x)
{
  static u_int32_t root[] = { 0, 64, 90, 110, ... 3343, 3343, 3344 };

  if ( x > 2731 ) {
    return fpsqrt32_compute(x);
  }
  return root[x];
}
```

For values that lie outside the input range, an additional function is provided for calculating the square root of a 32-bit fixed point value on the fly. The code is shown below and has been adapted from a contribution by Ken Turkowski to the popular series of books known as *Graphics Gems* [Pae95]. Resulting values were exhaustively tested and are accurate up to three decimal digits.

```
u_int32_t fpsqrt32_compute(u_int32_t x)
{

    register unsigned long root, remHi, remLo, testDiv, count;

    if (x == 0) return 0;

    root = 0;   /* Clear root */
    remHi = 0;  /* Clear high part of partial remainder */
    remLo = x;  /* Get arg into low part of partial remainder */
    count = 15 + (FP32_SHIFT >> 1);      /* Load loop counter */
    do {
        remHi = (remHi <<2) | (remLo >> 30); /* Get 2 bits of arg */
        remLo <<= 2;
        root <<= 1;   /* Get ready for the next bit in the root */
        testDiv = (root << 1) + 1;           /* Test radical */
        if (remHi >= testDiv) {
            remHi -= testDiv;
            root += 1;
        }
    } while (count-- != 0);
    return(root);
}
```

5.2.5 Lazy Evaluation for GP Aggregation Operations

In Section 3.2.5, we described the structure of general purpose state table addresses ($GP1$ through $GP250$). In summary, GP addresses are used by flows in a cluster-to-cluster application to solve application-specific problems. The first 128 offsets are

writable by application flows (each flow using its flow id as an offset) using a representational format defined by the application. Offsets in the range $[128, 255]$ contain the results of various aggregation operations. Some of these operations include *min*, *flow id of the min value*, *max*, *flow id of the max value*, *sum*, *mean*, *logical OR*, *logical AND*, etc.

With so many aggregation offsets, the question becomes how an AP can keep them all updated without allowing performance to suffer. After all, a single write to the flow-writable range $[0, 127]$ could potentially result in up to 128 updates to values in the range $[128, 255]$. Since a single CP packet holds up to four assignment operations, this could mean up to 512 updates total.

The answer lies in the well-known evaluation technique called *lazy evaluation*. Essentially, aggregation operations are performed on demand *only when explicitly requested* by a flow using the mechanisms described in Section 3.2.2. For example, the *logical OR* offset for general purpose address 7 ($GP7.OR$) will be updated only before the value is actually required to assign a report value in some CP header.

Further efficiency can be achieved by the use of *flag bits* in each state table offset. (Recall that 32-bit offsets are used in our state table implementation, each of which holds a 24-bit value and has 8 bits left over for miscellaneous use.) An *assigned* bit is used to distinguish between offsets that have been assigned and those that have not. Use of this flag reduces the number of values that need to be considered when performing aggregation operations. A *dirty* bit flag is used to indicate that an offset has been newly assigned. When no dirty bits have been detected for a particular address, then the results of a previous aggregation operation can be reused without calculating a new value.

5.3 Performance

In this section we present results that help to quantify the performance of our AP implementation, and to provide insight into what comprises CP packet handling overhead. First, we present the results of a *gprof* kernel module execution profile. Next, we describe timing measurements made in our laboratory testbed that quantify per packet overhead for various scenarios. Finally, we look at overall throughput results in comparison with the baseline IP forwarding capabilities of FreeBSD running on the machine described in Section 5.1.

5.3.1 Kernel Module Execution Profile

Table 5.1 shows a CP kernel module execution profile made with the well-known GNU utility *gprof*. In general, obtaining a FreeBSD kernel module profile using *gprof* is not an easy task. In part, this is because profiling requires a specially compiled kernel. More importantly, however, is the fact that a kernel module is dynamically linked. After loading the specially compiled module, a separate symbol table and offset address must be manually linked to the executables before *gprof* can identify module-based routines. [1]

Results show AP overhead to be dominated by TFRC calculations, comprising some 24% of overall cycle time. These calculations are done on a per-packet basis and use 64-bit fixed point computations for precision. Clearly more work needs to be done to improve the efficiency of this computation. Perhaps using 32-bit fixed point computations might help, as might pre-computing portions of the equation with predictable sub-results. Another idea might be to amortize the cost of the calculation over a small number of packets rather than re-computing for each packet arrival. Alternatively, employing a simpler bandwidth estimation algorithm might be advantageous.

The next three heavy hitters include `is_cp_host()`, `update_incoming_bwavail()`, and `cp_handler()`. `cp_handler()` is the main CP packet handling routine which includes, among other things, parsing CP headers, writing CP headers, maintaining network and flow statistics, and performing aggregation operations for general purpose registers. In general, it seems that very little can be done to reduce processing requirements for this routine.

`is_cp_host()` checks the CP flow table to see whether a packet belongs to CP or not. As explained above, this table is configured by an administrator using a CP system call designed especially for this purpose. `is_cp_host()` is a heavy hitter because it is called twice for every packet, once for the source address and once for the destination address. The hash algorithm used to check the table, while not particularly complex, is perhaps slightly more complex than it needs to be. An improved version might combine source and destination addresses in some way and then employ a more streamline hash function. The former would reduce the number of calls to `is_cp_host()` by half, and the latter would reduce the computation required for each call.

`update_incoming_bwavail()`, as well as `update_incoming_pktsize()` and

[1] Thanks to Kenneth G. Yocum from Duke University Computer Science for his generous assistance with *gprof* profiling.

Time %	***Calls***	***Function name***	***Brief summary***
24.0	1821	*tfrc_bwavail()*	Calculates bandwidth availability using TFRC equation and loss event history.
12.0	3594	*is_cp_host()*	Checks host table to determine whether UDP packet belongs to a cluster-to-cluster application.
12.0	1821	*update_incoming_bwavail()*	Updates statistics on estimated bandwidth availability (min, max, mean, mean deviation, etc.)
12.0	1797	*cp_handler()*	Main handling routine for a CP packet. Includes header parsing and assignment.
8.0	1779	*update_incoming_pktsize()*	Updates statistics on packet size (min, max, mean, mean deviation, etc.)
8.0	1779	*update_lossrate()*	Updates statistics on packet loss rate (min, max, mean, mean deviation, etc.)
5.0		*hook_ip_forward()*	Manipulates mbuf format, examines header information, and calls CP and IP handling routines.
4.0	8985	*L4_udp_rewrite_word()*	Modifies IP checksum based on changes in a 32-bit word.
4.0	6052	*fp64_update_avg_mdev()*	Updates average and mean deviation values given a new sample. (Uses 64-bit fixed point.)
3.0	3642	*fpsqrt32()*	Uses lookup table to obtain 32-bit fixed point square root value. Calculates if not found in table.
3.0	393	*timevalIncrUsec()*	Increment UNIX *timeval* data structure by specified value.

Table 5.1: Heavily hit functions as revealed by *gprof* execution profile.

`update_lossrate()` which received 8% of CPU cycles each, update statistics used in bandwidth availability estimation. This includes, most importantly, a weighted mean and mean deviation computation done by the helper routine `fp64_update_avg_mdev()`. Once again, 64-bit fixed point values are used for precision during the computation. A more streamline implementation, however, might use 32-bit fixed point values to reduce cycle count. A more efficient mean deviation calculation furthermore may be possible.

5.3.2 Measuring Per Packet Processing Overhead

To measure total per-packet processing overhead, we instrumented our kernel and CP module with timing calls before and after CP handling routines. Four cases were then examined:

- **Empty stub.** Control of IP packet handling is turned over to CP which then promptly returns control back to the IP stack. This measures the overhead of our "hook" (point of indirection) inserted into FreeBSD's native IP forwarding code.
- **Packet classification only.** CP handling includes classifying UDP packets as belonging to CP or not. This requires flow table lookups on source and destination addresses.
- **Normal operation.** CP packets are identified and handled for a single cluster-to-cluster application. Endpoints request report information from the network statistics (NET) and flows statistics ($FLOW$) addresses in the AP state table.
- **Worst case operation.** CP packets are identified and handled for a worst case scenario: 32 cluster-to-cluster applications, each with 128 flows. Each endpoint furthermore cycles through all 250 general purpose addresses in the AP state table, assigning flow offsets and requesting reports exclusively from aggregate function offsets.

To create the worst case operation scenario, endpoint flows were configured to cycle through both cluster ids and flow ids as CP packets were sent over the network. While not a typical use of CP, it was an effective means by which a small number of endpoints could introduce packets from a much larger set of cluster-to-cluster applications and emulate a very large number of flows. CP, in its current implementation, does not

	Mean	*Std. dev.*	*Minimum*	*Maximum*
Stub	111	24	100	1212
Classify only	225	88	172	4164
Normal (outbound)	4068	1038	3380	29032
Normal (inbound)	3415	7000	2796	492160
Worst (outbound)	4945	1122	3716	31184
Worst (inbound)	5796	32609	2776	515260

Table 5.2: CP packet handling overhead measured in cycles.

	Mean	*Std. dev.*	*Minimum*	*Maximum*
Stub	.036	.008	.033	.397
Classify only	.074	.029	.056	1.363
Normal (outbound)	1.331	.340	1.106	9.500
Normal (inbound)	1.118	2.290	.915	161.047
Worst (outbound)	1.618	.367	1.216	10.204
Worst (inbound)	1.897	10.670	.908	168.606

Table 5.3: CP packet handling overhead converted to microseconds.

preclude a single endpoint from participating in multiple applications or maintaining multiple flow identifiers.

For all cases above, 5000 timing samples were collected and stored in kernel memory. Upon termination of the run, the values were dumped to the system log file and then post-processed for various descriptive statistics. Results measured in cycles are given in Table 5.3. These values were converted to microseconds in Table 5.3 using the conversion factor *3056 cycles = 1 microsecond*. This equivalency was obtained using a calibration procedure that considers cycle counts over large time intervals.

We note from Table 5.2 that stub handling overhead is just over 100 cycles, and that adding packet classification approximately doubles this value. Normal and worst case packet handling, in contrast, are 150-250 times larger suggesting that the overhead for indirection and packet classification is relatively small.

Both normal and worst case scenarios show differing overheads for incoming versus outgoing packet handling. For the normal case, inbound packet handling shows 19 percent *less* cycle overhead than outbound packet handling. For the worst case scenario, inbound packet handling requires 17 percent *more* cycles. A key factor here appears

to be the type of state table reports requested by a flow. While flows in the normal scenario request network and flow information that has been pre-computed, flows in the worst case scenario request aggregate function offsets in general purpose addresses. Such functions require iterating across all flow offsets and possibly incurring cache misses in the process.

Also striking is the large variability associated with the inbound direction compared to the outbound direction. Standard deviation values for the normal inbound case are seven times larger than what they are for the outbound case. For the worst inbound case, they are 32 times larger. Maximum values for both reflect this difference as well. Overhead differences of this magnitude would appear to imply the presence of context switching. Indeed, kernel interrupts in FreeBSD have priority levels, and longer handling intervals are more vulnerable to such events.

Figure 5.1 shows packet handling overhead as a cumulative distribution function. Plot (a) is given in terms of clock cycles while plot (b) converts values to microseconds. These plots illustrate the significant difference in overhead between stub or classification overhead and normal operation overhead. Worst case operation overhead is seen to be somewhat more than normal operation overhead, although perhaps somewhat less than one might expect. Slopes are fairly steep indicating that most values lie within a relatively small range. Note, however, the "knee" that occurs in normal and worst case overhead data sets. Values at the knee appear to represent a predominating range, while those at the "tail" jump significantly. Once again, we believe this is due to additional overhead from context switching as the kernel handles higher priority interrupts.

Figure 5.2 shows packet handling overhead for normal and worst case scenarios expanded, with inbound and outbound handling shown separately. In general, the "knee" for inbound handing is higher than outbound handling, particularly for the normal case. Also evident is a heavy tail reflecting the large variability of numbers seen in Table 5.2 and Table 5.2.

5.3.3 Overall Forwarding Performance

In this section, we consider the overall forwarding performance of our AP implementation. To do so, we remove the 100 Mb/s bottleneck link in our laboratory testbed shown in Figure B.2 by reconfiguring it to be 1 Gb/sec. Then, we generate CP traffic in both directions at fixed aggregate rates. That is, sending endpoints simply ignore the bandwidth estimate made by CP and continue to send at exactly the rate given

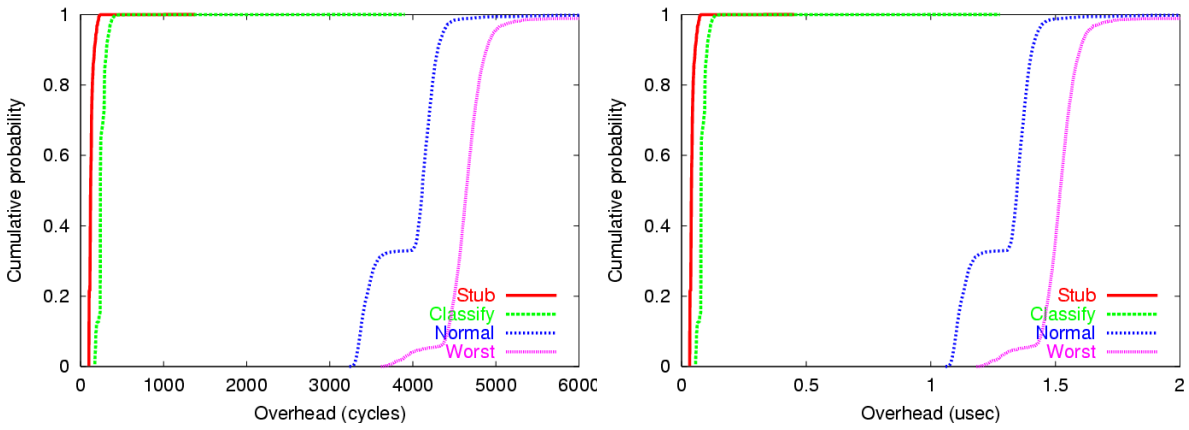


Figure 5.1: AP forwarding performance. Per packet overhead CDF expressed as (a) clock cycles and (b) microseconds.

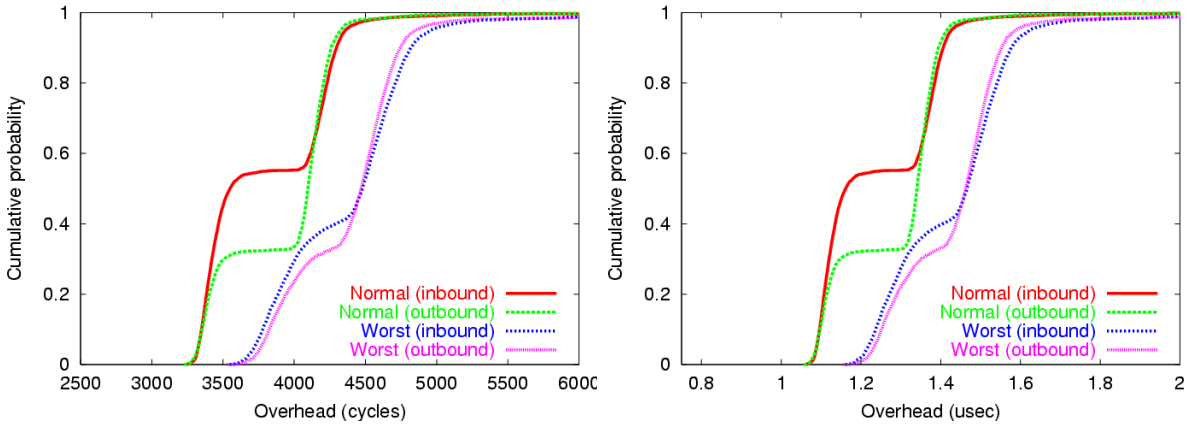


Figure 5.2: AP forwarding performance. Per packet overhead CDF expressed as (a) clock cycles and (b) microseconds.

by their configuration. The idea is to slowly increase the level of aggregate traffic until noticeable packet loss occurs and throughput levels no longer reflect offered load levels. At that point, we have reached the performance ceiling of our implementation.

To quantify this performance limit, we consider three metrics.

- **Throughput.** This is the level of aggregate CP traffic (measured in bits per second or packets per second) forwarded by an AP. *Outbound* throughput can be measured using traffic monitor hosts in the middle of the network (see Appendix B). *Inbound* throughput can be measured by combining receive statistics on application endpoints.

- **Throughput differential.** Calculated as *throughput/offered load*, this is an alternative way of looking at throughput that shows more clearly when forwarding performance is not keeping up. If APs can keep up with the offered load, then values will remain at 1.0. As the APs begin to have trouble, then values will decrease on the interval $[0, 1.0]$.

- **Loss fraction.** This is a receiver-based metric with values on the interval $[0, 1.0]$. Values greater than zero indicate that APs are beginning to have trouble keeping up.

To drive AP performance experiments, we used seven hosts on each cluster. Each host sent UDP or UDP-based CP packets at a fixed rate. Offered loads did not exceed 450 Mb/s aggregate in a single direction, or 65 Mb/s for each host. Packet size (including CP, UDP, and IP headers) was held constant at 1460 bytes.

While we originally planned to explore a number of configuration parameters (number of applications, number of flows, CP report selection, etc.), we found from our experiments that only three performance scenarios were necessary:

- **UDP.** This is a baseline configuration that considers the maximum forwarding performance of our laboratory network without using CP. APs in this configuration do simple IP forwarding, as do the "bottleneck" routers (see Figure B.2).

- **Normal operation.** This is a single cluster-to-cluster application sending CP packets using seven flows. Endpoints request report information from the network statistics (NET) and flows statistics ($FLOW$) addresses in the AP state table.

- **Worst case operation.** This is a worst case performance scenario: 32 cluster-to-cluster applications, each with 128 flows. Each endpoint furthermore cycles through all 250 general purpose addresses in the AP state table, assigning flow offsets and requesting reports exclusively from aggregate function offsets.

Figure 5.3, Figure 5.4, and Figure 5.5 show our overall results. Figure 5.3 and Figure 5.5 (a) present throughput results taken from our monitor host in the middle of our laboratory testbed (see Figure B.2). Throughput, in these plots, refers to the level of aggregate traffic seen *after a single AP*. That is, packets have been received from senders by an AP and forwarded over the cluster-to-cluster data path. Since each AP receives and handles packets going in both directions, throughput levels have been combined for a single total. Offered load refers to the combined traffic generated by application endpoints sending on both clusters of the application.

The results are striking. Aggregation points are able to handle just over 750 Mb/s, or approximately 65,000 packets per second, after which throughput levels off. This performance, we argue, should be adequate for most high-performance applications on the public Internet today. Just as striking is the fact that CP throughput levels, even for the worst case scenario described above, show almost no difference from those of basic IP (i.e., UDP) forwarding. We believe that the explanation for this performance outcome lies in the fact that APs in our laboratory testbed are not the performance bottleneck. Instead, the bottleneck routers, with lesser machine specs than our APs, are the constraining factor for all three scenarios. In support of this theory, we point out that each AP is an Intel-based machine with a 3.20 GHz Xeon processor, 8 KB of L1 cache, 512 KB L2 cache, 1 MB L3 cache, 2 GB of main memory, and a 533 MHz front side bus. In contrast, each bottleneck router is an Intel-based machine with a 1 GHz Pentium III processor, 256 KB L2 cache, 1 GB main memory, and a 133 MHz front side bus. Even with our artificially constructed worst case, we apparently were unable to create a performance bottleneck on our laboratory testbed due solely to AP processing overhead.

Figure 5.3 and Figure 5.5 (b) present throughput results taken from receivers. Throughput, in these plots, refers to the level of aggregate traffic after being forwarded by *both APs*. That is, packets have been handled at the first AP, traversed the end-to-end data path, and then been handled by the second AP. Once again, offered load and throughput values for both directions have been combined since each AP handles traffic from both directions.

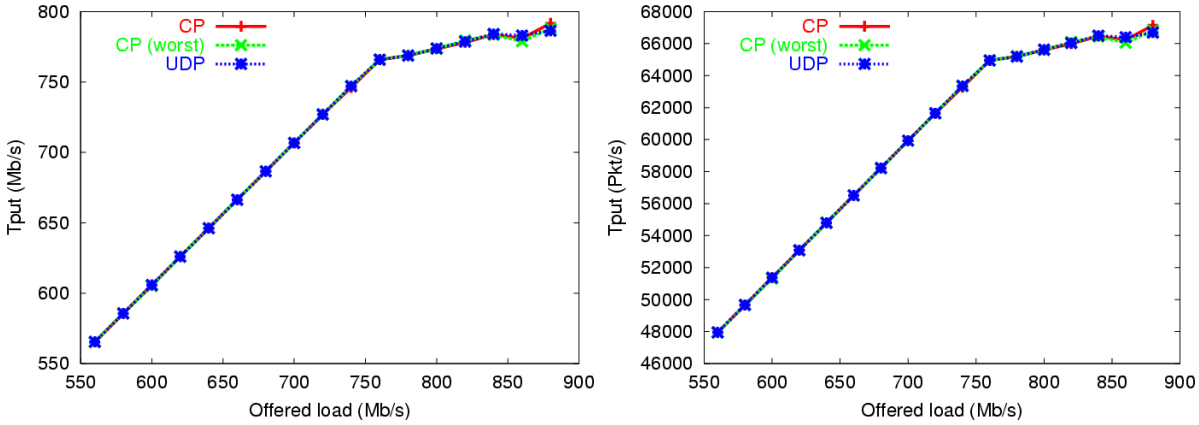


Figure 5.3: AP forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s as measured in the middle of the network.

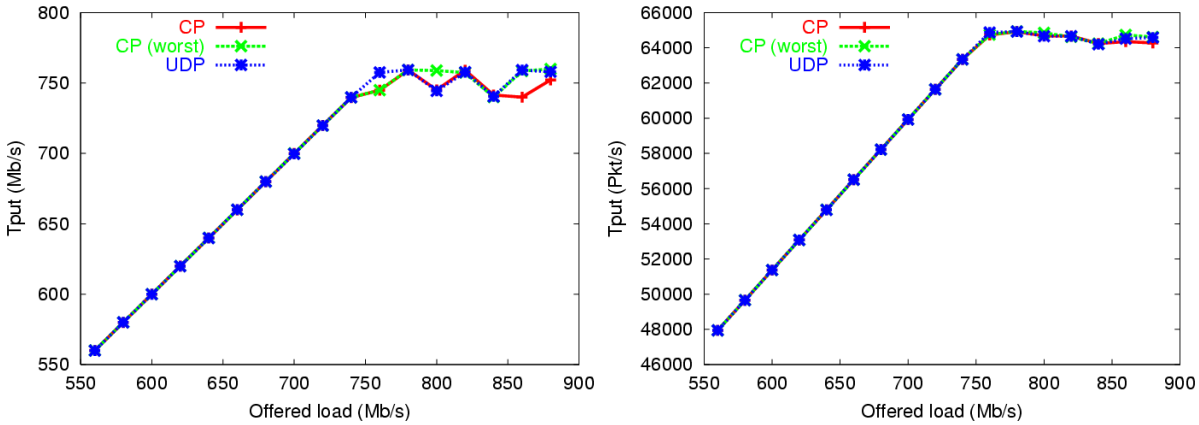


Figure 5.4: AP forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s as measured by receivers.

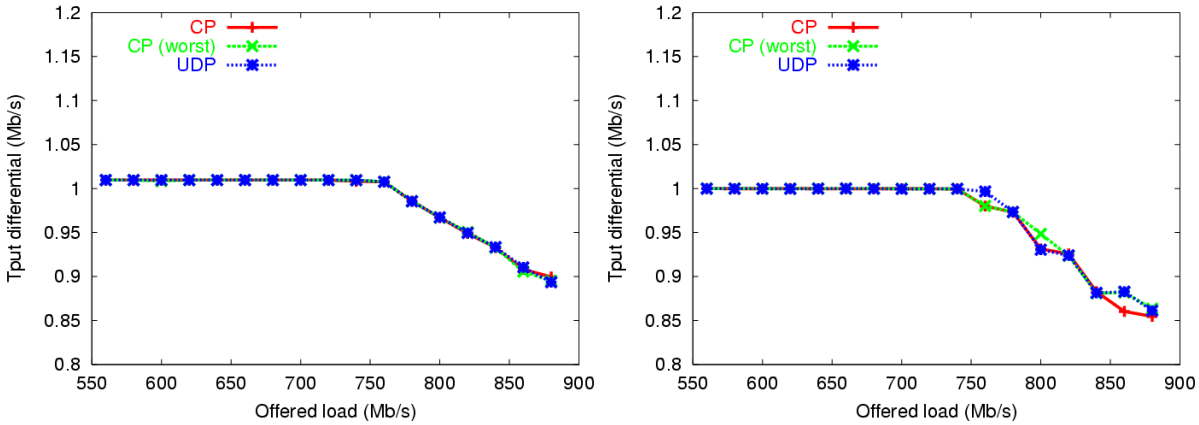


Figure 5.5: AP forwarding performance. Offered load versus throughput differential as measured (a) in the middle of the network and (b) by receivers.

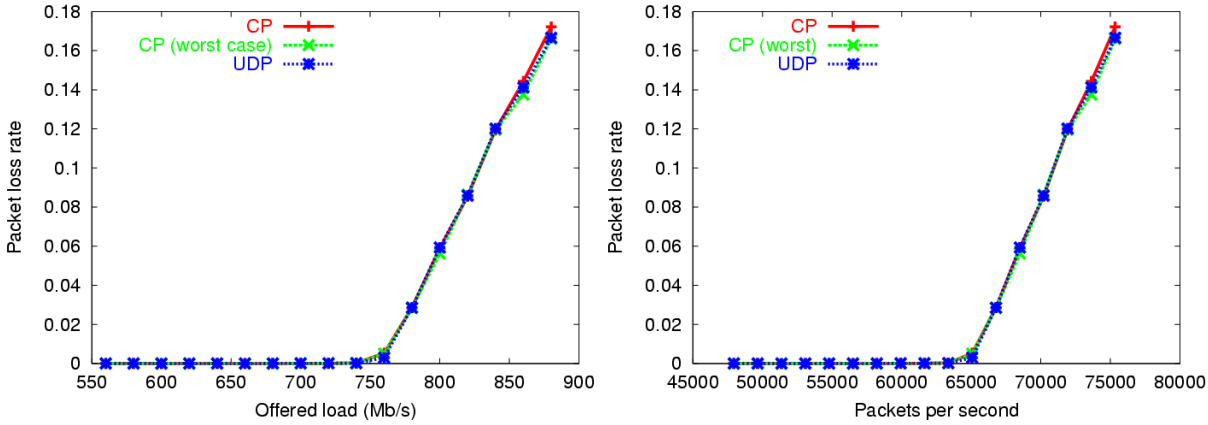


Figure 5.6: AP forwarding performance. Offered load in (a) Mb/s and (b) Pkt/s versus packet loss rate as measured by receivers.

Results are similar to the previous case except that maximum performance numbers drop just slightly for the two CP cases. While the UDP case shows a maximum throughput level of just over 750 Mb/s, the two CP cases (normal and worst) seem to slip slightly at this level. Interestingly, packet per second throughput results do not show this difference, with all cases resulting in the same 65,000 packets per second maximum value.

Loss results in Figure 5.6 once again support the case that no performance difference exists between CP and basic IP forwarding. In plot (a), we see that loss begins to appear at approximately 750 Mb/s, as measured by receivers. In terms of packets per second (plot (b)), 65,000 packets marks the beginning of non-zero loss levels.

The next two series of plots show the forwarding performance of each AP individually, including inbound, outbound, and combined performance numbers. An interesting contrast exists between *AP1* and *AP2* results: while uni-directional forwarding performance for *AP1* begins to level off only somewhat after 380 Mb/s (33,000 pkt/s), *AP2* shows a significant drop before that level. This can be observed by comparing both inbound (Figure 5.7 (a) and Figure 5.10 (a)) and outbound (Figure 5.7 (b) and Figure 5.10 (b)) plots, as well as both combined plots (Figure 5.9 and Figure 5.12). The reason for this asymmetry is not entirely clear. However, we note once again that values drop for all three cases: CP, CP (worst), and UDP. We believe this to be an indication that CP handling is not a critical factor in the performance issue. Instead, it seems most likely that the bottleneck router lying between *AP2* and the monitor host, with its lesser machine specifications, has reached its performance limit. Why

this limit is somewhat less than its companion bottleneck routing host is unclear.

Interestingly, this difference does not manifest itself in plots that focus on throughput measured in packets per second (Figure 5.7 (b) and Figure 5.10 (b), and Figure 5.8 (b) and Figure 5.11 (b)).

5.4 Summary

In this section, we described our AP implementation using a fairly high end but commodity PC workstation running FreeBSD. To achieve the best performance possible, we used kernel-level handling using a dynamically linked kernel module that inserts CP handling into the IP forwarding chain. Various techniques were described to keep our implementation efficient, including the avoidance of system time calls, fixed point calculations, IP checksum modifications, square root lookup table, and lazy evaluation for aggregation offsets in general purpose state table addresses.

Our implementation was able to achieve a throughput capacity of just over 750 Mb/s or 65,000 packets per second. In general, it did not appear that our AP implementation created a bottleneck in our experimental testbed, despite some minor variation in performance results between UDP and CP handling at the 750 Mb/s load level. Our implementation stands as proof that CP can be implemented inexpensively and with a performance level that is adequate for most cluster-to-cluster applications on the public Internet.

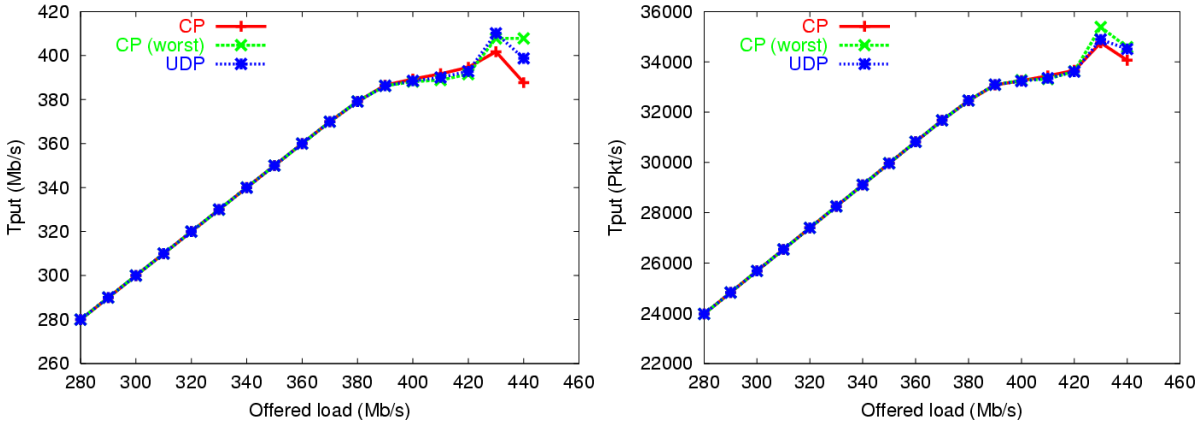


Figure 5.7: AP1 inbound forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s.

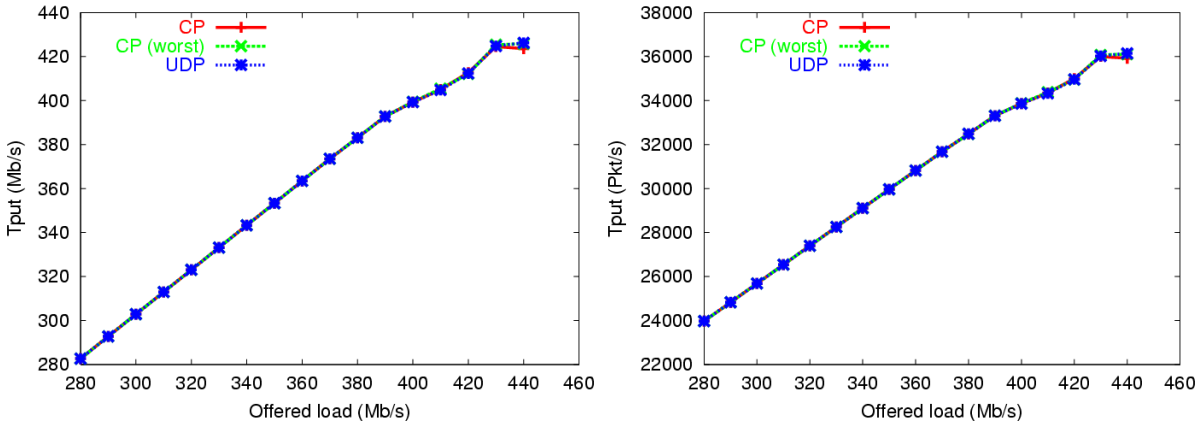


Figure 5.8: AP1 outbound forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s.

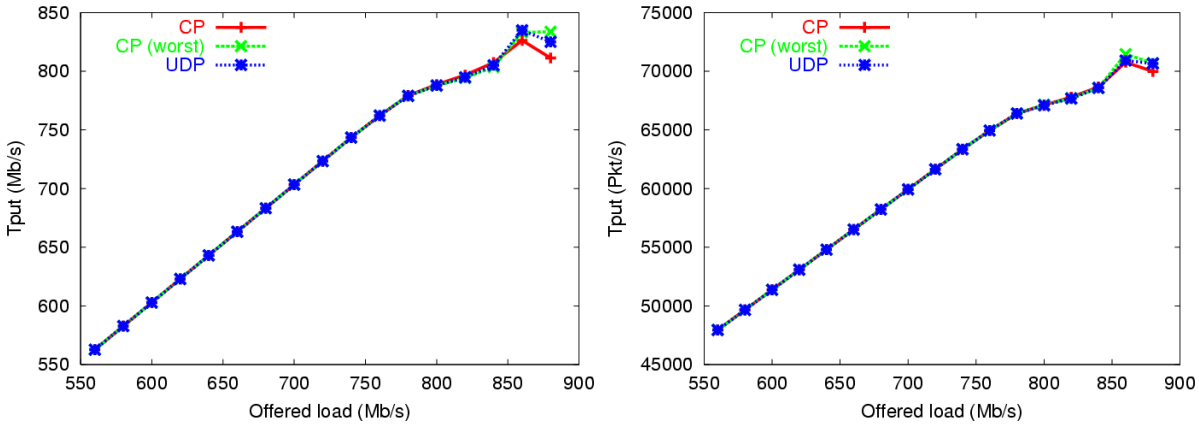


Figure 5.9: AP1 combined forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s.

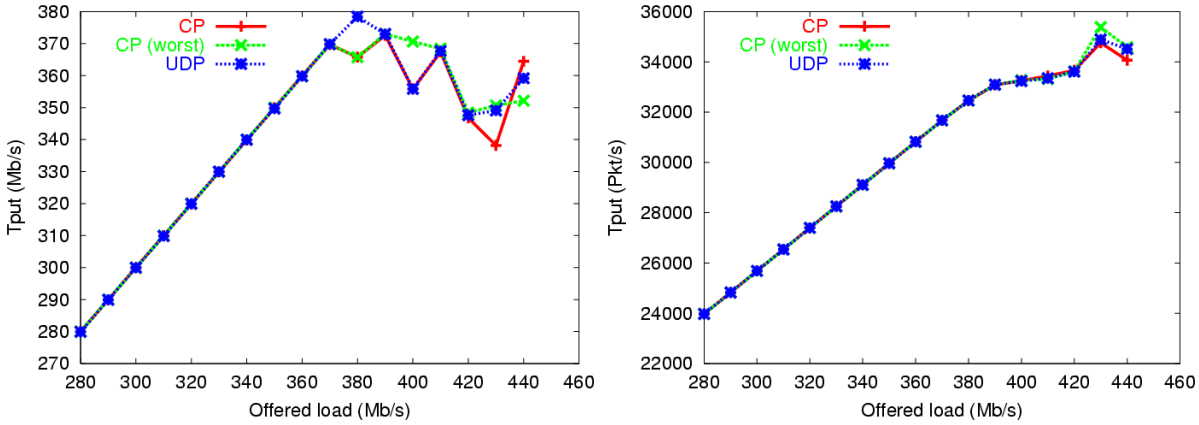


Figure 5.10: AP2 inbound forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s.

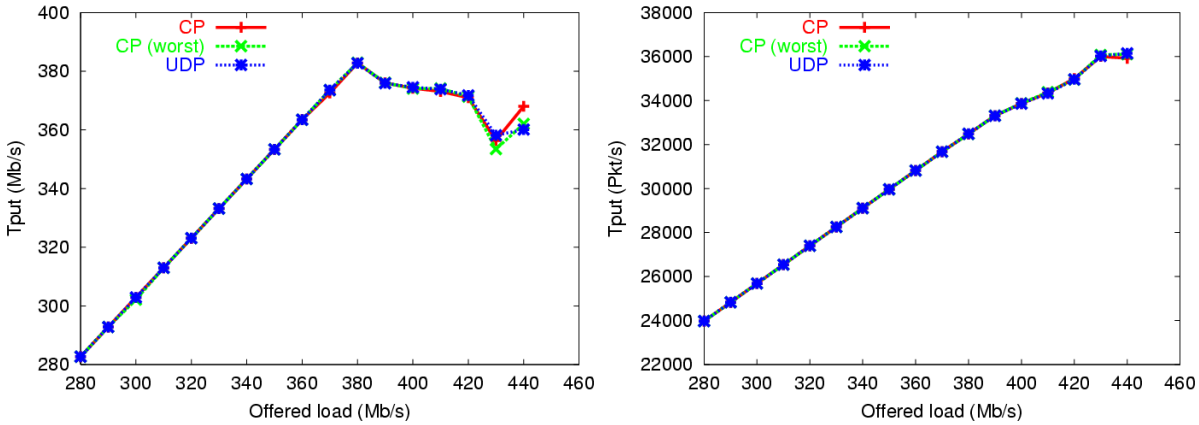


Figure 5.11: AP2 outbound forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s.

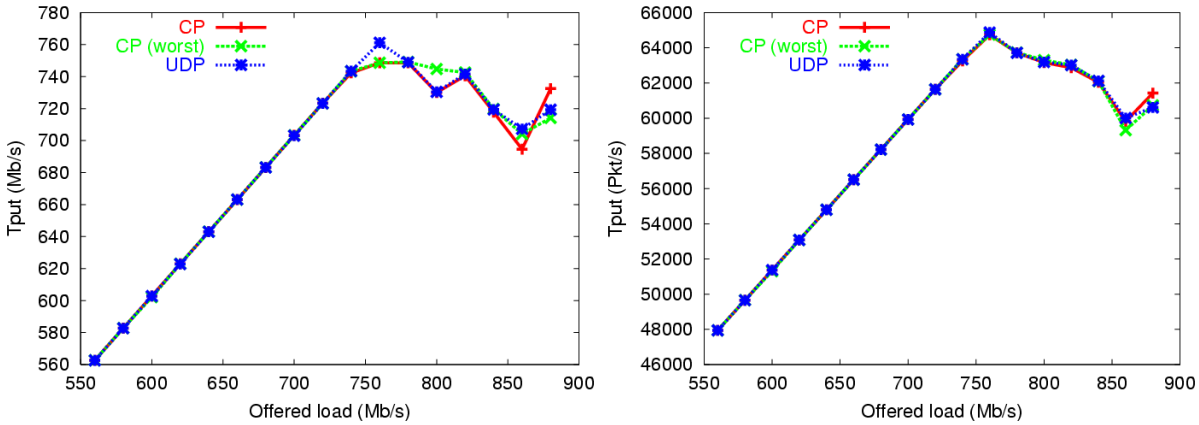


Figure 5.12: AP2 combined forwarding performance. Offered load versus throughput in (a) Mb/s and (b) Pkt/s.

Chapter 6

Coordinated Multi-streaming for 3D Tele-immersion

In this chapter, we examine how CP may be applied to a real-world application in order to increase communication efficiency and overall application performance. The problem to be solved is that of *video multi-streaming*, or the concurrent streaming of video frames by multiple application flows sharing the same cluster-to-cluster forwarding path. What makes this problem especially challenging are several stringent application requirements: reliable delivery, minimal buffering, and responsiveness to network congestion.

First, we describe the application, *3D Tele-immersion* or *3DTI*, and the general problem of multi-streaming. Central to this problem are the notions of a *frame ensemble* and *arrival asynchrony*. We then describe the role originally played by TCP in this application, and how CP can be used instead to both apportion bandwidth among streams in a coordinated way and to size send buffers dynamically in accordance with changing network path characteristics.

After describing our implementation briefly, we present results from our laboratory testbed, described in some detail in Appendix B. We divide these results into two sets: *equal frame size* and *unequal frame size*. As discussed in Section 6.5, the CP coordination algorithm for apportioning bandwidth within each set is somewhat different. A number of experimental configurations is then explored, including various network configurations (loss level, round trip time, background traffic load) and application configurations (number of hosts, frame size, frame size dispersion, and dynamic re-configuration).

We then present proof-of-concept results taken from the Abilene backbone network [Abi]. Multi-streaming in these experiments took place between the University of Pennsylvania in Philadelphia and the University of North Carolina at Chapel Hill over Abilene. Results test the effectiveness of our approach in an uncontrolled environment for bandwidth levels and host numbers that have been scaled up significantly.

Overall, our experiments demonstrate the performance improvement possible by using the Coordination Protocol. In particular, CP may be used in the multi-streaming context to simultaneously improve arrival synchrony, minimize network delay, and maintain throughput levels that are fair to competing TCP connections yet fully utilized. In contrast, multi-streaming using TCP results in a set of undesirable tradeoffs that depend upon send buffer size configuration. We show that no single configuration can match the performance of CP for this problem scenario.

6.1 3D Tele-immersion (3DTI)

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)* [TKS+03, KZM+03] system, 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. This scene brings a remote location (and its participants) to the user, creating a seamless continuum between the user's experience of local and remote space within the application.

3DTI[1] was jointly developed by the University of North Carolina at Chapel Hill and the University of Pennsylvania and provides an ideal environment for studying cluster-to-cluster data transport dynamics. The application is comprised of two multi-host environments, a *scene acquisition* subsystem and a *reconstruction/rendering* subsystem, that must exchange data in complex ways over a common Internet path.

The scene acquisition subsystem in 3DTI (see Figure 6.2) 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

[1] 3DTI is a sub-project of *Office of the Future* [RWC+98] that was described in Section 1.1. It focuses more narrowly on 3D reconstruction issues.

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 (using a shared 1394 Firewire bus) or network-based (using message passing).

The current version of 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.

The reconstruction/rendering subsystem in 3DTI represents essentially a cluster of data consumers. Using distributed processing, video frames taken from the same instant in time are compared with one another using complex pixel correspondence algorithms. The results, along with camera calibration information, are used to reconstruct depth information on a per pixel basis which is then assembled into view-independent *depth streams*. Information on user head position and orientation (obtained through head tracking) are then used to render these depth streams in real time as a view-dependent scene in 3D using a stereoscopic display.

Figure 6.1: 3D Tele-immersion.

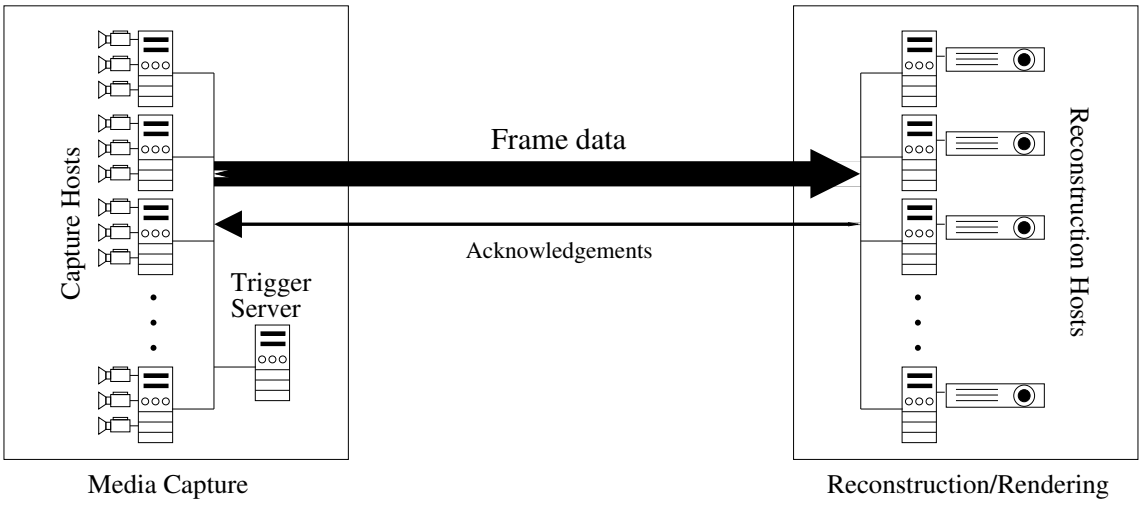


Figure 6.2: 3D Tele-immersion architecture.

6.2 The Problem of Multi-streaming

Our concern within 3DTI is with the problem of video multi-streaming between scene acquisition and 3D reconstruction subsystems. Specifically, we are interested in providing parallel streaming services for *frame ensembles* between computing clusters. Here we define a frame ensemble as a set of n video frames captured from n cameras at the same instant in time. Each frame will be streamed independently by a different source and destination host in 3DTI, although more complex configurations are possible.

What makes multi-streaming in 3DTI especially demanding, are several additional requirements. These include:

- **Reliable frame delivery.** The 3D reconstruction subsystem is unable to apply pixel correspondence algorithms to extract depth information if video frame data is incomplete. Thus, it is a basic application requirement that multi-streaming provide reliable data delivery.

- **Frame ensemble completeness.** In order for 3D reconstruction to proceed, all frames from the same ensemble must be present in their entirety. Once again, this is because pixels from different frames cannot be compared without a full set of source data. It is also because data and computation cannot be properly divided and distributed among reconstruction hosts without all parts being present.

- **Minimal end-to-end delay.** 3DTI is a real-time, interactive application. As

such, minimal end-to-end communication delay is a fundamental requirement. One implication of this requirement is that asynchrony among streams cannot simply be corrected with additional buffering at the receiver.

Congestion responsiveness is also an important requirement. In part, this is because 3DTI places such importance on reliable delivery and minimal end-to-end delay. Without mechanisms to adapt streaming rates to changing network conditions, queue build-up at bottleneck routers along the cluster-to-cluster data path is likely to occur. Increasing queue sizes means longer forwarding latency, packet loss as bursts in queue arrivals cannot be accommodated, and retransmissions. In addition, we wish to provide application streaming services that are fair to competing flows in the network and, by design, prevent the possibility of congestion collapse [FF99].

Both reliable transport and frame ensemble completeness requirements imply the need for *frame arrival synchrony*. This is the notion that frames within the same ensemble are received by hosts within the reconstruction subsystem at the same time. A low degree of frame synchrony will result in *stalling* as some flows within the application wait for other flows to finish. In general, frame arrival asynchrony slows the 3D reconstruction pipeline because receivers must wait for frame ensemble data to arrive in its entirety before parallel reconstruction can begin. Stalling furthermore results in low network utilization as some flows are forced to stop sending while they wait for the next capture trigger.

6.3 Multi-streaming with TCP

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 above, is an important requirement in the 3DTI multi-streaming problem domain. Second, it is congestion responsive. Use of TCP for multi-streaming in 3DTI insures that cluster-to-cluster 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 (e.g., 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

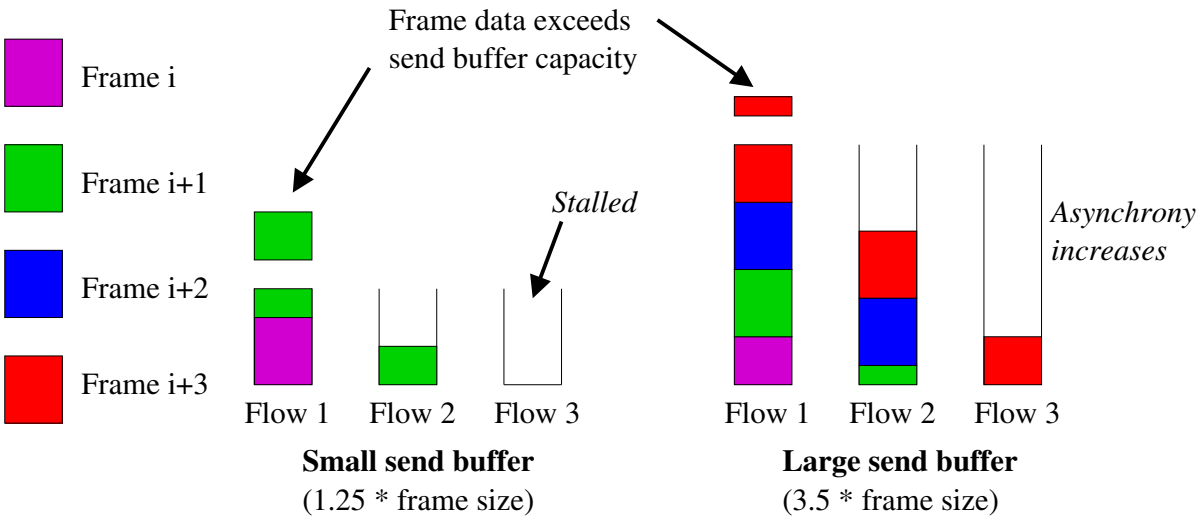


Figure 6.3: The effect of different send buffer sizes.

flows between the acquisition and reconstruction clusters resulted in large end-to-end latencies and asynchronous delivery of frames by different flows. Such inefficiency significantly slowed the capture-reconstruction pipeline and often interrupted normal operation of the application. (Our results later in this chapter will illustrate this.)

The problem with TCP in the multi-streaming context of 3DTI is the inherent *lack of coordination among flows*. That is, individual flows operate independently of peer flows within the same application. Each TCP flow independently detects congestion and responds to loss events using its well-know algorithm 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 few or no congestion events and maintain a congestion window that is, on average during the streaming interval, larger. For equal size frames, the result is that a subset of flows end up streaming frames belonging to the same ensemble more quickly and *at the expense* of peer flows that gave up bandwidth in the process.

The problem is even more extreme when video frames are of unequal size. Frames within an ensemble may differ in size after compression has been applied, or when different resolutions are used to favor camera angles close to an end user's field of interest. 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 an even higher probability of stalling as flows with less data finish first and wait as

flows with more data continue to send.

The problem of stalling can be mitigated, of course, by increasing send buffering. This approach, is, however, undesirable for two reasons. First, it increases end-to-end delay as additional buffer wait time is added to the transport pipeline. This is a highly undesirable result given that 3DTI is an interactive, real-time application. Second, it increases the potential for frame arrival asynchrony. To see this, consider Figure 6.3 which illustrates the effect of using small and large send buffer sizes. Small buffer sizes may result in frequent stall events as one flow waits without data for others to finish. This is seen as flow 3 waits for flows 1 and 2 in a buffer that is 1.25 times current frame size. Increasing buffer size to 3.5 times current frame size lessons the chance that any one flow has no data to send. However, the amount of frame asynchrony may increase dramatically. This can be seen in flow 3 which now streams frame i+3 while flow 1 streams frame i.

What is needed, we argue, is both *coordinated congestion response* and *network responsive send buffering*. The former prevents flows from seeing different congestion events and manages bandwidth distribution among flows in the application. The latter ensures that the amount of send buffering is adequate to maintain a full data pipeline at all times, but small enough to minimize unnecessary end-to-end delay. Since available bandwidth changes dynamically, it would furthermore be advantageous if such buffering could be adjusted dynamically to suit changing network conditions.

6.4 Multi-streaming with CP-RUDP

To address the problems described in the previous section, we turned to the Coordination Protocol. Using CP first required that we deploy CP-enabled software routers in front of capture and reconstruction clusters to act as APs. Details on hardware and software specifications of these routers were presented in Chapter 5. Essentially, each router is a high-end Intel-based machine running FreeBSD 4.9 with a special kernel module. This module extends IP forwarding capabilities to include all of the mechanisms described in Chapter 3 and Chapter 4.

Next, we developed a new CP-based protocol called *CP-RUDP* and deployed it on each endpoint host in the application. CP-RUDP is an application-level transport protocol that provides TCP-like reliable, in-order delivery semantics, but bases all send rate adjustments on information provided by CP. As its name suggests, CP-RUDP

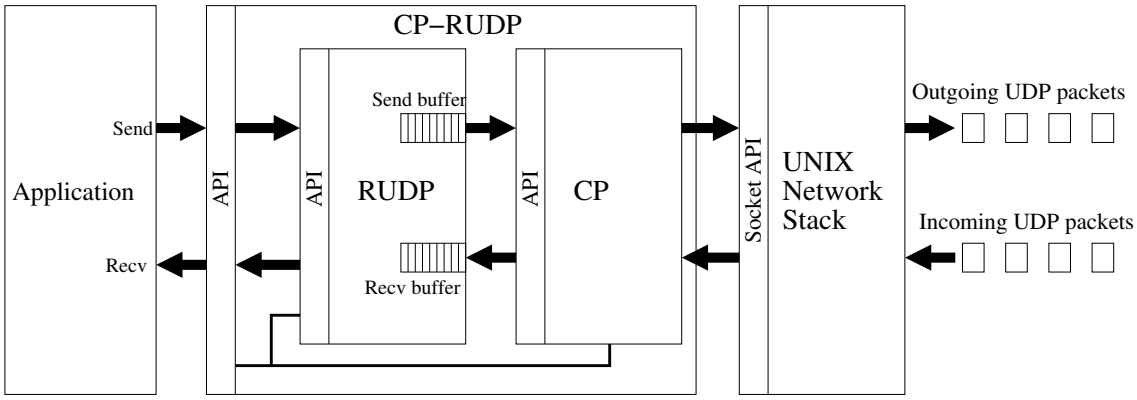


Figure 6.4: Diagram of CP-RUDP internals.

is a UDP-based protocol implemented at the application level. While this sacrifices some efficiency compared to a stack-based, kernel implementation, it is more easily deployable as end hosts do not require any operating system modifications.

At the core of CP-RUDP is *Reliable-UDP (RUDP)* [2], a connection-oriented transport protocol that provides reliable, in-order delivery semantics using UDP. RUDP manages UDP sockets much like standard TCP sockets in UNIX. This is because of RUDP's connection-oriented design which generally mimics that of TCP. A receiver will *bind* their socket to an address, *listen* for incoming connections, and then *accept* one when it arrives. Meanwhile, the sender will *connect* with a remote receiver before initiating a series of *send* calls, etc.

RUDP is implemented as a multi-threaded, UDP-based protocol that makes use of additional packet headers nested within the first 9-13 bytes of UDP data. Packet types include SYN, SYN-ACK, DATA, ACK, and FIN. Connection setup uses TCP's familiar three-way handshake packet exchange sequence, and connection termination involves a bi-directional FIN-ACK exchange. Stream-oriented data transport takes place using DATA packets that are acknowledged by the receiver using ACK packets. Both sequence numbers and packet numbers are used to track data and lost packets, and retransmissions are used to fill in gaps created by packet loss. (See Appendix C for a more complete description of RUDP.)

While RUDP provides reliable, in-order data delivery, it is entirely devoid of congestion control mechanisms. To provide this functionality, another layer of indirection is used between RUDP and the UNIX socket layer. As illustrated in Figure 6.4, this is

[2]Thanks to Travis Sparks and Ketan Mayer-Patel for their work on RUDP design and version 1.0 implementation.

the *CP layer* which provides rate control using information provided by CP. While the RUDP layer decides *what* to send, it is CP that decides *when* to send it.

CP is implemented as a library of functions that encapsulate RUDP packet data and headers into CP packets with an additional CP header. Using the mechanisms described in Chapter 3, CP will request and receive information from its local AP via these headers. In particular, it will receive information on the current bandwidth estimation for a single flowshare ($NET.bw$), the current round trip time ($NET.rtt$), the number of flows in the application ($FLOW.num$), and, for some configurations, information on the frame size streamed by peer flows in the application.

CP uses this information to perform two functions:

- Adjust the current sending rate to match conditions on the cluster-to-cluster data path, and
- Adjust RUDP's send buffer size to maintain a full transmission pipe at all times but without unnecessary buffering.

To implement the former, CP hints on available bandwidth are given to a rate-based packet scheduling routine that uses clock cycle time to track send intervals and schedule data transmissions. The latter is accomplished using a statically sized send buffer and a *threshold* value. The threshold value gives a maximum size value beyond which no new data will be accepted into the queue. Data that cannot fit into the send queue will be held in user space as the application blocks until space becomes available.

Both RUDP and CP operate as components in the CP-based transport protocol CP-RUDP. CP-RUDP's API looks much like the standard socket API of TCP seen on most UNIX systems (*socket(), bind(), listen(), connect(), close(), send(), receive(), setsockopt(), getsockopt()*, etc.), reflecting RUDP's connection-oriented approach to reliable, in-order transport. Additionally, functions may be added (e.g., *cprudp_get_rtt_est()*) that provide CP information directly to the application layer to drive application-level adaptation as needed. (See Appendix E for a list of currently supported CP-RUDP functions.)

CP-RUDP ultimately passes data to the UNIX socket API in the form of write calls sized so that they will not exceed the payload portion of a single UDP packet. The UNIX socket layer will, of course, add a UDP header (and subsequently an IP header) before transmitting the packet over the network. Incoming packets are delivered to CP-RUDP as single UDP datagrams using the UNIX socket API. Thus, extracting data

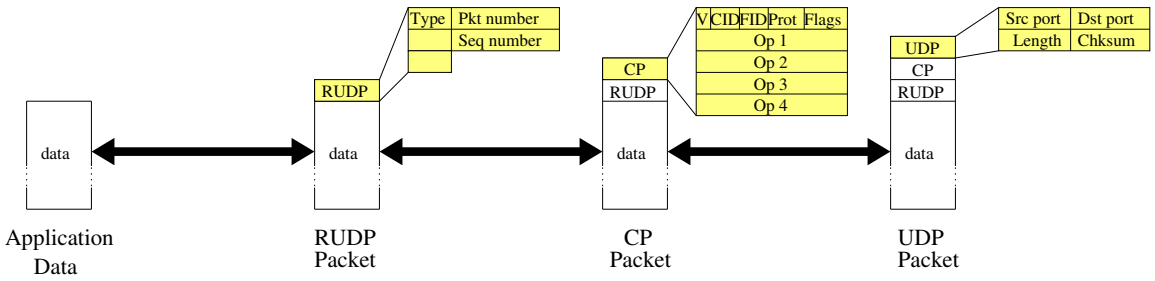


Figure 6.5: Chain of encapsulation in CP-RUDP.

and headers nested within a UDP data payload is an easy operation. (See Figure 6.4 and Figure 6.5 for illustrations.)

6.5 Coordination Schemes for CP Multi-streaming

In this section, we describe how 3DTI endpoints may be configured to use CP information to solve the coordination problem described in Section 6.3. In general, algorithms that make use of CP information to coordinate the sending behavior of multiple application flows are referred to in this dissertation as *coordination schemes*. Section 3.3.2 introduced the notion of a coordination scheme and provided several examples to illustrate.

The problem of flow coordination in 3DTI multi-streaming manifests itself in the issue of frame arrival asynchrony. As described in Section 6.3, uncoordinated flows using TCP may "see" congestion events on the shared data path somewhat differently, leading to adaptive behavior that varies across flows. (Recall Figure 1.3 and Figure 1.4 in Section 1.2.1.) This variation leads to variation in frame transfer time across flows for frames belonging to the same frame ensemble.

In fact, this problem has two cases: *equal frame size* ensembles and *unequal frame size* ensembles. In the former, all frames in the ensemble have exactly the same size. This may occur when each endpoint captures a frame with exactly the same resolution, using the same encoding scheme, and avoiding the use of compression algorithms that achieve different results depending upon image content. This is the default mode currently used by 3DTI.

To solve the flow coordination problem in the equal frame size case, we merely need to distribute bandwidth evenly across all flows participating in the multi-streaming. To accomplish this, we can rely on an inherent property of the Coordination Proto-

col architecture: *consistency of information across endpoints.* In the CP architecture, APs measure the changing properties of the cluster-to-cluster data path for the application as a whole and make a single estimate of the available bandwidth for a given cluster. Resulting values (round trip time, packet loss rate, available bandwidth, etc.) are propagated to *all* cluster endpoints using the state table mechanisms described in Chapter 3. Uniform values across all flows is ensured by commonality of source.

To distribute bandwidth evenly among all flows in 3DTI, then, we merely need to configure each endpoint to send at exactly the estimated bandwidth available to a single flowshare. CP-RUDP can obtain this value by requesting from its local AP a CP report from the $NET.bw$ cell in its state table. As updates are received, CP-RUDP on each endpoint can continually adjust its sending rate to conform to this value confident that peer flows in the same application are doing the same.

The situation for unequal frame sizes is somewhat more complicated. Here, our goal is once again the simultaneous arrival of frames within the same frame ensemble. However, due to different capture resolutions or the effects of data compression on images that vary somewhat in content, frame size is not uniform across flows. To coordinate multi-streaming in this case, we need to distribute bandwidth available to the application across flows *in proportion to the amount of frame data* that each has to send.

To do this, each endpoint first requires information on the bandwidth available to the application as a whole. CP-RUDP, running on each endpoint, can obtain this value by requesting the values $NET.bw$ (the estimated bandwidth available for a single flowshare) and $FLOW.num$ (the number of application flows) from its local AP. The bandwidth available to the application as a whole is then computed as the product of these two values.

To get the fraction of bandwidth available to an individual flow, each endpoint writes its frame size to a general purpose address that we will refer to as $FSIZE$. Recall that each cluster endpoint has a unique flow id, fid, that is used to designate the offset available for writing. $FSIZE.fid$ is thus the cell uniquely written by each endpoint. At the same time, an endpoint will request both $FSIZE.fid$ and $FSIZE.sum$ values from the local AP, where $FSIZE.sum$ represents the sum of all flow offsets in the $FSIZE$ address. That is, it is the total amount of frame data to be streamed in that ensemble. Requesting both $FSIZE.fid$ and $FSIZE.sum$ helps to insure that values are associated with the same frame ensemble since some lag exists before a new frame

size value can be assigned to $FSIZE.fid$.

Using $FSIZE.fid$ and $FSIZE.sum$, each endpoint can calculate a fraction that represents the portion of total ensemble data to be streamed by that individual flow. This fraction can then be multiplied by the total bandwidth available to the application as a whole to determine X, the flow-specific sending rate:

$$X = \frac{FSIZE.fid}{FSIZE.sum} * (NET.bw * FLOW.num) \tag{6.1}$$

The remaining issue is that of dynamic send buffer sizing. Recall from Section 6.3 that the issue here is how to maintain a buffer size that is simultaneously

- Large enough to maintain a full pipe (i.e., never fall short of its designated sending rate due to buffering limitations), yet
- Small enough to minimize the effect of buffer wait time on end-to-end streaming latency.

Since conditions on the cluster-to-cluster data path change over time, it would also be desirable if send buffer size could be adjusted *dynamically* to reflect changing network conditions.

Fortunately, application endpoints can easily solve the dynamic send buffer sizing problem using CP. Each endpoint already receives $NET.bw$ reports from the local AP as part of the flow coordination schemes above. Requesting the current round trip, $NET.rtt$, in addition would allow CP-RUDP to compute a bandwidth delay product (BDP) as $NET.bw * NET.rtt$. To accommodate additional room for retransmission data and data that is waiting to be acknowledged, we use an additional multiplicative factor of 1.5 as a heuristic. Thus, the instantaneous send buffer size, B, given current network conditions, can be calculated by each endpoint as:

$$B = 1.5 * (NET.bw * NET.rtt) \tag{6.2}$$

As described in Section 6.4, buffer size modifications can be implemented at each endpoint using a statically sized send buffer and a moving threshold value that regulates when new data can be admitted.

Experimental results in subsequent sections explore the effectiveness of these coordination schemes.

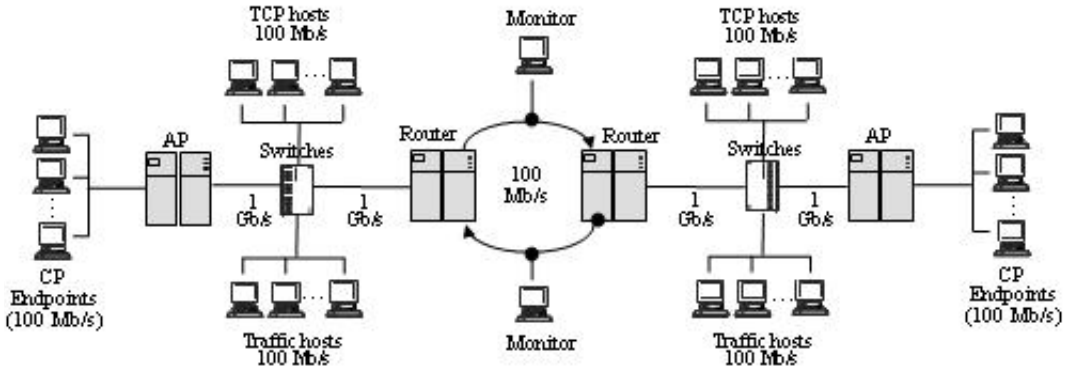


Figure 6.6: Experimental network setup.

6.6 Laboratory Testbed Experiments

In the next few sections, we present performance results taken from our laboratory testbed, described briefly here and more completely in Appendix B. Our overall goal is to compare the performance of 3DTI multi-streaming with and without the coordination mechanisms described in this chapter and, more generally, in this dissertation. In particular, we will demonstrate the effectiveness of our approach in improving frame arrival synchrony, reducing end-to-end delay, and maintaining throughput levels that utilize network bandwidth well yet are fair to competing TCP traffic.

6.6.1 Experimental Setup

Our experimental network setup is shown in Figure 6.6. (See Appendix B for a more complete description.) CP hosts and their local AP on each side of the network represent two clusters that are part of the same cluster-to-cluster application and exchange data with one another. Each endpoint sends data on a 100 Mb/s link to its local AP, a FreeBSD router that has been CP-enabled as described in Chapter 3 and Chapter 4. Aggregate cluster-to-cluster 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 6.6) and the well-known utility *iperf* [Ipe]. 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 6.6.2 then provides a way of quantifying CP's fairness to these competing flows.

Also sharing the bottleneck link for many experiments are background TCP 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 4.5.7 and in Appendix B.

Finally, network monitoring during experiments is done in two ways. First, *tcpdump* is used on monitor hosts to capture 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* [Ken98] 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.

Experiments in this section are run using the *frame* software utility [3] which emulates application behavior as described in Section 6.1. As with the actual 3DTI system, a trigger host is used to coordinate simultaneous frame capture across application endpoints in the media capture cluster. After each trigger, frame acquisition endpoints will write a frame to their send buffer (perhaps blocking for some time in the process) and then send a message to the trigger server indicating that they are ready for the next frame trigger. When a message has been received from all the endpoints, the trigger server initiates the next capture event. Receiving hosts in this architecture act simply as a data sink, though some instrumenting has been done to collect statistics on frame arrival performance.

"TCP" in our laboratory testbed refers to TCP Reno as implemented in FreeBSD 4.5 and RedHat LINUX 9 (kernel version 2.4). System configuration parameters relating to socket buffer size were increased to prevent buffer limitations from constraining overall congestion control behavior.

Multi-streaming using "CP" or "CP-RUDP" refers to the implementation described in Section 6.4 and the coordination schemes presented in Section 6.5. These schemes implement flow coordination by distributing bandwidth among endpoints in controlled ways and dynamically adjusting send buffer size based upon CP information.

[3] Thanks to Travis Sparks for his work on *frame* design and version 1.0 implementation.

Issue	Performance Metrics
Transport synchrony	*Completion asynchrony*, *Stall time*
Latency	*End-to-end delay*
Fairness/Network utilization	*Normalized flowshare*
Overall performance	*Frame ensemble (FE) rate*, *FE interarrival jitter*

Table 6.1: Multi-streaming performance issues and their corresponding metrics.

6.6.2 Performance Metrics

In this section, we define a number of metrics for measuring multi-streaming performance in 3DTI. Our performance discussion in this section will center around four issues: transport synchrony, latency, fairness and network utilization, and overall application performance. Associated with these issues are six key metrics: completion asynchrony, stall time, end-to-end delay, normalized flowshare, frame ensemble transfer rate, frame ensemble interarrival jitter. (See Table 6.1 for a summary.)

The level of *transport synchrony* in multi-streaming can be measured using two metrics. The first is *completion asynchrony* which quantifies how staggered frames within the same frame ensemble arrive at the receiving cluster. When frames within the ensemble are of equal size, completion asynchrony measures how evenly bandwidth was distributed among flows sharing the cluster-cluster data path. When frames are of unequal size, it measures how well flows have adapted their bandwidth usage to match the distribution of frame data in the ensemble.

Within any given frame ensemble i, there is some receiving host that receives frame i in its entirety first. Call this time of completion $c_{f,i}$. There is another host that receives frame i in its entirety last (i.e., after all other hosts have already received frame i). Call this 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. (See Figure 6.7.)

$$C_i = c_{l,i} - c_{f,i} \tag{6.3}$$

A second metric looking at transport synchrony in multi-streaming is *stall time*. The time interval between two frames within a given flow is typically small unless stalling occurs. A flow is said to *stall* when it completes its transmission of the current frame and must wait before sending a new one. Short waits may be caused by a flow's packet

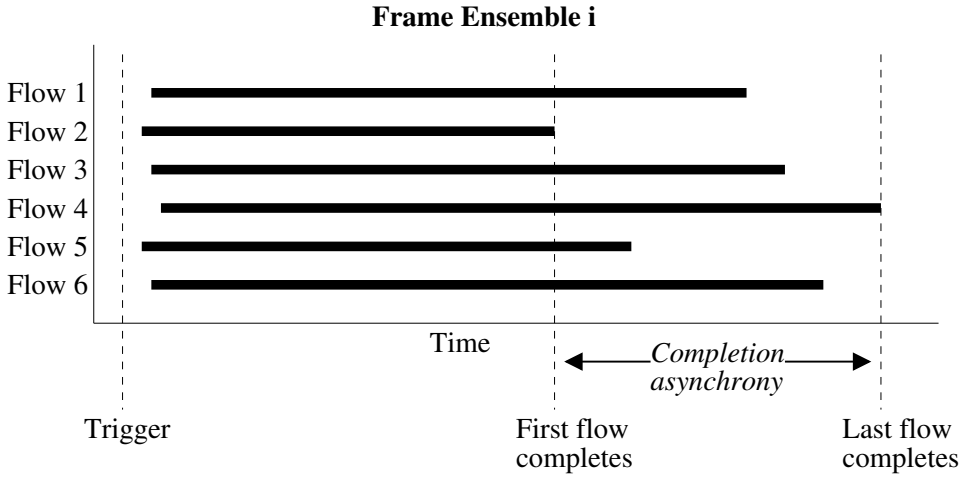


Figure 6.7: Completion asynchrony.

scheduling algorithm which regulates when packets from the new frame can be sent out over the network. More significant, however, are longer intervals that occur when a flow must wait for the next trigger event before streaming a new frame. Just how long it must wait depends upon how long it takes remaining flows to finish streaming their frame from the previous ensemble.

To capture this phenomenon, we simplify the notion of stall time to mean *the time interval between two frames within a single stream.* While this definition does not distinguish between routine packet spacing and interruptions in frame streaming due to wait events, it nevertheless is quite effective in quantifying the results of the latter when they occur. To see how stall time is calculated, we first note that each frame ensemble has a mean stall interval $s_{mean,i}$, measured simply as the average time across all flows between the completion of frame i and the beginning of frame $i+1$. *Stall time* is defined as the mean of the mean stall intervals s_{mean} for the entire run interval p.

A second performance issue is streaming latency. The transmission time for each frame, including send and receive buffering, is averaged across all flows into a mean delay value $d_{mean,i}$ for frame ensemble i. We define *end-to-end delay* to be the mean of all mean delay values d_{mean} for the run interval p. Delay values reflect a variety of factors including frame size, buffering at the sender, network queuing delay, and the number of retransmissions required to reliably transmit frame data in its entirety.

A third performance issue for multi-streaming in 3DTI is that of fairness to competing TCP traffic and network utilization. While seemingly two separate issues, here they are intricately related since aggregate multi-streaming traffic should take up to,

but not more than, its fair share of bandwidth. Taking too much bandwidth results in unfairness to competing TCP traffic sharing the same bottleneck link with application traffic. Taking too little bandwidth results in poor network utilization and hinders frame ensemble throughput generally.

To compare the bandwidth taken by flows in the application to that of TCP flows competing over the same bottleneck link, we 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.)

$$F_{CP-RUDP} = \frac{F_{CP-RUDP}}{F_{all}} \tag{6.4}$$

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.

A final issue is that of overall application performance. Here, two metrics are of interest. First is the *frame ensemble rate* which we define as the number of complete frame ensemble arrivals f over time interval p. In general, higher frame ensemble rate numbers indicate better network utilization and the absence of stalling due to frame transport asynchrony.

A similarly important metric is that of *frame ensemble interarrival jitter*, defined as the standard deviation of frame ensemble interarrival intervals t_i over a larger run interval p. Small jitter values are important to prevent the reconstruction/rendering pipeline from backing up or starving as the application runs in real-time.

It should be pointed out that frame ensemble rate and frame ensemble arrival jitter are, by themselves, inadequate for measuring the success of multi-streaming in 3DTI. This is because an application may do well in these areas but lack transport synchrony or have excessive end-to-end delay. The former, while not slowing the *transmission* pipeline, will greatly slow the *3D reconstruction* pipeline at the receiving cluster which cannot proceed without all frame ensemble data present. The latter undermines 3DTI's goal of being a real-time, interactive application. In fact, all of these metrics must be considered *simultaneously* to adequately gage the success of a multi-streaming scheme.

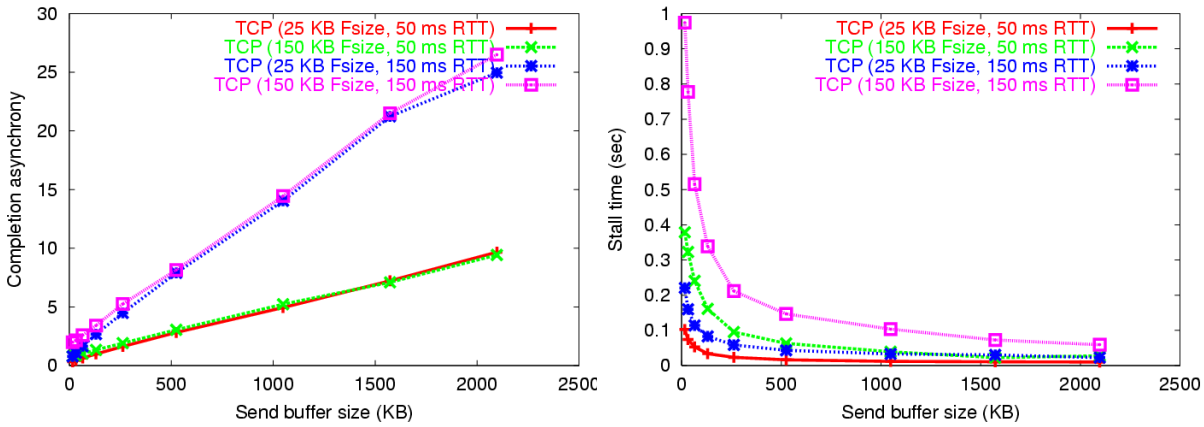


Figure 6.8: TCP send buffer size results. (a) Completion asynchrony and (b) stall time versus send buffer size.

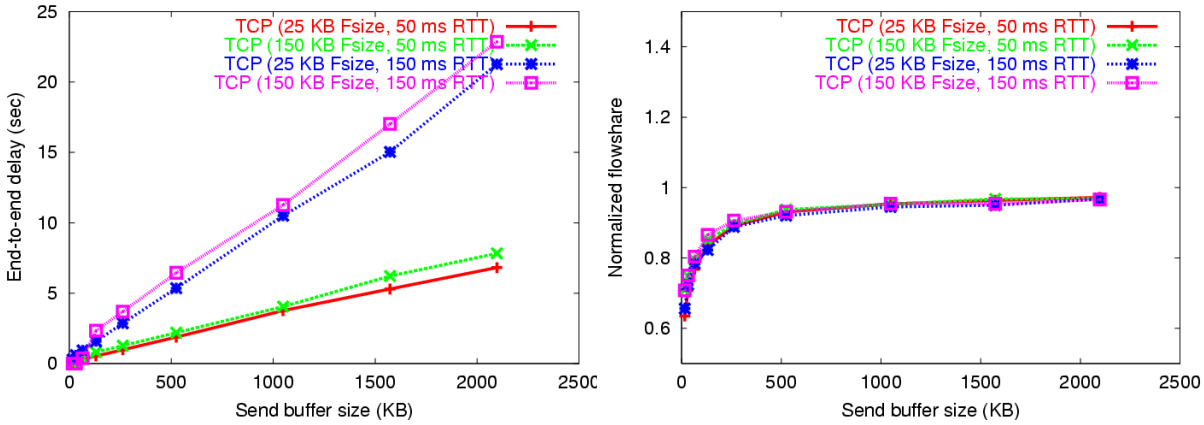


Figure 6.9: TCP send buffer size results. (a) End-to-end delay and (b) normalized flowshare versus send buffer size.

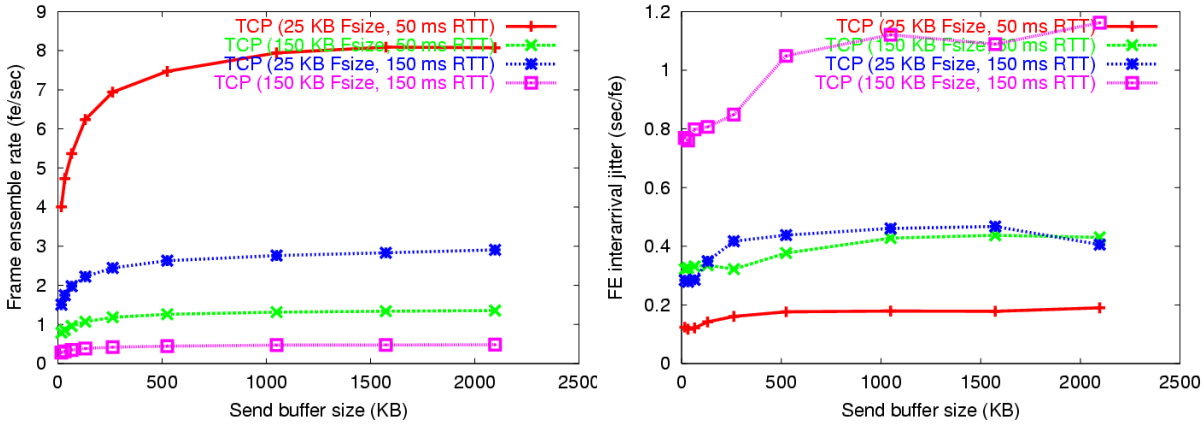


Figure 6.10: TCP send buffer size results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus send buffer size.

6.6.3 TCP Send Buffer Configuration

Experiments in this chapter will generally compare TCP, which 3DTI authors originally used for reliable multi-streaming, with CP-RUDP, our proposed scheme for reliable multi-streaming using the coordination mechanisms described in Chapter 3 and Chapter 4. Before doing so, however, we must first consider the issue of TCP send buffer size configuration.

TCP send buffer size can be manipulated using the *setsockopt()* system call in the UNIX socket API. The option name for this call is *SO_SNDBUF*. The minimum, maximum, and default values for this parameter are determined by kernel variables which vary in name across systems (FreeBSD, LINUX, Solaris, NetBSD, etc.). In this dissertation, maximum values have been increased significantly using the *sysctl* utility in order to allow for a wide range of values configured directly by the application.

To study the issue of TCP send buffer size, we ran four sets of experiments, each of which look at the effect of increasing send buffer size under a particular set of frame size and round trip time configurations. Send buffer size values were as follows: 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 1.5 MB, and 2 MB. Configurations included: (1) 50 ms round trip time and 25 KB frame size, (2) 50 ms round trip time and 150 KB frame size, (3) 150 ms round trip time and 25 KB frame size, and (4) 150 ms round trip time and 150 KB frame size. For each configuration, we ran three trials and then averaged the results to create a single data point for each metric. To avoid network interface contention within the same host, we ran only one stream on each available capture host for a total of six streams.

Bottleneck bandwidth was set to the default 100 Mb/s. *Dummynet* was used at each bottleneck router to emulate network characteristics like round trip time and network loss. In particular, *delay* was set to half the round trip time configuration (for a bi-directional sum equal to round trip time) and *packet loss rate* was set to .01. Experiments were given a 5-minute stabilization period after which data was collected for the next 10 minutes.

Figure 6.8, Figure 6.9, and Figure 6.10 show results for the metrics described in Section 6.6.2. In Figure 6.8 (a), we see that completion asynchrony increases linearly with send buffer size for all round trip and frame size configurations. While frame size seemed to make little difference, round trip determined the slope of the increase with 150 ms resulting in a larger slope than the 50 ms configuration.

Figure 6.8 (b) shows that stall time varies inversely with send buffer size. This

was true for all configurations, although different configurations showed different stall values for a given send buffer size. The most stalling occurred with the large frame size, large round trip time configuration. Next was large frame size, small round trip time. Next was small frame size, large round trip time. Finally, small frame size, small round trip time showed the lowest stall values. Differences in these configurations became less pronounced as send buffer sizes increased from 512 KB to 2.0 MB. Very small buffer sizes resulted in very large increases in stall time generally.

Figure 6.9 (a) shows the effect of send buffer size on end-to-end delay. As one might guess, delay values increase linearly with send buffer size. This is the effect of buffer wait time as frame data must pass through a large buffer before being transmitted over the network. Like completion asynchrony, larger round trip times resulted in a sharper slope and results appear to be insensitive to frame size.

Figure 6.9 (b) shows the results for normalized flowshare. It is interesting to note here that all configurations led to exactly the same results, demonstrating that only send buffer size was a critical factor. The results show that small buffer sizes result in under-utilization of network bandwidth. This can be seen in average values of .7 to .9 for small buffer size configurations. Only when buffer sizes were increased to 1 MB and beyond do values approach 1.0, indicating that flows received nearly their fair share of bandwidth.

Frame ensemble rates, given in Figure 6.10 (a), show the results for all configurations having the same shape as normalized flowshare results. This indicates that frame ensemble rates do indeed depend a great deal upon normalized flowshare values. However, each configuration resulted in different values. Rates were highest for the small frame size, small round trip time configuration. Next was small frame size, large round trip time. Next was large frame size, small round trip time. Finally, large frame size, large round trip time showed the lowest frame ensemble rates. This is not unexpected as clearly small frame size results in larger frame ensemble throughput, and small round trip times result in less asynchrony and stalling.

Finally, Figure 6.10 (b) shows the effect of send buffer size on frame interarrival jitter. In general, the large frame size, large round trip time configuration showed significantly more jitter than the other configurations. Similarly, the small frame size, small round trip time configuration showed significantly less jitter than the other configurations. The other two configurations both showed similar intermediate results. All configurations show somewhat more jitter as send buffer size increases, although the

effect quickly levels off once send buffer size achieves 512 KB.

After considering the results from each metric, the decision was made in this work to choose two TCP send buffer configurations for subsequent experimentation: 64 KB and 1 MB. Our reasoning was twofold. First, each configuration represents a polar opposite set of multi-streaming performance characteristics:

- **64 KB.** *Pros:* Low completion asynchrony, low end-to-end delay, somewhat lower frame ensemble interarrival jitter. *Cons:* Large stall times, low network utilization, low frame ensemble rate.
- **1 MB.** *Pros:* Small stall times, good network utilization, high frame ensemble rates. *Cons:* High completion asynchrony, large end-to-end delays, higher frame interarrival jitter.

By looking at the results from each extreme, we better understand the TCP configuration space than if we chose a single, middle-of-the-road value. Second, for any given set of results below, one can generally deduce the results for a median value simply by looking at the results for each extreme. When the extremes are divergent, the median value is generally in the middle. If the extremes coincide, then median values generally also coincide.

6.7 Laboratory Testbed Results: Equal Frame Size

In this section, we look at experimental results for equal frame size ensembles. Frames captured by each endpoint following a trigger in this scheme are identical in size. This will occur when all endpoints use exactly the same resolution, the same encoding scheme, and avoid applying compression algorithms. This is the default mode currently used by 3DTI.

Endpoints using CP-RUDP, as described in Section 6.5, are configured to send at exactly the rate given by $NET.bw$ in the AP state table. This is the congestion responsive send rate for a single flowshare, as estimated by each AP using the TFRC equation presented in Section 2.3.2. This scheme naturally results in an even distribution of available bandwidth among flows.

6.7.1 Round Trip Time

In this section we compare the performance of TCP and CP-RUDP under conditions of differing round trip time. To study this issue, we used the *dummynet* shaping utility on bottleneck routers to create the following round trip time values: 4, 10, 20, 50, 80, and 120 milliseconds. In addition, we ran two sets of results, one for a frame size of 25 KB and another for a frame size of 150 KB.

As with the TCP send buffer experiments, we used a total of six streams (one for each capture host available in our network), bottleneck bandwidth was configured to be 100 Mb/s, and *dummynet* packet loss rate was set to .01. Once again, we ran three trials and then averaged the results to create a single data point for each metric. Experiments were given a 5-minute stabilization period after which data was collected for the next 10 minutes.

Figure 6.11, Figure 6.12, and Figure 6.13 show the results for a frame size of 25 K, while Figure 6.14, Figure 6.15, and Figure 6.15 show results for a frame size of 150 K. Figure 6.11 (a) and Figure 6.14 (a) results show completion asynchrony increasing with round trip time for both TCP configurations. Values for TCP with large send buffer size increase much more sharply, however, than TCP with small send buffer size. CP-RUDP values increase somewhat as well, but remain significantly low throughout. In fact, they are less than the TCP small send buffer size in each and every configuration.

Figure 6.11 (b) and Figure 6.14 (b) results show CP-RUDP exhibiting minimal stalling for nearly all round trip time configurations, with a slight increase in stall time seen for the largest round trip time and the 25 K frame size. Meanwhile, both TCP configurations show significantly higher stall time numbers generally, and a marked increase in stall time as round trip time increases. This is most pronounced for the smaller send buffer configuration.

In Figure 6.12 (a) and Figure 6.15 (a), we see both TCP and CP-RUDP increasing their end-to-end delay values linearly as round trip time increases. But, a significant difference exists in both the amount of delay and the rate of delay increase. For the 25 KB frame size, TCP with a large send buffer configuration shows markedly higher delay values that increase sharply as round trip time increases. CP-RUDP, on the other hand, shows markedly low values that increase only slightly as round trip time increases. TCP with a small send buffer configuration is similar to CP-RUDP, but shows slightly higher values and slightly more increase. Interestingly, the situation between these two is swapped for the 150 KB frame size configuration, although the values remain very

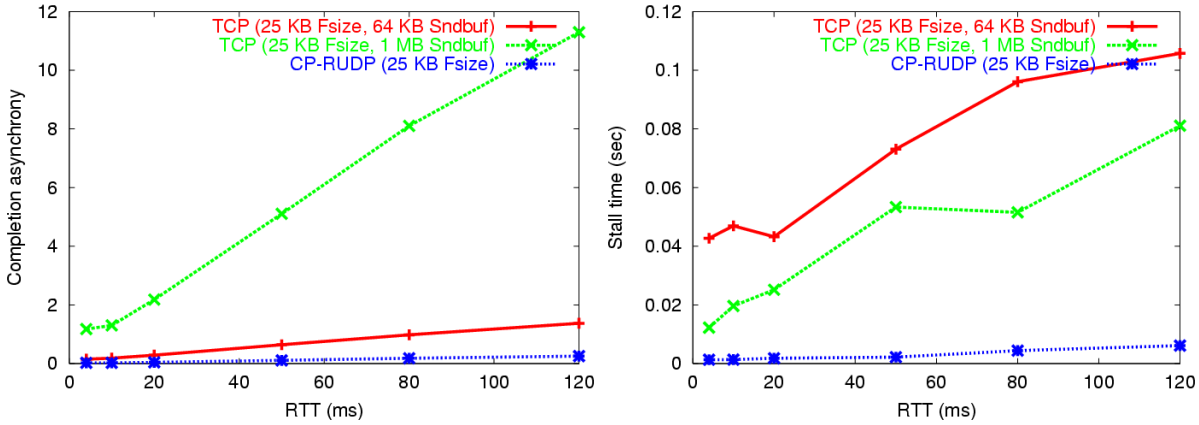


Figure 6.11: Round trip time results. (a) Completion asynchrony and (b) stall time versus round trip time.

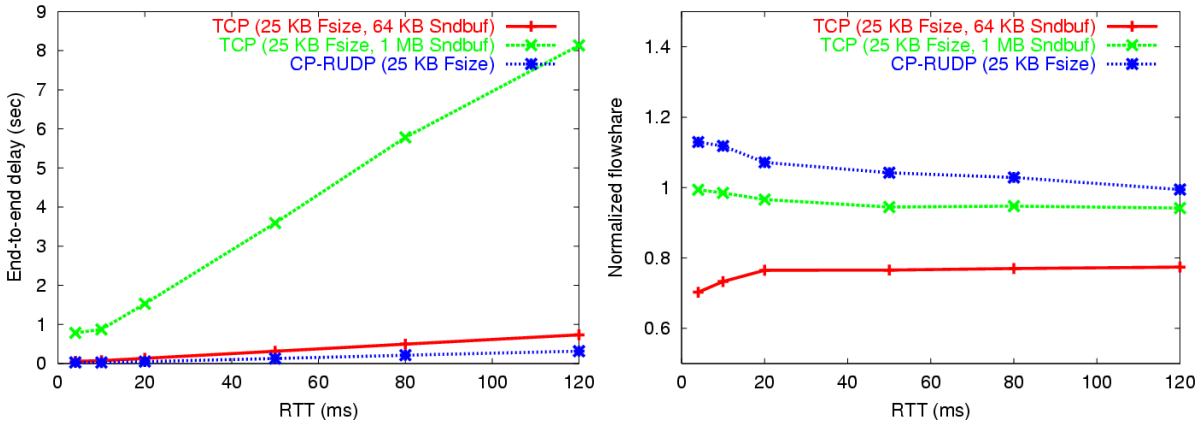


Figure 6.12: Round trip time results. (a) End-to-end delay and (b) normalized flowshare versus round trip time.

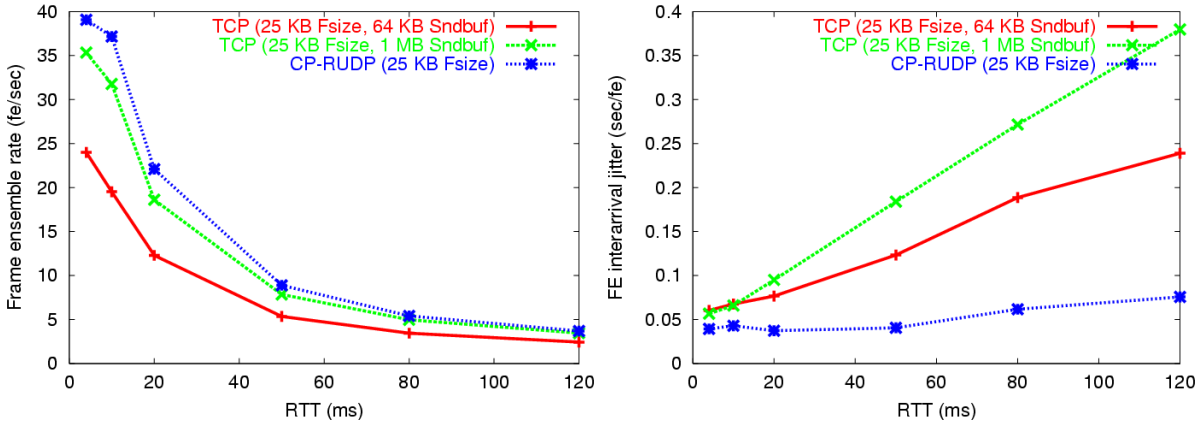


Figure 6.13: Round trip time results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus round trip time.

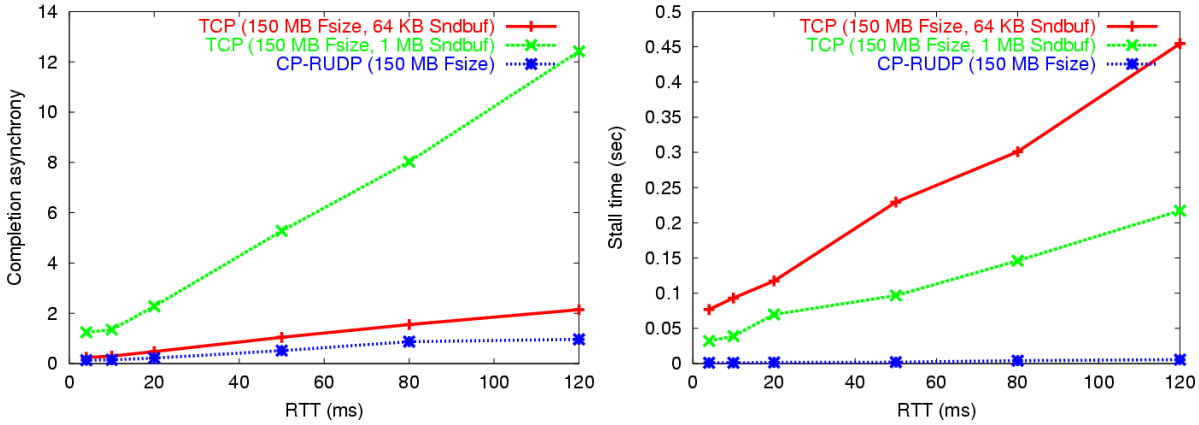


Figure 6.14: Round trip time results. (a) Completion asynchrony and (b) stall time versus round trip time.

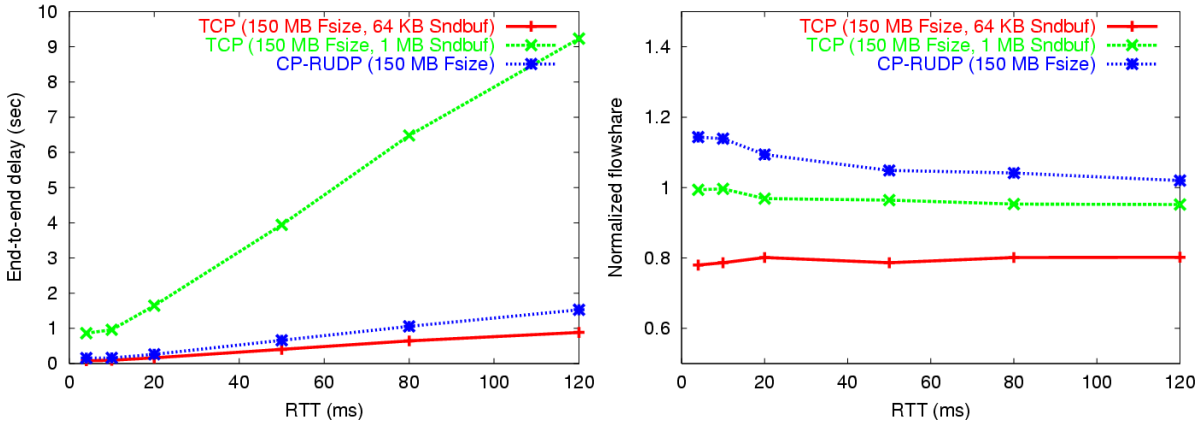


Figure 6.15: Round trip time results. (a) End-to-end delay and (b) normalized flowshare versus round trip time.

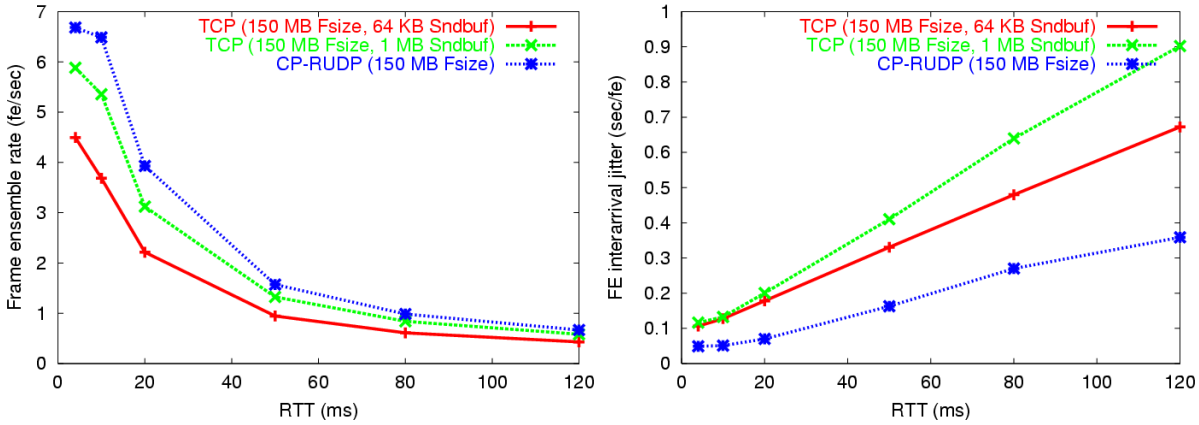


Figure 6.16: Round trip time results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus round trip time.

close.

Normalized flowshare results in Figure 6.12 (b) and Figure 6.15 (b) show TCP with a large buffer size configuration getting exactly its full share of bandwidth throughout all configurations of round trip time. (Values remain at 1.0 throughout.) Meanwhile, TCP with a small buffer size configuration gets less than its fair share, with values of approximately .8 throughout. Presumably, this is the result of significant stalling as seen in Figure 6.11 (b) and Figure 6.14 (b). CP-RUDP gets somewhat more than its fair share for very small round trip time values (4 and 10 ms), but values then remain between 1.1 and 1.0 as round trip time increases, with values approaching 1.0 as round trip time increases to 120 ms.

Frame ensemble rates in Figure 6.13 (a) and Figure 6.16 (a) show CP-RUDP receiving somewhat better rates than either TCP configuration for all round trip time settings. TCP with a large buffer size configuration does better than TCP with a small buffer size configuration. These results generally follow normalized flowshare results, but to some degree also reflect stall time results in Figure 6.11 (b) and Figure 6.14 (b). Recall that frame ensemble rate, while an important metric in examining overall multi-streaming performance, does not consider the issue of arrival synchrony or end-to-end latency.

Finally, Figure 6.13 (b) and Figure 6.16 (b) show frame ensemble interarrival jitter increasing as round trip time increases for all runs. However, both TCP configurations show more jitter and a greater increase in jitter than CP-RUDP. TCP with a large send buffer configuration scores highest in both of these areas.

6.7.2 Packet Loss Rate

In this section we compare the performance of TCP and CP-RUDP under conditions of differing packet loss rates. To study this issue, we once again used the *dummynet* shaping utility on bottleneck routers running FreeBSD. Our packet loss rate values were: .005, .01, .02, .03, and .04. In addition, we ran two sets of results, one for a frame size of 25 KB and another for a frame size of 150 KB.

As with the TCP send buffer experiments, we used a total of six streams (one for each capture host available in our network), bottleneck bandwidth was set to 100 Mb/s, and *dummynet* delay was set to 25 milliseconds (for a 50 ms total round trip time). Once again, we ran three trials and then averaged the results to create a single data point for each metric. Experiments were given a 5-minute stabilization period after

which data was collected for the next 10 minutes.

Figure 6.17, Figure 6.18, and Figure 6.19 show results for a frame size of 25 KB, and Figure 6.20, Figure 6.21, and Figure 6.22 show results for a frame size of 150 KB. In general, results are similar but with a few subtle differences. (Hence, both are presented here.) Figure 6.17 (a) and Figure 6.20 (a) show CP-RUDP exhibiting minimal completion asynchrony as packet loss rate increases. TCP with a small send buffer configuration exhibits somewhat more completion asynchrony. Both are relatively small, however, compared to TCP with a large send buffer configuration which exhibits a strikingly large amount of completion asynchrony that increases starkly as packet loss rates increase.

Figure 6.17 (b) and Figure 6.20 (b) results show CP-RUDP once again getting the lowest stall time values. These values increase slightly as packet loss rates increase, especially for the 25 KB frame size set. TCP with a large send buffer configuration shows somewhat higher stall time values. These values likewise increase somewhat with increasing packet loss rates. In contrast to both, however, is TCP with a small send buffer configuration which shows significantly larger values that increase significantly as packet loss rate increases. Interestingly, the rate of increase is sharper for the 150 KB frame size.

End-to-end delay values in Figure 6.18 (a) and Figure 6.21 (a) show CP-RUDP once again receiving the smallest delay values in the 25 KB frame size set. These values increase somewhat as packet loss rate increases. TCP with a small buffer configuration receives similarly small delay values, and in the 150 KB frame size set, the two are swapped. TCP with a large send buffer configuration shows strikingly higher values, and values that sharply increase as packet loss rates increase.

Figure 6.18 (b) and Figure 6.21 (b) show normalized flowshare values for both CP-RUDP and TCP that increase significantly as packet loss rate increases. For the 25 KB frame size, CP-RUDP values begin at approximately 1.1 and then increase to 1.33 as packet loss rates increase to .04. As discussed in Section 4.5.6, this is a known problem with the TFRC bandwidth estimation equation [Wid00]. It is enough for us to notice that values for .01 perform reasonably well, and hence are used extensively for the experimental work presented in this chapter.

Much more mysterious is why normalized flowshare values would increase for TCP as packet loss rates increase. For the 25 KB frame size, values for TCP with a small send buffer configuration begin at .78 and increase to about 1.13. Values for TCP with

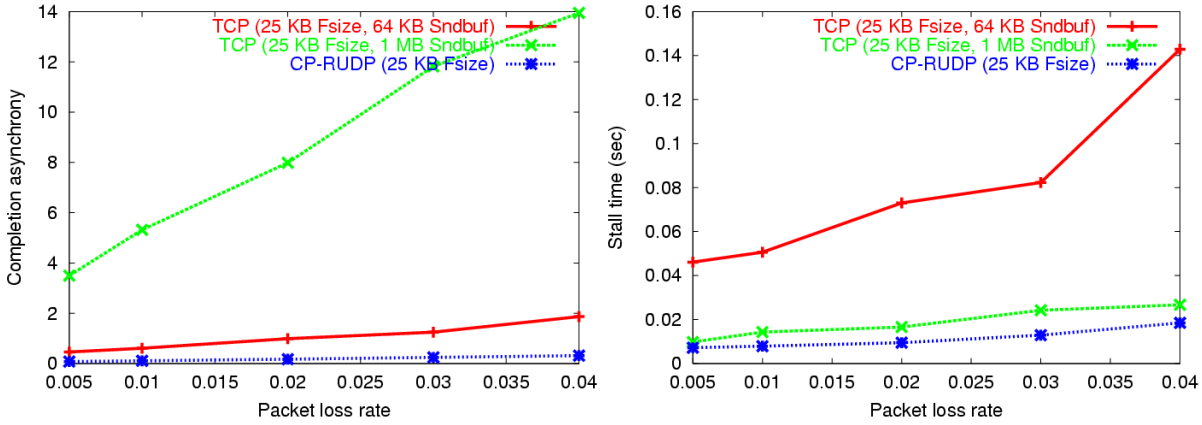


Figure 6.17: Packet loss rate results. (a) Completion asynchrony and (b) stall time versus packet loss rate.

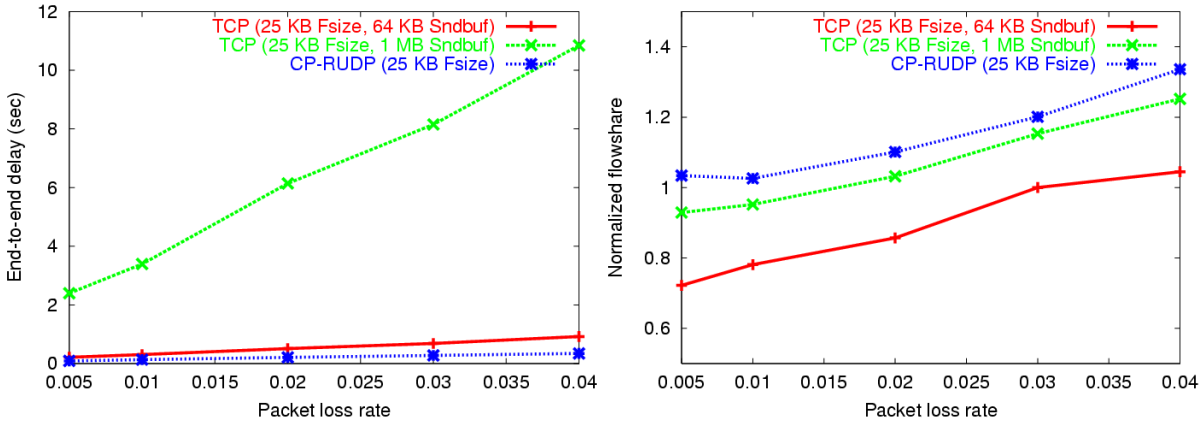


Figure 6.18: Packet loss rate results. (a) End-to-end delay and (b) normalized flowshare versus packet loss rate.

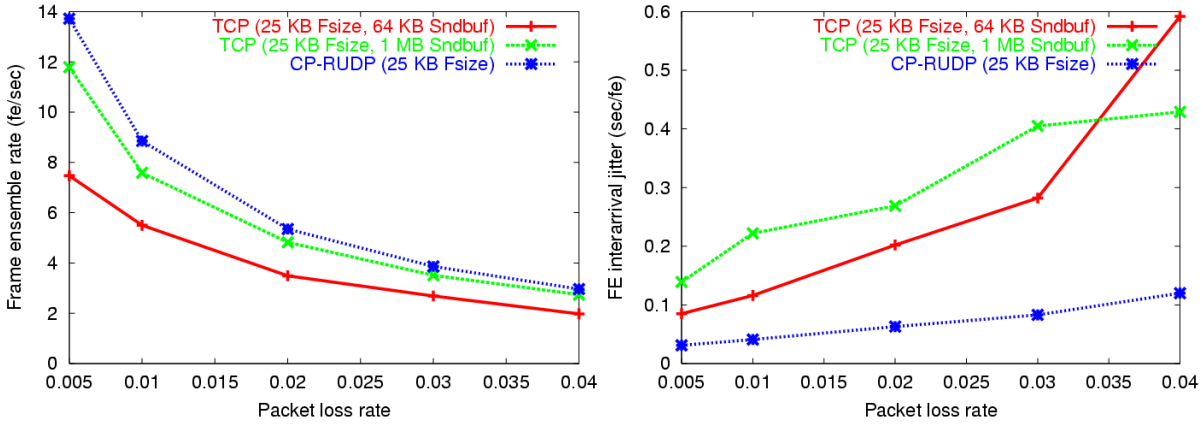


Figure 6.19: Packet loss rate results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus packet loss rate.

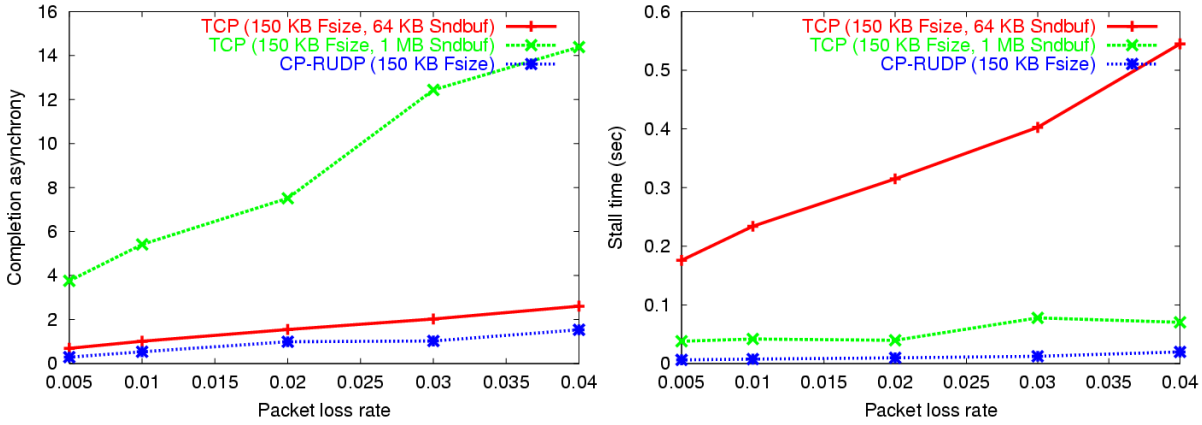


Figure 6.20: Packet loss rate results. (a) Completion asynchrony and (b) stall time versus packet loss rate.

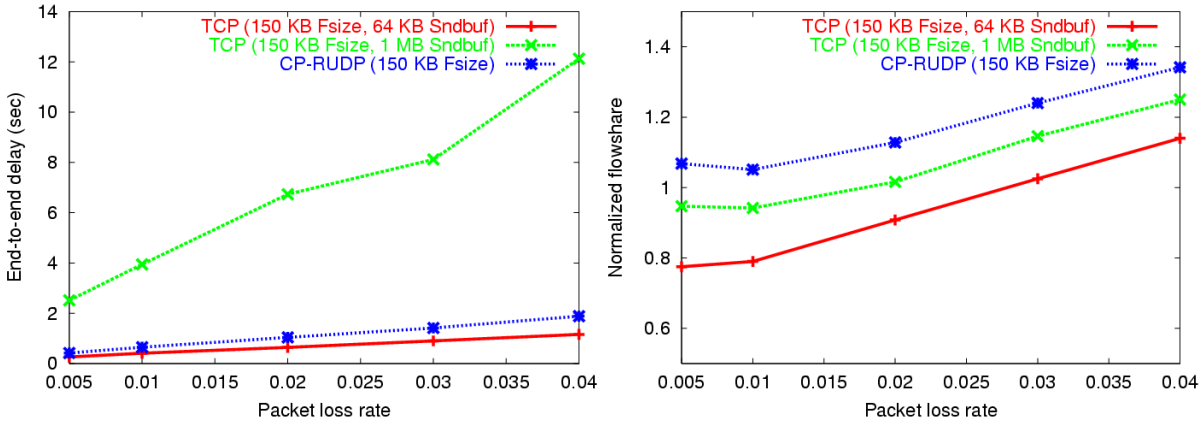


Figure 6.21: Packet loss rate results. (a) End-to-end delay and (b) normalized flowshare versus packet loss rate.

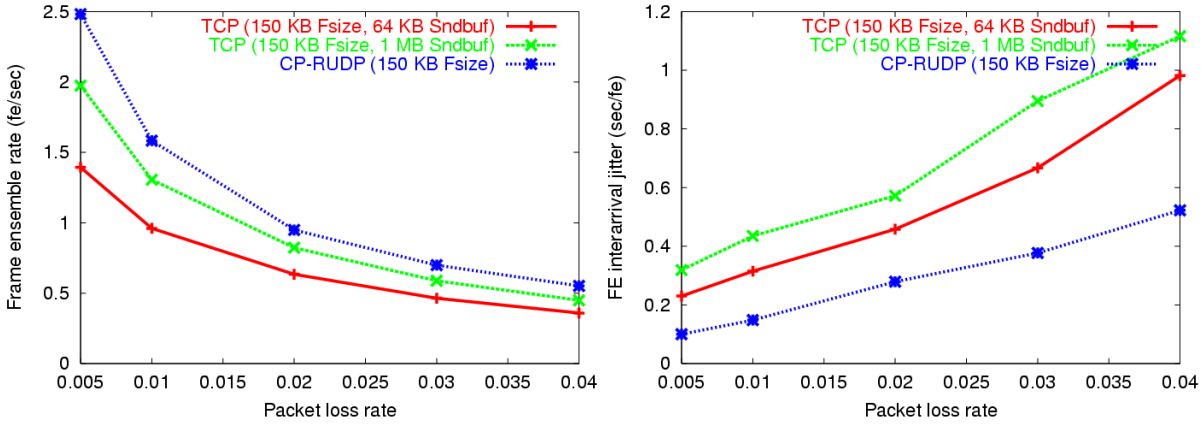


Figure 6.22: Packet loss rate results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus packet loss rate.

a large send buffer configuration begin at .95 and increase to about 1.26. It should be noted that frame senders in the network configuration shown in Figure 6.6 are Redhat LINUX 9 hosts paired with FreeBSD 4.5 receive hosts. In contrast, competing *iperf* flows that share the network bottleneck and are used for normalized flowshare calculations are generated by FreeBSD 4.5 pairs. One wonders whether differences in LINUX and FreeBSD TCP implementations become important in this context. [SK02], for example, describe a number of ways in which LINUX departs from more standard implementations.

Figure 6.19 (a) and Figure 6.22 (a) show both CP-RUDP and TCP dropping in frame ensemble rate as packet loss rates increase. Once again, values for CP-RUDP are higher for all configurations, following naturally from the fact that CP-RUDP receives somewhat more bandwidth and stalls much less than TCP. TCP with a small send buffer configuration shows lower frame ensemble rates than TCP with a large send buffer configuration.

Finally, Figure 6.19 (b) and Figure 6.22 (b) show frame ensemble interarrival jitter results. CP-RUDP is consistently lower than either TCP configuration, although values generally increase with packet loss values. Values also increase for both TCP configurations, but TCP with a large buffer configurations consistently shows the higher jitter values. The .04 packet loss level shows an interesting switch in results between the two TCP configurations for the 25 KB frame size.

6.7.3 Number of Streams

In this section, we compare the performance of TCP and CP-RUDP when the number of streams participating in multi-streaming varies. To study this issue, we used application configurations with 2, 3, 4, 5, and 6 streaming hosts. (The number of streaming hosts was constrained both by the number of CP hosts available in our laboratory testbed, and by the our wish to avoid the possibility of network interface contention by employing only one stream per host. See Section 6.10.1 for real-world results with a larger number of streams.) Once again, we ran two sets of results, one for a frame size of 25 KB and another for a frame size of 150 KB.

In keeping with our previous experiments, we used a bottleneck bandwidth of 100 Mb/s, a *dummynet* delay setting of 25 milliseconds (for a 50 ms total round trip time), and a *dummynet* loss setting of .01. Once again, we ran three trials and then averaged the results to create a single data point for each metric. Experiments were given a

5-minute stabilization period after which data was collected for the next 10 minutes.

Figure 6.23, Figure 6.24, and Figure 6.25 show results for a frame size of 25 KB, and Figure 6.26, Figure 6.27, and Figure 6.28 show results for a frame size of 150 KB. Overall, the number of streams had very little effect on CP-RUDP which shows nearly constant values for most metrics. In contrast, TCP values often varied with the number of streams somewhat, and sometimes in unpredictable ways.

Figure 6.23 (a) and Figure 6.26 (a) once again show CP-RUDP show exhibiting minimal completion asynchrony while TCP with a small send buffer configuration showed only slightly more. In stark contrast, TCP with a large buffer configuration showed much higher asynchrony numbers and a much sharper increase in values as the number of streams was increased.

Figure 6.23 (b) and Figure 6.26 (b) show CP-RUDP exhibiting negligible stall times for all configurations, while both TCP configurations received significantly higher values. TCP with a large send buffer configuration showed fairly constant values while TCP with a small send buffer configuration showed increasing values as the number of streams increased.

End-to-end delay results in Figure 6.24 (a) and Figure 6.27 (a) are somewhat surprising. CP-RUDP values for both configurations remain fairly low and constant as the number of streams increase. For both TCP values, however, values *decrease* linearly as the number of streams increase. It is unclear why this is the case. Once again, TCP with a large send buffer configuration exhibits far higher end-to-end delay values than TCP with a small send buffer configuration.

Normalized flowshare results are shown in Figure 6.24 (b) and Figure 6.27 (b). Results generally follow previous trends which show CP-RUDP receiving just slightly above its fair share level, TCP with a large send buffer configuration receiving very close to it, and TCP with a small send buffer configuration receiving strikingly less than its fair share. Values for CP-RUDP and TCP with a large send buffer configuration remain nearly constant as the number of streams increases for both frame size configurations. TCP with a small send buffer configuration, in contrast, shows numbers that drop. This follows from its increase in stall time as the number of streams is increased.

Figure 6.25 (a) and Figure 6.28 (a) show frame ensemble rate results. Once again, CP-RUDP shows the highest values due to strong utilization and few stall events. These values furthermore remain constant as the number of streams increases. TCP with a large send buffer configuration exhibits likewise constant values that are slightly less

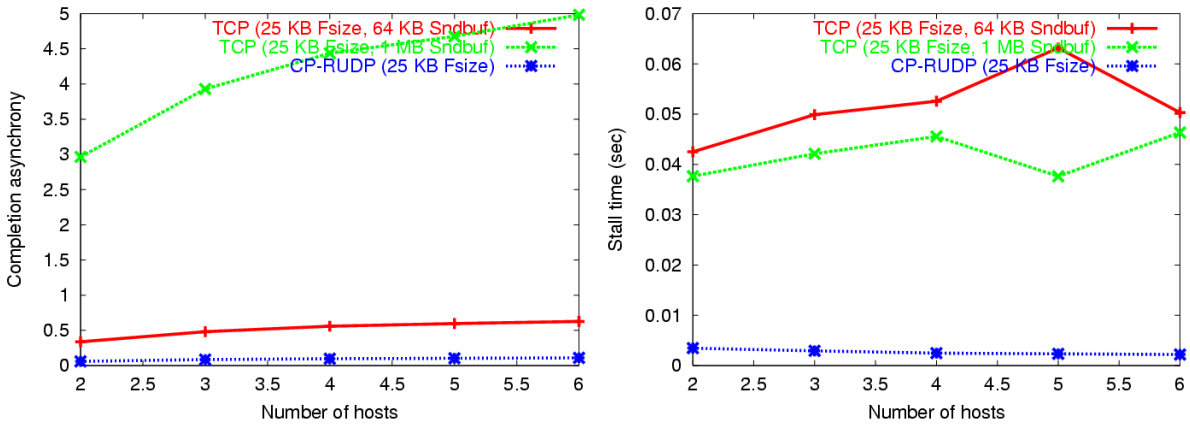


Figure 6.23: Number of streams results. (a) Completion asynchrony and (b) stall time versus number of streams.

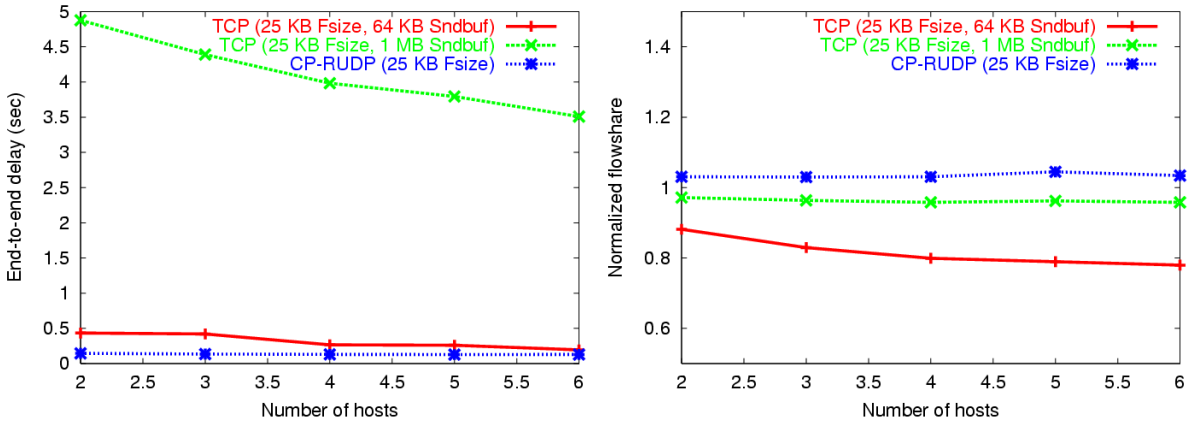


Figure 6.24: Number of streams results. (a) End-to-end delay and (b) normalized flowshare versus number of streams.

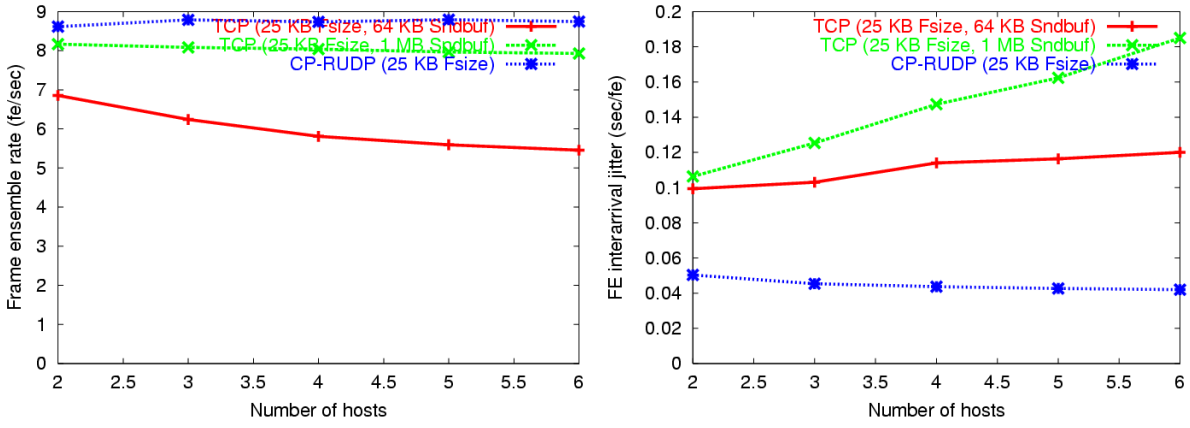


Figure 6.25: Number of streams results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus number of streams.

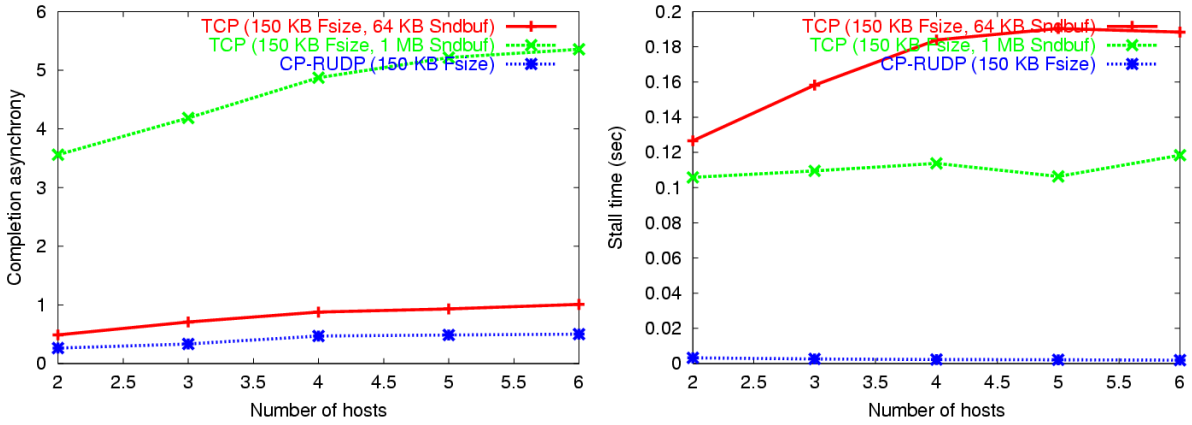


Figure 6.26: Number of streams results. (a) Completion asynchrony and (b) stall time versus number of streams.

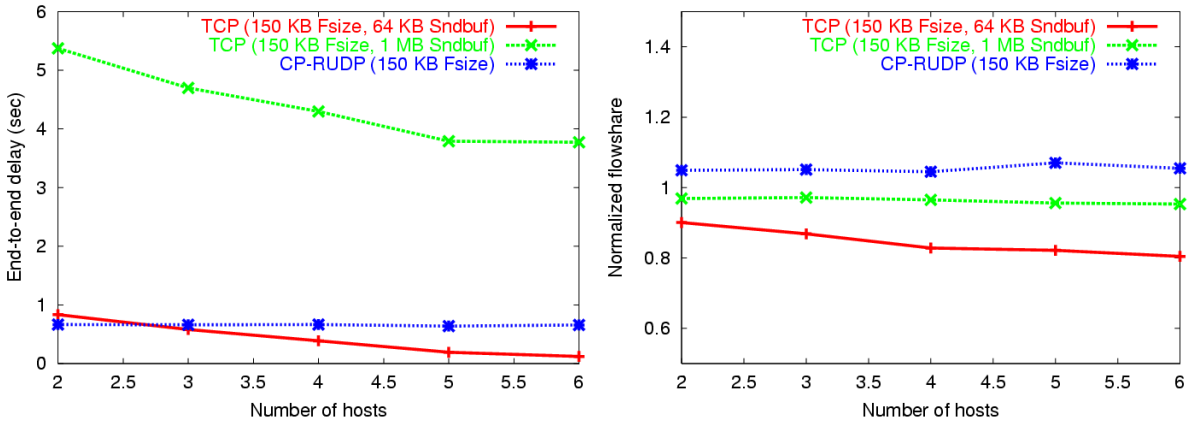


Figure 6.27: Number of streams results. (a) End-to-end delay and (b) normalized flowshare versus number of streams.

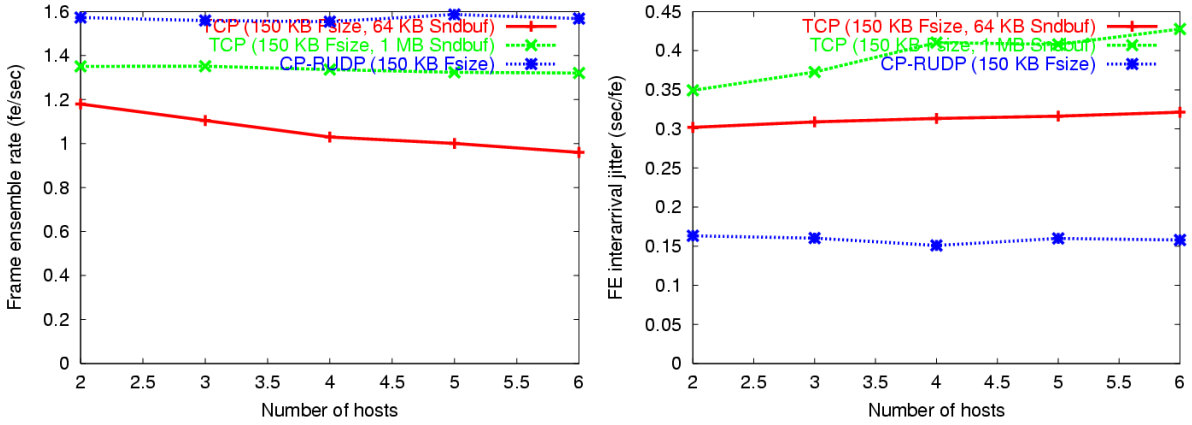


Figure 6.28: Number of streams results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus number of streams.

than CP-RUDP. TCP with a small send buffer configuration, in contrast, shows both much lower values and values that decrease somewhat as the number of streams increase. This follows from both decreasing normalized flowshare numbers, and increasing stall time numbers as the number of streams increases.

Finally, Figure 6.25 (b) and Figure 6.28 (b) show frame ensemble interarrival jitter results. CP-RUDP shows consistently low numbers that remain constant as the number of streams increases. TCP with a small send buffer configuration shows much higher numbers that likewise remain constant. In contrast, TCP with a large send buffer configuration shows higher numbers that increase as the number of streams increases.

6.7.4 Frame Size

In this section we compare the performance of TCP multi-streaming with that of CP-RUDP using various frame sizes. Unlike the unequal frame sizes of Section 6.8, frames within the same ensemble are always equal in size, as is frame size from one frame ensemble to the next throughout the entire run. To study this issue, we use separate runs for each frame size configuration and then compare performance results between different runs. Once again, three trials were run for each configuration and results were averaged to create a single data point for each metric.

Frame size can be easily configured using commandline arguments during *frame* invocation on sending endpoints. The following frame sizes were used: 10, 25, 50, 100, 150, and 200 KB. In keeping with previous experiments, we used a bottleneck bandwidth of 100 Mb/s, a *dummynet* delay setting of 25 milliseconds (for a 50 ms total round trip time), and a *dummynet* loss setting of .01. Experiments were given a 5-minute stabilization period after which data was collected for the next 10 minutes.

Completion asynchrony results in Figure 6.29 (a) show a dramatic difference between TCP with a large send buffer configuration and CP-RUDP. This gap is drastically reduced by decreasing send buffer size as seen in the much improved performance of TCP with a small send buffer configuration. Interestingly, values increase only slightly as frame size increases.

Stall time results in Figure 6.29 (b) show CP-RUDP exhibiting very low stall time numbers that remain constant as frame size increases. TCP with a small send buffer configuration shows numbers that increase somewhat as frame size increases, while TCP with a large send buffer configuration shows numbers that increase dramatically as frame size increases.

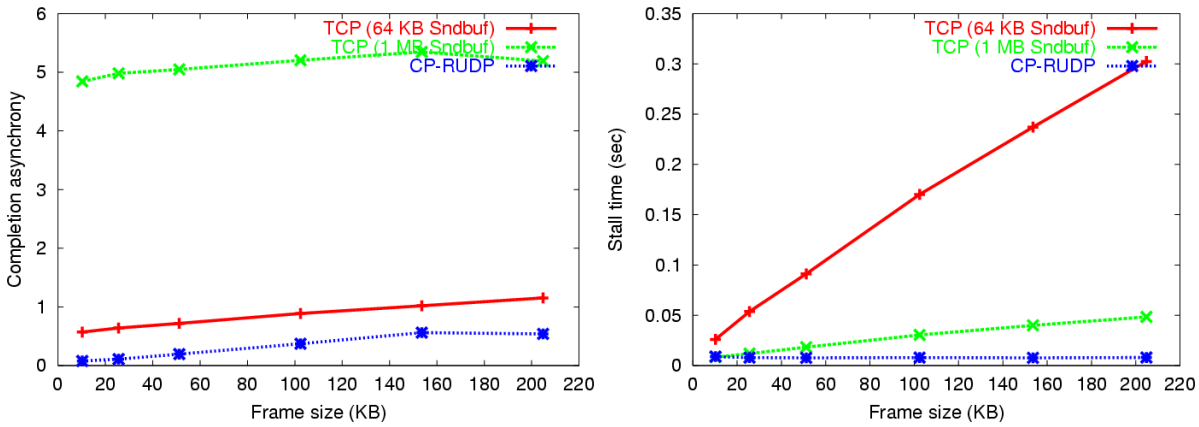


Figure 6.29: Frame size results. (a) Completion asynchrony and (b) stall time versus frame size.

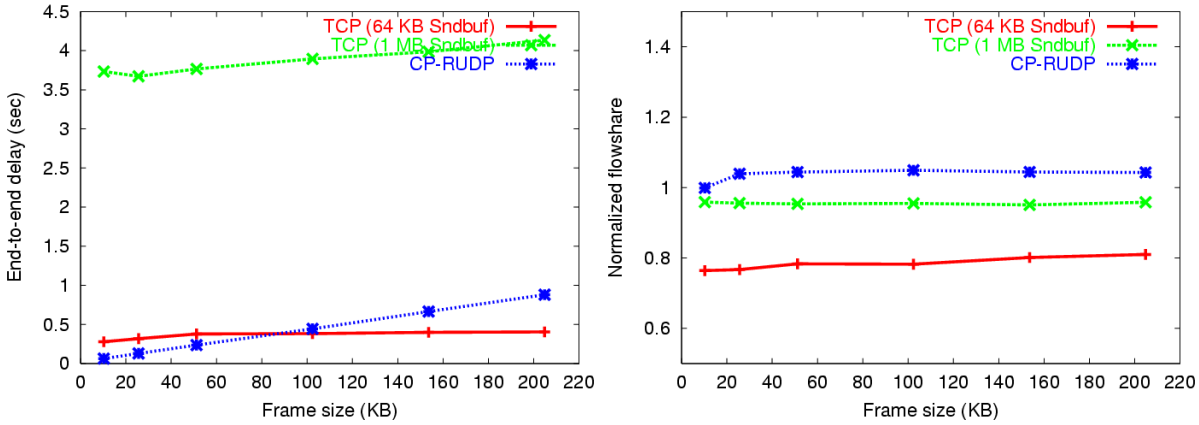


Figure 6.30: Frame size results. (a) End-to-end delay and (b) normalized flowshare versus frame size.

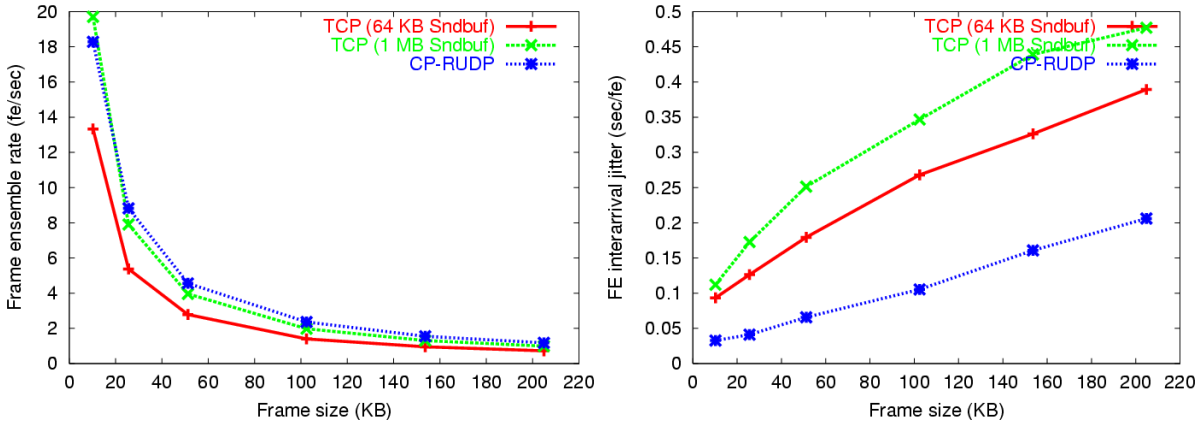


Figure 6.31: Frame size results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus frame size.

In Figure 6.30 (a), we see CP-RUDP with low end-to-end delay values that increase slowly with the size of the frame. TCP with a large send buffer configuration shows significantly higher values that likewise increase. Values for TCP with a small send buffer configuration, in contrast, rise somewhat and then maintain a constant value as frame size continues to increase. This puzzling behavior is not what it appears. A limitation in our delay measurement methodology makes it difficult to quantify end-to-end delay precisely as frame sizes become much larger than send buffer sizes. Hence, delay values are shown in the plot as leveling off when, in fact, end-to-end delay may continue to increase somewhat.

Figure 6.30 (b) shows normalized throughput results. As usual, TCP with a large send buffer configuration is fairly close to 1.0 for almost all frame size configurations, CP-RUDP is just above 1.0, and TCP with a small send buffer configuration is significantly lower. These results once again demonstrate that TCP with a small send buffer configuration fails to get its fair share of bandwidth due to stalling, while both TCP with a large send buffer configuration and CP-RUDP do fairly well without stealing bandwidth from competing bottleneck flows.

Figure 6.31 (a) shows that frame size is inversely related to frame ensemble rate for all CP-RUDP and TCP configurations. In general, CP-RUDP does just slightly better than TCP with a large send buffer configuration which does significantly better than TCP with a small send buffer configuration. These numbers reflect normalized flowshare results as seen in Figure 6.30 (b).

Finally, Figure 6.31 (b) shows all frame ensemble interarrival numbers to increase with frame size for both CP-RUDP and TCP. Values for CP-RUDP are the lowest, while TCP with a large send buffer configuration shows the highest.

6.7.5 Network Load

While testing CP performance using dummynet is instructive, a random loss model is somewhat unrealistic. Although effective in modeling low-level transmission errors, it fails to model losses that occur from network congestion. Such losses, the result of queue overflow in Internet routers, are known to be bursty and correlated. [Pax99] To better capture 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 [SCJO01] to emulate the behavior of Web browsers and the TCP traffic that browsers and servers generate on the Internet. Em-

pirical 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. (See Appendix B for a more complete description of *thttp*.)

We ran four *thttp* servers and four clients on each set of traffic hosts seen in Figure 6.6. 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 2000 to 6000. Figure B.5 shows the workload generated as the number of browsers emulated increases.

To monitor loss and to configure queue size, *ALTQ* (see Section B.1) was run on each bottleneck router. *ALTQ* is a FreeBSD kernel extension that supports the configuration and monitoring of outbound queuing on a FreeBSD forwarding host. *ALTQ* provides detailed statistics on packet forwarding and drop events, as well as queue length over time. We configured queue size to be 100 packets [4] and set our statistics collection interval to be 100 ms.

Figure 6.32 (a) shows queue length for a sample time interval on the bottleneck router between video capture and reconstruction hosts. Hosts send 150 KB frames using CP-RUDP and compete with 4000 *thttp* browsers. While mean queue length generally spikes at 40-60 packets, maximum values quite frequently spike to 100 packets. Figure 6.32 (b) shows what happens when queue length exceeds maximum. Here we note that packet losses seem to occur in "bursts" that correspond to moments when the queue length has reached and exceeded its maximum size. Figure 6.33 (a) and (b) show the bottleneck bandwidth to be nearly link capacity (100 Mb/s) and the number of packets per second to be between 1600 and 1950.

Figure 6.34 (a) shows the cumulative distribution function (CDF) for all mean queue length intervals. While more than 30 percent of all 100 ms intervals are zero, values increase significantly after that. Figure 6.34 (a) shows that the number of drops per interval increasing significantly for higher browser loads. Browser loads of 2000 show 90 percent of all intervals with zero drops, and of the remaining intervals, more than

[4] Experiments with larger queue sizes mandated larger browser loads to produce the same loss levels but showed little difference in resulting performance.

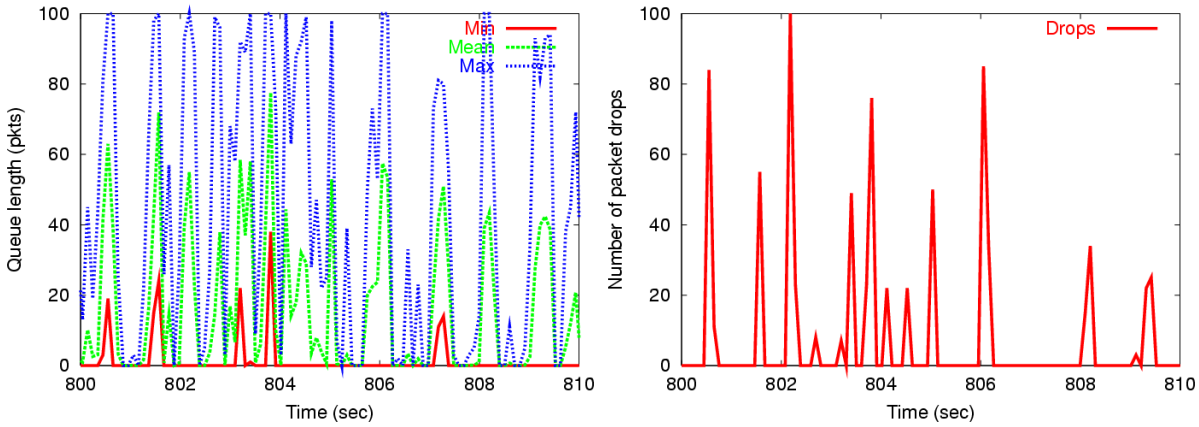


Figure 6.32: Bottleneck router queue results. (a) Queue length and (b) dropped packets for a sample CP-RUDP run with 4000 browsers and 150 KB frame size.

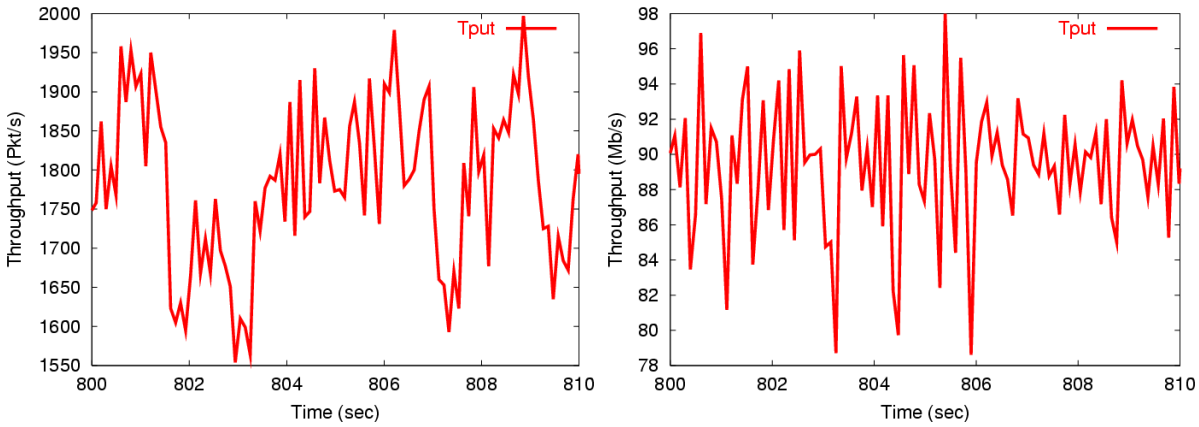


Figure 6.33: Bottleneck router queue results. Throughput in (a) packets per second and (b) megabits per second for a sample CP-RUDP run with 4000 browsers and 150 KB frame size.

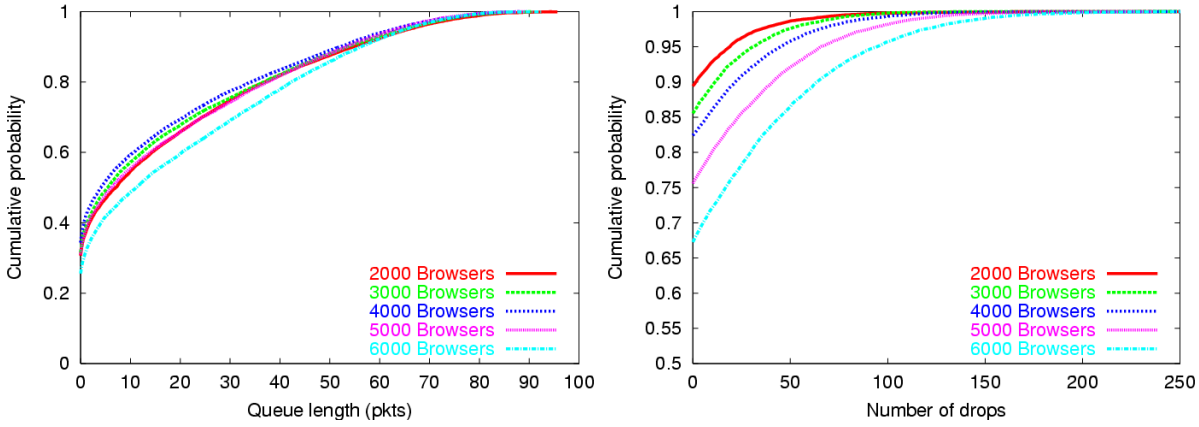


Figure 6.34: Bottleneck router queue results. (a) Queue length CDF and (b) dropped packets CDF for runs with CP-RUDP and 150 KB frames.

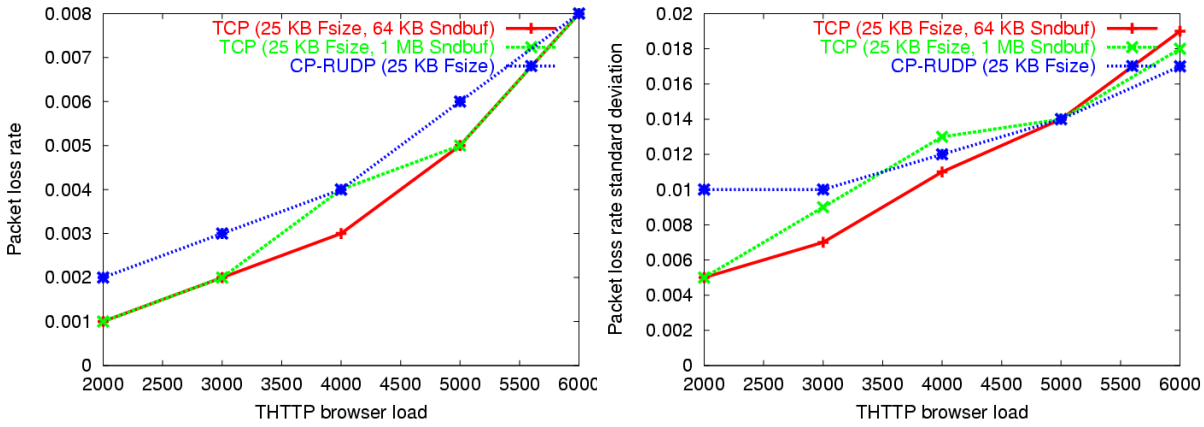


Figure 6.35: Network load results. (a) Packet loss rate and (b) packet loss rate standard deviation for 25 KB frame size.

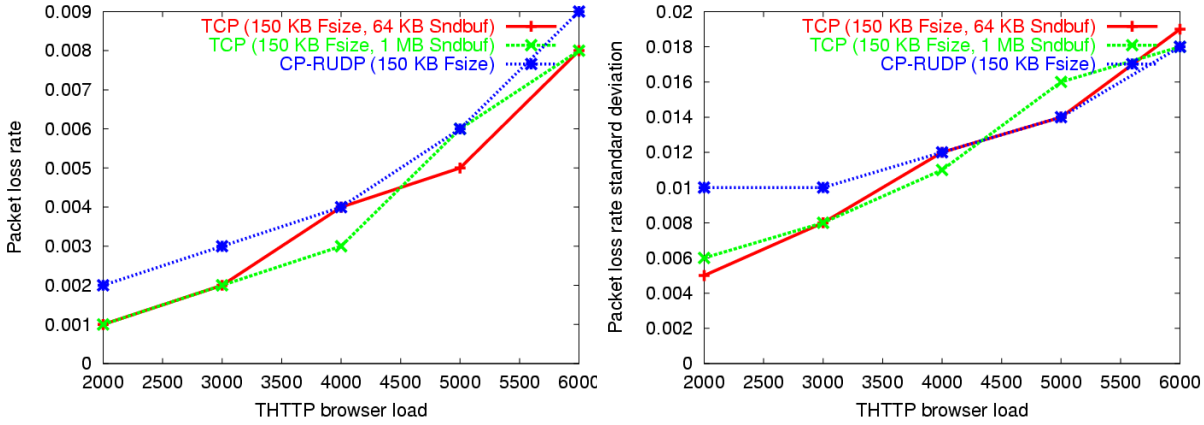


Figure 6.36: Network load results. (a) Packet loss rate and (b) packet loss rate standard deviation for 150 KB frame size.

half are less than 30 packets. In contrast, browser loads of 6000 show only 68 percent of all intervals with no drops, and a large fraction of remaining intervals with over 50 packet drops.

Figure 6.35 and Figure 6.36 show overall loss levels for the various run configurations in this experiment set. Once again, we ran TCP with two sender buffer configurations, 64 KB and 1 MB, and used two frame sizes, 25 KB and 150 KB. As before, we used a bottleneck bandwidth of 100 Mb/s and a *dummynet* delay setting of 25 milliseconds (for a 50 ms total round trip time). We ran three trials and then averaged the results to create a single data point for each metric. Experiments were given a 10-minute stabilization period after which data was collected for the next 10 minutes. Both

Figure 6.35 (a) and Figure 6.36 (a) show loss rates increasing from .001-.002 to .008-.009 as the number of browsers increase from 2000 to 6000. Standard deviations in Figure 6.35 (b) and Figure 6.36 (b) show increasing variance in values as the number of browsers increase, with CP-RUDP showing a markedly higher variance than TCP for the lowest browser number setting. Experiments with even lower browser numbers seem to indicate that TFRC has trouble estimating bandwidth when very little loss exists in the system. Values in the denominator of Equation 4.1 approach zero, triggering a minimum default sending rate that may or may not be well-matched to current conditions on the cluster-to-cluster forwarding path.

Figure 6.37 (a) and Figure 6.40 (a) show that as background TCP traffic increases, completion asynchrony remains consistently low for CP-RUDP. TCP with a small send buffer size also stays fairly low, although rising slightly. Stall time results in Figure 6.37 (b) and Figure 6.40 (b) once again show CP-RUDP being very low and relatively insensitive to increases in *thttp* traffic load. In stark contrast, both TCP configurations are significantly higher and increase as load increases.

End-to-end delay values in Figure 6.38 (a) and Figure 6.41 (a) show CP-RUDP and TCP with a small send buffer size configuration remaining quite low while for TCP with a larger send buffer size, this is not the case. In the latter, values are significantly higher and increase a great deal as the background traffic load increases. Normalized flow share values in Figure 6.38 (b) and Figure 6.41 (b) show CP-RUDP getting somewhat more than its fair share for the 2000 browser level, but then remaining fairly close to 1.0 for the other configurations. Once again, we suspect this is an issue related to too little loss in the system and the difficulty TFRC has with driving Equation 4.1 when values in the denominator approach zero. In general, both TCP configurations under-utilize network bandwidth for both frame size configurations, especially the smaller send buffer size configuration.

Figure 6.39 (a) and Figure 6.42 (a) show frame ensemble rate results. In all cases, as background traffic load increases, frame ensemble throughput decreases. As with normalized flowshare results in Figure 6.38 (b) and Figure 6.41 (b), CP-RUDP gets significantly better frame ensemble rates than TCP. The latter with a small send buffer configuration consistently performs the worst. Finally, Figure 6.39 (b) and Figure 6.42 (b) show CP-RUDP being relatively insensitive to increases in background traffic load in terms of frame ensemble interarrival jitter. In contrast, numbers increase significantly for both TCP configurations and are the highest for TCP with a large send buffer

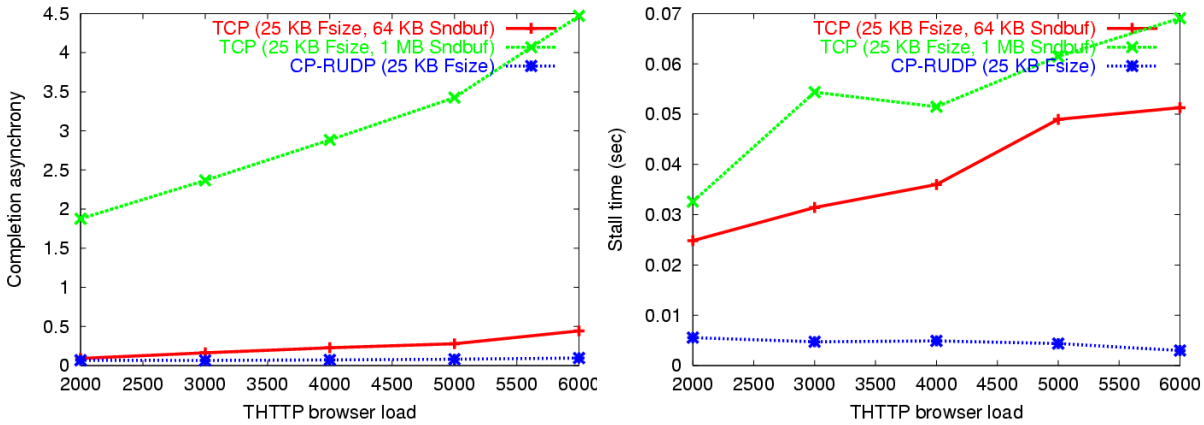


Figure 6.37: Network load results. (a) Completion asynchrony and (b) stall time versus network load.

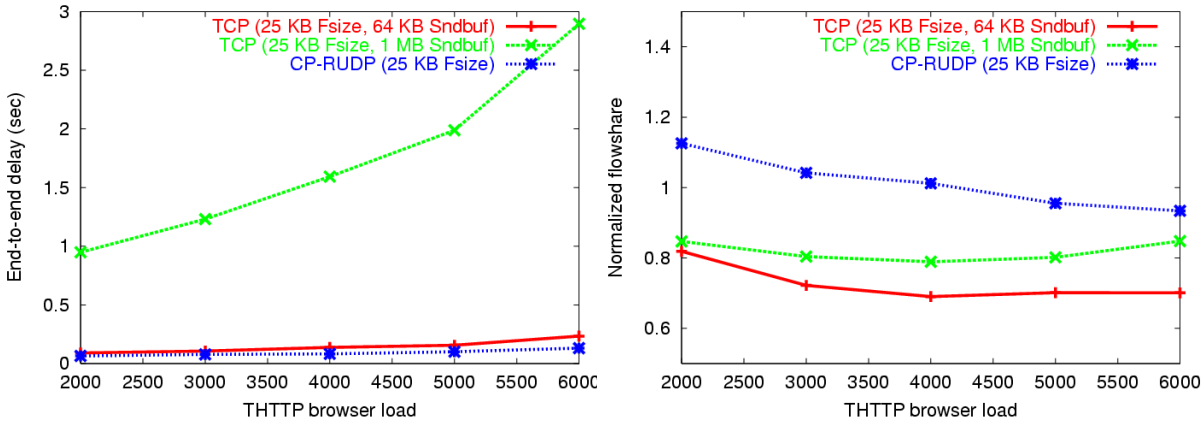


Figure 6.38: Network load results. (a) End-to-end delay and (b) normalized flowshare versus network load.

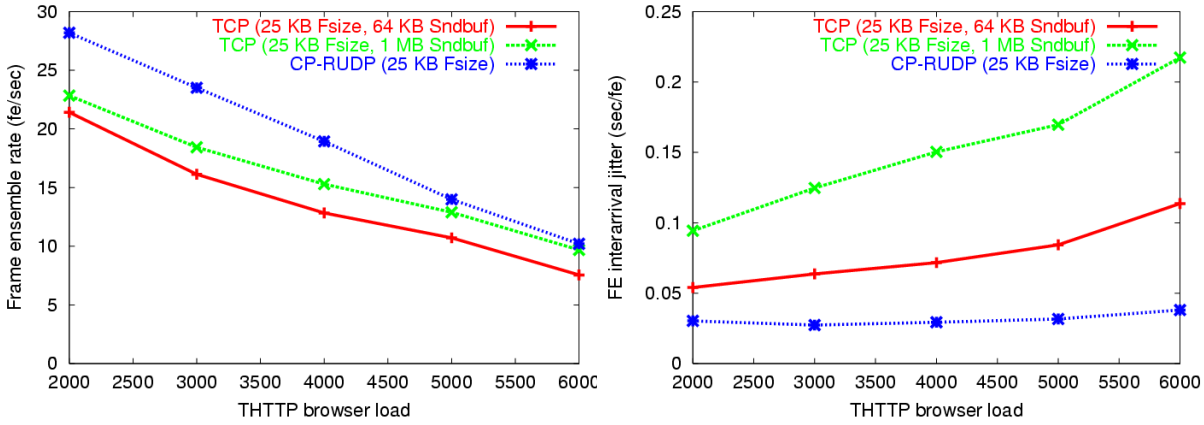


Figure 6.39: Network load results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus network load.

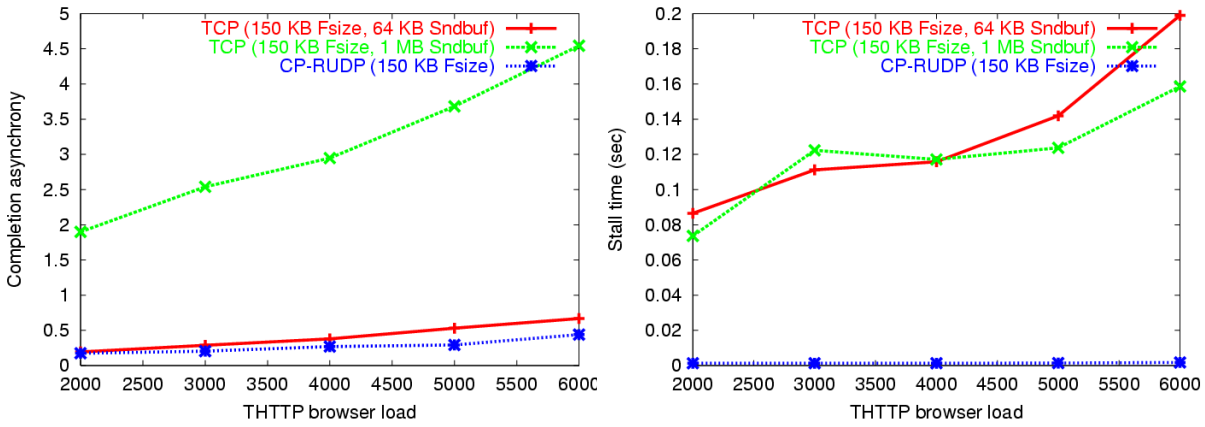


Figure 6.40: Network load results. (a) Completion asynchrony and (b) stall time versus network load.

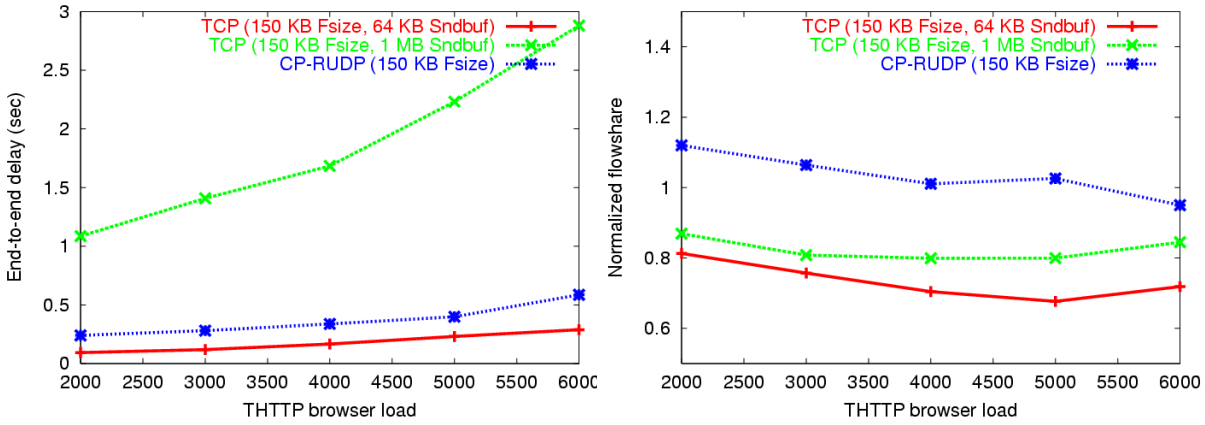


Figure 6.41: Network load results. (a) End-to-end delay and (b) normalized flowshare versus network load.

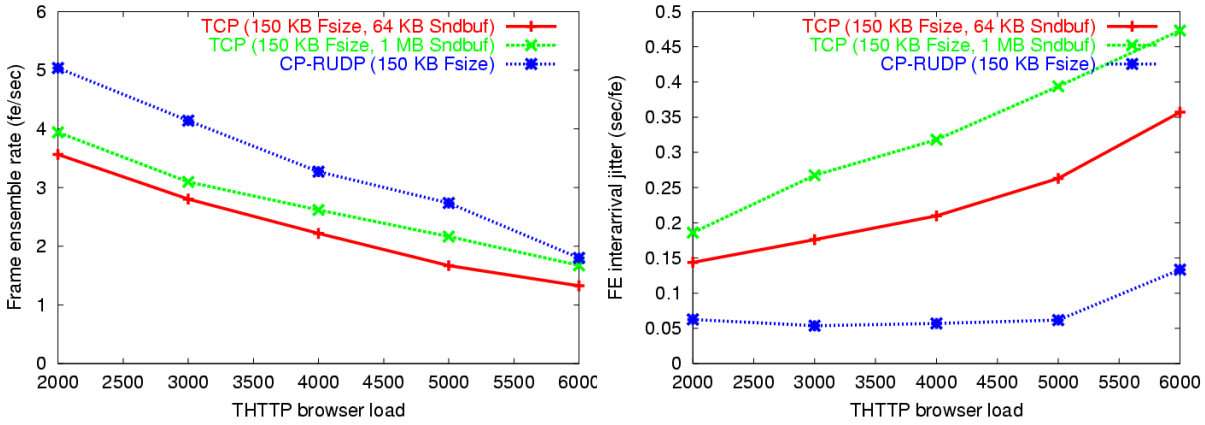


Figure 6.42: Network load results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus network load.

configuration.

6.7.6 Summary

Results for equal frame size configurations presented in this section demonstrate the superior performance of CP-RUDP over TCP. In support of this statement, we cite the following facts and observations:

- CP-RUDP invariably shows less completion asynchrony than either TCP configuration.
- CP-RUDP invariably shows less stall time than than either TCP configuration.
- CP-RUDP shows less end-to-end delay than either TCP configuration for 25 KB frames. For 150 KB frames, it is very close to that of TCP with a small send buffer configuration.
- CP-RUDP shows less frame ensemble interarrival jitter than either TCP configuration.
- CP-RUDP typically achieves full link utilization and, as a result, shows better frame ensemble rates.

We furthermore point out that no single TCP configuration can compete with the performance of CP-RUDP for all metrics simultaneously. While TCP with a small send buffer configuration offers low completion asynchrony, low end-to-end delay, and lower frame ensemble interarrival jitter, it also results in longer stall times, poor network utilization, and lower frame ensemble rate numbers. TCP with a large buffer configuration, on the other hand, offers lower stall times, good network utilization and fairness, and good frame ensemble rate numbers. At the same time, it results in high completion asynchrony, large end-to-end latency values, and higher frame ensemble interarrival jitter. CP-RUDP, on the other hand, *offers the best of both worlds* and exhibits performance that often exceeds the best that either TCP configuration can offer.

Caveats to this performance summary include mildly over-aggressive bandwidth estimation of CP-RUDP when (1) very little network load exists, and (2) very high random loss rates are seen. Each of these is an artifact of Equation 4.1, used by

CP-RUDP (and TFRC) to estimate a congestion-responsive sending rate. The former problem occurs when very small loss event rate values drive the denominator to near zero, and the latter occurs when the problem documented in [Wid00] is invoked. Both of these problems may be solved in the future as research in equation-based congestion control [FF99, PFTK98, FHPW00] continues to advance. When this occurs, CP-RUDP can be easily modified to incorporate the improvements to solve these performance issues.

6.8 Laboratory Testbed Results: Unequal Frame Size

In this section, we look at experimental results for unequal frame size ensembles. Unequal frame sizes within a frame ensemble may occur when a different capture resolution is used for frames within a user's field of interest, or due to the effects of data compression on images that vary somewhat in content. The unequal frame size scheme used in this section focuses on the former scenario, and in particular the worst case scenario in which the same capture hosts continually have more data to send than peer capture hosts in the same application. (Normally, a user's field of interest would change over time, potentially averaging out some frame size differences.)

Once again, six capture endpoints are used. Each, in this scheme, is configured to send either a large or small frame depending upon the configuration. Five configurations were explored as follows:

- ***Mmmmmm.*** Large, small, small, small, small, and small.
- ***MMmmmm.*** Large, large, small, small, small, and small.
- ***MMMmmm.*** Large, large, large, small, small, and small.
- ***MMMMmm.*** Large, large, large, large, small, and small.
- ***MMMMMm.*** Large, large, large, large, large, and small.

The precise size of "large" and "small" depends upon three configuration parameters: *mean frame size*, *number of frames*, and *dispersion factor*. Mean frame size, μ, is kept constant across all configurations in the set. In the results presented below, a value of

25 KB was used throughout. The number of frames, n, in an ensemble is likewise a constant. A value of six was used throughout corresponding to the number of capture hosts available in our laboratory testbed. Dispersion factor, f, is an input parameter and is used to calculate small frame size (S_{small}) as follows:

$$S_{small} = \mu - (f * \mu) \tag{6.5}$$

Values for f used for our results include 0, .05, .1, .2, .3, .4, and .5. Using S_{small}, large frame size (S_{large}) is then calculated as:

$$S_{large} = \frac{(\mu * n) - (S_{small} * n_{small})}{n_{large}} \tag{6.6}$$

For example, if $\mu = 25KB$, $n = 6$, and $f = .3$, then the following frame sizes (in bytes) are used:

- ***Mmmmmm.*** 64000, 17920, 17920, 17920, 17920, and 17920.
- ***MMmmmm.*** 40960, 40960, 17920, 17920, 17920, and 17920.
- ***MMMmmm.*** 33280, 33280, 33280, 17920, 17920, and 17920.
- ***MMMMmm.*** 29440, 29440, 29440, 29440, 17920, and 17920.
- ***MMMMMm.*** 27136, 27136, 27136, 27136, 27136, and 17920.

Note that the total number of bytes in a single frame ensemble remains constant across all configurations in the set.

Endpoints using CP-RUDP, as described in Section 6.5, are configured to send at a rate that reflects both network path conditions and the fraction of frame ensemble data represented by their particular frame size. Equation 6.1 shows exactly how this sending rate is calculated.

Finally, we also consider briefly the effect of changing frame sizes within the same run on multi-streaming performance. In this scenario, a particular configuration (e.g., *MMmmmm*) is continually re-mapped to endpoint hosts in a dynamic manner, simulating changes that might occur in 3DTI as a user changes their field of interest. This serves to demonstrate another important feature of the Coordination Protocol, namely the ability to perform *dynamic reconfiguration* within a flow coordination scheme. Dynamic reconfiguration is easily implemented using the state sharing mechanisms described in Chapter 3. In fact, the unequal frame size coordination scheme described in

Section 6.5 will naturally handle dynamic send rate reconfiguration in its current form. This is because endpoints continually update the AP state table with their current frame size as part of the algorithm.

6.8.1 Frame Size Dispersion (Random Loss)

In this section we compare the performance of TCP multi-streaming with that of CP-RUDP using unequal frame sizes and random loss generated by *dummynet*. Frame size configurations are described in the previous section (Section 6.8). We refer to these configurations here and within plot labels using the shorthand *Mmmmmm*, *MMmmmm*, *MMMmmm*, *MMMMmm*, and *MMMMMm*. A frame dispersion parameter is varied from 0 to .5 using a step size of .1. For each step, a new set of unequal frame size configurations is generated.

Throughout the run, we used a *dummynet* packet loss rate of .01. Bottleneck bandwidth was once again 100 Mb/s and round trip time was 50 ms. To simplify the results somewhat, a single TCP send buffer configuration was used (400 KB). This buffer size represents a compromise between the 64 KB and 1 MB extremes studied in Section 6.7. Experiments were given a 5-minute stabilization period after which data was collected for the next 10 minutes. Three trials were run for each configuration and results were averaged to create a single data point for each metric.

Plots are organized as follows: Figure 6.43, Figure 6.44, and Figure 6.45 show a comparison between TCP and CP-RUDP for two representative configurations. Figure 6.46, Figure 6.47, and Figure 6.48 show the complete TCP results for all configurations. Figure 6.49, Figure 6.50, and Figure 6.51 show the complete CP-RUDP results for all configurations.

It is interesting to note from Figure 6.49, Figure 6.50, and Figure 6.51 that CP-RUDP, with its coordination scheme described in Section 6.5, is *completely insensitive to both unequal frame size configuration and frame dispersion factor*. This is evident from the straight line seen across dispersion values, and the fact that these straight lines roughly coincide for each configuration.

In contrast, Figure 6.46, Figure 6.47, and Figure 6.48 show a great deal of performance variation, with the exception of completion asynchrony where all dispersion values result in roughly the same values for all unequal frame size configurations. *Mmmmmm* results in the worst performance, as evident by the largest increases in stall time (Figure 6.46 (b)), the largest decreases in normalized flowshare (Figure 6.47 (b)),

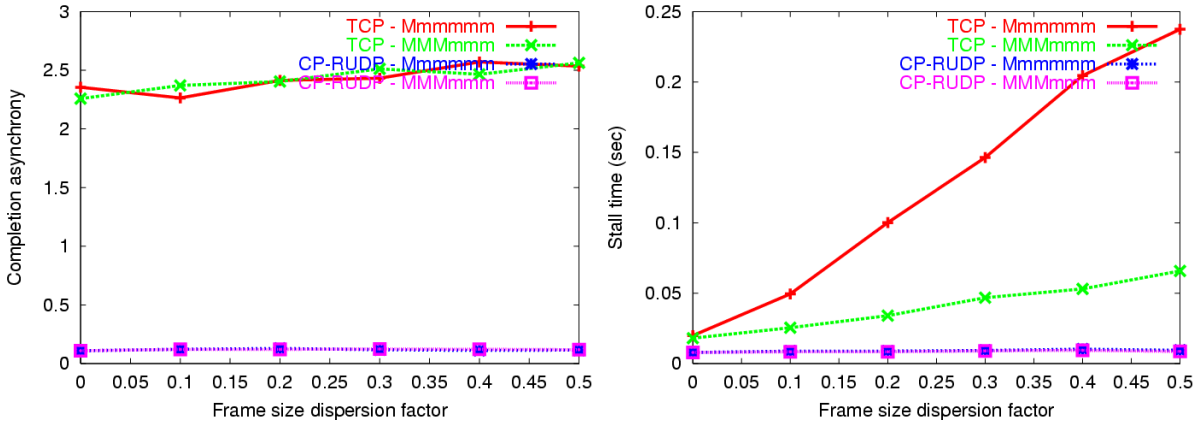


Figure 6.43: Frame size dispersion (random loss) results. (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

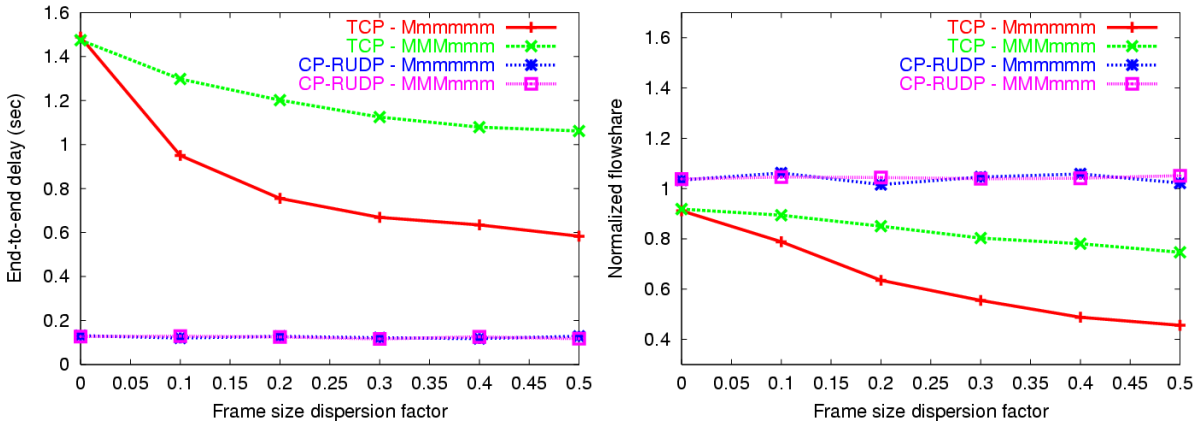


Figure 6.44: Frame size dispersion (random loss) results. (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

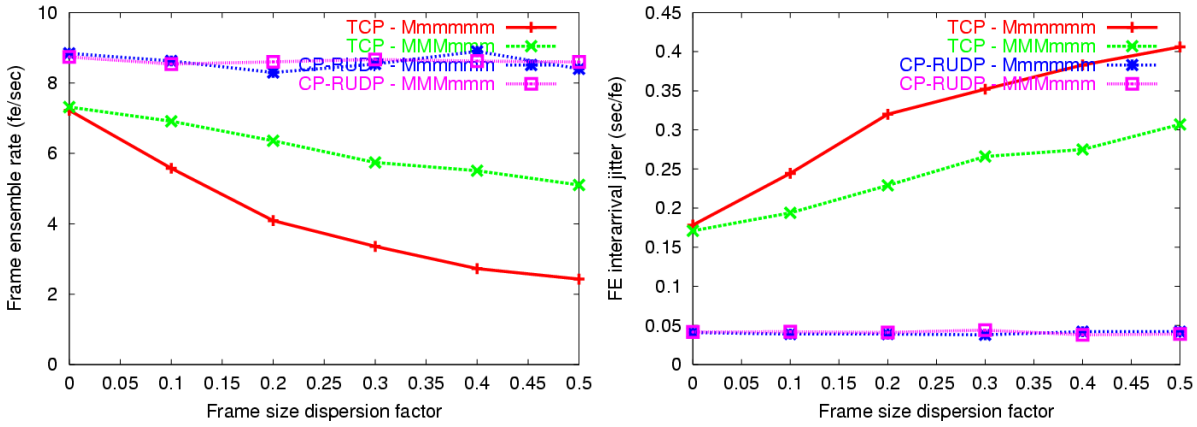


Figure 6.45: Frame size dispersion (random loss) results. (a) Frame ensemble rate and (b) frame interarrival jitter versus frame size dispersion factor.

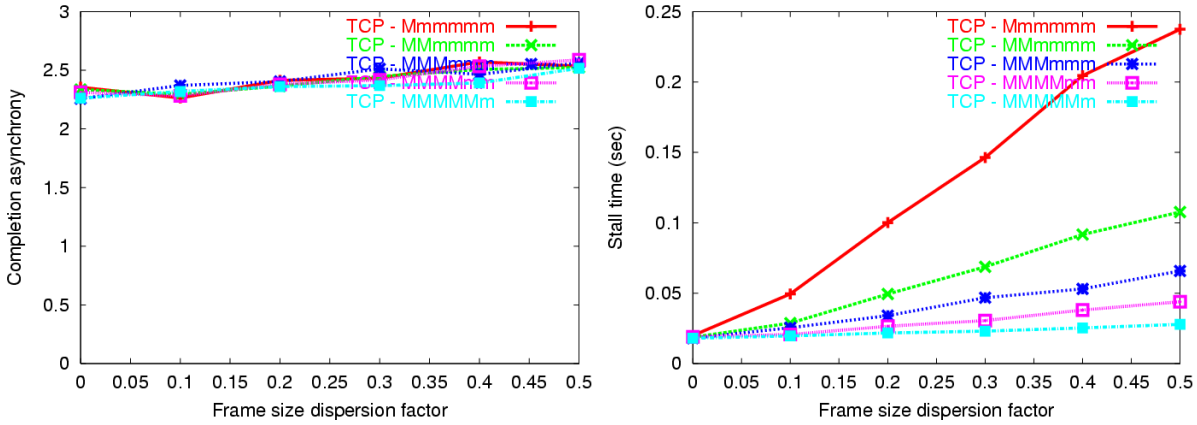


Figure 6.46: TCP frame size dispersion (random loss) results. (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

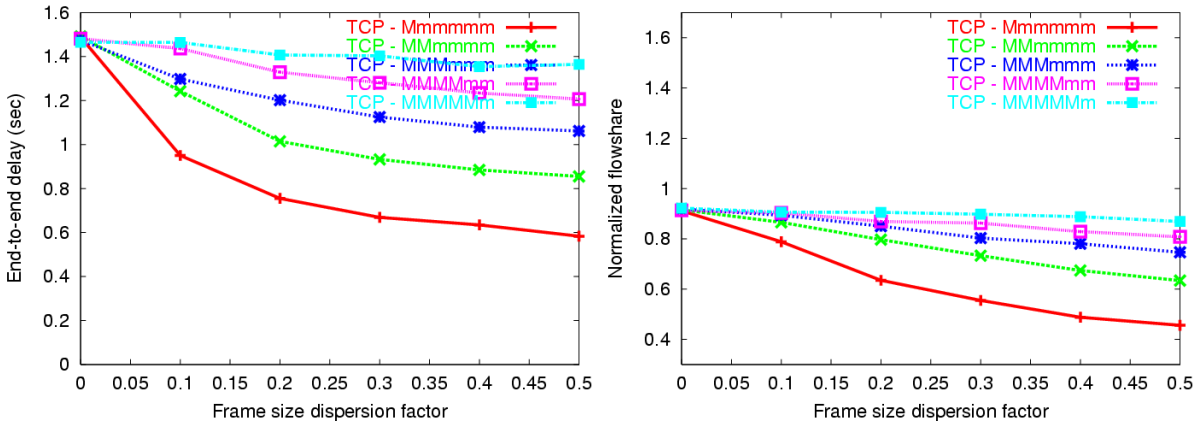


Figure 6.47: TCP frame size dispersion (random loss) results. (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

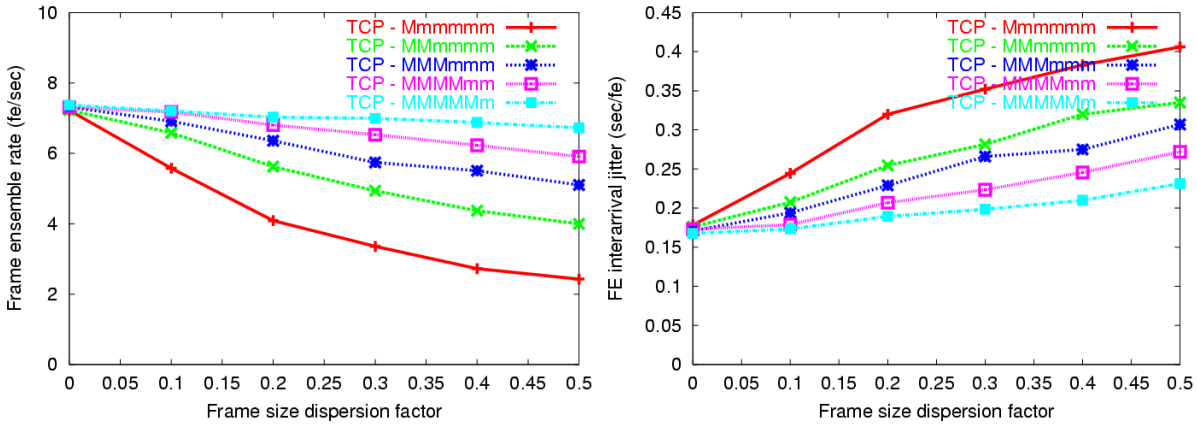


Figure 6.48: TCP frame size dispersion (random loss) results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus frame size dispersion factor.

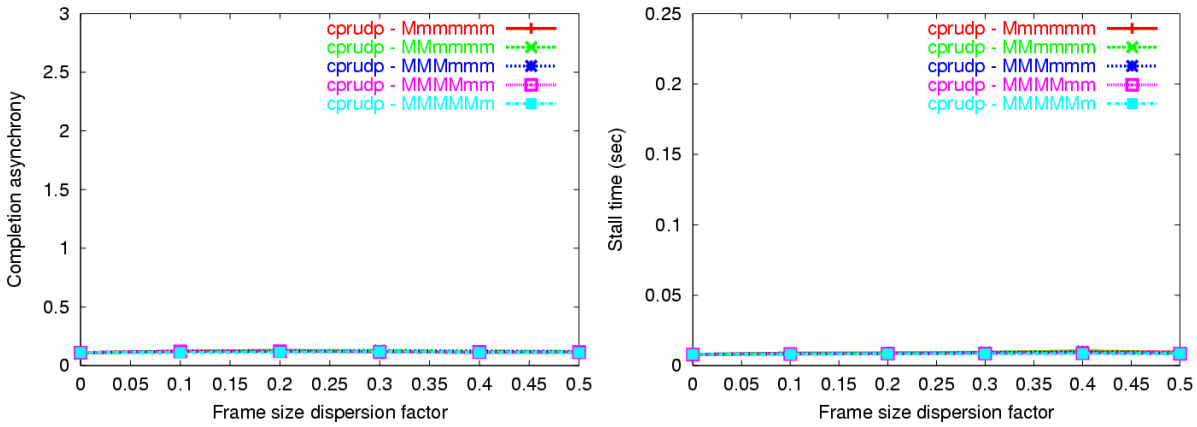


Figure 6.49: CP-RUDP frame size dispersion (random loss) results. (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

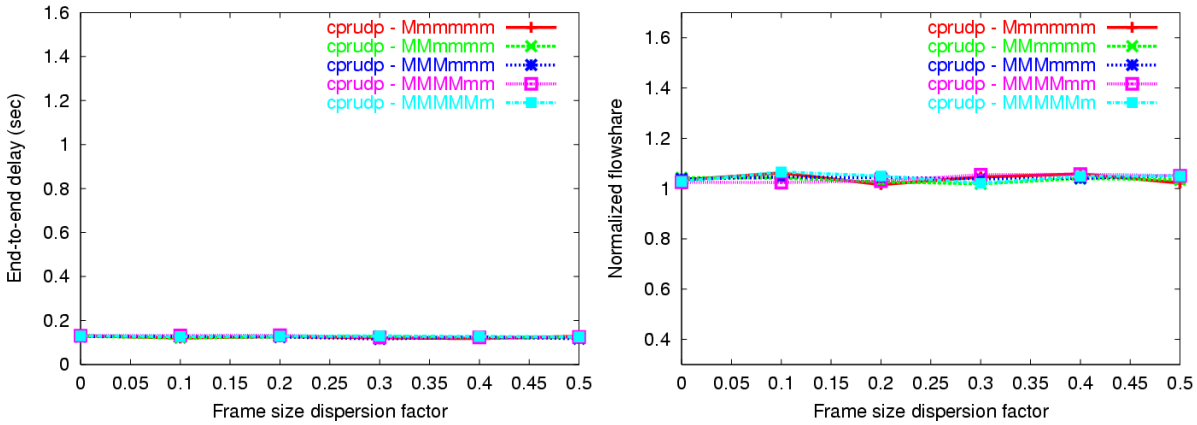


Figure 6.50: CP-RUDP frame size dispersion (random loss) results. (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

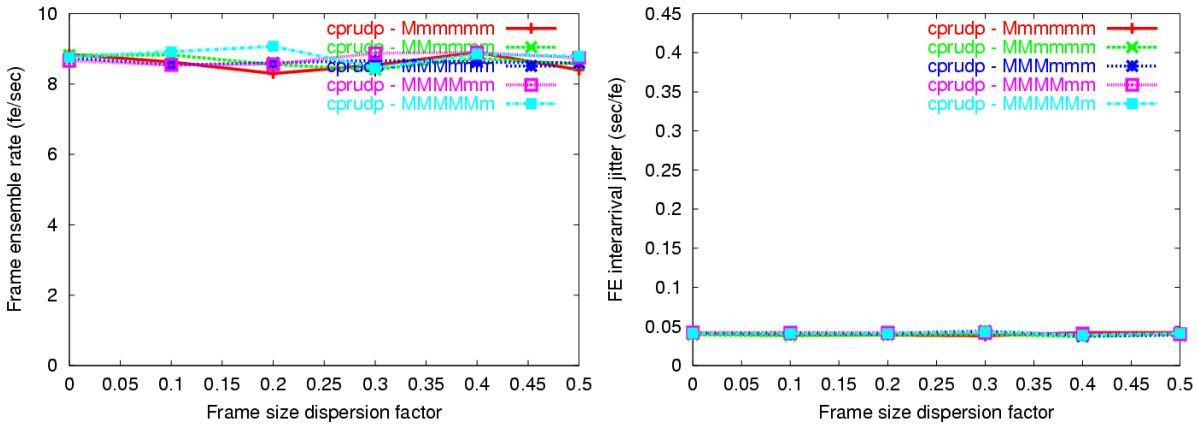


Figure 6.51: CP-RUDP frame size dispersion (random loss) results. (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus frame size dispersion factor.

the largest decreases in frame ensemble rate (Figure 6.48 (a)), and the largest increases in frame ensemble interarrival jitter (Figure 6.48 (b)). Interestingly, *Mmmmmm* did better in terms of end-to-end delay (Figure 6.47 (a)), presumably because many small frames naturally reduce the mean end-to-end values since they are averaged across flows. Completion asynchrony (Figure 6.46 (a)) appears to be dominated by send buffer size, and hence remained constant throughout.

The *MMMMMm* configuration resulted in the best TCP performance, as evident by the smallest increases in stall time (Figure 6.46 (b)), the smallest decreases in normalized flowshare (Figure 6.47 (b)), the smallest decreases in frame ensemble rate (Figure 6.48 (a)), and the smallest increases in frame ensemble interarrival jitter (Figure 6.48 (b)). Interestingly, *MMMMMm* did the worst in terms of end-to-end delay (Figure 6.47 (a)).

Comparing TCP with CP-RUDP for the *Mmmmmm* and *MMMmmm* configurations in Figure 6.43, Figure 6.44, and Figure 6.45, we see CP-RUDP significantly outperforming TCP in every metric. CP-RUDP shows far lower completion asynchrony numbers (Figure 6.43 (a)), low stall time numbers that do not increase with the dispersion factor (Figure 6.43 (b)), end-to-end delay values that are consistently low (Figure 6.44 (a)), normalized flowshare values that remain close to 1.0 (Figure 6.44 (b)), frame ensemble rate values that are consistently high for all dispersion values (Figure 6.45 (a)), and frame ensemble interarrival jitter values that remain consistently low (Figure 6.45 (b)). Clearly, the coordination scheme described in Section 6.5 has done its job in improving multi-streaming performance even more significantly than in various equal frame size cases studied in Section 6.7.

6.8.2 Frame Size Dispersion (Load)

In this section, we compare the performance of TCP multi-streaming with that of CP-RUDP using unequal frame sizes under conditions of substantial TCP background traffic workload. To generate this traffic, the *thttp* Web traffic emulator (see Appendix B) was used with a configuration of 5000 browsers. As mentioned in Section 6.7.5, *thttp* traffic loads generate network congestion behavior at bottleneck routers in a way that closely corresponds to the real Internet. This includes outbound queuing delay and packet loss patterns that are bursty and correlated. [Pax99]

Once again, frame size configurations use the *Mmmmmm*, *MMmmmm*, *MMMmmm*, *MMMMmm*, and *MMMMMm* scheme described in Section 6.8. A frame dispersion

parameter is varied from .1 to .5 using a step size of .1, and with a mean frame size of 25 KB. For each step, a new set of unequal frame size configurations is generated. Bottleneck bandwidth was once again 100 Mb/s and round trip time was 50 ms. The TCP send buffer configuration used was 400 KB. Experiments were given a 5-minute stabilization period after which data was collected for the next 10 minutes.

Once again, Figure 6.52, Figure 6.53, and Figure 6.54 show a comparison between TCP and CP-RUDP for two representative configurations. Figure 6.55, Figure 6.56, and Figure 6.57 show the complete TCP results for all configurations. Figure 6.58, Figure 6.59, and Figure 6.60 show the complete CP-RUDP results for all configurations.

In general, results match those in the previous section. CP-RUDP, with its coordination scheme described in Section 6.5, is almost completely *insensitive to both unequal frame size configuration and frame dispersion factor*. This is seen from Figure 6.58, Figure 6.59, and Figure 6.60 in the straight horizontal lines across all dispersion values, and the fact that lines for different unequal frame size configurations coincide with one another on each plot. There is one exception: normalized flowshare for the *Mmmmmm* configuration which increases somewhat as the frame dispersion factor increases from .2 to .5. It is unclear what is causing this anomaly.

In contrast, TCP in Figure 6.55, Figure 6.56, and Figure 6.57 exhibits a great deal of performance variation depending upon the metric. Once again, *Mmmmmm* results in the worst performance, as evident by the largest increase in stall time (Figure 6.55 (b)), the largest decrease in normalized flowshare (Figure 6.56 (b)), the largest decrease in frame ensemble rate (Figure 6.57 (a)), and largest increase in frame ensemble interarrival jitter (Figure 6.57 (b)). Interestingly, *Mmmmmm* did somewhat better in terms of completion asynchrony (Figure 6.55 (a)) and end-to-end delay (Figure 6.56 (a)).

The *MMMMMm* configuration once again resulted in the best TCP performance, as evident by the smallest increase in stall time (Figure 6.46 (b)), the smallest decrease in normalized flowshare (Figure 6.47 (b)), the smallest decrease in frame ensemble rate (Figure 6.48 (a)), and among the smallest increases in frame ensemble interarrival jitter (Figure 6.48 (b)). Interestingly, *MMMMMm* did the worst in terms of frame arrival asynchrony (Figure 6.46 (a)) and end-to-end delay (Figure 6.47 (a)).

Comparing TCP with CP-RUDP for the *Mmmmmm* and *MMMmmm* configurations in Figure 6.52, Figure 6.53, and Figure 6.54, we see CP-RUDP significantly outperforming TCP in every metric. CP-RUDP shows far lower completion asynchrony numbers that do not increase with the dispersion factor (Figure 6.52 (a)), low stall time numbers

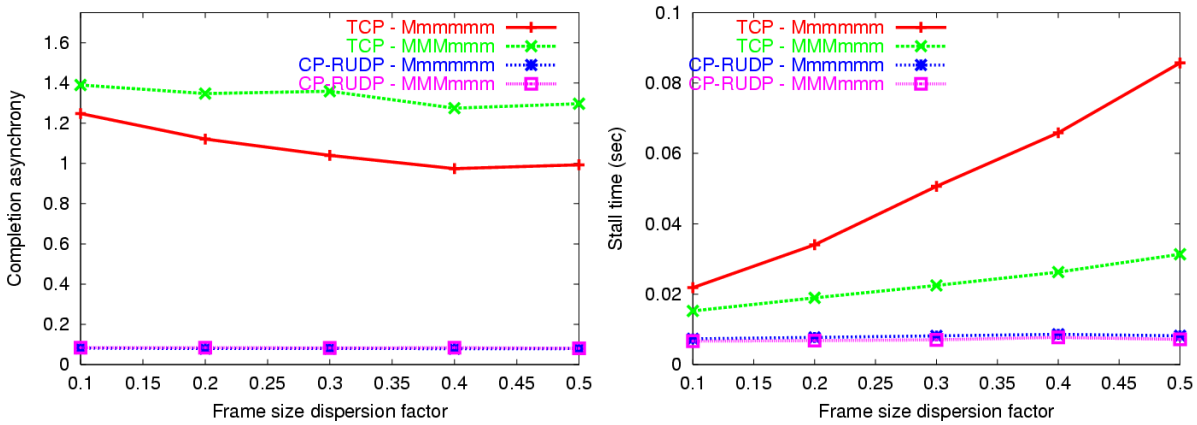


Figure 6.52: Frame size dispersion (load) results. (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

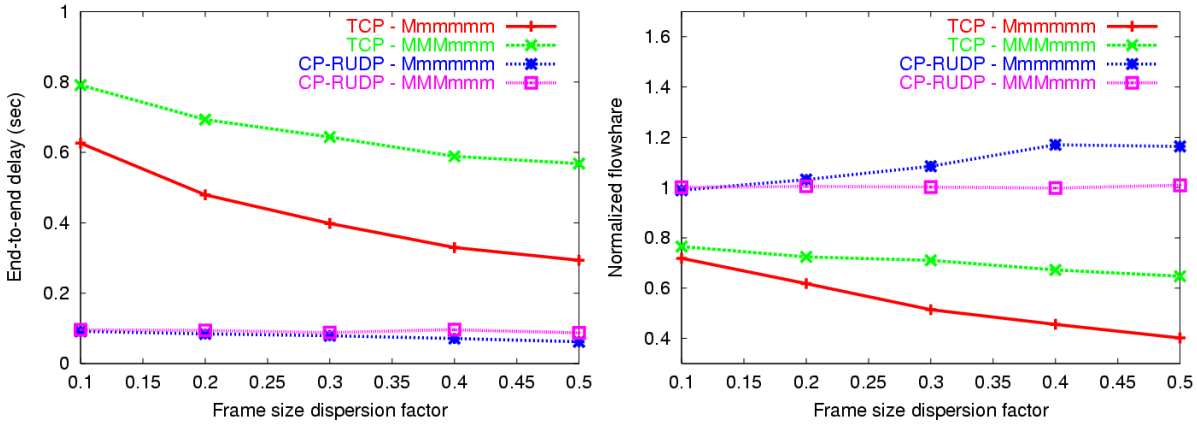


Figure 6.53: Frame size dispersion (load) results. (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

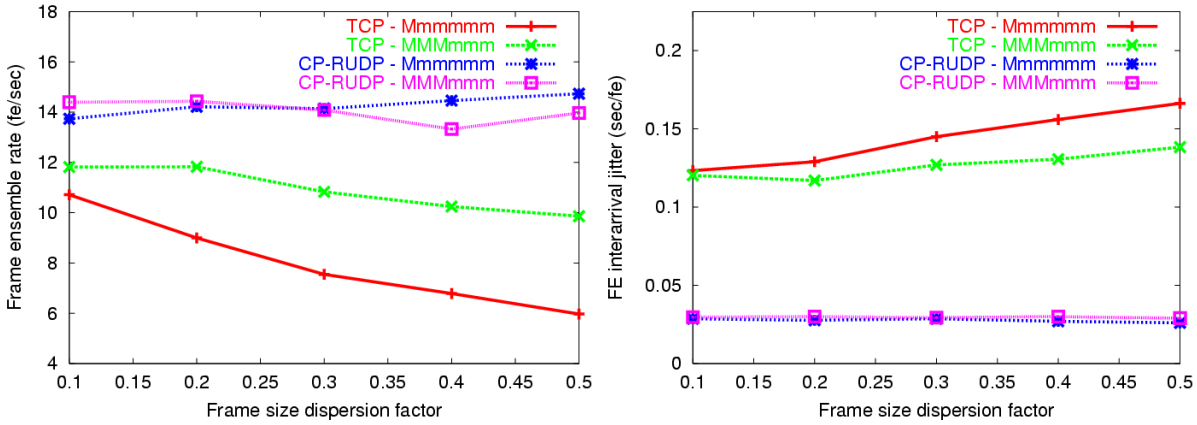


Figure 6.54: Frame size dispersion (load) results. (a) Frame ensemble rate and (b) frame interarrival jitter versus frame size dispersion factor.

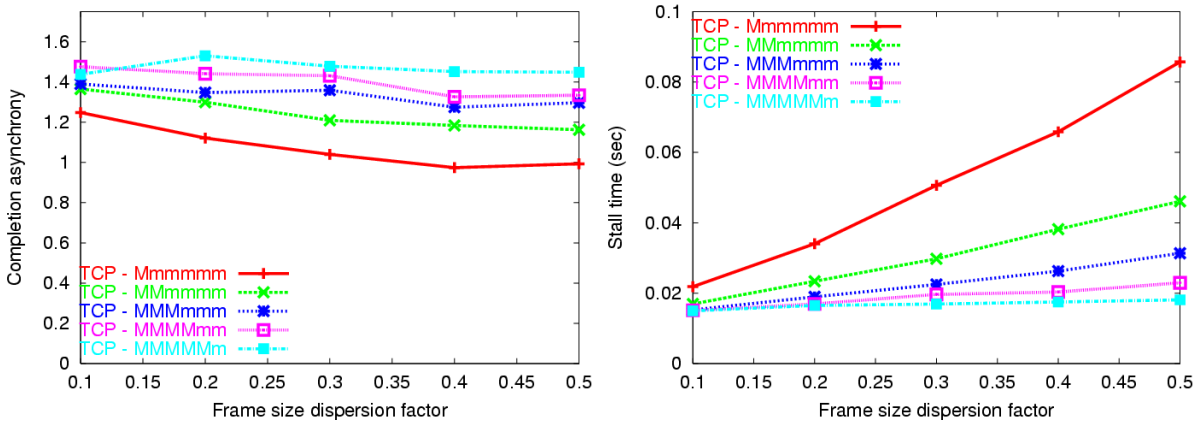


Figure 6.55: TCP frame size dispersion (load) results. (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

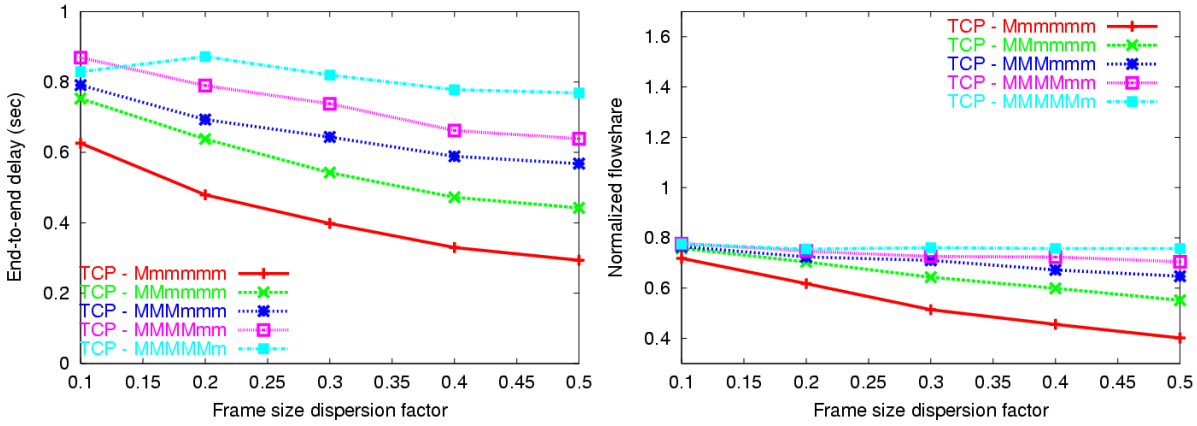


Figure 6.56: TCP frame size dispersion (load) results. (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

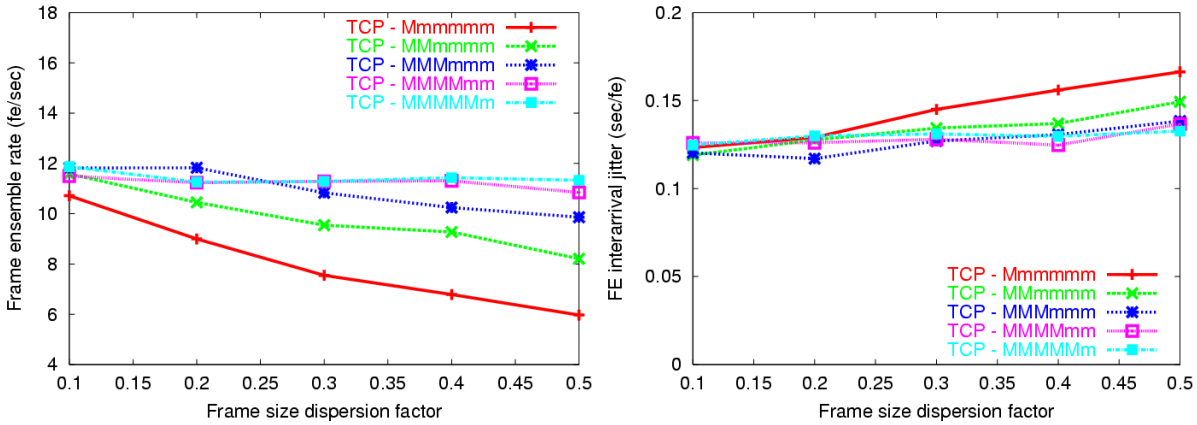


Figure 6.57: TCP frame size dispersion (load) results. (a) Frame ensemble rate and (b) frame interarrival jitter versus frame size dispersion factor.

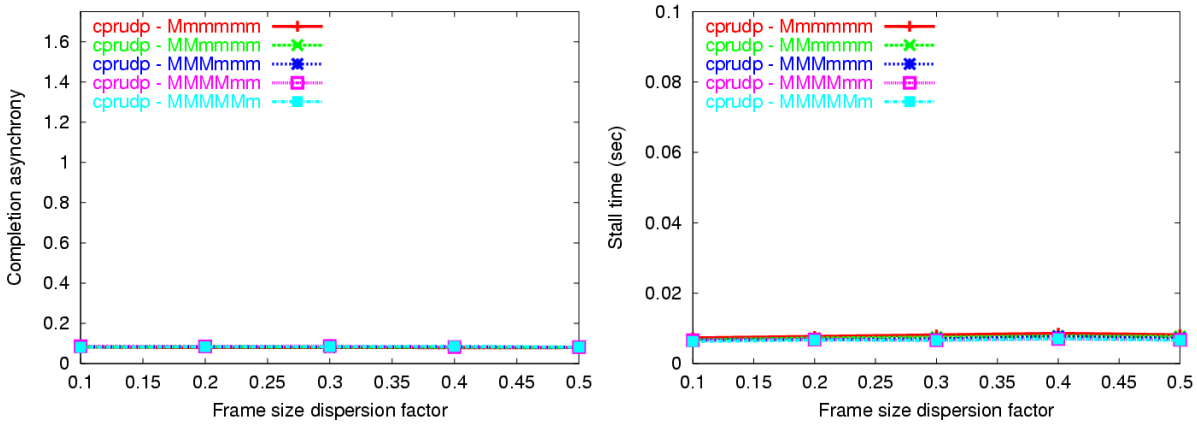


Figure 6.58: CP-RUDP frame size dispersion (load) results. (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

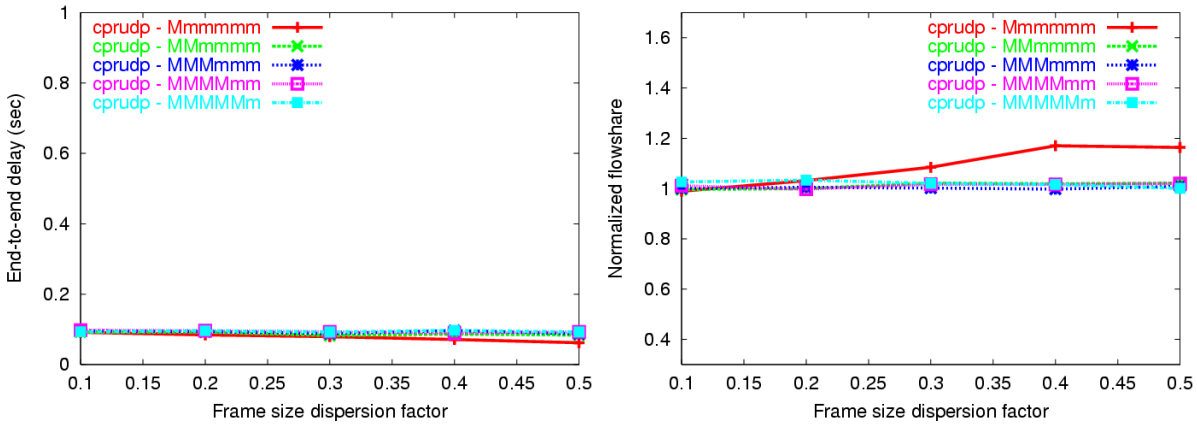


Figure 6.59: CP-RUDP frame size dispersion (load) results. (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

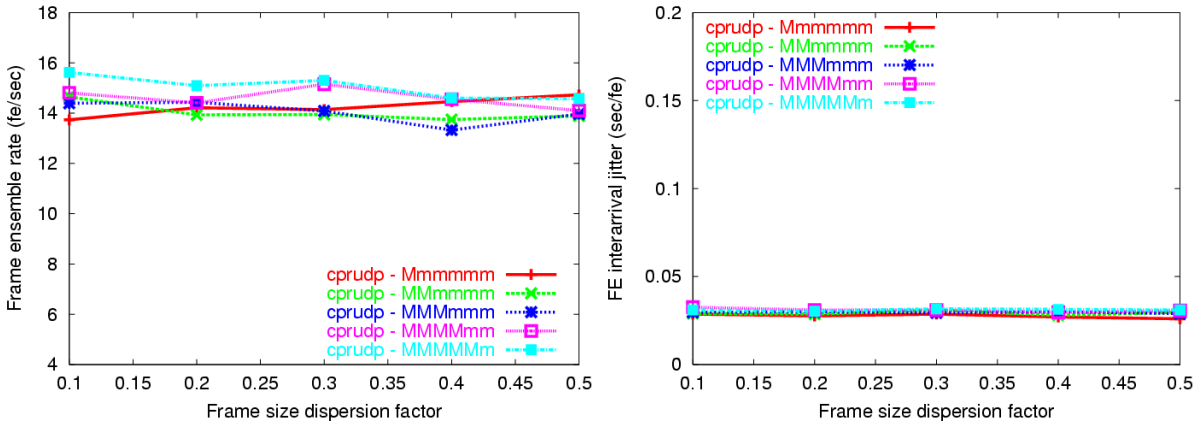


Figure 6.60: CP-RUDP frame size dispersion (load) results. (a) Frame ensemble rate and (b) frame interarrival jitter versus frame size dispersion factor.

that do not increase with the dispersion factor (Figure 6.52 (b)), end-to-end delay values that are consistently low (Figure 6.53 (a)), normalized flowshare values that remain fairly close to 1.0 and do not drop as the dispersion factor increases (Figure 6.53 (b)), frame ensemble rate values that are consistently high for all dispersion values (Figure 6.54 (a)), and frame ensemble interarrival jitter values that remain consistently low (Figure 6.54 (b)). Once again, the coordination scheme described in Section 6.5 has done its job in improving multi-streaming performance even more significantly than in various equal frame size cases studied in Section 6.7.

Finally, it is interesting to compare the throughput behavior of TCP and CP-RUDP within a run for a moment in order to better visualize *why* CP-RUDP performs so much better in the context of multi-streaming with unequal frame sizes. Figure 6.61 and Figure 6.62 show flow throughput results over time for CP-RUDP and TCP, respectively. Each of six flows is shown individually within each plot, with the first plot showing 5 seconds of a run and the second plot showing 15 seconds. Each run used the *MMmmmm* unequal frame size configuration with a maximum dispersion value of 0.5.

Figure 6.61 shows two of six CP-RUDP flows receiving consistently more bandwidth than the other four flows, a result in keeping with coordination scheme described in Section 6.5. These flows stream large frames and thus require a larger fraction of aggregate bandwidth to maintain synchrony with smaller frame size streams. Two other observations should also be noted. First, streams in each subgroup (large and small frame size) exhibit a substantial degree of send rate correspondence with one another. This is especially evident in the smaller frame size where throughput values nearly coincide for all four flows. Second, note that stall events are not seen. A stall event, if it occurred, would appear in the plot as one or more flows dropping to a zero throughput level for some transient period before throughput levels increase once again.

Figure 6.62 shows all six TCP flows competing with one another in an unsynchronized manner. While two flows tend to show higher throughput values than the other four, this is not always the case. In addition, stall events appear common as many flows have throughput values that drop to zero, sometimes for a substantial interval of time. In summary, flows lack coordination and, as a result, tend to use available bandwidth inefficiently. The result is higher completion asynchrony, larger stall times, lower network utilization, and the other effects described above.

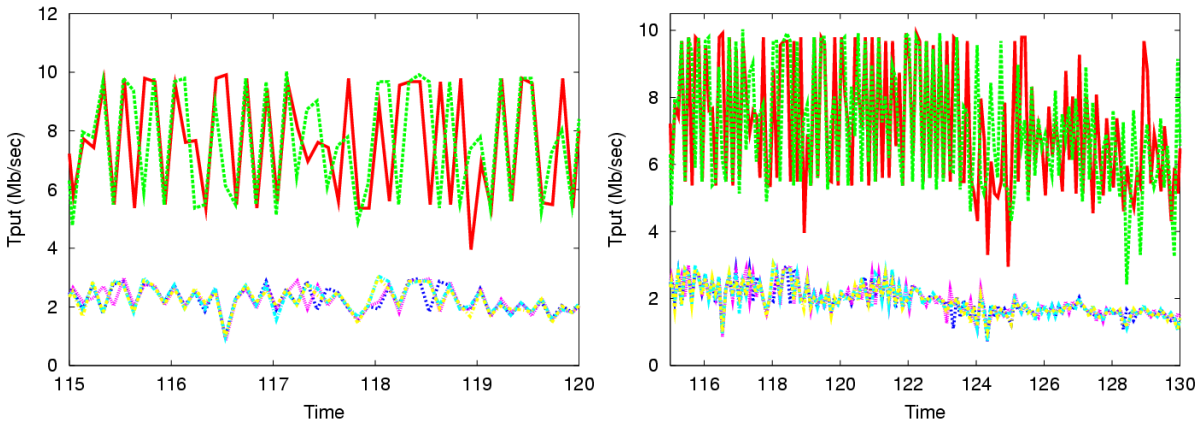


Figure 6.61: Unequal frame size. Throughput over time for CP-RUDP.

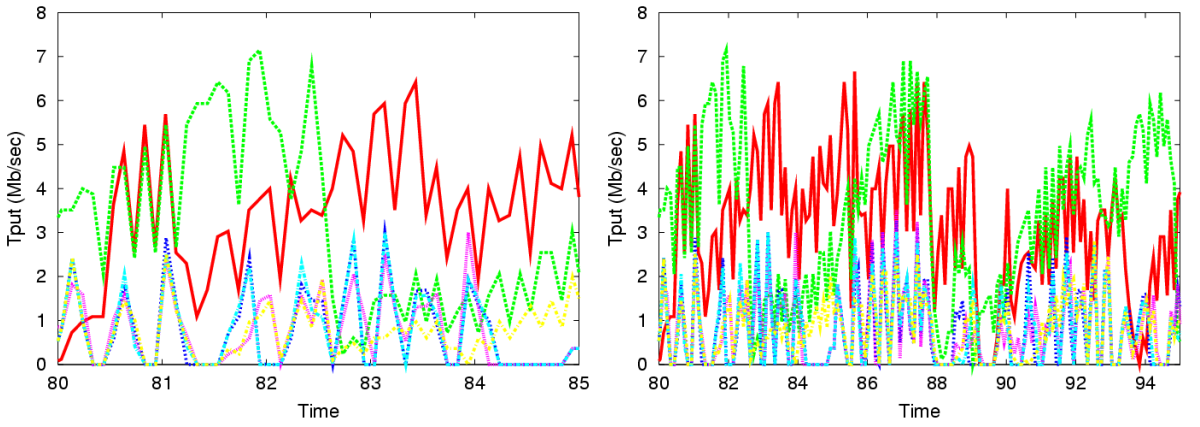


Figure 6.62: Unequal frame size. Throughput over time for TCP.

Metric	*TCP (64 KB Sndbuf)*	*TCP 1 MB Sndbuf*	*CP-RUDP*
Completion Asynchrony	0.173000	2.312000	0.063000
Stall Time	0.018633	0.008733	0.004200
End-to-end Delay	0.118800	1.345300	0.057567
Normalized Flowshare	0.672667	0.811000	1.095000
Frame Ensemble Rate	14.932333	18.869000	21.717000
FE interarrival Jitter	0.080333	0.117000	0.029667

Table 6.2: Unequal frame size with dynamic reconfiguration.

6.8.3 Dynamic Reconfiguration

In this section, we compare briefly the performance of TCP multi-streaming with that of CP-RUDP using unequal frame sizes that change among endpoints *dynamically* over time. The *MMmmmm* unequal frame size configuration was chosen for these experiments, with a 25 KB mean and maximum dispersion factor of 0.5. For each 100 frames, flows modify their frame size by trading configurations with a peer flow, meanwhile maintaining the invariant that two flows are streaming large frames at all times and four are streaming small frames. This dynamic re-configuration models a user's changing field of interest within the application, and the use of higher resolutions for streams that are currently within this field.

Once again, we use *thttp* to generate background Web traffic in a realistic way, with a total of 5000 browsers configured. Bottleneck bandwidth is 100 Mb/s and round trip time is 50 ms. Experiments were given a 5-minute stabilization period after which data was collected for the next 10 minutes. TCP runs were performed with two send buffer sizes: 64 KB and 1 MB.

Table 6.2 shows the results. In general, *CP-RUDP outperforms both TCP configurations for all metrics.* In particular, the following results are seen:

- Completion asynchrony is significantly lower for TCP with the small send buffer size than for TCP with the large send buffer size. Even so, this value is 2.7 times larger than CP-RUDP.

- Stall time is significantly less for TCP with the large send buffer size. It is still 2 times larger than CP-RUDP, however.

- End-to-end delay is least for TCP with a small buffer size. Still, it is 2 times larger than CP-RUDP.

- Normalized flowshare is higher for TCP with a large buffer size. This value is still almost 0.2 below the fair share value of 1.0. CP-RUDP achieves a value slightly greater than 1.0. These numbers are further reflected in frame ensemble rates which are 18.9 and 21.7, respectively.

- TCP with a smaller send buffer configuration shows the lowest values for frame ensemble interarrival jitter. Still, it is 2.7 times the performance value achieved by CP-RUDP.

Once again, the coordinated multi-streaming with CP-RUDP outperforms TCP substantially. It not only combines the best of both TCP send buffer configurations, it outperforms the best TCP can provide in either scenario.

Figure 6.63, Figure 6.64, and Figure 6.65 show sample flow throughput results over time for CP-RUDP, TCP configured with a small send buffer, and TCP configured with a large send buffer. Each of six flows is seen individually, with the first plot showing 5 seconds of a run and the second plot showing 15 seconds.

Figure 6.63 shows two of six CP-RUDP flows receiving consistently more bandwidth than the other four flows, following the coordination scheme described in Section 6.5. Unlike Figure 6.61 in the previous section, however, which two flows of the set changes over time. Figure 6.63 shows, for example, a transition between time 103 and 104. Once again, a high degree of synchrony is seen among flows, as well as an absence of stall events.

Figure 6.64 shows sample flow throughput results for TCP configured with a small send buffer size. Very little synchrony is seen among flows, and stall events (seen when throughput levels fall to zero) are commonplace. In Figure 6.65, we see flow throughput results for TCP configured with a large send buffer size. While overall throughput levels appear better, once again we see a lack of synchrony and numerous stall events. Several sub-intervals exist where one flow appears to dominate throughput. Each is followed by a substantial stall interval and then a subsequent interval of apparent competition among all flows.

These plots serve to illustrate the effects of flow coordination in the unequal frame size context where dynamic reconfiguration occurs periodically. The summary statistics in Table 6.2 quantify the overall outcome of these effects and generally make the case for a coordinated approach to multi-streaming in this application context.

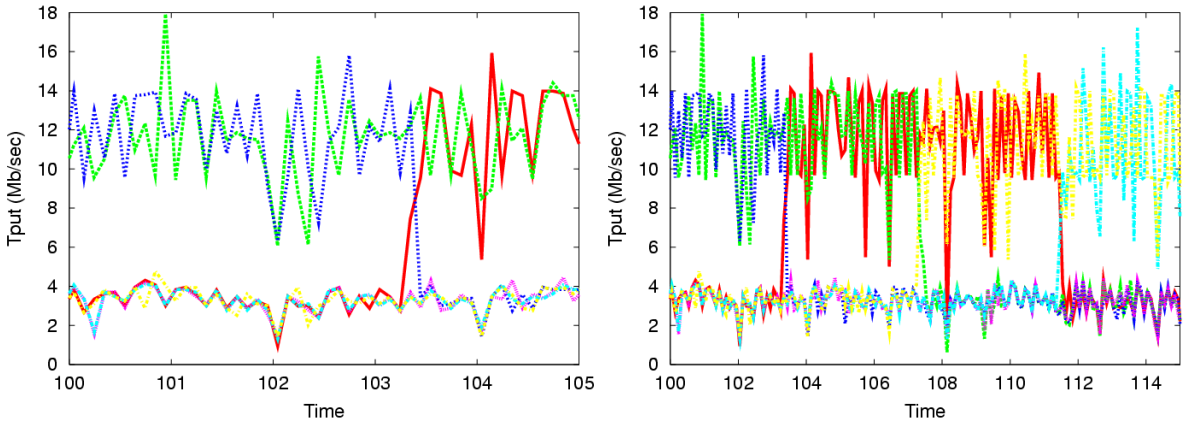


Figure 6.63: Unequal frame size with dynamic reconfiguration. Throughput over time for CP-RUDP.

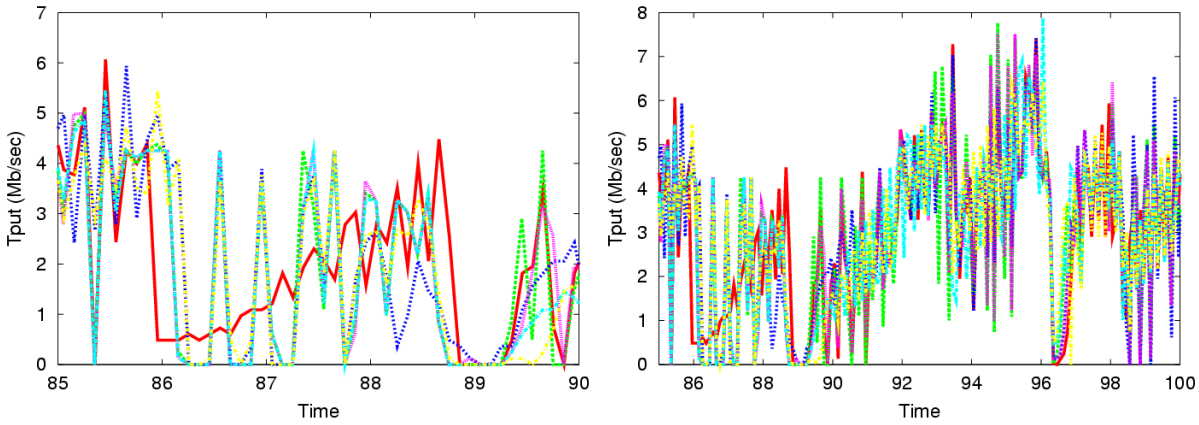


Figure 6.64: Unequal frame size with dynamic reconfiguration. Throughput over time for TCP with 64 KB send buffer configuration.

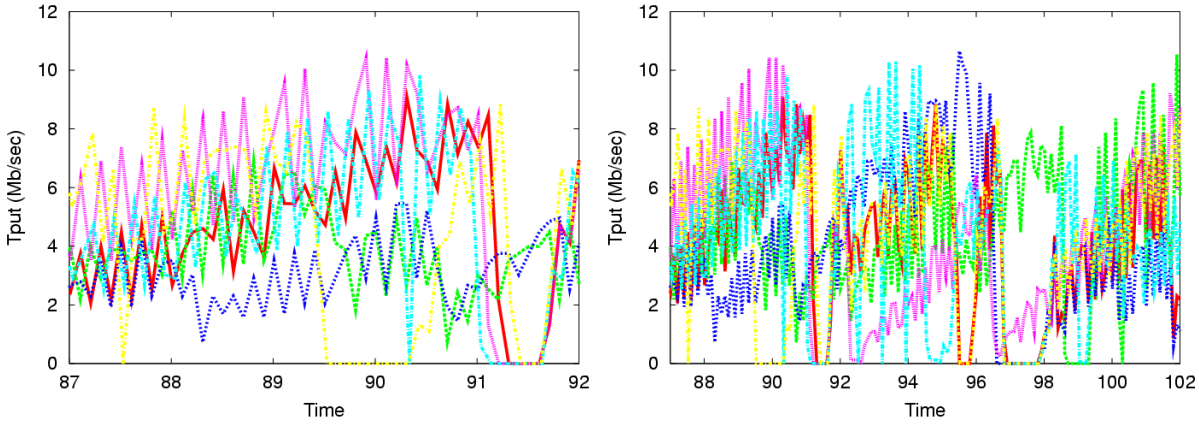


Figure 6.65: Unequal frame size with dynamic reconfiguration. Throughput over time for TCP with 1 MB send buffer configuration.

6.8.4 Summary

In this section, we compared CP-RUDP multi-streaming performance with that of TCP for scenarios in which frame size among flows is not equal. The coordination scheme described in Section 6.8 is applied by each CP-RUDP flow to scale its sending rate according to the frame size it is currently assigned. In contrast, flows in each TCP run operate in an independent manner with no higher-level coordination scheme present.

The results from Section 6.8.1 and Section 6.8.2 show CP-RUDP to be insensitive to both unequal frame size configuration and increasing differences in frame size (i.e., dispersion factor). Furthermore, CP-RUDP outperforms TCP for every metric considered in this chapter: completion asynchrony, stall time, end-to-end delay, normalized throughput, frame ensemble rate, and frame ensemble interarrival jitter. Each of these experiment sets focus on a worst case scenario: that which occurs when the same flows persistently stream larger frames than all other flows in the same application. In summary, the scenario is one which exacerbates the problems seen in Section 6.7 on equal frame sizes and shows even more clearly the benefits of flow coordination with CP-RUDP.

In Section 6.8.3, we saw that CP-RUDP could furthermore be used to coordinate multi-streaming when frame size is reconfigured among application flows dynamically. This models a 3DTI user scenario in which a user's field of interest, implemented with higher resolution video images, changes over time. Again, our results show CP-RUDP outperforming TCP in literally every metric considered in this chapter. Interestingly, the coordination scheme described in Section 6.8 needed no real changes to handle what might at first appear to be a difficult twist on the unequal frame size scenario. This is because CP-RUDP naturally provides the infrastructure for dynamically reconfiguring flow coordination on the fly.

6.9 Abilene Experiments

In this section we present experimental results taken from the Abilene backbone network comparing coordinated multi-streaming with CP-RUDP and uncoordinated streaming with TCP. These results are significant in two ways. First, they represent a chance to test out our multi-streaming scheme in a public network environment; that is, outside of our isolated laboratory testbed. In this way, they provide proof-of-concept

in an authentic context. Second, these results demonstrate CP-RUDP's performance for a significantly scaled context. Throughput levels achieved approached 200 Mb/s in these experiments, and the number of capture endpoints was scaled three to four times the original number used in our laboratory testbed.

Overall, the results underscore those seen for both equal and unequal frame size configurations in our laboratory testbed. As such, they provide additional evidence of CP's effectiveness in coordinating multi-streaming within the 3D Tele-immersion application.

6.9.1 Network configuration

The *capture cluster* (see Figure 6.2) for the Abilene experiments is located in the University of Pennsylvania GRASP [5] lab within the School of Engineering. The cluster is comprised of a collection of Dell Precision 530 workstations with dual 2.4 GHz Intel Xeon processors, 512 MB of RAM, and an Intel Pro/100 ethernet adapter (100 Mb/s) attached to a 100 MHz PCI-X bus. Each machine runs LINUX RedHat 9 (kernel version 2.4.20-8smp or 2.4.18-17.8.0smp).

Each host in the capture cluster sends packets over a local LAN to an AP with the same specifications as described in Chapter 5. This AP has a gigabit ethernet uplink (1000 Mb/s) to an Extreme Summit 1i switch which in turn has a gigabit ethernet uplink to a Juniper M10 router. This router is configured to do traffic shaping with bandwidth limiting set to 200 Mb/s and burst limiting set to 5 Mb/s. [6] A gigabit ethernet uplink leads to backbone routers within the MAGPI GigaPoP, and then an OC-48 uplink to the Abilene backbone network [Abi] administrated by Qwest.

The *reconstruction cluster* is located in the University of North Carolina's Computer Science Department. Machines consist of an array of Dell Precision 650 Workstations, each with a single 3.2 GHz Intel Xeon processor, 2 GB of main memory, and an onboard 10/100/1000 Mb/s Intel ethernet adapter. The bus architecture is PCI-X and runs at 100 MHz. Each machine runs RedHat LINUX version 9 (kernel version 2.4.20-19.9smp).

Each reconstruction host exchanges packets with a local AP (see Chapter 5 specifications) using a 100 Mb/s link. The AP has a gigabit ethernet uplink to an Enterasys Matrix E7 switch which in turn has an uplink to a Cisco 6500 router. This router is administrated by the university and represents the principal ingress/egress point

[5] GRASP is an acronym for General Robotics, Automation, Sensing and Perception.

[6] Bandwidth was leased from MAGPI GigaPoP for the purpose of these experiments.

for campus traffic. A gigabit ethernet uplink then connects this campus router to a Cisco 7609 router on the NC-REN (North Carolina Research Network) administrated by MCNC, a non-profit organization located in Research Triangle Park, NC. Within NC-REN in the RTP area, DWDM technology is used to create several gigabit ethernet "channels" between backbone routers, until finally a Cisco GSR 12410 forwards packets to the Abilene backbone network over an OC-48 uplink.

The traceroute listing below shows the hops between the capture cluster in Pennsylvania and the reconstruction cluster in North Carolina. APs in the trace are 128.91.58.1 and marina.cs.unc.edu. Round trip time is seen to be approximately 15.8 milliseconds.

```
ti04> traceroute 152.2.141.32
traceroute to 152.2.141.32 (152.2.141.32), 30 hops max, 38 byte packets
1 128.91.58.1 (128.91.58.1) 0.367 ms 0.281 ms 0.269 ms
2 EXTERNAL-GE2.ROUTER.UPENN.EDU (165.123.217.1) 0.406 ms 0.790 ms 0.396 ms
3 local.upenn.magpi.net (198.32.42.249) 0.361 ms 0.339 ms 0.338 ms
4 phl-02-01.backbone.magpi.net (216.27.100.221) 0.689 ms 0.576 ms 0.530 ms
5 remote.oc48.abilene.magpi.net (216.27.100.22) 2.969 ms 2.732 ms 2.763 ms
6 washng-nycmng.abilene.ucaid.edu (198.32.8.85) 7.093 ms 7.367 ms 7.220 ms
7 rlgh1-gw-abilene-oc48.ncren.net (198.86.17.65) 14.120 ms 14.068 ms 14.050 ms
8 rlgh7600-gw-to-rlgh1-gw.ncren.net (128.109.70.38) 14.389 ms 14.349 ms 14.303 ms
9 unc7600-gw-to-rlgh7600-gw.ncren.net (128.109.70.30) 15.556 ms 15.515 ms 15.480 ms
10 ciscokid.internet.unc.edu (128.109.36.253) 15.706 ms 15.695 ms 15.737 ms
11 marina.cs.unc.edu (152.2.137.137) 15.761 ms 15.707 ms 15.732 ms
12 lookout02a-colab.cs.unc.edu (152.2.141.32) 15.831 ms 15.814 ms 15.775 ms
```

The return path traceroute is shown below with a similar, but not identical, set of forwarding hops. Round trip time is approximately 16.1 milliseconds perhaps reflecting the addition of two hops.

```
lookout02a> traceroute 128.91.58.33
traceroute to 128.91.58.33 (128.91.58.33), 30 hops max, 38 byte packets
1 lookout-colab-router.cs.unc.edu (152.2.141.49) 0.369 ms 0.207 ms 0.218 ms
2 ciscokid-cs.net.unc.edu (152.2.31.1) 0.379 ms 0.422 ms 0.361 ms
3 unc7600.internet.unc.edu (128.109.36.254) 0.492 ms 0.496 ms 0.362 ms
4 rlgh7600-gw-to-unc7600-gw.ncren.net (128.109.70.29) 1.739 ms 2.011 ms 1.990 ms
5 rlgh1-gw-to-rlgh7600-gw.ncren.net (128.109.70.37) 1.859 ms 1.913 ms 1.745 ms
6 abilene-wash.ncni.net (198.86.17.66) 8.984 ms 8.981 ms 8.982 ms
7 nycmng-washng.abilene.ucaid.edu (198.32.8.84) 13.228 ms 13.597 ms 27.470 ms
8 local.oc48.abilene.magpi.net (216.27.100.21) 15.351 ms 15.382 ms 15.477 ms
9 phl-01-02.backbone.magpi.net (216.27.100.222) 15.853 ms 15.769 ms 15.726 ms
```

```
10 remote.upenn.magpi.net (198.32.42.250) 15.855 ms 16.007 ms 15.849 ms
11 external2-fe.router.upenn.edu (165.123.237.11) 16.116 ms 15.847 ms 16.096 ms
12 EXTERNAL3-GE.ROUTER.UPENN.EDU (165.123.237.15) 16.224 ms 15.900 ms 15.878 ms
13 ti-freebsd-router.grasp.upenn.edu (128.91.58.30) 15.882 ms 16.098 ms 15.982 ms
14 ti04-2.grasp.upenn.edu (128.91.58.33) 16.352 ms 16.021 ms 16.107 ms
```

Finally, we note that the GRASP host cluster in Pennsylvania possessed enough machines to separate frame capture hosts from competing TCP *iperf* hosts. This was not the case for North Carolina where the same set of machines acted as receivers for both contexts. We point out, however, that the flow of data was highly one-directional in that all frame and *iperf* data flows exclusively from Pennsylvania to North Carolina, and only acknowledgments flowed in the reverse direction. Therefore, there was little *sending* contention on North Carolina hosts. Furthermore, aggregate data communications for any given host at North Carolina generally did not exceed a maximum of 40 Mb/s. For this reason, we believe that using North Carolina hosts for both had very little effect on our results overall.

6.10 Abilene Results: Equal Frame Size

In this section, we look at experimental results for equal frame size ensembles over the Abilene backbone network. Frames captured by each endpoint following a trigger in this scheme are identical in size. This will occur when all endpoints use exactly the same resolution, the same encoding scheme, and avoid applying compression algorithms. This is the default mode currently used by 3DTI.

Endpoints using CP-RUDP, as described in Section 6.5, are configured to send at exactly the rate given by $NET.bw$ in the AP state table. This is the congestion responsive sending rate for a single flowshare as estimated by each AP using the TFRC equation presented in Section 2.3.2. This scheme naturally results in an even distribution of available bandwidth among flows. In addition, send buffer size is dynamically adjusted to match network conditions as described in Section 6.5.

6.10.1 Number of Streams

In this section, we compare the performance of multi-streaming with TCP and CP-RUDP over Abilene when the number of streams participating in multi-streaming varies. To study this issue, we used application configurations with 6, 9, 12, 15, 18,

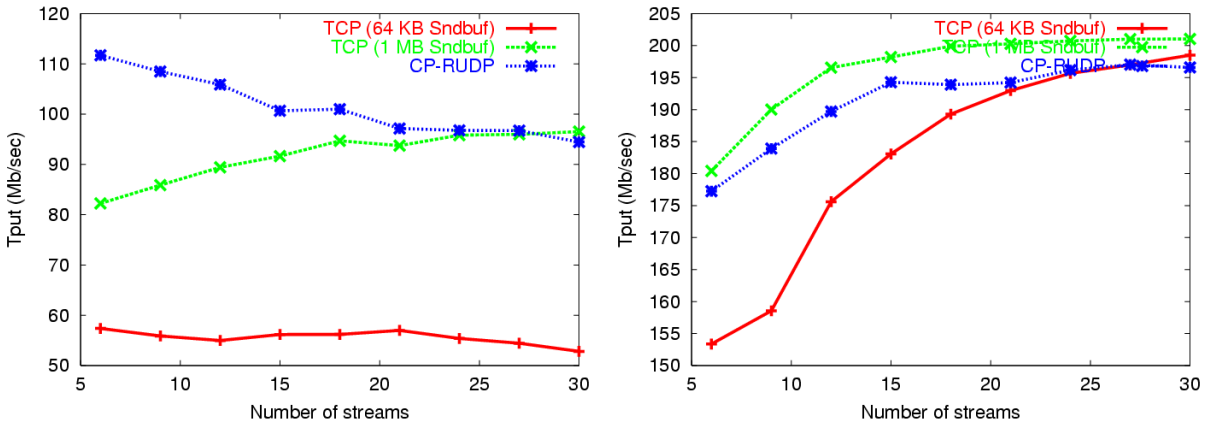


Figure 6.66: Throughput results. (a) Frame throughput and (b) aggregate throughput versus number of streams.

21, 24, 27, and 30 streams. (Six capture machines at Pennsylvania hosted multiple streams as necessary. A machine was additionally used as a trigger server.) We used a frame size of 100 KB for all runs in this experiment set. Two trials were run for each configuration. After a 90-second stabilization period, data was collected for 180 seconds.

Before looking at results for the six metrics presented in Section 6.6.2, it might first be instructive to look at the overall throughput results we achieved with our network setup. Figure 6.66 (b) shows the aggregate throughput for all experimental traffic between Pennsylvania and North Carolina for various stream number configurations. As the number of streams approaches 30, aggregate traffic levels off at the 195-200 Mb/s range. This corresponds to the bandwidth ceiling imposed by MAGPI at the University of Pennsylvania egress point.

Aggregate frame traffic is shown in Figure 6.66 (a) and depends upon the transport protocol used for multi-streaming. Both CP-RUDP and TCP with a large send buffer are in the 90-100 Mb/s level as the number of streams approaches 30. Interestingly, this level is higher for CP-RUDP with a small number of streams, but lower for TCP with a small number of streams. Throughput for TCP with a small send buffer configuration failed to exceed 60 Mb/s.

Figure 6.67 and Figure 6.68 show throughput by flow over time for sample TCP and CP-RUDP runs. A total of six streams was used for each run, and an interval of 10 seconds is presented. Figure 6.67 shows CP-RUDP flows behaving in a highly synchronous fashion with throughput levels for each flow largely coinciding with one

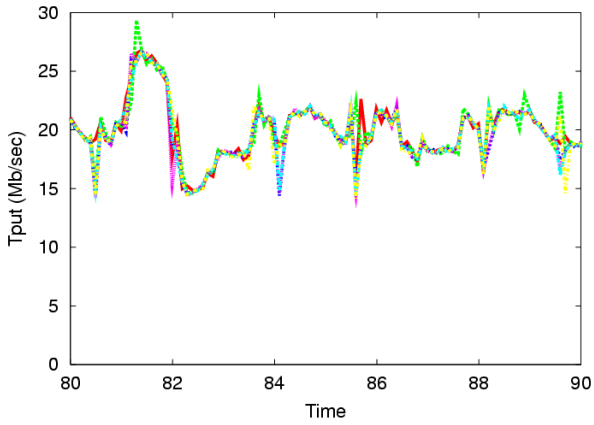


Figure 6.67: Throughput over time for 6 CP-RUDP flows.

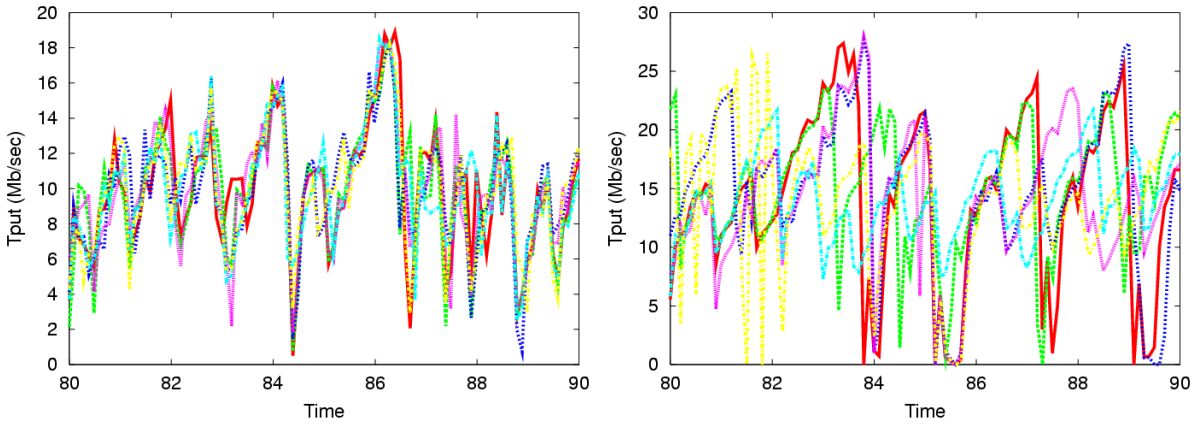


Figure 6.68: Throughput over time for (a) 6 TCP flows with 64 KB send buffer and (b) 6 TCP flows with 1 MB send buffer.

another for the entire interval. Stall events do not occur. Figure 6.68 (a) and (b) show TCP with a 64 KB and 1 MB send buffer configuration, respectively. Throughput in both plots appears to vary far more than Figure 6.67. TCP with a small send buffer configuration shows a fair amount of synchrony but significant oscillation. Interestingly, very few stall events occur, although sharp throughput drops are common. TCP with a large send buffer configuration shows greater asynchrony among flows, as well as numerous stall events.

Figure 6.69, Figure 6.70, and Figure 6.71 show results for the six metrics presented in Section 6.6.2. Figure 6.69 (a) shows the results for completion asynchrony. CP-RUDP shows the lowest completion asynchrony of all three experiment sets, with value rising only slightly as the number of streams increases to 30. TCP with a small send buffer configuration is only slightly higher, with a similarly small increase. TCP with a large send buffer size exhibits far higher completion asynchrony, with values that increase significantly as the number of streams increases.

Figure 6.69 (b) shows stall time results. CP-RUDP maintains low stall time values throughout, though interestingly, TCP with a large send buffer configuration shows even less. In contrast, stall time is significantly higher for TCP configured with a small send buffer configuration. This is particularly true as the number of streams exceeds 20.

In Figure 6.70 (a), we see CP-RUDP and TCP with a small send buffer configuration achieving similarly low end-to-end delay values for nearly all stream number configurations. As the number of streams increase, these values increase very slowly. TCP with a large send buffer configuration, in contrast, shows significantly higher end-to-end delay for all stream number configurations, with values increasing much more sharply as the number of streams increase.

Figure 6.70 (b) shows TCP configured with a large send buffer getting only slightly less than its fair share in the context of normalized throughput. Values seem to approach 1.0 as the number of streams increases to 30. CP-RUDP shows values that are significantly above fair while the number of streams is less than or equal to 12. Values significantly improve beyond that point and approach 1.0 as the number of streams increases to 30. The problem of over-aggressive behavior for small stream number configurations is, we expect, related to the TFRC problem mentioned in Section 6.7.5. The problem occurs when very little loss in the system drives the value of p in Equation 4.1 very close to zero. As p approaches zero in the denominator, bandwidth estimation

results tend toward infinity. To counter this problem, a default maximum rate is used until estimation results can improve. This value may not, however, be suitably matched to the properties of a particular cluster-to-cluster data path. In this case, it apparently results in somewhat more aggressive behavior than TCP. Finally, we note that TCP with a small send buffer size configuration gets significantly less than its fair share of bandwidth on an average, and values deteriorate significantly as the number of streams approach 30.

Frame ensemble rate results in Figure 6.71 (a) parallel those of normalized flow-share in Figure 6.70 (b). TCP configured with a large send buffer shows somewhat lower values than CP-RUDP for small stream numbers. As the the number of streams approaches 30, however, the frame ensemble rate becomes identical between the two experiment sets. TCP configured with a small buffer, in contrast, receives much lower values for all configurations. For all three runs, the frame ensemble rate generally decreases as the number of streams increases. This is to be expected as more flows require a portion of available resources for streaming and send rates for each stream are reduced in the process.

Finally, Figure 6.71 (b) shows frame ensemble interarrival jitter values as the number of participating streams increases. Values are the highest and rise the most for TCP configured with a large send buffer. Values are similar for CP-RUDP and TCP with a small sender buffer size for stream numbers up to 18. (TCP shows somewhat less jitter for some configurations.) After 18, however, jitter for TCP rises more than with CP-RUDP.

Overall, CP-RUDP performance is comparable or better than the best TCP configuration for any given metric. Like TCP configured with a small send buffer, CP-RUDP shows very low completion asynchrony numbers, very low end-to-end delay numbers, and fairly low frame ensemble interarrival jitter values. Like TCP configured with a large send buffer, however, CP-RUDP shows low stall time numbers, good network fairness and utilization (for 15 streams and higher), and good frame ensemble rates. What is important to note once again, however, is that *CP-RUDP provides all of these advantages simultaneously.* This is not possible with any single TCP configuration which must balance the inherent tradeoffs described in latter part of Section 6.6.3.

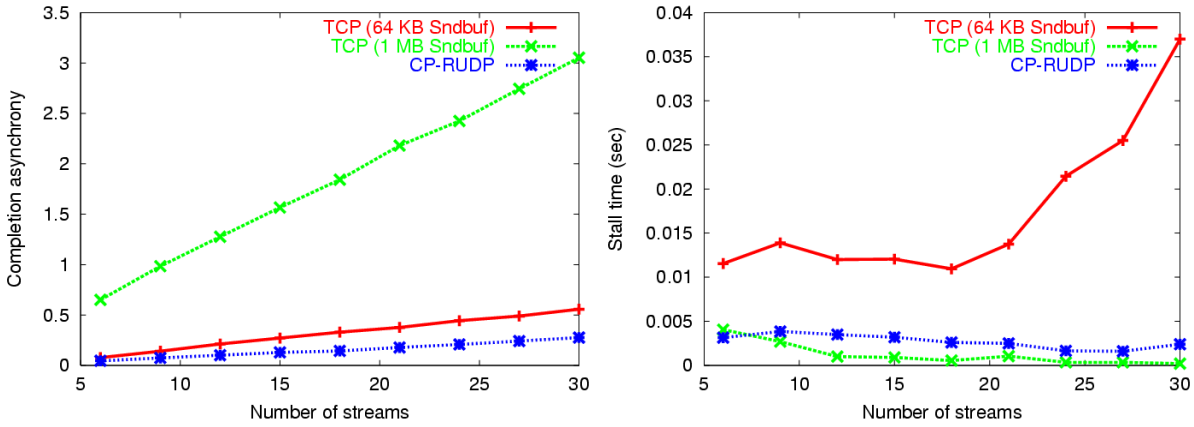


Figure 6.69: Number of streams results (Abilene). (a) Completion asynchrony and (b) stall time versus number of streams.

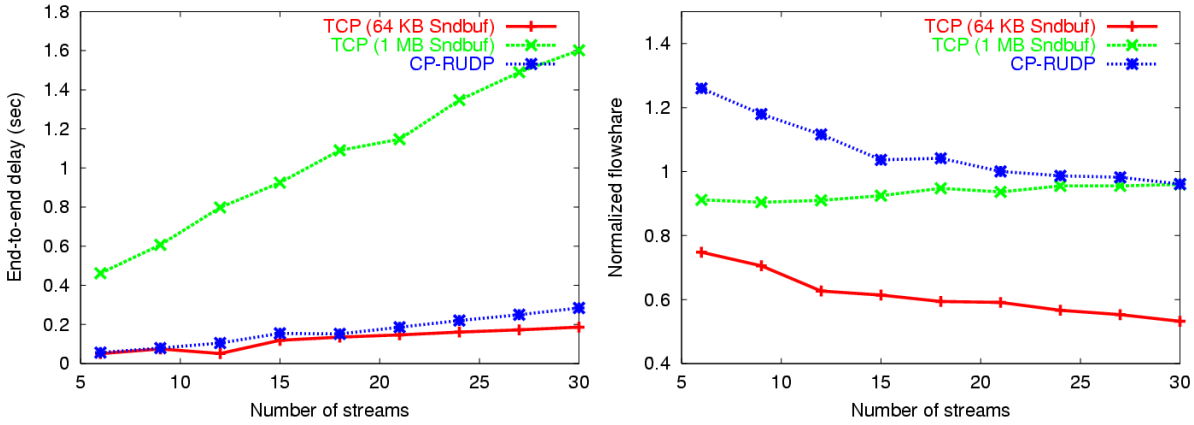


Figure 6.70: Number of streams results (Abilene). (a) End-to-end delay and (b) normalized flowshare versus number of streams.

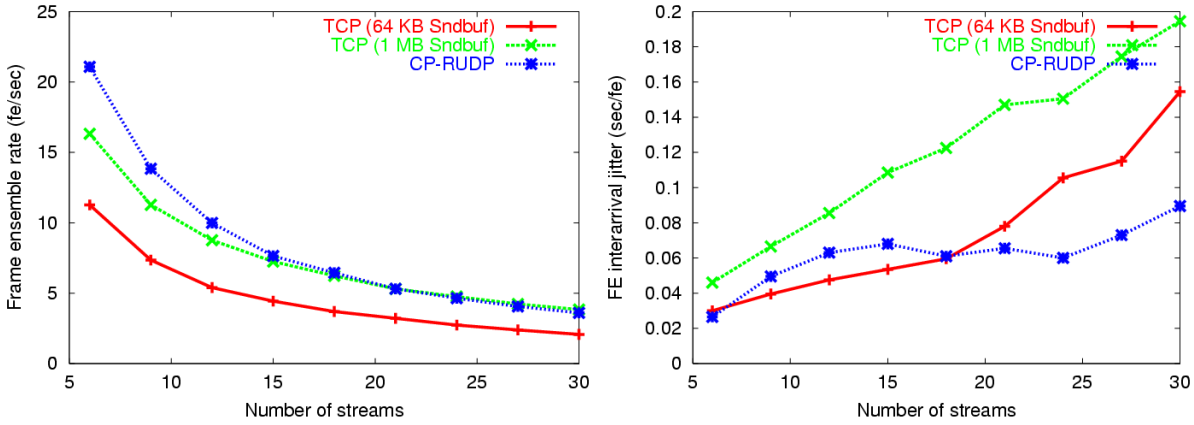


Figure 6.71: Number of streams results (Abilene). (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus number of streams.

6.10.2 Frame Size

In this section we compare the performance of multi-streaming with TCP and CP-RUDP over the Abilene backbone network using various frame sizes. To clarify, frames within the same ensemble are always equal in size across all streams, as are frames from one ensemble to the next within the same stream. Studying the effect of frame size entails, rather, changing the default frame size from one run to the next and comparing performance results for different runs.

Frame sizes used in this experiment set include 10, 25, 50, 100, 150, and 200 KB. 24 streams were used throughout, and results were averaged from two trials. Each trial was given a 90-second stabilization period and then data was collected for 180 seconds. Results are shown in Figure 6.72, Figure 6.73, and Figure 6.74.

Completion asynchrony results in Figure 6.72 (a) show CP-RUDP getting the lowest values for all configurations. Values for TCP configured with a small send buffer size are quite low as well, and in stark contrast to TCP large send buffer size results. In general, completion asynchrony increases only somewhat as frame size increases.

Figure 6.72 (b) shows TCP configured with a large send buffer size once again getting the lowest stall time averages. CP-RUDP values are nearly as low, however, and do not increase as frame size increases. In contrast, TCP configured with a small send buffer size shows much larger stall time values and increases considerably as frame size increases. Values for the 200 KB configuration are more than 15 times larger than for CP-RUDP.

Results for end-to-end delay are shown in Figure 6.73 (a). Values remain consistently low for CP-RUDP but increase somewhat as frame size increases. TCP configured with a small send buffer gets similarly low values throughout, but differs from CP-RUDP in that values do not increase as frame size increases. As mentioned in Section 6.7.4, the latter behavior is not what it appears. A limitation in our delay measurement methodology makes it difficult to quantify end-to-end delay precisely as frame sizes become much larger than send buffer sizes. Hence, delay values are shown in the plot as leveling off when, in fact, end-to-end delay may increase somewhat. Values for TCP with a large send buffer size are significantly larger than either of the other two configurations and grow slightly as the frame size configuration grows.

Figure 6.73 (b) shows normalized flowshare results. Both CP-RUDP and TCP with a large send buffer configuration maintain values that are very close to 1.0 for all configurations. In contrast, TCP configured with a small send buffer shows significantly

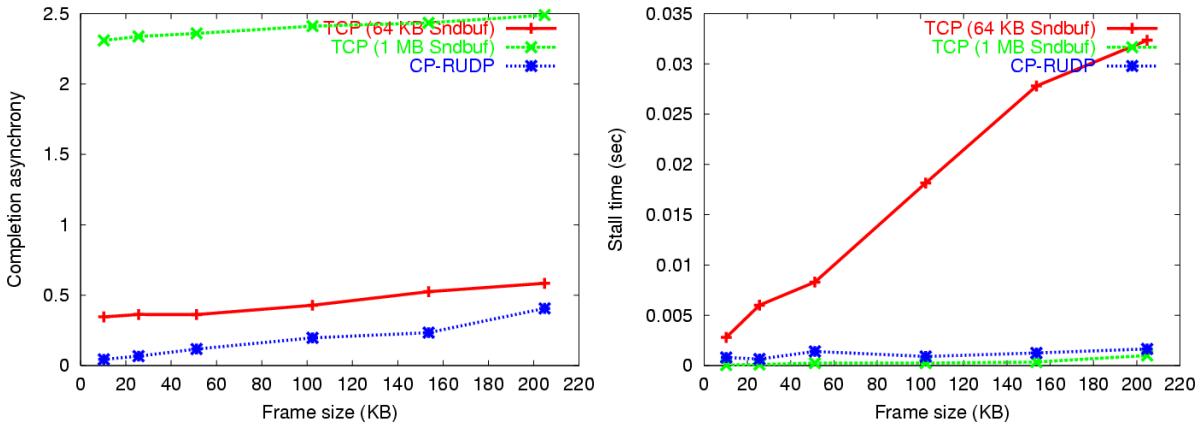


Figure 6.72: Frame size results (Abilene). (a) Completion asynchrony and (b) stall time versus frame size.

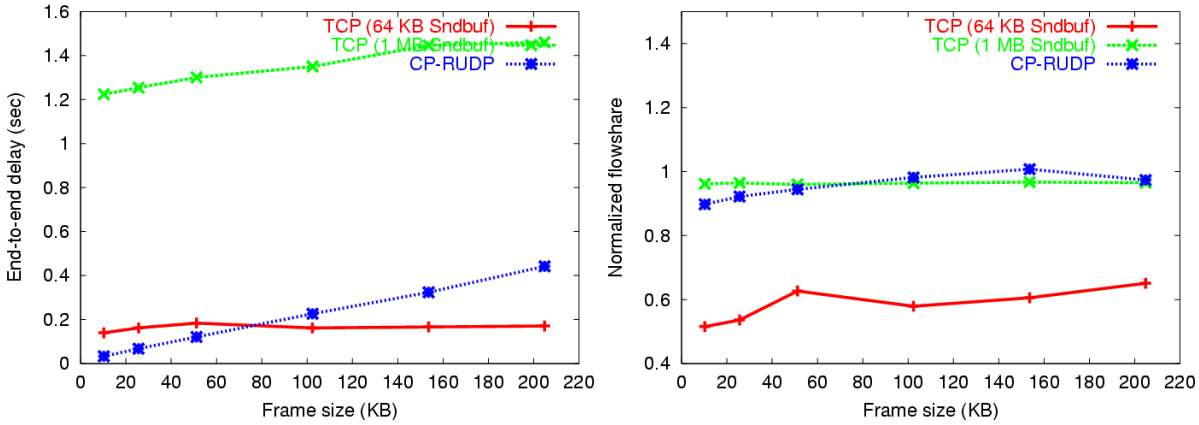


Figure 6.73: Frame size results (Abilene). (a) End-to-end delay and (b) normalized flowshare versus frame size.

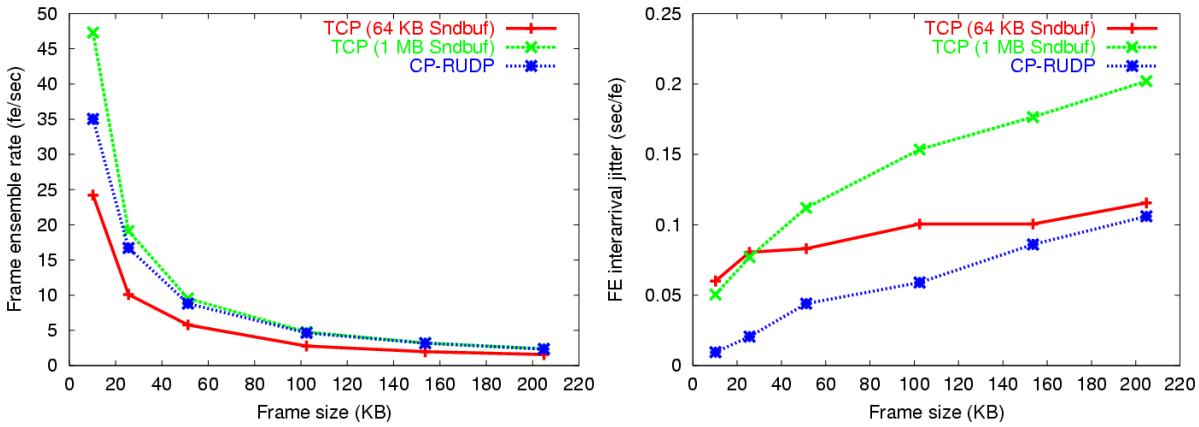


Figure 6.74: Frame size results (Abilene). (a) Frame ensemble rate and (b) frame ensemble interarrival jitter versus frame size.

low values (0.5-0.65) indicating that streams utilize much less bandwidth than they are entitled to. Frame ensemble rate results in Figure 6.74 (a) reflect these utilization results. CP-RUDP and TCP with a large send buffer configuration generally show very similar frame ensemble rates, but with TCP doing somewhat better for only the very smallest frame sizes. TCP with a small send buffer configuration achieves frame ensemble rates that are significantly lower for all configurations.

Finally, Figure 6.74 (b) shows CP-RUDP outperforming both TCP configurations for frame interarrival jitter. Values are seen to increase as frame size increases, but remain beneath TCP. TCP configured with a large sending buffer shows the most interarrival jitter, with values increasing significantly as frame size increases. TCP with a small sending buffer shows significantly large values that increase only slightly as frame size increases.

6.11 Abilene Results: Unequal Frame Size

In the next two sections we look at multi-streaming performance results over the Abilene backbone network for unequal frame size ensembles. As described in Section 6.8, such ensembles are comprised of some combination of large and small frames. This may occur in the 3DTI application when a different capture resolution is used for frames within a user's field of interest.

Section 6.11.1 focuses on the worst case scenario: the case when the same set of streams continually have more data to send than the remaining streams in the same application. To study this scenario, we chose several frame size configurations (namely, *Mmmmmm*, *MMMmmm*, and *MMMMMm*) and varied the frame size dispersion factor for each configuration. Section 6.8 describes this configuration in greater detail.

In Section 6.11.1, we consider the scenario in which streams change their frame size dynamically during multi-streaming. That is, for a given unequal frame size configuration, which frame size is streamed by a particular endpoint changes over time. Within 3DTI, this models a user's changing field of interest in the context of differing capture resolutions.

6.11.1 Frame Size Dispersion

Experiments in this section explore the effect of unequal frame size where application streams maintain a static configuration throughout the run. The unequal frame size configurations considered were chosen from the set presented in Section 6.8 and include *Mmmmmm*, *MMMmmm*, and *MMMMMm*. To map this scheme onto a scaled number of streams, the number of flows chosen was a multiple of six and the same configuration was simply applied multiple times. Throughout the experiments in this section, we used a total of 18 streams.

Like Section 6.8.1, we looked at the performance for these frame size configurations as the frame size dispersion factor increased. A mean frame size of 25 KB was used, with an ensemble size constant of 6 and a dispersion parameter that ranged between 0.1 and 0.5. Step size for the latter was set to be 1.0.

Once again, a 90-second stabilization interval was used before collecting performance data for 180 seconds. The results of two trials are averaged. Two TCP configurations are again studied, the first with a 1 MB send buffer size and the second with a 64 KB send buffer size. As explained in Section 6.6.3, results for intermediate sized send buffers would simply lie between the extremes explored in these experiments.

Figure 6.75 (a), Figure 6.78 (a), and Figure 6.81 (a) show completion asynchrony results. For all unequal frame size configurations, CP-RUDP values remain distinctively low and insensitive to increases in frame size dispersion. While both TCP configurations likewise appeared insensitive, values were significantly higher. For TCP with a smaller send buffer configuration, values were approximate 5 times larger. For TCP with a large send buffer configuration, values were more than 31 times higher.

Stall time results in Figure 6.75 (b), Figure 6.78 (b), and Figure 6.81 (b) differ somewhat. In the *Mmmmmm* results of Figure 6.75 (b), CP-RUDP maintains very low values that are insensitive to increases in frame dispersion. Meanwhile, TCP configurations show much larger values that increase significantly with the level of dispersion. In the *MMMmmm* results of Figure 6.78 (b) and the *MMMMMm* results of Figure 6.81 (b), however, all configurations appear somewhat less sensitive to increases in frame size dispersion and TCP with a large send buffer maintains the smallest stall time values. Notice that values, in general, are much lower than the *Mmmmmm* unequal frame size configuration.

In Figure 6.76 (a), Figure 6.79 (a), and Figure 6.82 (a) CP-RUDP receives the smallest end-to-end delay values. These values do not increase with the frame size

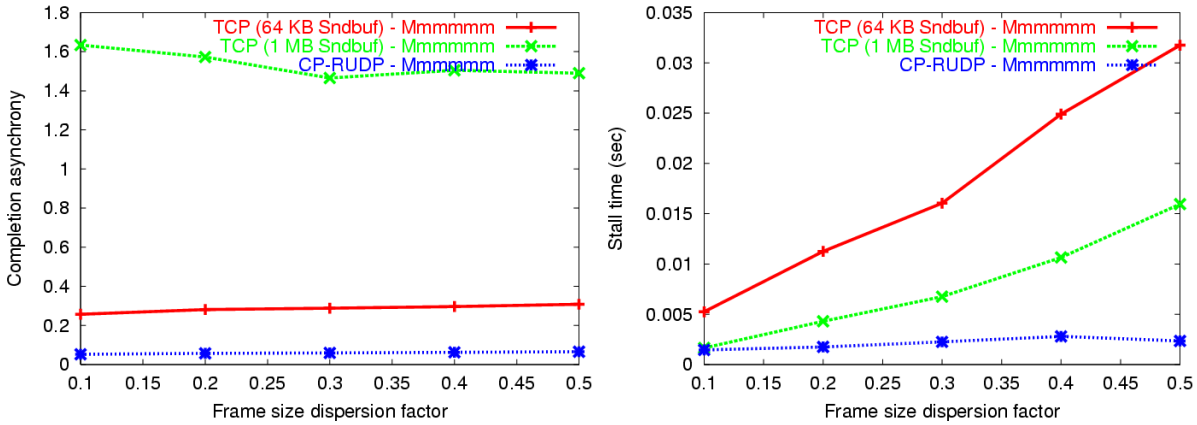


Figure 6.75: Unequal frame size results (Abilene). (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

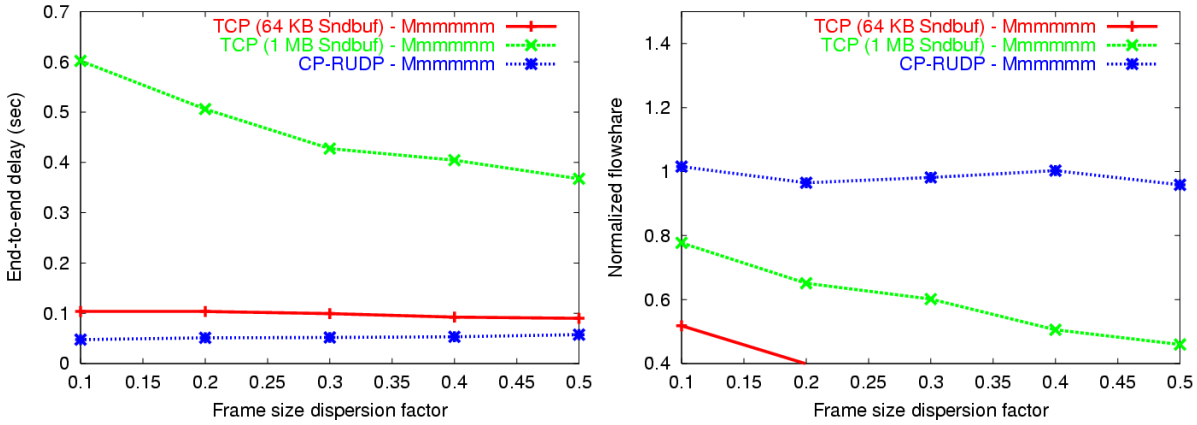


Figure 6.76: Unequal frame size results (Abilene). (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

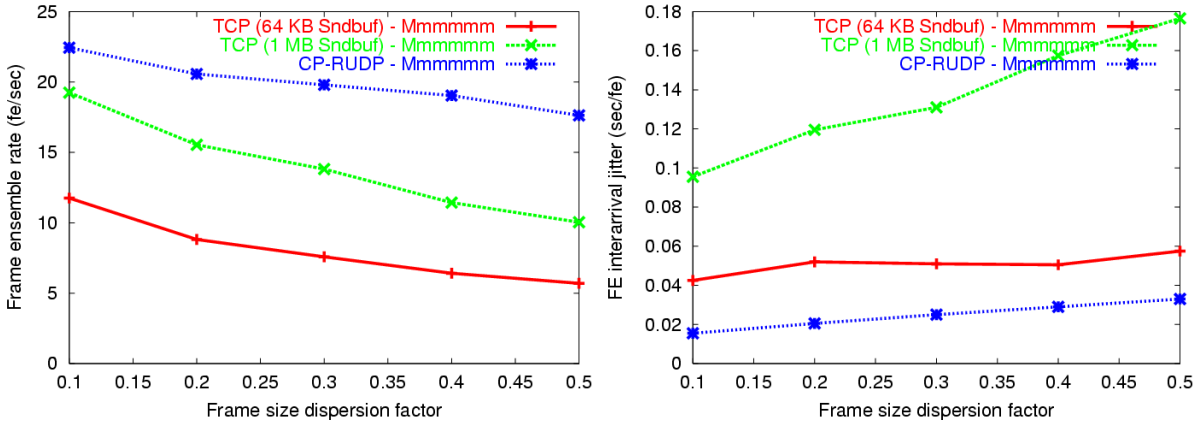


Figure 6.77: Unequal frame size results (Abilene). (a) Frame ensemble rate and (b) frame interarrival jitter versus frame size dispersion factor.

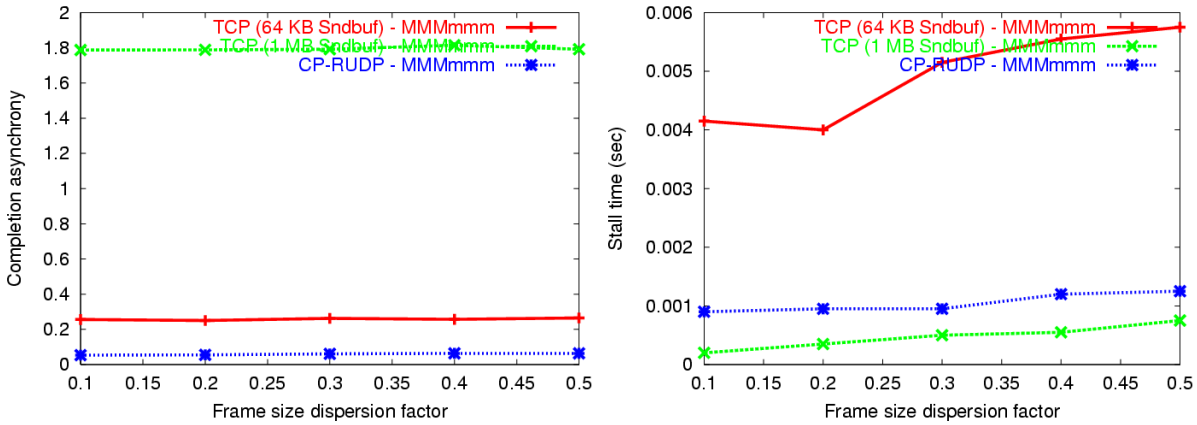


Figure 6.78: Unequal frame size results (Abilene). (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

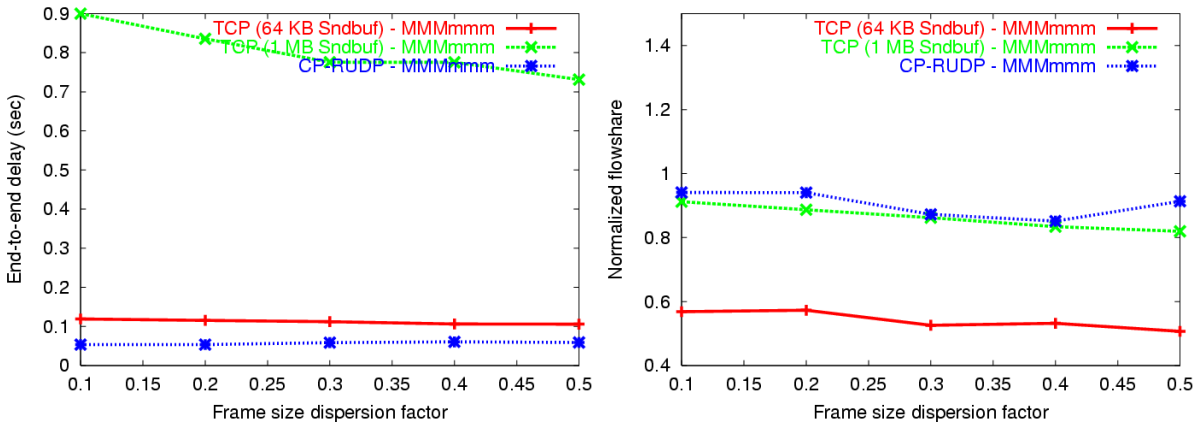


Figure 6.79: Unequal frame size results (Abilene). (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

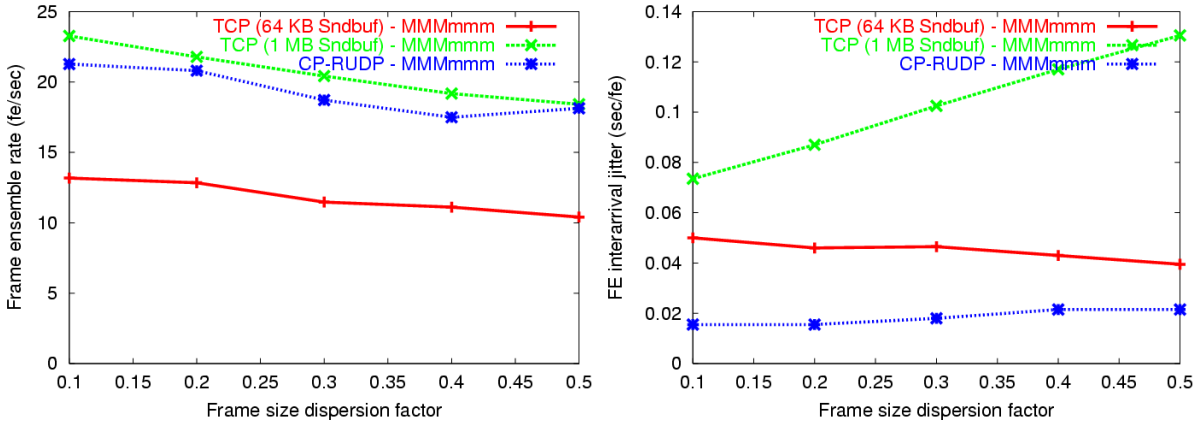


Figure 6.80: Unequal frame size results (Abilene). (a) Frame ensemble rate and (b) frame interarrival jitter versus frame size dispersion factor.

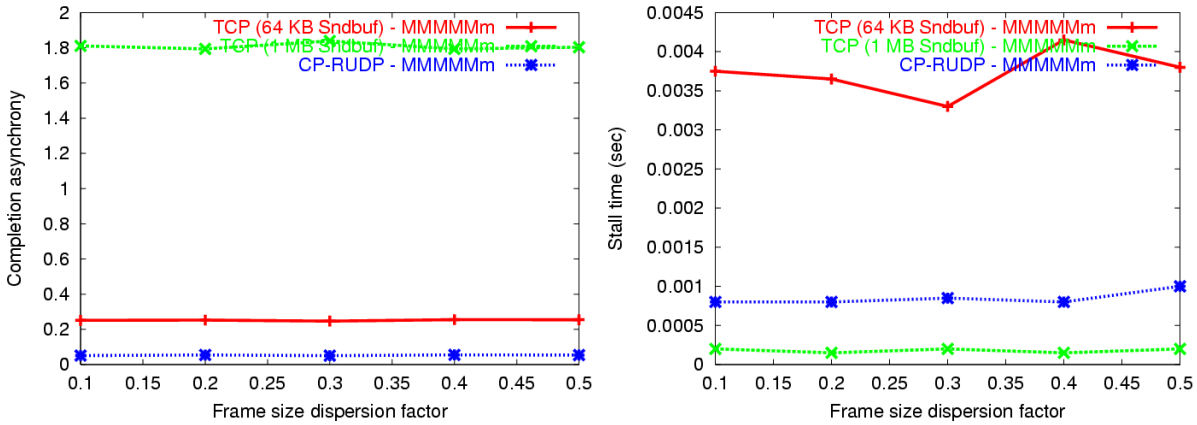


Figure 6.81: Unequal frame size results (Abilene). (a) Completion asynchrony and (b) stall time versus frame size dispersion factor.

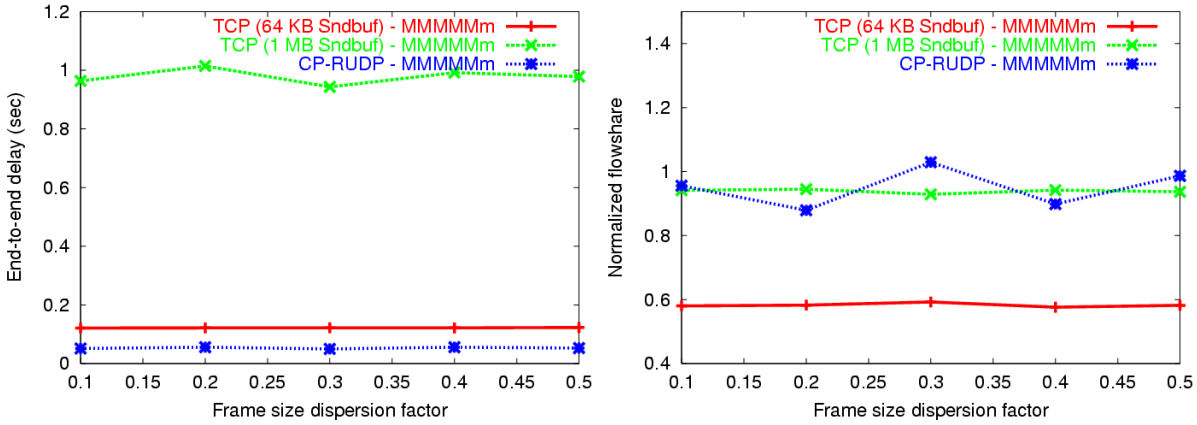


Figure 6.82: Unequal frame size results (Abilene). (a) End-to-end delay and (b) normalized flowshare versus frame size dispersion factor.

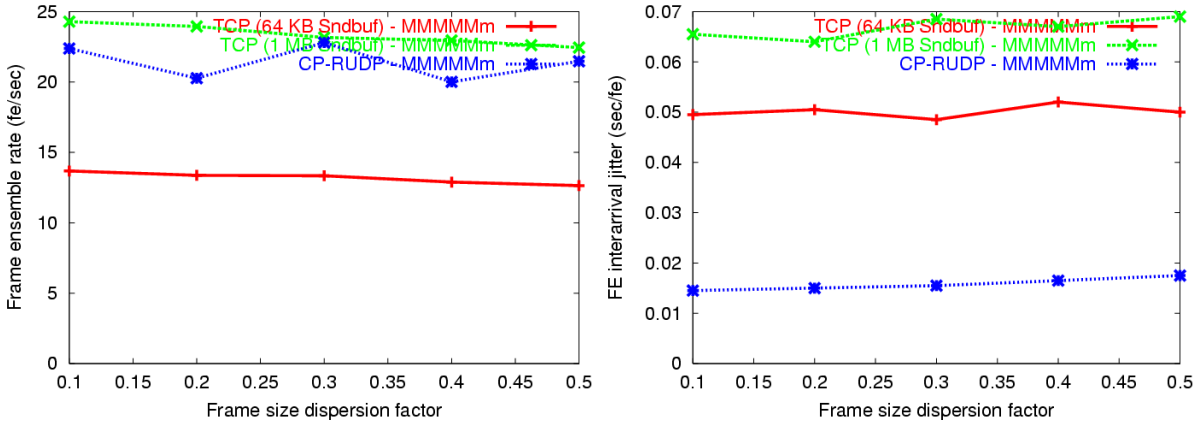


Figure 6.83: Unequal frame size results (Abilene). (a) Frame ensemble rate and (b) frame interarrival jitter versus frame size dispersion factor.

dispersion factor. TCP configured with a small send buffer maintains similarly small delay values while TCP configured with a large send buffer shows significantly higher values. The results of all unequal frame size configurations are very similar.

Figure 6.76 (b), Figure 6.79 (b), and Figure 6.82 (b) show normalized flowshare results. All unequal frame size configurations show CP-RUDP receiving very close to an ideal value of 1.0. In Figure 6.76 (b), TCP configured with a large send buffer receives significantly less bandwidth on average than CP-RUDP, while in Figure 6.79 (b) and Figure 6.82 (b), values are only slightly less. Similarly, TCP configured with a small send buffer shows exceedingly poor numbers for Figure 6.76 (b) (in fact, they fall out of the lower range shown on the plot) and poor but better numbers for Figure 6.79 (b) and Figure 6.82 (b).

These results are broadly reflected in the frame ensemble rate plots of Figure 6.77 (a), Figure 6.80 (a), and Figure 6.83 (a). In the first set, CP-RUDP achieves significantly better rates than TCP configured with a large send buffer which, in turn, achieves significantly better rates than TCP configured with a small send buffer. In the remaining cases, rates for CP-RUDP and TCP with a large send buffer are more similar, with the latter receiving only slightly better numbers. Both however, greatly outperform TCP configured with a small send buffer. All values decrease somewhat as the frame dispersion factor increases.

Finally, Figure 6.77 (b), Figure 6.80 (b), and Figure 6.83 (b) show CP-RUDP exhibiting the least frame ensemble interarrival jitter for all dispersion values. TCP with a small send buffer configuration shows higher values, and TCP with a large send buffer configuration shows even higher values. Values for the latter also increase significantly as the frame dispersion factor increases for Figure 6.77 (b).

Plots in Figure 6.84, Figure 6.85, and Figure 6.86 may be helpful in understanding the underlying behavior producing the performance results discussed in this section. Each shows flow throughput over time for a sample run, with the first plot presenting a 10 second time window and the second plot presenting a 50 second time window. The unequal frame size configuration used was *Mmmmmm*.

In Figure 6.84, we see CP-RUDP flows producing a fairly synchronized set of throughput levels. Large and small frame size streams are clearly represented in the plot, with throughput levels for individual flows largely coinciding. Note, however, the periodic drop in bandwidth across all flows. This was not seen in the laboratory testbed results of Section 6.8.2 and presents something of a mystery. It helps in explaining,

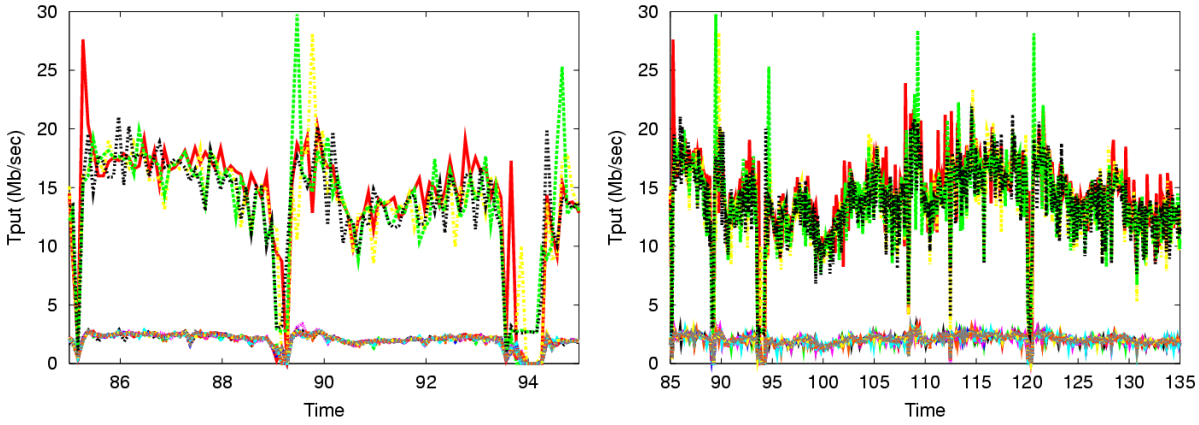


Figure 6.84: Unequal frame size (Abilene). Throughput over time for CP-RUDP.

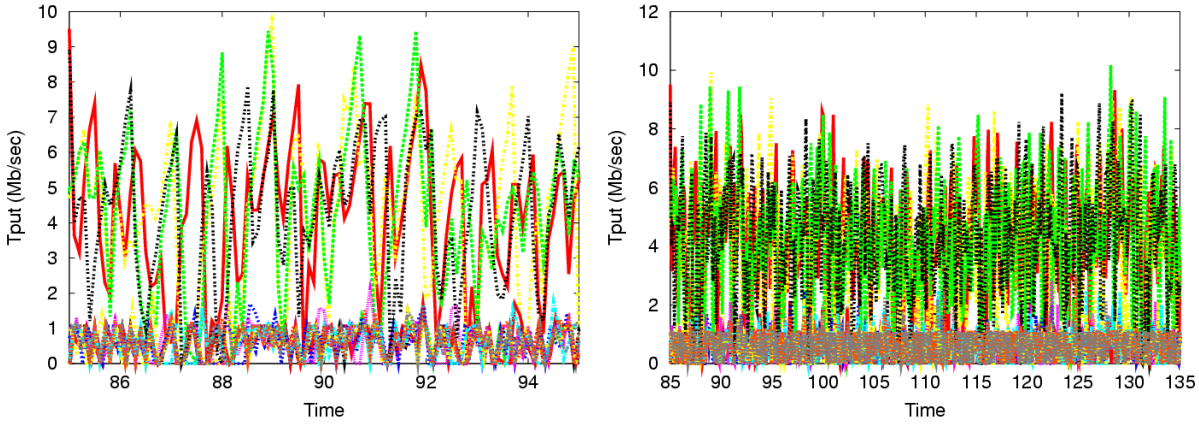


Figure 6.85: Unequal frame size (Abilene). Throughput over time for TCP with 64 KB send buffer.

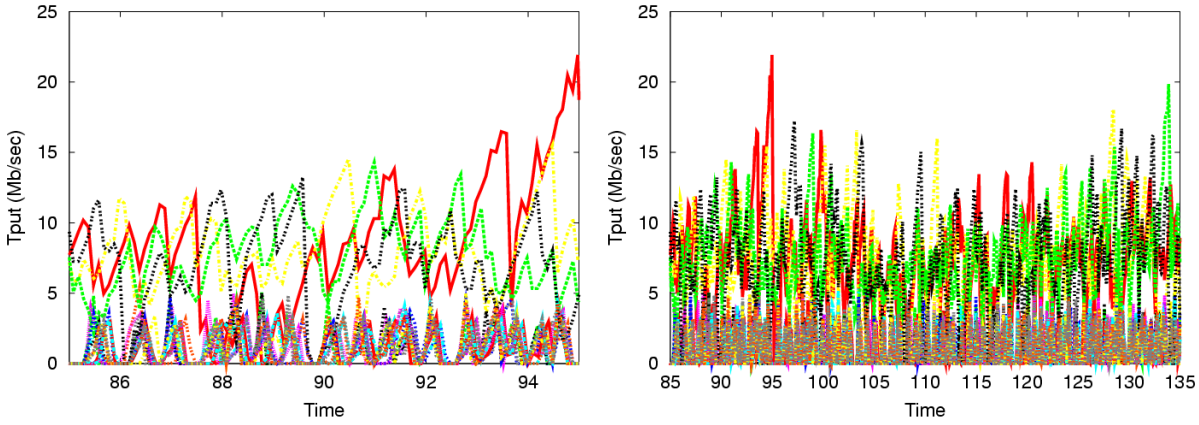


Figure 6.86: Unequal frame size (Abilene). Throughput over time for TCP with 1 MB send buffer.

Metric	*TCP (64 KB Sndbuf)*	*TCP 1 MB Sndbuf*	*CP-RUDP*
Completion Asynchrony	0.244000	1.765000	0.053500
Stall Time	0.003400	0.000200	0.002550
End-to-end Delay	0.119500	0.970100	0.047950
Normalized Flowshare	0.585000	0.945000	0.994500
Frame Ensemble Rate	13.698500	24.538000	21.585000
FE Interarrival Jitter	0.097000	0.329500	0.040500

Table 6.3: Unequal frame size with dynamic reconfiguration (Abilene).

however, the presence of some stalling seen in CP-RUDP and the odd inversion of frame ensemble rates between CP-RUDP and TCP with a large send buffer for some unequal frame size configurations.

Plots in Figure 6.85 show what appears to be an unsynchronized but bimodal set of throughput values for streams using TCP with a small send buffer configuration. Significant stalling among small frame size streams is seen as large frame size streams compete with one another. Note the large oscillations present in the latter.

Plots in Figure 6.86 illustrate a very similar phenomenon for TCP configured with a large send buffer. Competition among flows, large oscillations in bandwidth, and frequent stalling among small frame size flows characterize the plot.

6.11.2 Dynamic Reconfiguration

In this section, we compare briefly the performance of TCP multi-streaming over Abilene with that of CP-RUDP using unequal frame sizes that change among endpoints *dynamically* over time. Once again, the *MMmmmm* unequal frame size configuration was chosen for these experiments, with a 25 KB mean and maximum dispersion factor of 0.5. For each 100 frames, flows modify their frame size by trading frame size configuration with a peer flow, meanwhile maintaining the invariant that, for every set of six flows, two flows are streaming large frames at all times and four are streaming small frames. This dynamic re-configuration models a user's changing field of interest within the application, and the use of higher resolutions for streams that are currently within that field.

Experiments were given a 90-second stabilization interval before collecting performance data for 180 seconds. TCP runs were performed with two send buffer sizes: 64 KB and 1 MB.

Table 6.3 shows the results which may be summarized as follows:

- CP-RUDP has the lowest completion asynchrony. TCP with a small send buffer size is 4.7 times as large and TCP with a large buffer size is more than 33 times as large.
- Stall time is least for TCP with a large send buffer size configuration. In general, values are fairly low for all three streaming configurations.
- CP-RUDP shows the lowest end-to-end delay value. TCP values are 2.5 (small send buffer) and 20 (large send buffer) times larger.
- Normalized flowshare is very near 1.0 for CP-RUDP (.9945). The value for TCP with a large send buffer size configuration is likewise close to 1.0 (.9701). In contrast, TCP with a small send buffer size configuration shows a very low value (.5850).
- TCP with a large send buffer configuration receives the highest frame ensemble rate (24.5). CP-RUDP is similar (21.6) while TCP with a small send buffer configuration shows a significantly lower value (13.7).
- CP-RUDP shows the smallest frame ensemble arrival jitter value (.041) while TCP with a smaller send buffer configuration shows a similarly low value (.097). TCP with a large send buffer configuration shows a significantly larger value (.329).

It is indeed a curiosity that while CP-RUDP shows the best network utilization, as seen in the normalized flowshare results, it does not manage to receive the best frame ensemble rate values. Throughout our laboratory testbed results, the two metrics appeared directly related.

Upon closer examination, Figure 6.66 showed the maximum aggregate bandwidth achievable by CP-RUDP to be somewhat less than TCP with a large sender buffer configuration. Aggregate throughput numbers taken during the runs in this section would seem to support this case: for TCP configured with a large send buffer, approximately 199.6 Mb/s was achieved while for CP-RUDP runs, only 186.9 Mb/s was achieved. (The aggregate value includes both multi-streaming traffic and competing *iperf* traffic.)

In addition, we note the periodic drops in bandwidth seen in Figure 6.84 and Figure 6.87. We suspect a packet drop interaction between CP-RUDP and MAGPI's Juniper M10 traffic shaper in Philadelphia. (See Section 6.9.1.) This is supported by the fact that we saw no such effect in our laboratory testbed where throughput limits were a function of maximum link speed and not artificial rate limiting.

Still, it should be pointed out that background traffic on the Abilene backbone network during the time of these experiments was uncontrolled. It might well have been the case that TCP and CP-RUDP runs experienced different conditions leading to overall differences in aggregate throughput and, hence, differences in frame ensemble rates. If this is were the case, then CP-RUDP's normalized flowshare numbers (0.9945) convince us that CP-RUDP performed as well as could be expected given the conditions at the time.

Figure 6.87, Figure 6.88, and Figure 6.89 sample flow throughput results over time for CP-RUDP and two TCP configurations. 18 flows were used and the unequal frame size configuration was *Mmmmmm*. (Large and small frame sizes are traded dynamically between streaming hosts.) The first plot shows 10 seconds of a run while the second plot shows 50 seconds of the same run.

We see in Figure 6.87 the familiar throughput synchronization patterns of CP-RUDP, with distinctive small and large frame size streams. Both plots show streams that change from high throughput (large frame size) to low throughput (small frame size) sending modes and vice versa. Also seen are periodic drops in bandwidth. One wonders whether this is caused by traffic shaping on the Juniper M10 router as traffic is forwarded from MAGPI in Pennsylvania to the Abilene backbone.

Plots in Figure 6.88 show TCP configured with a small send buffer and plots in Figure 6.89 show TCP configured with a large send buffer. Streams show a markedly different character: complete lack of synchrony between streams, significantly large oscillations among all flows, and numerous stall events.

6.12 Abilene Results Summary

In this section, we have compared CP-RUDP multi-streaming performance with that of TCP over the Abilene backbone network between the University of Pennsylvania in Philadelphia and the University of North Carolina in Chapel Hill. Not only does this provide an authentic network context for proof-of-concept results, it tests the scalability

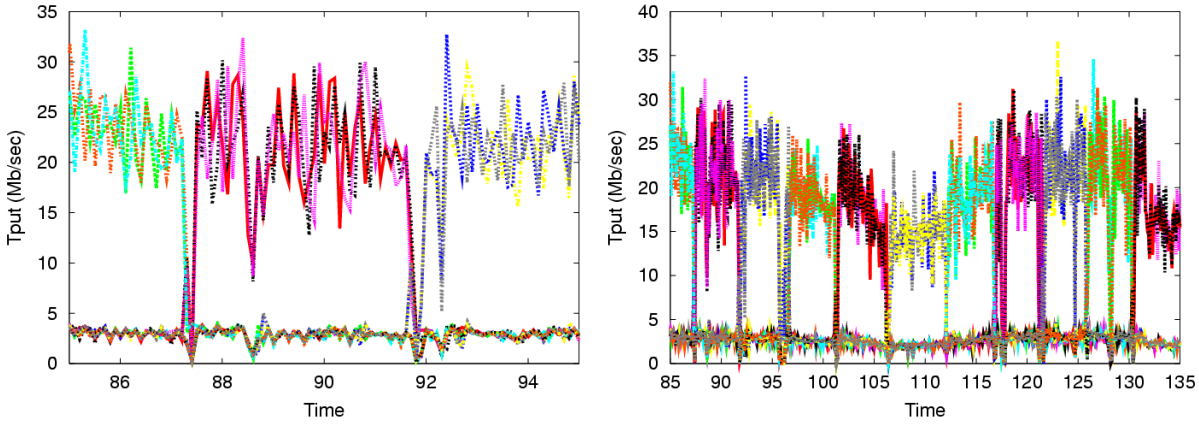


Figure 6.87: Unequal frame size with dynamic reconfiguration (Abilene). Throughput over time for CP-RUDP.

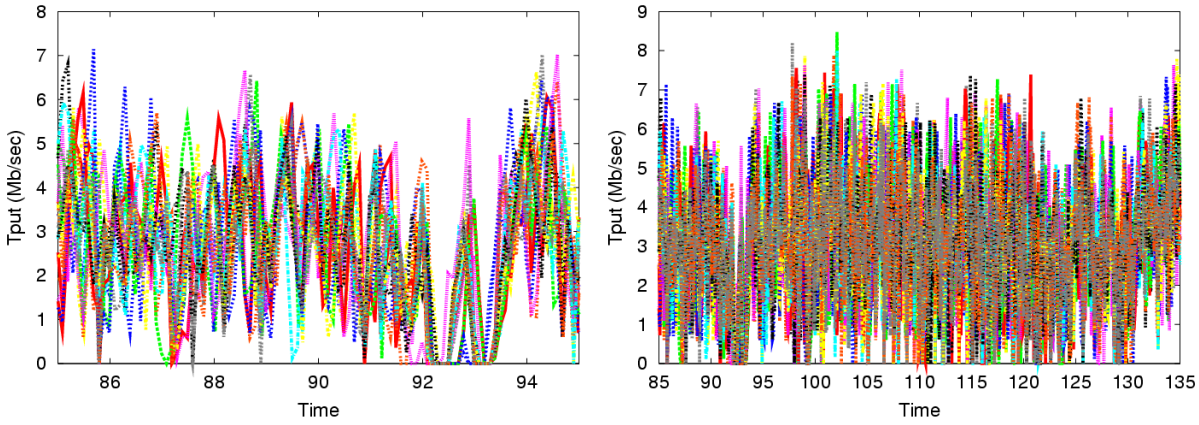


Figure 6.88: Unequal frame size with dynamic reconfiguration (Abilene). Throughput over time for TCP with 64 KB send buffer.

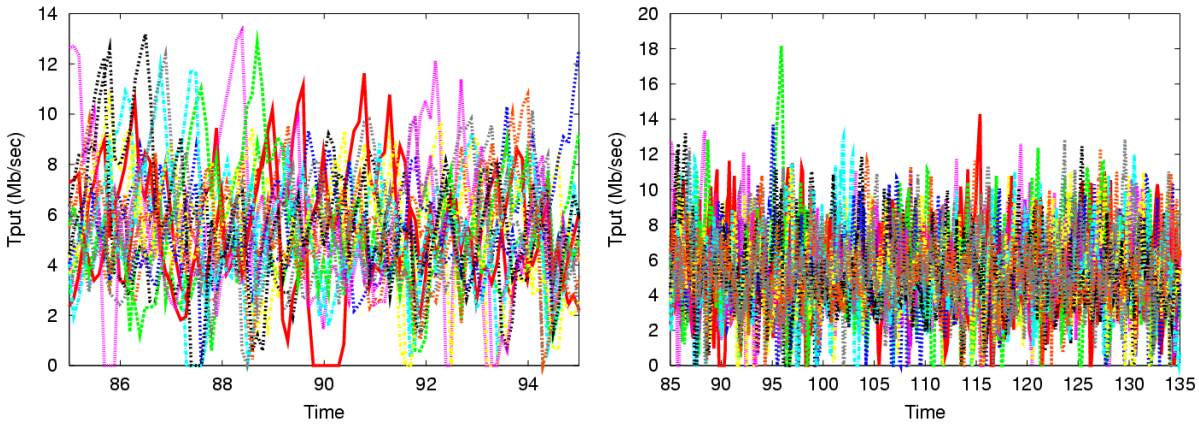


Figure 6.89: Unequal frame size with dynamic reconfiguration (Abilene). Throughput over time for TCP with 1 MB send buffer.

of CP-RUDP in terms of both aggregate throughput and number of streams.

In Section 6.10, we looked at equal frame size ensemble configurations with varying numbers of streams and varying frame sizes. In general, we found that CP-RUDP offers the lowest completion asynchrony and frame ensemble interarrival jitter numbers. As for the other four metrics, it achieves performance equivalent to the best of either TCP send buffer size configurations: stall rates that are nearly as low as TCP configured with a large send buffer, end-to-end delay values as low as TCP configured with a small send buffer, normalized flow share values as good as TCP configured with a large send buffer or better, and frame ensemble rates that are nearly as high as TCP configured with a large send buffer. Once again, we point out that CP-RUDP achieves all of these performance attributes *simultaneously*, and that no one TCP configuration is able to do so.

In Section 6.11, we looked at unequal frame size ensemble configurations, including static and dynamic modes of operation. We found that CP-RUDP once again offers comparable or better performance than the best of TCP: lower completion asynchrony values than TCP configured with a small send buffer, stall time results nearly as low as TCP configured with a large send buffer, end-to-end delay values that are less than TCP configured with a small send buffer, normalized flowshare values as good as or better than TCP configured with a large send buffer, frame ensemble rates comparable to TCP configured with a large send buffer, and frame ensemble interarrival jitter values that are better than TCP configured with a small send buffer.

While Abilene results were generally very successful in demonstrating CP-RUDP's performance advantages for multi-streaming over TCP, we once again note the periodic throughput drops that hampered CP-RUDP performance by lowering aggregate throughput somewhat and reducing overall frame ensemble rates. These can be seen in Figure 6.84 and Figure 6.87. We believe this is to be an artifact of traffic shaping at the MAGPI GigaPoP in Philadelphia and not an effect inherent to our approach or implementation. Further investigation, however, is needed.

Chapter 7

Conclusions and Future Work

The problem addressed in this dissertation is that of coordination among application flows sharing the same intermediary forwarding path between computing clusters. This problem is motivated by several trends in distributed multimedia applications: an increase in the number of computing hosts and communication devices deployed by a single application node, the use of multiple data flows to stream an ever-growing number of media types and manage sophisticated modes of user interaction, and an increase in application bandwidth requirements which make network resource management on the shared, public Internet essential.

The *cluster-to-cluster application* model captures these trends in a single network architecture that is at once a realistic application scenario in itself, and a building block for more complex, multi-cluster applications. The endpoints of communication in a cluster-to-cluster application are not single hosts, but clusters of computing hosts and associated devices. Data flows in this model share a common intermediary path between clusters, but not the entire end-to-end path.

Flows in a cluster-to-cluster application may employ a variety of transport protocols, each of which are selected to meet the communication requirements of a particular data type. At the same time, flows may share complex semantic relationships with one another in the larger context of the application. For example, flows may share temporal relationships and possess synchronization requirements, video streams may share geometric relationships based on camera positioning in the capture environment, text streams may annotate the changing content of media streams, control flows may give instructions for handling stream data based on current application state, etc. *Office of the Future* was cited as an example of a cluster-to-cluster application with strong semantic relationships among flows.

The fundamental problem in cluster-to-cluster applications like Office of the Future is that of *flow coordination*. Applications employ a large number of flows that share a common forwarding path between remote computing clusters. This path, since it typically cannot be provisioned end-to-end and is shared with other Internet flows, is a dynamic source of network latency, packet loss, and changes in available bandwidth. Ideally, an application would be aware of changing network conditions and make controlled adjustments to some or all flows to compensate for them. Adjustments would reflect application priorities and state, and exploit inter-stream tradeoffs to use limited network resources as effectively as possible.

In practice, however, application control over network resource usage in the face of changing network conditions is hardly ever realized. This is because application flows use transport protocols that operate in isolation from one another, share no consistent view of network conditions, and fail to respond to network delay and congestion in application-defined ways. The result is competition among flows rather than a coordinated use of resources. Fundamentally, lack of flow coordination stems from the inherent limitations of today's widely used transport-level protocols. Unicast protocols like TCP, UDP, SCTP, DCCP, TFRC, etc. were designed to operate independently and, as a result, lack mechanisms for coordinating with peer flows in the same application or configuring adaptive response behavior within a larger application context.

In-network approaches to this problem using well-studied techniques like traffic shaping or flow segmentation are undesirable for several reasons. First, they invariably require buffering as data from various flows awaits its rate-controlled time of transmission at management nodes. This increases end-to-end delay and hurts the performance of real-time, interactive applications. Second, the transparency of in-network approaches hides information from application endpoints. While transparency may be a feature in some contexts, here it prevents data sources from exercising adaptive behavior such as adjusting compression parameters, modifying media sampling rates, or interrupting low-priority data transfers temporarily. Finally, such approaches do not typically lend themselves to dynamic reconfiguration. As cluster-to-cluster applications change state and receive user feedback, flow requirements and priorities may be continually modified. While in-network based approaches like traffic shaping do not preclude reconfiguration, a source-based approach naturally affords more flexibility since control remains in the hands of the application as needs change dynamically.

7.1 Coordination Protocol Review

Our overall approach in designing the *Coordination Protocol* is to provide information sharing services to flow endpoints participating in the cluster-to-cluster application, and then to rely on application self-configuration to implement flow coordination in a problem-specific way. This approach avoids the use of in-network shaping, buffering, or scheduling mechanisms, and does not disrupt the semantics of end-to-end transport protocols employed by application flows. Our approach represents an *open architecture* in that CP provides a toolset for implementing application-specific flow coordination, not a ready-made solution.

In providing information sharing services, CP exploits the natural features of the cluster-to-cluster problem topology. First, cluster aggregation points (APs) provide a natural point of convergence on the outgoing data path, and a final aggregation hop before fanning out to application endpoints on the incoming path. As such, APs provide a natural point of information exchange that is within local administrative control. Second, application packets to and from individual endpoints provide a natural mechanism for exchanging information with an AP. Application packets between APs also provide a natural mechanism for exchanging probe information without introducing additional packets into the network.

Information exchange in CP is implemented using a per application *state table* maintained by each AP. The table is 256 KB in size and easily fits into AP memory as a cluster-to-cluster application runs. Application endpoints may read and write to cells in the table using CP headers piggybacked on application data packets. The CP state table maintains statistics on the cluster-to-cluster data path as well as the flows participating in the application. Some of the more important statistics include round trip time, packet loss rate, estimated bandwidth available to a single flow, number of flows in the application, average packet size, and aggregate application throughput.

Measurement of cluster-to-cluster data path conditions is accomplished on behalf of all application flows using probe information exchanges between APs. Rather than introducing new packets into the system, CP headers on existing application packets are simply overwritten with probe information as packets traverse the cluster-to-cluster data path in each direction. Since packets from all application flows can be used, path measurement is naturally fine-grained. Path information collected using this method includes both round trip time and packet loss events.

To estimate the bandwidth available to the cluster-to-cluster application, we leverage existing work on single flow, rate-based congestion control. In particular, our work has made extensive use of equation-based congestion control [FF99, PFTK98, FHPW00] and TCP-Friendly Rate Control (TFRC) [HFPW03]. It should be pointed out, however, that CP does not preclude the use of other congestion control algorithms as appropriate. In fact, nearly any single-flow, rate-based congestion control algorithm may be applied to the CP context to perform bandwidth estimation. This is because CP probing mechanisms and network statistics naturally provide the input parameters necessary to drive such algorithms. To demonstrate this, our work has made limited use of the *Rate Adaptation Protocol (RAP)* [RHE99] in addition to TFRC.

CP's approach to bandwidth estimation is to calculate a TCP-friendly sending rate appropriate for a single *flowshare*. An application may then multiply this value by the number of flows in the application (available from the AP state table) in order to obtain the aggregate bandwidth available to the application as a whole. In realizing this aggregate sending rate, an application may distribute bandwidth among flows in any manner. That is, it need not be the case that a single flow receives a single bandwidth flowshare.

Scaling the estimated bandwidth for a single flowshare to multiple flowshares was shown to cause problems for single-flow congestion control algorithms used in the estimation. This is typically because lost packet dynamics, an important input parameter to most algorithms, are strongly affected by the increase in packet arrival stream density. To solve this problem, we present a technique called *bandwidth filtered loss detection (BFLD)* that constructs a virtual packet arrival stream sub-sampled from the aggregate packet arrival stream. Experiments using the *ns2* simulator and our laboratory testbed demonstrate the effectiveness of our approach in maintaining TCP-friendly send rates for a wide variety of network conditions.

Experiments in our laboratory testbed explore the performance overhead of CP packet handling at AP routers. Using a 3.20 GHz Intel-based PC workstation configured to do software routing with FreeBSD, we were able to achieve a bi-directional throughput of 750 Mb/s or 65,000 packets. For the testbed configuration available to us, we were unable to reach a performance limit for CP handling distinct from baseline IP forwarding, even when going to considerable lengths to generate a worst case performance scenario. Our implementation, we argue, demonstrates the feasibility of our approach for most cluster-to-cluster applications on the commodity Internet today.

Finally, the effectiveness of our approach to flow coordination in cluster-to-cluster applications is demonstrated using the problem of *multi-streaming* in *3D Tele-immersion (3DTI)*. In this application, multiple endpoints capture video frames from different angles of a scene at the same instant of time. These frames must be streamed to a reconstruction cluster in a reliable manner such that the arrival time of frames within the same *frame ensemble* is approximately the same. CP can be used in this context to dynamically apportion bandwidth among endpoints in proportion to the amount of frame data each has to send. It may also be used to adjust send buffer sizing on each endpoint to dynamically match the bandwidth delay product on the cluster-to-cluster data path.

Experiments in our laboratory testbed demonstrate the effectiveness of CP in improving reliable multi-streaming performance when compared to uncoordinated TCP streaming. In particular, CP provides low arrival completion asynchrony, low end-to-end delay, and minimal stalling while, at the same time, maintaining frame ensemble streaming rates and network utilization that are appropriately high. While TCP may be configured to achieve some of these results, no single configuration is possible that can compete with CP which achieves all of them simultaneously. Results for unequal frame size ensembles demonstrate these conclusions even more strongly. Experiments using the Abilene backbone network support these conclusions in an uncontrolled environment with significant scaling.

7.2 Research Contributions

The contributions of this dissertation to the field of Computer Science may be summarized as follows:

- We identify an important class of forward-looking distributed multimedia applications known as *cluster-to-cluster applications* and describe their generalized characteristics.
- We define the *flow coordination problem*. This problem is fundamental to cluster-to-cluster applications, but also of interest to multiflow Internet applications generally.
- We propose a novel open architecture, called the *Coordination Protocol (CP)*, that solves the flow coordination problem in cluster-to-cluster applications.

- We identify the *multiple flowshare problem* and solve it using a novel technique called *bandwidth filtered loss detection (BFLD)*. Using this technique, almost any single-flow congestion control algorithm can be scaled to a multi-flow cluster-to-cluster application context.

- We implement an experimental prototype of CP and evaluate various aspects of its performance. Our implementation includes both kernel extensions for FreeBSD software routers and a reliable transport protocol called CP-RUDP that uses CP information to perform coordinated adaptation.

- We demonstrate how CP can be applied to the problem of multi-streaming in *3D Tele-immersion (3DTI)*, a complex cluster-to-cluster application developed at UNC, for dramatically improved communication performance.

7.3 Future Research Directions

In this section, we describe several directions for future work. These include applying CP to wireless cluster-to-cluster applications which have somewhat different assumptions than our original problem model, addressing the issue of network security, developing new CP-based transport protocols, and exploring novel coordination schemes that exploit CP information and state exchange mechanisms.

Wireless clusters. One important area of future work is modifying CP to work with wireless clusters. The cluster-to-cluster application model described in Section 1.4 makes the assumption that communication *within* an application cluster contributes very little to overall end-to-end delay and packet loss. This is because clusters are within local administrative control and can be provisioned to comfortably support the networking requirements of the application. This assumption is violated in wireless clusters where transmission errors and significant delay are inherent to the technology.

One approach that we have considered is designing transport protocols that can use CP to distinguish between *local* (i.e., wireless) and *AP-to-AP* sources of delay and loss. This can be done by creating mechanisms that measure end-to-end path properties and then compare them to CP state table values. Significant changes that diverge from CP values indicate local effects while corresponding changes point to the cluster-to-cluster data path. Adaptive measures taken by the application in response to these two types of network effects may differ as appropriate.

Security. Another area of future work is that of network security. Our current AP implementation uses a pre-configured flow table to recognize CP packets and authorize state table read and write operations. This flow table is configured manually by a network administrator who must know in advance the source and destination addresses/ports involved. While this setup provides some measure of protection against malicious users, it is not nearly enough. For example, packets with legitimate destination addresses but spoofed source addresses may be directed at APs and match entries in the flow table. Such packets may then modify state table values, falsify network probe information, create additional state tables for non-existent cluster-to-cluster applications, or contribute additional flows to existing applications where such flows do not exist.

To prevent such malicious behavior, an authentication protocol might be added to CP. The protocol could begin with a key exchange of some sort. Keys, for example, might be distributed to application endpoints offline, and placed in the AP flow table by an administrator. The initial exchange could then take place using the operation fields of CP headers in outgoing packets and report fields of incoming packets. Once the exchange has occurred, a validation token could be issued by the AP and then required by each CP packet before state table operations will be performed. Such tokens could be changed periodically for additional robustness against attack. How this validation token would be placed in the CP header is an issue for further exploration.

New CP-based transport protocols. As described in Section 3.3.1, a CP-based transport protocol provides an application with data transport services that take into account the larger context of application flow coordination. Since this context is application-specific, we expect many such protocols to be tightly woven to the problem domain and data types involved. The *CP-RUDP* prototype discussed in Section 6.4 serves as an example of one CP-based transport protocol designed to meet the specific requirements of multi-streaming in 3DTI. In general, the expanded operational and informational context of transport-level protocols create a wide open space for future research.

New CP-based transport protocols might display *peer dependencies* in a novel manner. For example, CP information sharing might allow the reliability semantics of one flow to be tied to the performance of another flow. Or, the mere presence of one flow may trigger the start or stop of other flows within the same application. CP-based transport protocols may interact with applications in a variety of ways. Some proto-

cols may operate in a fairly transparent manner and provide an API that looks much like the socket API on most UNIX systems. Others may provide only a thin library that exposes interactions with an AP state table to the application. Callbacks, for example, could be set up to drive application-level adaptation and allow the direct manipulation of transmission rates. A myriad of hybrid approaches is also possible.

New application coordination schemes. Coordination schemes, as described in Section 3.3.2, are used to define global application objectives and specify the behavior of individual flows using CP to achieve those objectives. In 3DTI, for instance, the goal of coordinated data transport is the synchronous arrival of video frames captured by multiple endpoints at the same instant in time. To achieve this goal for unequal frame sizes, application bandwidth is apportioned among streaming hosts in proportion to the amount of frame data each has to send. Implementing this scheme requires the use of general purpose, network, and flow addresses within the state table, and the exchange of frame size information among flows.

In general, creating new algorithms that exploit CP state sharing mechanisms to achieve flow coordination in various ways is a open area of future research. Many such algorithms will focus on how limited bandwidth may be apportioned among flows to most effectively achieve application goals during a particular interval of time. In addition, however, an application may use CP mechanisms to perform one or more types of *context-specific coordination*. That is, an application may use CP state exchange mechanisms to achieve coordination for any arbitrary problem. Some examples include leader election, fault detection and repair, media capture synchronization, coordinated streaming of multiple data types, distributed floor control, dynamic priority assignment, and various types of group consensus.

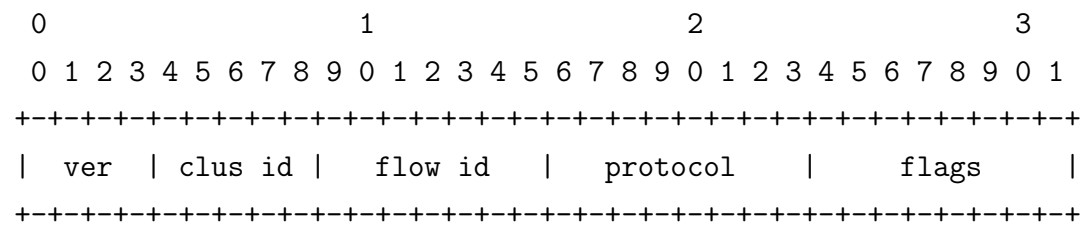
7.4 Summary

In this chapter we summarized the work presented in this dissertation. First, we reviewed the problem of flow coordination in distributed multimedia applications and several motivations. Then, we summarized on a very high level various features of the Coordination Protocol, our solution to this problem. Finally, we described several areas of future work including wireless clusters, network security, new CP-based transport protocols, and new flow coordination schemes.

Appendix A

CP Header Formats

A.1 Standard Prefix Format



This single-word prefix includes the following fields:

Version (4 bits)
: Coordination Protocol (CP) version number.

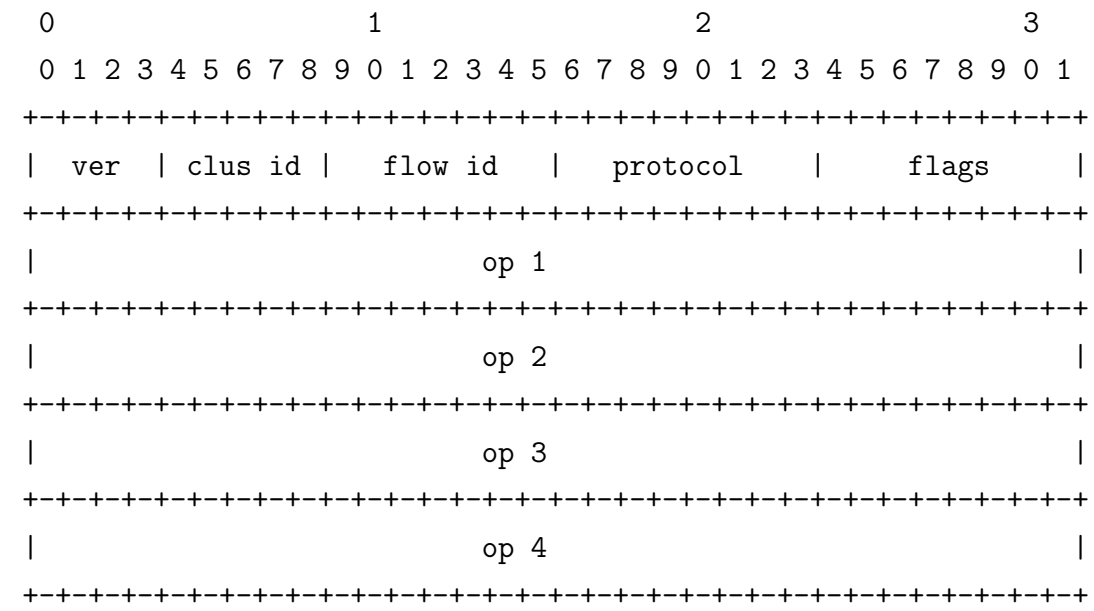
Cluster ID (5 bits)
: Cluster-to-cluster application identifier.

Flow ID (7 bits)
: Flow identifier.

Protocol (8 bits)
: Transport protocol employed by this flow.

Flags (8 bits)
: Flags directing AP to handle packet in special ways.

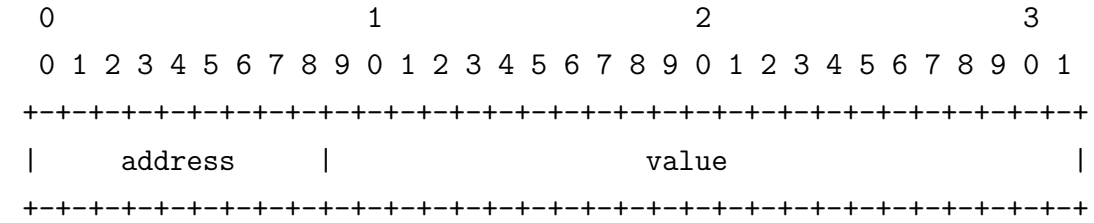
A.2 Format for Endpoint-to-AP Packet Exchanges



A.2.1 Operation Field Format

Each operation ("op") field may take one of two formats: general purpose (GP) address assignment and report address assignment.

General Purpose (GP) Address Assignment



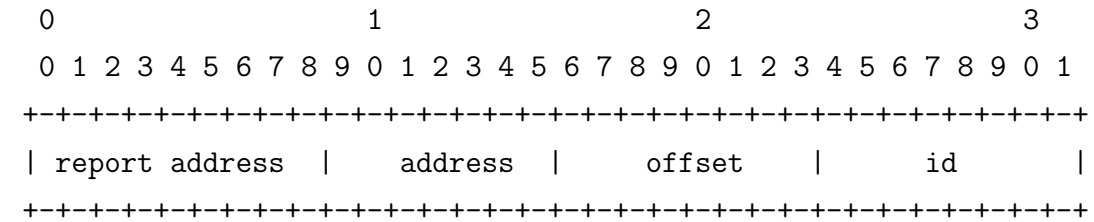
Address (8 bits)

General Purpose (GP) address to assign.

Value (24 bits)

Value to be stored at *Addr.flowid*.

Report Address Assignment



Report address (8 bits)

Report address to assign.

Address (8 bits)

Address locating reported value.

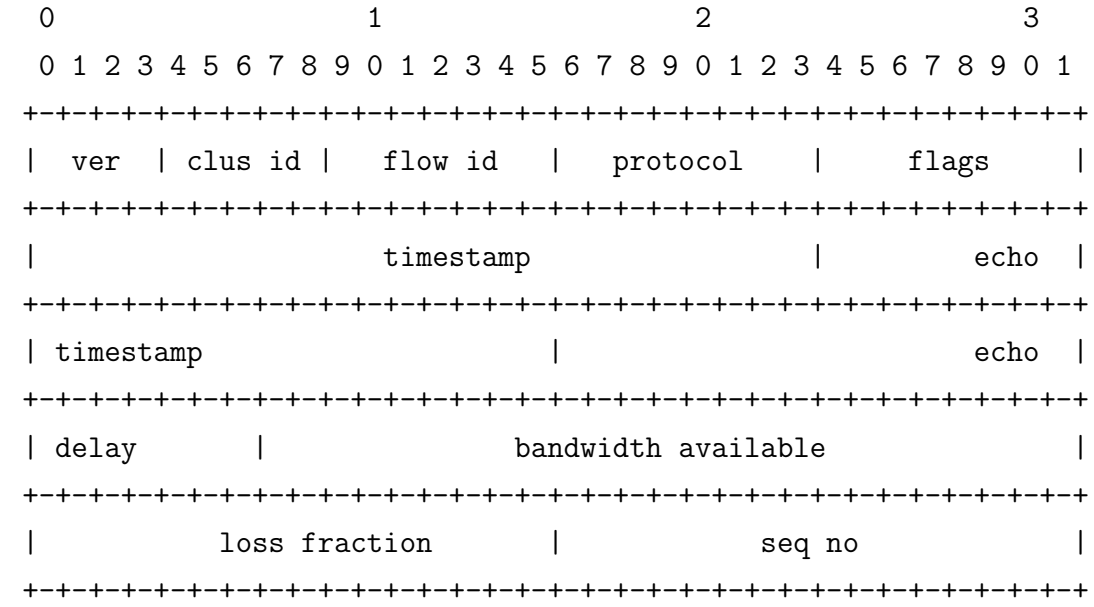
Offset (8 bits)

Offset locating reported value.

ID (8 bits)

Application-chosen identifier to be associated with reported value.

A.3 Format for AP-to-AP Packet Exchanges



Timestamp (24 bits)

Send time of packet.

Echo timestamp (24 bits)

Timestamp taken from the last packet received from remote AP.

Echo delay (24 bits)

Time between last packet received from remote AP and send time of current packet.

Bandwidth available (24 bits)

Estimated bandwidth (KB/sec) available to a single TCP-conformant flow.

Loss fraction (16 bits)

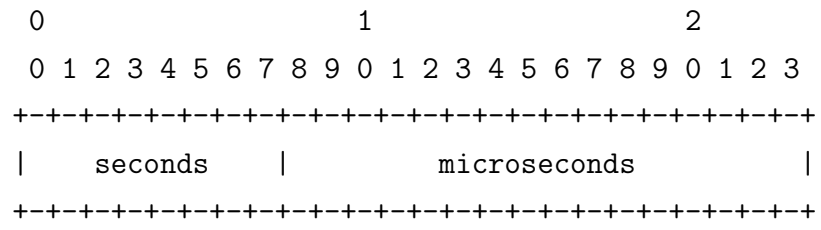
Fraction of packets lost. (Unit value is $1/2^{16}$.)

Sequence number (16 bits)

Sequence number used for detecting packet loss.

A.3.1 Timestamp Format

Timestamp, echo timestamp, and delay fields make use of the following format:



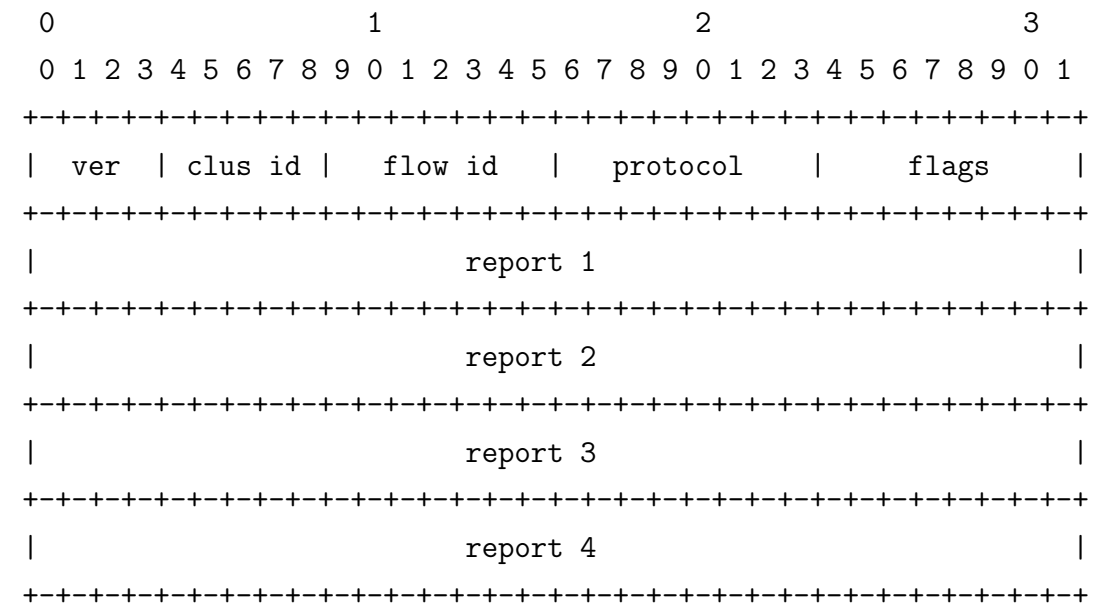
Seconds (8 bits)

8 least significant bits in 32-bit seconds portion of UNIX time.

Microseconds (16 bits)

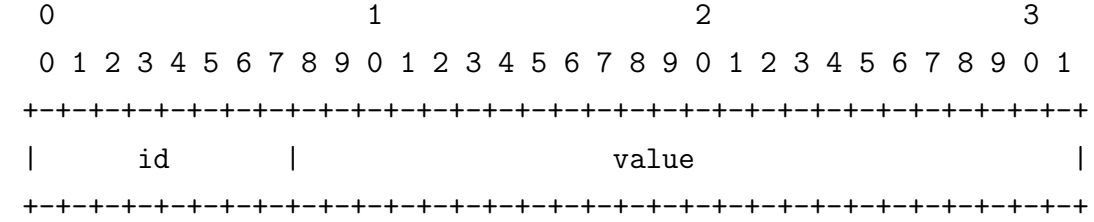
16 most significant bits in 32-bit microseconds portion of UNIX time.

A.4 Format for AP-to-Endpoint Packet Exchanges



A.4.1 Report Format

Each report field makes use of the following format:



ID (8 bits)

Application-defined identifier.

Value (24 bits)

Value taken from state table on local AP.

A.5 C Source Code for Generic CP Header

```
struct cphdr {

#if BYTE_ORDER == LITTLE_ENDIAN
      u_short ch_fid:7,    /* flow id */
      ch_cid:5,            /* cluster id */
      ch_ver:4;            /* version */
#endif
#if BYTE_ORDER == BIG_ENDIAN
      u_short ch_ver:4,    /* version */
      ch_cid:5,            /* cluster id */
      ch_fid:7;            /* flow id */
#endif
      u_char ch_prot;
      u_char ch_flags;

      u_long ch_w1;
      u_long ch_w2;
      u_long ch_w3;
      u_long ch_w4;
};
```

Appendix B

Laboratory Testbed

Much of the experimental work in this dissertation was done using the laboratory testbed pictured in Figure B.1 and shown diagrammatically in Figure B.2. Overall, the testbed is designed to emulate the topology shown in Figure 1.5, making further provisions for competing TCP flows used to measure fairness and additional background traffic used to generate various network loads.

CP hosts and their local AP on each side of the network represent two clusters that are part of the same cluster-to-cluster application and exchange data with one another. Each endpoint is an Intel-based machine running RedHat 9 (kernel version 2.4) or FreeBSD 4.5 and sending data on a 100 Mb/s link to its local AP. Each AP is an Intel-based machine (Xeon 3.06 GHz processor, 512K cache, 2 GB of memory) running FreeBSD 4.9 that has been configured to do software-based IP-forwarding using static routes. In addition, each AP has been CP-enabled with a special kernel module implementing all of the mechanisms described in Chapter 3 and Chapter 4. Aggregate cluster-to-cluster traffic arrives and leaves the AP on 1 Gb/s links. (See Chapter 5 for a more complete description of our AP implementation.)

At the center of our testbed are two routers connected using two 100 Mb/s FastEthernet links. This creates a bottleneck and, by configuring traffic from opposite directions to use separate links, emulates the full-duplex behavior seen on wide-area network links.

Figure B.1: Experimental network setup.

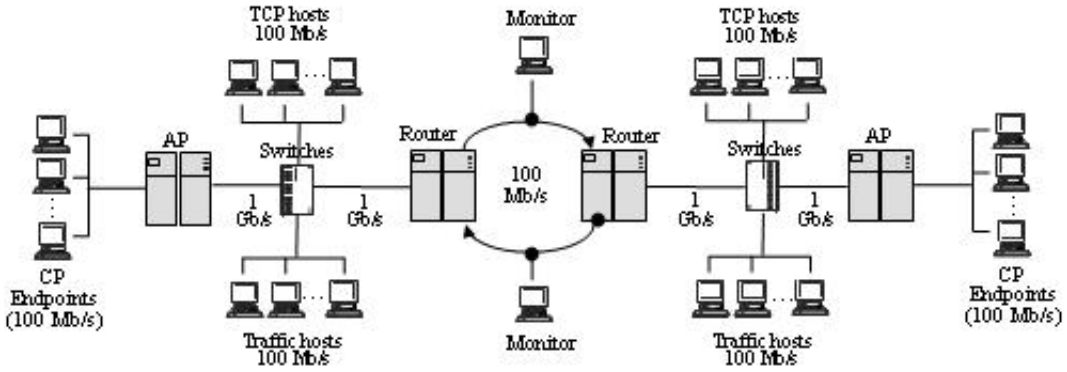


Figure B.2: Experimental network setup.

Each router is an Intel-based machine (1 GHz Pentium III with over 1 GB memory) running FreeBSD 4.5 and configured to do software-based IP-forwarding using static routes. In addition, each router runs *ALTQ* [Ken98] kernel extensions to FreeBSD and makes use of FreeBSD *dummynet* [Riz97] to emulate various link characteristics. (Each of these is explained below.)

In order to measure the fairness of cluster-to-cluster application flows to TCP flows sharing the same bottleneck link, we use two sets of end hosts (labeled "TCP hosts" in Figure B.2) and the well-known utility *iperf* [Ipe]. Each host is once again an Intel-based machine running FreeBSD 4.5. Iperf flows in this network are long-lived TCP Reno flows that compete with application flows on the same bottleneck throughout the experiment. The amount of bandwidth received by these flows is compared with that received by cluster-to-cluster application traffic in order to quantify CP's fairness to competing TCP flows. Some metrics used to make this comparison include *normalized throughput ratio* from Section 4.5.3 and *normalized flowshare* from Section 6.6.2.

Also sharing the bottleneck link for many experiments are background traffic flows that generate various network loads on the bottleneck link in the center of the network. The end systems used for these flows, labeled "Traffic hosts" in Figure B.2, are once again Intel-based machines of various specs running FreeBSD 4.5. Each runs either a client or server installation of the *thttp* Web traffic emulator described below.

Finally, network monitoring during these experiments is done in two ways. First, two Intel-based monitor hosts running FreeBSD 4.5 use *tcpdump* to capture TCP/IP headers from packets traversing the bottleneck link. These traces are then filtered and processed for detailed performance data. Second, a software tool is used in conjunction with *ALTQ* [Ken98] 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.

B.1 Emulation Tools

An important link emulation tool used in our laboratory testbed is *dummynet* [Riz97]. *Dummynet* is a FreeBSD utility that performs traffic shaping on a designated network interface at the kernel level. *Dummynet* may be used to classify packets and divide them into flows. A pipe abstraction is then applied that emulates link characteristics including bandwidth, propagation delay, queue size, and packet loss rate.

One way dummynet is commonly used in the laboratory testbed is to configure bottleneck link characteristics. This is done by creating a pipe on each of the two router hosts for the outbound, 100 Mb/s link. This pipe is then configured for particular delay values (to increase round trip time) and various packet loss rates. Dummynet may also be used by end systems to shape particular flows. For example, Web flows generated by the *thttp* traffic generator are assigned a random round trip time using dummynet as described below.

Another emulation tool used is *ALTQ* [Ken98]. *ALTQ* is a FreeBSD kernel module that extends IP-output queuing routines in order to enable research on queue-related issues in IP forwarding. Such issues include queue size, queue management discipline (RED [FJ93], ARED [FGS01], PI [HMTG01], REM [ALLY01], etc.), and various quality of service schemes (CBQ [FJ95], Diffserv [BCD+98], etc.). While these issues are of great interest to the network research community generally, ALTQ is used in this dissertation merely to configure queue size on the outbound link of bottleneck routers and log packet loss events that occur due to queue overflow. Information logged in this way provides a very detailed and accurate account of packet loss due to network congestion at bottleneck routers.

B.2 Background Web Traffic Generation

Synthetic Web traffic is used within the laboratory testbed for many experiments that look at network workload as an important input parameter. This traffic is generated by the *thttp* traffic generator developed at UNC Chapel Hill. The current version

Element	*Description*
Request size	HTTP request length in bytes.
Response size	HTTP reply length in bytes (top-level & embedded).
Page size	Number of embedded (file) references per page.
Think time	Time between retrieval of two successive pages.
Persistent connections	Number of requests per persistent connection.
Servers per page	Number of unique servers used for all objects in a page.
Consecutive page retrievals	Number of top-level pages requested from a server.

Table B.1: Elements of the HTTP traffic model.

of *thttp* [1] is based on a recent large-scale analysis of Web traffic in [SCJO01] and a modified empirical model of HTTP first proposed by [Mah97]. This analysis considers both HTTP/1.0 and HTTP/1.1 and reflects the use of HTTP in many contemporary browsers and servers.

Thttp works by dividing HTTP Web requests and responses into a set of measurable components or elements. The most important of such elements are given in Table B.1. For each element, [SCJO01] provides a distribution function that describes the distribution of values actually seen across a large number of Web transfers from the data set. For example, the distribution of HTTP request and response sizes is given in Figure B.3 and Figure B.4, respectively. [2]

To emulate Web traffic, both client and server portions of *thttp* are constructed as simple state machines that follow the HTTP model described in [SCJO01]. For each element in the model, the appropriate distribution function is sampled to obtain a value that will dictate behavior. For example, think time (the time between Web requests) is sampled by the client and then observed, request size is sampled before making a Web request with exactly that size, server response time and server response size are sampled by the server before responding with a Web object, and so on. Some more complex values in the model include the number of parallel TCP connections that are employed by the client browser, the number of nested objects associated with a Web page, and whether or not the client browser makes use of persistent connections to make multiple Web requests.

While the statistical similarity between *thttp* generated traffic and real Web traffic is impressive, what makes *thttp* an especially powerful tool for networking research is that

[1] See [CJOS00] and [LAJS03] for additional details on *thttp*.

[2] Thanks to Long Le for permission to use Figure B.3, Figure B.4, Figure B.5, and Table B.1. Each appears in [LAJS03].

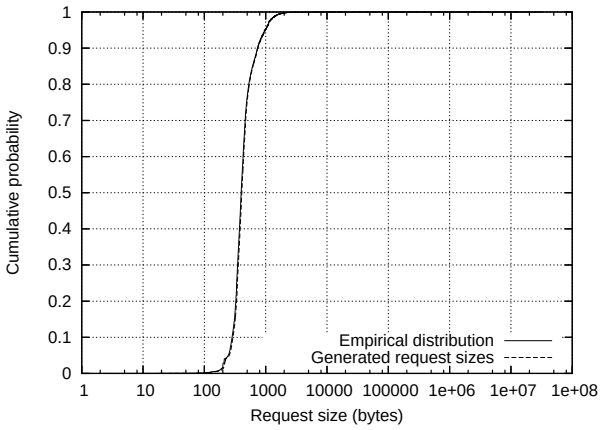


Figure B.3: Request size.

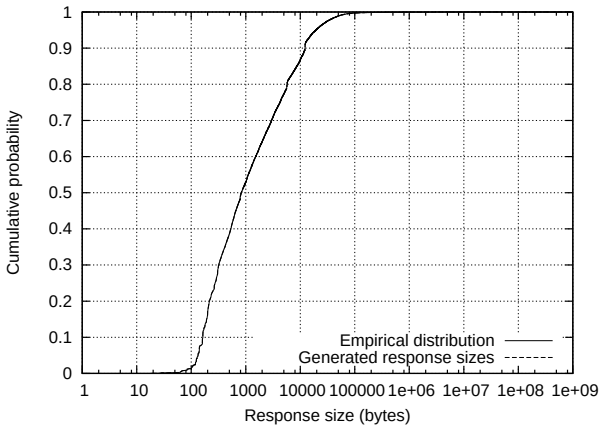


Figure B.4: Response size.

a small number of program instances can be used to emulate the behavior of *thousands* of Web users. Figure B.5 shows the results of calibration experiments in [LAJS03]. These experiments demonstrate the linear relationship between number of emulated browsers and network workload, and establish the ability of *thttp* to effectively emulate the behavior of more than 17,000 Web browsers. (Levels used in this dissertation do not exceed 6,000.)

In order to better emulate real-world Web traffic, *dummynet* was used to vary end-to-end network latency across flows. To do this, traffic hosts in the laboratory testbed, each running FreeBSD 4.5, were equipped with a locally-modified version of *dummynet* that created a random minimum delay between 10 and 150 milliseconds for each TCP flow generated by a *thttp* client. This delay scheme emulates the wide range of round trip times seen on the Internet as both Web browsers and Web servers exchange data in a world that is widely dispersed geographically. The values 5 and 150 were chosen because the range of round trip times approximate those seen in the continental United States. (See [CJOS00] for more information on this point.)

Finally, it should be noted that the Web traffic generated by *thttp* uses heavy-tailed distributions for both browser think times (OFF times) and server response size (ON times). This leads to a substantial degree of long-range dependency (LRD) in resulting aggregate packet arrival processes[LAJS03]. This property is important partly because it means that the traffic generated by *thttp* closely resembles that of real network traffic which as been shown to exhibit this characteristic. [WTSW97] It is also important because of the loss properties generated when drop tail queues on the outbound link of each bottleneck router interacts with such traffic. Instead of single

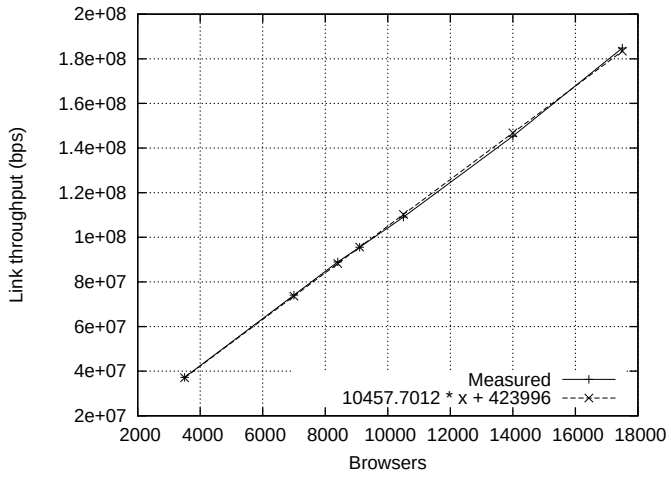


Figure B.5: *Thttp* browser configuration and resulting network load.

losses separated by random intervals, packet loss from queue overflow events are bursty and correlated. [Pax99] Using *thttp*, we are able to capture this real-world dynamic in our laboratory testbed.

thttp requires a startup period of approximately 20-30 minutes to achieve a stable workload level. Experiments in this dissertation make use of this rampup period before going on to collect various types of performance data.

Appendix C

Reliable-UDP (RUDP)

Reliable-UDP, or *RUDP* [1], is an application-level transport protocol implemented as an object-based library on top of UDP. Its principle function is to provide reliable, connection-based data transport *without congestion control*. (Additional layering based on CP will provide congestion control as explained in Section 6.4.) RUDP is stream-oriented, despite its implementation using UDP datagrams.

Like TCP, RUDP uses several packet types to manage connection setup, data exchange, and connection termination. These include SYN, SYN-ACK, DATA, ACK, and FIN. Connection setup is accomplished using TCP's familiar three-way handshake [Pos81] (SYN, SYN-ACK, ACK) to insure that each endpoint is able to participate. Similarly, connection termination is accomplished using a mutual FIN and ACK exchange. Data streaming is accomplished using sender DATA packets and receiver ACK packets. Each packet type is associated with a particular header as seen in Section C.1.

RUDP uses several mechanisms to implement reliability. These include the following:

- **Sequence numbers.** Data is uniquely indexed using per-packet sequence numbers. A DATA packet will include a sequence number of the data contained in the packet. An ACK packet will included a cumulative sequence number indicating that all data packets have been received up to that point.

- **Packet numbers.** Each data packet is assigned a monotonically increasing packet number, including retransmission packets. Packet numbers are echoed in each ACK packet and provide senders with information useful in disambiguating between duplicate ACKs and quickly recognizing lost packets.

[1] Thanks to Travis Sparks and Ketan Mayer-Patel for their work on RUDP design and version 1.0 implementation.

- **Hole numbers.** Acknowledgment packets each contain the sequence number of the first "hole" or "gap" in the receive buffer. Much like a lightweight version of TCP-SACK[MMFR96, FMMP00], this number acts as a retransmission hint; it allows the sender to retransmit a packet before duplicate acknowledgments and timeouts establish that a loss has occurred.

Section C.1 provides information on precisely how these values are contained within RUDP packet headers.

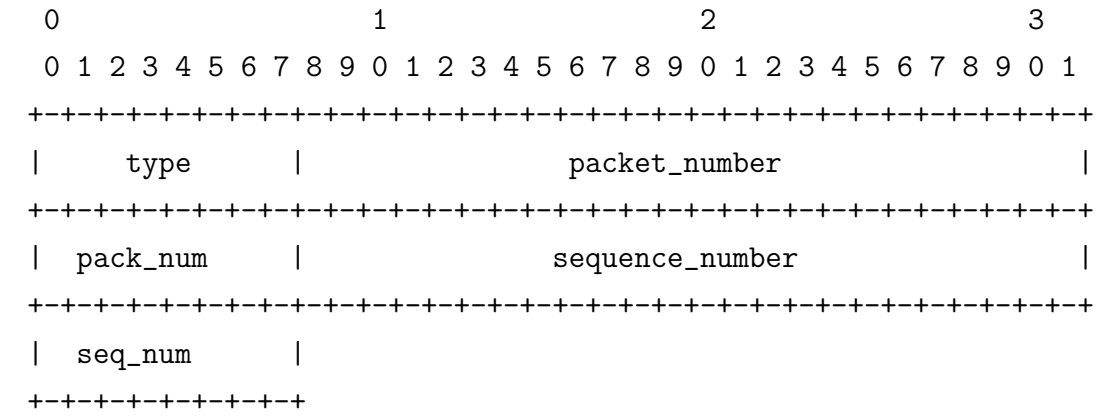
Internally, RUDP is implemented as a multi-threaded finite state machine. A multi-threaded design is required in order to listen for and handle incoming packets while servicing API calls (e.g., send, receive) from the application. The finite state machine is a familiar device for prompting actions in particular situations and transitioning to further actions in sequence. For example, connection setup may conveniently be described using a series of states and possible transitions depending upon a remote endpoint's response actions.

The use of multi-threading requires that shared internal data structures be protected using OS-supported mutual exclusion primitives. The POSIX thread library (*pthreads*) supplied this functionality and was used throughout. In particular, mutual exclusion primitives were needed for data structures related to send buffers, receive buffers, and a free list of pre-allocated data packets.

RUDP's application programming interface (API), in general, looks much like the standard TCP socket API in UNIX. In part, this is because of RUDP's connection-oriented design which mimics that of TCP. A receiver will *bind* their socket to an address, *listen* for incoming connections, and then *accept* one when it arrives. Meanwhile, the sender will *connect* with a remote receiver before initiating data transfer.

C.1 Header Formats

C.1.1 SYN and FIN Format (9 Bytes)

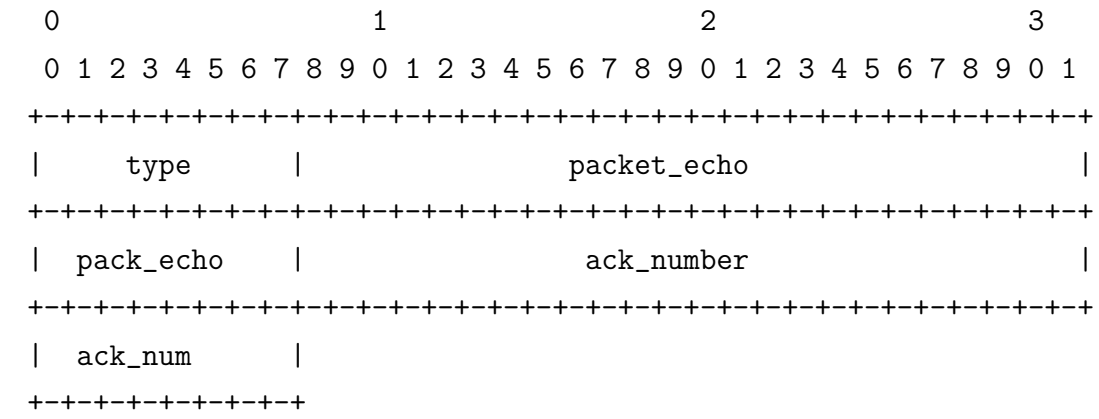


Type (8 bits)
: Packet type (SYN, FIN, data, etc.).

Packet number (32 bits)
: Packet number.

Sequence number (32 bits)
: Data sequence number.

C.1.2 SYN ACK Format (9 Bytes)



Type (8 bits)
: Packet type (SYN, FIN, data, etc.).

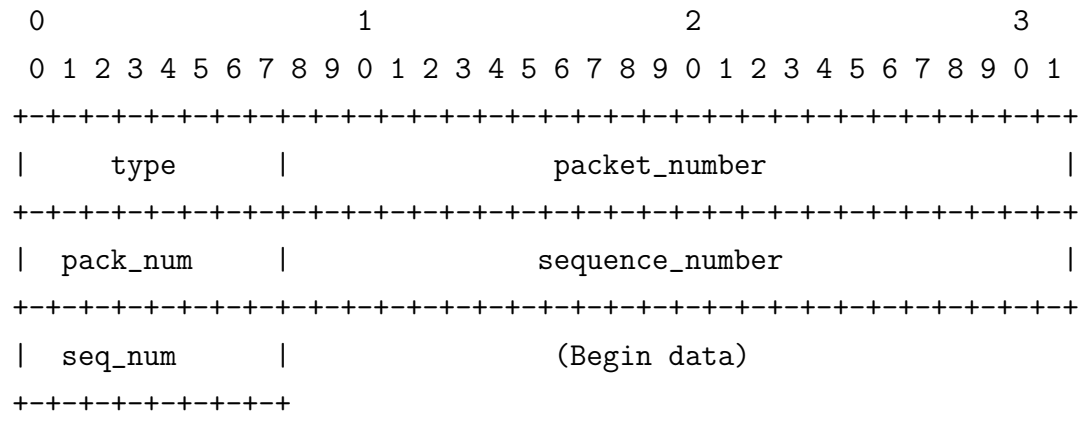
Packet echo (32 bits)

Packet number echo.

ACK number (32 bits)

Data sequence number acknowledged.

C.1.3 Data Format (9 Bytes)



Type (8 bits)

Packet type (SYN, FIN, data, etc.).

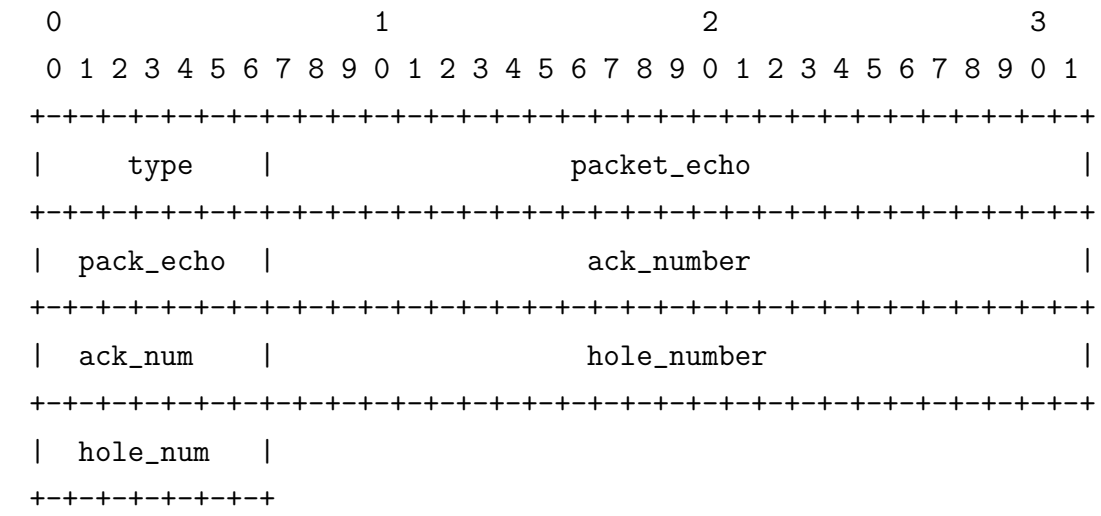
Packet number (32 bits)

Packet number.

Sequence number (32 bits)

Data sequence number.

C.1.4 ACK Format (13 Bytes)



Type (8 bits)
Packet type (SYN, FIN, data, etc.).

Packet echo (32 bits)
Packet number echo.

ACK number (32 bits)
Data sequence number acknowledged.

Hole number (32 bits)
Data sequence number of first packet missing.

C.2 Application Programming Interface (API)

C.2.1 Manipulating Connections

static RUDPConn *createConnection(int sock_fd, struct sockaddr_in *to_addr, const unsigned int cluster_id, const unsigned int flow_id) - Create a new connection.
static RUDPConn *getConnection(int fd) - Get a current connection.
static void closeConnection(RUDPConn *conn) - Close connection.
static void destroyConnection(RUDPConn *conn) - Destroy connection.

C.2.2 Manipulating Packets

static RUDPPacket* createPacket(RUDPConn *conn) - Create a new packet.
static void destroyPacket(RUDPPacket *p) - Destroy packet.
static void freePacket(RUDPConn *conn, RUDPPacket *p) - Free packet memory.

C.2.3 Read/Write Packet Header Fields

static int getPacketType(unsigned char *pdata) - Read packet type field.
static unsigned int getSeqNum(unsigned char *pdata) - Read sequence number field.
static unsigned int getPackNum(unsigned char *pdata) - Read packet number field.
static unsigned int getAckNum(unsigned char *pdata) - Read sequence number acknowledgement field.
static unsigned int getPackEcho(unsigned char *pdata) - Read packet number echo field.
static unsigned int getHoleNum(unsigned char *pdata) - Read hole number field.

static void setPacketType(unsigned char *pdata, int ptype) - Set packet type field.
static void setSeqNum(unsigned char *pdata, unsigned int snum) - Set sequence number field.
static void setPackNum(unsigned char *pdata, unsigned int pnum) - Set packet number field.
static void setAckNum(unsigned char *pdata, unsigned int anum) - Set sequnce number acknowledgement field.
static void setPackEcho(unsigned char *pdata, unsigned int pecho) - Set packet number echo field.
static void setHoleNum(unsigned char *pdata, unsigned int hnum) - Set hole number field.

Appendix D

CP Application Programming Interface (API)

D.1 Socket Library

struct cpcb* cpsock_create(const int socket_fd, const u_short cid, const u_short fid, const u_char protocol) - Create socket.
void cpsock_init_cpbuf(const struct cpcb *cb, struct cpbuf *b) - Initialize send buffer.
int cpsock_send(const struct cpcb *cb, const struct cpbuf *b) - Send data.
int cpsock_sendto(const struct cpcb *cb,const struct cpbuf *b, const struct sockaddr_in *sa) - Send data to designated address.
int cpsock_recv(struct cpcb *cb, struct cpbuf *b) - Receive data.
void cpsock_exit(struct cpcb* cb) - Terminate socket and exit.

D.2 Assignment Functions

void cpsock_set(struct cpcb* cb, const u_long opfield, const u_char addr, const u_long value) - Set operation field to given value.
void cpsock_set_report(struct cpcb* cb, const u_long opfield, const u_char addr, const u_char report_addr, const u_char report_offset, const u_char report_id) - Assign operation field to report request.
void cpsock_set_report_NOOP(struct cpcb* cb, const u_long opfield) - Assign operation field to NOOP.

D.3 Report Functions

int cpsock_verify_report_id(const struct cpbuf* cpb, const u_long reportno, const u_char report_id) - Return true if report number matches.
int cpsock_is_valid_report(const struct cpbuf* cpb, const u_long reportno) - Return true if not a NOOP report.
u_char cpsock_get_report_id(const struct cpbuf* cpb, const u_long reportno) - Return report ID.
u_long cpsock_get_report_value(const struct cpbuf* cpb, const u_long reportno) - Return report value.

D.4 CP Buffer

```
struct cpbuf {    /* cp buffer */

  void *buf;      /* ptr to allocated buffer */
  size_t buflen;  /* buffer length */
  size_t len;     /* data length (cphdr + cpdata) */
  void *data;     /* ptr to start of data */
};

struct cpcb {     /* cp control block */
  int sockfd;
  struct cphdr hdr;
};
```

struct cpbuf* cpbuf_create() - Creates CP buffer object.
void cpbuf_destroy(struct cpbuf *c) - Destroys CP buffer object.

Appendix E

CP-RUDP Application Programming Interface (API)

E.1 Connection Setup and Termination

int cprudp_socket() - Create CP-RUDP socket.
int cprudp_bind(const int rudpd, const int port) - Bind socket to an address.
int cprudp_listen(const int rudpd) - Listen for connections.
int cprudp_accept(const int rudpd, const unsigned int cluster_id, const unsigned int flow_id) - Accept connections.
int cprudp_connect(const int rudpd, const char* server, const int port, const unsigned int cluster_id, const unsigned int flow_id) - Connect with remote socket.
int cprudp_close(const int rudpd) - Close socket.

E.2 Configuration

int cprudp_set_send_buffer_size(const int rudpd, const int size) - Configure send buffer size.
int cprudp_set_receive_buffer_size(const int rudpd, const int size) - Configure receiver buffer size.
int cprudp_setsockopt(const int rudpd, const int level, const int optname, const void* optval, const int size) - Set (UNIX) socket option.
int cprudp_getsockopt(const int rudpd, const int level, const int optname, void* optval, int* size) - Get (UNIX) socket option.

E.3 Send and Receive

int cprudp_read(const int rudpd, const void *buffer, const int size) - Read data from socket.
int cprudp_write(const int rudpd, const void *buffer, const int size) - Write data to socket.
int cprudp_send(const int rudpd, const void *buffer, const int size, ...) - Send data using socket.
int cprudp_receive(const int rudpd, const void *buffer, const int size) - Send data using socket.

E.4 Miscellaneous Functions

unsigned int cprudp_get_rtt_est(const int rudpd) - Get RTT estimation.
const char* cprudp_get_remote_addr(const int rudpd) - Get address of local endpoint.
const char* cprudp_get_local_addr(const int rudpd) - Get address of remote endpoint.

BIBLIOGRAPHY

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

[Abi] Abilene. http://abilene.internet2.edu/.

[AEH86] S.R. Ahuja, J.R. Ensor, and D.N. Horn. The rapport multimedia conferencing system. *Proceedings of the Conference on Office Information Systems (ACM-SIGOIS '88)*, March 1986.

[ALLY01] Sanjeewa Athuraliya, Victor H. Li, Steven H. Low, and Qinghe Yin. REM: Active queue management. *IEEE Network*, 15(3):48 – 53, May/June 2001.

[BBP88] R. Braden, D. Borman, and C. Partridge. *RFC 1071: Computing the Internet Checksum*, September 1988.

[BCC+98] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. *RFC 2309: Recommendations on Queue Management and Congestion Avoidance in the Internet*. Internet Engineering Task Force, April 1998.

[BCD+98] D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *RFC 2475: An Architecture for Differentiated Services*. Internet Engineering Task Force, December 1998.

[BCZ98] S. Bhattacharjee, K. Calvert, and E. Zegura. Commentaries on active networking and end-to-end arguments. *IEEE Network*, May/June 1998.

[BEF+00] 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*, 33(5):59–67, May 2000.

[BRS99] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for internet hosts. *Proceedings of ACM SIGCOMM*, September 1999.

[CDO+00] L. Childers, T. Disz, R. Olson, M. Papka, R. Stevens, and T. Udeshi. Access grid: Immersive group-to-group collaborative visualization. In *Proceedings of the 4th International Immersive Projection Technology Workshop*, 2000.

[CGW02] Kenneth L. Calvert, James Griffioen, and Su Wen. Lightweight network support for scalable end-to-end services. In *Proceedings of ACM SIGCOMM*, August 2002.

[CJ89] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1–14, June 1989.

[CJOS00] M. Christiansen, K. Jeffay, D. Ott, and F.D. Smith. Tuning RED for web traffic. *Proceedings of ACM SIGCOMM 2000*, September 2000.

[CMK+99] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, , and Daniel Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communications Review*, 29(2):7–23, April 1999.

[CMT98] K. Claffy, G. Miller, and K. Thompson. The nature of the beast: Recent traffic measurements from an internet backbone. In *Internet Society INET'98*, 1998.

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

[FF96] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[FF99] 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.

[FGS01] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An algorithm for increasing the robustness of RED, 2001.

[FHPW00] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. *Proceedings of ACM SIGCOMM*, pages 43–56, 2000.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[FJ95] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 1(4):365–386, 1995.

[FMMP00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. *RFC 2883: An Extension to the Selective Acknowledgement (SACK) Option for TCP*. Internet Engineering Task Force, July 2000.

[GGPS96] L. Georgiadis, R. Guérin, V. Peris, and K. Sivarajan. Efficient network QoS provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking*, 4(4):482–501, 1996.

[Gri] Access Grid. http://www.accessgrid.org.

[H+99] M. Hicks et al. Plannet: An active internetwork. *Proceedings of INFOCOM'99*, pages 1124–1133, March 1999.

[HFPW03] M. Handley, S. Floyd, J. Padhye, and J. Widmer. *RFC 3448: TCP Friendly Rate Control (TFRC): Protocol Specification*. Internet Engineering Task Force, January 2003.

[HMTG01] C. V. Hollot, Vishal Misra, Donald F. Towsley, and Weibo Gong. On designing improved controllers for AQM routers supporting TCP flows. In *INFOCOM*, pages 1726–1734, 2001.

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

[Jac88] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[Ken98] C. Kenjiro. A framework for alternate queueing: Towards traffic management by PC-UNIX based routers. In *USENIX 1998*, pages 247–258, June 1998.

[KHF04] Eddie Kohler, Mark Handley, and Sally Floyd. *Internet Draft: Datagram Congestion Control Protocol (DCCP)*. Internet Engineering Task Force, November 2004.

[KK90] C. R. Kalmanek and H. Kanakia. Rate controlled servers for very high-speed networks. *Proceedings of the Conference on Global Communications (GLOBECOM)*, pages 12–20, 1990.

[KMJ+] S. Keshav, S. McCanne, S. Jamin, K.K. Ramakrishnan, R. Sethi, D. Ferrari, and S. Shenker. *REAL Network Simulator*. http://www.cs.cornell.edu/skeshav/real/overview.html.

[KSC91] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communication*, 9(8):1265–1279, 1991.

[KW99] H.T. Kung and S.Y. Wang. TCP trunking: Design, implementation and performance. *Proc. of ICNP '99*, November 1999.

[KZM+03] 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.

[LAJS03] Long Le, Jay Aikat, Kevin Jeffay, and F. Donelson Smith. The effects of active queue management on web peformance. *Proceedings of ACM SIGCOMM 2003*, August 2003.

[Lan86] K.A. Lantz. An experiment in integrated multimedia conferencing. *Proceedings of ACM Conference on Conference on Computer Supported Cooperative Work (CSCW '86)*, 1986.

[LDE+97] Jason Leigh, Thomas A. DeFanti, Andrew E.Johnson, Maxine D. Brown, and Daniel J. Sandin. Global tele-immersion: Better than being there. In *7th Annual International Conference on Artificial Reality and Tele-Existence*, December 1997.

[LGT98] Li-Wei H. Lehman, Stephen J. Garland, and David L. Tennenhouse. Active reliable multicast. In *INFOCOM (2)*, pages 581–589, 1998.

[Mah97] Bruce A. Mah. An empirical model of HTTP network traffic. In *INFOCOM (2)*, pages 592–600, 1997.

[MJV96] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *ACM SIGCOMM*, volume 26,4, August 1996.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *RFC 2018: TCP Selective Acknowledgement Options*. Internet Engineering Task Force, April 1996.

[N+99] E. Nygren et al. Pan: A high-performance active network node supporting multiple code systems. *Proceedings of OPENARCH'99*, 1999.

[Ols03] Robert Olson. *Access Grid Hardware Specification*. Argonne National Laboratory, July 2003. http://www.accessgrid.org/agdp/guide/spec.html.

[OMP02] D. Ott and K. Mayer-Patel. A mechanism for TCP-friendly transport-level protocol coordination. *USENIX 2002*, June 2002.

[OMP04] D. Ott and K. Mayer-Patel. Coordinated multi-streaming for 3D tele-immersion. *ACM Multimedia 2004*, October 2004.

[OSMP04] D. Ott, T. Sparks, and K. Mayer-Patel. Aggregate congestion control for distributed multimedia applications. *Proceedings of IEEE INFOCOM '04*, March 2004.

[Pae95] Alan W. Paeth. *Graphics Gems V (The Graphics Gems)*. Morgan Kaufmann, 1995.

[Pax97] V. Paxson. End-to-end internet packet dynamics. *Proceedings of ACM SIGCOMM*, 1997.

[Pax99] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[PCN00] P. Pradhan, T. Chiueh, and A. Neogi. Aggregate TCP congestion control using multiple network probing. *Proc. of IEEE ICDCS 2000*, 2000.

[PFTK98] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. *Proceedings of ACM SIGCOMM*, 1998.

[PG93] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[PJS99] M. Parris, K. Jeffay, and F. Smith. Lightweight active router-queue management for multimedia networking, 1999.

[PMM93] C. Partridge, T. Mendez, and W. Milliken. *RFC 1546: Host Anycasting Service*. Internet Engineering Task Force, November 1993.

[Pos81] Jon Postel. *RFC 793: Transmission Control Protocol*. Internet Engineering Task Force, September 1981.

[RF99] K. Ramakrishnan and S. Floyd. *RFC 2481: A Proposal to add Explicit Congestion Notification (ECN) to IP*. Internet Engineering Task Force, January 1999.

[RHE99] 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.

[Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM CCR*, 27(1):31–41, January 1997.

[RWC+98] 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 SIGGRAPH*, 1998.

[S+99] B. Schwarts et al. Smart packets for active networks. *Proceedings of OPENARCH'99*, 1999.

[SCJO01] 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.

[SK02] P. Sarolahti and A. Kuznetsov. Congestion control in linux tcp, 2002.

[SLR98] H. Schulzrinne, R. Lanphier, and A. Rao. *RFC 2326: Real time streaming protocol (RTSP)*. Internet Engineering Task Force, April 1998.

[SRC84] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[TKS+03] 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 Worksthop on Collaborative Virtual Reality and Visualization (CVRV 2003)*, October 2003.

[TW96] D. L. Tennenhouse and D. Wetherall. Towards an active network architecture. *Multimedia Computing and Networking*, January 1996.

[vdM+98] J. van der Merwe et al. The Tempest - a practical framework for network programmability. *IEEE Network Magazine*, 12(3), May/June 1998.

[Wet99a] D. Wetherall. Service introduction in an active network, 1999.

[Wet99b] David Wetherall. Active network vision and reality: lessons from a capsule-based system. *Operating Systems Review*, 34(5):64–79, December 1999.

[Wid00] Jorg Widmer. *Equation-Based Congestion Control*. PhD thesis, University of Mannheim: Dept of Mathematics and Computer Science, February 2000.

[Wil04] Lori Wilkerson. *The History of Video Conferencing*. Evaluseek Publishing, 2004.

[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

[XMS+00] Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. *RFC 2960: Stream Control Transmission Protocol*. Internet Engineering Task Force, October 2000.

[YdS96] Y. Yemini and S. da Silva. Towards programmable networks. *International Workshop on Distributed Systems Operations and Management*, October 1996.

[Zha95] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of IEEE*, 83(10):1374–96, October 1995.