Tuning TCP Performance in Beowulf Computers

Walter B. Ligon III, Scott P. McMillan, and Robert B. Ross

August 2, 1999

Abstract

Beowulf workstations have become a popular choice for high-end computing in a number of application domains. One of the key building blocks of parallel applications on Beowulf workstations are message passing libraries which utilize the Transmission Control Protocol (TCP) for cluster communications. As cluster network fabrics achieve higher bandwidths and shorter latencies, the TCP/IP software and protocols become the common case "bottleneck" for low latency communications. This bottleneck limits the applicability of Beowulf Workstations to very coarse grain applications.

Our research in this bottleneck area focuses on a "configurable" TCP protocol, which will allow the user to determine the optimal TCP transmission and retransmission parameters needed for specific applications. In this paper we discuss a system for configuring TCP to behave more appropriately in the Beowulf environment and present results indicating expected performance improvements when using this system.

1 Introduction

In recent years, clusters of workstations have become an inexpensive alternative to supercomputers in specific applications. Traditionally, slow network hardware has limited the applicability of clusters to coarse grain or embarrassingly parallel applications. Today, advancements in network hardware have provided clusters with very high bandwidth and low latency connections. Now, the software and protocols driving the network hardware have surfaced as the new common case bottleneck.

In clusters, the primary protocol used in communication has been the traditional Transmission Control Protocol / Internet Protocol (TCP/IP). The implementation of the TCP software and transmission protocols induce two kinds of latencies limiting the range of applications that can benefit from clusters: software and protocol. Generally, kernels implement TCP/IP in layers, leading to software induced latencies caused by passing data through multiple layers prior to reaching the network wire. The transmission protocols used in TCP lead to protocol induced latencies created by the inherent generality of the TCP/IP algorithms. This thesis attempts to tackle this second kind of latency by introducing a configurable transmission protocol concept that allows transmission algorithms to be tuned for specific applications and/or hardware.

In Section 2 we will cover the relevant background material for this work, including important TCP algorithms, Beowulf computing, and some preliminary work in this area. The configurable TCP options and an overview of the implementation are described in Section 3. Section 4 details the test system and the tests performed in evaluating the configuration options. Conclusions are drawn in Section 5 and possible future work is outlined.

2 Background

In order to effectively tune the TCP protocol in this environment, it is important to first understand TCP and its key algorithms. These algorithms must then be appropriately matched to the peculiarities of the Beowulf environment.

2.1 TCP/IP

The Internet Protocols have been established to standardize communication over the Internet. These standards and recommendations come in the form of documents published by the Internet Engineering Task Force (IETF) called Request for Comments (RFCs). Of all the Internet Protocols, TCP in particular has encouraged Internet growth by providing a reliable protocol that adjusts dynamically to network traffic conditions.

A number of RFCs cover the details of TCP [16, 7, 15, 3, 5, 2, 1, 10, 9, 4, 8]. This section will summarize the TCP characteristics to be manipulated in our experiments but will assume that the reader has some knowledge of TCP operation. The algorithms will be discussed in the context of the sample TCP sessions seen in Figures 9 and 10. The algorithms discussed are acknowledgement mechanisms, the congestion window algorithms, retransmission timeout (RTO) strategy, and round trip time (RTT) estimation. Each section will present an algorithm's equations and explain in a step by step fashion their complex interdependencies.

2.1.1 Acknowledgments

There are three different conditions which result in acknowledgements being sent, and the resulting acknowledgement is typically denoted as one of three types depending on which condition resulted in the acknowledgement being sent.

Delayed acknowledgments give the receiver the opportunity to merge multiple acknowledgments and/or data with the returned segment. Equation 1 shows the estimator used to calculate this delay time, called the acknowledgment timeout (ATO). Figure 9 displays this estimator in action on the right rung of the two ladder diagrams. For example, after receiving segment D2, the estimator uses the time between reception of D1 and D2 as the *time_between_data_receptions* (40 ms) variable which changes the ATO from 1 (10 ms) to 4 (40 ms). The ATO estimator is designed to correlate the frequency of the acknowledgments to the frequency of the incoming data.

$$ATO = ATO/2 + (time\_between\_data\_receptions) \quad (1)$$

Segment A2 represents the only actual delayed acknowledgment shown in Figure 9. The receiver can return two other types of acknowledgments: quick (A1) or forced (A11, A13). Quick acknowledgments only occur on the first data packet of a connection and they are designed to help the sender quickly get to equilibrium where it can send a full window of segments. Forced acknowledgments occur after the receiver sees two **full** segments as recommended in RFC-1122. This device is designed to provide the sender with more frequent measured RTT samples to keep the RTT estimators from aliasing.

When the receiver does return a delayed acknowledgment, as in segment A2, the measured round trip time encapsulates both the ATO and the actual network round trip time. This artifact, in combination with the use of fixed ATOs in some TCP implementations, limits the the lower range for the retransmission timeout. If the sender does allows the RTO to fall below this fixed ATO value, unnecessary retransmissions will occur because the sender RTO will expire before the receiver ATO.

2.1.2 Congestion Window Algorithms

The congestion window algorithms, including slow start and congestion avoidance, provide sender side flow control which work in conjunction with the receive window to limit the amount of data the sender has in transit over a connection [11]. With all of the complex interaction between the different congestion window algorithms, the explanation of the congestion window variable (*cwnd*) can be greatly simplified by focusing on the slow start and congestion avoidance algorithms.

The slow start threshold variable (*ssthresh*) determines whether the *cwnd* is updated according to the slow start or the congestion avoidance algorithm (see Equations 2 and 3). If the *cwnd* is below the *ssthresh* the slow start algorithm takes precedence, otherwise congestion avoidance takes over. The variable *ssthresh* initializes to a very large value and is set when an equilibrium point is reached, determined by loss of packets. This puts a connection into slow start initially, but turns control over to congestion avoidance after reaching equilibrium.

$$cwnd = cwnd + 1 \ (slow\ start) \quad (2)$$

$$cwnd = cwnd + 1/cwnd \ (congestion\ avoidance) \quad (3)$$

Figure 9 details how the Linux kernel actually performs the calculation updates shown in Equations 2 and 3 without using divides. This graphic shows the differences between increasing the *cwnd* quickly, as in the slow start region, and slowly, as in the congestion avoidance region. A typical connection should spend a short period of time in slow start and the majority of time at equilibrium performing congestion avoidance. This allows the sender to quickly find an appropriate send window for the connection and stabilize there.

The right ladder in Figure 10 gives an example of how the congestion window provides flow control. The sender can have only *cwnd* number of packets outstanding on a connection, otherwise it can no longer transmit and must stall waiting for an acknowledgment. After receiving the acknowledgment, the sender updates the *cwnd* and can transmit additional packets.

2.1.3 Round Trip Time

The RTT algorithm is designed to provide the sender with an estimation of the time between a packet transmission and the returned acknowledgment. The retransmission timeout calculator then utilizes the RTT estimations for smoothed round trip time (SRTT) and mean deviation (*mdev*) to accurately determine when a segment can be considered lost. Linux implements the RFC-1122 required RTT algorithm using Jacobsen's recommended gains for the SRTT and *mdev* [3].

Equations 4 and 5 represent the RTT estimators although the actual implementation uses shifts rather than multiplications and divides to speed up processing. The sender calculates the measured round trip time (MRTT) as the time between sending the first unacknowledged packet and receiving a new acknowledgment. For example, in Figure 9, the MRTT measurement for the returned acknowledgment A11 begins from D10 and not D11, although A11 covers the acknowledgment for both D10 and D11. The effects of the new measured round trip time on the SRTT and *mdev* can be seen on the left rung of the ladders in the graphic.

$$SRTT = SRTT * 7/8 + MRTT * 1/8 \qquad (4)$$

$$mdev = mdev * 3/4 + \mid SRTT - MRTT \mid * 1/4 \qquad (5)$$

2.1.4 Retransmission Timeout Strategy

The RTO algorithm is designed to give the sender an accurate determination of when a segment has been lost and should be resent. If a transmitted segment is not acknowledged before the RTO timer expires, that segment will be retransmitted. The left ladder in Figure 10 presents an example of this device in action. Since the acknowledgment segment D1 is lost, the sender never receives the acknowledgment. Although, the receiver did actually receive the segment, the sender has no way of knowing this in this example, so it retransmits D1 after the RTO timer expires.

The RTO algorithms use the senders estimated SRTT and *mdev* variables to determine how long to wait for an acknowledgment before deciding that a packet has been lost. For simplicity, the Jacobsen recommended RTO calculation [11] is presented in Equation 6, though the Linux kernel does make minor adjustments to this algorithm.

$$RTO = SRTT + 4 * mdev \qquad (6)$$

The example session also shows Karn's exponential backoff strategy [12] in action. The *backoff* variable is used to double the RTO value every time a packet is retransmitted and the *backoff* will not be reset until a **non**retransmitted packet has been acknowledged. This algorithm is designed to provide the RTT algorithms with an accurate measured round trip time on a packet that does not contain Karn's retransmission ambiguity.

It is important to note that the default RTO algorithm places both an upper limit (2 minutes) and lower limit (200 ms) on the timeout value. The lower limit accounts for the behavior of some TCP stacks.

2.2 Beowulf

The Beowulf-class parallel machine has evolved from early work in low cost computing. The first work in this area centered around clusters of workstations [6]. These clusters are often composed of existing workstations which are used as interactive systems during the day, can be heterogeneous in composition, and rely on extra software to balance the load across the machines in the presence of interactive jobs. As it became obvious that workstations could be used for parallel processing, groups began to build dedicated machines from inexpensive, non-proprietary hardware. These "Pile-of-PCs" consist of a cluster of machines dedicated as nodes in a parallel processor, built entirely from commodity off the shelf parts, and employing a private system area network for communication [17]. The use of off-the-shelf parts results in systems that are tailored to meet the needs of the users, built using the most up-to-date technology at the time of purchase, and cost substantially less than previous parallel processing systems. The Beowulf workstation concept builds on the Pile-of-PCs model by utilizing a freely available base of software. The free availability of most system software source encourages customization and performance improvements. Experiments have shown Beowulf workstations capable of providing high performance for applications in a number of problem domains.

One of the greatest strengths of commercial systems in general has always been the support, both in software and troubleshooting, that is made available to owners. Along this same vein the Beowulf community has banded together to build a software infrastructure and to assist one another with problems. Most of this software already existed, including the operating system, compiler, network file system, and most common utilities. However, it has become apparent that while this software is robust and fulfills users' needs, there is room for improvement. Parallel file systems such as PVFS [13] provide better I/O performance and consistency for parallel applications using distributed data sets, processor-specific compiler enhancements and libraries can boost application performance, and kernel modifications can provide services such as global process ID's, global signalling, and Distributed Shared Memory (DSM) which help build a more complete environment.

Along these same lines, the existing kernel communication protocols were built for general purpose networks. Thus, these protocols too could potentially be altered or rewritten to more effectively operate in the Beowulf environment. Thus, the impetus for our modifications and experiments.

2.3 Previous Study

Josip Loncaric et al. at the Institute for Computer Applications in Science and Engineering (NASA Langley) have performed testing on TCP connections using unidirectional small messages [14]. They have seen what they call "stalls" in TCP when passing short messages with the TCP_NODELAY option set (Nagle algorithm turned off). These stalls are caused by a combination of the TCP algorithms for congestion avoidance [1] and delayed acknowledgment [3].

Basically, when passing short messages in one direction, even with the Nagle algorithm off, no more than congestion window number of packets can be on the link unacknowledged. In addition, the remote end has a delayed acknowledgment strategy that prevents the acknowledgment from occurring before the ATO expires. The point of these strategies is to allow for packet conservation which makes perfect sense in a lot of applications, but will only hurt in these specific benchmarks.

To fix this problem, Loncaric et al. installed a kernel patch that removed the delayed acknowledgment strategy from the Linux 2.0.34 kernel when disabling Nagle's algorithm, instead immediately acknowledging all incoming packets for the connection. Loncaric notes on his web page that using this patch led to a factor of 20 improvement when sending 100,000 single byte messages. However, we will see that this type of simple modification on its own can have detrimental effects to other traffic patterns.

3 Protocol Modifications

TCP provides a good general purpose transmission algorithm that performs well under a wide variety of network conditions and speeds, however, in some network environments these algorithms may not perform optimally. For example, the Beowulf concept establishes networks as private and local, which may benefit from transmission algorithms geared towards this type of network. In addition, since the optimal transmission algorithms may not conform to standards set forth in the RFCs, Beowulf's closed network provides an excellent test platform by isolating these nonstandard algorithms from outside networks.

Applications running in the Beowulf environment may exhibit different communication patterns and may require different transmission algorithms for optimal network performance. A configurable model of transmission algorithms can accommodate this discrepancy by allowing each application to configure the transmission algorithms appropriate to their communication needs. However, the model presented here provides a system level configuration of TCP. This model is sufficient for a number of situations and led to a simpler implementation.

The following TCP algorithms can be configured in our model:

- Congestion window algorithms
- Acknowledgment algorithms
- Round trip time and mean deviation estimator
- Retransmission timeout calculation
- Experimental options (timestamp, window scaling, and selective acknowledgments)

In this section we provide a description of the configuration options available using our module and an overview of its implementation.

3.0.1 Acknowledgments

Three new options are available for acknowledgements:

Maximum Segments Before Forced ACK – Sets the maximum number of full segments to receive before an ACK is sent.

Fixed Delayed ACK Estimator – Causes the delayed ACK estimator function to always return a fixed value instead of calculating how long to wait.

Quick Delayed ACK Estimator – Causes the estimator function to always indicate that an ACK should be sent.

These new algorithms should provide experimental evidence on the effects of different acknowledgment timeout algorithms, testing how small versus large ATO values effect different communication patterns. In addition, the effects of changing the number of received segments before forcing an acknowledgment will be ascertained.

Using fixed ATO values prevents the receiver from adjusting dynamically to the frequency of data receptions as the default estimator does that is described in Section 2.1.1. Fixed estimators are normally designed to simplify the estimation function, but the purpose here is to see if speedup can be achieved on applications using Beowulf clusters.

RFC-1122 recommends that the the default value for the number of full segments received before forcing an acknowledgment be set to two[3]. This strategy provides the sender with frequent MRTT samples in order to obtain an accurate RTT estimation. However, this algorithm will be changed to see if larger values can reduce sender interrupt processing time by decreasing the number of returned acknowledgments.

3.0.2 Congestion Window

We provide two mechanisms for manipulating the congestion window algorithms:

Fixed Congestion Window – Sets the congestion window to a user defined fixed value.

Initial Window Value – Modifies initial window value to be set to a user specified value when the window is first initialized and whenever slow start is reset.

Forcing the *cwnd* to a constant value prevents it from adapting to changes in network conditions as the default algorithm would. However, since this algorithm will be used in a Beowulf cluster, the *cwnd* may not need to be adaptable. This strategy can put a connection directly into an equilibrium state if the appropriate value is chosen. Setting the initial congestion window allows connections to reach the equilibrium point more quickly, reducing the effects of slow start.

3.0.3 Round Trip Time

Two alternate RTT functions are made available:

Fixed Round Trip Time – Sets the RTT estimator function to return a user defined value at all times.

Current Round Trip Time – Sets the RTT estimator to use a simple estimation function which simply uses the current RTT shifted left by a user specified number of bits.

These algorithms allow us to examine how large and small SRTT values effect communication performance. In addition new algorithms will provide a fast acting SRTT estimator (Round Trip Time Current) for comparison against the complicated default smoothing estimator.

A constant value for the smoothed round trip time prevents the sender from adjusting dynamically to congestion points as it would with the default algorithm described in Section 2.1.3. Setting this value in a Beowulf cluster with a known network round trip time may actually boost performance in some situations. In contrast to the fixed value algorithm, using the current value as the basis for the SRTT value provides an estimator that will adjust very quickly to changes in network loads.

3.0.4 Retransmission Timeout

Three replacement RTO functions were implemented:

Fixed Retransmission Timeout – Sets the retransmission timeout function to return a constant, user specified value at all times.

Retransmission Timeout Upper Limit – Places a user defined upper limit on the value returned by the RTO function.

Limitless Retransmission Timeout – Removes any upper and lower limits on the values returned by the RTO function.

These algorithms provide the means for testing how large versus small values of the RTO perform on different patterns in a cluster.

A constant value for the retransmission timeout effectively wipes out all the estimating functions (SRTT, *mdev*, backoff, and RTO) used by default and described in Section 2.1.4. However, as with the RTT constants, Beowulf clusters have generally consistent network round trip times, which could make this algorithm beneficial in some situations. To appropriately control the RTO value, the bounding functions must be modified because it limits the RTO on the lower side, to handle fixed ATO implementations, and on the upper side, to keep the RTO from getting out of control. Since Beowulf clusters only communicate with Linux machines, the lower limit can be safely removed; however, care must be taken when changing the upper limit.

3.1 Implementation Overview

In order to facilitate testing of these new algorithms, two features of the Linux operating system were used: modules and the /proc file system. Linux modules provide the capability of attaching these new algorithms to the function hooks at runtime, while the /proc file system allows user level control of the modules. To reiterate, the modules contain all of the new algorithms and the user specifies which algorithm to attach by controlling the module through the /proc file system.

Two Linux modules, beo_config.o and beo_slow.o, contain all of the algorithms tested in this thesis. The former provides algorithms for ATO, RTO, and RTT estimators and calculators and the latter consists of the new algorithms for controlling the congestion window. After inserting the modules into the kernel using the “insmod” command, the following directories and files will be added to the /proc file system:

- /proc/beowulf (directory added by beo_config.o)
- /proc/beowulf/beo_config (file added by beo_config.o)
- /proc/beowulf/beo_slow (file added by beo_slow.o)

Once the modules are in place, users can use the /proc file system for control and status. Table 2 shows how to use the /proc file system for this functionality. More details are available in [?].

4 Testing and Results

This section will present testing methods and results aimed at analyzing the benefits of tuning TCP for various traffic patterns. In Section 4.1, the hardware setup and the test methodology will be documented. Section 4.2 will examine test benchmarks on individual modifications made to each transmission algorithm. Section 4.3 will specify, test, and compare an all encompassing transmission algorithm designed using the previous experiments to boost TCP performance on Beowulf clusters.

4.1 Test Setup

The following information describes the hardware setup for these tests:

- 17 nodes: P5-150, 64MB RAM, 2 Tulip NICs
- Linux v2.2.5, tulip.c v0.88
- Intel Express 510T 100mb switch connecting 16 slaves
- Head node connected to slaves via Asante 100mb hub

The head node spawns off all tasks and all message passing communication takes place on the switch. The test applications used either Beowulf Network Messaging (BNM) or native sockets as the communications transport layer. These message passing mechanisms provide direct evidence on the effects of the modifications made to TCP without interference from additional layers provided by PVM and MPI.

BNM is currently under development at the PARL Laboratory at Clemson University as a low level solution for task spawning and communication in the Beowulf environment. BNM has been implemented directly over the BSD sockets interface, providing a direct picture of how the TCP modifications affect communication latencies and bandwidths.

4.2 Preliminary Testing

The tests that were designed for this section isolate each of the modifiable algorithms presented in Section 3, allowing the individual effects on the various communication patterns to be examined. All the tests described in this section were implemented using the BNM message passing library. Section 4.2.1 tests one way message passing communications. Section 4.2.2 attempts to provide evidence on the effects of transmission algorithms on interactive communication.

4.2.1 Uni-directional Messages Tests

The uni-directional tests were designed to determine the effects of the individual algorithms on passing data from task to task in one direction. In these tests, two processes are spawned on two different processors with one task having rank 0 and the other rank 1. In all experiments, task 0 and task 1 reside on the same respective nodes and task 0 always passes data to task 1.

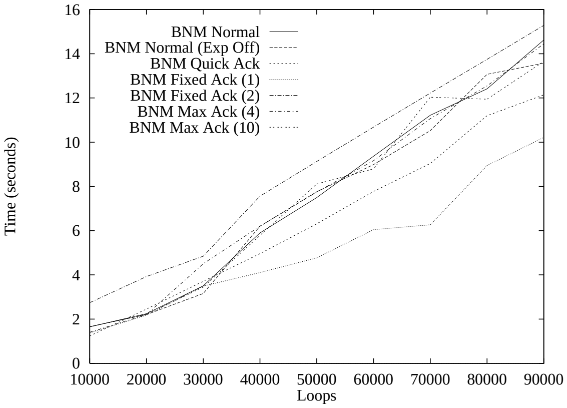


Figure 1: Uni-directional (1 Byte): Acknowledgments

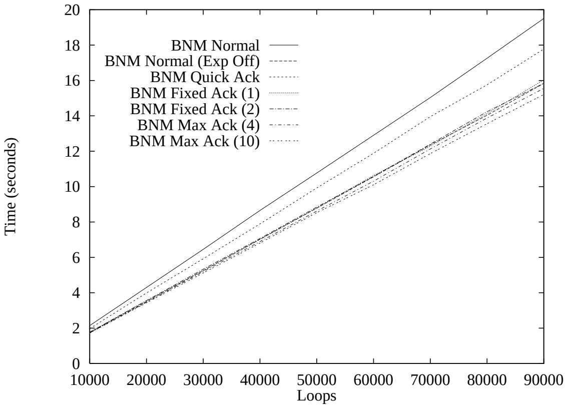


Figure 2: Uni-directional (Full): Acknowledgments

The number of iterations and the size of the TCP data payload to send were varied. The loop value was varied between 10000 and 90000 bytes with 1000 byte increments, and reported values are the average of five runs. Two distinct data payload (segment) sizes were tested: single-byte payloads and maximum size (full) payloads. The Nagle algorithm [15] was disabled for the single-byte payload tests. Experimental options were turned off in all cases except when otherwise noted.

The first pair of test runs (Figures 1 and 2) concentrated on the effects of varying acknowledgement algorithms. In these tests, seven different configurations are tested:

- normal kernel operation with experimental options on
- normal kernel operation with experimental options off
- quick acknowledgement of all packets
- fixed ATO values of both 10 and 20 ms

- forced ACK at both 4 and 10 full segments

Figure 1 displays results from the uni-directional small message experiments on the acknowledgment algorithms. This graph indicates that the “Max Ack” modifications did nothing to benefit or hinder the default algorithm. With small payloads such as these, the forced acknowledgement algorithm rarely has any effect on ACK transmission.

The figure does show that changes in the acknowledgment delay has a considerable impact on performance. Since forced acknowledgments never occur in this situation, the sender must rely on the ATO timer to expire before receiving an acknowledgment. If the sender has *cwnd* packets in transit, waiting for the acknowledgment stalls communication and degrades performance. The graph proves this by displaying attenuated performance as the fixed ATO value increases.

Executing quick acknowledgments on every packet guarantees timely acknowledgments that will prevent the sender from stalling. However, acknowledging every single segment does not conserve network resources and flooding the sender with acknowledgments forces continuous system interrupts that reduce performance. Figure 1 clearly displays this degradation and points to the lowest possible fixed ATO value as the best of both worlds. The 10 millisecond ATO provides timely responses and merges multiple acknowledgments, thereby conserving network resources and reducing the number of sender interrupts.

Figure 2 shows that the delayed acknowledgment algorithms have very little effect on full sized segments. Since by default the receiver acknowledges every two full sized segments anyway, the sender is rarely limited by the *cwnd*, preventing stalls. The graph does show slight benefit from removing the experimental options of TCP. This improvement can be attributed to the lower header overhead and the removal of the complex selective acknowledgement (SACK) implementation.

The full segment graph does show positive impact from changes in the forced acknowledgment algorithm. This improvement is caused by the reduction in the number of acknowledgments interrupting the sender. This clearly indicates the significance of sender interrupt processing overhead; however, too large of a value for forced acknowledgments can create conditions that will stall the sender waiting for an acknowledgment. In this case forcing an acknowledgement at 10 full-sized segments is most effective. Forcing acknowledgement at 4 full-sized segments was almost as effective, indicating that reducing the number of acknowledgements beyond this point is of little benefit.

The next pair of tests (Figures 3 and 4 concentrated on effects of varying the congestion window algorithms for these same single-byte and full segment traffic patterns. In the first test, using single-byte payloads, seven distinct

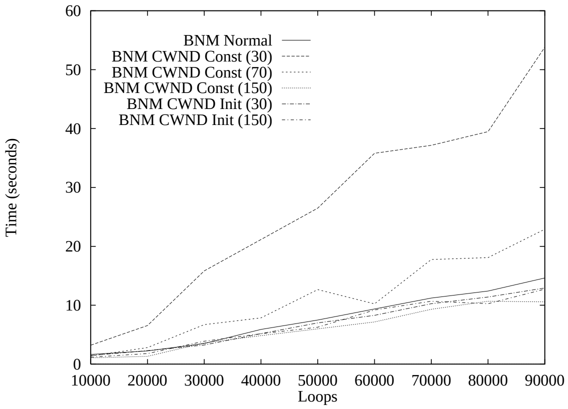


Figure 3: Uni-directional (1 Byte): Congestion Window

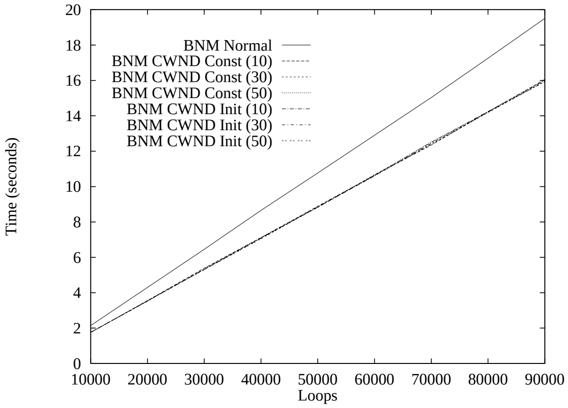


Figure 4: Uni-directional (Full): Congestion Window

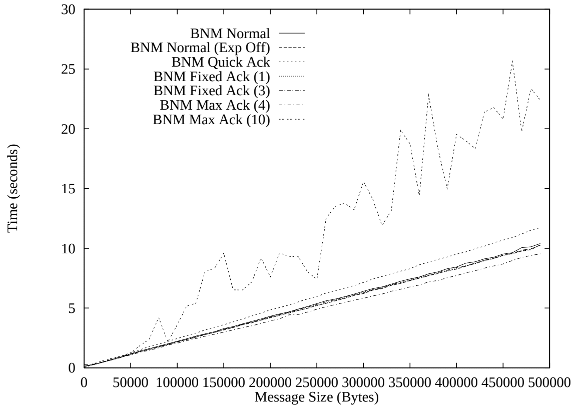


Figure 5: Bi-directional (Full): Acknowledgment

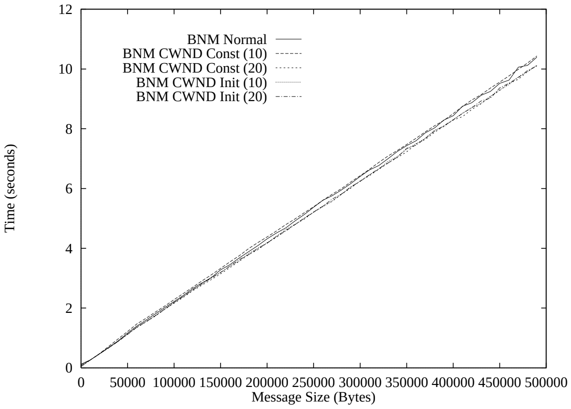


Figure 6: Bi-directional (Full): Congestion Window

configurations were tested:

- normal kernel operation with experimental options on
- fixed congestion window of 30, 70, and 150 packets
- initial window size of 10, 30, and 50 packets

Figure 3 presents test results for these runs. This graph indicates that the congestion window size has a slight effect on performance. Again, this effect is a result of the stalls seen by the sender when waiting for an acknowledgment. The large congestion windows allow the sender to continue to transmit where a small window would stall.

In the second test, using full-sized payloads, the fixed congestion window sizes tested were 10, 30, and 50 packets. Figure 4 shows that the slight benefits seen with small messages do not carry over to full sized segments. This fact comes directly from the TCP receive window limitations. The TCP packet format confines the window size to 64K and the silly window syndrome (SWS) avoidance algorithm bounds this 64K value by advertising a maximum 32K receive window. Since the sender can send no more than the minimum of the congestion window and receive window, congestion windows greater than 22 (approx. 32K) are useless when transmitting full segments.

Testing varying the RTO and RTT strategies showed little impact on this type of communication. Since little congestion occurs data losses are not common, which precludes the need for retransmissions.

4.2.2 Bi-directional Messages Tests

The bi-directional tests were designed to examine the effects of the transmission algorithms on two-way communication. In these tests, two tasks are spawned on two different processors with one task having rank 0 and the other rank 1. In all experiments, task 0 and task 1 reside on the same respective nodes and task 0 passes data to task 1 then task 1 passes back to task 2 ending the loop. Theses tests very the message size from 1 to 500,000 bytes.

Figure 5 presents the effects of the acknowledgment algorithms on the bi-directional messages with the following configurations:

- normal kernel operation with experimental options on
- normal kernel operation with experimental options off
- quick acknowledgement of all packets
- fixed ATO values of 10, 30, and 40 ms
- forced ACK at 4 and 10 full segments

The graph shows an improvement using a moderate forced acknowledgment value, but shows a degradation when using a large value. The moderate forced acknowledgment increases performance by reducing the time senders spend processing returned acknowledgments, while the large value decreases performance by causing the sender to stall waiting for an acknowledgment.

Figure 6 shows that the congestion window algorithms in the kernel are hard to beat in an individual test. Here the following configurations were examined:

- normal kernel operation with experimental options on
- fixed congestion window of 10 and 20 packets
- initial window size of 10 and 20 packets

The graph does show that too small a value for a fixed congestion window will degrade performance slightly. However, in the long term, all tested algorithms spend approximately equal amounts of time in equilibrium.

The graphs for the RTT and RTO results have been excluded because again these tests create no significant congestion leading to lost segments.

4.3 Beowulf Transmission Policy

The tests described in this section were designed to reveal the effects of multiply specified transmission algorithms on communication benchmarks and a sample cluster application. A logical combination of algorithms will be documented in Section 4.3 and tested in Sections 4.3.1 through 4.3.3. Section 4.3.1 utilizes the traditional NetPIPE benchmark to test network performance. Section 4.3.2 describes performance for a multi-node communication patterns and Section 4.3.3 examines implications on an actual cluster application using the Parallel Virtual File System (PVFS).

The Beowulf Transition Policy (BTP) will assemble the best combination of the individual algorithms tested for our test environment. The specified algorithms need to perform well individually as well in combination with the other algorithms. The algorithms that make up the BTP are as follows:

- Fixed ATO = 10 ms
- Forced acknowledgments at 4 full sized segments
- Initial congestion window = 20
- SRTT = 2*MRTT
- $mdev = 0$
- No bounds on the RTO

The acknowledgment strategy used in the BTP combines a small fixed ATO value of 10 ms with a moderate value of 4 for the forced acknowledgment. The small fixed ATO provides a timely acknowledgment for slow data transfers while reducing the amount of transmitted segments by attempting to merge multiple acknowledgments. The new forced acknowledgment combines with the ATO benefits to further decrease the frequency of acknowledgment which reduces the interrupt processing time on the sender.

The change in the forced acknowledgment value does reduce the RTT sampling rate, which would introduce aliasing in the default RTT algorithms. We account for this in BTP by utilizing a fast acting estimator based on the current MRTT. This estimator sets the SRTT to twice the current MRTT and is not effected by the decreased frequency of RTT samples. The BTP uses the default kernel RTO calculation, but removes the upper and lower limits to allow a wider range of RTO values.

The initial congestion window value of 20 was picked as a point where a connection can immediately get to equilibrium when transferring full sized segments. This algorithm also allows the congestion window to increment during frequent small packet transmissions. As an additional benefit, keeping the value of the congestion window over 4 meshes well with the forced acknowledgment value for the BTP by preventing stalls caused by the sender waiting for delayed acknowledgments.

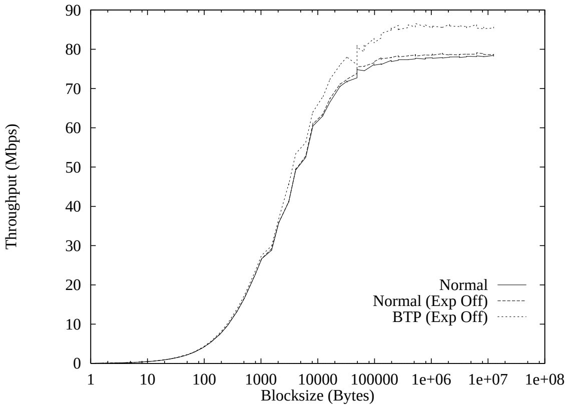


Figure 7: Netpipe Throughput

4.3.1 NetPIPE Tests

The NetPIPE benchmark measures network performance characteristics between two nodes. The NetPIPE benchmarks presented here utilize TCP sockets directly, and the same two nodes were used on all NetPIPE tests. The graph in Figure 7 represents the throughput on our switched fast ethernet network for the algorithms tested, which were:

- normal kernel operation with experimental options on
- normal kernel operation with experimental options off
- BTP with experimental options on
- BTP with experimental options off
- BTP without new congestion window policy

Figure 7 examines network throughput on both large and small message sizes. The BTP achieves greater than 6% improvement in bandwidth at large message sizes. These benefits can be attributed mainly to the combination of the congestion window and acknowledgment strategies. These strategies combine to bring a connection quickly to equilibrium while conserving acknowledgment packets. In addition, eliminating the experimental options when not experiencing heavy congestion contributes slight gains.

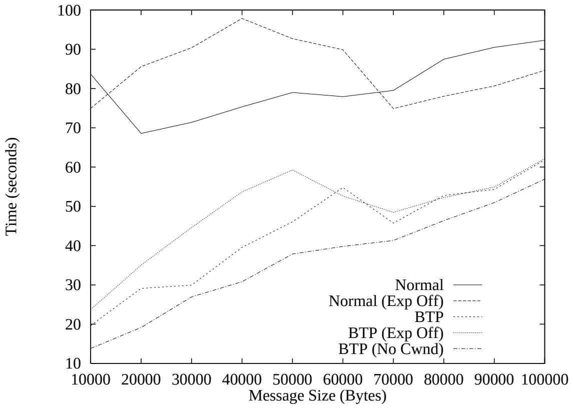


Figure 8: All-to-All Pattern Tests

4.3.2 All-to-All Tests

The All-to-All tests were designed to ascertain the effects on applications with large congestion points using BNM. These tests spawn 16 tasks on 16 nodes with the same ranking tasks always on the same nodes for each test. In the All-to-All tests, each task passes data to every other task and vice versa to complete an iteration. The message size was varied and 100 iterations were performed for each test.

Figure 8 presents the results for the same algorithms tested in the Netpipe tests. These graphs clearly indicate the usefulness of a user specified algorithm in the Beowulf environment. The combination of the acknowledgment and round trip time policies provides performance improvements of over 50% for some highly congested patterns. However, the BTP congestion window policy used actually degrades performance for these tests by inhibiting the connection from slowing down during packet losses.

The new RTT estimator seems to contribute the largest portion of the performance improvement seen in these tests. The use of this fast acting RTT algorithm in combination with the elimination of the 200 ms lower bound on the RTO allows the network protocol to adjust very quickly and accurately under dynamic loads. This benefit seems to decrease somewhat as the message size increases.

The BTP acknowledgment policies contribute during any bulk transfer as discussed in Section 4.3.1 and also reduce congestion by decreasing the number of acknowledgments. Figure 8 also shows that enabling the experimental options improves performance, which can be attributed to the ability of the SACK protocols to recover from multiple packet losses. In this particular communication pattern, the benefits achieved by SACKs outweigh the complexity of the implementation.

Table 1: Jacobi Results

BTP	45 seconds
Normal	63 seconds
Normal (Exp Off)	63 seconds

4.3.3 Jacobi Tests

The Jacobi tests were designed to show how real world applications might benefit from BTP. This application performs the traditional Jacobi iterative method using the Parallel Virtual File System (PVFS) with an out of core strategy.

The Jacobi method was used to solve a 2K x 2K matrix. The results shown in Table 1 indicate that for this size the BTP improves performance by 29%. The results obtained provide evidence of the benefit of configurable transmission algorithms on real world applications utilizing Beowulf clusters.

5 Conclusions

Experimental results presented in this work have proven the viability of configurable transmission protocols on Beowulf workstations. Performance improvements using new transmission strategies ranged from 6% to well over 50% depending upon the communication pattern. These benefits were obtained using simple and easy to implement algorithms geared to applications on the Beowulf architecture.

Future research and experimentation will focus on more application studies. Further investigation of boosting performance for applications using standard message passing libraries such as MPI and PVM, which use TCP for much of their communication, should be of direct benefit to the Beowulf community.

Future configurable transmission algorithms may further enhance performance by adding modifiable algorithms such as fast retransmit, or by implementing more complicated transmission algorithms. For example, new algorithms can be derived with detailed analysis of typical communication patterns on Beowulf clusters. Another approach might utilize theoretical mathematical models to describe the algorithms and speedup limitations. Either of these methods would have a good chance at boosting performance beyond the simple Beowulf Transmission Policy.

Even if research produces an optimal system level transmission policy for a specific architecture, applications still produce different communication patterns. These various communication patterns change the effects of the transmission protocols, moreover, running multiple applications concurrently completely alters the communication dynamics of the single process. Possible solutions to these problems would incorporate a system level strategy that accepts user level hints, giving the application some control but leaving the final decision to the underlying system policy.

References

[1] M. Allman, V. Paxson, and W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms (RFC-2001). Technical report, IETF Network Working Group, January 1997.

[2] F. Baker. Requirements for IP version 4 routers (RFC-1812). Technical report, IETF Network Working Group, June 1995.

[3] R. Braden. Requirements for internet hosts – communications layers (RFC-1122). Technical report, IETF Network Working Group, October 1989.

[4] R. Braden. T/TCP - TCP extensions for transactions: Functional specification (RFC-1644). Technical report, IETF Network Working Group, July 1994.

[5] S. Bradner. Key words for use in RFCs to indicate requirement levels (RFC-2119). Technical report, IETF Network Working Group, March 1997.

[6] Karen Castagnera, Doreen Cheng, Rod Fatoohi, Edward Hook, Bill Kramer, Craig Manning, John Musch, Charles Niggley, William Saphir, Douglas Sheppard, Merritt Smith, Ian Stockdale, Shaun Welch, Rita Williams, and David Yip. Clustered workstations and their potential role as high speed compute processors. Technical Report RNS-94-003, NAS Systems Division, NASA Ames Research Center, April 1994.

[7] D. Clark. Window and acknowledgement strategy in TCP (RFC-813). Technical report, Computer Systems and Communications Group, MIT, July 1982.

[8] V. Jacobsen. Compressing TCP/IP headers for low-speed serial links (RFC-1144). Technical report, IETF Network Working Group, February 1990.

[9] V. Jacobsen and R. Braden. TCP extensions for long-delay paths (RFC-1072). Technical report, IETF Network Working Group, October 1988.

[10] V. Jacobsen, R. Braden, and D. Borman. TCP extensions for high performance (RFC-1323). Technical report, IETF Network Working Group, May 1992.

[11] V. Jacobsen and M. Karels. Congestion control and avoidance. In *Proceedings of ACM SIGCOMM '88*, 1988.

[12] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. In *SIGCOMM '87*, August 1987.

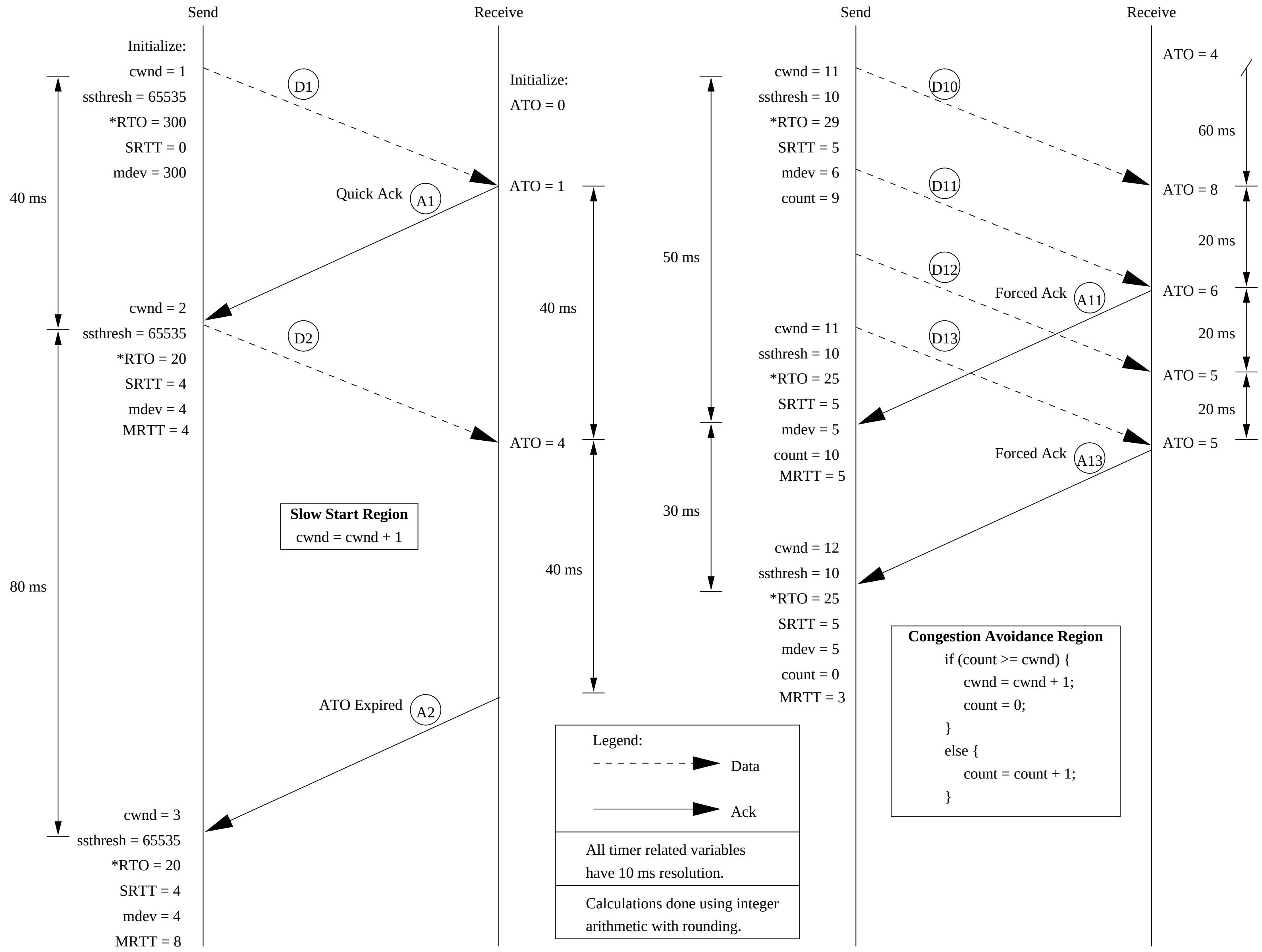
[13] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.

[14] Josip Loncaric. Linux TCP performance fix for short messages. http://www.icase.edu/coral/LinuxTCP.html, 1999.

[15] John Nagel. Congestion control in IP/TCP internetworks (RFC-896). Technical report, IETF Network Working Group, January 1984.

[16] Jon Postel. Transmission control protocol - DARPA internet program protocol specification (RFC-793). Technical report, Information Sciences Institute, University of Southern California, September 1981.

[17] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.



```
if (count >= cwnd) {
    cwnd = cwnd + 1;
    count = 0;
}
else {
    count = count + 1;
}
```

Figure 9: TCP Session Example #1

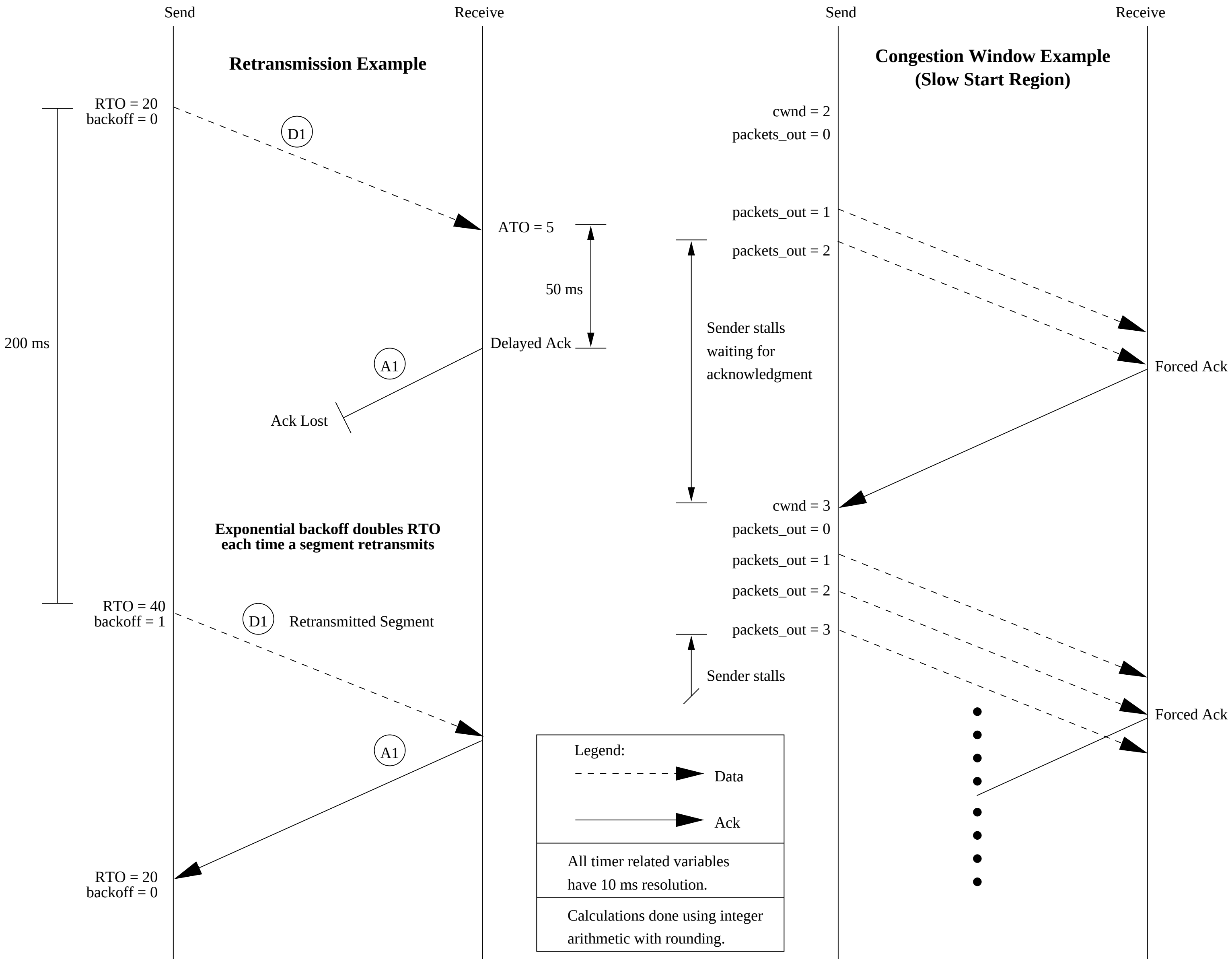


Figure 10: TCP Session Example #2

Table 2: Module/Proc File System

/proc Files	
/proc/beowulf/beo_config	Changes/Info on RTO, RTT, and ATO
/proc/beowulf/beo_slow	Changes/Info on congestion window
To Modify Algorithms	
echo -o [0/1] > /proc/beowulf/beo_config	Turn experimental options (SACK's, timestamp, and window scaling) on/off
echo -d [0/1/2] > /proc/beowulf/beo_config	Set ATO function (0 = off, 1 = quick, 2 = fixed)
echo -f <value> > /proc/beowulf/beo_config	Set fixed ATO value
echo -b [0/1/2] > /proc/beowulf/beo_config	Set RTO bound function (0 = off, 1 = no bounds, 2 = upper bounds
echo -t <value> > /proc/beowulf/beo_config	Set RTO upper bound
echo -s [0/1] > /proc/beowulf/beo_config	Set RTO calculator (0 = off, 1 = constant)
echo -k <value> > /proc/beowulf/beo_config	Set RTO constant
echo -r [0/1] > /proc/beowulf/beo_config	Set RTT estimator (0 = off, 1 = constant, 2 = current (shift)
echo -c <value> > /proc/beowulf/beo_config	Set RTT constant value
echo -h <value> > /proc/beowulf/beo_config	Set RTT shift value
echo -m <value> > /proc/beowulf/beo_config	Set beo_max_ack for forced acknowledgments
echo -i <value> > /proc/beowulf/beo_config	Reset to original values
echo -c 1 -f 1 -r 1 -d 1 -n 1 > /proc/beowulf/beo_slow	Set congestion window to constant value
echo -v <value> > /proc/beowulf/beo_slow echo -t <value> > /proc/beowulf/beo_slow	Set congestion window constant Set beo_init_cwnd for initial congestion window
echo -i <value> > /proc/beowulf/beo_slow	Reset to original values
To View Current Setup	
cat /proc/beowulf/beo_config	Display RTO, RTT, and ATO setup
cat /proc/beowulf/beo_slow	Display congestion window setup