TECHNIQUES AND PROTOCOLS FOR DISTRIBUTED MEDIA STREAMING

Ma Lin

(Ph.D.)

National University of Singapore

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2007

Acknowledgements

First of all, I would like to thank my advisor Dr. Ooi Wei Tsang, without whose guidance, both intellectual and emotional, I could not have completed my Ph.D. degree. He lead me to the door into the world of research, handed me the torch that illuminated a few steps ahead in the unknown world, tolerated my mistakes, and fortified my mind when I felt helpless.

I would also like to express my gratitude to Prof. A.L. Ananda, Dr. Chang Ee-Chien, and Dr. Wang Ye. They shared with me their wisdom of teaching and doing research, and encouraged me on every step forward during the candidature.

I cherish the time together with my fellow lab mates: Liu Yanhong, Gu Yan, Cheng Wei, Satish Verma, and Pavel Korshunov. Their constant encouragement and willingness until discuss helped me to insist to the end of the candidature.

The Department of Computer Science, National University of Singapore offered me the scholarship and a good place to study. This offer changed my life so much that I will always be thankful during the rest of my days.

I would like to thank Xiaoran, for sharing my joy and sadness, and for giving her sweet and patient love during my long march.

Finally, I am forever indebted to my parents and my family.

Table of Contents

TECHNIQUES AND PROTOCOLS FOR DISTRIBUTED MEDIA STREAMING

Ma Lin, Ph.D.

National University of Singapore 2007

Distributed media streaming employs multiple senders to cooperatively and simultaneously transmit a media stream to a receiver over the Internet. Having multiple senders have lead to both sender and path diversity and improved robustness in the system. But at the same time, distributed media streaming has raised many challenging and interesting research problems. In this dissertation, we investigate several of these problems that are related to media quality and fairness to other applications.

First, we study how streaming quality can be improved through distributed retransmission – retransmission from alternate senders rather than the origin of the lost packet. We explore the question of *whether distributed retransmission recovers more packet loss than non-distributed retransmission* by comparing two naive distributed retransmission schemes with the traditional non-distributed scheme. Through analysis, simulations, and experiments over the Internet, we found that distributed retransmission leads to fewer lost packets and shorter loss burst length. To address the practical issue of who to retransmit from, we propose a distributed retransmission scheme that selects a sender with the lowest packet loss rate to retransmit from. Results show that our proposed scheme effectively recovers packet losses and improves playback quality.

Second, we investigate the issue of TCP-friendliness in distributed media streaming. The traditional notion of TCP-friendliness is not suitable for multi-flow ap-

plications, such as distributed media streaming, as it is unfair to other single-flow applications. We therefore introduce the notion of task-level TCP-friendliness for distributed media streaming, where we require the total throughput for a set of flows belonging to the same task to be friendly to a TCP flow. To this end, we design a congestion control protocol to regulate the throughput of the flows in an aggregated manner. The regulation is done in two steps. First, we identify the bottlenecks and the subset of flows on the bottlenecks. Then, we adjust the congestion control parameter such that the total throughput of the subset is no more than that of a TCP flow on each bottleneck. Network simulation using multiple congestion scenarios shows the efficiency of our approach.

Third, we propose an unreliable, congestion-controlled transport protocol for media streaming, called TCP Urel. TCP Urel sends fresh data during retransmissions, and therefore keeps the congestion control mechanism of TCP intact. TCP Urel is simple to implement. We realized TCP Urel based on the existing TCP stack in FreeBSD 5.4, with less than 750 lines of extra code. Our experiments over a LAN testbed show that TCP Urel is friendly to different TCP versions and introduces little CPU overhead.

Biographical Sketch

Ma Lin was born in January, 1980 in the city of Hangzhou in Zhejiang Province, China. After he completed his secondary education at the Affiliated Middle School of Zhejiang University in 1998, he went on to pursue his undergraduate degree in the Department of Computer Science and Engineering, at Zhejiang University. He graduated with a Bachelor Degree in Computer Science in 2002, and then moved to Singapore to pursue a Ph.D. degree in School of Computing, National University of Singapore.

To Grandma.

Chapter 1

Introduction

The Internet, since its evolution from ARPANET in 1980s, has grown rapidly and has tremendously improved people's life in many aspects. The Internet traffic increases exponentially over the years [16]. Multimedia applications are among the most fascinating applications that fuel the growth of the Internet. One of these applications is Video on Demand (VOD) service, which streams multimedia content on demand over the Internet.

1.1 Background

Unlike bulk data transmission such as file transfer, realtime multimedia streaming has several distinguishing characteristics. First, media streaming is delay-sensitive. Packets arriving after its playback deadline cannot be played back. Second, multimedia data consumes large amount of bandwidth. For instance, an MPEG-4 video typically consumes 56Kbps to 2Mbps bandwidth [78]. Third, multimedia streaming tolerates some degree of data loss during transmission [24, 78]. These characteristics require supports on delay guarantees, bandwidth reservation, and flexible error control, which are not provided by the current Internet. VOD has an additional requirement that playback should start as soon as possible after a

user request.There are three communication models for VOD: unicast, multicast, and multipath streaming. Each model has its own weaknesses that hinder scalable delivery of VOD service.

1.1.1 Multimedia Streaming Models

The unicast model uses a streaming session between one sender and one receiver via one path, carrying either unidirectional or bidirectional traffic. This model is widely used because of its simplicity. For example, VoIP applications such as Yahoo Messenger and web-based VOD services such as Google Video use this model. In web-based VOD services, when a user clicks the start button on the web page, the client builds a connection to the server, which then streams[1] video content via the connection. As the number of session requests increases, the output bandwidth at the server becomes a bottleneck. VOD over unicast is not scalable. For instance, to scale their VOD service to millions of users, Google Video replicates video contents in multiple servers located at the edge of the Internet, reducing the burden on each server.

The multicast communication model uses a streaming session involving one sender and multiple receivers. The sender does not maintain one connection to each receiver. Instead, the stream is replicated at intermediate nodes along the path for distribution to the receivers. This way, the sender is able to serve multiple receivers while sending only one stream, significantly reducing the sender's outgoing bandwidth requirement. Two types of multicast exist, based on the layer in which the intermediate nodes reside. In *IP multicast* [94], the intermediate nodes are multicast-enabled routers. These routers support group management, packet

[1]More precisely, the video is progressively downloaded using HTTP protocol. In this dissertation we do not discriminate streaming with RTP or transmission with HTTP.

replication, and routing. The other type is *application-layer multicast* [79], which uses end host as intermediate nodes and implements functionalities of group management, packet replication, and routing in the application layer. It requires no changes to the routers in the current Internet, and therefore, can be deployed more easily.

The other multimedia streaming model is *multipath streaming*. As suggested by the name, it employs multiple (ideally uncorrelated) physical paths between the sender and the receiver and streams multimedia content by multiple flows on these paths [11, 14, 31, 33, 57]. Comparing to unicast, multipath streaming has the following advantages: (i) By scattering packets among different paths, it reduces the lost correlation between consecutive packets, hence reduces the quality impairment from burst loss on single path; (ii) it increases the throughput by using multiple flows; and (iii) with the heterogeneous channels, it offers the choice to prioritize which media data to send onto which paths and to adapt to dynamic network conditions. Nevertheless, multipath streaming still cannot scale, since, like unicast it uses one sender and one receiver in a session. When the number of receivers increases, the outgoing bandwidth at the sender becomes the bottleneck.

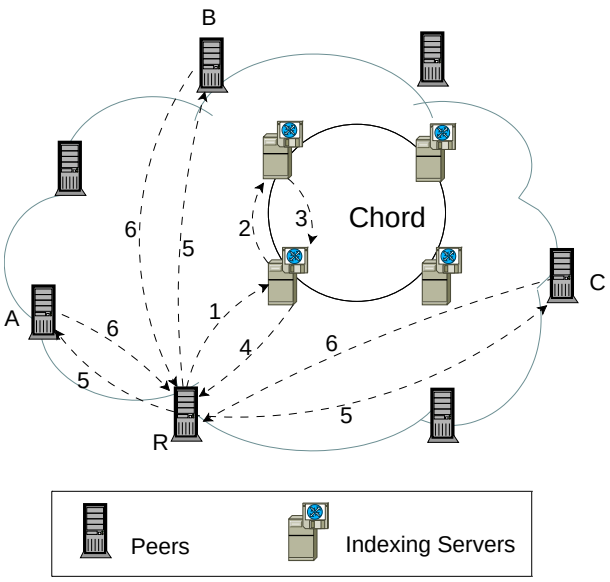
To provide scalable VOD service, we need a new model to disperse the streaming burden to multiple senders and to relieve the outgoing bottleneck at the sender. This approach is largely similar to that used for peer-to-peer (P2P) data sharing, which we will introduce next.

1.1.2 P2P data sharing

Despite legality issues, data sharing over P2P networks has exploded in the past few years. Since 1999, the percentage of P2P traffic on the Internet increases exponentially every year. According to a report from CacheLogic [9], by the end of

2004, P2P has taken up 60% of the total Internet traffic. The same report points out that more than 88.6% of the P2P traffic is for multimedia data. Large share of multimedia data in the P2P traffic implies a huge demand for multimedia content from the broadband home subscribers. In P2P data sharing, peers download data from several other peers. A peer acts as a client when it downloads data and acts as a server when it uploads data. Peers are end hosts on the Internet; they exchange data through ad hoc connections, which weave up an overlay of peers. Such overlay is scalable by nature: if a piece of data is popular, the number of receivers increases; these receivers, in turn, become potential senders later and contribute their storage and bandwidth to the overlay [84]. This large number of potential senders provides high scalability and makes P2P data sharing tremendously successful.

Many P2P overlays [54] make use of *distributed hash table*, which allows indexing of the resources, including regular files and multimedia data. For instance, in an indexing ring with N nodes, *Chord* [88] can locate a file in $log(N)$ rounds of message passing. These indexing techniques allow a user to efficiently locate available senders of a particular resource (e.g. a video clip in VOD).



Messages in session setup:

1. Peer R sends a search query to the nearest indexing server;
2. The query is forwarded among the indexing servers;
3. Result of the query is returned to the nearest indexing server;
4. The result is returned to R;
5. R contacts peers that are listed in the result for session set up.
6. Some peers reply with confirmations. Then, the session starts.

Figure 1.1: Framework of a VOD service over P2P overlay

With the large number of peers in the overlay, a user has a high chance of finding a popular video clip. Scalability and asynchronous availability, both offered by P2P, match the needs of VOD. A possible P2P VOD service framework is shown in Figure 1.1. In this framework, videos are stored in the peers and are indexed by the indexing servers, which form a Chord ring. Similar indexing network can be found in practical systems, e.g., *ed2k* and *kad* in eMule. After finding the senders, the receiver sets up a streaming session and begin receiving video. Unlike unicast, multicast, and multi-path streaming, due to the abundance of peers in the overlay, P2P network can supply multiple senders to one receiver. We refer to a session that involves multiple sender and one receiver as a *distributed media streaming* session.

1.2 Distributed Media Streaming

Distributed media streaming uses multiple senders to simultaneously and cooperatively stream multimedia data to a single receiver. In the literature, it is also known as *multi-source streaming* [2]. The streaming session from A, B, C to R in Figure 1.1 is a distributed media streaming session. In the rest of this section, we discuss some features of this model.

1.2.1 Receiver-Driven Protocol

Distributed media streaming typically adopts a receiver-driven protocol [50, 68, 81, 97], where the receiver (i) initiates the streaming session; (ii) measures the network statistics such as loss rate, bit rate, and delay on the different channels; and (iii) decides who sends which part of data at what time, to achieve the best quality. These decision tasks are performed by the receiver for two reasons. First, the receiver is the consumer of the media, hence it is fair for the receiver to spend

resources (CPU power, memory, and bandwidth) to decide how to stream. Second, since only the receiver communicates with all the senders, network statistics tracked at the receiver can be used for decision making without communication overhead among the senders.

Chakareski and Frossard [10] designed a sender-driven protocol for distributed media streaming, believing that the packet dependency is known prior at the senders rather than the receiver, therefore determining which packets to be sent by which sender should be carried out at the senders. Although this design shifts part of the decision making from the receiver to the senders, the packet assignment algorithm is still driven by the channel statistics measured by the receiver.

1.2.2 Advantages

Compared to unicast, multicast, and multipath streaming, distributed media streaming model is better suited for VOD service.

First, having multiple senders allows each sender to contribute less upload bandwidth in a session than having only a single sender. Requiring smaller upload bandwidth than the download bandwidth in the session matches well with the asymmetric download/upload bandwidth capacity in current broadband deployment. For instance, ADSL (ITU G.992.1) has a 8Mbps downstream and 1Mbps upstream, and CDLP (a proprietary Cable Modem standard made by Motorola) provides 10Mbps downstream and 1.532Mbps upstream[2]. Lowering the sending rate also matches users' general unwillingness to contribute uploading bandwidth in P2P system [23], and therefore can attract broader user base, which is essential to building a large scale P2P network and serving a large number of clients.

Second, although each sender contributes less bandwidth, contribution from

[2]http://en.wikipedia.org/wiki/Cable_modem

multiple senders allows aggregation of bandwidth. In single-sender models, the streaming rate is limited by the upload rate at the sender, which is likely to be smaller than the download bandwidth due to the asymmetric links and users' unwillingness to contribute. Distributed media streaming therefore can support higher streaming rate than single-sender models.

Third, while the failure of the sender in single-sender models stops media streaming to all the receivers completely, the same scenario causes less disruption to distributed media streaming. Data are still being received from other senders, allowing playback at a lower quality if proper coding methods are used. Therefore, distributed media streaming is more robust to sender failure than single-sender models.

Fourth, since distributed media streaming employs multiple channels, when one channel is congested, the receiver can still receive data through other channels. Diversified paths reduce packet loss correlation and impairment from burst loss when proper error recovery techniques [55] is used. This advantage is also exploited by multipath streaming [31]. We expect distributed media streaming to deliver better media playback than single-channel models in a lossy network.

1.3 Research Challenges

As they said, however, "*there ain't no such thing as a free lunch*[3]". While distributed media streaming has many advantages, it also brings new challenges. We highlight the major challenges below.

Sender Selection Given a potentially huge set of sender candidates returned by the indexing network (e.g. the Chord ring in Figure 1.1), we need to select some

[3] "There ain't no such thing as a free lunch." – R. A. Heinlein, *The Moon Is a Harsh Mistress*

of them as senders in the distributed streaming session. Many factors affect the selection. For instance, if each sender only stores part of the media content (as the case in Cui and Nahrstedt [19]), the selected group of senders must together provide the whole media content under request. Sending rate of the candidates is another factor: the selected group of senders should output a combined bit rate no less than the playback bit rate of required media under request. Other factors include delay, packet loss rate, and path diversity. Lower delay provides lower response time; lower packet loss rate leads to higher media playback quality; and higher diversity in paths from the senders to the receiver leads to fewer simultaneous packet losses.

Rate Allocation Rate allocation decides how fast a sender should send. Rate allocation can be considered in conjunction with sender selection: the total sending rate must exceed the minimum playback requirement. After the senders are selected and the session starts, the rate may also be dynamically adjusted among the senders to perform congestion control, to maintain the combined bit rate when one sender is in severe congestion, and to avoid overflowing the receiver buffer.

Data Assignment Data assignment determines which sender should send which part of the media content. A sender can send certain layer(s) for a multi-layer video, or certain chunk(s) in a single-layer video, or certain packet(s). In general, data assignment takes the rate allocated to the senders and the set of data units as inputs and computes a mapping between the data units and senders, optimizing the received media quality.

Error Recovery Error recovery is important as it reduces the quality impairment caused by packet loss on the Internet, and hence offers better perceptual quality at the receiver. Due to the existence of multiple senders, distributed media streaming can scatter FEC blocks among the senders or choose different senders

for retransmission. By avoiding correlated packet loss on same channel, error recovery schemes in distributed media streaming can improve the media quality significantly.

Congestion Control Congestion control is key to maintain the Internet stability [26]. Continuous streaming application like video streaming must be congestion controlled, so that their deployment do not unfairly compete with other network flows. The formulation of congestion control in distributed media streaming, however, is different from the one in single-sender models. Because multiple flows are involved, TCP-friendliness, the common goal of congestion control, needs to be redefined. Besides, due to the reverse tree topology in distributed media streaming, the congestion control scheme needs to dynamically adapt to the different bottlenecks in the tree.

Transport Protocol Distributed media streaming employs multiple flows to deliver media data cooperatively. A transport protocol is needed to stream each of these flows. Although congestion-controlled transport protocol in single path streaming is well studied, distributed media streaming has its own requirements that are not satisfied by existing protocols. First, the protocol needs not be reliable. Second, the protocol should notify the application about packet loss. The application can then recover losses based on its own policy.

Among the above problems, the first three have been well studied in existing literature [19, 35, 68, 81, 97]. The next two, however, are not studied previously. Due to their importance in constructing a usable and practical distributed media streaming system, we chose them as the topics of this dissertation. We also designed a transport protocol to satisfy the requirements of distributed media streaming, providing a solution to the last problem.

1.4 List of Contributions

The contributions of this thesis are as follows.

1.4.1 Retransmission for Distributed Media Streaming

We study the effectiveness of retransmission from senders other than the one that loses the packet and propose a retransmission scheme for distributed media streaming. Our scheme dynamically switches retransmitters when congestion appears, and selects the channel with the lowest loss rate for retransmission. By doing so, we successfully reduce the quality impairment caused by burst loss. We present the detail in Chapter 3.

The dynamic retransmission scheme is the first attempt to exploit path diversity in retransmission. The model and discussion in this study also apply to multipath streaming, which also streams via multiple channels concurrently.

1.4.2 Congestion Control for Distributed Media Streaming

We propose DMSCC, a congestion control scheme for distributed media streaming. We study existing measurement on congestion control and define a new notion of task-level TCP-friendliness for multi-flow applications: depending on the location of bottlenecks, the application flows in the bottleneck should offer a combined throughput that is TCP-friendly. We design DMSCC to achieve this goal. DMSCC has two relatively independent functionalities: throughput control and congestion location. When congestion occurs, the congestion location module identifies the bottleneck by observing the correlations among the one-way delay variation of the channels. The throughput control module then updates the increasing factor of AIMD loops of each flow on that bottleneck, so that the combined flow is friendly to other TCP flows on the same bottleneck. Our simulation shows that DMSCC is

able to achieve task-level TCP-friendliness in different congestion scenarios. Chapter 4 presents the details of this work.

DMSCC is the first congestion control method designed for distributed media streaming system. It is the first congestion control scheme that considers the changing location of congestion in a topology of reverse tree. The method of adjusting increasing factors to achieve certain bandwidth share of a TCP flow is also useful to other applications that need aggregate congestion control.

1.4.3 TCP Extension for Unreliable Streaming

Generally, TCP is regarded as unsuitable for continuous multimedia streaming. The reasons are that: (i) the sawtooth-like rate adaptation impairs the smoothness of the media quality, and (ii) the automatic retransmission can cause unbounded packet delay. For non-interactive applications such as VOD, the bit rate can always be smoothed by a receiving buffer; therefore, the issue of sawtooth-like bit rate fluctuation is not important. To tackle the second concern, we design a new option for TCP: TCP Urel, which does not retransmit when packets are lost, but maintains the congestion control operations of a TCP flow at the same time. To help application-level error recovery, TCP Urel also informs the application about the lost data, allowing error decision at the application layer.

TCP Urel improves the TCP friendliness of previous attempt to modify TCP into unreliable streaming protocol [63]. Compared to other existing protocols supporting unreliable but congestion controlled data delivery [41, 87], TCP Urel is a simple, easy to use alternative that can be used by multimedia streaming and other loss-insensitive applications over the Internet. Detail of TCP Urel will be presented in Chapter 5.

1.5 Structure of This Thesis

This thesis is structured as follows. Chapter 2 presents existing work in distributed media streaming and gives a detailed review of the field. Chapter 3 presents our study on retransmission in distributed media streaming. Chapter 4 describes DMSCC, the congestion control scheme for distributed media streaming. Chapter 5 presents TCP Urel, the TCP extension for unreliable streaming. We conclude this thesis in Chapter 6.

Chapter 2

Background and Related Work

Since 2002, distributed media streaming has been an active research topic. Several research groups identified many problems from different perspectives, under different network context, data types, and design objectives.

In this chapter, we present an overview of the existing work. First, we categorize these work according to their network models and data models. Then, we organize them according to their goals and present the schemes proposed for different network and data models. As a summary, we show a map of existing research at the end of this chapter. In the map, we also indicate how this thesis fits into the overall picture.

2.1 Network Models

Distributed media streaming can be used in different networks. The senders may be servers in a Content Delivery Network [3], end hosts in a peer-to-peer (P2P) overlay network [97], mobile users in a wireless LAN (WLAN) [48], or randomly scattered mobile nodes in a wireless mesh [49]. These networks can be simplified and abstracted as nodes (senders, receiver, and routers) and links (wired and wireless). In this section, we elaborate on these networks and how they are modeled

in existing work.

2.1.1 CDN

A CDN consists of a group of servers placed strategically across the Internet, often deployed over multiple backbones for high availability. They cooperate to deliver media content to the users, transparent to the clients [36]. By carefully placing the servers, CDN brings content nearer to the client, reducing response time and distributing load across the servers. In a CDN, servers typically are provisioned to have enough bandwidth to support unicast to the receivers. But we can still use multiple senders to reduce quality degradation when some links are congested.

Servers in CDN have large storage, fat pipe, and are highly available. So research efforts focus on modeling of the links, whose bandwidth, loss rate, and delay affect the media quality. Apostolopoulos et al. [3] model each link with a Gilbert model. A path from a sender to the receiver is a concatenation of several links. Paths from multiple senders start with different links at first, but merge with each other later, sharing the same links closer to the receivers. The authors study streaming from two senders and model the paths with three Gilbert models, one for each disjoint sub-path from the two senders, and one for the common sub-path. These three Gilbert models are equivalent to a 8-state Markov Model identified by an 8×8 transitional matrix. With this model, they capture the pattern of packet losses caused by either shared or independent congestion.

2.1.2 P2P

A P2P overlay relies primarily on the computing power and bandwidth of the end hosts in the overlay network. Unlike CDN, P2P overlay is decentralized: a pure P2P overlay does not have notions of clients or servers, but only peers, which

function as both "servers" and "clients" to other peers in the overlay [84].

Peers

Unlike dedicated servers (e.g., those in a CDN network), peers are typically broadband home users with limited uploading bandwidth. Therefore, one factor in modeling P2P network is the bandwidth of each sending peer. This factor is considered by Nguyen and Zahkor (2002a) [68], which adds peers to the sender set until the combined bandwidth of selected senders exceeds the requirement for streaming.

Senders can leave a session at anytime, as the action of end hosts is not predictable. When a sender leaves, the media quality degrades. The probability that a peer leaves during a session can be modeled using an on/off probability [35].

As the receivers will become senders after receiving the media data, the sending rate of a receiver is also a variable to be considered in the model. In the initial phase, there are only a few seeds. Thus, it is hard to serve many request simultaneously due to the limited bandwidth available from the seeds. To address this issue, Xu et al. [97] selects receivers with higher sending rate to serve.

Links

P2P overlay and CDN share similar link properties. Hefeeda et al. [35] model the paths as concatenation of links, with possible sharing of links among paths. Nevertheless, unlike Apostolopoulos et al. [3], who only consider packet loss, Hefeeda et al. model each link's bandwidth, packet loss rate, and delay. With these parameters, the authors can determine how much data can be transmitted in a period, how many packets will be lost, by how long would they be delayed, and to what extent these losses and delay would be correlated between two paths. By combining these information, they are able to select the best paths (hence the senders) to maximize the playback quality of a streaming session.

Estimating the parameters (bandwidth, loss rate, and delay) of each link, especially before streaming has begin is difficult. Estimation error of a parameter on one link may accumulate on links along the path and diminish the accuracy of the path model significantly. Instead, many researchers simplify the modeling of paths. Nguyen and Zahkor [68,69], and Xu et al. [97] model the paths as independent links with loss rate, which is measurable by end-to-end methods. Rejaie and Ortega [81] measure the bandwidth of these paths and estimate future bandwidth using TFRC [34].

2.1.3 Hybrid

CDN is more reliable when servers are not overwhelmed, whereas P2P scales better if videos are popular and are requested by many peers. To combine the merits from both sides, hybrid system with both centralized servers and decentralized peers are designed [6]. In such system, reliable servers can take over when a peer fails. While establishing a new connection from the replacement peer, the server helps significantly in reducing quality degradation.

Such system is a combination of the CDN and P2P models. Cui and Nahrstedt [19] use the servers with large bandwidth and large storage and characterize the peers as nodes with limited bandwidth and limited storage. The links of such network are the same as in CDN and P2P systems.

2.1.4 WLAN

Distributed media streaming may also be deployed over WLAN. WLAN differs from wired network as nodes in the same WLAN shares the same access point. Each connection to and from the nodes in the WLAN passes through the access point, even for communication between nodes in the same WLAN.

Li et al. (2005) [48] model the path between a sender and the receiver as the concatenation of two links, one from the sender to the access point, the other from the access point to the receiver. Since access point knows about the signal strength to and from the peers, based on which accurate estimation of loss rate, bandwidth, and delay of a link is possible, the authors place a proxy on (or near) the access point to coordinate distributed streaming.

Senders in the WLAN are not only characterized by their sending rate, but also the mobility. Li et al. (2005) [48] let the proxy trace the mobility of the nodes by looking at the changing of signal strength, and estimate the effects of mobility on the link quality for the next period of time.

2.1.5 Wireless Mesh

Wireless mesh is another type of wireless network, in which nodes are interconnected with each other to form a mesh. A node may have more than one neighbors; it not only receives packets from them but also relays packets to them. A node is both a consumer of packets and a router. Compared to WLAN, a wireless mesh is more extensible as each newly joined node expands the bandwidth and range of the network. It is also more robust since packets could be routed through another path if an intermediate node fails. In WLAN, in contrast, access point is the single point of failure.

Assuming nodes are cooperative, the end-to-end bandwidth is determined by the link bandwidth rather than the nodes' contributed output rate. The network modeling of wireless mesh focuses on the links rather than the nodes. Li et al. (2006) [49] model a wireless mesh as a time-varied directed graph. A directed edge captures a link and its loss rate, bandwidth, and delay. A edge exists only when the distance between two nodes is within the communication range.

Unlike wired network, links in wireless network suffer from interference. One simple model of interference is that, a link from node A to node B exists only when other nodes having B in their communication range do not transmit to B. Li et al. (2006) [49] adopt this interference model and construct a conflict matrix, whose indices are the links and whose elements denote whether two links interfere with each other.

2.2 Data Models

Considering the distance, loss rate, bandwidth, delay, and interference, links are concatenated to form paths and selection can be made to decide the best senders and routes for streaming

2.2.1 Single-Layer Coding

Single-layer coding (such as MPEG-2 [1]) codes video data into frames ordered by time. Each frame should be played back at a deadline, and frames are modeled as a sequence of packets ordered by playback time (display order). Packets may depend on other packets according to their frame type (Figure 2.1(a)). While transmitted via one single path, referred packets are transmitted before referring packets (coding order). Current work in distributed media streaming further simplify this model and assume that packets have the same size and are independent (Figure 2.1(b)), e.g., Nguyen and Zahkor (2005a) [68]. Although many studies concerning the dependency among frames and packets exist in the case of single path streaming and multi-path streaming [11, 38], there are no corresponding research in distributed media streaming.

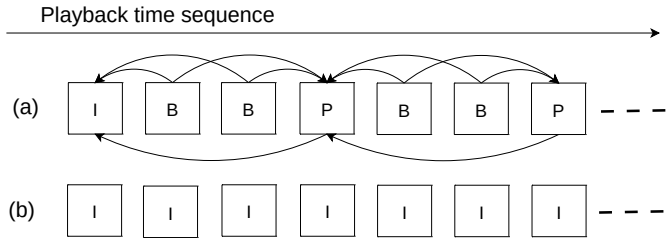


Figure 2.1: Packet dependency of single-layer coding: (a) dependent packets, (b) independent packets (assuming one frame per packet)

2.2.2 Multi-Layer Coding

Multi-layer coding [30] encodes the video data into two or more layers. The lowest layer (base layer) is essential to decoding but only produces a low quality video. The higher layers enhance the video quality after the lower layers are decoded. When bandwidth is limited the sender can drop the highest layers and offer a lower quality but continuous video playback. Multi-layer video are modeled such that (i) higher-layers packets depend on lower-layer packets, and (ii) frame-level dependencies are preserved among packets (Figure 2.2). Dependencies among packets have been studied to optimize the streaming quality in single path [52] and multi-path streaming [71], but there are no comparable study in distributed media streaming. Existing work based on multi-layer video [19, 81] only study dependency at the granularity of layers rather than packets.

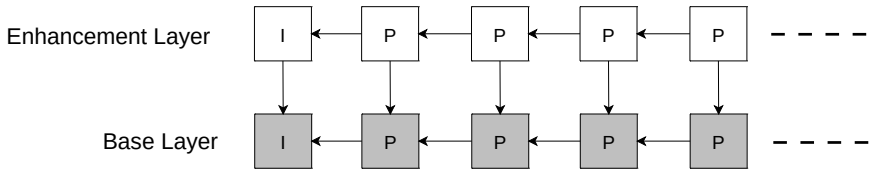


Figure 2.2: Packet dependency of multi-layer coding (2 layers)

2.2.3 Fine Granularity Scalable Coding

Fine granularity scalable coding (FGS) provides finer granularity on quality degradation. Unlike multi-layer coding, in which a enhancement layers has to be fully received to improve quality, FGS utilizes every received bit in enhancement layer to improves quality [51]. FGS shares similar packet dependency graph as multi-layered coding, therefore the problem of assigning layers to senders for better quality or lowest server burden may have similar solutions. For example, Cui and Nahrstedt [19] and Hsu [37] minimize the server's burden while streaming multi-layer video and FGS video in a hybrid network using distributed media streaming respectively using similar solutions.

2.2.4 Multiple Description Coding

Both multi-layer coding and FGS encode the media into layers with dependency: enhancement layers cannot be decoded if the base layer is not received. This model prioritizes lower layers over higher layers and does not fit the characteristics of the Internet, which is best effort and does not prioritize packets. Multiple description coding (MDC) produces multiple streams called descriptions that are independent from each other. When combined, these streams output video with higher quality. This nice property, however, costs higher bandwidth [77]. According to the study by Lee et at. [46], multi-layer coding gives better quality when bandwidth is less than the full-quality playback rate; single-layer coding has better quality when bandwidth is greater than the full-quality playback rate; and MDC gives worse quality in both cases. The merit of using MDC in distributed media streaming, however, is that we need not worry about which sender should send which description, since all descriptions are equally important. So the concern focuses on the selection of senders, which leads to different paths with different correlation.

The correlation could produce simultaneous channel loss or delay, causing quality degradation. Apostolopoulos et al. [3] selects paths that are maximally disjoint to reduce the correlation among the paths.

2.2.5 Forward Error Correction

FEC is normally regarded as a technique to recover packet loss, rather than a coding scheme for video. But since FEC packets may recover data, video quality can be improved by deciding the sending sequence (probably interleaving) of the data and FEC packets. As multiple paths exist, distributed media streaming raises the problems of which sender should send FEC packets, how much redundancy should be added, and how can FEC packets interleave with data packets. These problems lead to different rate allocation algorithms without [68] or with [69] FEC in the work by Nguyen and Zahkor.

2.3 Goals and Methods

The design of a distributed media streaming system depends on the network model, the data model, and the design goals. Existing work can be categorized according to their design goals: (i) minimum distortion, (ii) shortest buffering delay, (iii) minimum server load, and (iv) fastest service capacity growth. In this section, we shall present and compare these existing work, organized by each goal.

One common way to evaluate video quality is the distortion of each frame. Distortion, which is calculated as the mean square error of the differences in signals, measures the mismatch between a transmitted and decoded frame to the original frame [13]. Distortion is introduced by three factors during transmission: (i) insufficient bandwidth, thus, only part of the data can be sent; (ii) packet loss, only part of the data sent are received, and (iii) unpredictable delay variation – data

arrives after the playback deadline cannot be played back. Several publications in distributed media streaming minimize distortion by reducing the impairments from these three aspects. We will go through them in this section.

2.3.1 Bandwidth-Distortion Tradeoff

The combined sending rate of the senders decides the number of bytes that the receiver can receive in a time unit. Given a higher sending rate, more data is likely to be decoded, and therefore lower distortion can be achieved. The system designer can either optimize by constraining the maximum bandwidth or maximum distortion. The trade-off between bandwidth and distortion has been explored from three aspects.

The first question related to the trade-off is that, given a certain amount of bandwidth, how to maximize quality? The solution to this question depends on the data model and network model.

Nguyen and Zahkor (2002a) [68] transmits single-layer video packetized into fixed-size packets, with no dependency among the packets. In this simple data model, the distortion increases as the packet loss rate increases. Minimizing the distortion is therefore equivalent to minimizing the overall packet loss rate, which can be achieved by selecting senders with the lowest packet loss rate until the combined bandwidth reaches the requirement.

Rejaie and Ortega [81] transmits multi-layer coded video, where a layer is decoded only when all layers below are decoded. The distortion, hence, is related to the amount of *decode-able* data from different layers. In their system, called PALS, the receiver tracks and estimates the senders' bandwidth in the next time window. When the combined estimated bandwidth is higher than the playback rate, new layers are added; when it is lower, the top most layers are dropped.

The receiver also tracks the amount of received data in each layer and decides the bandwidth to allocate to each layer. These layers are weighted, and the combined bandwidth (counted as number of packet per window) are distributed to the layers according to the weights. The packets that fall into the window are ordered in a zigzag sequence, so packets in the lower layers are likely to be delivered first; therefore if the actual throughput is less than the estimated value, at lease the lower layers are sent. Another heuristics-based zigzag sequence is also proposed in their later work [2] to further improve the perceptual quality.

Li et. al (2005) [48] study distributed media streaming in a WLAN. The authors consider scenario where part of the multimedia content requested already resides on peers within the same WLAN[1], in which peers communicate via the same access point. Scalability in such a network is not the main issue, as the number of nodes is limited by the access point, which forms a bottleneck of the network. The authors set up a proxy near the access point to relay media data between the sender and the receiver for all distributed media streaming sessions. During a session, the proxy pulls data packets from the senders and push them to the client. To reduce packet losses due to limited bandwidth, the proxy only pulls data from a sender when both the sender-proxy and the proxy-client links have spare bandwidth. The frequency of pulling is determined by a timer, whose interval is inversely proportional to the link bandwidth estimated by TFRC. Given a selected sender-proxy link to deliver the next chunk of data, they employ a rate-distortion optimization framework derived from single path streaming [13] to schedule packets from that chunk on the sender-proxy link. Since wireless network is more error-prone, caching and retransmission are employed at the proxy, in order to conceal link-layer loss from

[1]The 802.11 has two basic modes of operation. Ad hoc mode enables peer-to-peer transmission between mobile units. Infrastructure mode, allows mobile units communicate via an access point. In Li et. al (2005) [48], infrastructure mode is under discussion.

the receiver. The proxy also handles joining and leaving of the peers and the setup for all streaming sessions.

The previous studies achieve minimum distortion under a given bandwidth constraint under different networks and data model. A question can be raised from the other angle of the trade-off between bandwidth and distortion: to reduce distortion to a certain level, what is the minimum bandwidth required? Majumdar et. al [58] solve the problem using bisection. Since number of packets to be sent is capped by the available bandwidth, and video quality increases as bandwidth increases, the authors try with half of the available bandwidth and find the best achievable quality – this is the same problem encountered by Nguyen and Zahkor (2005a) [68]. Depending on whether the achieved quality is higher or lower than the targeted quality, they subdivide the range of packet number and repeat the process in the upper half or lower half of the range, until the required quality is achieved.

2.3.2 Loss Rate-Distortion Tradeoff

Besides insufficient bandwidth, packet losses produce distortion as well. The loss rate-distortion trade-off in distributed media streaming is explored by three existing work [3,35,69], in different network, using different data model, and interpreting the trade-off differently.

The first work, by Nguyen and Zahkor (2005b) [69], considers the following scenario. Given different loss rate on different paths, what should the sending rate of each sender be? This problem is similar to that in Nguyen and Zahkor (2005a) [68]. But loss rate instead of bandwidth becomes the main factor that affects distortion since FEC protected packets are considered. Given a fixed level of FEC protection, the authors minimize the probability of irrecoverable loss by

determining the number of packets per FEC block that should be sent by each sender. Modeling each channel with a Gilbert model, the authors solve the problem for a two-sender case.

Hefeeda et. al [35] study the sender selection problem: Find a set of senders that minimizes the overall loss rate. Instead of assuming that each path is independent and selecting the senders based on end-to-end measurements (Figure 2.3), the authors propose that the common link among the paths should be considered (Figure 2.4). By inferring the approximate topology and measuring the available bandwidth on the links, they propose a topology-aware sender selection method, which selects senders with the highest quality. The quality of a sender, on the other hand, is weighted and calculated based on a packet loss model that considers peer availability, peers sending rate, and the available bandwidth along the path. The authors showed that their topology-aware sender selection delivers the lowest packet loss rate and the highest combined sending rate with or without peer failure, when compared to random selection and selection based on independent path assumption.

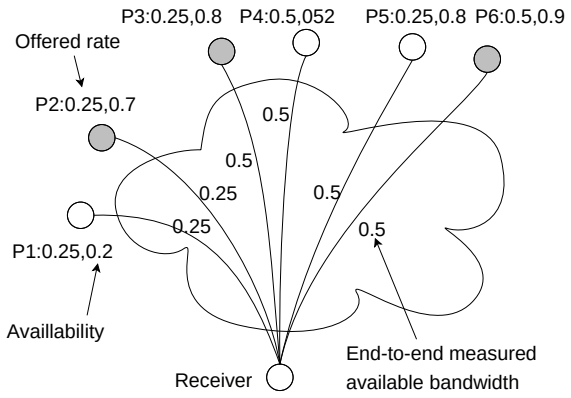


Figure 2.3: End-to-end selection, which does not consider shared segments (Figure excerpted from Hefeeda et. al [35])

Apostolopoulos et. al [3] apply MDC to distributed media streaming in a CDN

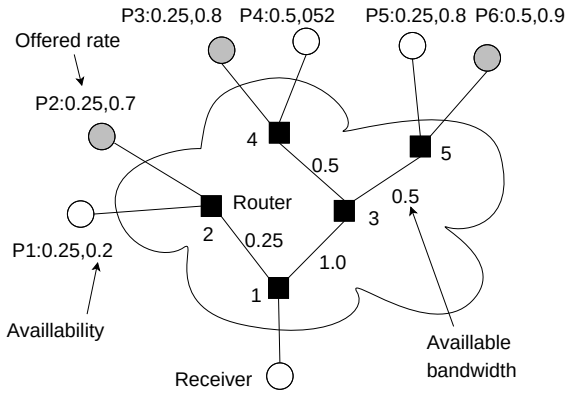


Figure 2.4: Topology-aware selection constructs an approximate topology and considers shared segments (Figure excerpted from Hefeeda et. al [35])

network. The authors recognize that the path from different servers to the same receiver may share congestion, which can produce simultaneous packet losses on different channels and increase the distortion by reducing the number of descriptions available for decoding. The authors propose a distortion model for MDC that takes path length and disjointness as input and computes the expected distortion. Based on this model, a set of servers that minimize the distortion are selected in the CDN networks as senders.

2.3.3 Delay-Distortion Tradeoff

Besides bandwidth and loss rate, delay is the third factor that introduces distortion. Two previous work [10, 68] consider this factor explicitly in distributed media streaming.

Nguyen and Zahkor (2005a) [68] study delay-distortion trade-off in the packet assignment problem. After deciding the sending rate of each senders, the receiver schedules packets among the senders. Packet assignment decides which packet should be delivered by which sender. Since multimedia playback is sensitive to delay; packets should arrive at the receiver as early as possible, so that the chance

of a packet missing its playback deadline is minimized. A packet therefore assigned to the sender that can deliver the packet to the receiver as early as possible. The expected arrival time of a packet from a given sender is estimated based on sending rate, round trip time, and the next time when the sender becomes available to send. The authors later extend the rate allocation algorithm to FEC protected single-layer media and applied the same packet assignment algorithm. The drawback of Nguyen and Zahkor (2005b) [69] (and [70]) is that it does not explain the packet assignment for FEC packets, whose time-based ordering is unclear. Time-based ordering, on the other hand, is vital for the packet assignment algorithm.

Chakareski and Frossard [10] also consider delay in the "sender-driven" model for distributed media streaming Figure 2.5. The authors propose that the receiver only collects the path information such as delay and packet loss rate. Instead of performing rate allocation and packet assignment at the receiver, their system sends these path information to all senders. The senders, in turn, independently calculate their sending rate and schedule the packets. During the computation, there is no communication among the senders. As network delay on reverse path prevent a sender from being updated on time about the path information, the probability distribution of delay is considered when estimating the distortion. The delay of path information changes the arrival time media data, the probability of decoding those data before playback time, and therefore, the distortion.

Running the rate allocation and packet assignment algorithms at the senders increases the burden of the senders. Furthermore this framework duplicates the same computation at different senders and limits the senders' capability to serve many simultaneous streams.

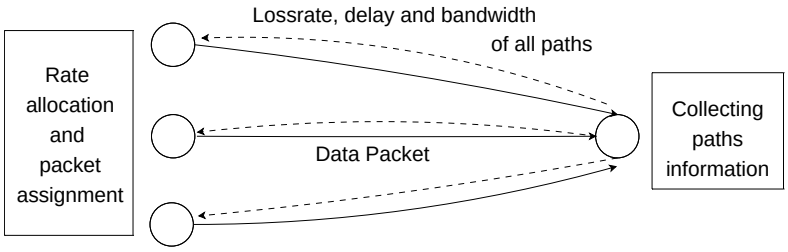


Figure 2.5: A "sender-driven" distributed media streaming system

2.3.4 Variation in Quality

Besides minimizing distortion, quality smoothness during playback (i.e., variation of distortion) is another desired property. A user may rather prefer a slightly coarser but smoother video. Nguyen and Cheung [67] explore flow control in distributed media streaming to produce a smoothed combined throughput by using multiple TCP connections, with reduced maximum window size. As the number of TCP flows increases, the maximum window of each TCP flow decreases. In case of a packet loss when a window should be halved, the window reduction becomes smaller due to the smaller maximum window (Figure 2.6(b)). When all connections suffer from packet losses at the same time, the combined window size reduces as much as one TCP connection. But the combined window size recovers much faster than a single TCP connection (Figure 2.6(c)), as the combined window increasing slope is proportional to the number of connections.

Although using multiple TCP flows compensates for smaller maximum window, it, however, makes the application unfair to other single-flow applications sharing the same bottleneck, and encourages abuse of multiple connection. Although the authors view adjusting the number of TCP connections as a mean to prioritize the application, the prioritization must be limited strictly to the applications owned by the same user. We elaborate on our view and solutions in Chapter 4 of this thesis.

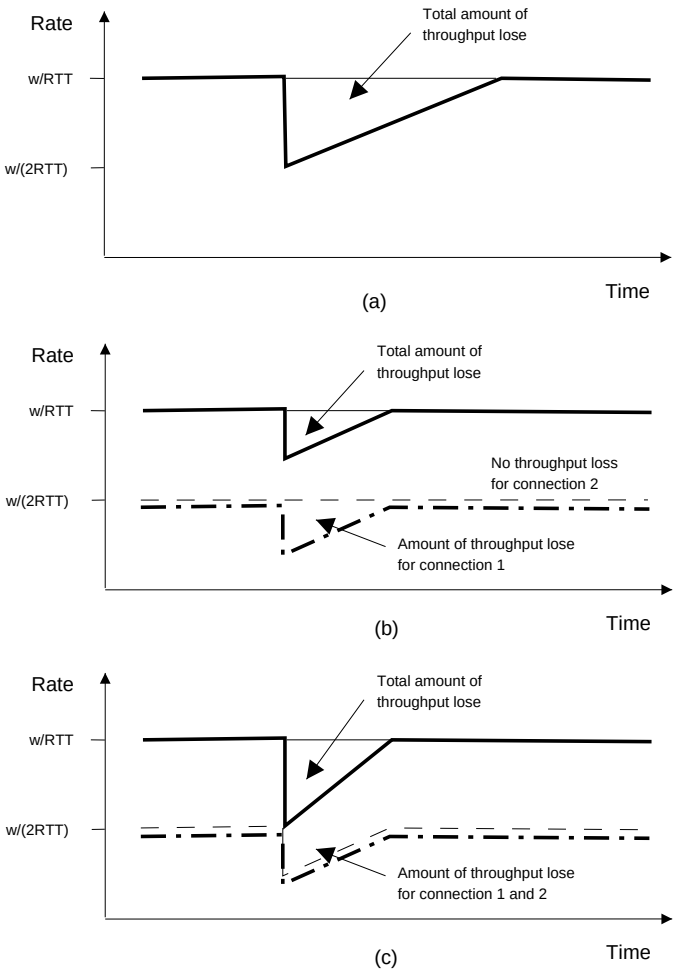


Figure 2.6: Illustration of throughput reduction for (a) one TCP connection with single loss; (b) two TCP connections with single loss; (c) two TCP connections with double losses (Figure excerpted from [67])

2.3.5 Shortest Buffering Delay

Xu et al. published their first study in distributed media streaming in 2002 (Xu et. al [97]) and studied data assignment among senders from a different perspective.

The data assignment algorithm differs from Nguyen and Zahkor (2002a) [68], even though they both target fixed size and sequentially ordered packet data, and both of them interleave packets among the senders. Nguyen's algorithm maximizes the buffering time of packets given certain playback time, whereas Xu et

al. minimize the buffering time before playback and maintain the continuity of playback at the same time. Xu et al. categorize a sender as *class-n* if its sending rate is $1/2^n$ of the playback rate. Playback can start once all packets are scheduled to arrive before their playback deadline in the future. Figure 2.7 shows an example where eight packets are assigned to four senders (one class-1, one class-2 and two class-3). Different assignment leads to different buffering delay: the first assignment needs $5t$ (equals to time to playback five packets) before starting playback, whereas the second assignments only needs $4t$. Given the sending rate of the senders, their algorithm computes a packet assignment such that the buffering time is minimized. A drawback is that they classify senders into discrete classes, which lead to under-utilization of peers' bandwidth. But this under-utilization can be rectified by defining multiple virtual peers in different classes on one physical peer, under the condition that the combined sending rate of these virtual peers equals to the sending rate of the physical peer.

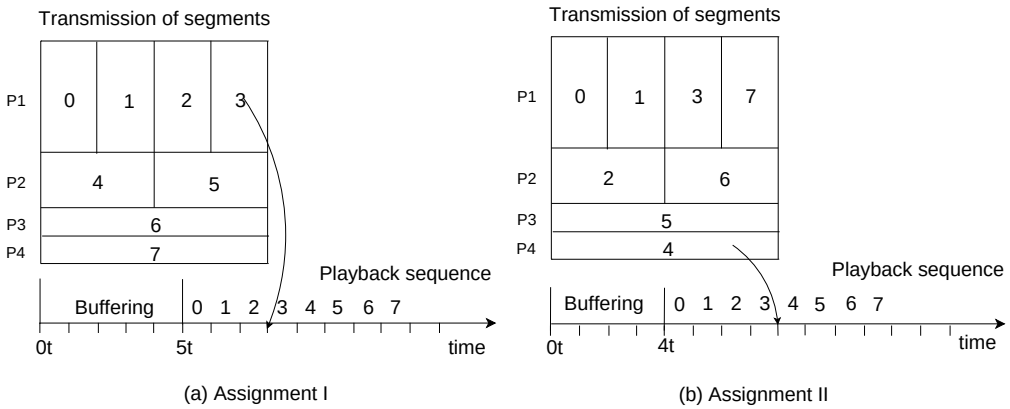


Figure 2.7: Difference media data assignments lead to different buffering delay (excerpted from [97])

2.3.6 Reducing Server Load

Minimizing distortion and buffering delay have direct impact on video quality at the receiver. Their scope, however, is limited to one session. For a distributed media streaming system, adopting a hybrid architecture (section 2.1.3), an important goal is to reduce server load, so that it is ready to serve requests when peer resources are not available. Three existing work [19, 20, 37] discuss this problem.

Cui and Nahrstedt [19] notice that when a peer-to-peer streaming system uses layered media, peer can provides only a limited number of layers. These limited layers further limit the layer availability of the downstream nodes. Although they study the problem in the context of multicast communication model, the model is similar to distributed media streaming in the following aspects: (i) data are cached in peers to serve other peers; (ii) multiple peers serve one peer simultaneous.

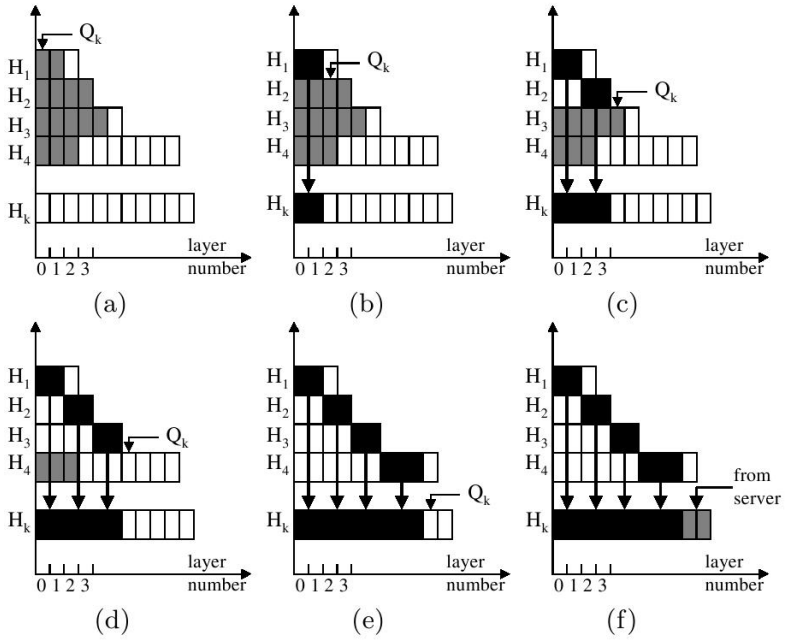


Figure 2.8: Steps (a)–(f) of deciding layers for each peer (Figure excerpted from Cui and Nahrstedt [19])

The authors characterize a sender by the video layers it stored and the available

bandwidth, and a receiver by its receiving rate. Assuming peers only cache layer incrementally, higher layers are always scarcer than lower layers in the system. When a peer caching higher layers transmits, less bandwidth remains for future use, diminishing the accessibility of the higher layers. Therefore, the authors propose that senders storing few layers (hence lower) should always be utilized before senders with more layers. Figure 2.8 shows the steps of deciding the layers that each peer should send. H_1 to H_4 are four sending peers. The squares besides stands for the layers that are available at those peers. The shadowed square corresponds to how many layers a peer's sending rate can stream. The black square shows the transmitted layers. Given a set of senders, the sender with least layers are picked to send as much layers as allowed by its sending rate; the rest request layers are picked recursively from the rest senders in the same manner.

Hsu and Hefeeda [37] applied a similar algorithm to FGS video and reduced the server load significantly. We believe, however, both studies over simplify the way layers are stored. Given limitation on local disk space and the shortage of high level layers, the peers have no reason to store layers in an incremental order, as assumed by both papers. How to optimize the availability of each layers in that case, remains an open problem.

Dana et. al [20] construct a hybrid distributed media streaming system over a BitTorrent [2] overlay network, where peers contribute upload bandwidth to reduce the burden of the centralized server. When a media content is requested, peers that can supply the requested content are looked up by using BitTorrent tracker. Data that has not passed the playback deadline is downloaded from those peers. At the deadline, the receiver downloads missing data from the centralized server. The authors found significant reduction in bandwidth usage at the servers when number of concurrent users increases. Although alleviated, the bottleneck still exists at the

[2]http://www.bittorrent.com/

server. Earlier chunks have early deadline and shorter time to be downloaded from the peers, and are mainly downloaded from the server. Therefore, the server may still suffer from scalability problem when dealing with flash crowds.

2.3.7 Service Capacity Amplification

When there are few senders in the overlay network for distributed media streaming, the system is not able to support too many clients. In this scenario, capacities amplification is a big concern. Requests should be served selectively, and priority should be given to those with higher capacity to serve others in the future. Xu et. al [97] propose that when facing a flash crowd of requesters with different uploading bandwidth, priority is given to those with higher uploading bandwidth, because they can amplify the system capacity faster. But when the load of requests is light, prioritization is renounced, so that requesters with lower uploading bandwidth are still served.

2.4 A Map of Research

2.4.1 Meddour's Overview

Meddour et al. published a survey [64] on P2P multimedia streaming, which covers an overview for distributed media streaming. They categorized several factors to be considered in this problem domain: (F1) appropriate video coding scheme, (F2) managing peer dynamicity, (F3) peer heterogeneity, (F4) efficient overlay construction, (F5) peer selection, (F6) measuring network condition, and (F7) incentive for peer participation. We believe (F3) and (F6) are less relevant to distributed media streaming, since they deal with the efficiency of the overlay. (F2), (F3) and (F5) are directly related to the problem of sender selection [35]. (F7) is

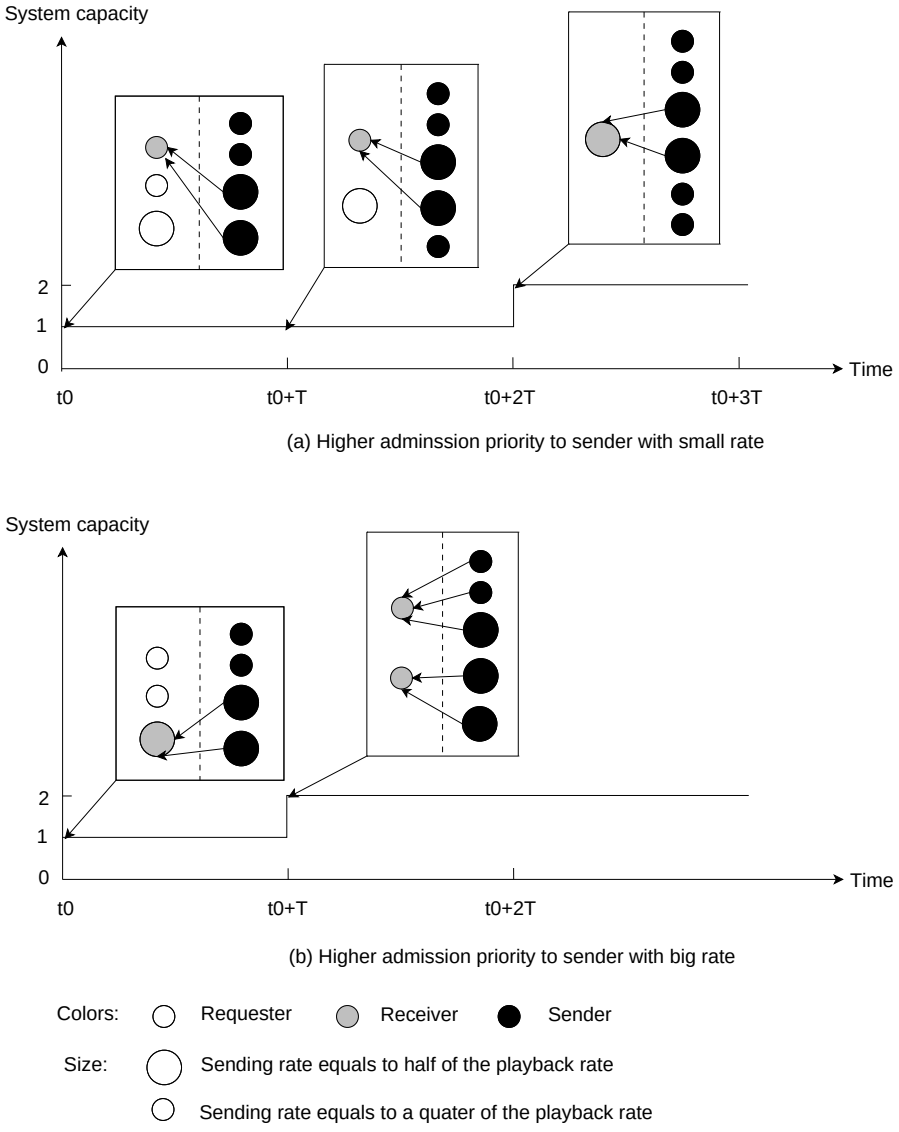


Figure 2.9: Different admission decisions lead to different growth of streaming capacity: (a) cannot serve two session simultaneously until after two session of streaming, whereas (b) can do it right after one session of streaming (excerpted from Xu et. al [97])

indirectly related to sender selection [19, 97]. Lowering the load on senders with larger contribution (on disk space or bandwidth) encourages peers to contribute more. (F1), (F2) and (F3) are important factors in rate allocation and packet assignment [68, 69, 81]. Allocating higher sending rate to senders experiencing

better network conditions would lead to better playback quality. Packet assignment must consider dependency among packets, which in turn, is related to the coding scheme. Table 2.1 shows a summary of the above points.

	Sender Selection	Rate Allocation	Packet Assignment
(i) Coding Scheme		[68]	[69, 81]
(ii) Peer Dynamicity	[48]	[48]	[48]
(iii) Peer Heterogeneity	[35]	[81]	[81]
(vii) Incentive	[19, 97]	[19]	

Table 2.1: Problems in distributed media streaming and the related factors

2.4.2 Our Map of Distributed Media Streaming

A distributed media streaming system aims to deliver high quality media content to large number of receivers on demand. To achieve this goal, most of the studies derive distortion models and minimize the distortion by properly allocate sending rate and schedule packets to the senders [48, 68, 81, 97]. Some achieve high quality by smoothing the sending rate [67]. Others concern service availability, and propose to reduce server load [19] or to increase system capacity faster [97]. As pointed out by Wu et al. [96], however, video streaming over the Internet has many other fundamental issues to cope with, such as error recovery and congestion control. While error recovery is partially studied in the context of FEC [69], [48], we believe that exploring retransmission – another important error recovery technique – is important. We also believe that congestion control is important as it can affect the practicality of distributed media streaming [26].

We summarize the current studies in the field by Figure 2.10. For each paper, the technique, the optimized metrics, and the design goal are connected together by lines of the same color. We also mark out the distinct positions of the study in this

thesis in the figure, by dotted red lines. We are the first to study retransmission and congestion control in distributed media streaming. We present our work in detail in the rest of this thesis.

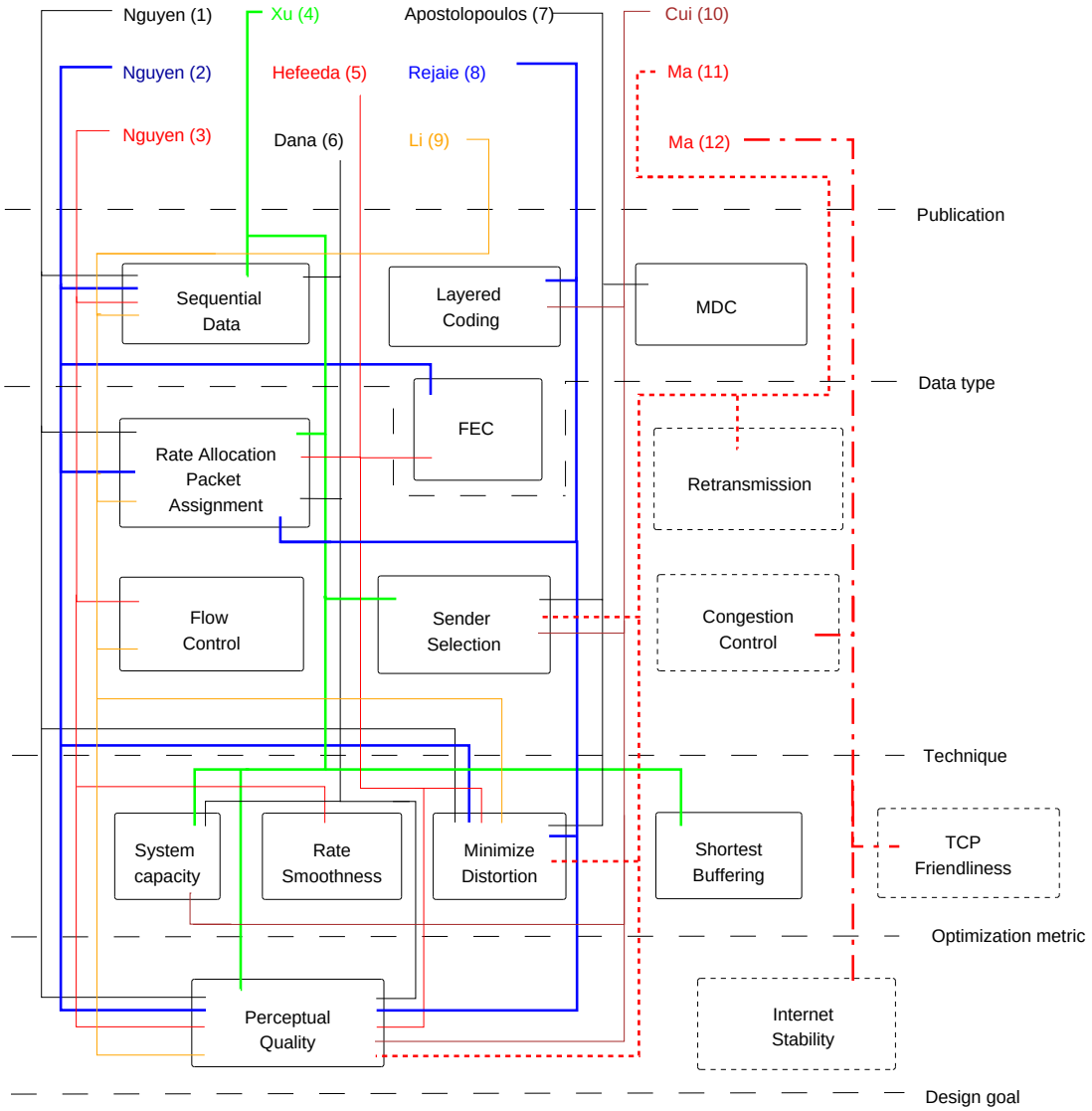


Figure 2.10: A map of current research in distributed media streaming, (1) Nguyen and Zahkor (2002a) [68], (2) Nguyen and Zahkor (2002b) [69], (3) Nguyen et. al [67], (4) Xu et. al [97], (5) Hefeeda et. al [35], (6) Dana et. al [20], (7) Apostolopoulos et. al [3], (8) Rejaie and Ortega [81], (9) Li et. al [48], (10) Cui and Nahrstedt [19], (11) Ma and Ooi [55], (12) Ma and Ooi [56]

Chapter 3

Retransmission in Distributed Media Streaming

Error recovery reduces packet loss, decreases distortion, and improves media quality at the receiver side. Conventional methods include FEC and retransmission. While FEC has been studied in distributed media streaming by Nguyen and Zahkor [69], retransmission, however, has not received much attention. In distributed media streaming of prerecorded video, buffering is acceptable at the receiver, giving opportunity to retransmit. This chapter applies retransmission to distributed media streaming.

3.1 Introduction

Distributed media streaming model differs from traditional models in that multiple channels are involved in one streaming session. The existence of multiple channels provides the option of retransmitting from a channel other than the one which lost the packet. Whether this new option would provide a better recovery rate remains unknown. If it would, the design of such a retransmission scheme would be interesting.

For ease of reference, we term schemes that retransmit from channels other than the original one as *distributed retransmission* and the traditional scheme as *non-distributed retransmission*. Our approach in this study is first to reveal whether distributed retransmission outperforms non-distributed retransmission in general, and then design a practical distributed retransmission scheme to further improve the recovery rate.

The scope of this study is limited to non-interactive streaming of pre-recorded media, where maintaining low end-to-end latency is not crucial. We also assume that the client buffers sufficient data to allow retransmission. The packets size are fixed and the bit rates of the senders are constant.

The main contributions of this study are as follows. To compare distributed and non-distributed retransmission schemes, two naive distributed retransmission schemes are presented and compared with non-distributed retransmission. After that we present a dynamic scheme for distributed retransmission, which changes retransmitter dynamically according to traced packet loss rate on different paths. By comparing the different distributed retransmission schemes and the non-distributed retransmission scheme under simulation or experiment over the Internet, we shows that, in general, distributed retransmission offers a higher loss recovery rate, and dynamic distributed retransmission further improves loss recovery rate.

The rest of the chapter is structured as follows. Section 3.2 presents related works on distributed media streaming and error recovery. Section 3.3 presents two naive design of distributed retransmission and compares them with non-distributed retransmission via mathematical analysis, intranet emulation, and real Internet experiments. Section 3.4 describes a dynamic distributed retransmission scheme and presents a comparison to other schemes via simulation and Internet experiments. Section 3.5 concludes this chapter.

3.2 Related Work

Few existing work on distributed media streaming has incorporated error recovery schemes. Nguyen et. al. [69] and Golubchik et. al. [31] make use of Forward Error Correction (FEC) in their streaming protocol. But no further analysis on effectiveness of recovery is presented. Rejaie et. al. [81] implicitly apply non-distributed retransmission in distributed media streaming for layered media, yet the effectiveness of retransmission is not explored.

On the other hand, the issue of retransmission has been extensively studied in single sender media streaming.

Papadopoulos and Parulkar's work [74] is one of the earliest that applies selective retransmission in continuous media streaming. As long as the round trip time is smaller than the time before the lost packet is to be played out, retransmission reduces loss drastically. Their evaluation reveals that retransmission copes well with burst loss. Our work is a natural extension of this work to distributed media streaming.

Perkins et. al. discuss sender-based recovery in multicast [75]. FEC and ARQ are both discussed. The authors suggest that ARQ works well in a low lossy environment and FEC performs better in non-interactive streaming with less overhead. While the survey points out the high overhead for ARQ, we must note that it is mainly due to the nature of multicast – whenever a packet is retransmitted, it is re-sent to the whole group. This overhead does not apply in distributed media streaming.

Loguinov and Radha [53] study retransmission time-out (RTO) estimation in NACK-based real time multimedia streaming. They show the inherent trade-off between the number of duplicated packets and the unnecessary waiting for timeout. They propose higher frequency RTT measurement in NACK-based protocol is

needed, in order to have an accurate RTO estimation.

Piecuch et. al. [76] designed a Selective Retransmission Protocol for multimedia streaming, based on the observation that multimedia data allows certain percentage of packet loss while request for stringent arrival time. Their protocol provides a compromise between unbounded delay of TCP and zero loss recover capability of UDP. Selective retransmission selects particular packets to retransmit, while our work selects particular channel to retransmit.

Several recent works have studied the use of TCP protocol for streaming media (e.g. see [42]). Although our work recovers lost data using retransmission just as TCP, we focus on selection of senders for retransmission, which can not be done in TCP.

3.3 Distributed versus Non-Distributed Retransmission

Traditional single channel streaming does not provide different channel to retransmit lost packets, therefore only non-distributed retransmission are allowed. Due to the existence of multiple channels in distributed media streaming, distributed retransmission becomes an option. Yet the performance of distributed retransmission remains unknown. Since packet loss is generally a consequence of network congestion, the following packets are likely to experience the same congestion and suffer from a high loss rate. In this case, distributed retransmission, which retransmits a lost packet from a different channel, avoids the congestion, and therefore should offer a better recovery rate.

In this section, we present two naive distributed retransmission schemes, and study their recovery rate by modeling, analysis, simulation, and experiments on the Internet.

3.3.1 Two Naive Distributed Retransmission Schemes

In order to compare the recovery effectiveness of distributed retransmission with non-distributed retransmission, we designed two naive distributed retransmission schemes called ARQ-D and ARQ-RR. ARQ-D is a scheme with a dedicated retransmitter among the senders. A sender with the lowest loss rate is manually selected as a retransmitter. While other senders send data packets, it retransmits if their packets are lost. ARQ-RR is a scheme that rotates retransmission task among the senders in a round robin manner. For comparison, we call the non-distributed retransmission scheme ARQ-O, in which the receiver asks the original sender for retransmission.

Given the same channel conditions, ARQ-D uses the best channel to retransmit, therefore provides the best recovery rate a distributed retransmission scheme could have if it does not switch retransmitter during the session. In ARQ-RR, on the other hand, the retransmission is carried out randomly on different channels without any selection, therefore the scheme corresponds to the average performance of distributed retransmission. The effectiveness of distributed retransmission is better than non-distributed retransmission.

3.3.2 Model and Assumptions

In this section, we present our model and assumptions that help to simplify the analysis.

To be fair to all three schemes, only one retransmission is performed for every loss. It simplifies the analysis yet is sufficient to show their effectiveness in loss recovery: the same bandwidth is used for each packet recovery, but different effective loss rate can be achieved.

The time between sending a lost packet and re-sending its ARQ packet is δ.

Value of δ is decided by round trip time (RTT) of the channels. In order to be fair, we assume that all senders have the same RTT in analysis.

To calculate recovery rate, we do not assume any special coding scheme such as layered coding or MDC.

The sequence of data packets are sent by the senders in a round robin manner to reduce consecutive packet loss when one channel is congested.

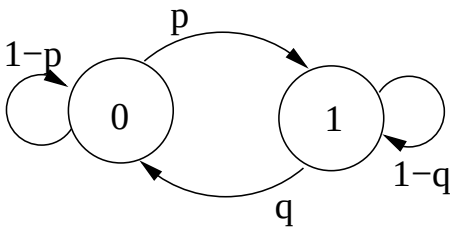


Figure 3.1: Gilbert Model

Most of the time senders send packets at a constant rate: one packets for one time unit. This assumption allows us to use a Gilbert model[1] to model the state of channels, with regard to the changing of channel states in discrete time (Figure 3.1). For senders sending both data packets and ARQ packets, however, we assume the ARQ packets do not delay data packets. When retransmission happens, bit rate is increased to send both data packet and ARQ packet in the same time unit. This simplification removes the cumulative delay of data packets imposed by retransmission.

We also assume that the retransmission does not deteriorate channel quality; video streaming requires certain network quality: if retransmission affects the channel quality, the channel is generally not suitable for video streaming.

For ease of analysis, we also assume independence of channels among the senders. As we will see in Section 3.4, this simplification is not valid in real world.

[1] p is the probability of channel state transforming from good to bad, q is the one vice versa. For more on Gilbert model, refer to the paper by Bolot et. al. [7]

For analytical purpose, however, it is necessary. This assumption is also embraced by other works [68, 81].

3.3.3 Mathematical Analysis

We now model these different schemes analytically and analyze their effective loss rate and expected burst length.

Let n be the number of senders. Treating each sender as an independent channel, we denote the set of senders as $c_1, c_2, ..., c_n$. Each channel c_i is modeled using a Gilbert model with parameter p_i and q_i, where p_i is the probability of transition from good state (denoted as 0) to bad state (denoted as 1) and q_i is the probability of transition from bad state to good state (see Figure 3.1).

We introduce some additional notations as follows. Let L_i be the average packet loss rate of channel c_i. L_i can be estimated from the Gilbert model and is given as $p_i/(p_i + q_i)$.

Given the Gilbert model, we can also estimate $P_i(\delta)$, which is the probability of transition from a good state to bad state for channel c_i after δ time units (or, equivalently, after sending δ packets). Similarly, we can compute $Q_i(\delta)$ as the probability that a channel c_i goes from a bad state to a good state after δ time units. $P_i(\delta)$ and $Q_i(\delta)$ are analogous to p_i and q_i in Gilbert model in terms of good-to-bad and bad-to-good state transitions. These two probabilities are:

$$P_i(\delta) = L_i - L_i(1 - p_i - q_i)^\delta \tag{3.1}$$

$$Q_i(\delta) = 1 - L_i - (1 - L_i)(1 - p_i - q_i)^\delta \tag{3.2}$$

We denote the value $1 - P_i(\delta)$ as $\bar{P}_i(\delta)$. This value corresponds to the probability that the state is good after δ time units, given that the current channel state is good. Similarly, we use $\bar{Q}_i(\delta)$ to denote the probability of channel transiting from bad state to bad state after δ time units, i.e., $\bar{Q}_i(\delta) = 1 - Q_i(\delta)$.

Effective Loss Rate

Using L_i and $\bar{Q}_i(\delta)$, we can now compute the effective loss rate of distributed streaming under different error recovery schemes. The effective loss rate, or *unusable rate* for short, is the probability that a packet is lost and cannot be recovered. Unusable rate reveals the average quality degradation of the received media.

We now consider the unusable rate of ARQ-based schemes. A packet is unusable if the packet is lost, and the retransmitted packet is lost as well. For simplicity, we only model the case for single retransmission here. The unusable rate is therefore estimated as the probability that the data packet is lost, and the retransmitted packet is lost after time δ.

ARQ-D: Without loss of generality, let c_n be the dedicated retransmitter channel and the other $n-1$ channels be data channels. The probability that a data packet is lost is given by $\sum_{i=1}^{n-1} L_i/(n-1)$ and the probability that the ARQ packet is lost is L_n. The unusable rate of ARQ-D scheme, V_{ARQ-D} is therefore given by

$$V_{ARQ-D} = \frac{L_n}{n-1}\sum_{i=1}^{n-1} L_i$$

ARQ-O: Since the data packet is sent on the same channel as retransmitted packet, the loss probability for data packet and retransmitted packet is correlated. We know that L_i is the probability that data packet is lost on channel c_i and $\bar{Q}_i(\delta)$ is the probability that a packet is lost in the same channel after time δ. Assuming that retransmission occurs after time δ, the unusable rate for that channel is therefore $L_i\bar{Q}_i(\delta)$. Averaging over n channels, we have the expected unusable rate as

$$V_{ARQ-O} = \frac{1}{n}\sum_{i=1}^{n}(L_i\bar{Q}_i(\delta))$$

ARQ-RR: Under this scheme, the retransmitted packet is sent by different senders. With probability $1/n$, it is sent by the original sender. For a channel c_i, the probability that the data packet is lost is L_i, and the probability that its

retransmitted packet is lost is $(\bar{Q}_i(\delta) + \sum_{j \neq i} L_j)/n$. The unusable rate is therefore given by

$$V_{ARQ-RR} = \frac{1}{n^2} \sum_{i=1}^{n} (L_i(\bar{Q}_i(\delta) + \sum_{j \neq i}^{n} L_j))$$

We have derived the expected unusable rate for the three ARQ schemes as a function of Gilbert model's parameters (p_i, q_i). We can now plot these functions.

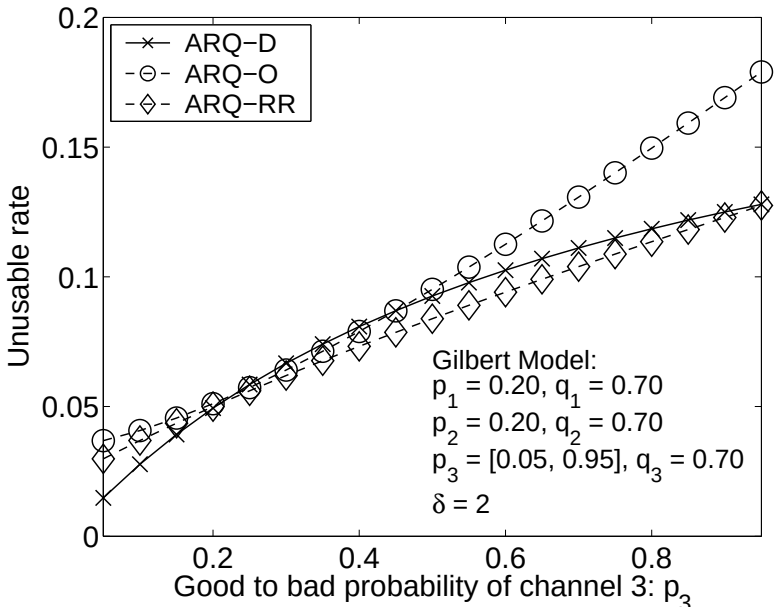


Figure 3.2: Unusable Rate vs. p_3

For simplicity, only the case where $n = 3$ is considered. We vary the condition of channel c_3 (dedicated retransmission channel in ARQ-D scheme) and plot the unusable rate in Figure 3.2 and 3.3 with $\delta = 2$. Figure 3.2 shows that when the channel condition of c_3 is better then the other two channels, the ARQ-D scheme gives lower unusable rate. This behavior is expected since the probability of successful retransmission is higher in this scenario. As the probability of good-to-bad transition increases for channel c_3, the unusable rate for ARQ-D increases and become worse than ARQ-RR. We expected this trend as well, since ARQ-RR rotates among the channels for retransmission and for two out of three retransmissions, it chooses a better quality channel than channel c_3. An important observation from

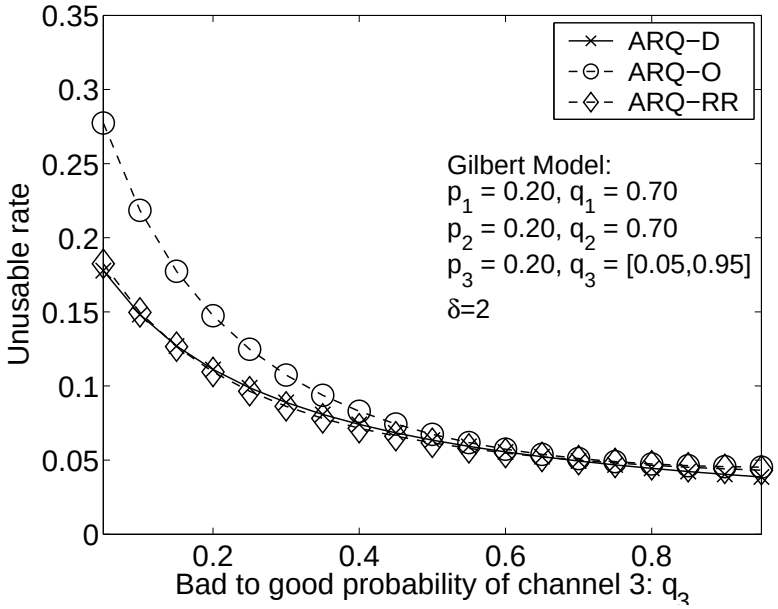


Figure 3.3: Unusable Rate vs. q_3

this figure is that, using an unrealistically bad channel under the ARQ-D or ARQ-RR schemes would still give lower unusable rate, compared to ARQ-O scheme. Figure 3.3 shows the results when we vary the bad-to-good transition probability of channel c_3. We can see that the burstiness does not differentiate the unusable rate of ARQ-RR and ARQ-D schemes much. Both of these schemes give lower unusable rate than ARQ-O schemes, especially when the channel is bursty.

Figure 3.4 and 3.5 plot the unusable rate as we vary the conditions of channel c_1. In these plots, we configure channel c_3, the dedicated retransmission channel, as a good quality channel with $p_3 = 0.1$ and $q_3 = 0.95$. These figures show that by using a good quality retransmission channel, we can achieve much lower unusable rate if we use ARQ-D scheme compared to ARQ-RR or ARQ-O schemes.

In the above figures, we set the delay between an ARQ packet and its corresponding lost to 2 ($\delta = 2$). The effects of δ on the three ARQ schemes are plotted in Figure 3.6 with the same Gilbert setting as above mentioned figures. It is obvious δ affects effectiveness of ARQ-O and ARQ-RR, but not ARQ-D.

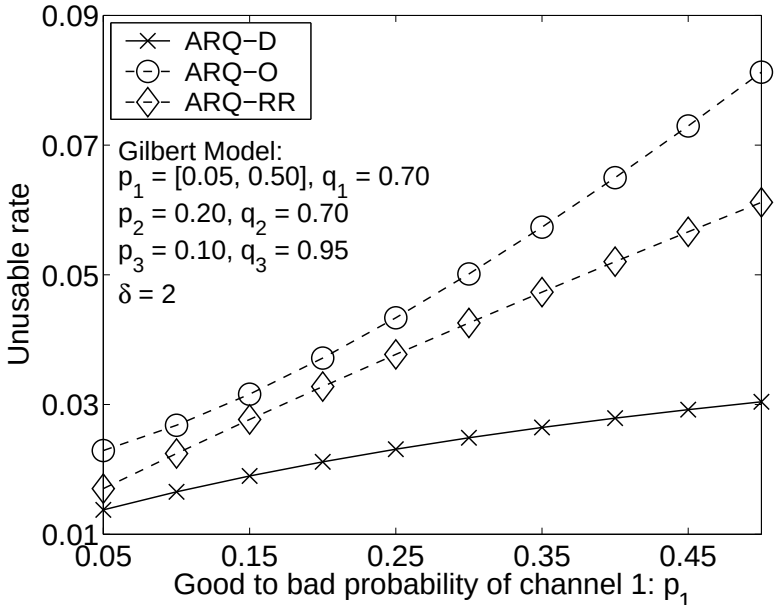


Figure 3.4: Unusable Rate vs. p_1

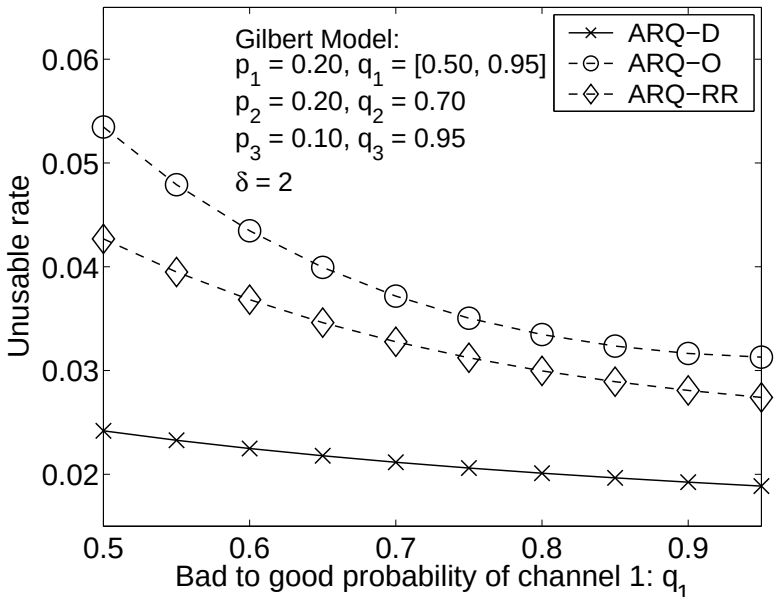


Figure 3.5: Unusable Rate vs. q_1

Expected Burst Length

To study the expected packet loss burst length, we further simplify our model to homogeneous channels. In other words, we use the same Gilbert model with parameter (p, q) to model all channels. We also restrict our model to three channels

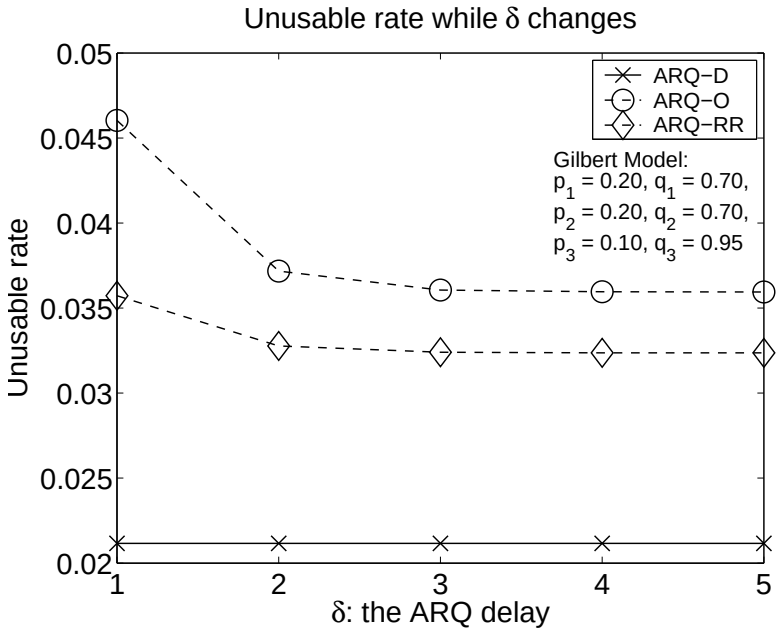


Figure 3.6: Unusable rate versus δ (Analysis)

only.

Despite these vast simplifications, the analysis for expected burst length is still quite complex. For each error recovery scheme, there are four cases to consider. A burst of packet loss, or *gap*, of length m always starts with a usable packet, followed by m consecutive unusable packets and ends with another usable packet. A usable packet is either delivered, or lost but recovered. Thus, we have to consider the cases where the gap begins and ends with both delivered packets (Case 1), begins with a lost but recovered packets and ends with a delivered packet (Case 2), begins with a delivered packet and ends with a lost but recovered packet (Case 3) and begins and end with both lost but recovered packets (Case 4). We will analyze these four cases separately, and use $\alpha_i(m)$ to denote the probability that the burst length is m for Case i. The probability of burst length m occurring is thus $\sum_{i=1}^{4} \alpha_i(m)$.

While deriving the probability of occurrence of a gap of length m, we will only explain in details Case 1 in ARQ-D scheme, and list the equations of gap length for ARQ-O and ARQ-RR schemes without further explanations, as the derivation

is similar.

ARQ-D: For Case 1, with $m = 1$,

$$\alpha_1(1) = L^2(1-L)(1-p) \tag{3.3}$$

.

$\alpha_1(1)$ is given as the probability that packet in channel c_1 is delivered, $1-L$, and a packet in channel c_2 is lost and not recovered, L^2, and the next packet in channel c_1 is delivered, $1-p$. This argument can be generalized for value of m larger than 1, giving

$$\alpha_1(m) = L^3p^2(1-q)^{2m-3},\ m \geq 2. \tag{3.4}$$

The probability for the other cases are given as:

$$\alpha_2(m) = \alpha_3(m) = L^3q^2(1-q)^{2m-2} \tag{3.5}$$

$$\alpha_4(m) = L^3q^2(1-q)^{2m-1}. \tag{3.6}$$

ARQ-O: Similar to the analysis of ARQ-D, we compute $\alpha_i(m)$ for all four cases.

$$\alpha_1(m) = \begin{cases} (1-L)^2L\bar{Q}(\delta), & m = 1, \\ (1-L)L^2\bar{Q}(\delta)^2(1-p), & m = 2, \\ L^3\bar{Q}(\delta)^mq^2(1-q)^{m-3}, & m \geq 3. \end{cases} \tag{3.7}$$

$$\alpha_2(m) = \alpha_3(m) \tag{3.8}$$

$$= \begin{cases} (1-L)L^2\bar{Q}(\delta)Q(\delta), & m = 1, \\ L^3\bar{Q}(\delta)^mQ(\delta)q(1-q)^{m-2}, & m \geq 2. \end{cases} \tag{3.9}$$

$$\alpha_4(m) = L^3\bar{Q}(\delta)^mQ(\delta)^2(1-q)^{m-1} \tag{3.10}$$

ARQ-RR: For ARQ-RR scheme, since the retransmitted packet is sent by the senders, in a round robin manner, we compute the loss rate of the retransmitted

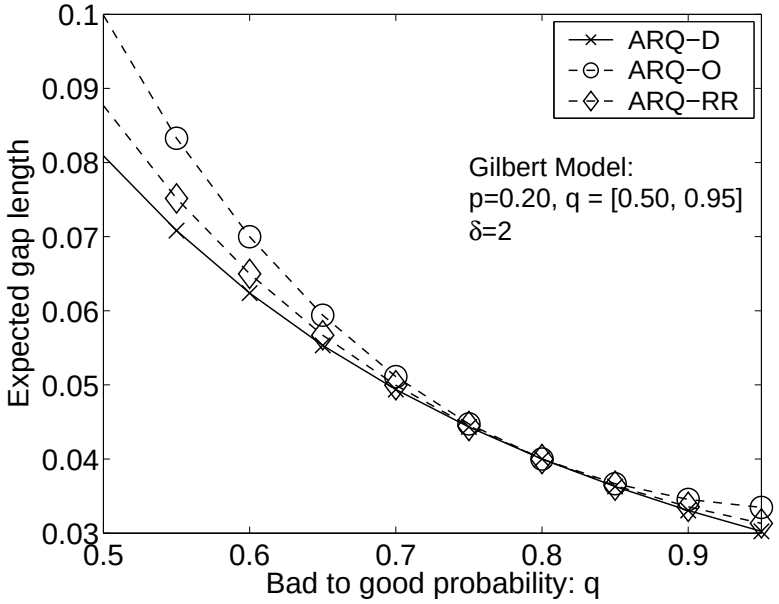


Figure 3.7: Expected Gap Length vs. q

packet first. We denote this loss rate as L'.

$$L' = \frac{1}{3}(2L + \bar{Q}(\delta)) \tag{3.11}$$

Using derivation similar to previous schemes, we have

$$\alpha_1(m) = \begin{cases} (1-L)^2 LL', & m = 1, \\ (1-L)L^2 L'^2(1-p), & m = 2, \\ (1-L)L^2 L'^m pq(1-q)^{m-3}, & m \geq 3. \end{cases} \tag{3.12}$$

$$\alpha_2(m) = \alpha_3(m) \tag{3.13}$$

$$= \begin{cases} (1-L)L^2 L'(1-L'), & m = 1, \\ (1-L)L^2 L'^m(1-L')p(1-q)^{m-2}, & m \geq 2. \end{cases} \tag{3.14}$$

$$\alpha_4(m) = L^3 L'^m (1-L')^2 (1-q)^{m-1} \tag{3.15}$$

The probability of different gap length is plotted in Figure 3.7 using the derived expressions with varying bad-to-good transition probability q and $\delta = 2$. We can see that ARQ-D scheme gives shortest expected gap length. We omit the curve that

shows the effect of Gilbert parameter p on expected gap length as the differences among the schemes are too small to be interesting.

3.3.4 Experimental Evaluation

To evaluate the ARQ schemes and verify our analysis, we implemented an RTP-based distributed streaming system for MP3 audio based on the LIVE555.COM[2] media streaming library using the three proposed ARQ schemes for retransmission. We conducted experiments over PlanetLab for realistic network settings, and over our Intranet under controlled network environment.

For each experiment, the system streams a 31.8 second MP3 audio file, consisting of 1224 application data unit (ADU),packetized based on RFC3119 **??** using three senders. Each ADU is approximately 0.4KB, with one packet consists of 2 to 5 ADUs. ADUs are interleaved among the senders so that a lost packet from one sender will not caused consecutive ADUs to be lost. In our experiments, we measure unusable rate of ADUs and burst length of ADUs, as these metrics are more meaningful than unusable rate and burst length of packets.

Experiments over Intranet

We first present our results based on experiments over Intranet using simulated packet loss. Our goal is to further strengthen our observations since the analytical results obtained in previous section is based on simplifying assumptions such as homogeneous channels and fixed δ.

Using the same Gilbert model parameters as in Section 3.3.3, we first verified our analytical results. Collected over 20 runs, our simulation results give very similar curves. One such set of curves, which corresponds to Figure 3.4 is shown in Figure 3.8.

[2]http://www.live555.com/liveMedia

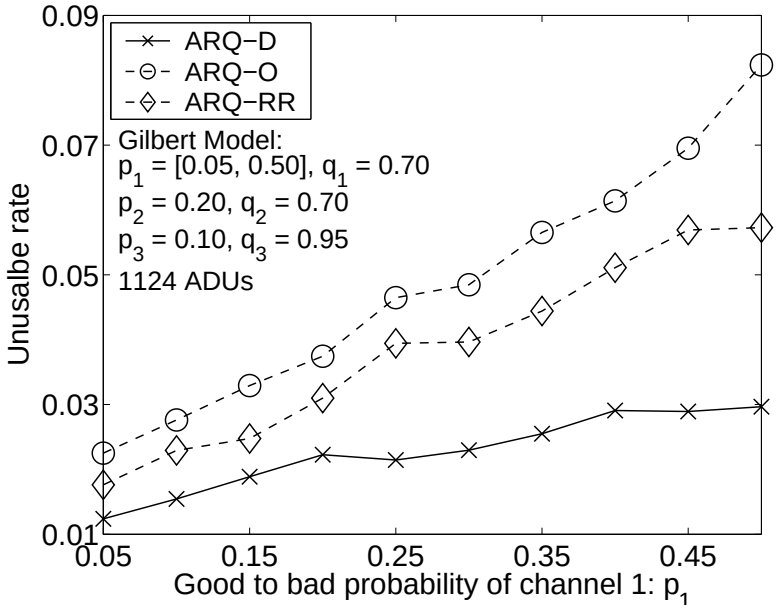


Figure 3.8: Unusable rate vs. p_1 (Simulations)

Next, we study the effect of heterogeneous channels on burst length. We focus mainly on results for bursty loss with length larger than one, as we find that the results for gap length of one follows closely the behavior of the curves for unusable rate (e.g., see Figure 3.9).

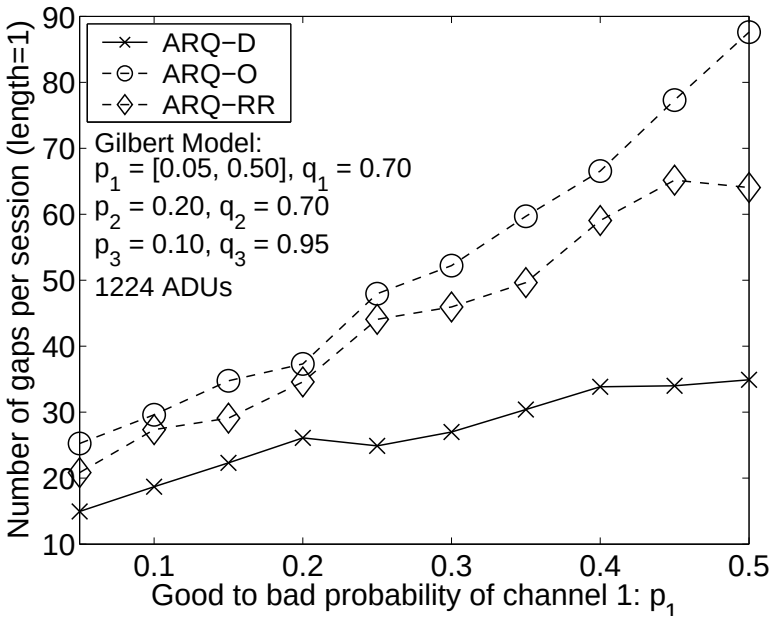


Figure 3.9: Effect of p_1 on Number of Gaps with Length 1

Figure 3.10 and 3.11 show the number of gaps in ADU with gap length larger than one. They indicate that ARQ-D has fewer bursty losses compared to ARQ-RR and ARQ-O as we vary the condition of channel c_1.

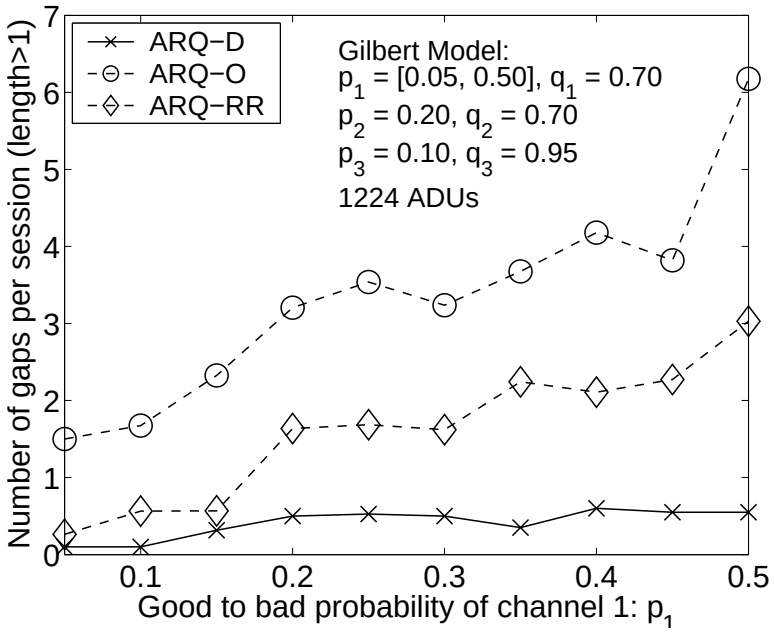


Figure 3.10: Effect of p_1 on Number of Gaps with Length > 1

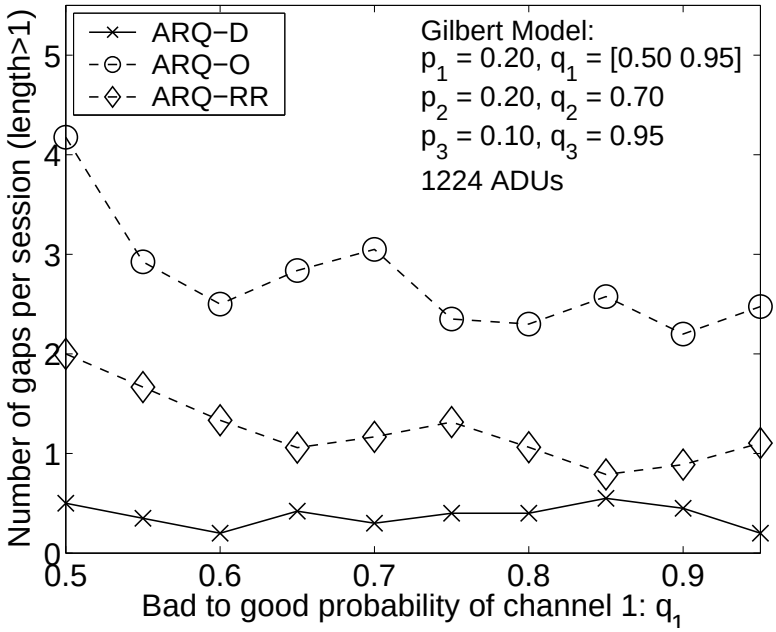


Figure 3.11: Effect of q_1 on Number of Gaps with Length > 1

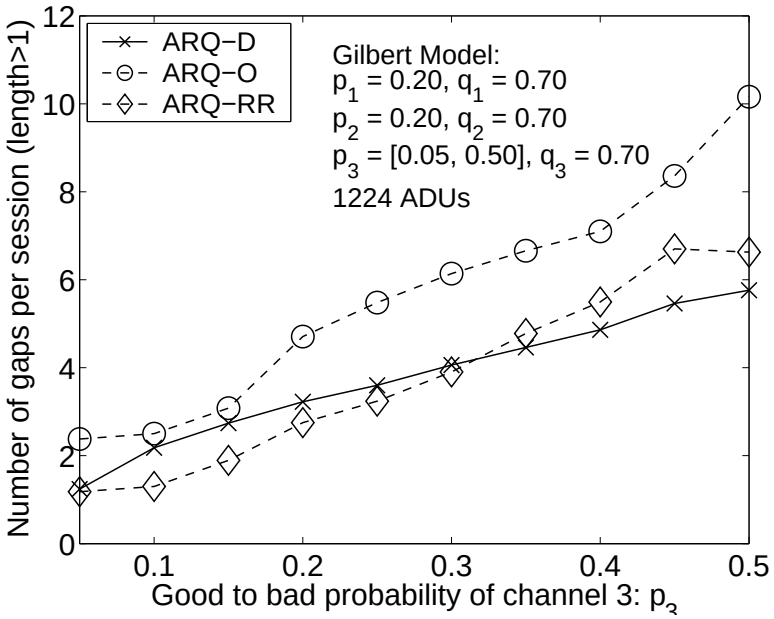


Figure 3.12: Effect of p_3 on Number of Gaps with Length > 1

A more interesting observation can be found in Figure 3.12 and 3.13, which vary the condition of channel c_3, the dedicated retransmission channel. We can see in Figure 3.12 that even when channel c_3 is less lossy, using ARQ-D scheme leads to slightly more lengthy ADU gaps than ARQ-RR. The cause of this behavior is that, in our model, ARQ-D uses only two channels for data transmission while ARQ-RR uses all three. Thus, the probability of getting two consecutive losses is higher for ARQ-D. Figure 3.13 shows that when channel c_3 is bursty, ARQ-D can result in most number of ADU gaps. Again, this result can be explained by the fact that ARQ-D uses only two channels for data transmission. When the retransmission channel is bursty, probability of recovering from two consecutive data loss decreases. The number of gaps, however, drops rapidly as channel c_3 becomes less bursty.

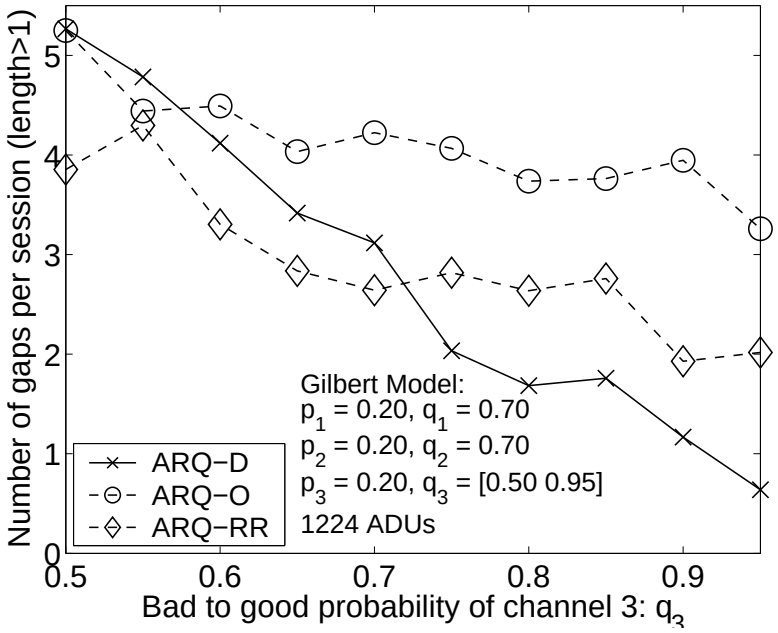


Figure 3.13: Effect of q_3 on Number of Gaps with Length > 1

Experiments over PlanetLab

Besides experiments under controlled environment within our Intranet, we conducted real experiments over PlanetLab, a wide-area test-bed for large scale distributed applications to see how the schemes performed under realistic network conditions. We use three remote senders plus one local receiver[3]. The measured loss rate of the channels are 13.34%, 11.60% and 12.34% respectively for c_1, c_2 and c_3. Due to the unpredictability of network conditions, we increase the number of runs per experiments to 50.

Figure 3.14 presents the average unusable rate of different error recovery schemes. The PlanetLab test results show that under realistic network conditions, ARQ-D has the lowest average unusable rate.

Figure 3.15 shows the average frequency of single loss and burst loss with length larger than 1, per session. The results from our PlanetLab experiments

[3]planetlab2.ie.cuhk.edu.hk (c_1), planetlab2.cis.upenn.edu (c_2), planet1.cc.gt.atl.ga.us (c_3) and soccf-planet-002.comp.nus.edu.sg.

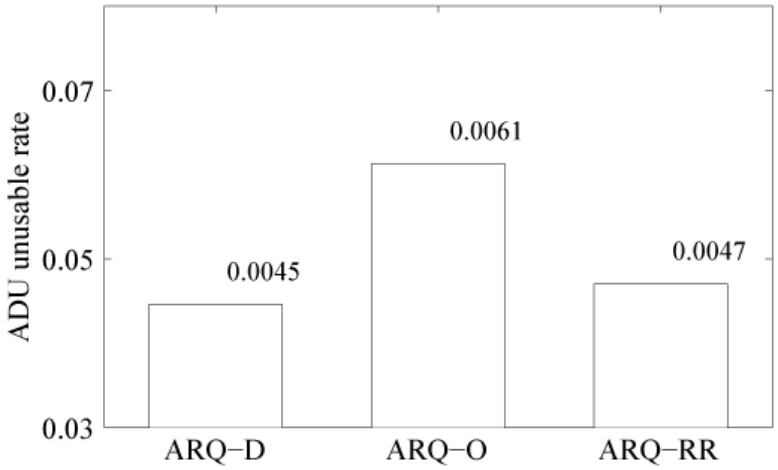


Figure 3.14: Unusable rate (PlanetLab)

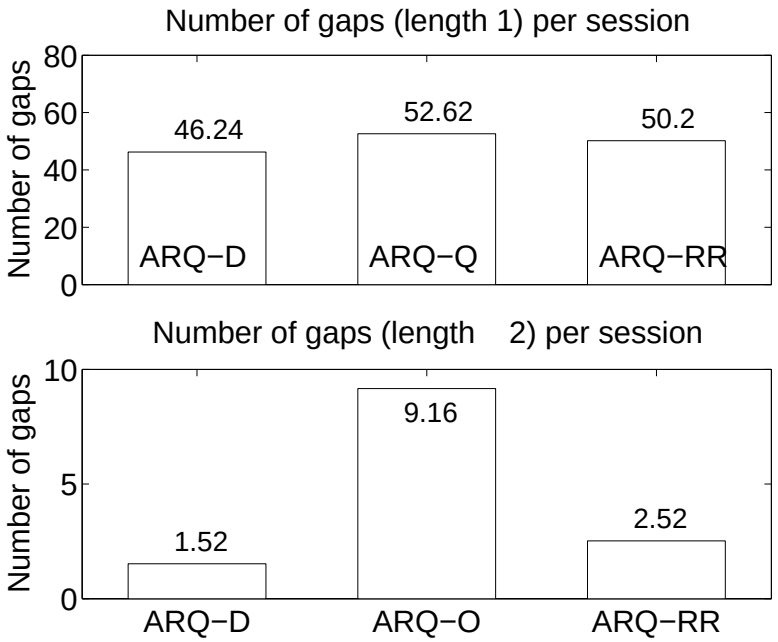


Figure 3.15: Gaps per session (PlanetLab)

indicate that ARQ-O can result in long gaps, while ARQ-D achieves least number of gaps. We also observe that the performance of ARQ-RR does not differ much from ARQ-D. This observation suggests that in the case where channel conditions are unknown, ARQ-RR could be a good retransmission scheme. By requesting a different sender for retransmission each time, the receiver experiences average channel conditions in the long run.

Both analysis and the Internet experiment show that distributed retransmission outperforms non-distributed retransmission. The two distributed retransmission schemes, however, does not consider changing network condition while selecting the retransmitter. To overcome this drawback, we design a dynamic retransmission scheme that can further improve the recovery rate.

3.4 A Dynamic Distributed Retransmission Scheme

In the previous section, we show the superiority of distributed retransmission over non-distributed retransmission scheme by demonstrating their much lower effective loss rate both analytically and experimentally. The two distributed retransmission schemes, however, are naive: (i) ARQ-D uses the channel with the lowest loss rate for retransmission. But it does not dynamically change retransmitter during the session, and the bandwidth of the dedicated retransmission channel is wasted if there are no packet loss. (ii) ARQ-RR selects retransmitter in round robin manner, i.e. the channels have equal chances of being assigned a retransmission task regardless of their loss rate. In this section we present ARQ-L, a distributed retransmission scheme that dynamically switches the retransmitter when packet loss rate changes. The new scheme is simple yet effective in avoiding retransmission on congested channels, and showing notable improvements over ARQ-RR and ARQ-D.

3.4.1 Description of ARQ-L

The idea of ARQ-L is simple: the receiver tracks the packet loss rate on each path. When a packet loss is detected, retransmission is performed by the path with the lowest packet loss rate. Unlike ARQ-D that dedicates a sender (hence a path) to retransmission, all senders in ARQ-L send data. While the lost packets are

retransmitted from the fixed retransmitter in ARQ-D, they are retransmitted from the best sender at that moment. The best sender, with the lowest probability to fail the retransmission, is selected dynamically. If ARQ-D has shown that retransmit from the path with the lowest packet loss rate provides the best recovery rate (Figure 3.8), ARQ-L provides a finer granularity of the packet loss rate over the time. ARQ-L also utilizes bandwidth better by sending data on every paths.

Packet Loss Detection

A packet loss plays two roles in ARQ-L. (i) It is used to count packet loss rate, which is the indicator of path quality. (ii) It triggers retransmission. The two functionalities, however, require different features on packet loss detection. On one hand, in order to update the loss rate of each paths, detection needs to be fast, so that the information is on time. Yet it is easy to misclassify a delayed packet as lost. On the other hand, while triggering retransmission, loss detection needs to be accurate, since unnecessary retransmission consumes extra bandwidth [53]. As a result, in ARQ-L, we adopt two methods for loss detection: a timeout scheme similar to RTO in TCP ensure the prompt update of loss rate, and a gap-based method to trigger retransmission.

Timeout-based detection sets a deadline for each packet to arrive. If the packet does not arrive on time, it is considered lost. Before calculating the deadline of the packet we assume bit rate and packet size are fixed. The assumption is reasonable in a short period of time, and is embraced by other works [68, 97]. For simplicity, in our experiments, we use constant bit rate and packet size for the whole session.

Following the assumption, the packet interval would be fixed and denoted as τ. If the first packet in the period is sent at t_1, the n^{th} packet should be sent at t_n, where $t_n = t_1 + (n-1)\tau$, t_1 equals to $t_s + RTT/2$, and t_s is the time when the

receiver sends the first request in this period. RTT is the round trip time that are sampled once for every period. In our experiment, as we do not change sending rate, it is fixed before the streaming session.

Like in TCP [86], the maximum allowed delay variation is four times the jitter. The deadline of n^{th} packet would be $t_n + RTT/2 + 4 * D.$, where D is the jitter (a.k.a. smoothed mean deviation) of the one way delay. In ARQ-L it could be measured by the jitter of inter arrival time of consecutive packets. Denoting the newly measured inter arrival time as $\hat{\tau}$, D is updated by: $D \leftarrow D + \alpha * (\hat{\tau} - \tau)$, on every received packet with a expecting sequence number. The variable α is the deviation gain and is set to 0.25 [86].

Timeout-based loss detection reports losses when packets are delayed. It is fine for congestion detection, but it should not trigger retransmission if the packet is finally received. Therefore timeout-based method is used only to update loss rate.

Gap-based loss detection is used to trigger retransmission without unnecessary retransmission request. Upon receiving a packet with a discontinuous sequence number, all packets in the sequence gap are considered lost, and retransmission request are sent to the sender with lowest loss rate. If multiple senders have the same loss rate, one of them is randomly picked for retransmission.

Gap-based detection brings extra delay before retransmission. The amount of the delay (period without receiving any packet), however, is not likely to be greater than the play-back buffering time at the receiver.

3.4.2 Simulation

We evaluate ARQ-L by comparing the unusable rate (effective loss rate) of ARQ-L, ARQ-D, and ARQ-RR. The comparison are made in the same network environment via in simulation and experiments over the Internet.

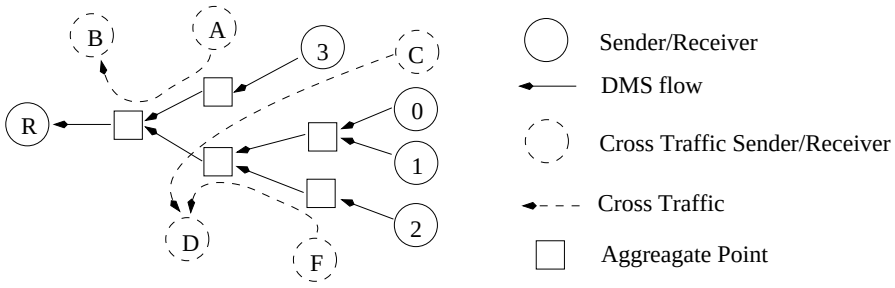


Figure 3.16: A distributed media streaming session with background traffic

On the topology of Figure 3.16, we conduct the following simulation using ns-2 to show this advantage of ARQ-L. The aggregate bit rate is 120kbps and packet size is 0.6kB. Therefore, for ARQ-RR and ARQ-L, the four senders send at 30kbps each, and for ARQ-D, the three senders send at 40kbps each.

Background CBR flows C-D, F-D, and A-B congest shared path of flow 0 and 1, path of flow 2, and path of flow 3 respectively. The arrival of background flows follow the *poisson process*, i.e., the inter-arrival time of congestion follows *exponential distribution*. The average inter-arrival time of these flows are 60, 20 and 20 seconds for C-D, F-D and A-B respectively. The average duration of the flows are 30 seconds. Because flow C-D appears less frequently than the other two, loss rate of flow 0 or 1 are smaller than loss rate of flow 2 or 3. Hence in ARQ-D, sender 0 is chosen as retransmitter; in ARQ-RR, three senders take turns to retransmit; and in ARQ-L, the sender with the lowest loss rate at the moment retransmits.

We plot the throughput of both data and ARQ packets in each paths. A decreased throughput of data packets reveals a congestion in that channel, and an increased throughput of ARQ packets shows retransmission request being sent to that sender. Studying the throughput of the two types of packets on the paths reveals how the three schemes pick retransmitters. The four sub-graphs in every single figure (Figure 3.17, 3.18 and 3.19) correspond to the four channels in the

system. Horizontal axis is for time slot, and vertical axis is for number of packets. From top to bottom, they are channel 0, 1, 2 and 3 respectively. In every sub-graph, solid line tells the number of data packets (throughput) that are received during that time slot (in length of five seconds). When a sender is congested, the solid line declines. Dash line tells the number of ARQ requests sent to that sender during the same time slot. A good scheme should avoid sending ARQ requests to a congested sender: when solid line is low, dash line should be low. Please note dash line could be higher than solid line when solid line is in its peak: when all the data packets are received (the channel is not congested), throughput of ARQ packets could be higher than data packets.

Given the sending rate and packet size in the experimental setting, the maximum packet rate on data channel of ARQ-D would be around 40 packets per slot $(120/(3 \times 0.6 \times 8) \times 5)$, and the one of ARQ-RR and ARQ-L would be 32 packets per slot $(120/(4 \times 0.6 \times 8) \times 5))$.

Figure 3.17 shows the throughput of the four channels in ARQ-RR scheme. It shows that, many ARQ requests are sent to congested senders. For example, during the period from slot 5 to slot 20 of channel 0 and 1, when the zero throughput of data packets indicates channel 0 and 1 are extremely congested, ARQ requests (shown in dash line) are still sent to sender 0 and 1. Every sub-graph has cases where solid line is in the bottom, but dash line is still high. The reason is that ARQ-RR does not select path for retransmission, hence ARQ requests are sent to each channels regardless of whether it is congested or not.

Figure 3.18 shows the throughput of the four channels in ARQ-D scheme. Channel 0 is dedicated for retransmission and does not send data packet, so it should not have throughput of data packets (solid line). However for the purpose of comparison, a solid line is copied from the throughput of channel 1, as we have shown in Figure 9 (not shown in this draft) that channel 0 and 1 have similar throughput, as

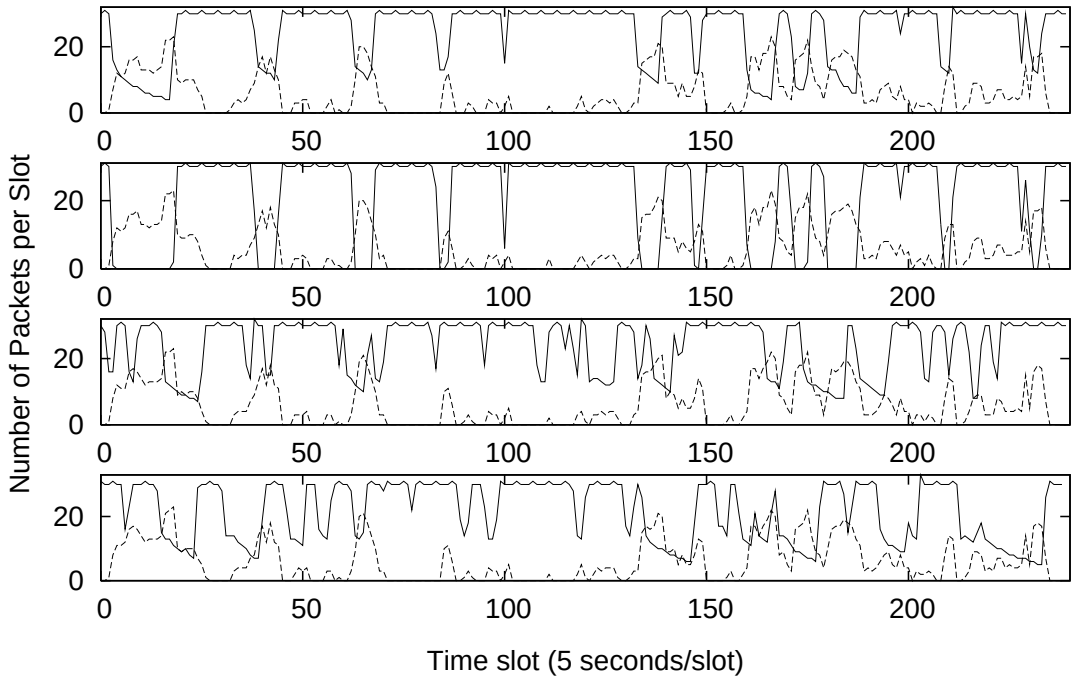


Figure 3.17: Throughput of channels in ARQ-RR

all congestion are shared by them. It is obvious that retransmission requests were sent to sender 0, even when its channel was severely congested, e.g. slot 5 to 20 in channel 0. In this case, ARQ-D scheme has a very poor recovery performance, as all ARQ packets during that period suffer from severe congestion and are very likely to be lost.

Figure 3.19 shows the throughput of the four channels in ARQ-L scheme. In this scheme ARQ requests are sent to the channel that are least congested (highest data packet throughput). Occasionally the dash line exceeds the solid line, e.g. around slot 140 on channel 2. But if we compare the four channels at the same time slot, the selected sender (with the highest dashed line among the four) also has the highest solid line among the four channels. This indicates that in the case of packet loss, ARQ-L is able to find the sender with the relatively best channel condition, and request retransmission from that sender. This ability is the key to

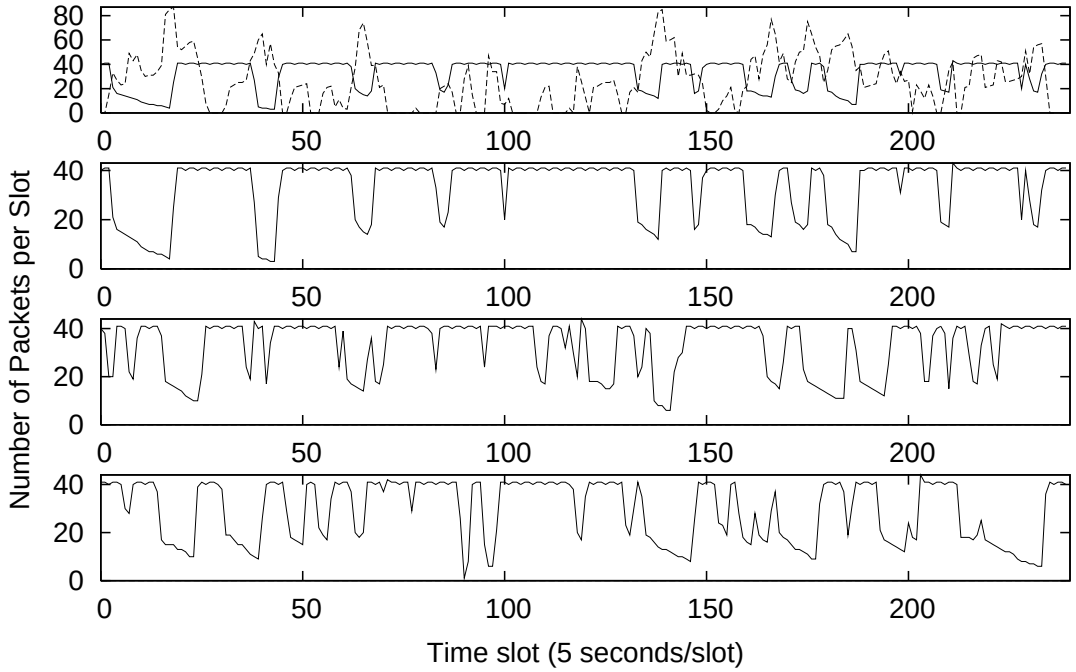


Figure 3.18: Throughput of channels in ARQ-D

improve the recovery effectiveness.

Scheme	ARQ-RR	ARQ-D	ARQ-L
Total data packet	30000 (100%)		
Received before recovery	19291 (64.30%)	19304 (64.35%)	19207 (64.02%)
Received after recovery	24517 (81.72%)	25477 (84.92%)	27295 (90.98%)
ARQ requested	10709 (100%)	10691 (100%)	10793 (100%)
ARQ received	5226 (48.80%)	6173 (57.74%)	8088 (74.94%)

Table 3.1: Statistics of each session.

We repeated the above simulation for 20 times for every scheme. The average value of (i) the total number of data packets, (ii) the number of received data packets before recovery, (iii) the number of received packets after recovery, (iv) the number of sent ARQ requests, and (v) the number of received ARQ packets in one session are listed in Table 3.1. The similar figures in the row *Received before*

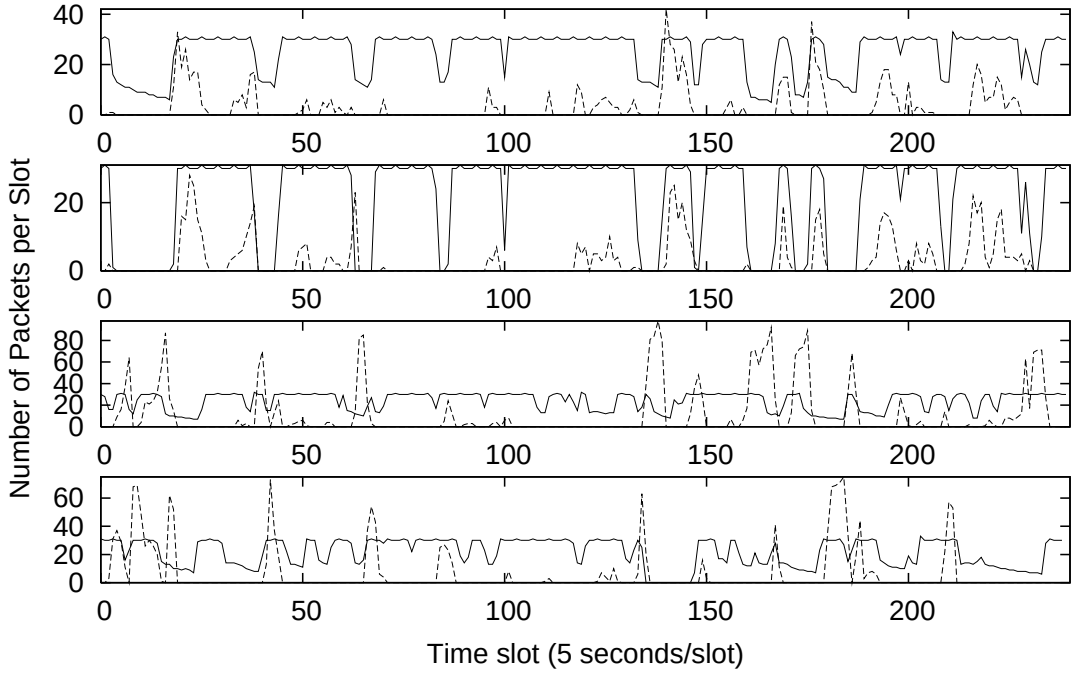


Figure 3.19: Throughput of channels in ARQ-L

recovery and *ARQ requested* indicate that the channel quality are generally the same for the three schemes, thus the comparison among them is fair. The figures in the row *Received after recovery* show the final received number of packets after applying the different retransmission scheme. A higher figure in this item stands for better overall performance. ARQ-L has the highest usable rate (90.98%). The figures in the row *ARQ received* shows how successful the retransmission is and reveals why a retransmission scheme produces certain level of usable rate. ARQ-L has the highest receiving rate on ARQ packet (74.94%). Overall, the above table demonstrates that ARQ-L outperforms the other two when congestion happens on the shared link. It recovers the highest number of lost packets among the three schemes.

3.4.3 Experiment over PlanetLab

Besides simulation with ns-2, we also conduct test over PlanetLab, which is an uncontrolled environment. In order to test the effectiveness of error recovery, we conduct our simulation in a large scale, hopefully to collect enough samples with packet loss during the streaming session. We randomly pick 92 nodes from PlanetLab. These 92 nodes are then randomly grouped into 23 groups, each of four nodes. Each of the group has one receiver and three senders, which are also randomly assigned. By doing so, we make the network path and condition as general as possible. Every group will repeat 30 runs of streaming. In each run, ARQ-RR, ARQ-L, and ARQ-D are tried consecutively to maintain temporal locality of the channel characteristics among the schemes. The dedicated retransmitter in ARQ-D is the sender with the smallest loss rate in ARQ-RR and ARQ-L tests in the same run. Each session is a 30-second MP3 streaming. We successfully collect 1372 session records from the experiments.

To show the three schemes actually encounter similar channel conditions in the uncontrolled environment, for every scheme, we plot the ratio of sessions that see a particular number of packet loss in all the channels. The graphs are shown in Figure 3.20. We have the following observation: (i) Number of packet loss per session is exponentially distributed for every scheme. (ii) ARQ-D has a slightly better lossy environment than the other two schemes. Lower percentage of its sessions have large number of packet loss. (iii) The test environment for the three schemes are comparable to each other. It is fair to compare the recovery effectiveness of ARQ-L with that of ARQ-RR. The average number of channel packet loss are: 17, 16 and 16 per session, for ARQ-D, ARQ-RR and ARQ-L respectively.

Among the 1372 session records, 1064 of them were collected when at least one channel has at least one packet loss (channel lossy condition). Out of these 1064

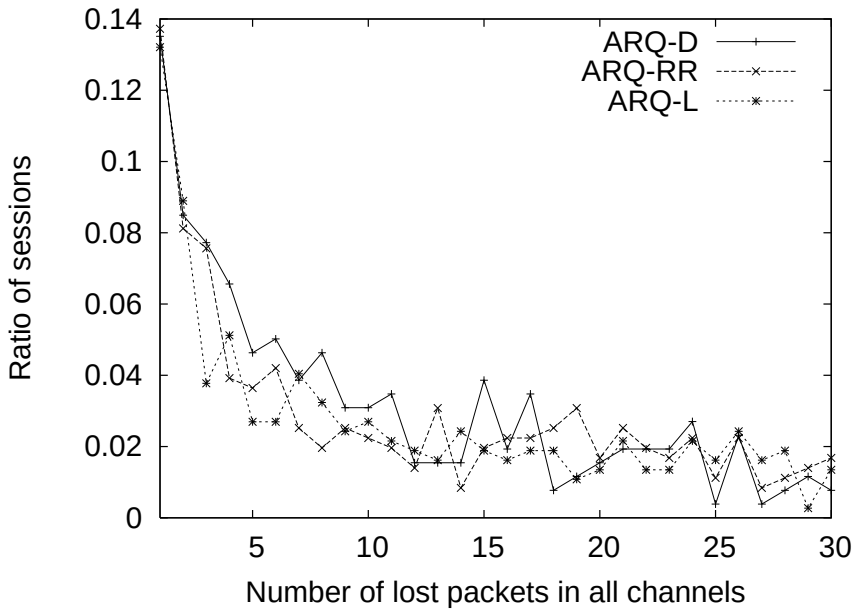


Figure 3.20: Similar network environment for the three schemes.

recordes, 378 are records of ARQ-D; 331 are from ARQ-RR; and 355 are from ARQ-L.

Under the uncontrolled environment, we have no idea of when, where and how long the congestion happens. So it is hard to compare the three schemes in exactly same condition. But it is meaningful to see in this uncontrolled environment, how much ratio (in total tries) of a particular scheme can give a certain level of effective loss rate. We look at the number of effective ADU loss (ADUs that are not recovered) per session. The ratio of sessions (in lossy condition) with particular number of effective ADU loss is plotted. A good scheme has high ratio of sessions with zero or small number of effective ADU loss, meaning a large portion of the sessions in lossy environments lose zero or few ADUs. The graph is shown in Figure 3.21.

The graph shows that ARQ-RR fails to recover more packets compared to the other two schemes. Only a bit more than 37% of its session in lossy channel stream-

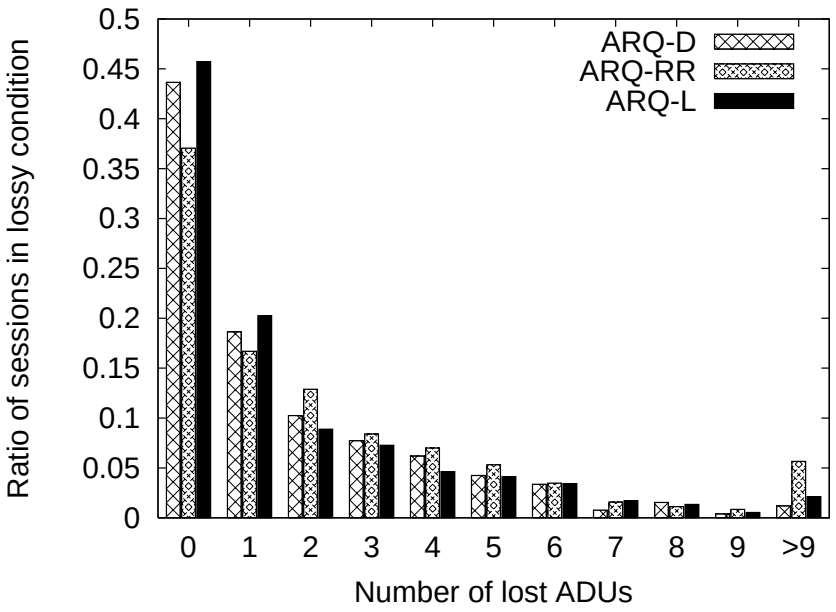


Figure 3.21: Ratio of sessions with particular number of effective loss

ing has zero loss, while ARQ-D and ARQ-L has 43.65% and 45.72% respectively. Compared to the other two schemes, ARQ-RR is less skewed, meaning a higher chance to lose more ADUs. ARQ-L is the most skewed, showing that ARQ-L successfully reduces number of packet loss per session. In this lossy environment, the average number of effective ADU loss per session are 1.6702, 2.0356 and 1.5333 for ARQ-D, ARQ-RR, and ARQ-L respectively. ARQ-L has the smallest number of unrecoverable ADU loss. Although comparing to ARQ-D, the reduction on effectiveness may not look significant, but this is due to the good network environment during the test. During our test, the network shows few bursty losses and most of the packets are sporadic, diminishing the advantage of ARQ-L to forecast the best channel for retransmission. ARQ-L, however, is obviously better than ARQ-RR, showing that its effectiveness of error recovery is above the average.

The simulation and experiment over the Internet show that ARQ-L offer better performance when burst loss appears, especially when the dedicated retransmitter

in ARQ-D is congested, as it avoids retransmission through congested channel.

Although ARQ-L does not assume any topology knowledge of the streaming session, and either does it identify the location of packet loss in the topology, ARQ-L is able to avoid retransmission from a path that is suffering from burst loss. The assumption is that, when a router is congested (hence burst loss happens), all the flows passing through it suffer from packet loss. By observing packet loss, ARQ-L avoids retransmitting on these congested paths.

ARQ-D has a degraded performance if shared path of data channel and ARQ channel are congested. In the analysis in Section 3.3, we assume channel independence. Correlated loss of two channels is not considered. In practice, however, where distributed media streaming has a reverse-tree-like topology, and therefore losses on two channels would be highly correlated if their shared path is congested. In this case, the ARQ-D's retransmitter, despite having a lowest loss rate in long term, has a high probability of losing ARQ packets.

3.5 Conclusion

In this chapter, we discuss the problem of using retransmission for error recovery in distributed media streaming. First, we propose distributed retransmission as a new principle for retransmission. The effectiveness of distributed retransmission is shown by analysis, simulation, and experiments over the Internet. Second, we propose a dynamic distributed retransmission scheme, ARQ-L, which selects retransmitter according to current loss rate of the channels. Our simulation and experiment show that ARQ-L copes well with burst loss and outperforms the other two naive distributed retransmission methods.

In our study, we assume only one retransmission, in order to compare the different schemes fairly. But in reality multiple retransmissions can be allowed

to further reduce loss rate. How multiple retransmissions can be performed given different delays and playback deadline for the video packets could be an interesting problem for future study.

Chapter 4

Congestion Control in Distributed Media Streaming

Retransmission improves the quality of playback. But whether distributed media streaming system can be deployed not only relies on the quality being delivered, but also depends on whether the deployment is fair to other applications sharing the Internet. One issue is whether distributed media streaming regulates its bandwidth consumption responsively and fairly according to the network condition. This chapter presents a method to achieve this responsiveness via congestion control.

4.1 Introduction

Existing studies on distributed media streaming mainly focus on improving the streaming quality. Some of them reduce distortion [69, 81] or buffering time [97], the other improves server utilization [19, 37]. This chapter studies how to ensure traffic of distributed media streaming responsive to network congestion. According to Floyd [25], a major issue of the Internet is *the potential for future congestion collapse due to flows that do not use responsible end-to-end congestion control.* Since the current Internet is a shared network, irresponsible flows that do not regulate the

sending rate during congestion can deteriorate the situation and probably makes the network unusable. Congestion control is required for deployment of distributed media streaming system.

Since the first reported congestion collapse in the mid 1980's [66], the dominant protocol, TCP, has been re-engineered to incorporate Additive Increase Multiplicative Decrease (AIMD) congestion control scheme, which is acknowledged as one of the key factors to the success of the Internet [25]. Flows are regarded as good citizen of the Internet if their long term rate is equal to or less than a TCP flow in the same network, or TCP-friendly. TCP-friendliness is the *de facto* judgment of whether a flow is congestion-controlled and can to be deployed on the Internet.

In this study, we investigate the congestion control problem in distributed media streaming. Congestion control in media streaming with one sender is a well studied problem (e.g., see the survey by Widmer et al. [93] and references therein). In distributed media streaming, however, the problem of congestion control is more complicated than the single sender scenario.

A distributed media streaming session contains multiple media flows (called *DMS flows*) from different senders. These flows may or may not pass through the same bottleneck. Ensuring TCP-friendliness of each DMS flow is not sufficient: their combined throughput is larger than the other TCP flows on the same bottleneck. This unfairness encourages abuse by selfish users — by increasing the number of concurrent flows, a user can grab larger bandwidth share at the bottlenecks. We need a different type of congestion control – one that controls the *aggregate* throughput of the DMS flows such that their combined throughput is TCP-friendly. We call such aggregate congestion control as *task-level congestion control*.

Aggregate congestion control methods exist in the literature [4, 32, 73, 85], but do not apply to distributed media streaming. In distributed media streaming, the

flows from multiple senders converge on their way to the receiver, forming a reverse tree (see Figure 4.1 for an example). The DMS flows only share parts of their links, so they may experience different delay and congestion. The existing methods of aggregate congestion control, however, assume that the flows traverse through the same path and share the same bottleneck.

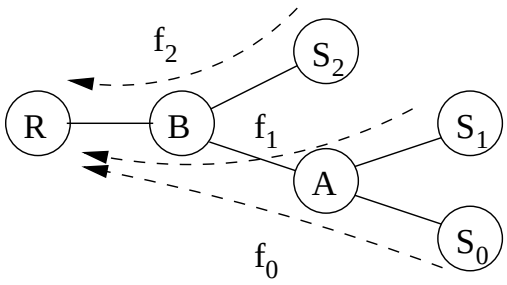


Figure 4.1: Reverse Tree Topology in Distributed Media Streaming.

We now illustrate the problem of congestion control in distributed media streaming through an example (see Figure 4.1). A host R requests for some media content from senders S_i. DMS flows (f_i) from the senders travel through different IP-level paths and join each other at routers A and B. We term routers like A and B as *aggregation points*. Throughout this study, we use the term *link* to refer to the set of physical links between a sender and an aggregation point (e.g., S_0-A), two aggregation points (e.g., A-B), or an aggregation point and the receiver (e.g., B-R). The set of DMS flows on a link is unique. Determining the set of DMS flows on a link is important, as it is the element upon which TCP-friendliness is enforced. Section 4.3 elaborates on this point.

In the reverse tree, congestion can occur on any link. If it occurs on R-B, the aggregate of f_0, f_1 and f_2 should be friendly to TCP flows on link R-B. But if the congestion occurs on A-B, only the aggregate of f_1 and f_2 needs to be friendly to TCP flows on link A-B. Flow f_2, on the other hand, can consume as much bandwidth as it wants. Similarly, if the congestion occurs on link S_0-A, only f_0

needs to be TCP-friendly.

The above example shows the difficulty in congestion control of distributed media streaming – the set of DMS flows to be controlled depends on where congestion appears. So the solution needs to first identify the flows sharing the same congestion, and then regulate them accordingly.

This study proposes a complete framework called DMSCC to achieve the above tasks. DMSCC tracks packet losses at the receiver as an indication of congestion and identifies the location of congestion by correlating the one-way delays between sender/receiver pairs. Additive increase, multiplicative decrease (AIMD) algorithm, with carefully adjusted increasing factor, regulates the throughput of the DMS flows on a bottleneck and produces a TCP-friendly flow aggregate. If there are k DMS flows on a bottleneck, they are regulated such that, in ideal situation, each flow consumes $1/k$ of the bandwidth of a TCP flow in a comparable network condition. As a result, the flow aggregate consumes as much as one TCP flow and is friendly to TCP. We use only TCP Reno in this study, but the scheme is applicable to other versions of TCP.

When the throughput of the each flow is regulated, the receiver needs to decide which packets each sender should send to conform to the new throughput constraint. This and other issues (e.g., what to retransmit, media coding methods used) are orthogonal to congestion control and are beyond the scope of this study.

The rest of the chapter is organized as follows. Section 4.2 presents some related work. In Section 4.3, we make a case for task-level TCP-friendliness and formulate the congestion control problem to achieve task-level TCP-friendliness in distributed media streaming. Section 4.4 describes the framework of DMSCC and presents our assumptions. Section 4.5 presents the methods to control throughput of DMS flows. Section 4.6 describes how DMSCC locates congestion in a reverse tree. Section 4.7 shows how DMSCC combines congestion location and throughput control to

achieve TCP-friendliness at the task level. Section 4.8 presents simulation results, which validate our design. Section 4.9 concludes the chapter.

4.2 Related Work

End-to-end, TCP-friendly, congestion control has been studied for many years. As new application appears, the research focus of this topic shifted from unicast, to multicast, and more recently to flow aggregate. We briefly review these work in this section.

Methods for congestion control of unicast can be categorized into either window-based or rate-based methods. Window-based methods [5, 29] use a congestion window that is similar to TCP. By adjusting the size of congestion window, the sending rate are controlled. Rate-based methods directly control the sending rate, and can be further divided into AIMD-based and equation-based methods. AIMD-based methods [80] apply TCP's AIMD algorithm to compute sending rate. Equation-based methods [34] calculate the sending rate using an equation, which takes network parameters (loss rate, RTT and MTU) as input, and output the estimated throughput of a conformant TCP flow.

Congestion control in IP multicast focuses on receiver-driven layered multicast protocols. The pgmcc scheme [82] focuses on how to scale feedback to large number of receivers. RLC [91] and PLM [47] focus on how to estimate and utilize the available bandwidth by changing the number of subscribed layers, while keeping the flows fair to TCP.

Congestion control for unicast and multicast aims to achieve fairness of one flow to TCP flows. In distributed media streaming, however, we want to achieve fairness of flow aggregate. This congestion control problem is therefore similar to the problem of aggregated congestion control.

Our work is more related to the study on aggregate congestion control. Aggregate congestion control pursues the fairness of a group of flows. Congestion Manager (CM) [4] uses one AIMD congestion window adjustment loop for the flow aggregate to achieve a fair combined throughput. CP [73] adopts equation-based rate adaptation [27] with packets sub-sampling to achieve fair bandwidth share. MPAT [85] keeps multiple bandwidth estimation loops and allows the application to allocate bandwidth to different flows while ensuring that the total throughput is fair. Hacker et al. study parallel TCP flows [32] and mimic TCP flows with longer RTT, so that flows in the aggregate consume less bandwidth than a TCP flow, making the aggregate TCP-friendly.

Aggregate congestion control is relatively new, and researchers still have different views on the definition of TCP-friendliness of flow aggregate. Some believe that the flow aggregate should be fair to one TCP flow, so that software that uses concurrent downloading do not gain advantage by establishing multiple flows, and therefore does not encourage abuse using multiple flows [32]. Others allow an aggregate of n flows to have equal bandwidth share to that of n TCP flows [73,85]. Their argument is that, since traditional TCP-friendliness is between flows, granting n flows a throughput equivalent to n TCP flows does not breach TCP-friendliness.

MulTCP [18] and TCP-P [12] also allow a group of TCP flows between the same sender receiver pair to adjust the aggressiveness and produce bandwidth that are equals to k normal TCP flows. Both work adjust aggressiveness by increasing the number acknowledgment before increasing congestion window. TCP-P [12] even allow k to be less than 1.

Regardless of the differences, these studies apply congestion control on a fixed set of flows. In distributed media streaming, congestion control needs to be applied on different sets of flows on different links. A new congestion control method is therefore required.

4.3 Problem Formulation

4.3.1 Task-level TCP-Friendliness

The term TCP-friendly is commonly used to describe a flow whose arrival rate at steady state is no more than the arrival rate of a TCP flow under the same network condition (such as packet loss rate and round trip time). We refer to congestion control schemes that aim to produce TCP-friendly flows as *flow-level congestion control*. Several work in the literature extends the notion of TCP-friendliness to coarser granularity. Hacker et al. [32] consider parallel TCP flows and propose an approach where multiple parallel TCP flows in one session are friendly to a single (unmodified) TCP flow. We call this approach *task-level congestion control*. Finally, congestion manager [4] seeks fairness of flow aggregate between a pair of hosts. We refer to this approach as *host-level congestion control*.

We believe that task-level congestion control is appropriate for Internet applications, including distributed media streaming. Congestion control pursues fair sharing of bandwidth at a bottleneck, and fairness is meaningful only when the entity of bandwidth consumption is identified. Such entity should have two properties: (i) An entity consumes bandwidth to complete a well-defined task for an end user; (ii) Creating more entities does not make completing the task better or faster. The second property is crucial in removing the motivation to abuse the network using multiple entities.

For example, an FTP file downloading session is an entity – the task is well defined, and downloading another file does not accelerate the completion of the current task. In this single-flow task, task-level congestion control is equivalent to flow-level congestion control. On the other hand, some applications (e.g., FlashGet[1]) allow users to download the same file with multiple flows concurrently. In this case,

[1] www.flashget.com

the multi-flow downloading session is one entity – (i) the task is still downloading of a file, and (ii) creating another multi-flow session for the same file does not speed up the current downloading. Task-level congestion control takes the whole downloading task as the entity of bandwidth consumption and keeps the total throughput friendly to TCP. Contrarily, flow-level congestion control only requires TCP-friendliness of individual flow. Therefore, the task consumes more bandwidth than a TCP flow, gaining advantage over other single-flow tasks. Without task-level TCP-friendliness, selfish users can use more flows to grab more bandwidth on bottlenecks.

4.3.2 The Criterion for Task-Level TCP-Friendliness

We now formally describe the goal of task-level congestion control.

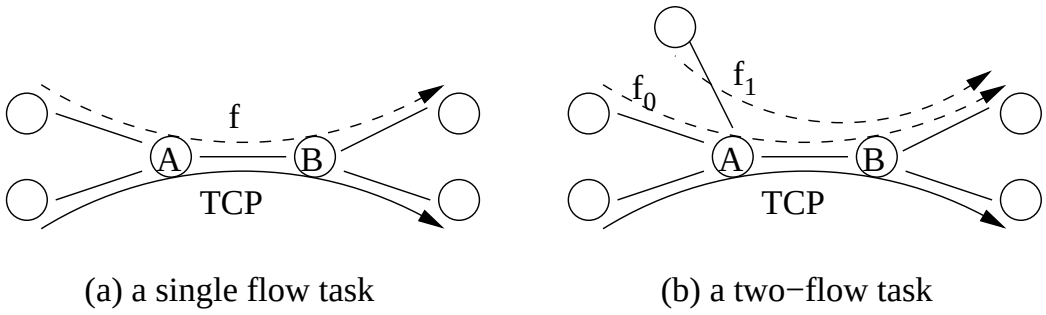


Figure 4.2: A Single-Flow Task and a Two-Flow Task.

A Single-Flow Task

First, let's consider a task with only one flow, as shown in Figure 4.2(a). The flow f and a TCP flow share bottleneck A-B. As the task has only one flow, task-level TCP-friendliness is equivalent to flow-level TCP-friendliness. Assuming that the RTT of both flows are the same, TCP-friendliness is achieved if the following equation holds:

$$B = B_{TCP}$$

where B and B_{TCP} are the throughput of f and the TCP flow, respectively.

Consider a more general case where the two flows experience different RTT. TCP's congestion control algorithm is biased against flows with larger RTT [28]. Despite efforts to correct such unfairness (e.g., TCP Libra [60]), this unfairness persists in current TCP implementations. On the other hand, B×RTT of the two TCP flows remain the same if they experience the same loss rate. For flow f and the TCP flow in Figure 4.2(a), it is reasonable to assume a similar loss rate: A-B is the only bottleneck on their paths, and active queue management, such as RED, tries to drop packets from both flows in a fair manner. Therefore, under different RTT, TCP-friendliness is ensured by:

$$B \times RTT = B_{TCP} \times RTT_{TCP} \tag{4.1}$$

where RTT and RTT_{TCP} are the RTT of flow f and the TCP flow, respectively.

A Multi-Flow Task

We now extend Equation 4.1 to handle a multi-flow task sharing the same bottleneck with other TCP flows.

Consider a multi-flow task (e.g., Figure 4.2(b)). The two flows f_0 and f_1 share bottleneck A-B with a TCP flow. Task-level TCP-friendliness requires the flow aggregate to be friendly to a TCP flow. If we treat the flow aggregate as a single flow, task-level TCP-friendliness is the same as flow-level TCP-friendliness. Therefore, Equation 4.1 holds; except that, B is now the combined throughput of f_i, and RTT is the average round trip time of f_i:

$$RTT = \frac{1}{B} \sum_{f_i \in O} (b_i \times rtt_i)$$

where O is the set of flows in the flow aggregate, b_i and rtt_i are the throughput and round trip time of flow f_i. By replacing B and RTT, we extend Equation 4.1

to consider multi-flow tasks:

$$\sum_{f_i \in O} (b_i \times rtt_i) = B_{TCP} \times RTT_{TCP} \tag{4.2}$$

Equation 4.2 provides the criterion for task-level TCP-friendliness on a given bottleneck. *Formally, a task is TCP-friendly if the combined $B \times RTT$ of its flows is equal to that of a TCP flow on the same bottleneck.*

The Goal of DMSCC

We now apply Equation 4.2 to the problem of congestion control in distributed media streaming. Consider a distributed media streaming session as shown in Figure 4.1. As bottlenecks form on different links, the flow aggregates on them contain different sets of DMS flows. The criterion of task-level TCP-friendliness for distributed media streaming should consider multiple bottleneck locations with different sets of flows.

Let l_j be a link, and a TCP flow passing through l_j be TCP_j. As the set of DMS flows flowing through each link is distinct, we can represent a link using its set of DMS flows. We use set notations to represent relationships among the flows and the links. The notation $f_i \in l_j$ means that flow f_i passes through link l_j; and $l_i \supset l_j$ means that the flows on l_i are a proper superset of flows on l_j, or l_i *dominates* l_j for short.

Distributed media streaming is task-level TCP-friendly when, *on any bottleneck* l_j, the following inequality holds:

$$\sum_{f_i \in l_j} (b_i \times rtt_i) \leq B_{TCP_j} \times RTT_{TCP_j} \tag{4.3}$$

The above criterion is an inequality, as a DMS flow may experience multiple bottlenecks.

4.4 Model and Assumptions

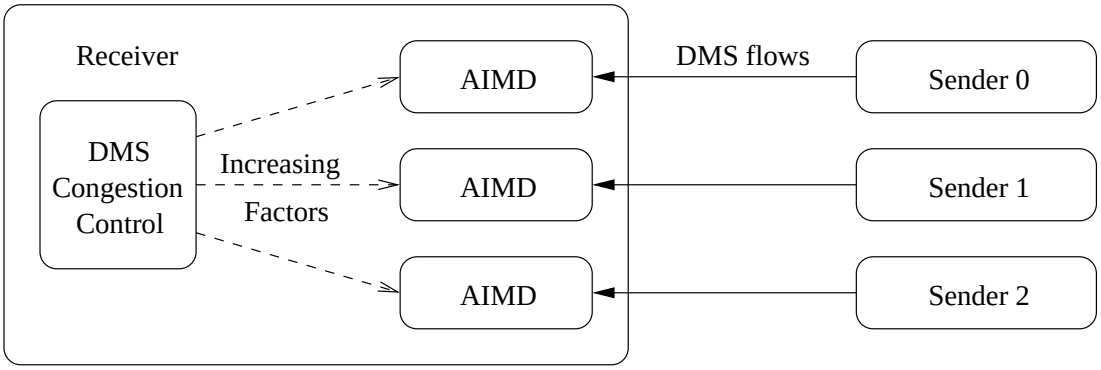


Figure 4.3: A Three-Sender Session.

Our congestion control scheme, DMSCC, is designed to ensure that Inequality 4.3 is satisfied on any congested link in a distributed media session. DMSCC is a receiver-driven protocol – the receiver pulls the data from the senders by sending requests with sequence numbers, and the senders reply with data. The receiver therefore controls the sending rate of each senders and is the natural place to implement the congestion control protocol.

Figure 4.3 shows the relationship between DMSCC and the DMS flows in a distributed media streaming session with three senders. There are three connections between the receiver and the senders. At the receiver, each connection is controlled by an AIMD loop similar to TCP. The increasing factors of these AIMD loops are controlled by the DMSCC module in the receiver. We will show in Section 4.5 how the increasing factors of individual DMS flows are determined. But first, in this section, we introduce the framework of DMSCC and present our assumptions in the design of our protocol.

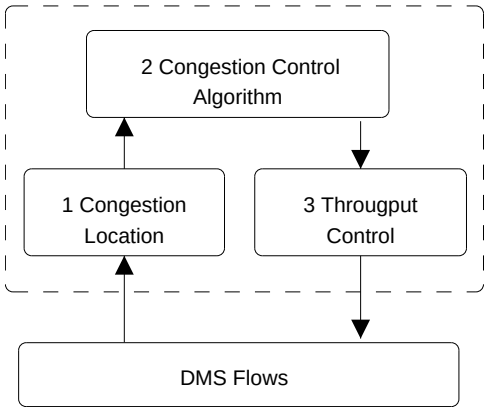
4.4.1 AIMD versus Equation-Based

AIMD and equation-based method [27] are two common methods for regulating the throughput of a non-TCP flow. We use AIMD method to regulate DMS flows in DMSCC for the following reason.

Equation-based methods rely on long term observation of network parameters such as loss rate and smoothed RTT. These parameters are used in an equation to estimate the long term throughput that is fair to TCP. This long term observation is meaningful only in cases where flows share the same path, and bottlenecks affect the same set of flows. In distributed media streaming, the congestion may affect different set of DMS flows at different bottlenecks. Thus, a long term observation might become outdated and fail to capture the congestion on a particular bottleneck. On the other hand, AIMD methods respond quickly to a packet loss and adapt swiftly to congestion on new bottleneck. Although it is argued that AIMD produces saw-tooth like throughput, in non-interactive streaming, as in the case of distributed media streaming, buffering can be used to smooth the playback at the receiver.

4.4.2 DMSCC

The framework of DMSCC is shown in Figure 4.4. DMSCC has two relatively independent functionalities: throughput control (Section 4.5) and congestion location (Section 4.6). These two functionalities cooperate to perform task-level congestion control on DMS flows. When congestion occurs, the congestion location module identifies the bottleneck. The throughput control module then updates the increasing factor of AIMD loops of each DMS flow on that bottleneck.



1. Congestion location module monitors packet loss and delays and estimates the location of congestion.
2. Based on the location of congestion, congestion control algorithm decides the increasing factor of each flows.
3. Throughput control module then set increasing factors of each flow to regulate their rate.

Figure 4.4: Framework of DMSCC

4.4.3 Assumptions

Before proceeding to descriptions of DMSCC, we first clarify our assumptions. First, we assume that the paths among the receiver and senders form a reverse tree rooted at the receiver, and this topology is known by the receiver. Second, we assume that DMS flows on the same bottleneck link experience similar loss rate. This assumption is reasonable when active queue management schemes such as RED is used. Third, we focus on links with high multiplexing factors, where loss rate is decided by the background traffic rather than the DMS flows. Lastly, we can reasonably assume that the number of senders in a DMS session is typically small (less than 10). Thus, scaling DMSCC to large number of senders is not an issue.

4.5 Throughput Control

In this section, we describe how to control the throughput of a DMS flow using AIMD algorithm such that it achieves a fixed fraction of the throughput of a TCP flow. In order for an aggregate of k DMS flows to be fair to a single TCP flow, DMSCC tries to control the throughput of each of the DMS flow to be $1/k$ of the throughput of a conformant TCP flow.

We derived our method from the well-known Mathis Equation [62]. Mathis et al. assume that packet losses are distributed in such a way that, if the loss rate is p, then for every $1/p$ packets, one packet is lost. Figure 4.5 shows the variation of congestion window in such an ideal lossy channel. W denotes the size of the congestion window (in number of packets) before packet loss. Every packet loss reduces the congestion window to $W/2$. The congestion window then increases by α packets for every RTT, until the next packet loss occurs.

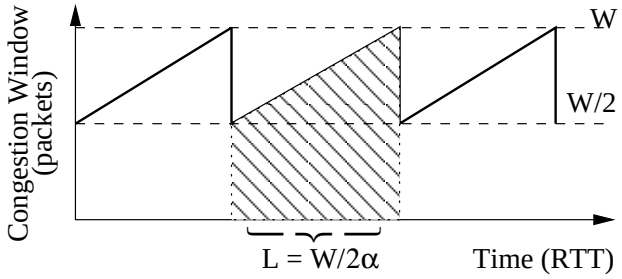


Figure 4.5: Evolution of Congestion Window Under Periodic Loss.

The variable α is the *increasing factor*. If we let the period (in RTT) between every two packet losses be L, then

$$L = \frac{W/2}{\alpha}.$$

The total number of packets received during that period can be calculated as the size of the shaded area S:

$$S = \frac{3}{2} \times \frac{W}{2} \times L = \frac{3W^2}{8\alpha}. \tag{4.4}$$

From the assumption of ideal packet loss pattern, we know that the number of packets between two packet losses is $1/p$, that is,

$$S = \frac{1}{p}. \tag{4.5}$$

From Equation 4.4 and 4.5, we obtain:

$$W = \sqrt{\alpha} \times \sqrt{\frac{8}{3p}}.$$

The throughput of a flow is proportional to the average size of congestion window, which is:

$$\overline{W} = \frac{3}{4}W = \sqrt{\alpha} \times \frac{3}{4}\sqrt{\frac{8}{3p}}. \tag{4.6}$$

Equation 4.6 provides us a way to change the throughput by adjusting its increasing factor α. If we want a DMS flow to have β times the throughput of a TCP flow, whose increasing factor is 1, then

$$\begin{aligned} &\overline{W} = \beta \times \overline{W_{TCP}} \\ \Rightarrow &\sqrt{\alpha} \times \frac{3}{4}\sqrt{\frac{8}{3p}} = \beta \times \frac{3}{4}\sqrt{\frac{8}{3p}} \\ \Rightarrow &\alpha = \beta^2. \end{aligned} \tag{4.7}$$

Equation 4.7 tells us that, for the throughput of a DMS flow to be β times of a conformant TCP flow, we need to set its increasing factor α to β^2. We tested this observation in the following simulation (Simulation 1) using ns-2.28.

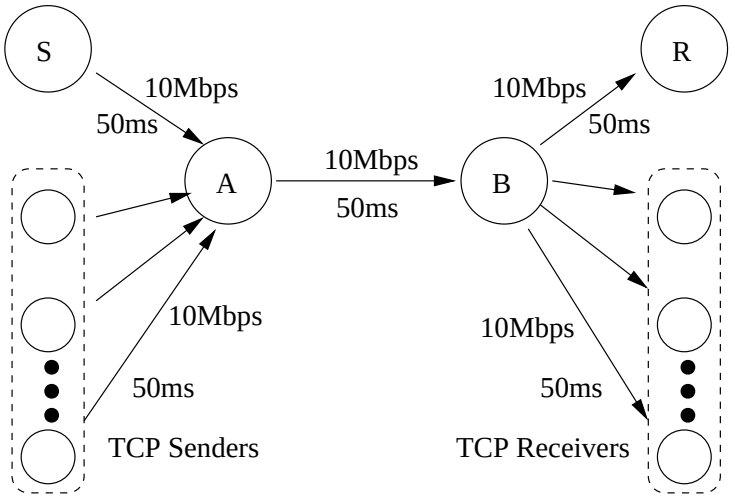


Figure 4.6: Topology of Simulation 1.

The topology of the simulation is shown in Figure 4.6. The bottleneck between nodes A and B has a bandwidth of 10Mbps and a delay of 50ms. Node A is a

RED gateway using ns-2.28 default setting[2]. Fifty TCP Reno flows pass through the bottleneck and produce congestion. A DMS flow is sent from S to R. Its increasing factor α changes based on the value of β according to Equation 4.7. We increased β from 0.1 to 1.4 (note that in DMSCC, we are interested only in $\beta \leq 1$) and observed the ratio of the throughput of the DMS flow to the average throughput of TCP flows. For each value of β, we repeated the simulation 20 times and computed the average ratio.

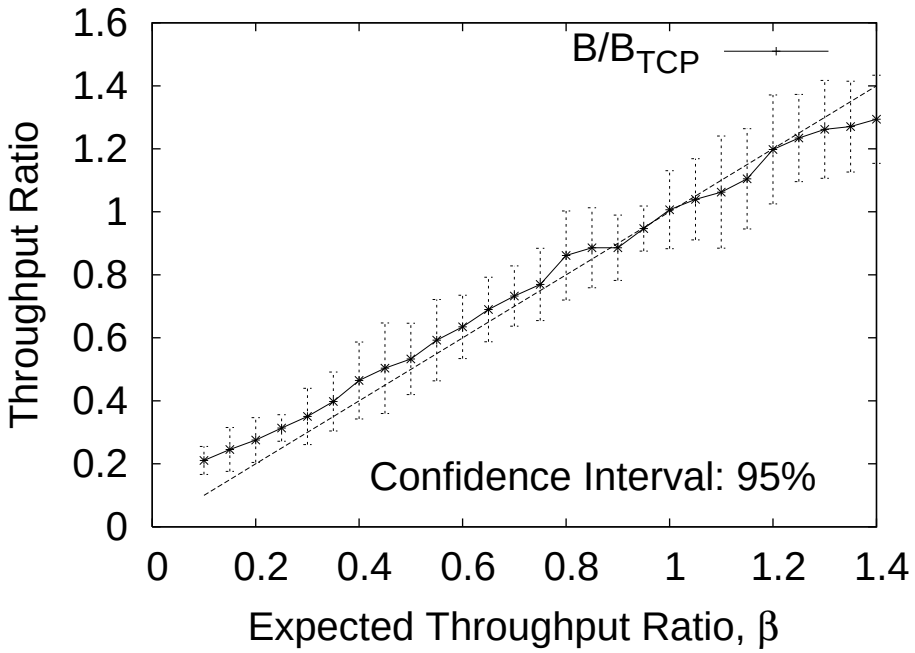


Figure 4.7: Simulation 1: Throughput Ratio as β Changes.

The result is plotted in Figure 4.7. The x-axis, β, is the expected throughput ratio (β). The y-axis is the ratio observed when setting α to β^2. Figure 4.7 shows that as β changes, the actual throughput ratio is close to β when β ranges from 0.2 to 1.0. The result shows the effectiveness of Equation 4.7.

Mismatch between the actual throughput ratio and β is observed in Figure 4.7 for small β and large β. This mismatch is due to bursty packet losses in the

[2]queue length = 50, min_thresh = 5, max_thresh = 15, gentle-enabled, and mark_p = 0.1

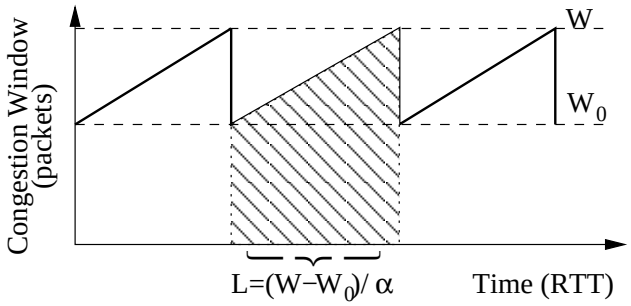


Figure 4.8: Effects of Minimum Congestion Windows.

simulation, which violates the assumption that packet losses are evenly distributed. During the bursty loss period, the congestion window becomes small. When the halved congestion window is less than the minimum window, the latter dominates the throughput and skews the throughput ratio from β. We elaborate on this below.

To study the effects of minimum congestion window over throughput, we make a similar deduction as in Figure 4.5. Let W_0 be the minimum window. When the loss rate is high, congestion window is rarely greater than $2 \times W_0$, since it encounters packet losses frequently. On every packet loss, as $W_0 > W/2$, the congestion window is reduced to W_0. Figure 4.8 shows the evolution of window size in this situation. We can derive W as:

$$\begin{cases} L = \frac{W - W_0}{\alpha} \\ S = WL - \frac{(W-W_0)L}{2} \\ S = \frac{1}{p} \end{cases}$$
$$\Rightarrow W = \sqrt{\frac{2\alpha}{p} + W_0^2}.$$

In a TCP flow, α equals to 1, hence the throughput ratio can be represented as:

$$\frac{B}{B_{TCP}} = \frac{W}{W_{TCP}} = \sqrt{\frac{2\alpha + pW_0^2}{2 + pW_0^2}}$$

We further divide this value by β and denote the resulting value as R. Ideally, R equals to 1 (i.e., throughput ratio equals β).

$$R = \frac{B/B_{TCP}}{\beta} = \sqrt{\frac{2 + pW_0^2/\alpha}{2 + pW_0^2}} \tag{4.8}$$

$$= \sqrt{1 + \frac{(1-\alpha)W_0^2}{2\alpha/p + \alpha W_0^2}} \tag{4.9}$$

Equation 4.8 tells us that when the size of congestion window is dominated by the minimum window size, smaller α (therefore smaller β) increases R, i.e., the throughput of the DMS flow becomes larger than expected. Similarly, larger α (and β) decreases R, and the throughput of DMS flow is less than expected. This equation explains the discrepancy between B/B_{TCP} curve and the expected line in Figure 4.7.

Equation 4.9 tells us that, for a DMS flow with a given α ($\alpha < 1$), if loss rate p increases, R ($R > 1$) will increase, i.e., the actual throughput will be larger than the expected value, and the difference will be enlarged.

Although mismatch of the throughput ratio exists and is found to be inevitable in lossy environment, the method still manage to control the throughput of a flow to reasonable level of accuracy. Note that when the channel is highly lossy, media streaming is generally not usable anyway. Thus the larger mismatch in throughput ratio in this case is less of a concern in our context.

We have described our method to control the throughput of a DMS flow on a bottleneck. To apply it in DMSCC, we need to find out where the bottlenecks are, so that we can regulate the throughput of DMS flows on these bottlenecks. We describe our approach to locate the congested bottlenecks in the next section.

4.6 Congestion Location

An ideal solution to locate a congestion should work as follow: (i) when a congestion causes a packet loss on a DMS flow, the solution should be able to tell which link is congested, so that DMS flows on the affected link can be regulated, (ii) when the congestion subsides, the solution should sense it, so that the regulation on the DMS flows previously imposed can be lifted. Such ideal solution is difficult to achieve in a tree topology: (i) there may be multiple, simultaneous congestion on different links in the tree, and (ii) the same flow might experience congestion on different links.

Rubenstein et al. [83] partially solved this problem for the case with one shared bottleneck. Based on the observation that a shared congestion produces highly correlated one-way delay on flows, they compare the cross-correlation of two flows and the auto-correlation of one of them. The shared bottleneck link is identified as one where the cross-correlation is larger. For details of Rubenstein's technique, please refer to the original paper [83].

Rubenstein's method works well when each flow experiences one congestion. To use the same correlation test when a flow passes through multiple congested links is difficult. On a shared bottleneck, the delay of the flows might contain too much noise induced by other congested links. Solving the congestion location problem completely in the distributed media streaming scenario remains a difficult and open problem. In this study, we extend Rubenstein's method to identify multiple bottlenecks in the case where the delay values on the shared bottleneck has limited interference from other congested links.

We use *CorrTest(i,j)* to denote the correlation test of Rubenstein applied on flow i and flow j. When *CorrTest(i,j)* returns 1, the two flows share a bottleneck; when it returns 0, no shared bottleneck is detected. We apply the test over a

window of one-way delays recorded using probe packets sent together with flows i and j. We use probes to maintain certain minimum sampling frequency. Without probes, flows i and j may not send any packet for a long period due to congestion windows. Probes are tiny packets that consume negligible bandwidth (0.8KBps in our simulation). In the rest of this section, we explore congestion location step by step, and then propose our method.

First, consider a simple case where only one link is congested. In this case, we can directly apply the correlation test. The method is listed as Algorithm 1, The method is called whenever a packet loss is detected on flow f_i. It applies correlation test on (the probes of) f_i and other DMS flows and adds DMS flows that are correlated with f_i into a set C_f. The least dominant link that contains the set of flows in C_f is returned as the shared bottleneck.

Algorithm 1 OneBottleneck(f_i)

INPUT: f_i {the flow whose packet is lost}

Let F be the set of all flows and L be the set of all links;

$C_f \leftarrow \{f_j | CorrTest(i,j) = 1, \forall f_j \in F\}$;

$C_l \leftarrow \{l | l \supseteq C_f, l \in L\}$;

OUTPUT: Link $l \in C_l$ such that $l_k \supseteq l, \forall l_k \in C_l$;

The situation is more complex when two links are congested simultaneously. For instance, in Fig 4.1, when two bottlenecks S_0-A and A-B coexist, one-way delay of f_0 is worsen by both congestion, but one-way delay of f_1 is only affected by congestion at A-B. When a packet is lost, $CorrTest(0,1)$ can return either 1 or 0, depending on which bottleneck dominates value of delay during that sampling period. When the queuing delay on one bottleneck is temporally reduced by congestion control of background traffic or packet dropping, the queuing delay on the other bottleneck can remain high and continue to dominate the end-to-end delay.

So, $CorrTest(0, 1)$ may return 0 even when A-B is congested due to domination of bottleneck S_0-A on the one-way delay of f_0, making it less correlated with f_1.

Whereas a $CorrTest(0, 1)$ value of 0 does not necessary imply no shared bottleneck, a value of 1, however, does confirm the existence of shared congestion on f_0 and f_1. Our observation is that, if the congestion is shared, $CorrTest(0, 1)$ may return 1 from time to time after every packet loss. Based on this observation, we use a history-based method to update the set of current bottlenecks. We denote C as a set of current congested links and H as a FIFO queue of previously detected congested links due to the most recent h packet loss. When a packet loss is detected on f_i, H is updated as in Algorithm 2.

Algorithm 2 OnPacketLoss (f_i)

INPUT: f_i, H, h

$l \leftarrow OneBottleneck(f_i)$;

if $|H| = h$ **then**

  $dequeue(H)$; {phase out old bottleneck}

end if

$enqueue(H, l)$; {phase in new bottleneck}

$C \leftarrow \{l | l \text{ in } H\}$;

OUTPUT: C, H

We can view H has a history of bottleneck detection record. On every packet loss, the oldest record in H is phased out. If no other records in H refers to the same bottleneck, the bottleneck is removed from the output. In other words, if a link is not identified as a bottleneck during the most recent h packet loss event, the congestion on the link is likely to have subsided. The length of the queue, h, should be long enough so that H is able to buffer all current congested links. If h is too small, H may phase out existing bottlenecks and update C incorrectly.

On the other hand, h needs not be too large, as the probability of having many simultaneous bottlenecks is small. Our experiments on a four-sender session show that value of h beyond 8 produces little improvement in accuracy of C, so we use $h = 8$ in our protocol.

After C is updated by Algorithm 2, C contains the set of current bottlenecks. For instance, in the previous example with simultaneous congestion on link S_0-A and A-B, Algorithm 2 may return C =$\{S_0$-A, A-$B\}$ or $\{S_0$-A, A-B, S_1-$A\}$. In the second set, S_1-A is a false detection. To understand this, imagine that the bottleneck A-B causes a packet loss on f_1. When performing $CorrTest(1, 0)$, the result can be 0 as we have analyzed. Therefore, Algorithm 1 returns S_1-A as a bottleneck. But, fortunately, the false detection does not affect the correctness of DMSCC, as we shall see in the next section.

4.7 Congestion Control

4.7.1 Updating the Increasing Factors

After identifying the set of bottlenecks, the next step is to adjust the increasing factors of the DMS flows on the bottlenecks so that their combined throughput is TCP-friendly. Given C, the set of current bottlenecks, Algorithm 3 construct another set C' containing the set of bottlenecks that are not dominated by any other bottlenecks in C. For each of the bottlenecks in C', the algorithm sets the increasing factor of the DMS flows that pass through it to $1/n^2$ according to Equation 4.7, where n is the number of DMS flows going through a bottleneck.

To understand the reason why DMS flows are adjusted according to the dominant bottlenecks, let us consider the previous example of simultaneous congestion on S_0-A and A-B in Figure 4.1. Suppose that, after a packet loss, Algorithm 2 returns C =$\{S_0$-A, A-B, S_1-$A\}$. Link A-B dominates the other two links. Conges-

Algorithm 3 UpdateAlpha (C)

INPUT: C

$C' \leftarrow \{l|\ \nexists l_i \in C : l_i \supset\ l, l \in C\}$;

for all $l \in C'$ **do**

$n \leftarrow |\{f_i\}|, f_i \in l$; {number of DMS flows}

$\alpha_i \leftarrow 1/n^2, \forall f_i : f_i \in l$; {increasing factor}

end for

tion on A-B requires the aggregate of f_0 and f_1 to be TCP-friendly. According to Equation 4.7, α_0 and α_1 should be set to 1/4. Congestion on the other two links requires each of f_0 and f_1 to be TCP-friendly and thus both α_0 and α_1 should be set to 1. Setting the increasing factor to 1, however, makes the flow aggregate on A-B unfriendly. Considering the goal of DMSCC (Equation 4.3), α_i should be set conservatively to 1/4. In short, the dominant bottleneck restricts the aggressiveness of the DMS flows, and therefore the increasing factor should be set according to the dominant links. This property also allows Algorithm 4.2 to return false bottlenecks (e.g. S_1-A) that are dominated by the shared bottleneck (e.g., A-B). Such false bottlenecks do not affect the correctness of DMSCC.

4.7.2 Bottleneck Recovery

The above mentioned algorithms run whenever a packet loss is detected. When congestion subsides and there is no more packet loss, we need to reset α_i to 1 so that the network bandwidth can be fully utilized. Having no packet loss to trigger the reset of α_i, we adopt a timer-based method. A timer is refreshed when packet loss is detected. If no packet loss is detected within t seconds, the increasing factors of all DMS flows are reset to 1. This method ensures that after congestion disappears, in at most t seconds, αs are reset to allow DMS flow to

fully utilize available bandwidth. But if the bottleneck is still there when timer expires, resetting all α will make the flow aggregate unfriendly to TCP. To prevent such over aggressiveness of DMS flows, we (i) set t conservatively long (15 seconds in our simulation), and (ii) retain the value of C and H while resetting α_i. The latter helps Algorithm 2 to set α_i back to the right value immediately if packet loss reappears.

4.8 Simulation and Discussion

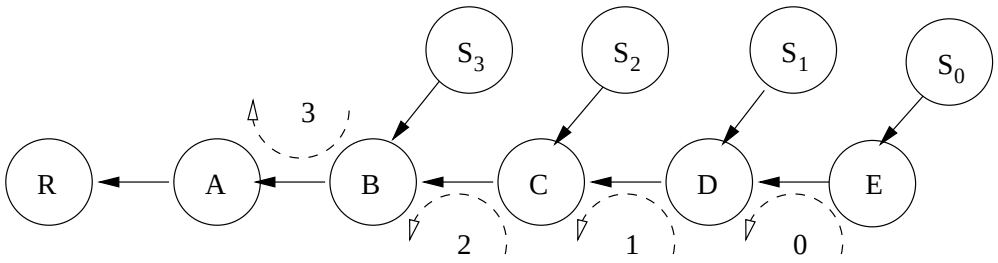


Figure 4.9: Topology of Simulation 2.

We constructed Simulation 2 in ns-2.28 to validate our design. Figure 4.9 shows a topology with four senders S_0, S_1, S_2, and S_3, and one receiver R. DMS flows converge on the way to R in the order of S_0, S_1, S_2, and S_3. Besides f_i, the senders also send CBR probes to the receiver using UDP, at 40 bytes per packet, 20 packets per seconds. The sample length for one-way delay records is 20 (one second in length) for correlation computation; according to Rubenstein et al. [83] this length gives nearly 90% of accuracy in correlation test. All links are configured with bandwidth of 5Mbps, delay of 20ms, and default RED setting in ns-2.28. Background traffic may congest link l_0, l_1, l_2 or l_3 to produce bottleneck. The background traffic consists of 20 TCP Reno flows on every bottleneck. The RTTs of background TCP flows are set to 120ms.

The simulation aims to show that DMSCC leads to task-level TCP friendliness,

achieving our goal stated at the end of Section 4.3. When background traffic produces congestion on a link, the throughput of f_i and the RTT_i are measured to calculate B×RTT of the flow aggregate on the link. The average B×RTT of the TCP background flows is also calculated. If B×RTT of the flow aggregate is less than or equal to the average of a TCP flow, then task-level TCP-friendliness (Equation 4.3) is achieved. Figure 4.10 shows B×RTT of the TCP flows (average) and the flow aggregate; each subgraph corresponds to one link.

To show the ability of DMSCC to identify the dominant bottleneck, we generate background traffic such that Link 0 is congested during time 0 - 100s and Link 2 is congested during time 50 - 150s. The first subgraph in Figure 4.10 shows that during time 0 - 50s, B×RTT of the flow aggregate on Link 0 roughly equals to a TCP flow; and the third subgraph shows that during time 100 - 150s, B×RTT of the flow aggregate on Link 2 is also similar to a TCP flow. This confirms that task-level TCP-friendliness is achieved when only one congestion exists in the topology. During time 50 - 100s, when both Link 0 and Link 2 are congested, we find that B×RTT of the flow aggregate on Link 2 equals to a TCP flow, and that B×RTT of the flow aggregate on Link 0 is less than a TCP flow. This result demonstrates that DMSCC is able to identify that Link 2 dominates Link 0, and therefore sets the increasing factor accordingly to achieve task-level TCP-friendliness in the case of simultaneous congestion.

To show that DMSCC is able to utilize bandwidth fully when congestion disappears, Link 1 and Link 3 are congested during time 150 - 200s and 250 - 350s respectively. The streaming session completes at time 400s. In the second subgraph (Link 1), at 200s, TCP flows disappear. The value of B×RTT of the flow aggregate increases quickly to the maximum playback rate of the media, demonstrating that the available bandwidth is fully utilized when there is no congestion. We can also see in the last subgraph (Link 3), starting from time 350s, B×RTT

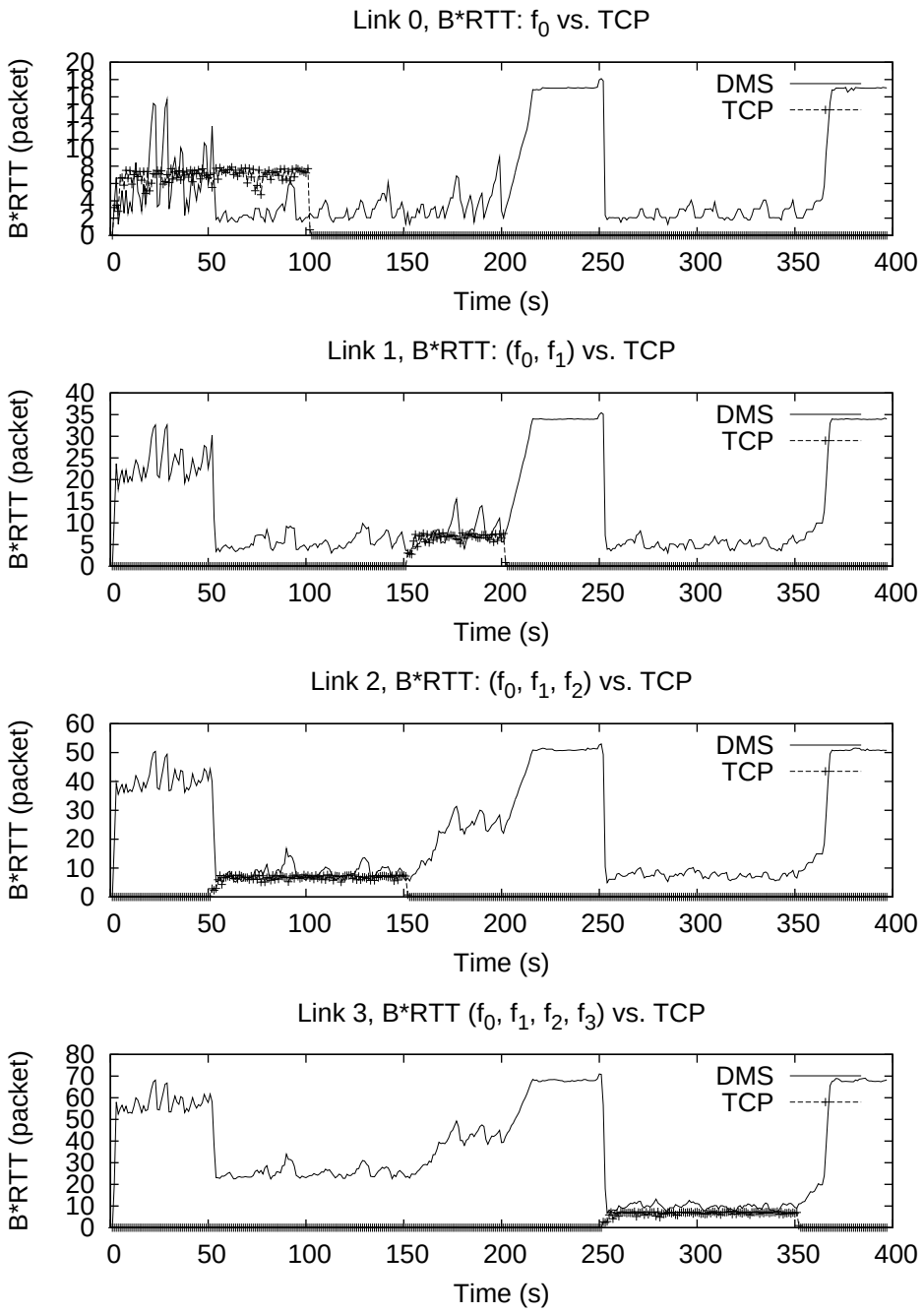


Figure 4.10: B×RTT of Aggregate DMS flows and TCP Flow on Each Links.

of the flow aggregate increases slowly at first and increases faster later. The small slope is due to the small increasing factor ($\alpha_i = 1/16$), which is determined by the congestion on Link 3. But when packet is not seen for a period of 15 sec (value of t in this simulation), it is likely that the congestion has disappeared. DMSCC

therefore sets α_i to 1, allowing throughput to increase quickly, achieving better bandwidth utilization.

4.8.1 The sensitivity of h

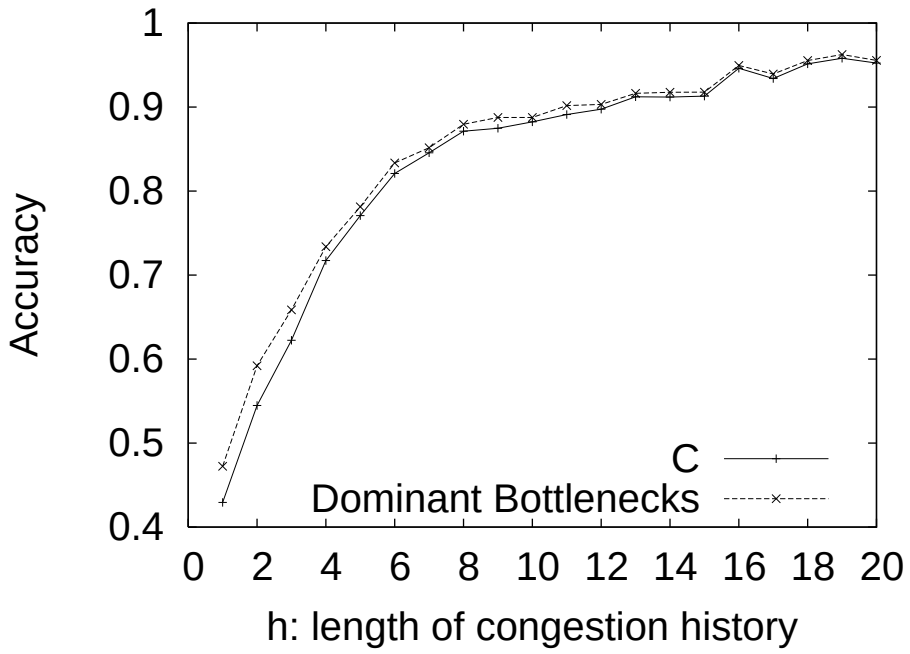


Figure 4.11: Effects of h on Accuracy of C and Dominant Bottlenecks.

When presenting Algorithm 2, we mentioned that the length of congestion history h controls the update frequency of C, and, therefore, affects the accuracy of α. The value of h is empirically set to 8 in the simulation. We changed h from 1 to 20 and ran the above simulation 50 times each. On every packet loss, C (the detected bottlenecks) is compared with the actual bottlenecks, and the detected dominant bottlenecks (which affects the value of α) are compared with the real dominant bottlenecks. The accuracy is defined as the number of correctly detected bottlenecks over the number of bottlenecks. Figure 4.11 shows the average accuracy of C and dominant bottlenecks, when h changes. The accuracy of dominant bottleneck is higher than that of C, indicating that even if false detection on bot-

tlenecks exists, the dominant bottleneck could still be correctly detected. After the value of h exceeds 8, both curves increase slower: larger h contributes less to the accuracy of α.

4.9 Conclusion

In this chapter, we introduce the problem of congestion control in DMS system. It differs from previous congestion control problems as it involves multiple flows traversing through different paths. A better definition of TCP-friendliness is needed to further explore the problem. We therefore introduce the notion of task-level TCP-friendliness in this study. We then formulate a criterion for task-level fairness in the context of distributed media streaming. We divide the problem of congestion control in distributed media streaming into two sub-problems. The first is how to locate congestion in a reverse tree topology. The second is how to control the throughput of a DMS flow using AIMD loop such that the combined throughput on the bottleneck is TCP-friendly.

This study is the first one to address the problem of congestion control in distributed media streaming. The concept of task-level TCP-friendliness gives a different perspective to the meaning of TCP-friendliness, and it is usable in other scenario such as peer-to-peer file sharing. Our method to control the aggregate throughput of DMS flows might be useful in other context as well, including controlling the throughput of parallel TCP connections.

DMSCC has several limitations. Our throughput control algorithm is based on Mathis equation, and therefore does not work accurately in all network conditions (e.g., when loss is frequent and bursty). Our congestion location algorithm relies on Rubenstein's method. Identifying location of congestion in multiple congestions scenario with high delay interference remains a challenging problem. Our future

work aims to address these limitations.

Chapter 5

TCP Urel: A TCP Option for Unreliable Data Streaming

DMSCC regulates the combined throughput of multiple DMS flows by adjusting the increasing factors of individual flows, which in turn are controlled by TCP-like AIMD loops; yet, TCP, due to the possible unlimited delay from automatic retransmission, is not suitable for multimedia streaming. This chapter introduces a technique to retain AIMD and at the same time remove retransmission for TCP. This technique, TCP Urel, is not only useful to distributed media streaming system, but is also useful to a much broader range of multimedia streaming applications.

5.1 Introduction

As broadband services penetrate into residential places, multimedia applications such as video on demand, on-line game, and video conferences become more and more popular over the Internet. One recent successful example is *YouTube, Inc.* When the multimedia applications make Internet versatile, many of them, however, lack disciplines on bandwidth usage. For instance, Mena et al. studied the RealAudio traffic from a popular Internet audio server and found that RealAudio

traffic shows behavior that is not TCP-friendly [65]. Nichols et al. discovered that Windows Streaming Media is not TCP-friendly [72]. The irresponsible bandwidth usage may harm the stability of the Internet, since the end-to-end congestion control of TCP is an important reason that keeps the Internet from congestion collapse [25]. Therefore a TCP-friendly congestion control protocol for multimedia streaming applications is needed.

Although TCP is regarded as the key stone of keeping Internet stability thanks to its *Additive Increasing Multiplicative Decreasing* (AIMD) congestion control algorithm, it is not directly applicable to the above mentioned applications. One reason is that, these applications may not need reliable data delivery. For instance, packet loss may lower the quality of a Fine Granularity Scalable (FGS) video, which, however, may still outperform a video with prefect picture but frequently stalls. The other reason is that Automatic Repeat reQuest (ARQ) of TCP is not suitable for multimedia applications, since the persistent retransmission may produce unbounded delay on a segment. The segment may have been useless at the time it is being retransmitted, e.g., frames in the future no longer depends on this segment. The urgent need of congestion control in multimedia streaming together with the drawbacks of applying existing TCP suggests that a congestion-controlled but unreliable protocol is needed.

Existing solutions develop their congestion control algorithms on top of UDP [21] or in the transport layer [41]. Both of the solutions have certain shortcoming (Section 5.2). Instead of developing a protocol from scratch, our solution extends TCP for unreliable but congestion controlled streaming. The core idea is to send fresh data in each TCP segment, even in segments that are originally generated for retransmission. This strategy does not recover packet loss. In this way, our solution saves bandwidth from retransmission and uses it to deliver fresh data. It also avoids unbounded retransmission delay in original TCP protocols.

We designed a new TCP option, TCP *Urel*, short for UnRELiable data streaming. When the option is on, segment sending and receiving procedures are modified such that the payload of retransmission segments are replaced with fresh data but the other dynamics in TCP remain the same. This study presents both the design and the implementation of TCP Urel on FreeBSD 5.4. By comparing its throughput with the one of TCP, we found that it is TCP-friendly to different versions of TCP. Through counting the CPU cycles on acknowledging and sending TCP Urel segments, we show that TCP Urel is computationally efficient. We observe the segments that carry no useful data, and reveal that TCP Urel utilizes bandwidth almost fully.

The structure of this chapter is as follow. In Section 5.2, related work is presented and the motivation to design TCP Urel is explained with more details. In Section 5.3, we describe the design and implementation details of TCP Urel. Section 5.4 presents our experiment results. Section 5.5 concludes the chapter.

5.2 Related Work and Motivation

Congestion control could be implemented at the application layer, on top of UDP or raw IP [21, 59]. Application layer resides in the user space in current operating system (such as FreeBSD). Operation of protocols that reside in the user space involves frequent switches between user and kernel in operating systems. Due to the high overhead of kernel-user switch, (i) high speed data transmission becomes expensive in terms of CPU time, and (ii) scalability becomes a problem at servers with large number of concurrent connections. Even after carefully exploiting a set of low level interfaces in order to improve the efficiency of the application layer protocol, Edwards and Muir [21] still reported that their protocol is slower than the kernel version. Due to these reasons, we believe that congestion control should

be realized in the transport layer, which resides in the kernel space.

Datagram Congestion Control Protocol (DCCP) [41] is the recent effort to design a set of congestion-controlled protocols for unreliable data delivery in the transport layer. Currently it defines two profiles CCID2 and CCID3. CCID3 controls flow rate using TFRC [34] and is designed for streaming that requires smooth rate. CCID2, using the TCP-like AIMD algorithm, is designed to achieve as much bandwidth as possible while being TCP-friendly. TCP Urel is comparable to CCID2 for the similarity on their goals and algorithms. While DCCP has been under standardization for years, CCID2 still does not have an usable implementation. To the recent date (March 21, 2007), both the implementations of CCID2 in FreeBSD KAME [1] project and in Linux [2] are still under development. The other potential problem for CCID2 is that, it is designed to be friendly to TCP Sack: *in the long run, the throughput of CCID2 equals to a competing TCP Sack flow with the same network parameters (RTT, packet size and loss rate)* [92]. But TCP Sack does not behave the same as other TCP versions [22]. For example, it acquires a higher throughput than TCP Reno in burst loss [8]. While CCID2 may be sufficient for Internet dominated by TCP Sack, it may not be suitable for networks where other TCP variations dominate. For instance, Linux by default uses TCP BIC [98], which is more aggressive than TCP with standard AIMD algorithm. The problem could become worse in the future when TCP evolves. Due to the complexity resulted from the different functions of sequence number in DCCP and TCP [41], modifying CCID2 to achieve TCP-friendliness to a new version of TCP could be much more difficult than evolving TCP itself.

With the above considerations, extending TCP becomes an attractive choice. The advantage of extending TCP instead of building a protocol from scratch is

[1] http://www.kame.net

[2] http://linux-net.osdl.org/index.php/DCCP

that: (i) it is in the kernel space, and is therefore highly efficient; (ii) it reuses the congestion control module of TCP entirely, hence is friendly to TCP traffic; (iii) by not modifying other TCP functions such as connection establishment, tearing down and security, the new protocol inherits all the features from the well-studied, developed and deployed TCP protocol; (iv) when TCP changes, the unreliable protocol changes accordingly without hefty modifications; (v) inheriting the API of TCP, the new protocol is easy to use. The modification on TCP is supported by the extensible option field. A new functionality that is triggered by a new option, could be inserted without much interference over other functionalities.

One can argue that TCP's AIMD is not enough for applications that requires smooth bandwidth adaptation. That is true but the penalty of a smoother throughput is a slower rate adaptation. TCP-like congestion control allows application to exploit bandwidth by quickly adapting to the network changes. Smoothness should only be considered when the application really has the requirement. Otherwise, TCP-like congestion control should be used [34]. For instance, allowing receiver-side buffering, video streaming could use TCP-like congestion control and gain higher throughput [40]. As a matter of fact, TCP is widely used in commercial streaming system. Both Real Media and Windows Media support TCP streaming. A recent measurement study found that 72% of on-demand and 75% live streaming traffic use TCP [90].

Another argument for not using TCP to stream multimedia, is that TCP is a stream-oriented protocol. Application data that previously layers on top of UDP may not be easily layered on top of TCP, as the application data unit (ADU) may not fit well into the TCP segment, the size of which is determined by the current congestion window. Specifically we list two concerns here, and our explanations are followed.

(i) *If the ADU is smaller than TCP segment size, it has to wait and may suffer from extra delay.* There are two cases. If the sending rate is high, the delay caused by accumulating a segment is short. Compared to the delay at the receiving buffer in most applications, the delay to accumulate a segment (1.5KByte in Ethernet) is negligible. If the data rate is low and the application cannot wait to accumulate for a full size segment, TCP provides the PUSH flag to send the message as soon as congestion control allows.

(ii) *The boundaries of ADU may not match with the boundaries of message in a stream oriented protocol.* This mismatch may lead to the following concerns. First, for TCP Urel, ADU boundary may not be identified as easily as in UDP packets (assuming variable UDP packets), but RTP framing over TCP [45] solves this problem easily. Second, one message loss in TCP Urel could jeopardize many ADUs, but it is the same for fixed size UDP packets, which also carries data from different ADUs. Although variable size UDP packets do not have this problem, it incurs other drawback: the overhead for UDP packet header increase when the ADU becomes small. We regard the comparison of applying TCP Urel or variable size UDP for media streaming as an open problem, which cannot diminish the usability of TCP Urel. Third, ADU loss can be detected when a UDP (variable size, one ADU in one UDP packet) packet is lost. By outputting meta data, however, a stream oriented protocol is also able to inform the application about which part of the data is lost.

TCP RC [63] is a TCP modification for multimedia streaming. It changes the receiving procedure, so that all segments are acknowledged even when they are lost. As a result, the sender never retransmits. The effective reduction on end-to-end delay of TCP RC shows the merit of no retransmission. But ignoring packet loss breaches congestion control loops of TCP – without the knowledge of packet loss,

the sender cannot react to congestion, making TCP RC unfriendly to standard TCP. Our work provides a safe way to remove retransmission without jeopardizing congestion control.

TCP Urel is different from TCP Trunking [43] although the later also provides TCP-friendly congestion control to unreliable flows. TCP Trunking relies on congestion control instructions from a TCP flow that is streamed side-by-side with the UDP flow. But TCP Urel only streams one flow and performs congestion control by itself.

The purpose of TCP Urel is very similar to that of MTP [15]. The principle of transmitting fresh data instead of retransmission are the same. But TCP Urel adopts a Urel sequence aside from the original TCP sequence number. The existence of two different sequence numbers allows clearer separation of retransmission from other TCP functionalities (i.e. congestion control), therefore allows a very simple design and implementation. Unlike MTP, which is implemented in ns-2 simulation, TCP Urel is implemented over real TCP stack with very little code.

TCP Urel realizes the part of unreliable data streaming functionality in previous partial order service (POS) in TCP extension [17]. Unlike POS, which provides partial reliability, TCP Urel leaves retransmission to the application. This decision is based on the observation that current media streaming applications varies on whether, when and from where a packet loss could be recovered. Leaving error recovery to the application entitles more flexibility to the application implementer, and greatly reduces the complexity of designing and implementing TCP Urel.

The idea of TCP Urel is related to the work of late data choice (LDC) [44]. In LDC, the authors proposed to *choose which data to send immediately before transmission*, with the aim of reducing the delay cost from buffering. Realizing the fact that the payload of a TCP segment could be decided right before it is sent out, we are inspired that the congestion control, the connection maintenance, and

other functionalities of TCP protocol can actually be separated from the data it is delivering. The key is to detach the correspondence between the TCP sequence number and the segment data. Though related to LDC, TCP Urel is different both at the sender and at the receiver. LDC focuses on API, through which user data is streamed with minimum delay; whereas TCP Urel focuses on detaching retransmission from congestion control. Details of TCP Urel will be shown in the next section.

5.3 Design of TCP Urel

5.3.1 The Overall Idea

TCP Urel is realized by defining the Urel option in the existing TCP protocol. When this Urel option is on, TCP changes its sending (output) and receiving (input) procedures such that, (i) while sending a segment, fresh data is always delivered, even if the segment is a retransmission segment, and (ii) upon receiving any segment, payload is handed to the application in the same order as they are sent.

Figure 5.1 depicts this idea. AIMD algorithm at the sender decides the size of the next segment to send. But the payload in the segment could be refilled right before sending out the segment. Once the segment arrives at the receiver, the payload is immediately buffered, reassembled and submitted to socket buffer. Information extracted from the TCP header is used to generate acknowledgment, which is crucial to retain the dynamics of TCP flow in the original form (congestion control, flow control, etc.).

Before explaining the details of TCP Urel, we introduce some variables in the FreeBSD TCP code. These variables are used for congestion control in original TCP protocol, but behave slightly different in TCP Urel. We first explain how

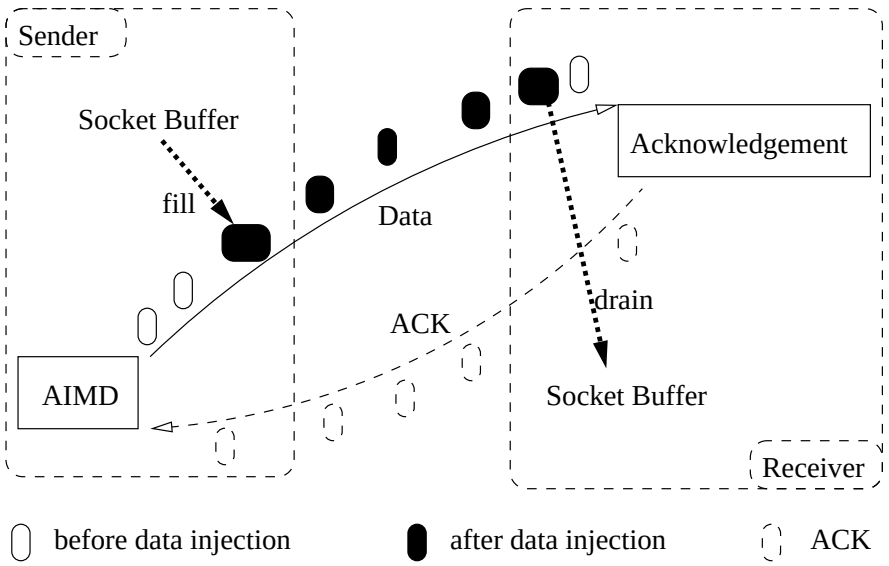


Figure 5.1: Overall idea of TCP Urel

these variables are used in original TCP.

(i) *snd_una* is the TCP sequence number of the lowest TCP segment that is not acknowledged. The segments before *snd_una* are all acknowledged, and can be discarded at the TCP sender.

(ii) *snd_max* is the highest TCP sequence number that has been sent in the current session. The segments between *snd_una* and *snd_max* may (if Sack enabled) or may not have been acknowledged. Those unacknowledged are used to estimate the amount of data that TCP has left in the network (on-fly data). This amount is compared with the size of congestion window, in order to determine the size of the next segment that can be sent [95]. Therefore, *snd_una* and *snd_max* are important to the dynamics of congestion control.

(iii) *snd_nxt* is updated by the congestion control algorithm. In sender's socket buffer, the byte pointed to by *snd_nxt* begins the next segment to be sent. TCP segments carry this sequence number in the TCP header. Upon receiving the segment, the receiver determines the acknowledgment sequence

number by this sequence number plus the size of the payload. Since acknowledgment affects the sender's behavior, *snd_nxt* is vital to the dynamics of TCP.

In TCP Urel some operations related to these variables are changed, which are shown in the rest of this section.

5.3.2 Sending Procedure

The basic idea of TCP Urel at the sender is that, the AIMD algorithm still decides the size of segment to send, but fresh data is always filled into the payload right before the segment is sent out. When there is no packet loss, TCP always fill fresh data into the segment. In this case, Urel option does not make a difference. But when retransmission occurs, TCP resend old data. TCP Urel *replaces the old data in the segment with the same amount of fresh data* and sends them out. As a result, to implement TCP Urel, we only need to modify the behavior of TCP sender on the path of retransmission. The shadow area in Figure 5.2 shows these modifications.

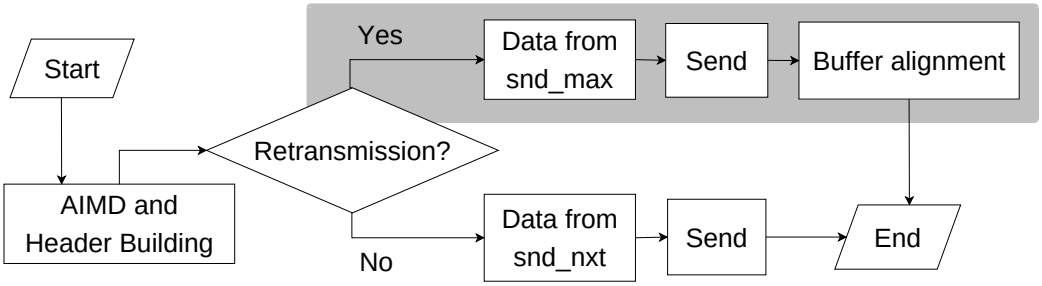


Figure 5.2: Sender modifications on the retransmission path

One key difference between the path of standard TCP sender and TCP Urel sender is that, the former always sends data pointed by *snd_nxt*, whereas the latter always sends data pointed by *snd_max*. *snd_nxt* points to the data that TCP is

going to send, whereas *snd_max* is the highest sequence number of segments TCP has ever sent. In retransmission, *snd_nxt* lags behind *snd_max*. By sending the data pointed by *snd_max*, TCP Urel delivers fresh data in every segment.

While the above modification seems straightforward, it is actually error-prone, due to the important role of *snd_max* in TCP's congestion control. Since *snd_max* is related to the calculation of in-flight data, changing *snd_max* can eventually change the size of next segment and therefore the sending rate. Due to this reason, TCP Urel should not to modify the value of *snd_max*, or else, the congestion control mechanism will be affected. In other words, after sending data pointed to by *snd_max*, TCP Urel is not allowed to increase *snd_max* to the new position. But contrarily, *snd_max* by definition, is supposed to point to the data with the highest sequence number that has ever been sent, otherwise TCP Urel cannot send fresh data in the next segment. To solve this problem, *buffer alignment* is needed in the sending path of TCP Urel.

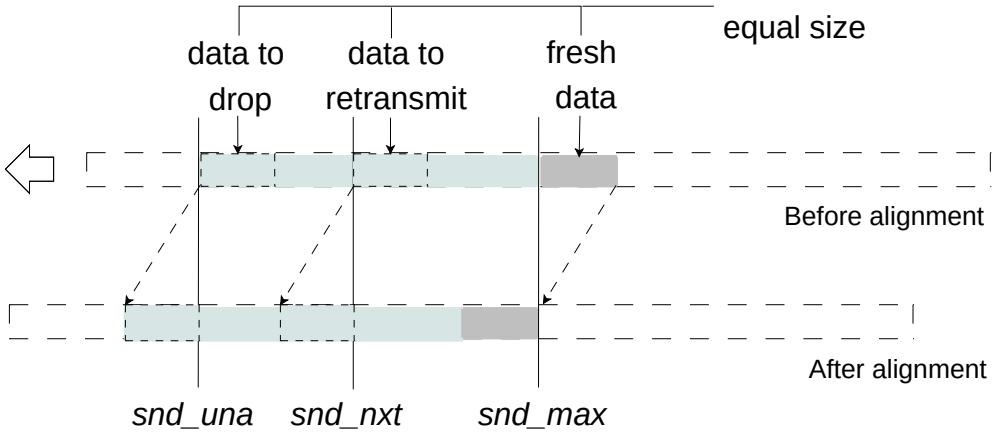


Figure 5.3: Before and after buffer alignment

Buffer alignment is performed by dropping certain amount of data in the socket buffer. To TCP, socket buffer looks like a tape. Sequence numbers *snd_una*, *snd_nxt* and *snd_max* are cursors on the tape. Since TCP Urel cannot move *snd_max*, in order to match *snd_max* with the newest data on the tape, TCP Urel moves the

tape (see Figure 5.3). Before buffer alignment, the payload size has been decided. The original TCP intends to retransmit the data that is pointed by *snd_nxt*. But in TCP Urel, it is replaced by the same amount of fresh data. Once sent, these data are not fresh any more. To point *snd_max* to the next chunk of fresh data, the whole tape is moved leftwards, by dropping the chunk of old data pointed by *snd_una*.

In FreeBSD 5.4, buffer alignment is realized by calling sbdrop(), which drops data in the head of the tape with the size of the previous sent segment. After buffer alignment, although the value of *snd_una* and *snd_nxt* remain the same, they do not point to the previous data any more. But this is acceptable for TCP Urel, since (i) the value of *snd_una*, instead of the corresponding data, affects the behavior of congestion control; (ii) the value of *snd_nxt*, instead of its corresponding data, affects the acknowledgment; and (iii) data pointed by *snd_una* and *snd_nxt* has been sent. Once the data is sent, it becomes useless in TCP Urel.

The merits of buffer alignment are that, (i) values of *snd_una* and *snd_max* are not changed, therefore original congestion control is preserved; (ii) *snd_max* keeps pointing to the fresh data; and (iii) value of *snd_nxt* remains still, so the TCP sequence number in the TCP header is not affected, and hence the connection dynamic is preserved.

5.3.3 The Urel Option

After the description of the sending procedure, one may notice that in TCP Urel, the TCP sequence number dose not correspond to the bytes in the stream. For example, a "retransmission" segment carries an old TCP sequence number, but the data is different from the lost segment. In TCP Urel, the function of TCP sequence number is to retain the protocol dynamics of the original TCP, including

congestion control. Lacking the correspondence to the data byte, TCP sequence number cannot be used for data reassembling at the receiver side. Therefore, a new field indicating *the original position of the bytes in the stream* must be included in the TCP header, as part of the Urel option. We call the new field *data sequence number*. The format of TCP Urel option is depicted in Figure 5.4.

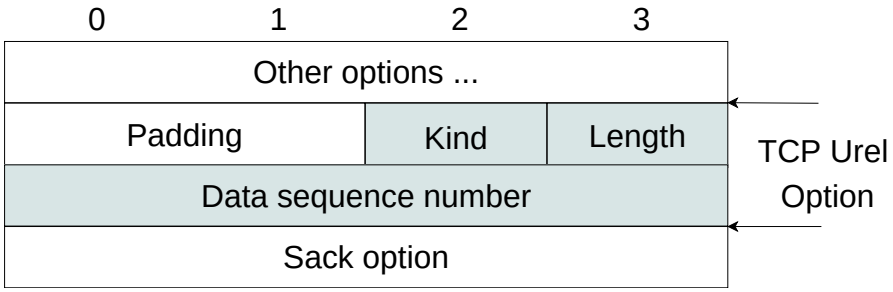


Figure 5.4: The format of Urel option in TCP header

Kind is the option identifier, which is set to 27 for TCP Urel[3]. *Length* equals to 6, meaning Urel option occupy six bytes. Both Kind and Length are one byte long. *Data sequence number* is the data sequence number of the first byte carried in this segment. It is a four-byte sequence number, and starts from zero for the first byte in the first segment. Having the same length as a TCP sequence number, data sequence number allows sufficient space for wrapping around. Data sequence number increases by one on each byte sent. At the receiver side, the payload in the incoming segments are reassembled based on this data sequence number.

Space Concern of Sack Blocks

Adding options to the existing TCP header, however, brings a new implementation issue. In FreeBSD, the option field in the TCP header requires a four-byte alignment. Excluding the Time Stamp option (12 bytes including padding, enabled by

[3]Option number from 0 to 26 are implemented in FreeBSD 5.4, therefore we pick the next: 27. But this option number may change to avoid conflicts to other options under development.

default) [39], there are 28 bytes left for Sack [61] in the 40-byte option field, allowing three Sack blocks. When the Urel option (8 bytes including padding) is added in, the free space for Sack reduces to 20 bytes, allowing only two Sack blocks. In other words, the insertion of Urel option may reduce the effectiveness of Sack option in TCP Urel. This issue, however, can be solved by removing Urel option from the acknowledgements, as Sack blocks only appears in the acknowledgements. The modification changes the Sack-based TCP Urel into a single directional streaming protocol, i.e., it only passes application data from the sender to the receiver. But the single directional streaming model is sufficient for non-interactive streaming, such as VOD. The Urel option does not affects TCP Reno and NewReno, and therefore are bidirectional while adopting congestion control of TCP Reno and NewReno.

5.3.4 Receiver Procedure

As mentioned before, when TCP Urel option is enabled, TCP sequence number does not correspond to the position of bytes in the original data sequence. So we cannot rely on the reassembling function in the original TCP to reorder the data. The idea of TCP Urel at the receiver side is that, after verifying the checksum and extracting the options from the header, TCP Urel buffers the payload before handing it to the original reassemble function. If necessary, TCP Urel needs to reassemble the buffered data or submit them to socket buffer and signal the application for reading. Both buffering and submitting functions require that the original code does not submit data to socket buffer any more. For FreeBSD 5.4, the TCP Urel implementation in TCP input function removes original calls of sbappend() which submits data to socket buffer. Please note that we still need original TCP reassembling based on TCP sequence number, as the queue it maintains plays a

vital role in (selective) acknowledgment, which eventually affects the behavior of congestion control. The receiving path of TCP Urel is shown in Figure 5.5.

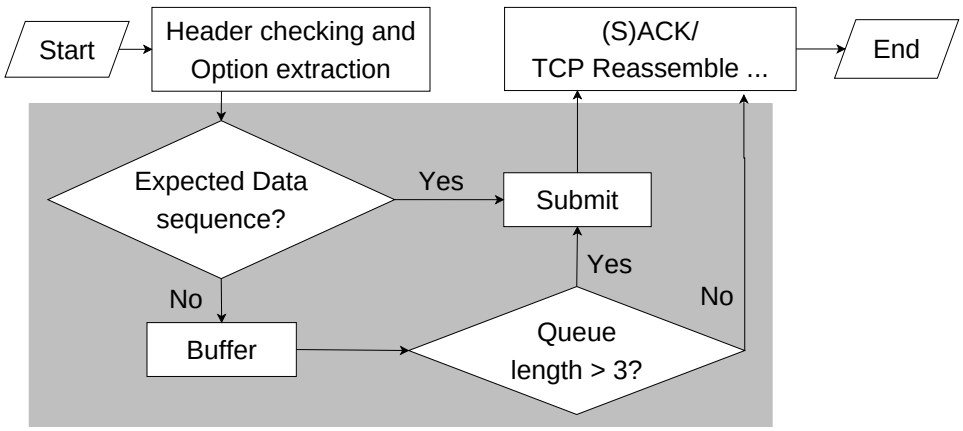


Figure 5.5: Receiver modification before handling packets to (S)ack

In the figure, the shaded area shows the position of buffering and submitting in the receiving path. Expected data sequence number is the sequence number that follows the highest data sequence number of the received bytes. If no loss or packet reordering occurs, every arrival segment has a data sequence number equals to the expected one. The arrival segment is submitted immediately. In case of packet loss or packet reordering in the network, data sequence numbers arrives out of order. Borrowing the idea from TCP reordering threshold, we pick three segments as the reordering threshold. If a discontinuous data sequence is caused by packet reordering, TCP Urel assumes that the continuity could be rebuilt in three segments. If more than three segments arrive with data sequence numbers higher than the expected one, a packet loss is assumed and all the data buffered because of this loss will be submitted. Note that there could be other ways to determine the value of reordering threshold; For example, RR-TCP [99] changes it adaptively to avoid false retransmission. It is, however, out of the consideration of this study.

5.3.5 Urel Negotiation

Urel is an option added to the existing TCP protocol. The TCP stack on a host may or may not support TCP Urel. Therefore, during connection establishment, negotiation is necessary. Urel negotiation is started by the connection initiator. Unlike TCP Sack, we do not define extra option for Urel negotiation, as the Urel option format in Figure 5.4 is sufficient for the purpose. The procedure of negotiation is shown in Figure 5.6. The connection initiator sends a SYNC message with Urel option enabled, and transits into *Urel wait* state. If the other side is able to work under Urel mode, it replies with a SYNC/ACK message with Urel option and transits into *Urel mode*. Upon receiving this message, the initiator knows that the other side supports Urel, and transits into the *Urel mode*. Data streaming can then start. If the other side does not support Urel, it will simply ignore the Urel option in the first SYNC message, and reply with a SYNC/ACK without Urel option. The initiator then finds that TCP Urel is not supported on the other side, and gives up the attempt to use TCP Urel. The data sequence number during the negotiation is ignored, as the size of the payload in the segments is zero and no data is delivered.

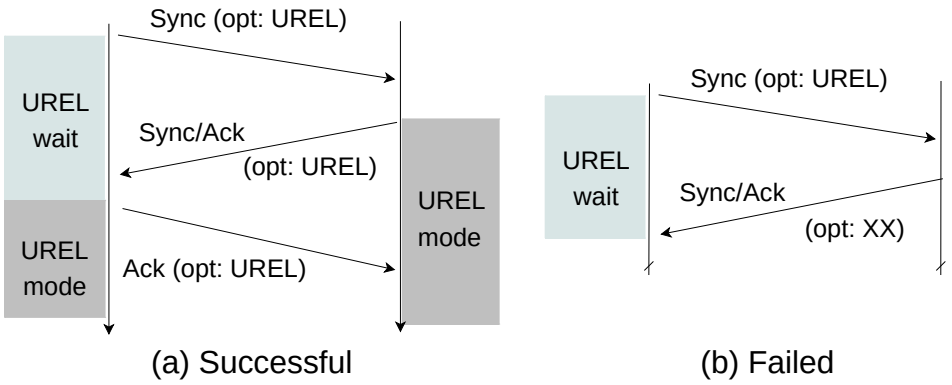


Figure 5.6: Urel Negotiation

```
1 ...
2 int sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)
3 int urel = 1;
4 setsockopt(sock, IPPROTO_TCP, TCP_UREL, (char*) &urel, sizeof(urel));
5 ...
6 bind(sock, (struct sockaddr*) &saddr, sizeof(saddr));
7 listen(sock, 5);
8 ...
```

Figure 5.7: Source code from a simple server using TCP Urel

5.3.6 Application Programming Interface

As an option of TCP protocol, TCP Urel is easy to use. By adding a few lines to a normal TCP program, we can set up a TCP Urel session. Figure 5.7 lists code that set up a socket, and start listening to it. The code is similar to the normal way a listening socket is set up using TCP. The only difference is line 3 and 4, where the TCP Urel option is turned on, enabling unreliable streaming. Other APIs such as listen(), connect(), accept(), and send() work the same way as before.

5.3.7 Possibility of Bandwidth Wastage

TCP Urel utilizes every segment to transfer fresh data, therefore its data consumption rate at the sender is faster than existing TCP versions. An application is responsible for providing enough data in socket buffer, so that the sending rate does not diminish due to exhaustion of data.

Nevertheless, no matter how large the socket buffer is, there is a possibility that when TCP Urel is about to use a "retransmission" segment, the cursor *snd_max*

reaches the upper bound of the socket buffer. In this situation, the unmodified TCP part will generate a retransmission segment for transmission. This segment with this particular TCP sequence number must be sent in order to remain the action of original TCP, including the dynamics of acknowledgment and TCP reassembling at the receiver side. However since *snd_max* is at the upper-bound of the socket buffer, there is no fresh data to be filled into the payload. In this case, we perform a retransmission as original TCP: this segment does not carry a Urel option nor a data sequence number, and it provides no useful data to the receiver. Detecting no Urel option in the TCP header, the receiver's Urel code (shaded area in Figure 5.5) skips this segment. The segment then goes through the remaining part of the receiver's procedure to maintain the dynamics of TCP, which relies on the TCP sequence number of the segment. But TCP Urel does not submit data from this segment to the application. Transmission in the above case leads to bandwidth wastage. But we will show in Section 5.4.3 that the amount of wastage is negligible.

5.3.8 Support for Partial Reliability

While most video/audio real-time streaming applications do not require reliability, some applications may require partial reliability. For instance the distributed streaming system described in previous chapter requires the freedom of selective retransmission. Although TCP Urel itself does not provide selective retransmission, it is designed to provide support to retransmission if needed. More specifically, TCP Urel informs the receiving application the position of missing bytes when segment loss occurs. Whether to retransmit, and how to retransmit is decided by the application. For instance, the receiver may send retransmission request in a UDP packet. Upon receiving the request, the sender refill the lost data into the socket buffer of TCP Urel for transmission, or it may send it in a UDP reply

packet. The difference is that the former strategy keeps strict TCP friendliness by using TCP Urel, whereas the UDP packets in the later method consume extra bandwidth. TCP Urel does not specify which strategy an application should adopt.

The advantages of supporting partial reliability but do not provide it directly in TCP Urel are that, (i) it keeps TCP Urel simple, (ii) it gives the freedom of retransmission to the application. This freedom allows the application to decide whether to retransmit. Besides, it also allows the application to decides how to retransmit, as described in last chapter. Compared to protocols that provide partial reliability (e.g. PR-SCTP [87]) on single path, the simple support from TCP Urel may be more flexible and suits larger variety of applications.

The support to partial reliability is realized by adding meta data into the submitted stream at the receiver side. The data received by the application is specially formatted. The byte stream is divided into chunks, each of which has a fixed formatted chunk header that indicates the reliability and the length of the data in this chunk. The rest of the chunk contains byte stream that is received from the sender. The format of a chunk is shown in Figure 5.8.

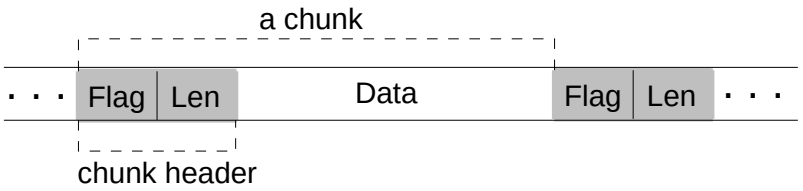


Figure 5.8: Chunk format of the received stream

The 2-byte *Flag* field describes the reliability of the data in the chunk. Currently only the first bit in *Flag* is used. If a segment is received from the network with a continuous data sequence number, the first bit is set to 1. If a segment is lost, a gap in data sequence number will be detected by TCP Urel. When submit-

ting data to socket buffer, TCP Urel sets the first bit in *Flag* of the chunk header to 0. Then it allocates an all-zero byte stream to fill up the gap in data sequence, and submit the stream to socket buffer. Other bits in *Flag* are reserved for future extension.

The 2-byte *Len* field stores the length of the data in the chunk. By reading this field, an application is able to process the data chunk by chunk.

There are two points to be noted. First, a chunk does not have any semantical meaning to the application. The purpose of inserting chunk header in the byte stream is to allow the application to know the location of lost bytes. Problems such as whether these bytes are important, whether retransmission of these bytes is needed, and how to retransmit them, etc. are decided by the application. Second, a chunk header is inserted by TCP Urel at the receiver side, before the byte stream is handed to the application. Therefore, the chunk header does not consume extra bandwidth. This insertion of chunk header does cost some CPU time overhead, but it is small and acceptable (Section 5.4.2).

5.4 Evaluation

In this section, we will evaluate TCP Urel in three aspects. First, we show that TCP Urel is friendly to difference versions of existing TCP. Second, we show that TCP Urel is highly efficient. Comparing to existing TCP protocol, running Urel option only increases CPU cycle by a constant number, regardless of the increasing loss rate. Third, we will show that the bandwidth wastage described in Section 5.3.7 is negligible.

We implemented TCP Urel on FreeBSD 5.4. The source code added into the original kernel is less than 750 lines, among which about 151 are debugging code or preprocessor directives that can be further trimmed.

The test-bed used for evaluation is shown in Figure 5.9. The prefix of the names of the end hosts, tcp, urel and dccp, denotes the type of flows between the host pairs and the suffix s and r represents sender and receiver respectively. For simplicity, in our emulation, a particular type of flow is run on the host pair named after the flow. For example, a TCP Urel flow is the flow sent from urels to urelr. Host phoebe is a FreeBSD 5.4 box with dummynet[4] enabled to emulate a bottleneck, whose configuration parameters will be reported along with experiment results.

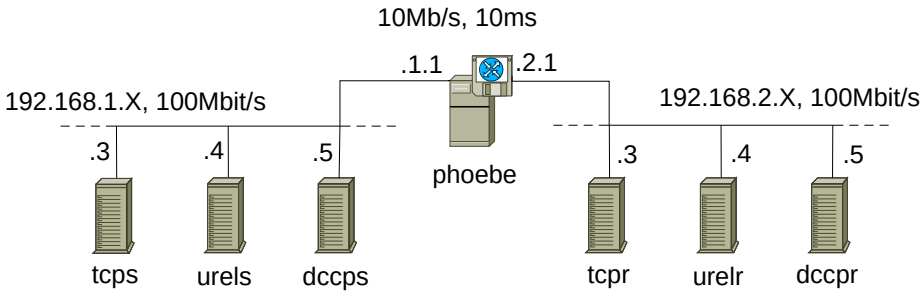


Figure 5.9: The illustration of the test-bed

5.4.1 TCP Friendliness

To show the TCP friendliness of TCP Urel, a comparison between throughput of TCP Urel, DCCP CCID2[5] and other TCP flows would be persuasive. To the date of this writing, however, there is still no usable CCID2 implementation. CCID2 implementation on Linux and in FreeBSD KAME tree is under development. The available code by Lulea University of Technology [6] is able to run, but TCP friendliness is not provided and we believe the code is still immature. Therefore, to evaluate the TCP friendliness of CCID2, we borrowed data from Takeuchi's work [89],

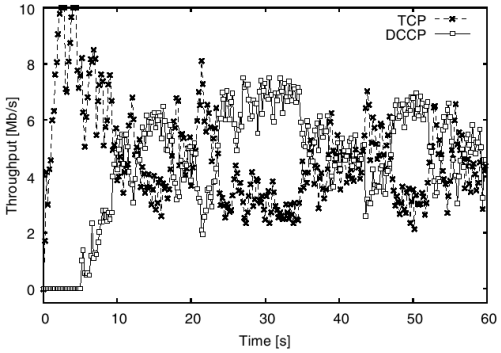
[4]http://info.iet.unipi.it/∼luigi/ip_dummynet/

[5]DCCP CCID3 adopts TFRC for congestion control, hence is less comparable to TCP Urel which uses AIMD for congestion control.

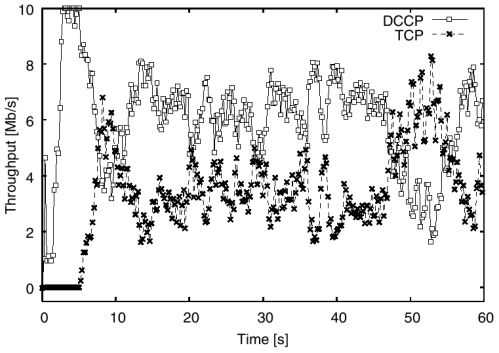
[6]Source code and patch for FreeBSD 6.1 available at: http://mobqos.ee.unsw.edu.au/∼lochin/

which evaluates CCID2 using ns-2 simulation.

TCP Friendliness of DCCP CCID2



(a) TCP Sack before DCCP CCID2



(b) DCCP CCID2 before TCP Sack

		TCP	DCCP	Total
Reno	TCP first	4.73	4.66	9.37
	DCCP first	3.75	5.79	9.54
Newreno	TCP first	4.84	4.50	9.30
	DCCP first	4.10	5.28	9.38
Sack	TCP first	4.42	4.96	9.38
	DCCP first	3.69	5.97	9.66

(c) Summary of stationary throughput (Mb/s), TCP and CCID2

Figure 5.10: TCP friendliness of DCCP CCID2, from Takeuchi et al.'s work [89]

The topology of simulation by Takeuchi et al. [89] is the same as in Figure 5.9. The bottleneck has a bandwidth of 10Mb/s, and a droptail queue with length 20 packets. Propagation delay are 10ms on the bottleneck link[7], 3ms between the senders and phoebe, and 2ms between phoebe and the receivers.

The plots and table in Figure 5.10 are directly borrowed from their work. For comparison, we will show similar figures plotted from experiments on TCP Urel

[7]The bottleneck is a link in Takeuchi et al.'s ns-2 simulation. In our emulation, delay is produced on phoebe using dummynet.

from our emulation (Figure 5.11 and Figure 5.12).

Once DCCP CCID2 flow starts 5 seconds after/before a TCP flow. From Figure 5.10(a) and Figure 5.10(b), it shows that, CCID2 consumes more bandwidth than TCP Sack flow. The stationary throughput listed in Figure 5.10(c) confirm this observation: when CCID2 starts first it consumes more bandwidth than TCP flows, no matter which version of TCP it is competing with; and even when TCP starts first, CCID2 still grabs bandwidth from TCP Sack.

TCP Friendliness of TCP Urel

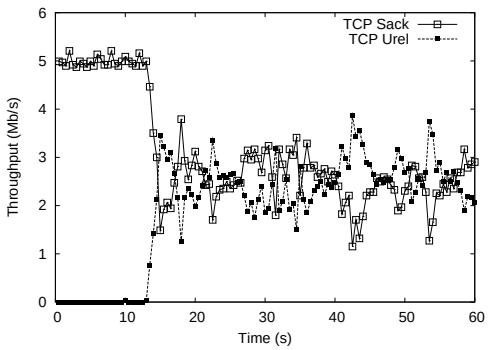
On the test-bed described in Figure 5.9, we set the bandwidth, the propagation delays, and the queue length exactly the same as the ns-2 simulation by Takeuchi et al. [89]. Three points are different and should be explained before we presenting the results.

(i) We start one flow 10 seconds (instead of 5 seconds) after the other, to produce two flows that are apart in time for readability of the graph. Each flow lasts for 60 seconds.

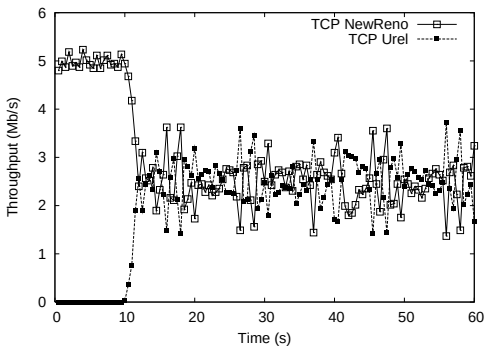
(ii) Although the propagation delay are set according to the ns-2 simulation, the actual RTT experienced by the flows in the emulated environment are different from those in the simulation. In our emulation, we observe average RTT varying from 48ms to 58ms, which produces a smaller total throughput than previous ns-2 simulations. The reason might be the subtle differences between an emulated network and a ns-2 simulation, e.g. the clock granularity of dummynet might cause extra delay as well. Producing smaller throughput, though, the emulation still shows the TCP-friendliness of TCP Urel to corresponding TCP flows in the same emulated network.

(iii) We not only carry out the emulation in droptail queue, but also in RED

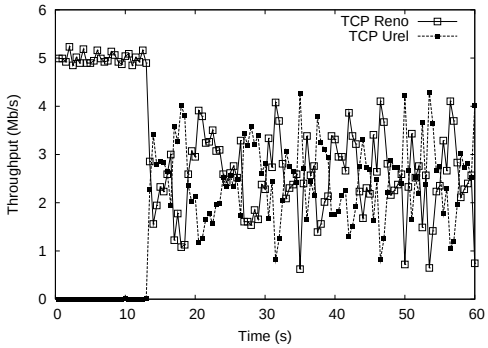
queue[8]. We present results for both queuing discipline.



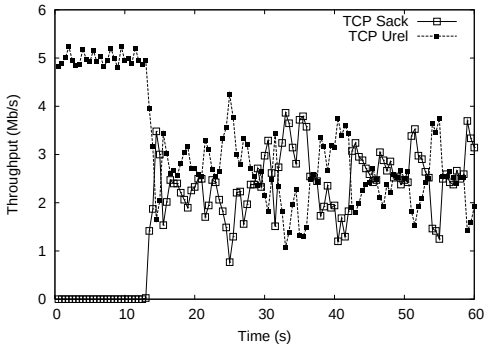
(a) TCP Sack before TCP Urel



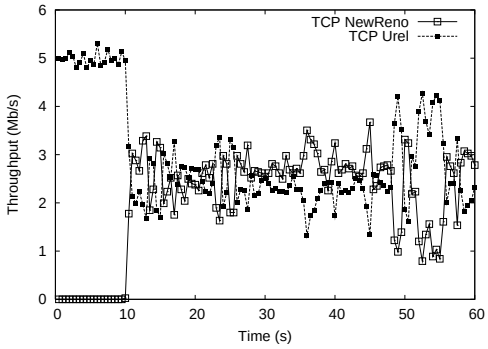
(b) TCP NewReno before TCP Urel



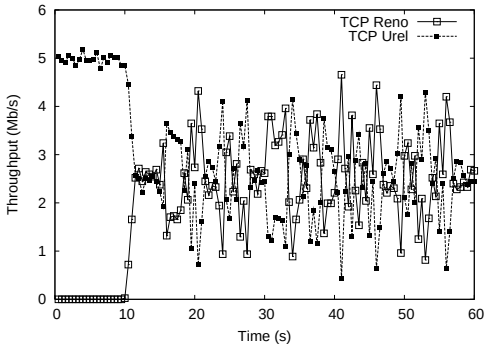
(c) TCP Reno before TCP Urel



(d) TCP Urel before TCP Sack



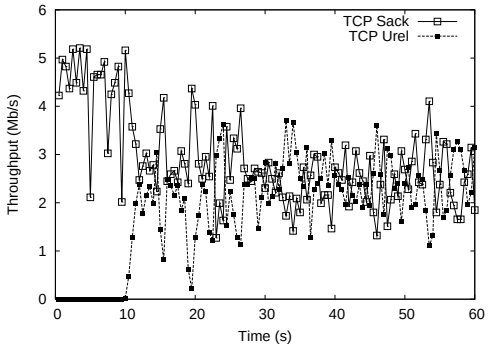
(e) TCP Urel before TCP NewReno



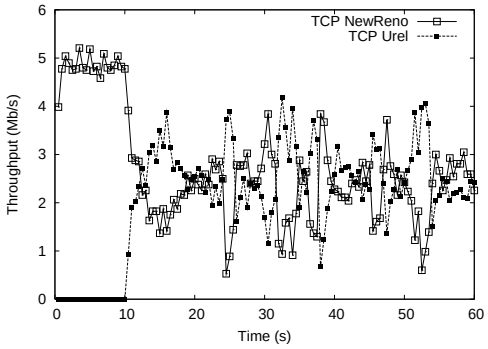
(f) TCP Urel before TCP Reno

Figure 5.11: Throughput convergence of TCP Urel and other TCP flows, in a droptail queue

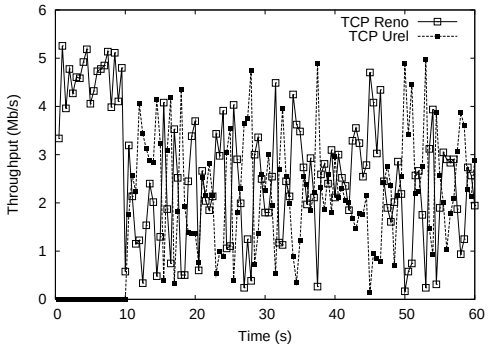
[8]RED parameters: queue weight 0.002, minimum threshold 5, maximum threshold 20, and maximum dropping probability 0.1. They are set according to *RED: Discussions of Setting Parameters*, by Sally Floyd, from a November 1997 email message. Available at http://www.icir.org/floyd/REDparameters.txt



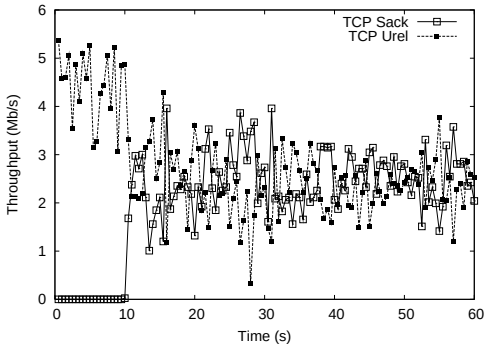
(a) TCP Sack before TCP Urel



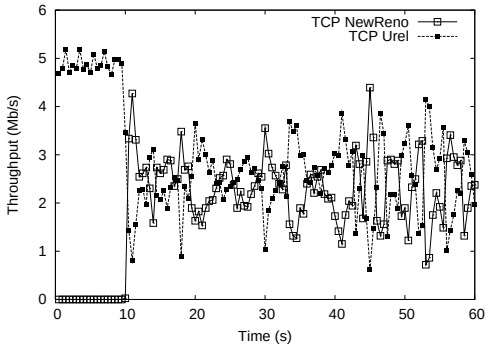
(b) TCP NewReno before TCP Urel



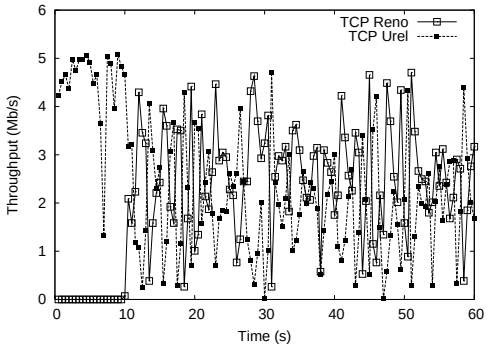
(c) TCP Reno before TCP Urel



(d) TCP Urel before TCP Sack



(e) TCP Urel before TCP NewReno



(f) TCP Urel before TCP Reno

Figure 5.12: Throughput convergence of TCP Urel and other TCP flows, in a RED queue

Figure 5.11 shows the competition between TCP Urel and different TCP flows in a droptail bottleneck. Figure 5.11 (a) and (d) show the throughput of TCP Urel in competition with TCP Sack when TCP Sack starts before (a) or after (d) TCP Urel. (b), (e) and (c), (f) are similar presentations for TCP Urel against

TCP NewReno and TCP Reno respectively. In every figure, TCP Urel runs on the same TCP congestion control scheme as the TCP version it is competing with. When competing with TCP Sack, we use TCP Urel with Sack option enabled; when competing with TCP NewReno, we use TCP Urel with Sack option disabled and NewReno option enabled. Configuration of the TCP versions (and TCP Urel versions) is done via command sysctl in FreeBSD 5.4.

From the plots we can see that by using different versions of congestion control scheme, TCP Urel is able to maintain friendliness to different types of TCP flows, no matter which flow starts first. This friendliness is further confirmed by the average throughput listed in Table 5.1. We collect the average throughput of each flow over 10 runs each. Since each flow-pair share the same competition period (50 seconds) as well as the same length of channel monopoly period (10 seconds each), the throughput should be roughly the same if they are friendly to each other. Comparing to data from Figure 5.10(c), where larger than 30% a throughput difference are observed, the throughput of each flow pairs in Table 5.1 shows good proximity.

Table 5.1: Summary of throughput (Mb/s) in a droptail queue

		TCP	Urel	Total
Reno	TCP first	2.979	3.005	5.984
	Urel first	2.986	2.964	5.950
NewReno	TCP first	2.966	2.878	5.844
	Urel first	2.879	2.997	5.876
Sack	TCP first	2.979	3.033	6.012
	Urel first	2.962	2.999	5.961

Besides droptail, we also tested TCP Urel over a RED queue, with the same bandwidth, delay, and queue length. Similar graphs are generated (Figure 5.12),

Table 5.2: Summary of throughput (Mb/s) in a RED queue

		TCP	Urel	Total
Reno	TCP first	2.629	2.564	5.193
	Urel first	2.559	2.634	5.193
NewReno	TCP first	2.837	2.845	5.682
	Urel first	2.811	2.843	5.654
Sack	TCP first	2.844	2.760	5.604
	Urel first	2.777	2.818	5.595

and the TCP-friendliness of TCP Urel is further confirmed by the average throughput in Table 5.2. Based on the roughly equal throughput of the competing flows, we conclude that TCP Urel retains friendliness to TCP Sack, Reno, and NewReno in both droptail and RED queues.

5.4.2 Protocol Efficiency

Network layer Efficiency

Being an extension based on existing TCP implementation, TCP Urel does have extra cost over original TCP. The following experiment shows the cost of TCP Urel in CPU time. In FreeBSD 5.4, tcp_input() and tcp_output() are the two functions that are changed by TCP Urel. tcp_input() takes care of incoming segments, and tcp_output() is in charge of outgoing segments. When a segment arrives, tcp_input() is called to handle it; after that, tcp_output() is called to send a new segment (e.g., an acknowledgment). tcp_input() is called in ip_input() when IP layer handles the incoming packet. Figure 5.13 shows the above loop. Our method of counting the efficiency of TCP Urel is to count the CPU ticks before and after calling of tcp_input() in ip_input(). The difference of the two ticks are the total CPU cycles

consumed by TCP to process an acknowledgment and to send data, with or without Urel option.

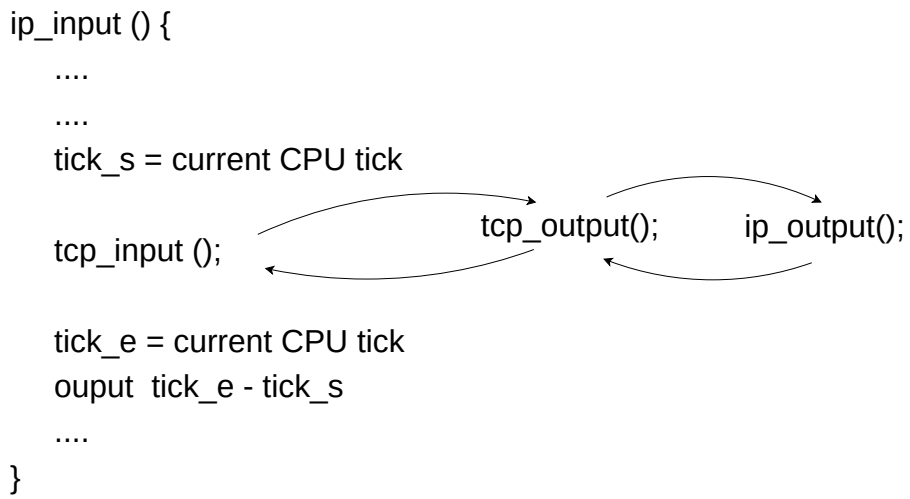


Figure 5.13: The method to measure efficiency

We change the packet loss rate in the bottleneck on phoebe from 0 to 12%, covering loss rate of practical networks. For each packet loss rate, we run a 60 seconds TCP Sack flow or TCP Urel flow. The CPU cycles described above are recorded every time ip_input() calls tcp_input(). Each flow is repeated five times and the average CPU cycle is then calculated and plotted, for both the sender and the receiver.

Figure 5.14 shows the CPU cycle of TCP with or without Urel option at the sender; Figure 5.15, shows the same measurement at the receiver. In both figures, TCP Urel costs more than standard TCP flow, which is expected. We observe the following:

(i) The cost of both protocols at both sides rises as packet loss rate increases, due to handling of retransmission and congestion control.

(ii) At the sender side (Figure 5.14), comparing to TCP Sack, the extra cost of TCP Urel is dominated by tcp_output(), where data in "retransmission"

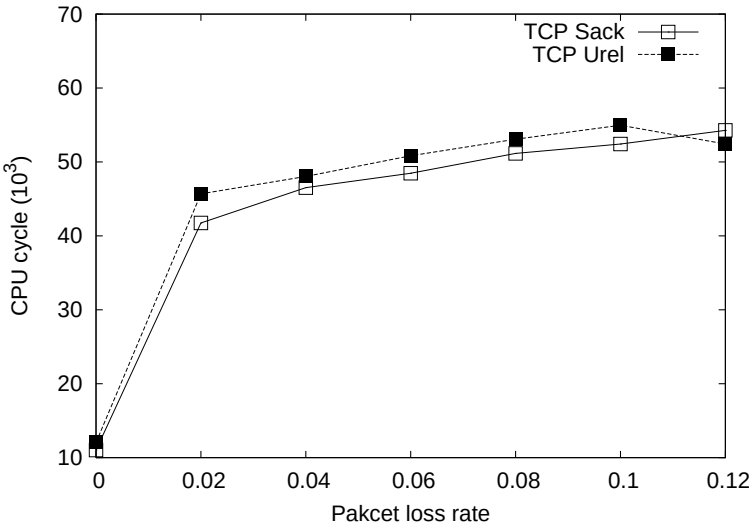


Figure 5.14: Average CPU cycle at the sender side

segments are refilled. But the overhead at TCP Urel's sender is negligible compared to TCP Sack.

(iii) At the receiver side (Figure 5.15), the overhead of TCP Urel is caused by insertion of meta data into every segment. By comparing the curves of TCP Urel and TCP Sack, however, we believe that the extra cost is constant regardless of the increase of packet loss rate. This property indicates that regardless the network congestion, the scalability of TCP Urel is comparable to TCP Sack.

It would be more persuasive if similar cost from DCCP CCID2 could be plotted. But because of the immaturity of the code, we believe measurement on CCID2 could be unfair and the result would be misleading. In our primary test on Lulea's CCID2 implementation the sender side gives a much higher (2∼9 times) cost than TCP Urel in different packet loss rate. The cost at the receiver side changes wildly and is not evaluative to us.

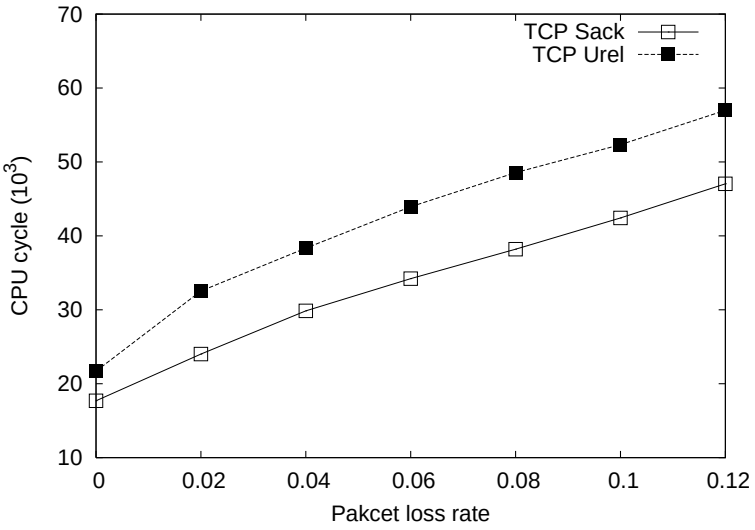


Figure 5.15: Average CPU cycle at the receiver side

Application Layer Efficiency

TCP Urel inserts meta data after receiving a packet; and we have shown (Fig 5.15) the overhead for this insertion is constant and acceptable. The application now need to remove meta data for every packet. To measure the overhead of this removal in application layer, we record the CPU clock ticks used for searching and removing meta data. We streamed a 60-second session for 10 times, and recorded 405351 buffer reading at the receiver; among which, 405305 reads just one packet (i.e. the buffer length equals to the payload length plus the meta data length). Since the reading of single packet buffer dominates, we study the overhead of such needs.

In Figure 5.16, we count the CPU ticks spent on removing their meta data for the first 10000 packets. The ticks show the same pattern in the remaining 395305 packets. The average ticks for the 405305 packets is 297.61, i.e., meta data removal costs an overhead of about 297 CPU ticks per packet. This time is roughly between 0.88 and 15 microseconds, with a mean of 3 microseconds. The variation depends on process swapping in the CPU.

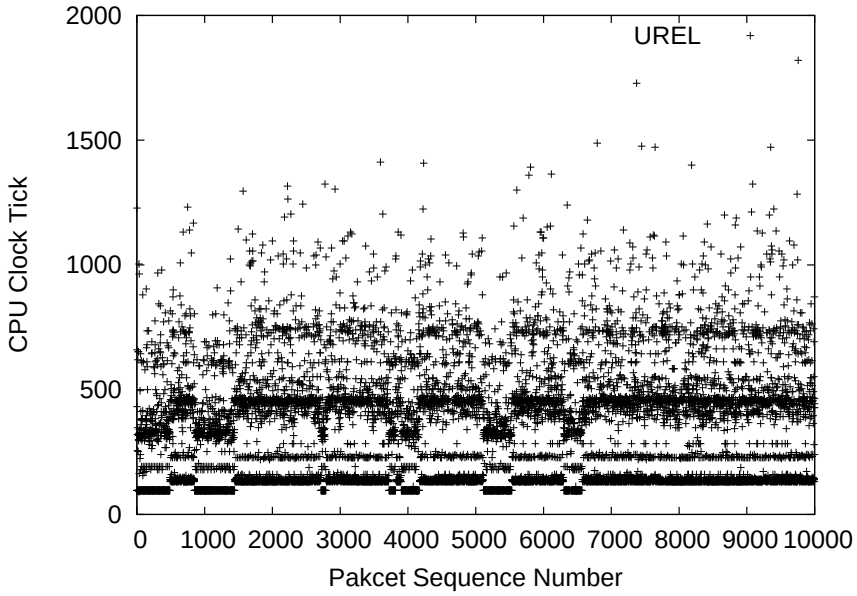


Figure 5.16: CPU ticks to remove meta data in application layer

5.4.3 Bandwidth Wastage

We have discussed the possible bandwidth wastage in TCP Urel in Section 5.3.7. Here we show how small the waste is. Experiment setting is exactly the same as in previous section. But in each streaming session, we count the number of "retransmission" segments and the number of segments that are not refilled with fresh data (thus wasted). The percentage of wasted data out of "retransmission" segments is computed and listed in Table 5.3. The sender side socket buffer is set to 64KBytes. Table 5.3 shows that the percentage of waste is not linearly related to packet loss rate. Waste only happens when socket buffer exhaustion coincides with "retransmission". From the results, we can say that the waste is practically negligible: even with a waste percentage of 1.38%, when the packet loss rate of 4%, the wasted bytes in total data is around 0.05%.

Table 5.3: Percentage of waste bytes in "retransmission" segments

Loss Rate %	2	4	6	8	10	12
Waste %	0.39	1.38	0.68	0.87	0.39	0.77

5.5 Conclusion

In this chapter, we presented TCP Urel, a TCP option for congestion controlled but unreliable streaming. As an extension of existing TCP, it has a set of simple API that is easy to use. With little modifications on existing TCP, we achieve unreliability, but yet retain TCP friendliness to different versions of TCP. Further, TCP Urel costs little CPU overhead. As a TCP option, Urel is able to keep TCP friendliness even when TCP itself evolves in the future. Being simple, efficient, and easy to use, TCP Urel offers one more choice for congestion-controlled unreliable streaming.

TCP Urel is not designed to challenge DCCP CCID2 in all respects, because DCCP has many features that is not foreseen at the age when TCP was originally designed, such as the reverse path congestion control scheme. But we believe that, due to its similarity to other non-Urel TCP, TCP Urel could be very easily adopted by applications that previously use TCP for streaming.

Our future work includes an extensive evaluation of applying TCP Urel to applications. Comparative study between TCP Urel, SCTP and DCCP CCID2 will also be conducted.

Source code of TCP Urel and the full set of emulation scripts based on FreeBSD 5.4 are available at http://www.comp.nus.edu.sg/∼malin.

Chapter 6

Conclusion and Future Work

Our work in distributed retransmission, DMSCC, and TCP Urel has contributed towards the improvement and deployment of distributed media streaming over the Internet. There are, however, many issues remain to be addressed. In this chapter, we conclude our work and outline possible future extensions.

6.1 Distributed Retransmission

Through comparisons to non-distributed retransmission, we show the effectiveness of distributed retransmission in distributed media streaming in reducing both the effective loss rate and packet loss burst length. The effectiveness of distributed retransmission comes from avoiding retransmission on the path that originally lost the packet; this principle reduces the chance of missing the retransmitted packet due to error burst. We propose a distributed retransmission scheme, ARQ-L, that keeps track of packet loss rate on each channel and retransmits only from the channel with the lowest packet loss rate. Experiments show that this scheme provides the lowest effective packet loss rate among the distributed retransmission schemes.

Distributed retransmission is not only useful to distributed media streaming.

Its principle applies to multi-path streaming and other applications that involve multiple sources/channels.

Research can be extended in the following aspects regarding distributed retransmission. First, if the bandwidth of the channels are variable and retransmission consumes limited bandwidth, how should the retransmitter be chosen? In such model, retransmitting a packet may delay the other data packets and reduce the media quality at the receiver. Choices must be made to balance the loss rate, the delay, and the bandwidth to achieve the lowest effective loss rate. Second, we use on packet losses to estimate the quality of the channels and choose the retransmitter. But, one-way delay, which reveal congestion in the network earlier than packet loss, could be an alternative metric that can be used to decide retransmitter. A distributed retransmission scheme that uses delay as indicator of channel quality (e.g., channel correlation) would be an intersting study.

6.2 DMSCC

We study congestion control in distributed media streaming and design a scheme to achieve task-level TCP-friendliness. We present the idea of task-level congestion control, which identifies a bottleneck and enforces TCP-friendliness over the subset of the application flows that pass through the bottleneck. We found that by adjusting the increasing factor of the AIMD algorithm of a congestion controlled flow, we can control its steady state throughput in a bottleneck. We also found that by observing the correlation of one-way delay of the paths, we can detect the location of the congestion and the set of application flows upon which TCP-friendliness should be enforced. DMSCC combined the above two components: it detects the correct set of flows using congestion location, and it changes their increasing factors to make their total throughput TCP-friendly.

The concept of task-level TCP-friendliness gives a different perspective to the meaning of TCP-friendliness. It is usable in other scenarios where multiple flows are engaged in the same application, and where bottleneck affects different set of flows (e.g., multi-source peer-to-peer file sharing). The method to control the aggregate throughput of DMS flows might be useful in other contexts as well, including controlling the throughput of parallel TCP connections.

Our throughput control algorithm is based on Mathis equation, and therefore does not work accurately in all network conditions (e.g., when loss is frequent and bursty). Our congestion location algorithm relies on Rubenstein's method. Identifying location of congestion in multiple congestions scenario with high delay interference remains a challenging problem. Our future work aims to address these limitations.

6.3 TCP Urel

We extend TCP for unreliable data streaming. By keeping TCP sequence number for congestion control and carrying data sequence number for data ordering, TCP Urel is able to avoid retransmission and keep congestion control intact. We present the detailed design and implementation of TCP Urel, and we evaluate its TCP-friendliness as well as protocol efficiency.

The usage of TCP Urel is much broader than distributed media streaming. It can be applied to other loss insensitive streaming applications, that require TCP-like AIMD congestion control. Changing existing TCP-based streaming applications to use TCP Urel is extremely easy, and the retransmission can be handled by application layer flexibly.

As future study, comprehensive comparison between DCCP CCID2 and TCP Urel could be carried out. Application specific measurement study of TCP Urel

should also be conducted to provide a complete evaluation of the protocol.

6.4 Availability of Code

All the code and scripts that are necessary to reproduce the experimental results are available at `http://www.comp.nus.edu.sg/~malin`. They include a Live555.COM based distributed MP3 streaming program for testing distributed retransmission, an ns-2 simulation package for DMSCC, a FreeBSD 5.4 implementation of TCP stack with TCP Urel, and all the experimental scripts.

Bibliography

[1] ISO/IEC 13818: Generic Coding of Moving Pictures and Associated Audio (MPEG-2).

[2] V. Agarwal and R. Rejaie. Adaptive multisource streaming in heterogeneous peer-to-peer networks. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, USA, December 2004.

[3] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee. On multiple description streaming with content delivery networks. In *Proceedings of IEEE INFOCOM '02*, New York, New York, USA, June 2002.

[4] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *Proceedings of ACM SIGCOMM '99*, Cambridge, Massachusetts, USA, September 1999.

[5] D. Bansal and H. Balakrishnan. Binomial congestion control algorithms. In *Proceedings of IEEE INFOCOM '01*, Anchorage, Alaska, USA, April 2001.

[6] H. G.-M. Beverly Yang. Comparing hybrid peer-to-peer systems. In *Proceedings of the International Conference on Very Large Data Base*, Rome, Italy, September 2001.

[7] J. C. Bolot, S. Fosse-Parisis, and D. Towsley. Adaptive FEC-based error control for Internet telephony. In *Proceedings of IEEE INFOCOM '99*, New York, New York, USA, March 1999.

[8] R. Bruyeron, B. Hemon, and L. Zhang. Experimentations with TCP selective acknowledgment. *ACM SIGCOMM Computer Communication Review*, 28(2):54–77, April 1998.

[9] CacheLogic Research. Peer-to-peer in 2005. `http://www.cachelogic.com/research/p2p2005.php`.

[10] J. Chakareski and P. Frossard. Distributed sender-driven video streaming. In *Proceedings of Visual Communications and Image Processing*, San Jose, California, USA, Jan 2006.

[11] G. Cheung and W. Tan. Reference frame optimization for multi-path video streaming using complexity scaling. In *Proceedings of Packet Video Workshop*, Irvine, California, USA, December 2004.

[12] S. Cho and R. Bettati. Adaptive aggregated aggressiveness control on parallel TCP flows using competition detection. In *Proceedings of IEEE International Conference on Computer Communications and Networks*, Arlington, Virginia, USA, October 2006.

[13] P. Chou and Z. Miao. Rate-distortion optimized streaming of packetized media. Technical report, Microsoft Research Technical Report MSR-TR-2001-35, February 2001.

[14] A. L. H. Chow, L. Golubchik, J. C. S. Lui, and W.-J. Lee. Multi-path streaming: optimization of load distribution. *Performence Evaluation*, 62(1-4):417–438, 2005.

[15] J. Chung, M. Claypool, and R. Kinichi. MTP: a streaming-friendly transport protocol. Technical report, Technical Report WPI-CS-TR-05-02, Worcester Polytechnic Institute, May 2005.

[16] K. G. Coffman and A. M. Odlyzko. Internet growth: is there a "Moore's Law" for data traffic? *Handbook of Massive Data Sets*, pages 47–93, 2002.

[17] T. Connolly, P. Amer, and P. Conrad. An Extension to TCP : Partial Order Service, RFC1693, November 1994.

[18] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportionally fair sharing TCP. *ACM SIGCOMM Computer Communication Review*, 28:53–67, July 1998.

[19] Y. Cui and K. Nahrstedt. Layered peer-to-peer streaming. In *Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Monterey, California, USA, June 2003.

[20] C. Dana, D. Li, D. Harrison, and C.-N. Chuah. BASS: BitTorrent assisted streaming system for video-on-demand. In *Proceedings of IEEE International Workshop on Multimedia Signal Processing*, Shanghai, China, October 2005.

[21] A. Edwards and S. Muir. Experiences implementing a high performance TCP in user-space. In *ACM SIGCOMM '95: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, New York, New York, USA, August 1995.

[22] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3):5–21, July 1996.

[23] M. Feldman, C. Papadimitriou, J. Chuang, and I. Stoica. Free-riding and whitewashing in peer-to-peer systems. In *PINS '04: Proceedings of the ACM SIGCOMM Workshop on Practice and Theory of Incentives in Networked Systems*, Portland, Oregon, USA, August 2004.

[24] R. Finlayson. A more loss-tolerant RTP payload format for MP3 Audio, RFC 3119, June 2001.

[25] S. Floyd. Congestion Control Principles, RFC2914, September 2000.

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

[27] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proceedings of ACM SIGCOMM '00*, Stockholm, Sweden, August 2000.

[28] S. Floyd and V. Jacobson. Traffic phase effects in packet-switched gateways. *ACM SIGCOMM Computer Communication Review*, 21(2):26–42, April 1991.

[29] S. Floyd and E. Kohler. Profile for DCCP congestion control ID 2: TCP-like congestion control. `http://www.ietf.org/internet-drafts/draft-ietf-dccp-ccid2-10.txt`, March 2005. IETF Internet draft.

[30] M. Ghanbari. Two-layer coding of video signals for VBR networks. *IEEE Journal on Selected Areas in Communications*, 7(5):771–781, June 1989.

[31] L. Golubchik, J. C. S. Lui, T. F. Tung, A. L. H. Chow, W.-J. Lee, G. Franceschinis, and C. Anglano. Multi-path continuous media streaming: what are the Benefits? *Performence Evaluation*, 49(1-4):429–449, 2002.

[32] T. J. Hacker, B. D. Noble, and B. D. Athey. Improving throughput and maintaining fairness using parallel TCP. In *Proceedings of IEEE INFOCOM '04*, Hong Kong, China, March 2004.

[33] H. Han, S. Shakkottai, C. Hollot, R. Srikant, and D. Towsley. Overlay TCP for multi-path routing and congestion control. In *ENS-INRIA ARC-TCP Workshop*, Paris, France, November 2003.

[34] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): protocol specification, RFC3448, January 2003.

[35] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava. PROMISE: Peer-to-peer media streaming using CollectCast. In *Proceedings of ACM International Conference on Multimedia*, Berkeley, California, USA, November 2003.

[36] M. Hofmann and L. R. Beaumont. *Content Networking: Architecture, Protocols, and Practice*. Morgan Kaufmann Publisher, 2005.

[37] C. Hsu and M. Hefeeda. Optimal bit allocation for fine-grained scalable video sequences in distributed streaming environments. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, USA, Jan 2007.

[38] C.-M. Huang, K.-C. Yang, , and J.-S. Wang. Error resilience supporting bidirectional frame recovery for video Streaming. In *Proceedings of IEEE International Conference on Image Processing*, Singapore, October 2004.

[39] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance, RFC1323, May 1992.

[40] T. Kim and M. Ammar. Receiver buffer requirements for video streaming over TCP. In *Proceedings of Visual Communications and Image Processing Conference*, San Jose, California, USA, January 2006.

[41] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: congestion control without reliability. In *Proceedings of ACM SIGCOMM '06*, Pisa, Italy, September 2006.

[42] C. Krasic, K. Li, and J. Walpole. The case for streaming multimedia with TCP. In *Proceedings of International Workshop on Interactive Distributed Multimedia Systems*, Ancaster, UK, September 2001.

[43] H. Kung and S. Wang. TCP trunking: design, implementation and performance. In *Proceedings of International Conference on Network Protocols*, Toronto, Canada, October 1999.

[44] J. Lai and E. Kohler. Efficiency and late data choice in a user-kernel interface for congestion-controlled datagrams. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, USA, January 2005.

[45] J. Lazzaro. Framing Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP) packets over connection-oriented transport, RFC 4571, July 2006.

[46] Y.-C. Lee, J. Kim, Y. Altunbasak, and R. M. Mersereau. Performance comparisons of layered and multiple description coded video streaming over error-prone networks. In *Proceedings of IEEE Conference on Communications*, Seattle, Washington, USA, May 2003.

[47] A. Legout and E. W. Biersack. Pathological behaviors for RLM and RLC. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video*, Chapel Hill, North Carolina, USA, June 2000.

[48] D. Li, C.-N. Chuah, G. Cheung, and S. J. Yoo. MUVIS: multi-source video streaming for video-on-demand over IEEE 802.11 WLAN. *Journal of Communications and Networks - Special Issue on Towards the Next Generation Mobile Communications*, 7(2):144–156, June 2005.

[49] D. Li, Q. Zhang, C.-N. Chuah, and S. J. B. Yoo. Multi-source multi-path video streaming over wireless mesh networks. In *Proceedings of IEEE International Symposium on Circuits and Systems*, Island of Kos, Greece, May 2006.

[50] J. Li. PeerStreaming: a practical receiver-driven peer-to-peer media streaming system. Technical report, Microsoft Research Report MSR-TR-2004-101, September 2004.

[51] W. Li. Overview of fine granularity scalability in MPEG-4 Video Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):301–317, March 2001.

[52] Y. J. Liang, E. G. Steinback, and B. Girod. Real-time voice communication over the Internet using packet path diversity. In *Proceedings of ACM International Conference on Multimedia*, Ottawa, Ontario, Canada, September 2001.

[53] D. Loguinov and H. Radha. Retransmission schemes for streaming Internet multimedia: evaluation model and performance analysis. *ACM SIGCOMM Computer Communication Review*, 32(2):70–83, April 2002.

[54] K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7:72–93, 2005.

[55] L. Ma and W. T. Ooi. Retransmission in distributed media streaming. In *Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Stevenson, Washington, USA, June 2005.

[56] L. Ma and W. T. Ooi. Congestion control in distributed media streaming. In *Proceedings of IEEE INFOCOM '07*, Anchorage, Alaska, USA, May 2007.

[57] Z. Ma, H.-R. Shao, and C. Shen. A new multi-path selection scheme for video streaming on overlay networks. In *Proceedings of IEEE International Conference on Communications*, Paris, France, June 2004.

[58] A. Majumdar, R. Puri, and K. Ramchandran. Distributed multimedia transmission from multiple servers. In *Proceedings of IEEE International Conference on Image Processing*, Rochester, New York, USA, September 2002.

[59] K. Mansley. Engineering a user-level TCP for the CLAN network. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O Convergence*, New York, New York, USA, December 2003.

[60] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Sanadidi, and M. Roccetti. TCP-Libra: exploring RTT fairness for TCP. Technical report, UCLA Computer Science Department Technical Report TR050037, 2005.

[61] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment options, RFC2108, October 1996.

[62] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27:67–82, July 1997.

[63] D. McCreary, K. Li, S. A. Watterson, and D. K. Lowenthal. TCP-RC: a receiver-centered TCP protocol for delay-sensitive applications. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, USA, January 2005.

[64] D.-E. Meddour, M. Mushtaq, and T. Ahmed. Open issues in P2P multimedia streaming. In *Proceedings of Multimedia Communications Workshop: State of the Art and Future Directions*, Istanbul, Turkey, June 2006.

[65] A. Mena and H. Heidemann. An empirical study of RealAudio traffic. In *Proceedings of IEEE INFOCOM '00*, Tel Aviv, Israel, March 2000.

[66] J. Nagle. Congestion Control in IP/TCP Internetworks, RFC896, January 1984.

[67] T. Nguyen and S. Cheung. Multimedia streaming with multiple TCP connections. In *Proceedings of International Performance Computing and Communications Conference*, Phoenix, Arizona, USA, April 2005.

[68] T. Nguyen and A. Zahkor. Distributed video streaming over the Internet. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, USA, January 2002.

[69] T. Nguyen and A. Zahkor. Distributed video streaming with forward error correction. In *Proceedings of Packet Video Workshop*, Pittsburgh, Pennsylvania, USA, April 2002.

[70] T. Nguyen and A. Zakhor. Multiple sender distributed video streaming. *IEEE Transactions on Multimedia*, 6(2):315–326, April 2004.

[71] V. T. Nguyen, E.-C. Chang, and W. T. Ooi. Layered coding with good allocation outperforms multiple description coding over multiple paths. In *Proceedings of IEEE International Conference on Multimedia and Expo*, Taipei, Taiwan, China, June 2004.

[72] J. Nichols, M. Claypool, R. Kinicki, and M. Li. Measurements of the congestion responsiveness of Windows Streaming Media. In *Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video*, County Cork, Ireland, June 2004.

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

[74] C. Papadopoulos and G. Parulkar. Retransmission-based error control for continuous media applications. In *Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Zushi, Japan, April 1996.

[75] C. Perkins, O. Hodson, and V. Hardman. A survey of packet loss recovery techniques for streaming audio. *IEEE Network Magazine*, 12(5):40–48, September/October 1998.

[76] M. Piecuch, K. French, G. Oprica, and M. Claypool. A selective retransmission protocol for multimedia on the Internet. In *Proceedings of International Symposium on Multimedia Systems and Applications*, Boston, Massachusette, USA, November 2000.

[77] R. Puri, K. R. K. Lee, and V. Bharghavan. Application of FEC based multiple description coding to Internet video streaming and multicast. In *Proceedings of Packet Video Workshop*, Forte Village Resort, Sardinia, Italy, May 2000.

[78] R. Koenen (Editor). Overview of the MPEG-4 Standard. ISO/IEC JTC1/SC29/ WG11 (2001).

[79] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of International Workshop on Networked Group Communication*, London, England, November 2001.

[80] R. Rejaie, M. Handley, and D. Estrin. RAP: an end-to-end rate-based congestion control mechanism for realtime streams in the Internet. In *Proceedings of IEEE INFOCOM '99*, New York, New York, USA, March 1999.

[81] R. Rejaie and A. Ortega. PALS: peer-to-peer adaptive layered streaming. In *Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Monterey, California, USA, June 2003.

[82] L. Rizzo. pgmcc: a TCP-friendly single-rate multicast congestion control scheme. In *Proceedings of ACM SIGCOMM '00*, Stockholm, Sweden, August 2000.

[83] D. Rubenstein, J. Kurose, and D. Towsley. Detecting shared congestion of flows via end-to-end measurement. *IEEE/ACM Transactions on Networking*, 10(3):381–395, June 2002.

[84] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, USA, January 2002.

[85] M. Singh, P. Pradhan, and P. Francis. MPAT: aggregate TCP Congestion management as a building block for Internet QoS. In *Proceedings of IEEE International Conference on Network Protocols*, Berlin, Germeny, October 2004.

[86] W. R. Stevens. *TCP/IP Illustrated, Volume 1. The protocols*, chapter 21. Addiso-Wesley, 1994.

[87] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension, RFC3758, May 2004.

[88] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transaction on Networking*, 11(1):17–32, 2003.

[89] S. Takeuchi, H. Koga, K. Iida, Y. Kadobayashi, and S. Yamaguchi. Performance evaluations of DCCP for bursty traffic in real-time applications. In *IEEE/IPSJ International Symposium on Applications and the Internet (SAINT)*, Trento, Italy, January 2005.

[90] J. van der Merwe, S. Sen, and C. Kalmanek. Streaming video traffic: characterization and network Impact. In *Proceedings of the International Workshop on Web Content Caching and Distribution*, Boulder, CO, USA, August 2002.

[91] L. Vicisano, J. Crowcroft, and L. Rizzo. TCP-like congestion control for layered multicast data transfer. In *Proceedings of IEEE INFOCOM '97*, Kobe, Japan, April 1997.

[92] M. Vojnovic and J.-Y. L. Boudec. On the long-run behavior of equation-based rate control. In *ACM SIGCOMM '02: Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pittsburgh, Pennsylvania, USA, August 2002.

[93] J. Widmer, R. Denda, and M. Mauve. A survey on TCP-friendly congestion control. *IEEE Network Magazine, Special Issue on Control of Best Effort Traffic*, 15(3):28–37, May 2001.

[94] R. Wittmann and M. Zitterbart. *Multicast Communication: Protocols, Programming, and Applications*. Morgan Kaufmann Publishers, May 2000.

[95] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

[96] D. Wu, Y. Hou, W. Zhu, Y. Zhang, and J. Peha. Streaming video over the Internet: approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology, Special Issue on Streaming Video*, 11(3):282–300, March 2001.

[97] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On peer to peer media streaming. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[98] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control for fast, long distance networks. In *Proceedings of IEEE INFOCOM '04*, Hong Kong, China, March 2004.

[99] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: a reordering-robust TCP with DSACK. In *Proceedings of the IEEE International Conference on Networking Protocols*, Atlanta, GA, USA, November 2003.