

Implementation and Performance Evaluation of TCP Boston A Fragmentation-tolerant TCP Protocol for ATM Networks *

Azer Bestavros Gitae Kim
best@cs.bu.edu kgt.jan@cs.bu.edu
Computer Science Department
Boston University
Boston, MA 02215

ABSTRACT:

In this paper, we overview the implementation of TCP Boston—a novel fragmentation-tolerant transport protocol, especially suited for ATM's 53-byte cell-oriented switching architecture. TCP Boston integrates a standard TCP/IP protocol, such as Reno or Vegas, with a powerful redundancy control mechanism based on AIDA—an adaptive version of Rabin's IDA dispersal and reconstruction algorithms. Our results 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).

1 Introduction

In the last few years, the Transmission Control Protocol (TCP) [15]—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. However, the introduction of the Asynchronous Transfer Mode (ATM) technology and attempts to integrate that technology with IP protocols have raised many questions regarding the effectiveness of using TCP/IP over ATM networks [7, 12, 17].

The poor performance of TCP over ATMs is mainly due to *packet fragmentation*, which occurs when an IP packet flows into an ATM virtual circuit through the AAL5 (ATM Adaptation Layer 5)—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, 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.

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] (called Par-

tial Packet Discard (PPD) in [17]) and the Early Packet Discard (EPD) [17]. 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 [17].

The simulations described in [17] 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 using EPD approaches that of TCP/IP in the absence of fragmentation. However, these results were obtained for a network consisting of a single ATM switch. However, for 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, these techniques are not likely to scale for large, multi-hop ATM networks. In this paper, we present the implementation and performance evaluation of TCP Boston, a novel transport protocol 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.

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

2 TCP Boston: Principles and Implementation

AIDA Characteristics: 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 (IDA) of Michael O. Rabin [16]. 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 cells in such a way that recombining *any* m of these cells, $m \leq N$, using IDA's *reconstruction operation*, is sufficient to retrieve S . Figure 1 illustrates the dispersal, transmission, and reconstruction of an object using IDA. The dispersal and reconstruction operations are simple linear transformations using *irreducible polynomial arithmetic*, which can be performed in real-time [4].

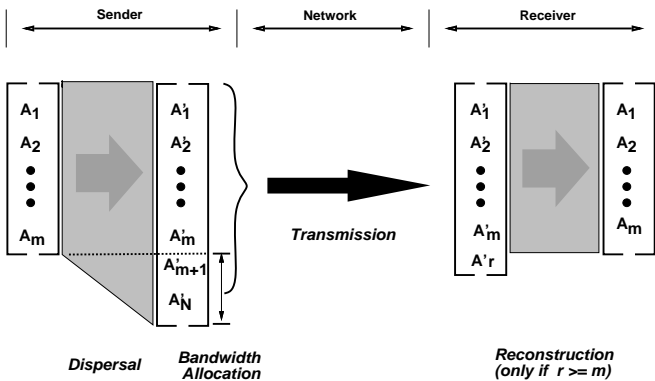


Figure 1. AIDA dispersal and reconstruction

In IDA, there is *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) [5]. Using AIDA, a *bandwidth allocation* operation is inserted after the dispersal operation but *prior* to transmission. 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).

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 incorporation of AIDA in a TCP protocol allows us to strike a critical balance between these two alternatives. To explain how this could be done, consider the following scenario. The sender disperses an outgoing m -cell segment (packet) into N cells, but sends a packet of only m of these cells to the receiver, where $N \gg m$. 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 acknowl-

edge 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.

TCP Boston: Implementation TCP Boston can be implemented over both ATM and packet-switched networks for reliable transfer of data. The protocol consists of three top-level components: a *Session Manager*, a *Segment Manager*, and a *Flow Manager*. Figure 2 details the interactions among these modules, summarizing the implementation semantics during packet transmissions and receptions.

Session Manager: 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 is required by the receiver for reconstruction purposes, such as the value of m for example), is piggy-backed onto the protocol packets during the three-way handshaking at the connection establishment phase.

Segment Manager: Three functions that are unique to TCP Boston are: (1) segment *encoding* (at the sender), (2) segment *reconstruction* (at the receiver), and (3) *redundancy control* (at the sender).

Segment encoding: Before transmitting a data block (segment) of size b bytes, the encoder divides the data block into m cells of size c , where $m = b/c$ bytes. Then, the m cells are processed using IDA to yield N cells for some $N \gg m$. 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 header is required for cell identification, which is 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.

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 to that packet. If it did, it reconstructs the original segment using the proper IDA reconstruction matrix transformation. When a partial packet is received, it keeps the received cells in its buffer for later reconstruction.

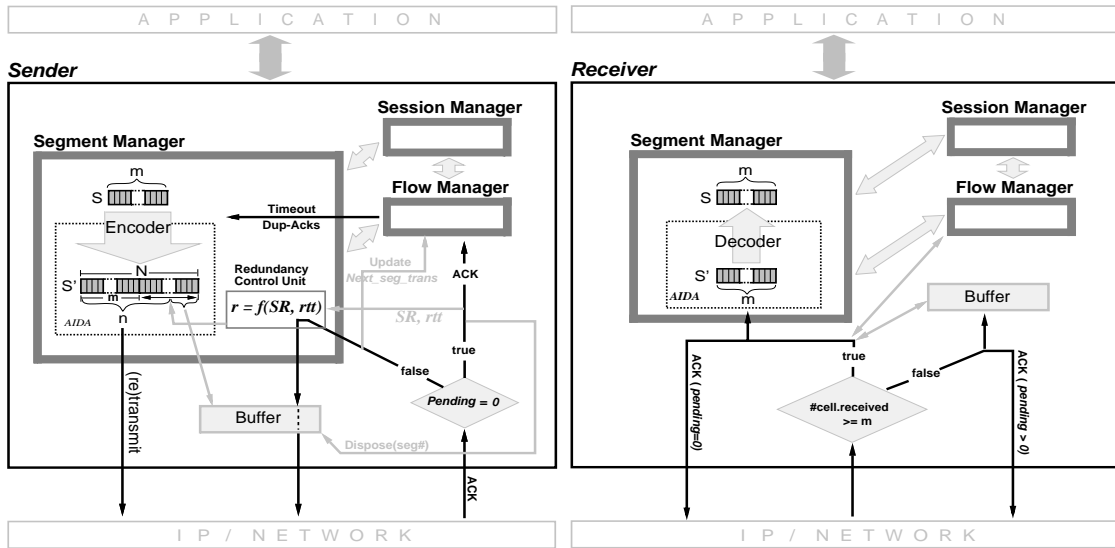


Figure 2. TCP Boston: Top-level modules, Session Manager, Segment Manger, and Flow Manager, and their interactions during packet transmissions and receptions for sender (left) and receiver (right).

Redundancy Control: This module closely interacts with TCP's Flow Manager to estimate the redundancy rate, γ , at the time of each packet (re)transmission. Feedback information such as Round Trip Time (rtt) and Success Rate (SR) in the acknowledgment (hereinafter referred to as an ACK) can be used as network congestion information to compute γ . Figure 3 shows the transmission window managed by this module. Prior to a packet (re)transmission, the module estimates γ ($0 < \gamma < 1.0$), and adds ($\gamma \times original_packet_size$) data units (*i.e.*, cells) to the original packet. For example, when an m -cell segment is to be transmitted, it computes γ to decide n , the *transmission window size* ($= m + m \times \gamma$), which represents the size of the packet that will actually be transmitted.

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).

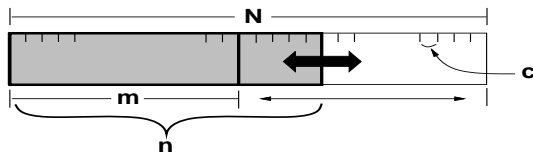


Figure 3. Transmission Window managed by Redundancy Manager

Flow manager: The main function of this module is flow and transmission control. To that end, any feedback-based TCP flow control algorithm (*e.g.*, Tahoe, Reno, or Vegas) can be used with

TCP Boston, with a minor modification to handle the revised feedback mechanism of TCP Boston. To manage its unique feedback scheme effectively, the data structure for ACK needs to be augmented with an integer field, *Pending*, to record the number of pending cells. The Flow Manager's end-to-end feedback and reaction scheme includes a set of provisions for handling three possible cases of packet receptions at the destination: (1) a complete packet has been received, (2) a partial packet has been received (*i.e.* some of the packet's cells were lost), or (3) an entire packet is missing (*i.e.*, all cells were lost). The procedures followed by the receiver and sender for each of the above cases are described below:

Upon a complete packet reception: The receiver and sender follow the feedback and reaction scenarios of conventional flow control scheme, *i.e.*, receiver sends an ACK, signaling a normal reception of a packet and sender adjust its congestion window accordingly and transmits next batch of segments. In addition, receiver resets *Pending* in the ACK to signify a complete packet reception. Also, if the complete packet is the result of multiple rounds of receptions, the receiver disposes any space used for the packet from its buffer.

Upon a partial packet reception: The receiver keeps the received cells in its buffer for later reconstruction, finds out the number of missing cells, enters that number in the *Pending* field of the ACK, which is transmitted immediately. 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 immediately (*i.e.*, fast transmission).¹ Upon receiving an ACK with a positive value for *Pending*, the sender finds the cells from its buffer that belong to the segment specified by the *Segment#* field, and *immediately* transmits the pending number of cells ($= Pending$) plus redundant cells determined by the redundancy control module (*i.e.*, fast transmission). At the

¹ In non-TCP Boston fast transmission is not possible since there is no concept of partial packet reception, and thus no ACKs that signal partial packet reception.

same time, it updates the protocol variable, *Next_seg_trans*, to prevent a duplicate retransmission of the same segment in the future (*i.e.*, TIMEOUT or three successive dup-ACKs). Notice that the partial delivery of a packet does not result in updating the *Last_segment#* for the receiver's TCP window manager.² 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 fast-transmit the missing cells.

Upon an entire packet miss: The receiver and sender follow the conventional flow control semantics, *i.e.*, the receiver does not send an ACK for the missing packet. Instead, it sends duplicate ACKs on subsequent complete packet receptions. At retransmission time (due to TIMEOUT or three dup-ACKs) the sender encodes the original segment again, instead of using the cells in the buffer that were already encoded.

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 [13]) with and without switch-level enhancements (*e.g.*, EPD [17]). 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 (*e.g.*, TCP Vegas [8]). 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.

3 Performance Evaluation

Simulation Environment: We performed a host of simulation experiments to evaluate the performance of TCP Boston against that of TCP Reno, under UBR service in ATM networks. Due to space limitations, we only present the baseline simulation results. For a more elaborate treatment, we refer the reader to [6].

The simulated network consists of a single ATM switch connecting 16 source nodes and 1 sink node. The link bandwidth in the network is set to 1.5 Mbps with propagation delay of 10 msec. 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 [9]. 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, 11]. AAL divides IP packets into 48-byte units for trans-

mission as ATM cells, and appends 0 to 47 bytes of padding to the end of data.

Each simulation uses 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 an average 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. Four 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 [14], 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) [10] was used for both packet-switched and ATM network simulations. To simulate TCP Boston, we modified ns extensively to implement the three main modules described in the previous section.

Performance Characteristics of TCP Boston: The performance of TCP Boston versus that of TCP Reno was measured in four metrics: loss rate, response time, retransmission rate, and effective throughput. 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 4(a) 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.

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.

Figure 4(b) shows 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. In Reno's case, the extremely low goodput at small buffer sizes turned out to be 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.

²This enables the receiver to send duplicate ACKs to signal congestion to the sender.

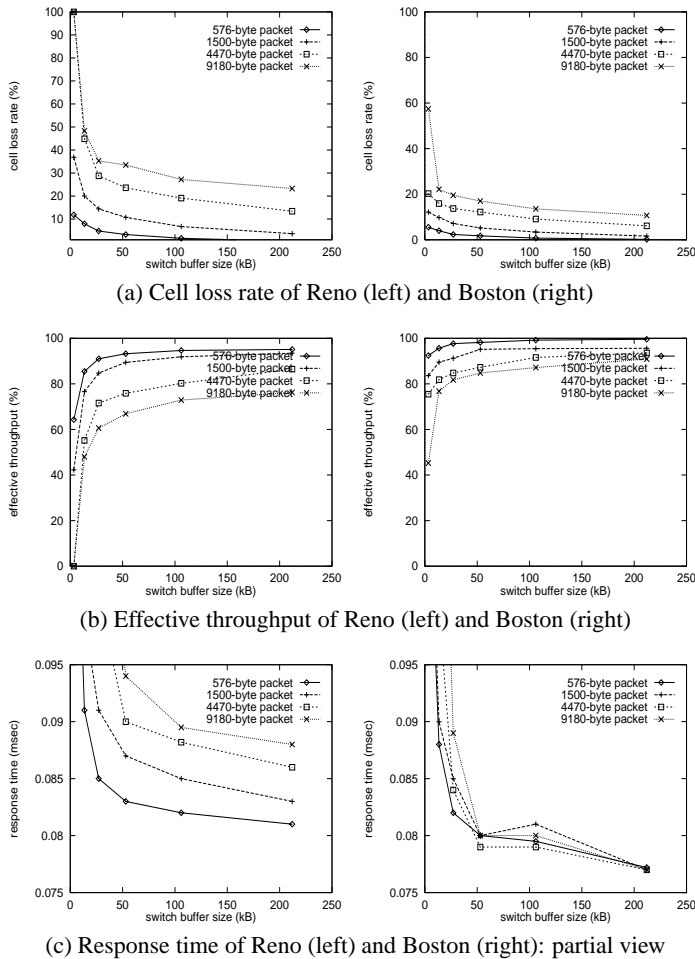


Figure 4. TCP run for an 64 kB window size

Figure 4(c) compares the average response time of the two protocols under 64 kB TCP window size. The response time represents the average time for an application at a higher layer to receive a byte. For buffer sizes between 3.5 kB and 13.5 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. As the bottleneck 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, as 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 7 μ sec/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.

We have conducted a host of experiments to further characterize the performance of TCP Boston under various conditions. The results of these experiments are presented in [6].

4. Summary

TCP Boston integrates a standard TCP/IP protocol, such as Reno [13] or Vegas [8], with a powerful redundancy control mechanism based on AIDA encoding [5] to turn the fragmentation of IP packets [17] into an advantage—thus enhancing the performance of TCP/IP in general and its performance in ATM environments in particular. In this paper, we presented the implementation of TCP Boston and briefly summarized its performance characteristics.

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] A. 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.
- [5] A. Bestavros. An adaptive information dispersal algorithm for time-critical reliable communication. In I. Frisch, M. Malek, and S. Panwar, editors, *Network Management and Control, Volume II*. Plenum Publishing Corporation, New York, New York, 1994.
- [6] A. Bestavros and G. Kim. TCP Boston: A Fragmentation-tolerant TCP Protocol for ATM Networks. In *Proceedings of Infocom'97: The IEEE International Conference on Computer Communication*, Kobe, Japan, April 1997. To appear.
- [7] A. Bianco. Performance of the TCP Protocol over ATM Networks. In *Proceedings of the 3rd International Conference on Computer Communications and Networks*, pages 170–177, San Francisco, CA, September 1994.
- [8] L. Brakmo, S. O'Maley, and L. 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, February 1994.
- [9] T. Chen and S. Liu. *ATM Switching System*. Artech House, Inc., 685 Canton St., Norwood, Ma 02062, 1995.
- [10] S. Floyd. Simulator Tests. Available from <ftp://ftp.ee.lbl.gov/papers/simtests.ps.Z>. ns(v1.0b4) is available at <http://www-nrg.ee.lbl.gov/nrg/>, July 1995.
- [11] A. Forum. *ATM User-Network Interface Specification*. Prentice Hall, Inc, Englewood Cliffs, New Jersey 07632, 1993.
- [12] 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.
- [13] 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.
- [14] S. Mirchandani and R. Khanna, editors. *FDDI Technology and Applications*. John Wiley & Sons, Inc., 1993.
- [15] J. Postel. Transmission Control Protocol. In *RFC 793*. September 1981.
- [16] M. 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.
- [17] 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.