Proceedings of Infocom'97: The IEEE Conference on Computer Communications, April 1997, Japan.

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

Abstract

We propose 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.

1. Introduction

The flexibility and popularity of TCP/IP[20, 19] 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 [11]. However, recent studies [7, 15, 22] 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*, which occurs when an IP packet flows into an ATM virtual circuit through the AAL5 (ATM Adaptation Layer 5). 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)[1] [2] and the Early Packet Discard (EPD) [22]. 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 earlier cells still in the switch buffer. Notice that both SCD and EPD require modifications to switch-level software to be aware of IP packet boundaries—a violation of the layering principle that was deemed unavoidable for performance purposes in [22].

The simulation results described in [22] show that both SCD and EPD improve the effective throughput of TCP/IP over a single ATM switch. For realistic, multi-hop ATM networks the cumulative wasted bandwidth may be large, and the impact of the ensuing packet losses on the performance of TCP is likely to be severe.[2] 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 preced-

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

[1] Also called Partial Packet Discard (PPD) in [22].
[2] Analytical comparisons for multi-hop networks appear in [6].

ing 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 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 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.

2. 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 [21]. 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. The dispersal and reconstruction operations are simple linear transformations using *irreducible polynomial arithmetic* and can be performed in real-time.[3] The dispersal operation amounts to a matrix multiplication 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. 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.

[3]For more details, we refer the reader to [21, 4].

Several *redundancy-injecting* protocols have been suggested in the literature, whereby redundancy is injected in the form of parity, which is only used for error detection and/or correction purposes [14]. The IDA approach is radically different in that redundancy is added *uniformly*; 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 as shown in figure 1. 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).

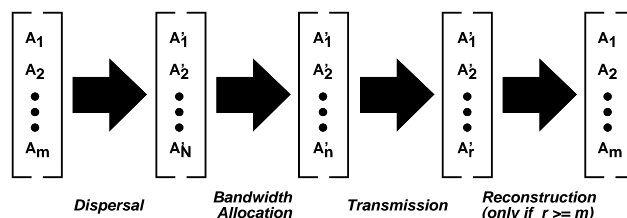


Figure 1. AIDA dispersal and reconstruction

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 [8], 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[4] could be quite valuable for real-time communication as reported in [5].

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 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.

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

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 2 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.

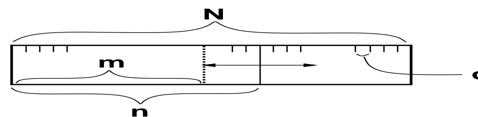


Figure 2. AIDA Transmission Window

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).

3. TCP Boston: Overview and Implementation

The purpose TCP Boston 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 53-byte cells, 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*.

Session Management: The protocol manages a TCP session in three phases: a *connection establishment* phase, a *data transfer* phase, and a *termination* phase. The pur-

pose 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.[5]

Segment Management: Processes for (1) segment encoding and (2) segment reconstruction are unique to TCP Boston.

At the source, segment encoding works as follows: 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 $N >> m$. 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).

At the sink, segment reconstruction works as follows: 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 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: 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.[6] 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.

In our current implementation, the protocol is composed of three 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 3 depicts the configuration and interaction of the three modules for both the sender and the receiver.

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.[7] 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.

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 [16]) with and without switch-level enhancements (*e.g.*, EPD [22]). 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) [9]. 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.

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

[6] This enables the receiver to send duplicate ACKs to signal a packet drop to the sender.

[7] 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.

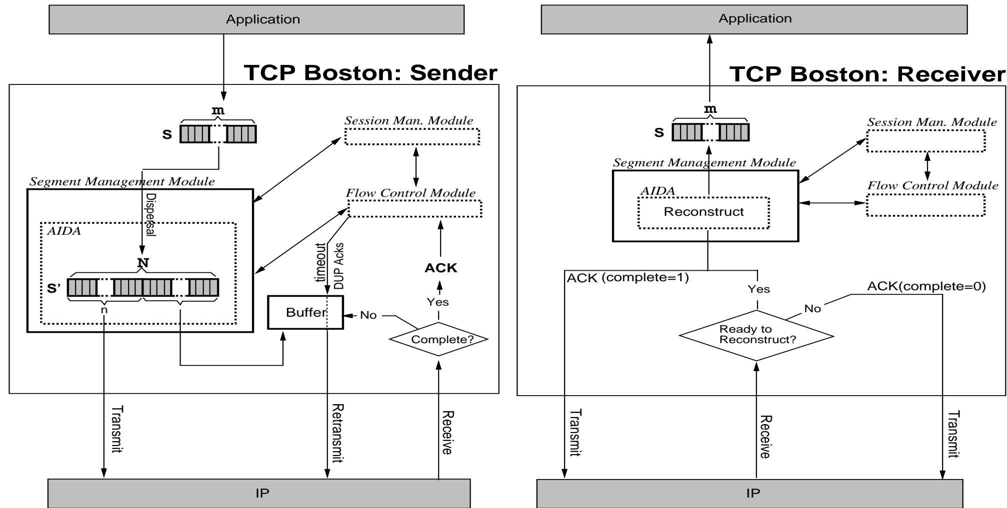


Figure 3. TCP Boston: Protocol Outline for Sender (left) and Receiver (right)

4. TCP Boston: Performance Evaluation

In this section we present the results of our experimental evaluation of TCP Boston. For a more detailed treatment, we refer the reader to [6].

4.1. Simulation Environment

As shown in Figure 4, 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 is set to 1.5 Mbps with propagation delay of 10 msec. The link bandwidth does not represent any particular technology. It was chosen to simulate a relatively low bandwidth-delay product network. This configuration simulates a WAN environment with a radius of 3,000 km and a bottleneck link bandwidth of 1.5 Mbps.

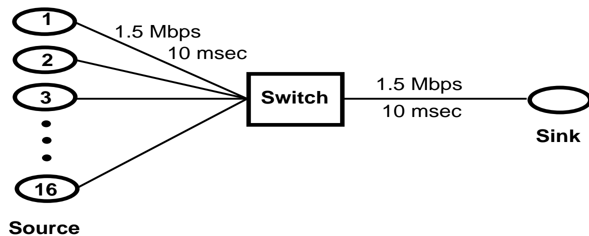


Figure 4. Configuration of simulated network.

The ATM switch is a simple, 16-port output-buffered single-stage switch [10]. 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, 13]. 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. 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: 576 bytes for IP packets, 1,500 bytes for Ethernet, 4,470 bytes for FDDI link standards [17], and 9,180 bytes which is the recommended packet size for IP over ATM [3]. The values for the maximum TCP window size are 8, 16, 32, and 64 kB. Buffer sizes used for the ATM switch are 64, 256, 512, 1,000, 2,000, and 4,000 cells.

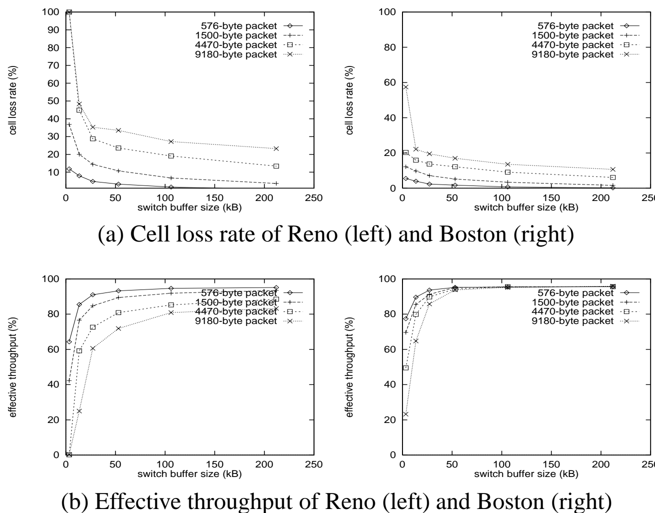
The LBNL Network Simulator (ns) [12] was used for both packet-switched and ATM network simulations. To simulate TCP Boston, we modified ns extensively to implement the three main modules described in the previous section. Since ns is originally designed to support packet-switched network environments, 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, 13]. Also, the link

layer of ns has been modified to include basic functions of ATM switches and virtual circuit management. 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*.

4.2. 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. 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 5(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. 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.



(a) Cell loss rate of Reno (left) and Boston (right)

(b) Effective throughput of Reno (left) and Boston (right)

Figure 5. TCP run for an 64 kB window 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 5(a), and thus were omitted from this paper. Figure 5(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. 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 augment traffic self-similarity, which causes performance degradation especially when the buffer space is limited [18]. 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.

Table 1 compares the average response time of the two protocols under 64 kB TCP window size. The response time in the table represents an average time taken for an application at a higher layer to receive a byte.

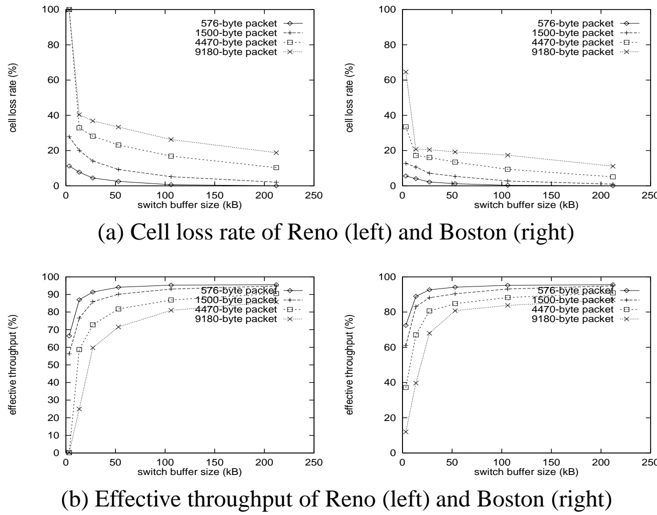
For buffer sizes between 3.5 kB and 13.5 kB, Reno's average response time increases hyper-exponentially for the two larger packet sizes, and the ratio between Reno's response time and Boston's response time increases sharply. As the bottleneck buffer size decreases, the cell drop rate increases, resulting 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 μsec/byte on average. Notice

Packet Size (Byte)	Buffer Size (kB)											
	3.5		13.5		27		54		104		208	
	Bos	Ren	Bos	Ren	Bos	Ren	Bos	Ren	Bos	Ren	Bos	Ren
512	111	119	88	91	82	85	80	83	79.5	82	77.2	81
1,518	114	138	90	99	85	91	80	87	79	85	77	83
4,352	212	893	100	118	84	101	79	90	79	88.2	77	86
9,180	591	3,125	129	138	89	116	80	94	79	89.5	77	88

Table 1. Response times of Boston and Reno over ATM for 64 kB window size

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.

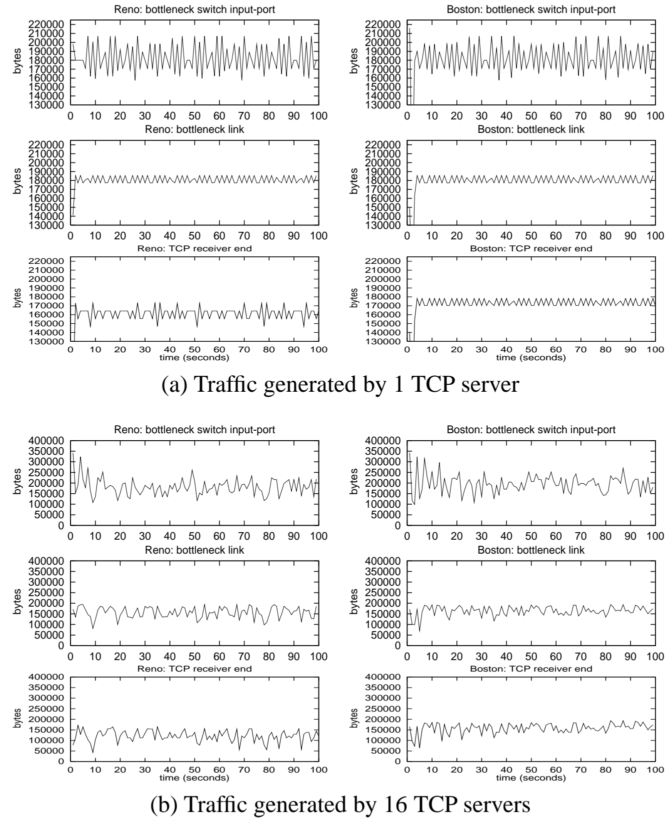
So far, the results we have presented for Boston and Reno were under a TCP window size equal to 64 kB. The results for the two protocols under window sizes of 32 kB, 16 kB, and 8 kB show a gradual convergence in the performance of the two protocols as the window size decreases. Figure 6 shows the impact of a small 8 kB TCP window size.



(a) Cell loss rate of Reno (left) and Boston (right)

(b) Effective throughput of Reno (left) and Boston (right)

Figure 6. TCP Run for an 8 kB window size



(a) Traffic generated by 1 TCP server

(b) Traffic generated by 16 TCP servers

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 at the 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.

Figure 7. Traffic generation

4.3. Effect of TCP Boston on Flow Control

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

Despite Boston's aggressive use of bandwidth, our experiments confirmed that it conserves the basic dynamics of the underlying TCP flow control. Figure 7(a) shows the traffic pattern of Reno (left) and Boston (right) for the first 100

seconds in 700 simulated seconds when 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 axis). The bottleneck switch buffer size was set to 27 kB (512 cells), 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.[8] 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[9]. The aggregated traffic when the 16 TCP sources are competing is captured in figure 7(b), 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 consistently able to provide a more gracefully degrading performance (compared to TCP Reno) when network resources become limited.

5. Summary

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 [21]). We have presented our implementation of TCP Boston and have shown its performance superiority through simulation method, when compared to TCP techniques that are more vulnerable to fragmentation, namely TCP Reno and TCP Reno with EPD switch-level enhancements.

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

[9] 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).

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 Frisch, Malek, and Panwar, editors, *Network Management and Control, Volume II*. Plenum Publishing Co., NY, NY, 1994.

[6] A. Bestavros and G. Kim. TCP Boston: A Fragmentation-tolerant TCP Protocol for ATM Networks. Technical Report BUCS-TR-96-014, Boston University, Computer Science Department, July 1996.

[7] 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.

[8] E. Biersack. Performance Evaluation of Forward Error Correction in ATM Networks. *CACM*, pages 248–257, Aug 1992.

[9] 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, Febrary 1994.

[10] T. Chen and S. Liu. *ATM Switching System*. Artech House, Inc., 685 Canton St., Norwood, MA 02062, 1995.

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

[12] 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.

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

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

[15] 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.

[16] 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.

[17] S. Mirchandani and R. Khanna, editors. *FDDI Technology and Applications*. John Wiley & Sons, Inc., 1993.

[18] K. Park, G. Kim, and M. E. Crovella. The Effects of Traffic Self-Similarity on TCP Performance. Technical report, Boston University Computer Science Department, 1996.

[19] J. Postel. Internet Protocol. In *RFC 791*. September 1981.

[20] J. Postel. TCP. In *RFC 793*. September 1981.

[21] 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.

[22] 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.