TCP Boston

A Fragmentation-tolerant TCP Protocol for ATM Networks*

Azer Bestavros
best@cs.bu.edu

Gitae Kim
kgtjan@cs.bu.edu

Computer Science Department
Boston University
Boston, MA 02215

Tel: (617) 353-9726
Fax: (617) 353-6457

Abstract

The popularity of TCP/IP coupled with the premise of high speed communication using Asynchronous Transfer Mode (ATM) technology have prompted the network research community to propose a number of techniques to adapt TCP/IP to ATM network environments. ATM offers Available Bit Rate (ABR) and Unspecified Bit Rate (UBR) services for best-effort traffic, such as conventional file transfer. However, recent studies have shown that TCP/IP, when implemented using ABR or UBR, leads to serious performance degradations, especially when the utilization of network resources (such as switch buffers) is high. Proposed techniques—switch-level enhancements, for example—that attempt to patch up TCP/IP over ATMs have had limited success in alleviating this problem. The major reason for TCP/IP's poor performance over ATMs has been consistently attributed to packet fragmentation, which is the result of ATM's 53-byte cell-oriented switching architecture.

In this paper, we present a new transport protocol, TCP Boston, that turns ATM's 53-byte cell-oriented switching architecture into an advantage for TCP/IP. At the core of TCP Boston is the Adaptive Information Dispersal Algorithm (AIDA), an efficient encoding technique that allows for dynamic redundancy control. AIDA makes TCP/IP's performance less sensitive to cell losses, thus ensuring a graceful degradation of TCP/IP's performance when faced with congested resources. In this paper, we introduce AIDA and overview the main features of TCP Boston. We present detailed simulation results that show the superiority of our protocol when compared to other adaptations of TCP/IP over ATMs. In particular, we show that TCP Boston improves TCP/IP's performance over ATMs for both network-centric metrics (*e.g.*, effective throughput) and application-centric metrics (*e.g.*, response time).

Keywords: ATM networks; TCP/IP; Adaptive Information Dispersal Algorithm; congestion control; performance evaluation.

*This work has been partially funded by NSF grant CCR-9308344.

1 Introduction

In the last few years, the Transmission Control Protocol (TCP) [22]—a reliable transport protocol that uses a window-based flow and error control algorithm on top of the Internet Protocol (IP) layer—has emerged as the standard in data communication. The proliferation of TCP/IP is clearly manifest in the vast array of services and applications that rely on TCP's robust functionality and its hiding of the underlying details of networks of various scales and technologies, from Local Area Networks (LANs) to Wide Area Networks (WANs), and from Ethernets to Satellite networks.

Recently, the introduction of the Asynchronous Transfer Mode (ATM) technology has raised many questions regarding the effectiveness of using TCP over ATM networks. The ATM technology is a connection oriented, 53-byte cell-based transport technology, which offers high-speed switching for both LANs and WANs. ATM is designed to support a variety of applications with diverse requirements ranging from the real-time constrained delivery of live audio and video, to the best-effort delivery of conventional data such as FTP and email [12].

The flexibility and popularity of TCP/IP coupled with the premise of high speed communication using emerging ATM technology have prompted the network research community to propose and implement a number of techniques that adapt TCP/IP to ATM network environments, thus allowing these environments to smoothly integrate (and make use of) currently available TCP-based applications and services without much (if any) modifications [12]. However, recent studies [8, 17, 24] have shown that TCP/IP, when implemented over ATM networks, is susceptible to serious performance limitations.

The poor performance of TCP over ATMs is mainly due to *packet fragmentation*. Fragmentation occurs when an IP packet flows into an ATM virtual circuit through the AAL5 (ATM Adaptation Layer 5), which is the emerging, most common AAL for TCP/IP [1] over ATMs. AAL5 acts as an interface between the IP and ATM layers. It is responsible for the task of dividing TCP/IP's large data units (*i.e.*, the TCP/IP packets) into sets of 48-byte data units called *cells*. Since the typical size of a TCP/IP packet is much larger than that of a cell,[1] fragmentation at the AAL is inevitable. In order for a TCP/IP packet to successfully traverse an ATM switching network (or subnetwork), all the cells belonging to that packet must traverse the network *intact*. The loss even of a single cell in any of the network's ATM switches results in the corruption of the entire packet to which that cell belongs. Notice however that when a cell is dropped at a switch, the rest of the cells that belong to the same packet still proceed through the virtual circuit, despite the fact that they are destined to be discarded by the destination's AAL at the time of packet-reassembly, thus resulting in low effective throughput.

[1] This is mainly due to TCP/IP's headers (the minimum number of bytes required for commonly used TCP/IP header fields is 40).

There have been a number of attempts to remedy this problem by introducing additional switch-level functionalities to preserve throughput when TCP/IP is employed over ATM. Examples include the Selective Cell Discard (SCD)[2] [2] and the Early Packet Discard (EPD) [24]. In SCD, once a cell c is dropped at a switch, all subsequent cells from the packet to which c belongs are dropped by the switch. In EPD, a more aggressive policy is used, whereby all cells from the packet to which c belongs are dropped, including those still in the switch buffer (*i.e.* preceding cells that were in the switch buffer at the time it was decided to drop c). Notice that both SCD and EPD require modifications to switch-level software. Moreover, these modifications require the switch-level to be aware of IP packet boundaries—a violation of the layering principle that was deemed unavoidable for performance purposes in [24].

The simulation results described in [24] show that both SCD and EPD improve the effective throughput of TCP/IP over ATMs. In particular, it was shown that the effective throughput achievable through the use of EPD approaches that of TCP/IP in the absence of fragmentation. It is important to note that these results were obtained for a network consisting of a single ATM switch. For realistic, multi-hop ATM networks the cumulative wasted bandwidth (as a result of cells discarded through SCD or EPD) may be large, and the impact of the ensuing packet losses on the performance of TCP is likely to be severe. To understand these limitations, it is important to realize that while dropping cells belonging to a packet at a congested switch preserves the bandwidth of that switch, it does not preserve the ABR/UBR bandwidth at all the switches preceding that (congested) switch along the virtual circuit for the TCP connection. Moreover, any cells belonging to a corrupted packet which would have made it out of the congested switch will continue to waste the bandwidth at all the switches following that (congested) switch. Obviously, the more hops separating the TCP/IP source from the TCP/IP destination, the more wasted ABR/UBR bandwidth one would expect even if SCD or EPD techniques are used. This wasted bandwidth translates to low effective throughput, which in turn results in more duplicate data packets transmitted from the source, in effect increasing the response time for the applications.

To summarize, techniques for improving TCP/IP's performance over ATMs based on link-level enhancements do not take advantage of ATM's unique, small-sized cell switching environment; they *cope* with it. Furthermore, we argue that these techniques are not likely to scale for large, multi-hop ATM networks.

In this paper, we present a new transport protocol, TCP Boston, that turns fragmentation into an advantage for TCP/IP, thus enhancing the performance of TCP in general and its performance in ATM environments in particular. The rationale that motivates the design of TCP Boston lies in our answer to the following simple question: *Could a partial delivery of a packet be useful?* Our

[2]Also called Partial Packet Discard (PPD) in [24].

answer is *yes*. In other words, the *en route* loss of one fragment (or more) from a packet does not render the rest of the fragments belonging to that packet useless. TCP Boston manages to make use of such partial information, thus preserving network bandwidth. At the core of TCP Boston is the Adaptive Information Dispersal Algorithm (AIDA), an efficient encoding technique that allows for dynamic redundancy control. AIDA makes TCP/IP's performance less sensitive to cell losses, thus ensuring a graceful degradation of TCP/IP's performance when faced with congested resources.

This paper focuses on the protocol, implementation, and performance analysis of TCP Boston. Simulation and analysis are used to measure and comparatively evaluate the performance of TCP Boston in an ATM UBR environment, in which ATM rate-based congestion control is not available.

The remainder of this paper is organized as follows. In section 2, we introduce AIDA and overview the main features of TCP Boston. In section 3, we present our simulation model and experimental setup. In section 4, we present and discuss the simulated performance of TCP Boston and compare it to that of TCP Reno, and to that of TCP Reno with the EPD switch-level enhancements. These simulations demonstrate the superiority of our protocol—measured using both network-centric metrics (*e.g.*, effective throughput) and application-centric metrics (*e.g.*, response time)—when compared to other adaptations of TCP/IP over ATMs. An analytical formulation at the end of that section allows us to extend our conclusions about the performance of TCP Boston to multi-hop network environments. We conclude in section 5 with a summary and a discussion of our on-going work.[3]

2 TCP Boston: Principles, Protocol, and Implementation

We start this section with an introduction to AIDA. Next, we show how to incorporate AIDA into the TCP/IP stack, and we discuss the various implementation aspects that are incorporated in the current version of TCP Boston (hereinafter interchangeably referred to by "Boston") that is used in our simulations.

2.1 AIDA: An Introduction

AIDA is a novel technique for dynamic bandwidth allocation, which makes use of minimal, controlled redundancy to guarantee timeliness and fault-tolerance up to *any* degree of confidence. AIDA is an elaboration on the Information Dispersal Algorithm of Michael O. Rabin [23], which has been previously shown to be a sound mechanism that considerably improves the performance

[3]For a smoother presentation flow, studies related to ours are discussed throughout the paper, rather than in a specific section.

of I/O systems, parallel/distributed storage devices [4], and real-time broadcast disks [7]. The use of IDA for efficient routing in parallel architectures has also been exploited in [19].

To understand how IDA works, consider a segment S of a data object to be transmitted. Let S consist of m fragments (hereinafter called *cells*). Using IDA's *dispersal operation*, S could be processed to obtain N distinct pieces in such a way that recombining *any* m of these pieces, $m \leq N$, using IDA's *reconstruction operation*, is sufficient to retrieve S. Figure 1 illustrates the dispersal, communication, and reconstruction of an object using IDA. Both the dispersal and reconstruction operations can be performed in real-time. This was demonstrated in [5], where an architecture and a CMOS implementation of a VLSI chip that implements IDA was presented.[4]

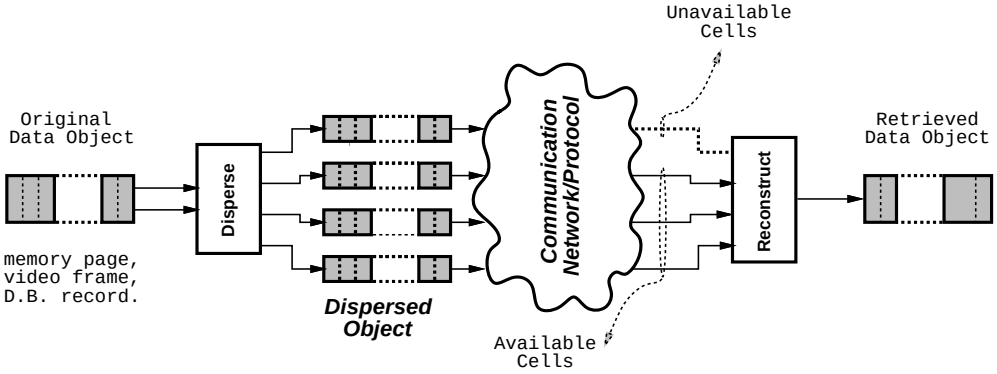

Figure 1: Dispersal and reconstruction of information using IDA.

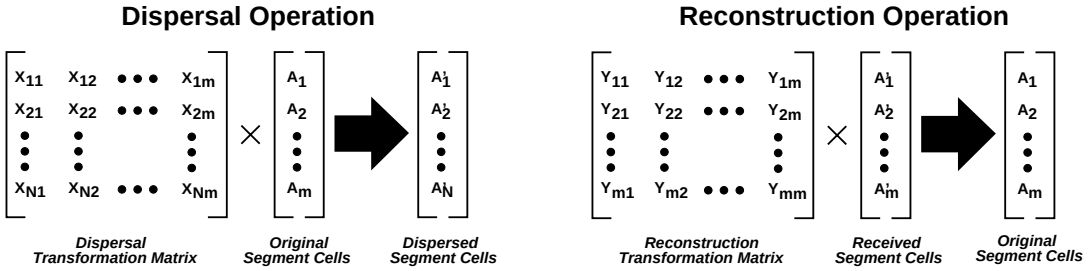

Figure 2: The Dispersal and Reconstruction operations of IDA.

The dispersal and reconstruction operations are simple linear transformations using *irreducible polynomial arithmetic*.[5] The dispersal operation shown in figure 2 amounts to a matrix multiplica-

[4]The chip (called SETH) has been fabricated by a MOSIS (using a 3-micron process) and tested in the VLSI lab of Harvard University, Cambridge, MA. The performance of the chip was measured to be about 1 Mbps. By using proper pipelining, more elaborate designs, and a more advanced VLSI fabrication process, this figure can be boosted significantly.

[5]For more details, we refer the reader to [23, 5].

tion (performed in the domain of a particular irreducible polynomial) that transforms the m cells of the original file into the N cells to be dispersed. The N rows of the transformation matrix $[x_{ij}]_{N\times m}$ are chosen so that any m of these rows are mutually independent, which implies that the matrix consisting of any such m rows is not singular, and thus inversible. This guarantees that reconstructing the original file from *any* m of its dispersed cells is feasible. Indeed, upon receiving any $r \geq m$ of the dispersed cells, it is possible to reconstruct the original segment through another matrix multiplication as shown in figure 2. The transformation matrix $[y_{ij}]_{m\times m}$ is the inverse of a matrix $[x'_{ij}]_{m\times m}$, which is obtained by removing $N - m$ rows from $[x_{ij}]_{N\times m}$. The removed rows correspond to the cells that were not used in the reconstruction process. To reduce the overhead of the algorithm, the inverse transformation $[y_{ij}]_{m\times m}$ could be precomputed for some or even all possible subsets of m rows.

Several *redundancy-injecting* protocols (similar to IDA) have been suggested in the literature. In most of these protocols, redundancy is injected in the form of parity, which is only used for error detection and/or correction purposes [16]. The IDA approach is radically different in that redundancy is added *uniformly*; there is simply *no* distinction between data and parity. It is this feature that makes it possible to scale the amount of redundancy used in IDA. Indeed, this is the basis for *Adaptive* IDA (AIDA) [6]. Using AIDA, a *bandwidth allocation* operation is inserted after the dispersal operation but *prior* to transmission as shown in figure 3. This bandwidth allocation step allows the system to *scale* the amount of redundancy used in the transmission. In particular, the number of cells to be transmitted, namely n, is allowed to vary from m (*i.e.*, no redundancy) to N (*i.e.*, maximum redundancy).

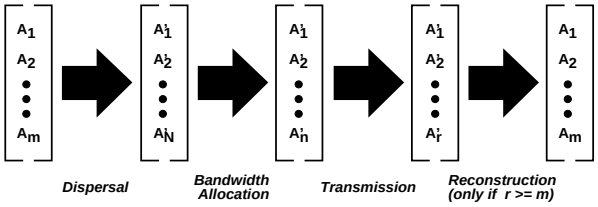

Figure 3: AIDA dispersal and reconstruction

2.2 AIDA Characteristics

In order to appreciate the advantages that AIDA brings to TCP Boston, we must understand the main difficulty posed by fragmentation. When a cell is lost *en route*, it becomes impossible for the receiver to reconstruct the packet to which that cell belonged unless: (1) there is enough extra (redundant) cells from the packet in question to allow for the recovery of the missing information (*e.g.*, through parity), or (2) the cell is retransmitted.

The first solution above suggests the use of spatial redundancy to mask erasures (cell losses). While feasible, such a technique may be quite wasteful of bandwidth (since the redundant information will have to be communicated whether or not erasures occur), and is not likely to help when cell losses exceed the forward erasure capacity of the encoding scheme, which is almost certainly the case since cells are typically dropped in "batches" when switches run out of buffer space. An example of the use of this approach is the study in [9], which suggests the use of *Forward Error Correction* (FEC) for real-time, unreliable video communication over ATM. In that study, FEC was shown to allow the trading of bandwidth for timeliness. FEC's performance was shown to depend on many parameters including the network load, the level of redundancy injected into FEC traffic, and the percentage of connections (traffic) using FEC. FEC was shown to be most effective when corruption is restricted to few cell erasures per data block (*e.g.*, video frame).

Similar to FEC, AIDA supports the use of spatial redundancy to mask erasures. Furthermore, when incorporated with TCP, AIDA allows this support to be fully integrated within the flow control mechanism of TCP, thus making it possible to perform forward error correction *without* necessarily overloading the network resources. For example, if network congestion is detected, one could increase AIDA's level of spatial redundancy (thus protecting against likely cell drops), while decreasing TCP's congestion window size (thus protecting against buffer overflow by reducing the number of bytes "on the wire"). This integration of redundancy control and flow control in a reliable transport protocol[6] could be quite valuable for real-time communication as reported in [6].

The second solution above suggests the use of temporal redundancy to recover from erasures. Two possibilities exist—each representing an extreme in terms of the functionality required at the sender and receiver ends. The first extreme would be for the receiver to do nothing, and simply wait for the sender to automatically retransmit all cells from the packet in question as would be dictated by TCP's packet acknowledgment protocol. This is exactly what current adaptations of TCP over ATMs do (including the SCD and EPD techniques). As we explained before such an approach is not effective in terms of its use of available bandwidth, especially in multi-hop networks. Of course it has the advantage of being quite simple to implement since it requires no additional functionality at the sender and receiver ends. The other extreme would be for the receiver to keep track of which cells are missing and then to request retransmission of only those cells. This technique, which we will revisit later in this paper, has the advantage of being effective in terms of its use of available bandwidth, but may result in considerable overhead, especially when the level of fragmentation (*i.e.* number of cells per packet) is high.

The incorporation of AIDA in a TCP protocol allows us to strike a critical balance between the above two extremes. To explain how this could be done, consider the following scenario. The sender

[6]FEC is *not* a reliable transport mechanism.

disperses an outgoing m-cell segment (packet) into N cells, but sends a packet of only m of these cells to the receiver, where $N >> m$. Now, assume that the receiver gets r of these cells. If $r = m$, then the receiver could reconstruct the original segment, and acknowledge that it has *completely* received it by informing the sender that it needs *no* more cells from that segment. If $r < m$, then the receiver could acknowledge that it has *partially* received the packet by informing the sender that it needs $(m - r)$ more cells from the original segment. To such an acknowledgment, the sender would respond by sending a packet of $(m - r)$ fresh cells (*i.e.* not sent the first time around) from the original N dispersed cells. The process continues until the receiver receives enough cells (namely m or more) to be able to reconstruct the original segment.

Two important points must be noted. First, using AIDA, *no* bandwidth is wasted as a result of packet retransmission or partial packet delivery; every cell that makes it through the network is used. Moreover, this cell-preservation behavior is achieved *without* requiring individual cell acknowledgment. Second, using AIDA, *no* modification to the switch-level protocols is necessary. This stands in sharp contrast to the SCD and EPD techniques, which necessitate such a change. The incorporation of AIDA into TCP/IP over ATMs requires only additional functionality at the interface between the IP and ATM layers (*i.e.*, the AAL), which we discuss later in the paper.

Figure 4 shows the transmission window managed by AIDA in TCP Boston. As explained before, prior to a packet transmission, AIDA encodes the original m-cell packet into N cells ($N >> m$). Based on network congestion conditions, it dynamically adjusts n the *transmission window size*, which represents the size of the packet to be actually transmitted.

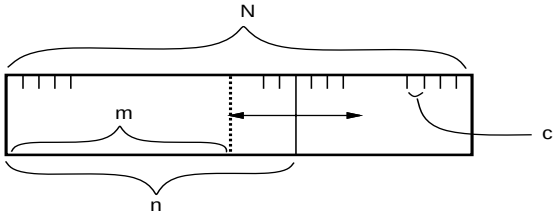

Figure 4: Transmission Window managed by AIDA

The transmission window manager can be custom-tuned to meet the spatial redundancy requirements of particular applications or services. For example, time-critical applications may require that the level of spatial redundancy be increased to mask cell erasures (up to a certain level), and thus to avoid retransmission delays should such erasures occur. By avoiding such delays, the likelihood that tight timing constraints will be met is increased (at the expense of wasted bandwidth).

In this paper, and since we focus on the incorporation of AIDA into the *bandwidth preserving* TCP/IP protocol stack, we do not exploit this feature any further. However, it should be clear

that the ability of AIDA to support the use of spatial redundancy makes it possible to extend TCP Boston seamlessly to support applications/environments for which timeliness (as opposed to preservation of bandwidth) is a key requirement.

2.3 Overview of TCP Boston

In this section we explain the essential aspects of our implementation of TCP Boston, with a special emphasis on those elements that are uncommon in other TCP implementations.

The purpose of this protocol is to provide a reliable transfer of data for end-to-end applications. The protocol, when properly tuned, can be implemented over both ATM and packet-switched networks. But, since it is designed in such a way that it takes advantage of ATM's relatively small-sized cell (*i.e.*, 53 bytes) environment, it can achieve a high performance gain when it is deployed over ATM networks.

The main functions included in the protocol are: *session management*, *segment management,* and *flow control and transmission.* We give a summary of these functions below:

Session Management: The protocol manages a TCP session in three phases: a *connection establishment* phase, a *data transfer* phase, and a *termination* phase. The purpose of these phases, as well as the functions performed therein, generally follow those of current TCP implementations, except that information specific to IDA which are required by the receiver for reconstruction purposes (such as the value of m for example), are piggy-backed onto the protocol packets during the three-way handshaking at the connection establishment phase.[7]

Segment Management: Processes for (1) segment encoding (at the source) and (2) segment reconstruction (at the sink) are unique to TCP Boston. These processes are described below.

- **Segment encoding:** Given a data block (segment) of size b bytes, the protocol divides the data block into m cells of size c, where $m = b/c$ bytes. Next, the m cells are processed using IDA to yield N cells for some $N >> m$. For example, if b = 1,000 bytes and c = 50, then $m = 20$, and N could be set to 40. For each cell, one byte of heading is required for identification purposes. This would be needed during reconstruction at the receiver end. Once this encoding is done, the first m cells from the segment are transmitted as a single packet and the unused $N - m$ cells are kept in a buffer area for use when (if) more cells from that segment must be transmitted to compensate for lost cells (see below).
- **Segment reconstruction:** When a packet of cells is received, the protocol first checks if it has accumulated m (or more) different cells from the segment that corresponds

[7] For efficiency, such information could be permanently *"coded"* into TCP Boston.

to that packet. If it did, it reconstructs the original segment using the proper IDA reconstruction matrix transformation and signals the flow control component to send an acknowledgment (hereinafter referred to as an `ACK`) indicating that reconstruction was successful. If not, it keeps the received cells for later reconstruction, and signals the flow control component to send an `ACK`, piggy-backed with the number of cells that have been accumulated so far from the segment. Such an `ACK` would inform the sender that reconstruction is not possible, and that the pending number of cells from that segment need to be transmitted at the time of next packet retransmission.

Flow Control and Transmission: Flow control determines the dynamics of packet flow in the network, which in turn affects the end-to-end performance of the system. Any feedback-based TCP flow control algorithm (*e.g.*, Tahoe, Reno, and Vegas) can be used with TCP Boston with a minor modification to handle the revised feedback mechanism of TCP Boston. When an `ACK` arrives, the sender checks a flag to determine if that `ACK` signals the successful reconstruction (at the receiver) of a segment. If it does, the sender calls the standard `ACK` procedure. If it doesn't, the sender extracts from the `ACK` the number of cells r received so far (see above) and then prepares $m - r$ additional cells from the desired segment in a single *new* packet that will be transmitted at the next retransmission time. This process continues until the receipt of an `ACK` from the receiver indicating that the segment has been successfully reconstructed, in which case any remaining cells from that segment are discarded from the sender's buffer.

Notice that the partial delivery of a packet does not result in updating the received-segment number for the receiver's TCP window manager.[8] Also, an `ACK` signaling a partial packet delivery does not cause an increase in the sender's congestion window. Rather, it acts as a hint to the sender to update the number of cells included in the next packet retransmission.

One of the most promising aspects of our protocol is the flexibility it provides for use in different networking environments and for different performance requirements. On the one hand, all the parameters of the transmission window (shown in figure 4) can be *adjusted* at the connection establishment phase of a given session to provide performance characteristics that fit best the requirements of that session. On the other hand, these parameters could be *preset* to provide "standard" levels of service. For example, if TCP Boston is to be used in a packet-switched environment with an MTU of 1K bytes, then by setting $c = 1,024$, $N = n = m = 1$, it behaves exactly like (*i.e.*, reduces to) the "default" underlying TCP protocol—be it Tahoe, Reno, or Vegas. However, if TCP Boston is to be used over an ATM network, then by setting $c = 48$ (*i.e.*, the

[8] This enables the receiver to send duplicate `ACK`s to signal a packet drop to the sender.

length of data bytes for an ATM cell), we make each ATM cell represent an AIDA cell. In this case, one cell drop at an ATM switch results in exactly one cell loss at the TCP receiver, thus providing the maximum protection against fragmentation. Such a protection comes at the price of an increase overhead due to the larger IDA transformation matrices.[9] To reduce this overhead, one could set c to a larger value, say $c = 192$, which would fragment each AIDA cell into four ATM cells—a fragmentation level of 1-to-4, which is much better than the 1-to-22 level for a 1K-byte TCP packet.

2.4 Implementation of TCP Boston

In our current implementation, the protocol is composed of three main modules: a *Session Management Module*, a *Segment Management Module*, and a *Flow Control Module*. Each of these modules executes the corresponding function described in the previous section. Figure 5 depicts the configuration and interaction of the three modules for both the sender and the receiver.

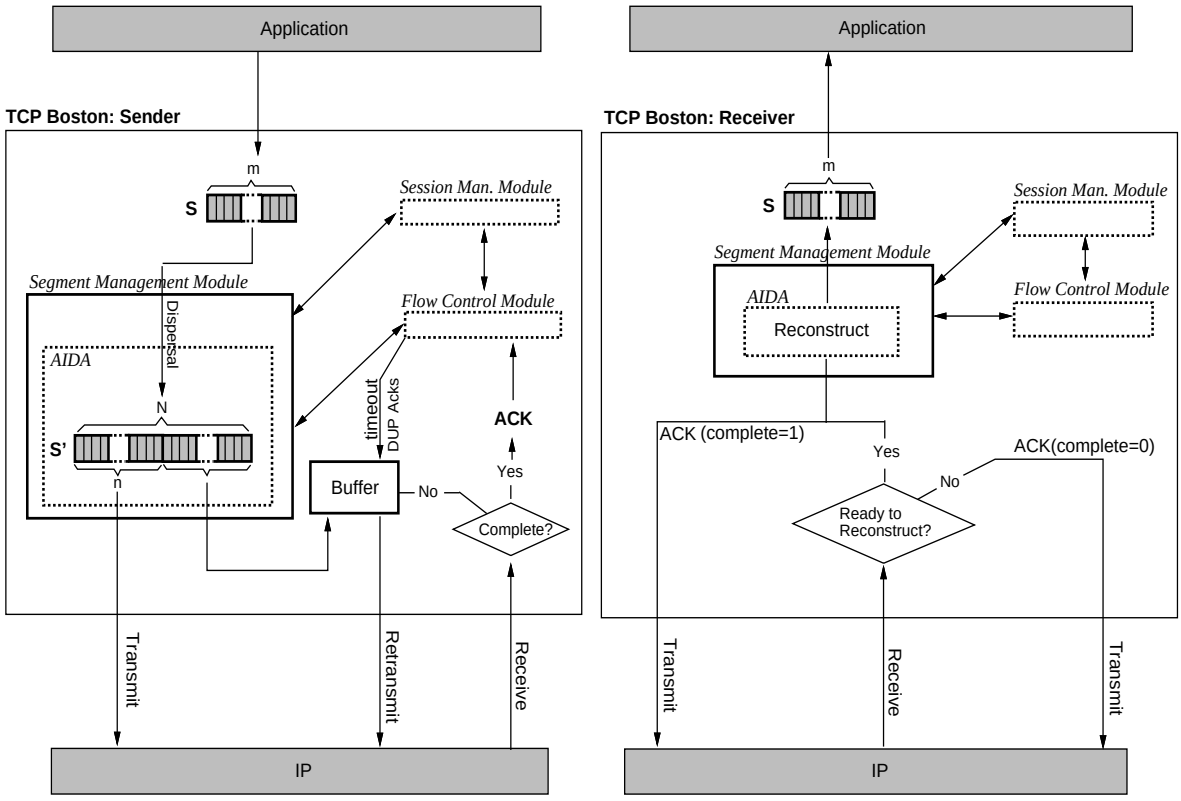

Figure 5: TCP Boston: Protocol Outline.

[9]Recall that $m = b/c$, thus a small c results in a larger m, and consequently larger IDA matrices (see figure 2).

The use of TCP Boston in ATM environments requires a modification to the AAL5 functionality; namely, AAL5 must allow the reassembly of partial IP packets when ATM cells are missing, instead of simply discarding such packets. To that effect, we propose two simple solutions. The first is that AAL5 could simply insert dummy cells in place of any missing cells.[10] If the assembled IP packet is to traverse other subnetworks, the insertion of these dummy cells may be deemed wasteful of bandwidth. The second solution remedies this by requiring AAL5 to simply pack the available cells into an IP packet (*i.e.* no dummy cells are inserted), and to update the IP headers to reflect among other things, the new (shorter) length of the IP packet. Notice that this second solution—while preserving bandwidth for inter-network traffic—requires non-trivial modifications to AAL5. However, since the handling of the IP packets is one of the main functions of AAL5, we believe that the required modifications are acceptable and conformant with the layering principle.[11] Moreover, this second option, which is the one chosen for our implementation of TCP Boston, requires more elaborate modules at the receiver end to validate partial packets.

For simulation purposes, we tuned the system so as to use *no* spatial redundancy. We chose to do so for three reasons: (1) We wanted to evaluate the effectiveness of TCP Boston in dealing with fragmentation. This required that our measurements be unaffected by the forward error correction capability provided by AIDA, which is enabled through spatial redundancy. (2) We wanted to compare the performance of TCP Boston with that of other TCP implementations (*e.g.*, TCP Reno [18]) with and without switch-level enhancements (*e.g.*, EPD [24]). Since these other protocols do not support forward error correction, this feature of TCP Boston had to be turned off. (3) To work properly, the dynamic redundancy control mechanism of TCP Boston requires a congestion avoidance algorithm that provides accurate forecasting of network congestion. An example of such an algorithm is the one used in TCP Vegas, which provides better congestion forecast by detecting the incipient stages of congestion before losses start to accrue (rather than using the loss of segments as a signal of congestion) [10]. TCP Reno, which was the best available option in the simulation package at the time of our experiment, is reactive (rather than proactive), and thus would not bring much performance benefits when used to forecast congestion for the dynamic redundancy control mechanism in our protocol.

[10] Identifying missing cells in an ATM environment is quite simple since cells are transmitted over a virtual channel, and thus delivered to AAL5 in-order.

[11] The overhead imposed by the processing of IP packets at the AAL5 layer is minor, when compared to the switch-level modifications proposed by other techniques (like SCD and EPD), which require the switch-level software to be aware of IP packet boundaries—a serious violation of the layering principle.

2.5 An Alternate Implementation

As noted above, for the experiments presented in this paper, we have opted to disable the spatial redundancy feature of TCP Boston. In a real system, if spatial redundancy is not required by any (*e.g.* real-time) applications, then the following simpler implementation of TCP Boston is possible. Instead of using AIDA, the protocol simply divides a packet into a set of blocks[12] before the sender transmits the packet. The receiver simply generates one `ACK` per packet, with a piggy-backed bitmap to notify the sender of the identity of the missing blocks. Upon retransmission, the sender includes only those missing blocks.

The above scheme has the advantage that it does not involve any data encoding, and thus is computationally more efficient. But when compared with TCP Boston, it has the disadvantage of an increased overhead—namely an increase in the size of the block headers and the bitmaps associated with the `Acks`. Notice that the bitmap overhead could be rather large—the larger the MTU, the larger the number of bits required for a bitmap representation. For example, an MTU of 9,180 bytes requires a bitmap of 25 bytes in each `Ack`.

3 Simulation Environment

We measure the performance characteristics of our protocol under UBR service in ATM networks. We also measure the performance of Reno under the same environment to compare it with that of Boston's. Figure 6 illustrates the network topology used in the simulation.

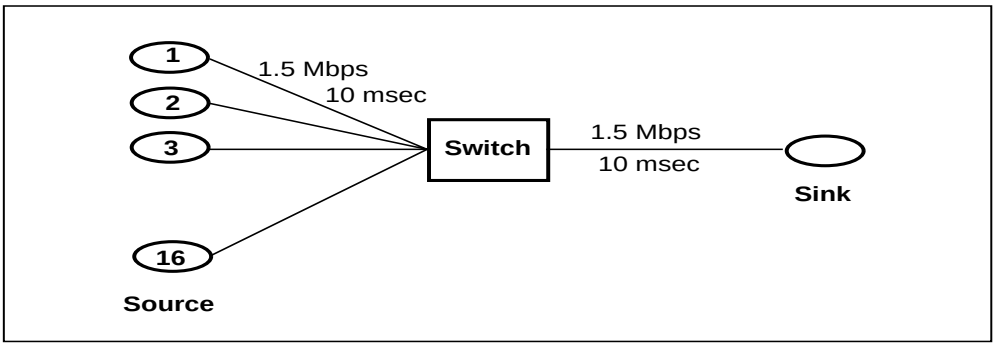

Figure 6: Configuration of simulated network.

The simulated network consists of 16 source nodes and 1 sink node, where all the nodes are connected to a single switch node. The link bandwidth in the network is set to 1.5 Mbps with propagation delay of 10 msec. The link bandwidth does not represent any particular technology. It

[12]Preferably, the size of each block (including header fields for identification purposes) would be chosen to be equal to the data portion of a cell, *i.e.*, 47 bytes as in TCP Boston.

was chosen to simulate a relatively low bandwidth-delay product (approximately 300 cells) network, without adjusting the TCP performance parameters that are sensitive to link delays. This configuration simulates a WAN environment with a radius of 3,000 km and a bottleneck link bandwidth of 1.5 Mbps.

The ATM switch is a simple, 16-port output-buffered single-stage switch [11]. When the output port is busy, a cell at the input port is queued into the output buffer of the simulated switch. When the output-buffer is full, an incoming cell destined to the output port is dropped. The output buffer is managed using FIFO scheduling, and cells in input ports are served in a round-robin fashion to ensure fairness.

In our simulator, the ATM Adaptation Layer (AAL) implements the basic functions found in AAL5, namely fragmentation and reconstruction of IP packets [1, 15]. AAL divides IP packets into 48-byte units for transmission as ATM cells, and appends 0 to 47 bytes of padding to the end of data. For simplicity, the trailer part[13], which is usually attached in the last cell by AAL5, has not been implemented in our simulator. A special flag in the cell header is used to mark the last cell in a packet. To support TCP Boston the destination AAL reconstructs a packet out of the received cells even when the resulting packet is incomplete. Incomplete packets are discarded by the destination AAL for Reno implementation.

Our simulations use a total of 16 TCP connections, each is established for one of the configuration's source-sink pairs. Each source generates an infinite stream of data bytes. Each simulation runs for 700 simulated seconds to transfer a total of 120 MB of data.

The parameters used in the simulation include the TCP packet size, the TCP window size, and the switch buffer size. Three different packet sizes were selected to reflect maximum transfer unit (MTU) of popular standards: 512 bytes for IP packets, 1,518 bytes for Ethernet, 4,352 bytes for FDDI link standards [20], and 9,180 bytes which is the recommended packet size for IP over ATM [3]. The values for the TCP window size are 8 kB, 16 kB, 32 kB, and 64 kB. Buffer sizes used for the ATM switch are 64, 256, 512, 1,000, 2,000, and 4,000 cells.

The LBNL Network Simulator (ns) [14] was used for both packet-switched and ATM network simulations. To simulate TCP Boston, we modified ns extensively to implement the three main modules (*i.e.*, the *Session Management*, *Segment Management*, and *Flow Control* modules) described in the previous section. Since ns is originally designed to support packet-switched network environments, major modifications were necessary to allow it to support ATM-like network environments. In particular, the essential functions of AAL5 were added to simulate the handling of IP packets (*i.e.*, fragmentation and reassembly of IP packets) [1, 15]. Also, the link layer of ns has been modified to include basic functions of ATM switches and virtual circuit management. While

[13]The 8-byte trailer contains the packet size and an error-checking code.

the resulting ATM simulator provides enough functionality to allow for the simulation of ATM network environments with a reasonable degree of accuracy for our purposes, it may require further enhancements to be used as a fully-functional, stand-alone ATM simulator. In addition to the above necessary modifications, the ns package has also been enhanced to allow for the gathering of additional performance statistics, such as *effective throughput* (hereinafter interchangeably termed *goodput*), *cell loss rate*, *effective packet loss rate*, and *response time*.

4 Performance Analysis

In this section we analyze the performance of TCP Boston and compare its performance with that of TCP Reno over ATM. We also present a method to estimate, utilizing the simulation results, the link bandwidth loss under Reno and Reno with EDP over multi-hop ATM networks.

4.1 Performance Characteristics of TCP Boston

We measured the performance of TCP Boston versus that of TCP Reno using four metrics: loss rate, response time, retransmission rate, and effective throughput, which are essential categories of QoS. Unless otherwise noted, each one of the graphs presented in this section portrays one of these performance metrics (on the y-axis) as a function of the switch buffer size (on the x-axis). The function is shown as a family of curves, each corresponding to one of the four different packet sizes considered.

Figure 7 shows the loss rates of Reno and Boston over an ATM network. The loss rate for Reno refers to the packet loss rate caused by cell drops at the ATM switch. In both plots, as the size of the switch buffer decreases, the loss rate gradually increases until the buffer size reaches 50 kB. From this point on, the loss rate for Reno grows exponentially toward the marginal buffer size, while the the loss rate for Boston increases at a much slower pace, except for the largest packet size.

The ratio between Reno's loss rate and Boston's loss rate increases toward the marginal buffer size. This increase is more pronounced as the packet size increases. This is because, as the packet size increases, the number of cells per packet increases, and the chance of a cell in a packet being dropped at a switch increases (as a result of fragmentation), which results in a packet loss under Reno. For small buffer sizes, this phenomenon becomes more remarkable, resulting in near 100% packet loss rate for Reno when the buffer size is smallest. On the contrary, using Boston, cells that are not dropped will be accumulated for eventual packet reconstruction at the receiver end, thus reducing the chance of repeated retransmissions. This leads to a relatively lower cell loss rate.

According to our simulation results, the ratio between Reno's loss rate and Boston's loss rate

ranges between 1.7 and 2.4, which means that about one half of the cells in a packet are dropped by switch buffer overflow on average.

The plots for retransmission rates for both Boston and Reno are almost identical to the plots for cell loss rates shown in figure 7, and thus were omitted from this paper.

Figure 8 shows the average response time of the two protocols under 64 kB TCP window size. The y-axis represents the average response time per byte (*i.e.*, the average time taken for an application at a higher layer to receive a byte).

For buffer sizes between 30 kB and 250 kB, Reno's average response time increases hyper-exponentially for the two larger packet sizes, and the ratio between Reno's response time and Boston's response time increases sharply. By comparing figure 8 with figure 7, we can easily recognize the relationship between cell loss rates and response time. On the one hand, when the buffer size decreases, the cell drop rate increases, resulting in a larger number of packets being corrupted and discarded for Reno, which in turn results in the retransmission of the same packet repeatedly, and hence sharply increasing Reno's response time. For Boston, the increased cell drop rate results in a proportional amount of additional cell transmissions (but not as many as in Reno's case), which results in a gradual increase in response time. On the other hand, when the buffer size increases, less cells are lost, increasing the probability of successful packet transfer in a minimal number of rounds, which in turn results in good response times for both protocols, with Boston edging Reno by a margin of 0.007 msec/byte on average. Notice that this difference is per byte. Thus, for large-size file transmissions, the impact on the response time may be non-negligible, even when the buffer size is moderately large.

Figure 9 is the same as Figure 8 except that its range of y-axis have been reduced to amplify the region where data points were clustered.

Figure 10 show the effective throughput (goodput) for Reno and Boston under a 64 kB TCP window size. The effective throughput refers to a throughput where only the bytes that are useful at application layer are considered.

The goodput of Reno stays low, especially for larger-size packets, throughout the entire range of buffer sizes, while that of Boston approaches the optimal level near 100 kB buffer sizes and stays almost optimal for larger buffer ranges. The low goodput of Boston under small buffer size is caused by the link idle time (since all the cells that pass through the link are counted as useful cells). The link idle time is the result of the interaction between the 16 source-sink pairs, each of which runs under TCP (Reno in our case) flow control algorithm. Recent studies on network traffic have shown that TCP can generate traffic self-similarity, which in turn causes performance degradation especially when the buffer space is limited [21] (*i.e.*, the aggregated TCP traffic can generate bursts that cause uneven flow of traffic, which eventually leads to lower bandwidth utilization under limited

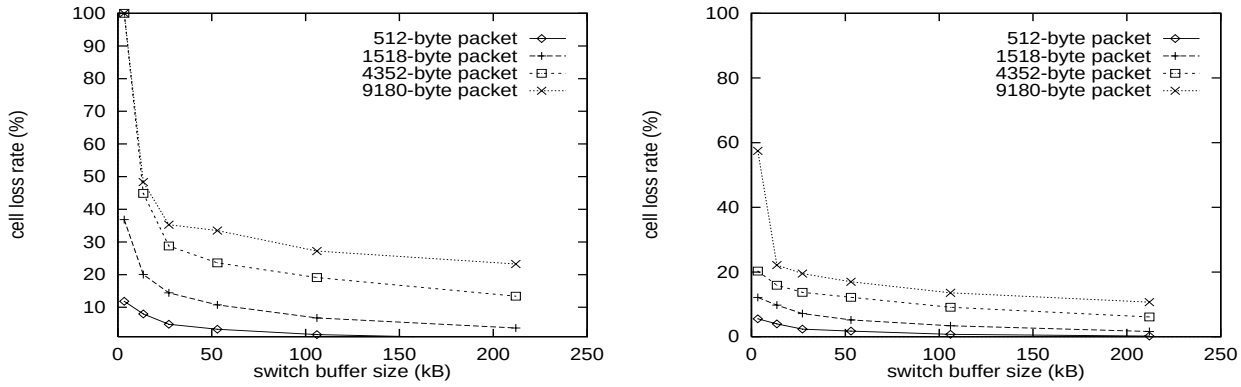

Figure 7: TCP Run: Cell loss rate of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 64 kB window size

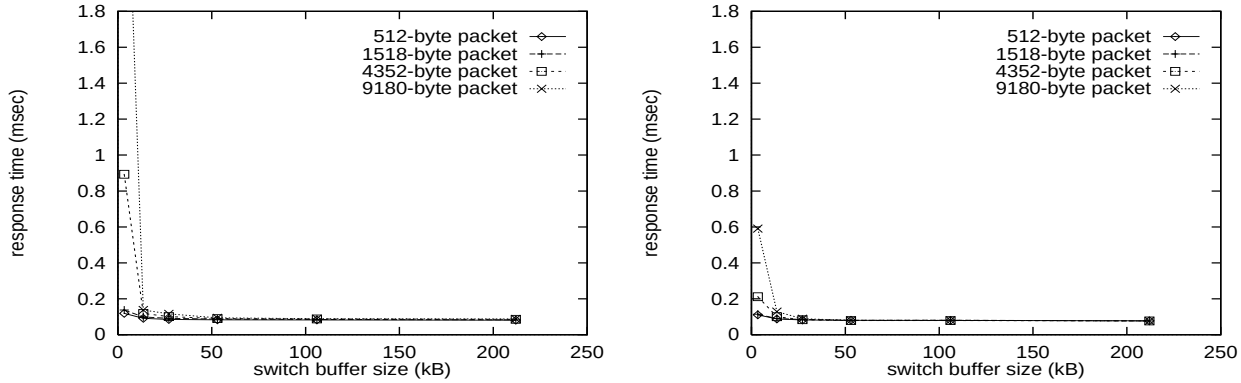

Figure 8: Response time of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 64 kB window size

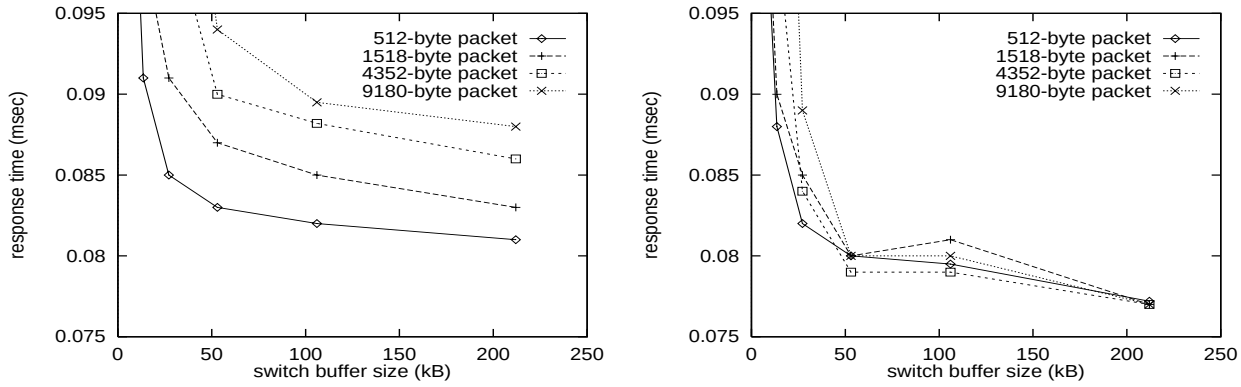

Figure 9: Response time of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 64 kB window size

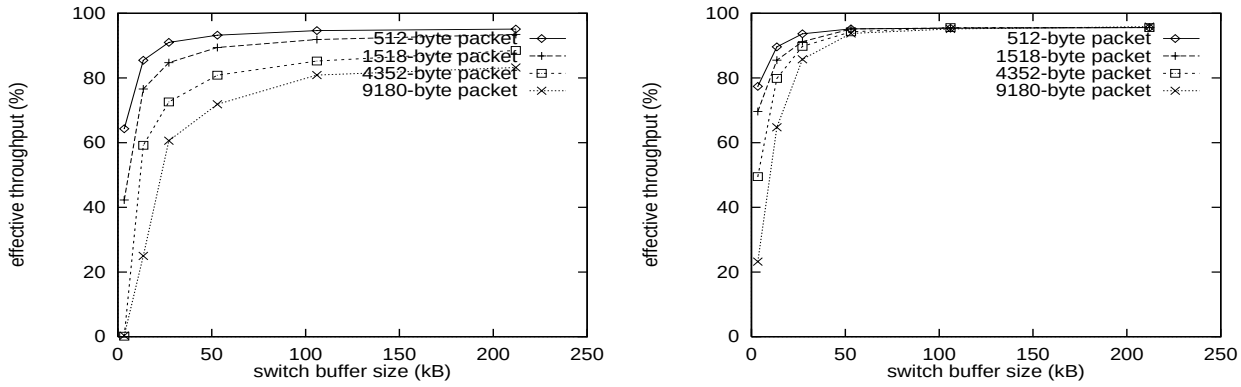

Figure 10: TCP run: Effective throughput of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 64 kB window size

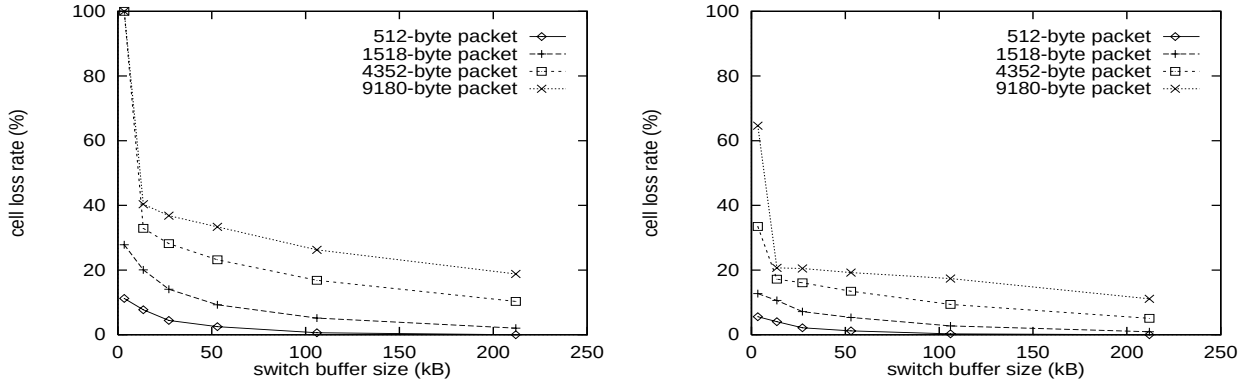

Figure 11: TCP run: Cell loss rate of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 8 kB window size

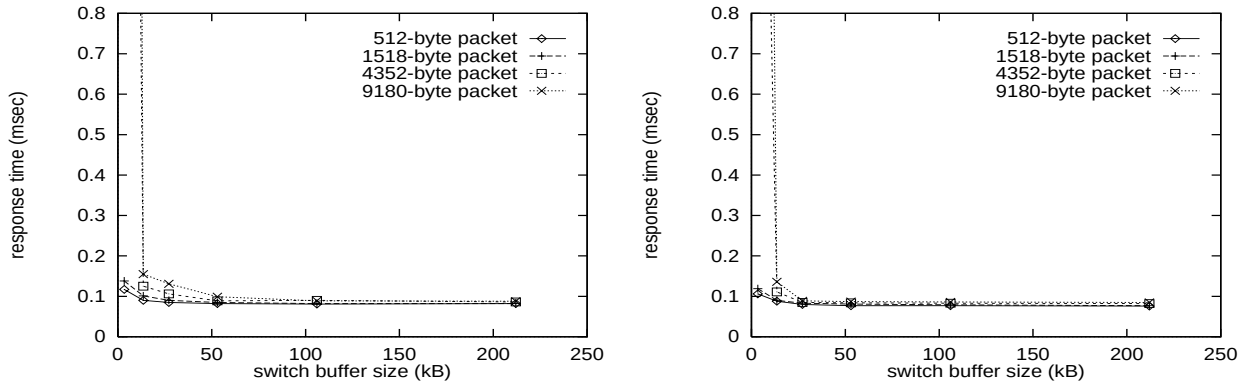

Figure 12: Response time of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 8 kB window size

buffer space). In Reno's case, the extremely low goodput at small buffer sizes is the result of the wasted bandwidth due to cells that pass through the bottleneck switch but get discarded at AAL5, as well as the link idle time that affects Boston.

So far, the results we have presented for Boston and Reno were under a TCP window size equal to 64 kB. The results for the two protocols under window sizes of 32 kB, 16 kB, and 8 kB show a gradual convergence in the performance of the two protocols as the window size decreases. Figures 11 – 14 show the impact of a small 8 kB TCP window size.

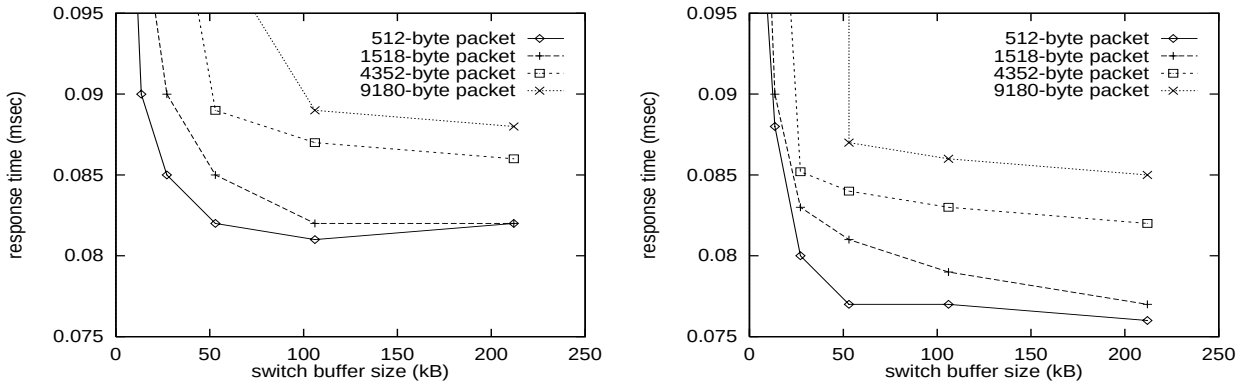

Figure 13: Response time of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 8 kB window size

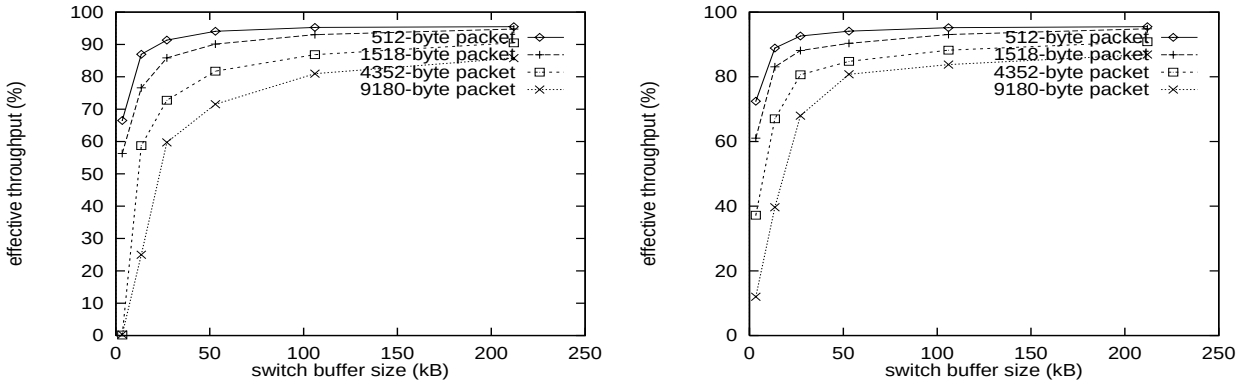

Figure 14: TCP run: Effective throughput of Reno (left) and Boston (right) over ATM as a function of switch buffer size, for 8 kB window size

4.2 Effect of TCP Boston on Flow Control

Boston's ability to accept incomplete packets (as opposed to counting such packets as lost ones) is likely to impact the flow control behavior by making it less sensitive to network congestion, and

thus more aggressive in its use of network bandwidth. To understand how this could happen, it suffices to note that using Boston, retransmitted packets are smaller (containing only the pending number of cells) and thus more likely to be delivered intact. Therefore, the likelihood that a sender utilizing TCP Boston will detect a packet loss (as a result of repeated acknowledgments received for the same packet) is reduced, which in turn, increases the probability that the sender will not decrease the congestion window (not to mention the possibility that it may even increase it). The result of this phenomenon is a minute increase in Boston's cell drop rate, compared to the actual cell loss rate of Reno. [14]

Despite Boston's aggressive use of bandwidth (and the resulting small increase in cell drop rate), it conserves the basic dynamics of the underlying TCP flow control, without causing adverse effects on the traffic flow in the network. Instead, it brings an increased effective throughput, which in turn results in an overall increase in other performance categories, such as response time, retransmission rate, and cell loss rate.

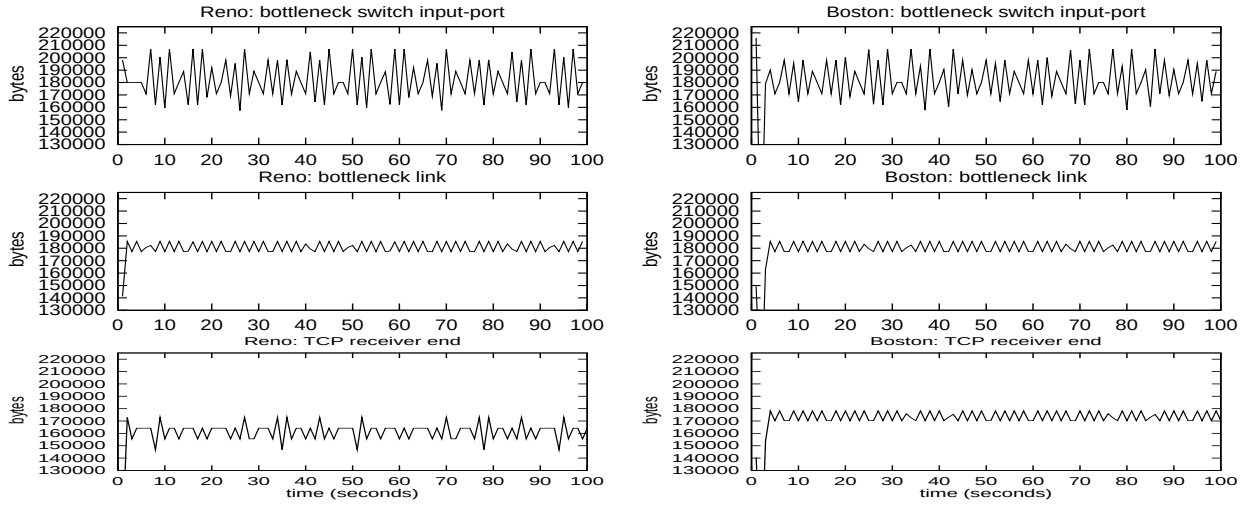

Figure 15: Traffic generated by a single TCP server, captured at input-port of the bottleneck switch (top), bottleneck link (middle), and TCP receiver end (bottom) for Reno (left) and Boston (right): 64 kB window, 27 kB (512-cell) bottleneck switch buffer, 9180-byte/packet. Each plot depicts the first 100-second period of the 700-second simulation, where the total bytes measured per second are plotted on the y axis as a function of the elapsed time (seconds).

Figure 15 shows the traffic pattern of Reno (left) and Boston (right) for the first 100 seconds in 700 simulated seconds when a single TCP server is active in the network (the total bytes measured per second are plotted on the y axis as a function of the elapsed time in seconds plotted on the x

[14] In all our simulations, Boston exhibited a maximum of 2% higher cell loss rate than the actual cell loss rate of Reno. This percentage becomes smaller as the cell loss rate increases.

axis). The bottleneck switch buffer size was set to 27 kB (512 cells)[15], which resulted in a cell drop rate of 2.3% for Boston and a packet drop rate of 4.9% for Reno. The top two graphs show the total bytes measured at the input port of the bottleneck switch (*i.e.*, total bytes generated by the source), the middle two show the total bytes passing through the bottleneck link, and the bottom two represent the total bytes accepted by the receiver-end. In the top and middle graphs, Boston (right) and Reno (left) do not show visible differences in the amount of total bytes, though Boston generated 3.2 % more traffic than Reno during the entire period of the 700 simulation seconds.

The bottom two plots show the dramatic difference between the two protocols, where Boston's acceptance and Reno's discarding of incomplete packets result in a big gap between the two protocols in the plots. In particular, Boston showed 7.83% increase in the effective throughput over Reno in this scenario.[16] The low effective throughput of Reno is the result of AAL5 cell discards and the smaller average TCP window size at the Reno's TCP source. Reno's small average window size is caused by frequent cell drops that result in full-length packet retransmissions as mentioned earlier, which leads to a higher packet drop rate, which in turn causes more frequent initialization of congestion window at the TCP source[17].

Figure 16 shows the traffic generated by a single source while 16 TCP sources are competing for bandwidth. The performance parameters used in this experiment are the same as the ones used for the simulations in figure 15. The total bytes generated by the Boston source is 6.7 MB, whereas the total generated by Reno corresponds to 5.1 MB for the 700 simulated seconds. The effective throughput achieved by Boston was 5.1%, whereas that of Reno was 3.8 %. The aggregated traffic when the 16 TCP sources are active is captured in figure 17, where Boston achieved a 78.9% effective throughput, whereas Reno achieved 59.1%.

The above experiments as well as others (not included in this paper) show that as the number of TCP sources increase, the performance gap between Boston and Reno is more pronounced. This is because resources (such as switch buffers) become limiting factors as more TCP sources compete for them. In our experiments, we have observed that while packet size, window size, and switch buffer size play important roles that affect performance, TCP Boston was consistenly able to provide a more gracefully degrading perforamance (compared to TCP Reno) when network resources become limited.

[15]Other parameters that are specific to this simulation experiment include a 64 kB TCP window size and a 9180-byte packet size.

[16]The effective throughput of Reno was 83.1%, whereas that of Boston was 89.6%.

[17]The average window size of Boston during the 700 simulation seconds was 27.7 kB (*i.e.*, 3.02 segments), and Reno maintained an average window size of 23.9 kB (*i.e.*, 2.57 segments).

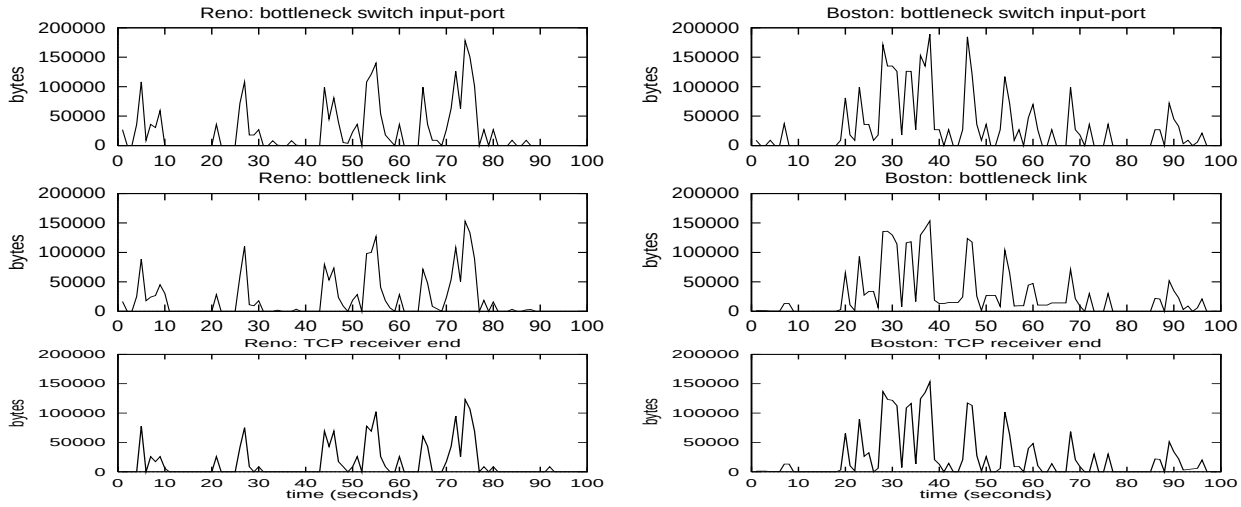

Figure 16: Traffic generated by a single TCP server with 16 TCP pairs running, captured at input-port of the bottleneck switch (top), bottleneck link (middle), and TCP receiver end (bottom) for Reno (left) and Boston (right): 64 kB window, 27 kB (512-cell) bottleneck switch buffer, 9180-byte/packet. Each plot depicts the first 100-second period of the 700-second simulation, where the total bytes measured per second are plotted on the y axis as a function of the elapsed time (seconds).

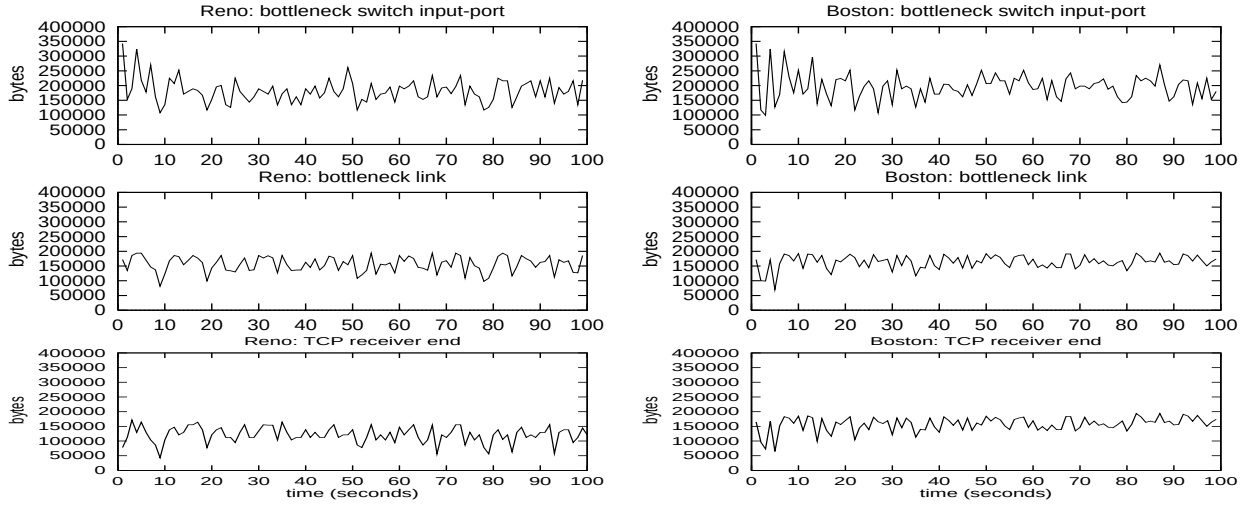

Figure 17: Traffic generated by 16 TCP servers, captured at input-port of the bottleneck switch (top), bottleneck link (middle), and TCP receiver end (bottom) for Reno (left) and Boston (right): 64 kB window, 27 kB (512-cell) bottleneck switch buffer, 9180-byte/packet. Each plot depicts the first 100-second period of the 700-second simulation, where the total bytes measured per second are plotted on the y axis as a function of elapsed time (seconds).

4.3 Performance Evaluation for Multi-hop Networks

In the remainder of this section we analyze the performance of the three techniques, Boston, Reno/EPD, and plain Reno, in terms of the bandwidth wasted as a result of cell or packet losses in a *multi-hop* network.

Rather than relying entirely on simulation (where the simulated network topology and the simulation parameters are preset), we rely on a simple, yet general formulation of the wasted bandwidth in a multi-hop network, based on measurements obtained from simulating a network with a single congested switch (similar to the one used earlier in this section, and the ones used in [24] and [9]).

The metric to be used in our analysis of the wasted bandwidth in a multi-hop network is the *Byte-Hop product* (BHops) [13]. When data (packet, cell, *etc.*) of size n bytes travels for h hops, its Byte-Hop product (*i.e.*, the bandwidth consumed by the data), is $n \times h$.

Consider a *single* source-sink TCP connection in a multi-hop network such that the number of hops between the source and the sink (*i.e.*, the length of the virtual channel) is h. Furthermore, assume that the virtual channel that connects the source and the sink traverses a congested switch after an average of δ hops.[18] For simplicity (and without loss of generality), we will assume that there is only one such congested switch along the virtual channel and that its state of congestion (*i.e.* the probability that a cell will be dropped at that switch) is independent of the TCP protocol used by the source-to-sink connection under consideration.[19] Let $(1 - p)$ denote that probability (*i.e.* p denotes the probability that a cell will not be dropped at the congested switch).

Given the above definitions and assumptions, we are now ready to define the wasted BHops caused by cell drops for each of the three protocols, assuming that the total number of packets to be transferred is t and that the number of cells per packet is m.

TCP Boston: Since cells are dropped with a probability $(1 - p)$ after an average of δ hops from the source, it follows that the total wasted BHops caused by the protocol is simply:

$$Waste_{Boston} = (1 - p) \cdot \delta \cdot m \cdot t \quad (1)$$

TCP Reno with EPD: Since all the cells belonging to a packet are discarded when any cell from that packet is dropped at the congested switch, it follows that the total wasted BHops is given

[18] Another way of stating this is that the average distance that a cell must travels until it is dropped (*en route* from source to sink) is δ.

[19] This assumption will hold if the volume of UBR traffic at the switch is much larger than the amount of traffic going through the connection under consideration. If this condition is not satisfied, then our assumption will simply favor Reno and Reno/EPD, and thus is a conservative assumption to establish Boston's superiority.

by:

$$Waste_{EPD} = (1-p^m)\cdot\delta\cdot m\cdot t \quad (2)$$

TCP Reno: Since no cells other than the dropped one(s) are discarded at the congested switch, it follows that the total wasted BHops is given by:

$$Waste_{Reno} = (1-p^m)\cdot((1-p)\cdot\delta + p\cdot h)\cdot m\cdot t \quad (3)$$

When we assume that each switch on the virtual channel between the source and sink has an equal probability of being the congested switch, the average distance that a cell travels in the path before being dropped (if it is dropped) reduces to $\frac{1}{2}h$ hops (away from the source). Substituting in the above equations we get the following ratios of wasted BHops for the three TCP protocols.

$$Waste_{Boston} : Waste_{EPD} : Waste_{Reno} = 1 : (1-p^m)\left(\frac{1}{1-p}\right) : (1-p^m)\left(\frac{1+p}{1-p}\right) \quad (4)$$

where $q = (1-p^m)$ represents the probability of a packet loss as a result of one or more cells being dropped at the congested switch. For values of $p \approx 1$ (*i.e.* small cell drop rates), equation 4 reduces to:

$$Waste_{Boston} : Waste_{EPD} : Waste_{Reno} = 1 : m : 2m \quad (5)$$

On the other hand, for smaller values of p (*i.e.* large cell drop rates), equation 4 reduces to:

$$Waste_{Boston} : Waste_{EPD} : Waste_{Reno} = 1 : \frac{1}{1-p} : \frac{1+p}{1-p} \quad (6)$$

By measuring the cell loss rate $(1-p)$ at a congested switch (either in a real system or in a simulated system), one could predict, using the above equation, the ratio between the bandwidth wasted by each one of the three protocols in a multi-hop network. For example, if the cell drop rate is measured to be 10% (*i.e.* $p = 0.9$) and $m = 10$, we get a value of $q = 0.18$. Substituting in equation 4, we get:

$$Waste_{Boston} : Waste_{EPD} : Waste_{Reno} = 1 : 6.51 : 12.38 \quad (7)$$

from which we infer that Reno wastes almost twice as much bandwidth as Reno/EDP, which in turn wastes more than six times as much bandwidth as that wasted by Boston.

Equation 4 indicates that both Reno and Reno with EPD enhancements are highly susceptible to fragmentation, since the amount of BHops they waste increases with m, the fragmentation level. On the contrary, Boston is tolerant to fragmentation and is only sensitive to the cell loss ratio

$(1-p)$. Furthermore, equation 5 shows that Reno and Reno with EPD are especially susceptible to fragmentation when the network is not congested. When the network is congested, equation 6 indicates that, as expected, all three protocols perform poorly, with Boston providing the least wasted BHops.

5 Summary and Future Work

In this paper we presented TCP Boston, a novel, fragmentation-tolerant TCP protocol, especially suited for ATM network environments. TCP Boston integrates a standard TCP/IP protocol (such as Reno or Vegas) with a powerful encoding mechanism based on AIDA (an adaptive version of Rabin's IDA dispersal and reconstruction algorithms [23]). We have presented our implementation of TCP Boston and have shown its performance superiority when compared to TCP techniques that are more vulnerable to fragmentation, namely TCP Reno and TCP Reno with EPD switch-level enhancements. Our performance evaluation was done in two steps. The first provided us with basic simulation results for a congested ATM switch, whereas the second allowed us to analytically predict the wasted bandwidth for TCP Boston, Reno, and Reno with EPD in a multi-hop network.

We are currently working on an improved version of TCP Boston, which does not require our current (minor) modification of the AAL5 layer. Also, we are working on improving our implementation of the IDA dispersal and retrieval algorithms to reduce (or eliminate) the buffer space requirement at the sender and receiver ends. This is particularly important in high bandwidth-delay product network environments. Finally, we are looking into the issues involved in providing dynamic redundancy control mechanisms in TCP Boston by incorporating better congestion forecasting capabilities such as those found in TCP Vegas. This would allow TCP Boston to support real-time systems through the careful tradeoff of spatial redundancy for timeliness. Our future work includes deploying TCP Boston on an experimental basis on a real ATM switching environment, which is currently being installed in our department.

References

[1] ANSI. AAL5 – A New High Speed Data Transfer AAL. In *ANSI T1S1.5 91-449*. November 1991.

[2] G. Armitage and K. Adams. Packet Reassembly During Cell Loss. *IEEE Network Mag.*, 7(5):26–34, September 1993.

[3] R. Atkinson. Default IP MTU for use over ATM AAL5. In *RFC 1626*. May 1994.

[4] Azer Bestavros. IDA-based disk arrays. Technical Memorandum 45312-890707-01TM, AT&T, Bell Laboratories, Department 45312, Holmdel, NJ, July 1989.

[5] Azer Bestavros. SETH: A VLSI chip for the real-time information dispersal and retrieval for security and fault-tolerance. In *Proceedings of ICPP'90, The 1990 International Conference on Parallel Processing*, Chicago, Illinois, August 1990.

[6] Azer Bestavros. An adaptive information dispersal algorithm for time-critical reliable communication. In Ivan Frisch, Manu Malek, and Shivendra Panwar, editors, *Network Management and Control, Volume II*. Plenum Publishing Corporation, New York, New York, 1994.

[7] Azer Bestavros. AIDA-based real-time fault-tolerant broadcast disks. In *Proceedings of RTAS'96: The 1996 IEEE Real-Time Technology and Applications Symposium*, Boston, Massachusetts, May 1996.

[8] A. Bianco. Performance of the TCP Protocol over ATM Networks. In *Proceedingds of the 3rd International Conference on Computer Communications and Networks*, pages 170–177, San Francisco, CA, September 1994.

[9] Ernst Biersack. Performance Evaluation of Forward Error Correction in ATM Networks. *Comm. of ACM*, pages 248–257, August 1992.

[10] Lawrence Brakmo, Sean O'Maley, and Larry Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. Technical Report TR 94 04, The University of Arizona Computer Science Department, Tucson, AZ 85721, Febrary 1994.

[11] Thomas Chen and Stephen Liu. *ATM Switching System*. Artech House, Inc., 685 Canton St., Norwood, Ma 02062, 1995.

[12] Douglass E. Comer. *Internetworking with TCP/IP*, volume 1. Prentice Hall Inc., Englewood Cliffs, NJ, 1995.

[13] Petter B. Danzig, Richard S. Hall, and Michael F. Schwartz. A Case for Caching File Objects Inside Interneworks. Technical Report CU-CS-642-93, Department of Computer Science, University of Colorado – Boulder, March 1993.

[14] Sally Floyd. Simulator Tests. Available in `ftp://ftp.ee.lbl.gov/papers/simtests.ps.Z`. ns is available at `http://www-nrg.ee.lbl.gov/nrg`., July 1995.

[15] ATM Forum. *ATM User-Network Interface Specification*. Pretice Hall, Inc, Englewood Cliffs, New Jersey 07632, 1993.

[16] Garth A. Gibson and David A. Patterson. Designing disk arrays for high data reliability. *Journal of Parallel and Distributed Computing*, 17(1-2):4–27, January/February 1992.

[17] M. Hassan. Impact of Cell Loss on the Efficiency of TCP/IP over ATM. In *Proceedings of the 3rd International Conference on Computer Communications and Networks*, pages 165–169, San Francisco, CA, September 1994.

[18] V. Jacobson. Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno. In *Proceedings of the British Columbia Internet Engineering Task Force*, July 1990.

[19] Yuh-Dauh Lyuu. Fast fault-tolerant parallel communication and on-line maintenance using information dispersal. Technical Report TR-19-1989, Harvard University, Cambridge, Massachusetts, October 1989.

[20] Sonu Mirchandani and Raman Khanna, editors. *FDDI Technology and Applications*. John Wiley & Sons, Inc., 1993.

[21] Kihong Park, Gitae Kim, and Mark E. Crovella. The Effects of Traffic Self-Similarity on TCP Performance. Technical report, Boston University Computer Science Department, 1996.

[22] J. Postel. Transmission Control Protocol. In *RFC 793*. September 1981.

[23] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

[24] A. Romanow and S. Floyd. Dynamics of TCP Traffic over ATM Networks. *IEEE Journal on Selected Areas in Communication*, 13(4):633–641, May 1995.