Guaranteed ATM: Design, Verification, and Performance Optimization.

Purvesh B. Thakker

This report is submitted in partial fulfillment for the requirements for Highest Honors

Summer 1998

Abstract

This project involves designing and implementing an ATM network transport called Guaranteed ATM. ATM's philosophy does not guarantee the delivery of data since many new network demands, such as voice and video, do not need all data to reach its destination. Applications that do require a delivery guarantee must provide it in software. Fore Systems, a leader in ATM technology, provides a good starting point with its simple hardware interface, so Guaranteed ATM lies on top of Fore's interface providing a system of error and flow control. The layer lives up to its name of "Guaranteed" thanks to a succession of torture tests. After optimization, the primary bottlenecks exist outside of the Guaranteed ATM layer. The conclusion reveals ways to enhance future performance including integration with Fore System's interface and synchronization enhancements.

Acknowledgements

I would like to thank Dr. George for inviting me to join the HCS Lab and complete this project under his guidance. With his passion for excellence, he has inspired me in my pursuit of knowledge. I would also like to thank all of the students in the lab for their lessons and comradeship. Though they deserve it, there are too many to name individually. When I began in January, I knew little about programming and even less about networking. After six months of patience and motivation from all of these individuals, the knowledge accumulated to the point where I could compose a lengthy thesis with plenty of things left unsaid. I only hope that the small piece of work that I have contributed will make it easier for others to learn what I have learned.

Table of Contents

1. Introduction

Asynchronous Transfer Mode (ATM) has built up industry momentum as a networking technology of the future. Networking technology often get confused with cabling technology. Cabling technology delivers information from one device to another, while networking technology describes how the switches, routers, and other equipment move information about the entire network. ATM provides a networking philosophy that can operate using many different types of cables. It operates at throughput speeds of 155 Mb/s over cables that meet the Optical Cable 3 (OC3) specification. ATM removes the distinction between local area and wide area networks, and bridges the gap between the many types of networks in place today. Currently, three different networks may come into homes and offices - a telephone network, a television network, and a data network. With ATM, one system of cables can provide all three services while opening up new synergistic possibilities. Videoconferencing provides a good example. Today telephones provide two-way voice, televisions deliver one-way voice and video, and computers allow document publishing among other things. If all three services came over the same network, individuals could see and speak to each other, while at the same time drawing diagrams to facilitate explanations, editing documents simultaneously, and generally sharing applications. While ATM opens up new possibilities by providing a bridge over these different networks, ATM's philosophy does not guarantee the delivery of data. This project involves the design and implementation of an ATM transport called Guaranteed ATM. The rest of this section describes the following project goals: (1) guaranteed delivery; (2) maximized performance; and (3) future integration with SCALE.

1.1 Guaranteed delivery

The first project goal involves guaranteed delivery. Of course, the system cannot promise true guaranteed delivery, but for purposes of this project guaranteed delivery is defined as one bit error per year of 1Mb/s transmission. Unlike other data networks, ATM makes its "best effort" to deliver data. In other words, when traffic becomes congested at any switch or other device, that device will begin skipping information to relieve congestion instead of backlogging the entire system. During rush hour traffic in a big city, a wide area of traffic can come to a standstill due to a single bottleneck. If cars did not have to reach their destination, a congested intersection could skip them relieving the rest of the road system. Much of the information travelling over an ATM network, such as voice and video, will maintain a high level of quality even with the lost information. In fact telephone and television networks operate using a "best effort" philosophy. Data communication, however, often requires guaranteed delivery. ATM networks leave this task to be accomplished in software.

1.2 Maximized performance

The project's second goal involves reducing bottlenecks in the Guaranteed ATM layer. Since the focus is on guaranteed delivery, a ready interface with the hardware was chosen as the block to build upon. Fore Systems leads the world in ATM technology and the Fore Application Programmer Interface (API) provides an easy to use interface with the hardware, so it was an obvious choice [Fore93]. Modifying the Fore API to integrate with Guaranteed ATM's requirements would enhance performance in the future. Figure 1 shows how Guaranteed ATM interfaces with the outside world. To maximize performance, two performance variables are measured. Throughput provides the primary measure of performance, and latency provides the secondary one. Throughput is simply the amount of data that passes to and from the application layer in one second. Most performance numbers published take into account only simplex communication. This project attempts to focus on full-duplex throughput since that is a more realistic scenario. Real applications usually have more than one thread or process that want to send and receive data at the same time. Latency is simply the time it takes an application on one computer to send a message to the application on the other computer.

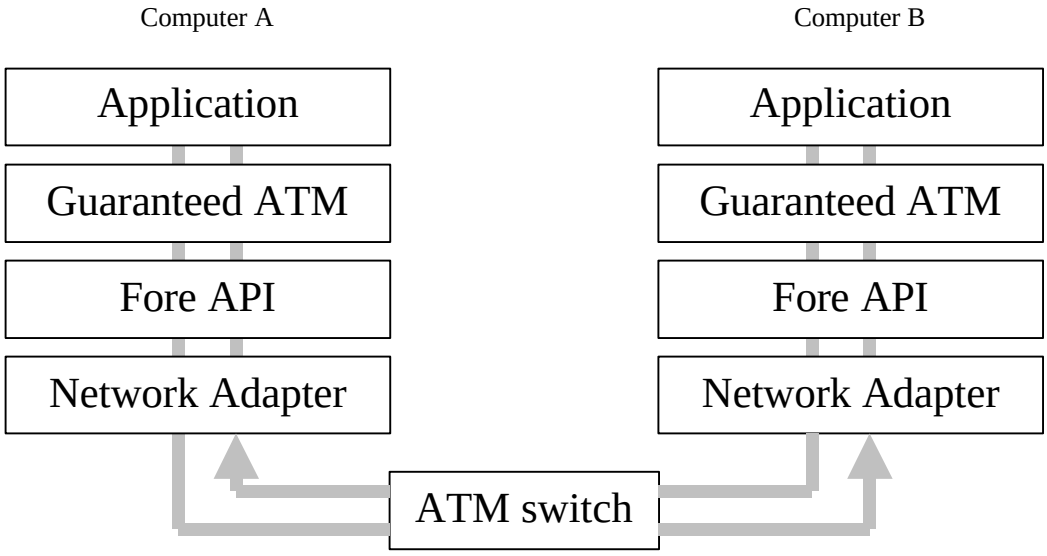


Figure 1: Guaranteed ATM's interface with the outside world.

1.3 Future Integration with SCALE

Although this project creates a guaranteed ATM interface, it was started thanks to an outside motivation. This outside motivation was the SCALE Suite, or Scalable Cluster Architecture Latency-hiding Environment. It is currently being developed at the High-performance Computing and Simulation Lab (HCS) at the University of Florida. The SCALE Suite provides a platform for efficient distributed computing over high performance networks. SCALE was designed to run over several different types of networks simultaneously, including ATM. The project's third and final goal is to allow painless integration with SCALE in the future [George96, Suite97a].

2. Design Options

To implement a guaranteed interface, some possible architectures should be explored. These architectures should correct for frames of data that may be lost or corrupted in transit. Two mechanisms are implemented to provide guaranteed delivery, flow control and error control. The two are easily confused since they are implemented simultaneously, but they address very different problems. Error control guarantees that data comes into the receive buffer accurately, while flow control ensures that data leaves the receive buffer correctly. Flow control assures that a transmitting station does not overwhelm a receiving station with data. The receiver has a data buffer where information is stored until higher layers of software retrieve it. In the absence of flow control, the receiver's buffer may overflow and overwrite data that has not yet been retrieved by the application. Error control refers to mechanisms that detect and correct errors that occur in the transmission of frames of data.

2.1 Flow Control

In order to ensure that the receiver is not overwhelmed with data, flow control techniques usually involve the following ingredients:

1. *Positive Acknowledgements:* The receiver returns a positive acknowledgement when frames are received.
2. *Retransmission after timeout:* The source retransmits a frame that has not been acknowledged after a predetermined time has elapsed.

The simplest form of flow control is known as "stop-and-wait." In stop-and-wait flow control the sender sends a frame and then waits for the receiver to send back an acknowledgement. If an acknowledgement is not received after a certain timeout period, then the information is retransmitted. In this way, the sender will never send the next frame of data until the receiver is ready for it, and the sender will continually retransmit the data until it is accepted. As an extra precaution, the output frames are alternately labeled with a "0" and "1" sequence number. If an acknowledgement gets lost in transit, this extra precaution informs the receiver that a second copy of the same frame has arrived. Without this precaution, the receiver would accept the copy as the next frame of

data. Although effective, stop-and-wait does not provide efficient throughput. A piece of data spends a large portion of its time travelling through the network. Because of this factor, the sender and receiver spend far too long waiting for each other to reply. The sliding window provides a more efficient method of implementation.

The sliding widow mechanism allows more than one frame to be in transit at one time. This simultaneous transmission cuts down on waiting time allowing greater throughput. The sliding window operates as shown in Figure 2. When the window reaches the end of the buffer, it wraps back around to the beginning. Three window variables need to be tracked, *buffer_begin* , *buffer_cur*, and *buffer_end*. *Buffer_end* is increased when additional data is copied into the buffer. *Buffer_cur* points to the frame that is to be transmitted next. *Buffer_begin* is increased to close the window once an acknowledgement has been received. Each frame has a frame number associated with it. When an *ack* is received, it is implied that all previous data was also received correctly. If no acknowledgement is received after a certain timeout period, then all pending frames are resent [Stallings94].

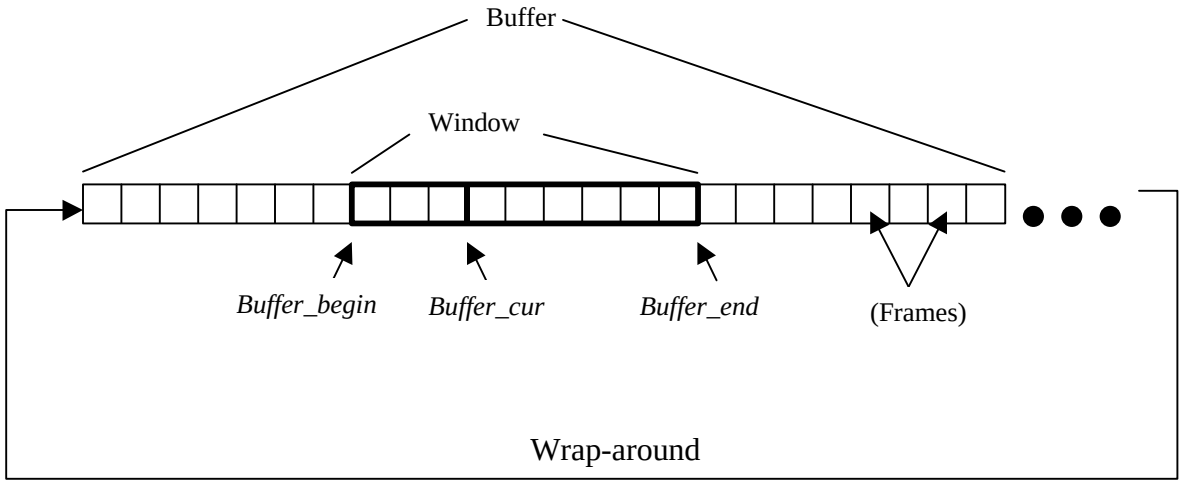


Figure 2: Sliding Window Protocol

2.2 Error Control

In order to ensure that correct data enters the receive buffer, error control usually adds two ingredients to those in flow control. These ingredients are collectively referred to as Automatic Repeat Request (ARQ).

1. *Positive Acknowledgements (Ack):* The receiver returns a positive acknowledgement when error free frames are received.
2. *Retransmission after Timeout:* The source retransmits frame that has not been acknowledged after a predetermined time has elapsed.

3. *Error Detection:* Implemented with checksums.
4. *Negative Acknowledgement (Nack) and Retransmission:* The receiver returns a negative acknowledgement to frames in which an error was detected causing the sender to retransmit those frames. Although negative acknowledgements are not required, they help with performance since expiring timeouts are not depended upon [Stallings94].

Error detection is usually accomplished with a simple checksum. Data is summed on the send side and attached to the frame. Upon receipt of the data, the receive side sums up all of the data and compares this sum to the one passed. If the sums do not match, then the receive side knows that the data was corrupted in transit. Although nearly fool proof, this technique is not perfect. There is a slim chance that two or more data corruptions will occur that will offset each other. Sometimes two wrongs do make a right. Corruptions of any kind are so rare to begin with that the corruption cancelling possibility is nearly zero. In all of the performance testing done for this project, not one bad checksum surfaced. An additional feature sometimes added to error detection is forward error correction. This feature involves sending additional information that will not only allow the receiver to detect an error, but will also allow the receiver to correct the error without a retransmit. Forward error correction is accomplished through a matrix multiply that adds additional bits to the data and is conceptually a compressed form of sending two copies of the data. This technique was not implemented in this project because of how rare data corruption is and how easily a retransmit can be accomplished. Forward error correction is normally only used in corruption-prone, high-latency communication such as with satellites and submarines.

2.3 Automatic Repeat Request (ARQ)

Although treated as separate concepts, flow control and error control are implemented together. Three methods of implementation are stop-and-wait ARQ, go-back-N ARQ, and selective-reject ARQ. Stop-and-wait ARQ has the same performance disadvantage of the flow control scheme that it is based upon. Both go-back-N and selective-reject ARQ are based upon the sliding window flow control with one critical difference. Selective-reject ARQ resends only frames which receive a *nack* or which time out. Go-back-N ARQ resends the bad frame and all subsequent frames. Although selective-reject ARQ appears to be more efficient, in reality the performance improvement is often negligible. Because selective-reject ARQ requires more complex logic at both the send and receive ends, it is rarely implemented. For this project go-back-N ARQ was the reliability mechanism chosen and selective-reject ARQ was left as a possible future improvement.

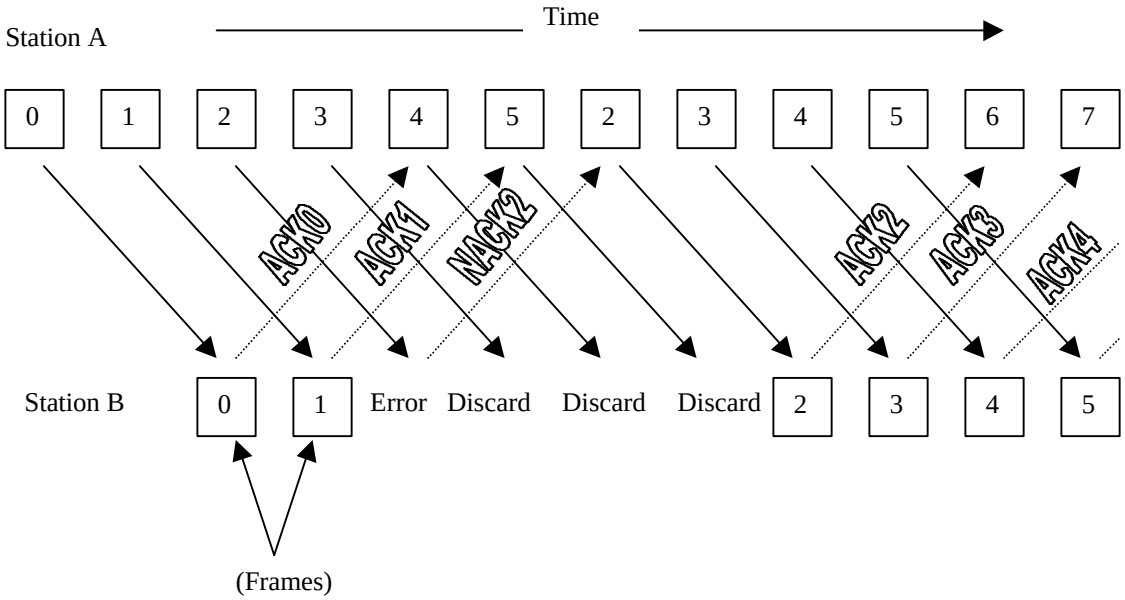


Figure 3: Go-back-N ARQ in action.

Figure 3 visually demonstrates how go-back-N ARQ handles an error during full-duplex communication. From station B's perspective, Frames 0 and 1 were received successfully and acknowledged, then an error occurred. Station B responded with a *Nack2* and then ignored all subsequent frames until the desired one was received. Since the expected frames followed next, the system came back on track. From station A's perspective, frames 0 through 5 were sent. During that time A*ck0* and A*ck1* were received verifying receipt of those frames. Then a *Nack2* was received, so frames 3 through 5 were resent getting the system back on track.

Table 1 explains how go-back-N ARQ handles the many different error scenarios that may occur. Frames, *acks*, and *nacks* may be damaged or lost in transit. When additional frames are to follow quickly, the system responds differently than when data transmission suddenly stops. For this reason, the two cases are presented separately.

Case 1: Additional frames to follow quickly.

	Damaged in transit	Lost in transit
Frame	B detects error and sends *nack*. A receives *nack* and retransmits all subsequent frames that it has transmitted.	B receives next frame, ignores that frame, sends *nack* for previous frame, and ignores all subsequently received frames until the proper frame is received.
Ack	*Ack* is ignored. Subsequent *ack* will probably do the job of the damaged *ack*. Otherwise A will timeout and retransmit the frame. B will receive the second copy of the frame, ignore that copy, and send a *nack* for the next frame implying successful receipt of the previous frame.	Subsequent *ack* will probably do the job of the lost *ack*. Otherwise A will timeout and retransmit the frame. B will receive the second copy of the frame, ignore that copy, and send a *nack* for the next frame implying successful receipt of the previous frame.
Nack	*Nack* is ignored. A will eventually timeout and retransmit the frame and all subsequent frames.	A will eventually timeout and retransmit the frame and all subsequent frames.

*Case 2: No additional frames to follow quickly***.**

	Damaged in transit	Lost in transit
Frame	B detects error and sends *nack*. A receives *nack* and retransmits all subsequent frames that it has transmitted. (Same as above)	A will timeout and retransmit additional frames.
Ack	*Ack* is ignored. A will timeout and retransmit the frame. B will receive the second copy of the frame, ignore that copy, and send a *nack* for the next frame. A will receive the *nack* for the next frame implying successful receipt of the first frame. Since no data is waiting in the output buffer no further action is taken.	A will timeout and retransmit the frame. B will receive the second copy of the frame, ignore that copy, and send a *nack* for the next frame. A will receive the *nack* for the next frame implying successful receipt of the first frame. Since no data is waiting in the output buffer no further action is taken.
Nack	*Nack* is ignored. A will eventually timeout and retransmit the frame.	A will eventually timeout and retransmit the frame.

Table 1: Go-back-N ARQ error scenarios when A transmits to B.

Go-back-N ARQ was the error and flow architecture of choice because it satisfies the project goals best. First, it provides guaranteed delivery as evidenced by all the scenarios in Table 1. Second, go-back-N ARQ maximizes performance thanks to two factors: (1) multiple frames can be in transit simultaneously minimizing waiting time; and (2) negative acknowledgements quickly indicate to the sender when an error occurs. The error and flow control system has no impact on the third goal of future integration with SCALE, since error and flow control is transparent to the application layer [Stallings94].

3. Implementation

Once the guaranteed delivery architecture has been decided upon, the details of implementation can be dealt with. This section first gives an overview of how the many operations interconnect and then uses flow diagrams to illustrate in great detail how the operations work.

3.1 Implementation Overview

The error and flow control was implemented on top of Fore System's Application Programmer Interface (API). The Fore API provides an easy to use interface with the hardware. It provides simple functions that establish a connection, send data, receive data, and close a connection [Fore93]. Using the Fore API, go-back-N ARQ was implemented in a full-duplex fashion with a separate send stream and receive stream. Although separately executed on a given computer, these separate streams are tied together in critical ways. When the receive stream receives *acks*, the receive stream must let the send stream know that an *ack* has arrived. Also the receive stream is responsible for sending *acks*, *nacks*, and data retransmits. The send stream will need to be paused for a moment while the receive stream completes these tasks. The send and receive streams consist of the following operations:

- Send stream
  1. "Copy In"
  2. "Empty Output Buffer"
  3. "Send Frame"
  4. "Timeout Thread"

- Receive stream
  1. "Copy Out"
  2. "Fill Input Buffer Thread"
  3. "Receive Frame"

"Timeout Thread" and "Fill Input Buffer Thread" run continuously in the background. The other operations are presented in a way that resembles the actual code and is conceptually easiest to grasp. Figure 4 shows how all of these operations interconnect. A send stream and receive stream run on each computer simultaneously. Guaranteed ATM divides the application layer's messages into frames of data. The division of frames, *acks*, and *nacks* into smaller ATM cells is transparent to Guaranteed

ATM thanks to the Fore API. The Fore API accepts the *acks*, *nacks*, and frames as single blocks and returns them as single blocks. All of the operations will be discussed in greater detail later.

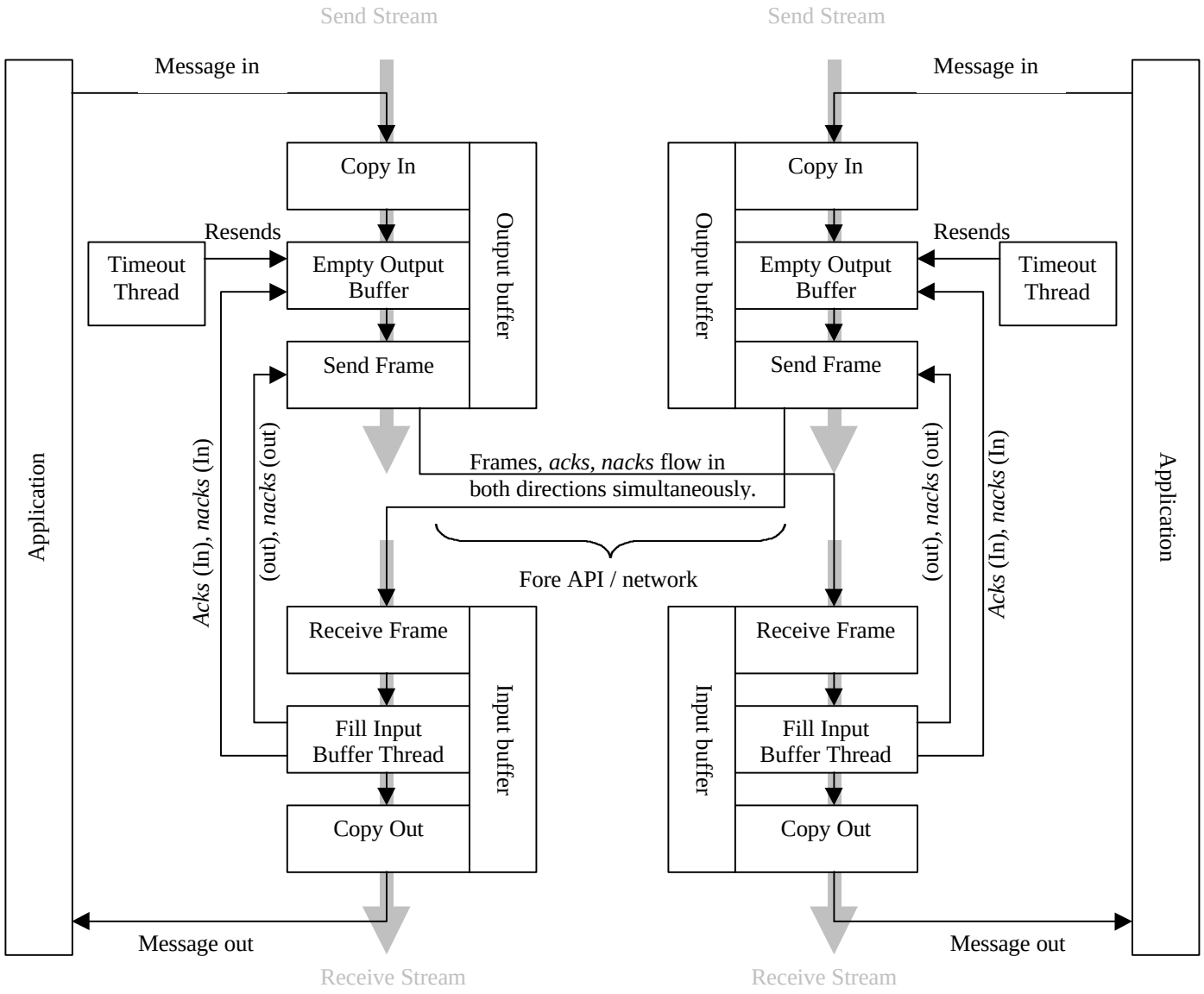
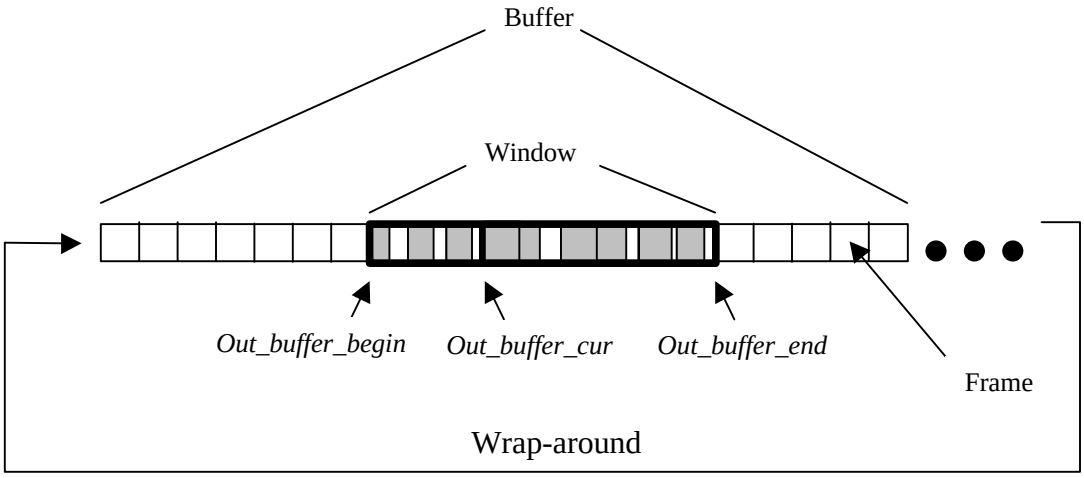
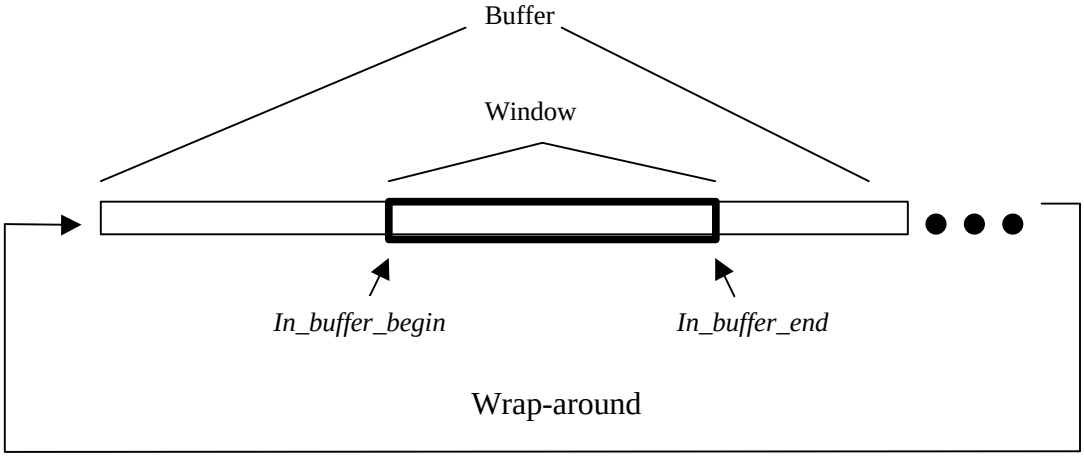


Figure 4: Full-duplex interconnection of operations.



(a) Structure of output buffer



(b) Structure of input buffer

Figure 5: Stucture of (a) output and (b) input buffers

Figure 5 shows the structure of the output and input buffers. The grey areas indicate partially filled frames in the output buffer. Partially filled frames are sent immediately instead of waiting for additional data to avoid delays. Data that has been sent, but not acknowledged, is between the *out_buffer_begin* and *out_buffer_cur* marker. Unsent data is between the *out_buffer_cur* and *out_buffer_end* markers. The output buffer must maintain the frame structure because it may need to retransmit a frame, but the input buffer is not structured with frames since data is not returned to the application in frame form. The application does not care which frame brought the data. It only cares to receive the correct data in the correct order. The input data is simply placed contiguous to previously received data. The received data that has not been retrieved by the application can be found between the two markers. Both buffers will wrap around to the beginning when reaching their physical end.

3.2 Send Stream

The send stream immediately sends data as it is copied into the output buffer. Then a continuously running timeout thread checks to see if any data has been sitting in the buffer for too long without an acknowledgment. If so, then the data is retransmitted.

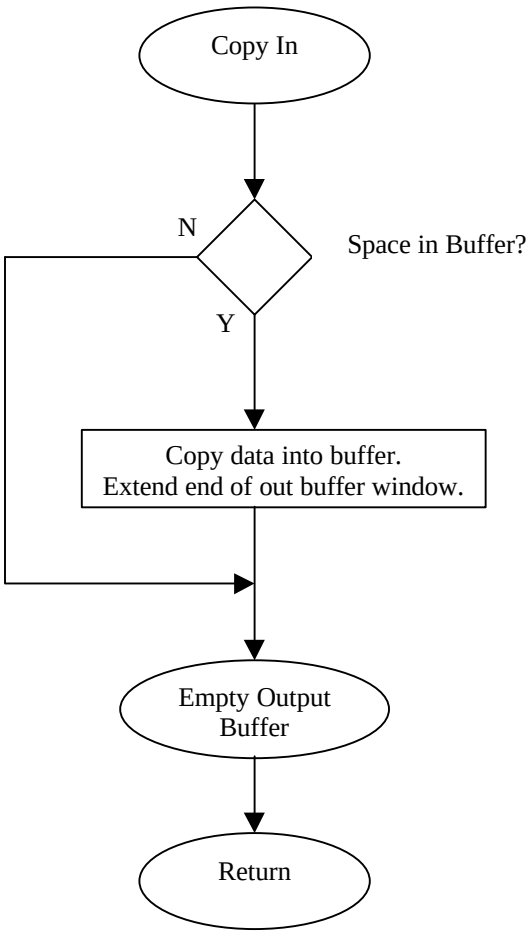


Figure 6: The "Copy In" function

3.2.1 "Copy In" function

Figure 6 shows a flow diagram of the "Copy In" function. Its purpose is to copy data into the send buffer. "Copy In" is called only from the application layer.

The function first checks to make sure that enough room is available in the buffer. If not then it returns with a flag indicating that the buffer was full. If enough room is available, then the function copies the data into the output buffer and extends the *out_buffer_end* marker (see Figure 5a). Once finished copying, the "Copy In" function calls the "Empty Output Buffer" function and then returns control back to the application layer.

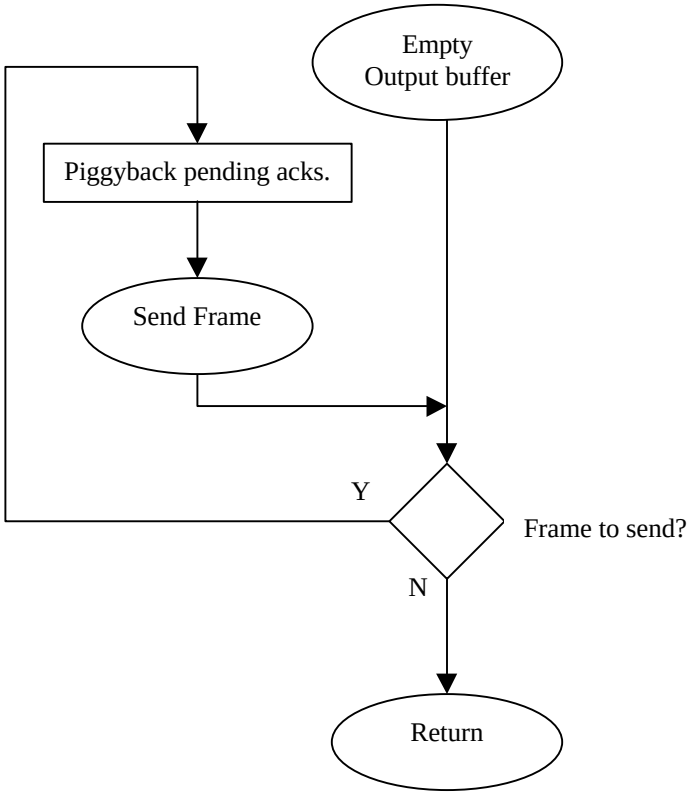


Figure 7: The "Empty Output Buffer" Function

3.2.2 "Empty Output Buffer" function

Figure 7 shows a flow diagram of the "Empty Output Buffer" function. This function's purpose is to send all of the unsent data. It may be called when "Copy In" receives new data, when "Fill Input Buffer Thread" receives a *nack*, and when "Timeout Thread" has a data timeout.

The unsent data can be found between the *out_buffer_cur* marker and the *out_buffer_end* markers (see Figure 5a). The function first checks to see if any frames are unsent. If so, then it piggybacks any acknowledgements that are pending (see Section 5.2.3), and then calls the "Send Frame" function. After the frame has been sent, the "Empty Output Buffer" function repeats the process until all unsent data has been transmitted.

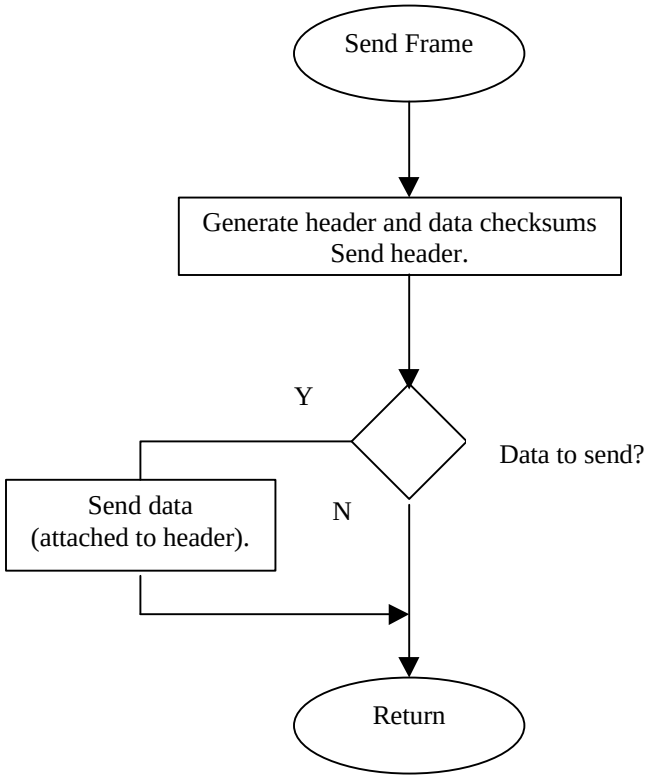


Figure 8: The "Send Frame" function

3.2.3 "Send Frame" function

Figure 8 shows a flow diagram of the "Send Frame" function. The "Send Frame" function's purpose is to generate a checksum and send whatever is passed to it. It may be called when "Empty Output Buffer" has a frame to send, or when "Fill Input Buffer Thread" has an *ack* or *nack* to send.

"Send Frame" generates the header and data checksums, and then sends entire frame out to the network using the Fore API.

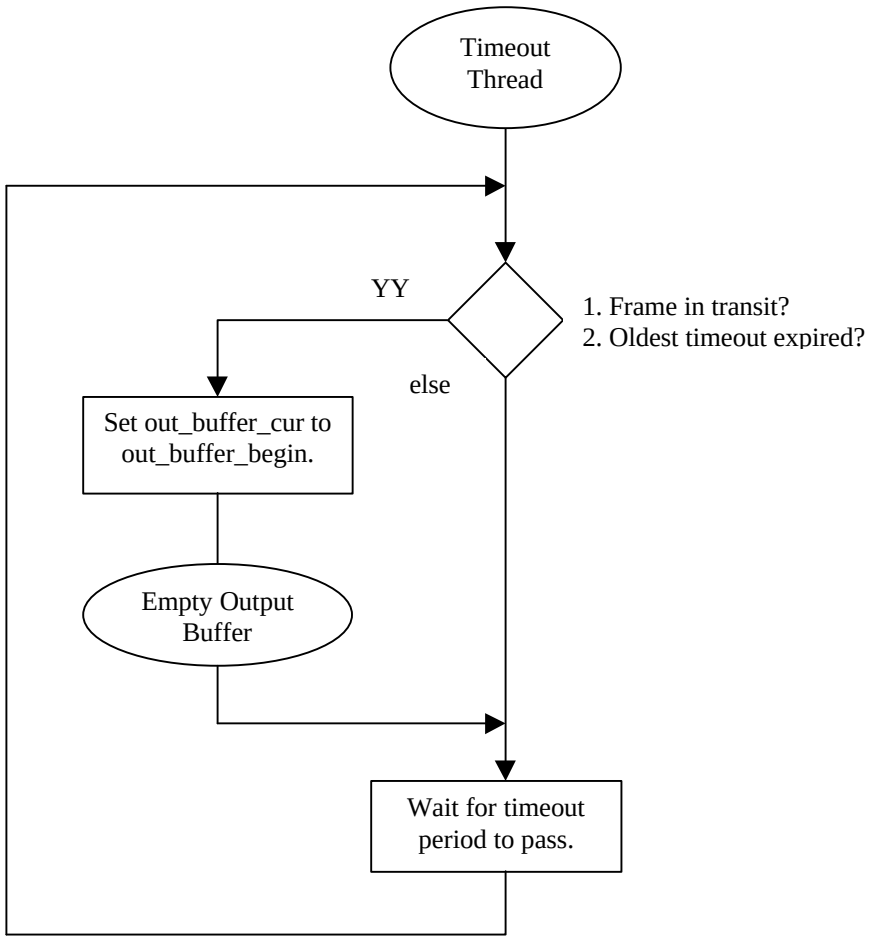


Figure 9: The "Timeout Thread"

3.2.4 "Timeout Thread"

Figure 9 shows a flow diagram of the "Timeout Thread." It is charged with checking to see if any timeouts have occurred. It runs continuously in the background.

The "Timeout Thread" checks to see if the oldest sent data has been awaiting an *ack* for a specified number of seconds. If so, then *out_buffer_cur* is set back to *out_buffer_begin* (Figure 5a). Moving the *out_buffer_cur* marker effectively tells the send stream that all of the data in the buffer awaiting an *ack* needs to be resent. "Empty Output Buffer" is called next to actually retransmit the data. After performing this check, the thread sleeps for the timeout period and repeats.

3.3 Receive Stream

The receive stream continuously retrieves frames off of the network as they arrive and sends the appropriate *ack* or *nack* response to those frames. Any *acks* or *nacks* that are received with the data are also dealt with by the receive stream.

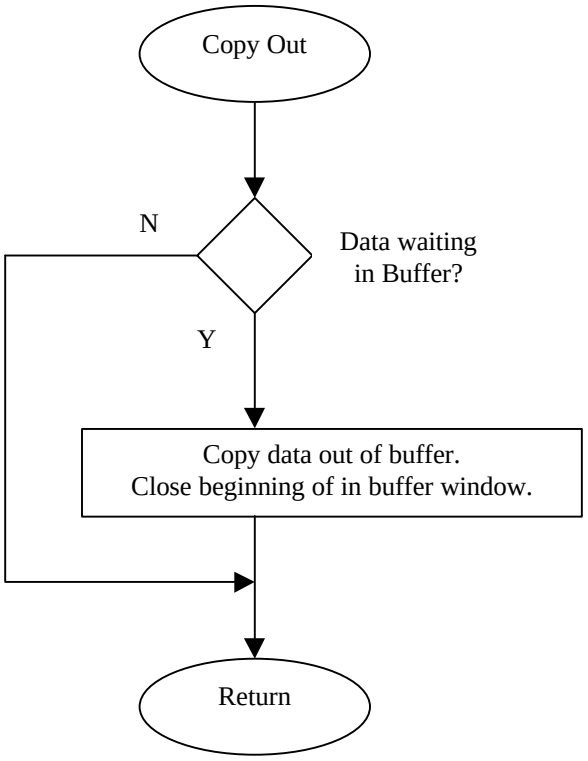


Figure 10: The "Copy Out" function

3.3.1 "Copy Out" function

Figure 10 shows a flow diagram of the "Copy Out" function. The function's purpose is to copy data out of the receive buffer to the application layer. It is called directly by the application layer.

The function simply checks to see if enough data is waiting in the input buffer. If enough data is not available, then a flag is returned to application layer. If enough data is available, then it is copied out and *in_buffer_begin* is increased to free the newly available buffer space (see Figure 5b).

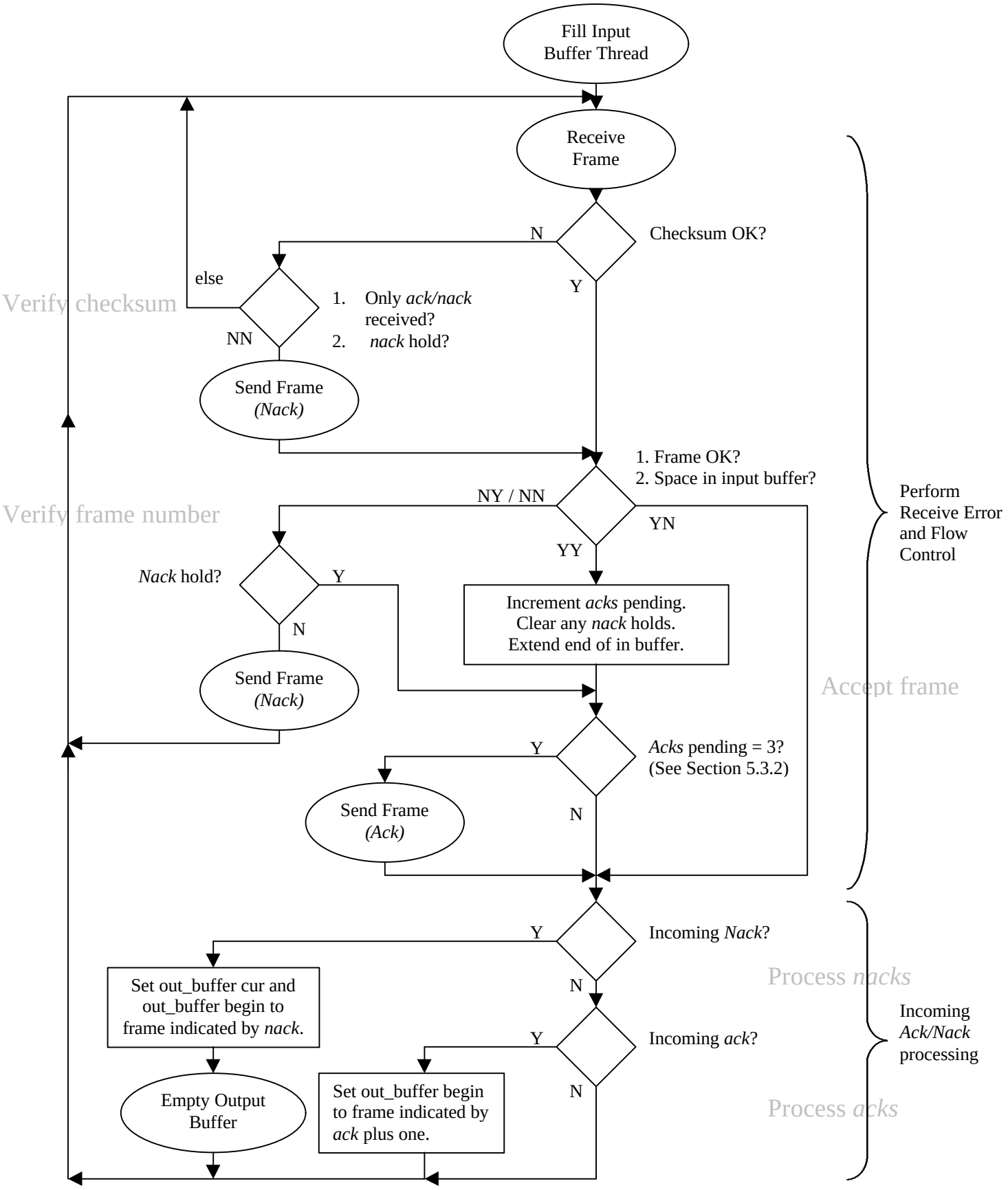


Figure 11: The "Fill Input Buffer Thread"

3.3.2 "Fill Input Buffer Thread"

Figure 11 shows a flow diagram of the "Fill Input Buffer Thread." The thread is charged with receiving data from the network. It runs continuously in the background in order to receive all frames as they arrive.

This thread is the most complex of all of the operations. Most of the critical error and flow control decisions are made here. Only a brief explanation of the thread's parts is attempted. For more detailed information, Figure 11 should be analyzed. The checksum status of an incoming frame was passed to this thread from the "Receive Frame" function. This thread first checks the checksum status of the incoming frame. If only an *ack* or *nack* is received with a bad checksum, then a nack should not be sent since no frame retransmission is necessary. Next the frame number is verified if any data is received. Incoming information will not contain any data if only an *ack* or *nack* is received. Finally, incoming *acks* and *nacks* are processed. If an *ack* is received, the *out_buffer_begin* variable will be increased to the indicated frame in order to free buffer space. If a *nack* is received, both the *out_buffer_cur* and *out_buffer_begin* variables will be set to the indicated frame and "Empty Output Buffer" will be called. One other contingency should be noted. If the input buffer is full, information is still retrieved off of the network. If all reception was simply stopped, the send stream would not receive any *acks* or *nacks* freezing output as well. Any payload contained in these intercepted frames is simply thrown out since the error control mechanism will later provide a correction.

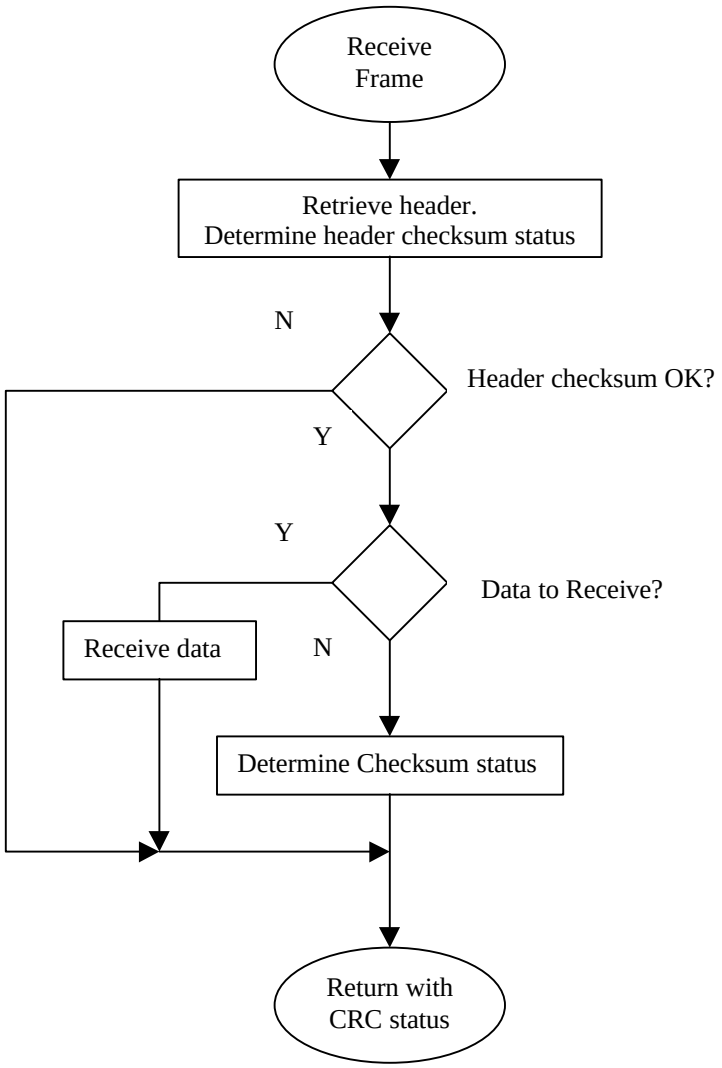


Figure 12: The "Receive Frame" function

3.3.3 "Receive Frame" function

Figure 12 shows a flow diagram of the "Receive Frame" function. The function's purpose is to retrieve data off of the network and determine its checksum status. It is called only from the "Fill Input Buffer Thread" operation.

This function first retrieves the header off of the network and determines its checksum status. If the checksum is bad, the function will return with a flag. If the checksum is successfully verified, then the header contains the length of the data to follow. If any data is to follow, then it is also retrieved off of the network and its checksum status is checked and returned to the "Fill Input Buffer Thread."

4. Verification

Since things do not usually work out as well in practice as they do in theory, some method of verifying the go-back-N ARQ implementation must be employed. Only after surviving a series of stress tests will the system prove that it is worthy of the description "guaranteed delivery." The following stress tests attempt to simulate the behavior of the application and network layers under normal and extreme conditions. The tests are run using two different frameworks. The "Half-Duplex Framework" uses one thread to alternate between sending and receiving a fixed number of messages in a row. More realistic data patterns are sent through the "Full-Duplex Framework." These streams employ full-duplex communication through the use of two simultaneously running threads, one for sending and one for receiving. Under both of these systems, extreme conditions are tested in which damaged data and lost data are simulated. Once the Guaranteed ATM survives these tests, the algorithms making up the go-back-N ARQ architecture are verified and performance improvements become the primary focus.

4.1 Test Application

This section describes two different test frameworks and some of the bugs that they uncovered. Within these two frameworks, normal conditions and error prone conditions are simulated. In order to allow detection of bad data at the application layer, a counter is used at the send and receive sides. The number on the counter is sent as the data and then incremented. The receive end then compares the data received to its counter to see if incorrect data is passed.

4.1.1 Half-Duplex Framework

The "Half-Duplex Framework" involves using a single thread to send and receive a fixed number of frames. Although real traffic may never actually look like this, these tests offered a good method of verifying algorithms. The principal advantages of this framework over running two separate send and receive streams is that the tester knows exactly what the receiver should be receiving and in exactly what order. With two simultaneously running send and receive threads, each side sends an undetermined of number of frames before receiving an undetermined number of frames. The single stream method also makes pausing the test when an error occurs much simpler since two threads do not need to be stopped simultaneously. This testing framework is used to verify mechanisms such as buffer copies, sliding window protocols, header assembly and disassembly, error detection, *ack* and *nack* mechanisms, timeouts, and many others. Almost all algorithm bugs are uncovered using these simple tests.

One of the most difficult software mistakes to correct was the input buffer overrun bug. This bug occurred when data was received past the end of the input buffer without wrapping around to the beginning. The bug was difficult to detect because it did not surface until long after it had been created and only showed when frames of certain

lengths were sent. The overrun also did not happen at the location where it seemed to occur. When frames of a certain length were tested, the system would return incorrect data to the application layer at seemingly random times. Isolating the location of the problem implied that something was wrong at the receive end with the "Copy Out" function. Initial inspection did not reveal any deficiencies. After verifying that the data was received correctly, suspicion of the "Copy Out" function grew. It turns out that a scenario had not been taken into account when modifying the structure of the input buffer. The initial structure of the input buffer consisted of placing received frames into a space reserved for that frame number even if the entire space was not used. With this structure, room was always available at the end of the input buffer for one entire frame. Unfortunately this system did not make very good use of the space in the buffer since enormous gaps could be left. The buffer was restructured so that all data was stored contiguous to the data already received. This restructuring allowed for a subtle bug when data was received at the physical end of the buffer. The "Receive Frame" function would overrun the data past the physical end of the buffer without wrapping around. The solution employed involved allocating extra buffer space at the physical end of the buffer past the wrap around point. Now when the data overrun scenario occurrs, the "Fill Input Buffer Thread" copies the overrun data into the physical beginning of the buffer. Numerous algorithm, flow control, and error control bugs such as this were uncovered using the "Half-Duplex Framework."

4.1.2 Full-Duplex Framework

The "Full-Duplex Framework" involves using two separate threads on both the host and client computers to simultaneously transfer data in both directions. Although debugging is more difficult with this framework, the traffic patterns are more realistic. This framework is best used after a test has succeeded in the half-duplex framework in order to minimize debugging. New complications emerge in full-duplex communication such as when both the send and receive streams of data on a computer share resources. An example involved the *out_buffer* markers.

Both the send stream of data and the receive stream of data use the *out_buffer_begin* and *out_buffer_cur* variables. The send stream uses the *out_buffer_begin* variable to determine the current size of the buffer and the *out_buffer_cur* variable to know which frame to send next. The receive stream may modify these variables upon receipt of an *ack* or *nack* and may then depend on these variables during frame resends. If full-duplex communication is used, then one stream may change one of these variables while the other stream depends on that variable remaining constant. This conflict does not arise in the "Half-Duplex" framework since the send and receive streams do not ever run simultaneously. (I.e. At the point when this bug was discovered, "Fill Input Buffer Thread" and "Timeout Thread" were not yet continuously running threads). In order to prevent this conflict, the functions that use these variables were kept from running at the same time through the use of mutexes. A mutex is simply a flag that indicates that a resource such as a variable is being reserved.

The lost header bug was another bug recognized through the "Full-Duplex Framework." The lost header bug occurred due to the fact that the header and body of

the data were sent separately. What happened when header was lost was that the first 10 bytes of the payload were assumed to be the header. Then the last 10 bytes of the data would include the next header. The system would never resynchronize itself. This scenario did not pass any corrupt data to the application layer since the checksum did not allow any data to pass into the input buffer, but the scenario did cause a freeze-up. In order to protect against this bug, separate checksums were used for the header and body instead of a combined checksum. In addition, the header was turned into a 16-byte block instead of a 10-byte one. With this setup a header first did not pass its checksum test. Then the data was retrieved in 16-byte increments until it was all gone and a correct checksum was found with the next header. The 16-byte data gobbling worked because the data was always passed in increments of 64 as discussed in Section 6. Although this bug would also show up in the "Half-Duplex Framework," it was not discovered until the higher traffic patterns of the "Full-Duplex Framework" were tested.

4.2 Extreme Condition Tests

Extreme condition tests are used to verify that the system responds correctly to adverse conditions. These conditions include network errors in which information is damaged or lost, and application anomalies in which the application layer behaves in an irregular fashion.

4.2.1 Simulation of Network Errors

Two types of network errors are simulated, damaged data and lost data. Damaged data is simulated by altering checksums immediately before data is sent. This alteration causes the receiver to think that the data was corrupted in transit since the checksum does not pass. The alteration occurs at random times at a predetermined frequency so that many error patterns are simulated such as may occur in a low-quality network connection. Lost data is simulated in a similar fashion by intercepting the frames immediately before they are sent and skipping the send. This dropped frame error also occurs at random times at a predetermined frequency.

Figures 13 and 14 show how Guaranteed ATM's effective throughput changes at various damaged frame and lost frame frequencies. These tests were run part-way through the project using a slightly old version of the Guaranteed ATM, but still provide a good feel for its error response. The most significant delays occur when the system depends on timeouts, such as when a *nack* does not reach its destination. Under normal conditions, this is a rare scenario since two errors have to occur back-to-back. First a frame has to be lost or damaged and then the *nack* that follows would have to be damaged. The bars in Figure 14 do not decrease regularly because of the nature of the measurement. The long delays caused by depending on timeouts provide irregular results. A larger set of samples would have helped identify stray measurements.

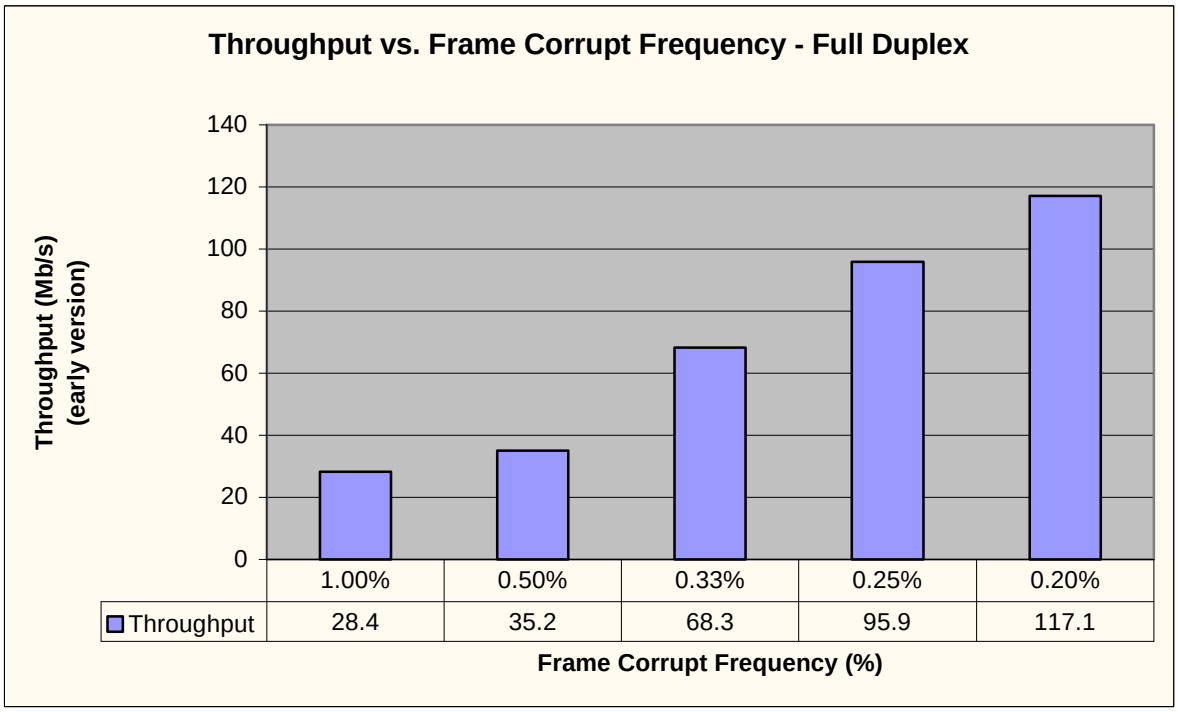


Figure 13: Guaranteed ATM's performance response to frame corruptions.

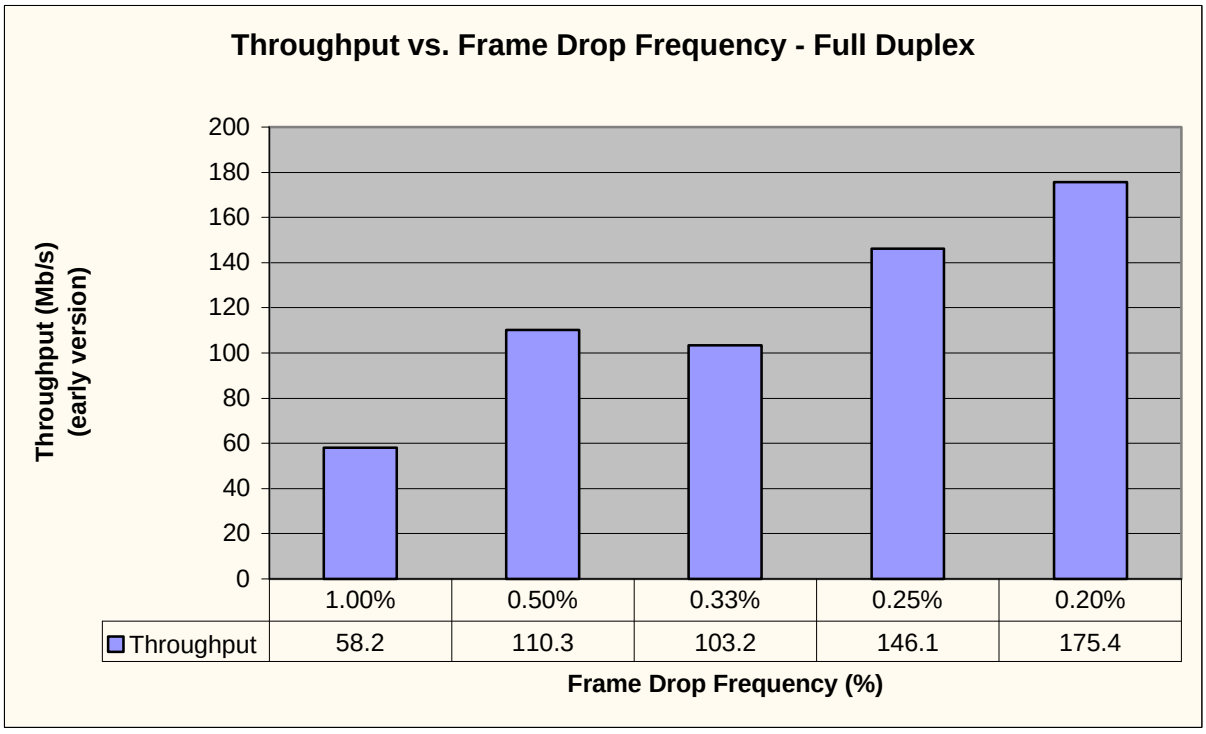


Figure 14: Guaranteed ATM's performance response to dropped frames.

4.2.2 Simulation of Application Anomalies

Two types of application anomalies were simulated, random delays and random message sizes. When these two tests are run together, a good simulation of real data patterns results. Unusual situations such as a long pause between sends in one direction may occur. When a long delay happens, Guaranteed ATM responds in different ways than if data follows immediately as described in Section 2.3. The random size send is a good test of buffer copy algorithms. These tests were run successfully near completion of the project. Although they did not turn up any new bugs, running the tests was worthwhile since they increased confidence in the system's ability to deliver guaranteed correct data.

5. Optimization

High performance is an important complement to a bulletproof system. Once the system proves itself to be reliable, performance enhancements are focussed upon. All performance tests use a pair of 200-MHz UltraSPARC workstations with 256 MB of memory and *ForeRunner* SBA-200 ATM network adapters. The workstations are connected by through a *ForeRunner* ASX-200BX ATM switch at 155 Mb/s. The setup is depicted by Figure 15. This section first describes the variables that can be measured and possibly used to identify bottlenecks. Next, half and full-duplex performance optimization are discussed. Finally, several performance benchmarks are presented.

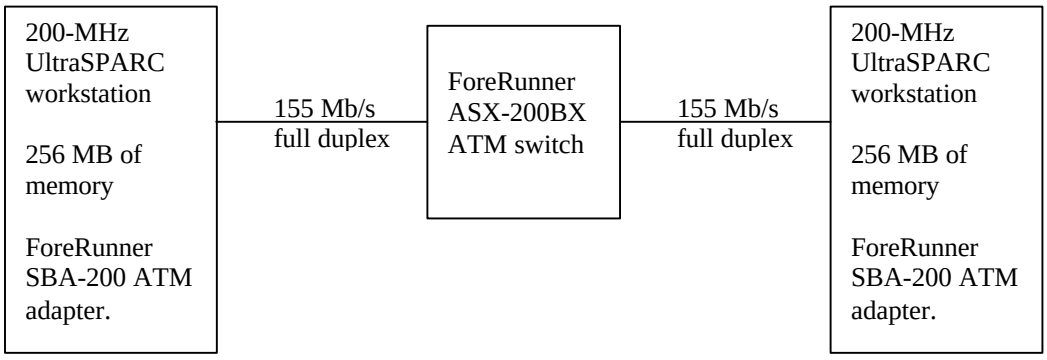


Figure 15: Testbed Setup

5.1 Variables

In order to decide what improvements to make and where to make them, numerous variables were varied and measured. These variables are the lens through which the network's performance is measured. Once a time hog is found, that part of the system is optimized. Table 2 shows the independent variables altered and the dependent variables analyzed.

Variable	Type	Description
Send/Recv length	Ind	Amount of data in sends and receives.
Send/Recv Num	Ind	Number of times each to send and receive.
Frames sent	Ind	Number of frames sent, including resends.
Acks received	Dep	Number of *acks* received (sum of forced and piggybacked *acks* received).
Acks received, forced	Dep	Number of forced *acks* received. (See Section 6.2.3)
Acks received, piggybacked	Dep	Number of piggybacked *acks* received.
Nacks sent	Dep	Number of *nacks* sent.
Nacks Received	Dep	Number of *nacks* received.
Frame eat frequency	Ind	Frequency that frames are dropped.
Checksum alter frequency	Ind	Frequency that checksums are altered.
Frames eaten	Dep	Number of frames dropped.
Checksums altered	Dep	Number of checksums altered.
Receive errors	Dep	Number of receive errors (sum of bad checksums and framing errors).
Bad checksums	Dep	Number of bad checksums received.
Framing errors	Dep	Number of framing errors received.
Frames ignored C	Dep	Number of frames ignored because of bad checksums.
Frames ignored F	Dep	Number of frames ignored because of bad frames.
Timeouts	Dep	Number of data timeouts that occur.
Time	Dep	Time it takes to complete send of given number of frames of given length.
Throughput	Dep	Effective Throughput - Number of bits that were sent and received per second by the application layer.
Latency	Dep	Time it takes for a frame to go from the application layer of one computer to the application layer of the other.
Latency breakdown times	Dep	Time it takes to get through any given segment of code.

Ind = Independent, Dep = Dependent.

Table 2: Variables used for performance optimization

5.2 Half-Duplex Performance

The majority of optimizing occurred while the system was still operating in half-duplex mode. These optimizations carried through to the full-duplex communication when the conversion was made.

5.2.1 Buffer Copy Optimization

The first version of the go-back-N ARQ did not run with half-duplex throughput speeds greater than 15 Mb/s. After analyzing the breakdown times of the various segments of code, it was clear that the buffer copies were a significant source of the latency. The two buffer copies were taking over 500 μs each. Commenting out the buffer copies provided a quick test of how much throughput would improve if this segment of code was sped up. This test showed that performance would increase to as much as 80 Mb/s.

For simplicity the buffer copies were initially implemented by copying one character at a time. It turns out that the HCS Lab faced this challenge before with Myrinet networks. The answer was a highly optimized direct memory-to-memory copy created in assembly code. This memory-to-memory copy was designed to copy a memory block of 64 bytes in the least time [SPARC94]. After making the change at both the send and receive ends, the buffer copy latencies fell to below 40 μs, and half-duplex throughput shot up to about 75 Mb/s.

5.2.2 Checksum Optimization

The checksum segment of code was the next significant source of latency optimized. The initial implementation faced the same weaknesses that the buffer copy first faced. The data was being added up one character at a time. Commenting out the checksums foretold a performance increase to around 110 Mb/s.

Fortunately, the checksum does not have to be the actual sum of all the bytes of data. As long as all of the data is included in some sort of sum and that same summation is performed on both the send and receive side, error detection will work. The solution was to add up the data in the largest chunk possible, eight bytes at a time. Since only four bytes of the checksum could be sent, the final sum needed to be reduced to four bytes. Throwing out the top digits of the sum would effectively be the same as not providing a checksum for half of the data. To preserve all of the checksum information in a four byte number, the most significant four bytes of the sum and the least significant four bytes of the sum were added together. Optimizing the checksums sped up half-duplex throughput to 100 Mb/s.

5.2.3 Acknowledgement Optimization

A common optimization involves the acknowledgements. Every frame requires an *ack*, and sending all of these *acks* by themselves eats up precious bandwidth. Piggybacking attempts to minimize the impact that *acks* have on throughput. Instead of sending the *acks* separately, the *acks* are piggybacked onto outgoing data. A problem occurrs, however, when data is flowing in only one direction. The sender sends all of the frames that it is allowed to send, but is not able to send more right away because it has not received any *acks*. Eventually the sender would timeout, retransmit, and receive a *nack* that does the job for the unsent *acks* as described in Section 2.3. Since this process takes far too long, an *ack* force mechanism was implemented in addition to the *ack* piggybacking.

The idea behind the *ack* force mechanism was to force *acks* periodically to prevent the slow-down described earlier. Several methods were tried. One involved a timer that would check at regular times to see if any *acks* were waiting to be sent. It was, however, difficult to choose a time that would work well with both large and small sized frames. Another technique involved creating a thread every time an *ack* was to be sent. This thread yielded a certain number of times to give the send stream an opportunity to piggyback the *ack*. Then the *ack* thread forced the *ack* if it had not been piggybacked. This technique did not piggyback *acks* reliably even in high traffic situations. The mechanism settled upon involves an *ack* counter. The "Receive Thread" increments this counter whenever an *ack* is to be sent, but does not always send that *ack*. If this counter reaches a predetermined threshold, then the *ack* is forced. If an *ack* is piggybacked before the *ack* counter reaches its threshold, then the counter is reset back to zero. This technique ensures that the *ack* is sent in a timely manner, whether piggybacked or forced. It also allows the majority of *acks* to be piggybacked during high traffic when piggybacking is most important.

The next issue is how high to set the counter. Figure 16 shows that any number greater than one is large enough to maximize throughput. The experiment was run before the latest version of Guaranteed ATM was finished, but the curve still demonstrates the relationship between throughput and the *ack* counter. A value of three was chosen for the *ack* counter so that during the transmission of a window of data, at least two *acks* would be sent and a minimum number of *acks* would be forced.

Half-duplex communication with all *acks* being piggybacked allowed a throughput of about 126 Mb/s. Some half-duplex performance was later lost when the system was adapted to run in full-duplex due to synchronization waits. Half-duplex performance, however, still peaked at 119 Mb/s as discussed in Section 6.

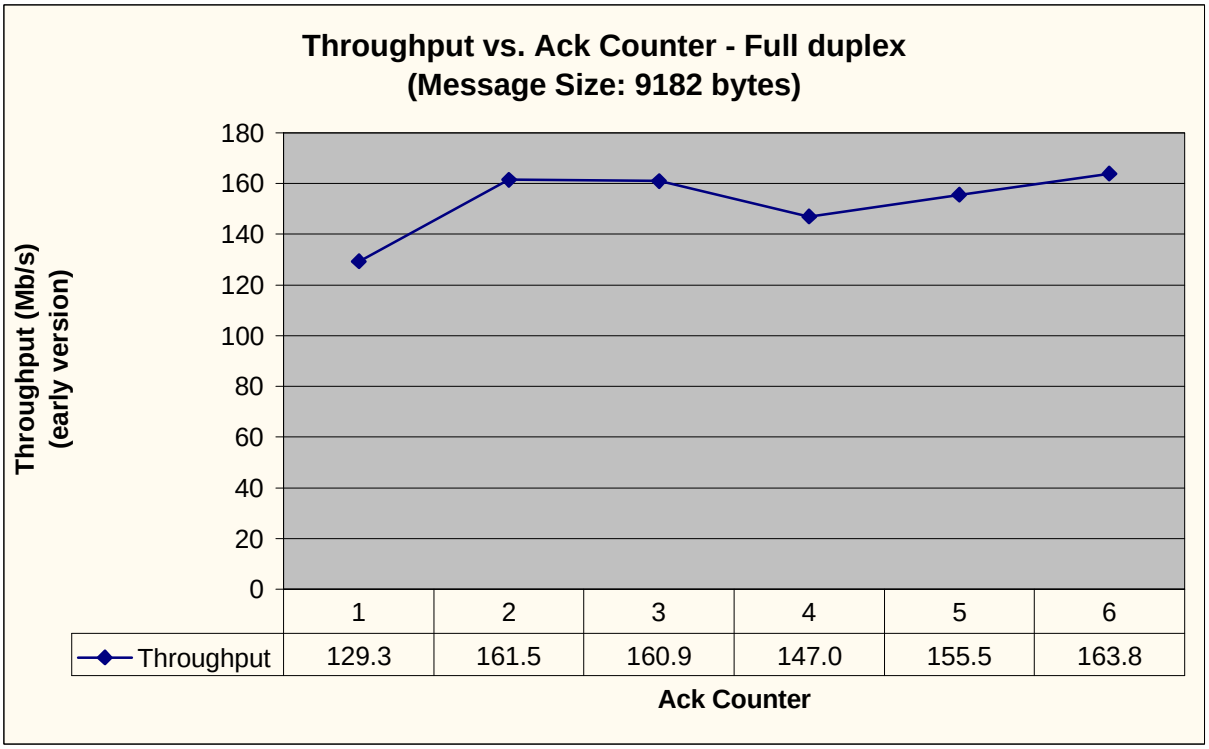


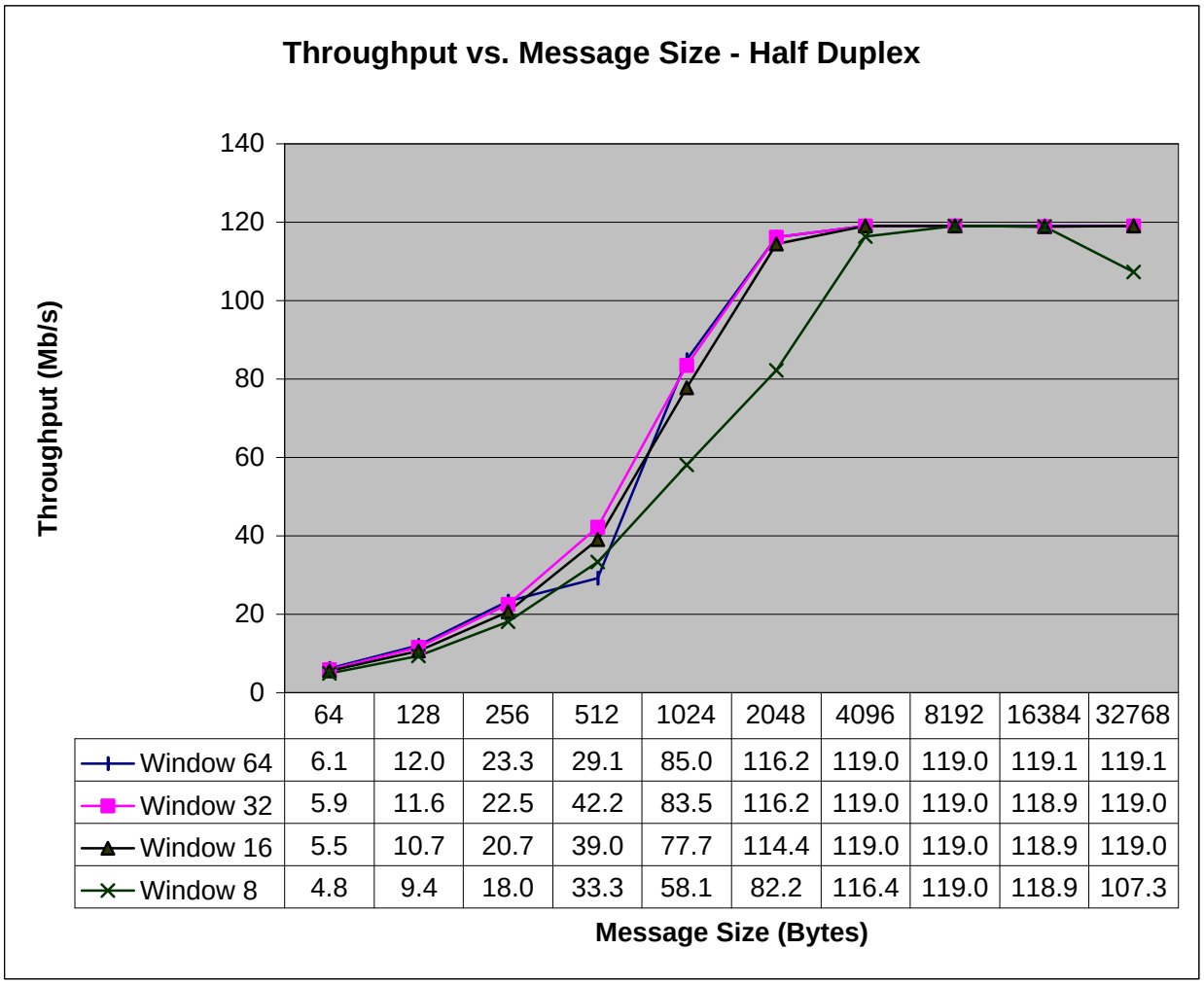
Figure 16: Guaranteed ATM's response to selective acknowledgments.

5.2.4 Header Optimization

The header was originally sent separately from the payload wasting bandwidth. This optimization combined the header with the payload. The header was initially sent separately for two reasons: (1) it contained the length of the payload to follow; and (2) leaving space for the header in the output buffer would provide too many complications at once. Walking through the process of sending a combined header and payload will help to illustrate why the first reason provides no obstacle. The header and payload are sent together with a single invocation of the Fore API *atm_send* function. On the receive side, the header is received first, its checksum is verified, and then the length of the data to follow is determined. Next, the indicated number of bytes are requested. Leaving space in the output buffer for headers proved to be challenging, but was possible. At the beginning of each frame's buffer space, room was left for the header. The end of the header was 64-byte end-aligned, so that the payload would be 64-byte aligned allowing the memory copy optimization to function. The combined header optimization improved performance dramatically for small-sized frames almost doubling throughput.

5.2.5 Window Optimization

The final optimization involved choosing the appropriate window size. The Fore API allows messages of 9182 bytes to be sent at one time. A window size of eight would therefore require buffers of size 8 x 9182 = 73456 bytes. Doubling the window size to 16 would double the required buffer size. The size of the buffer begins to get out of hand quickly, so experiments must be run to determine what size buffer provides near maximum performance with minimum memory usage. Figure 17 shows that a window size of 16 is the smallest size that will provide maximum performance, so it was chosen.



	64	128	256	512	1024	2048	4096	8192	16384	32768
Window 64	6.1	12.0	23.3	29.1	85.0	116.2	119.0	119.0	119.1	119.1
Window 32	5.9	11.6	22.5	42.2	83.5	116.2	119.0	119.0	118.9	119.0
Window 16	5.5	10.7	20.7	39.0	77.7	114.4	119.0	119.0	118.9	119.0
Window 8	4.8	9.4	18.0	33.3	58.1	82.2	116.4	119.0	118.9	107.3

Figure 17: Effective throughput with different window sizes.

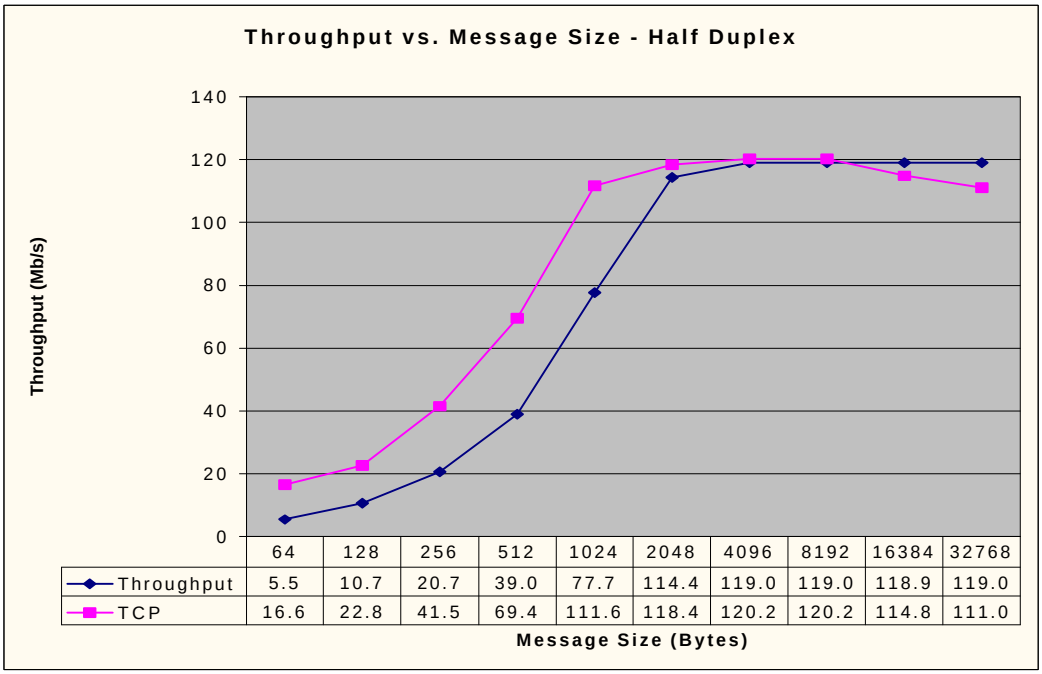
5.3 Full-Duplex Performance

Once half-duplex communication was optimized and the system was converted to full-duplex communication, little more optimization was required. After the conversion, the vast majority of issues that surfaced were reliability related. Once the reliability issues were corrected, full-duplex communication proved to yield higher numbers simply because of its nature.

In order to see what happens when migrating to full-duplex communication, imagine two water pipes with a sender at each end. One pipe is leaned so that balls placed in it roll in one direction, while the other sends balls in the other direction. Both of these water pipes exist in half-duplex communication, but you are only allowed to use one hand at a time to place balls in one tube and take them out of the other. Maximum throughput occurs in half-duplex communication when each side alternately sends an entire window of data and receives an entire window of data, so imagine dropping seven balls down one tube. At the other end your friend receives seven balls, and then sends one acknowledgement ball and seven data balls back. When going back and forth like this one pipe always remains unused, except for maybe some *acks*. Notice also that the pipe sending data does not lose any bandwidth to *acks*.

In full-duplex communication you are allowed to both send and receive at the same time, so you can use both hands. Now throughput is greater, but not quite twice as great, because each pipe also has to carry some *acks*. Still, full-duplex communication should achieve nearly double the performance of half-duplex communication. Since half-duplex reaches 119 Mb/s, full-duplex should approach 238 Mb/s. This prediction turns out to be reasonably accurate since full-throughput actually peaked at 216 Mb/s. This and other performance numbers will be further discussed in the next section.

6. Performance Benchmarks



	64	128	256	512	1024	2048	4096	8192	16384	32768
Throughput	5.5	10.7	20.7	39.0	77.7	114.4	119.0	119.0	118.9	119.0
TCP	16.6	22.8	41.5	69.4	111.6	118.4	120.2	120.2	114.8	111.0

Figure 18: Effective half-duplex throughput seen by the application.

Figure 18 shows how throughput varied with the size of the message. Performance drops off with smaller sized messages because of the added overhead processing. Much of the same code that runs during transmission of large messages, must also be run for small messages. Although this processing time remains constant, the amount of data delivered shrinks. Work done at Syracuse University shows that a maximum bandwidth of 121.8 Mb/s is delivered by Fore Systems API interface [Subra]. With maximum throughput of 119.0 Mb/s, Guaranteed ATM can take advantage of nearly all the available bandwidth.

The figure also compares Guaranteed ATM's throughput curve with the throughput curve of TCP, the internet's error control mechanism. Clearly TCP performance is superior. Cornell's Active Messages project originally achieved throughput results with the Fore API similar to this project's results [Active95]. They eventually optimized performance further by modifying the Fore API to work well with small message sizes. This change gave results far superior to those using the original Fore API [U-Net95].

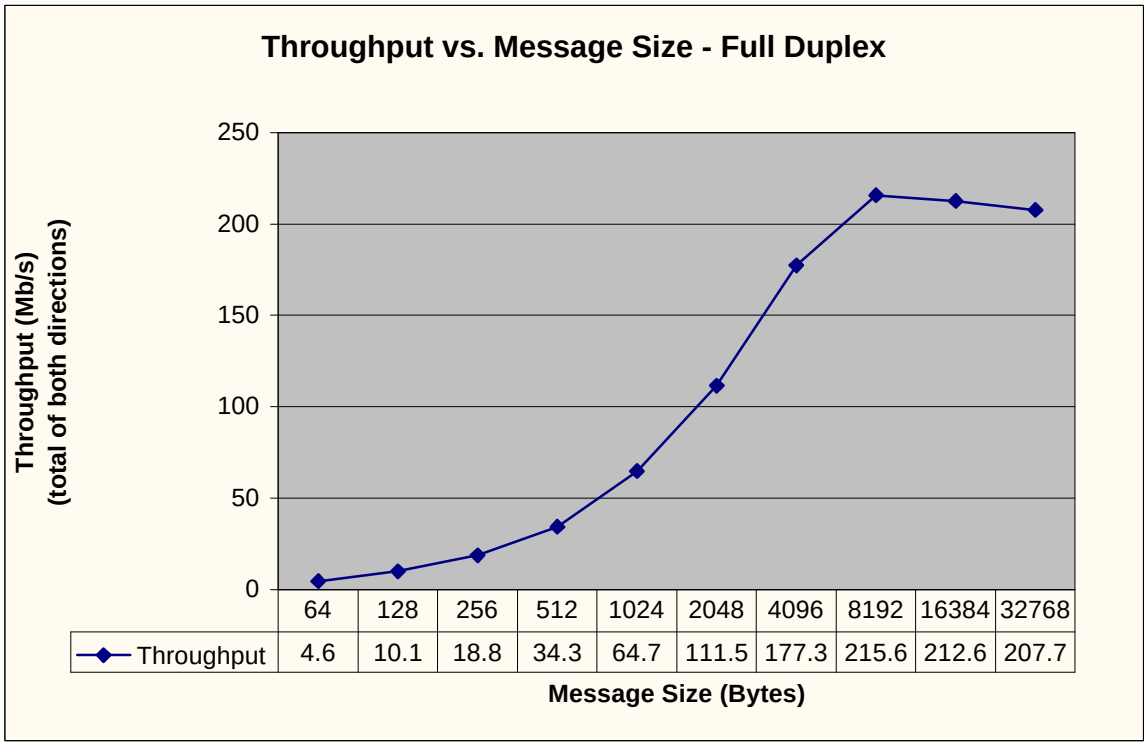


Figure 19: Effective full-duplex throughput seen by the application.

Figure 19 shows Guaranteed ATM's full-duplex throughput curve. The full-duplex throughput did not quite double the half-duplex throughput curve because the data must share bandwidth with *acks* (see Section 5.3). In addition, the full-duplex framework tends to send large numbers of messages in one direction before sending large numbers of messages in the other direction. This tendency reduces potential throughput numbers significantly since data is often flowing in one direction at a time.

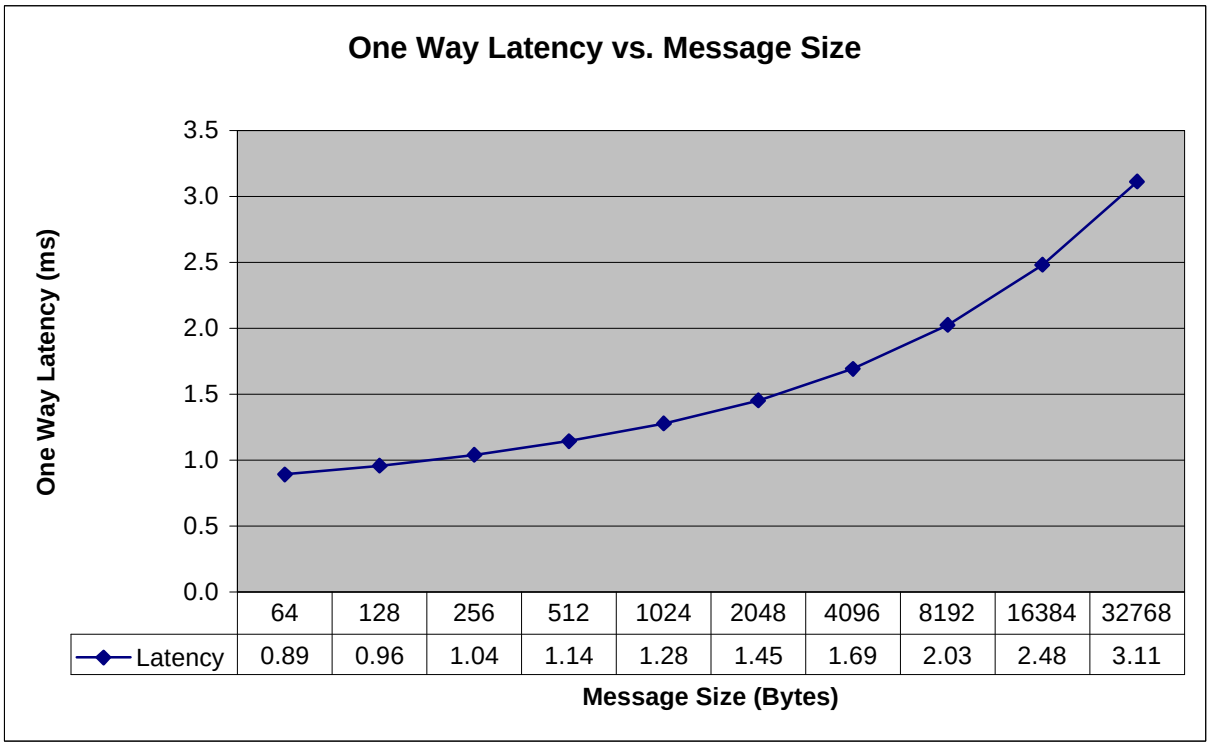


Figure 20: Latency seen by application

Figure 20 shows Guaranteed ATM's latency curve. One-way latency is found by dividing two-way latency in half. Latency increases with larger messages primarily because it takes longer to transmit more data. In addition it takes longer to perform the buffer copies, generate checksums, and verify checksums. The latency for 64-byte messages is comparable to that attained by other projects using the Fore API. Cornell University's Active Messages project shows that the Fore API yields a one-way latency of .85 ms without guaranteed delivery [Active95]. Guaranteed ATM's latency of .89 ms with guaranteed delivery seems to fall in line with Cornell's result. Cornell eventually optimized performance further by modifying the Fore API to work well with small messages. With these Fore API modifications Cornell achieved latencies of 71 µs without guaranteed delivery, and latencies of 157 µs with guaranteed delivery through TCP [U-Net95].

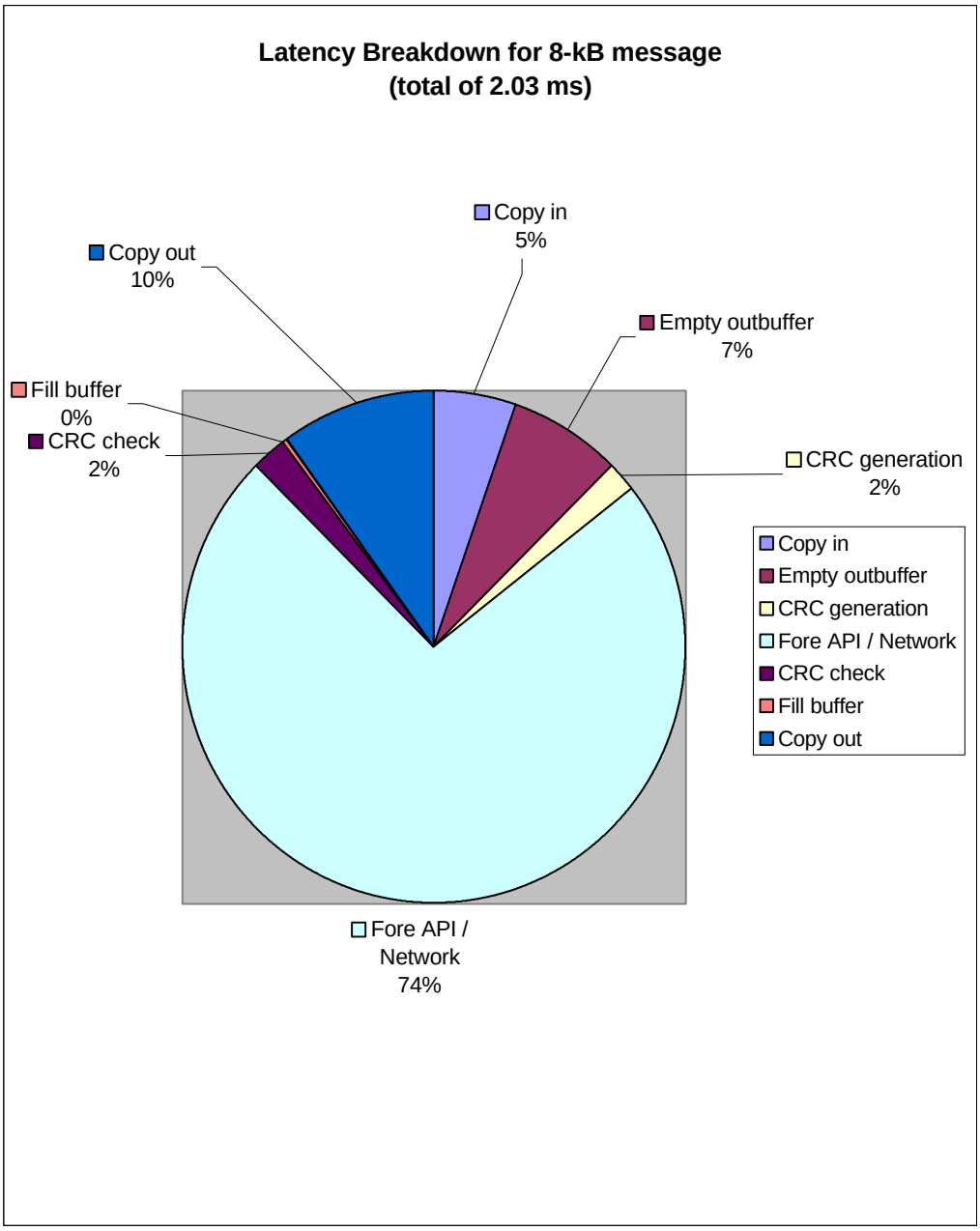


Figure 21: Breakdown of latency for 8-kB message.

Figure 21 shows how an 8-kB message spends its time in transit. These measurements do not account for process switches. Instead the measurements start at the entrance of an important piece of code and stop at the end of it. Three points should be noted. First, the frame spends a majority of its time in the Fore API and the network. Work done at University of Illinois at Urbana-Champaign discusses what the breakdown times should look like. They indicate that the network should take about 30% of the latency [Karam94, Lauria98]. Figure 21 does not give a direct comparison, but it does show that the Fore API and the network together take 74% of the latency. This

discrepancy provides evidence that modifying the Fore API would dramatically improve the total latency.

The second point to note is that some functions take longer than they should. The "Empty Output Buffer" function takes a significant amount of time even though it should take almost no time since it is pure logic. The "Copy Out" function also seems to take much longer than the "Copy In" function even though they are copying the same amount of data. These delays happen because of synchronization cost. An operation often has to wait for other operations to finish with the variables that it needs. An estimate of the synchronization cost would give a good idea of how much latency would improve with synchronization enhancements. Since the "Empty Output Buffer" function is all logic, all of the time it uses can be considered synchronization cost. The difference in the time the two copies take gives a minimum value for the synchronization cost at those functions since they should take the same time. This calculation gives a total of 12% as the minimum time lost to synchronization. To estimate the maximum, consider that the copy routines should not take much longer than the checksum routines since the same amount of data needs to be cycled through. This assumption gives a synchronization loss of 11% at the copy routines and 18% total. Thus synchronization cost can be estimated to be between 12% and 18%.

The third point of note involves a curious phenomenon. Although 26% of latency is within the Guaranteed ATM layer, Figure 18 shows that nearly all of the Fore API's bandwidth can be used. Maximum throughput for this message size is achievable due to the buffering being used. The buffering ensures that a continuous stream of data flows through the network thanks to a pipelining effect. While the Fore API sends one frame of data, the next frame is being prepared and stored in the buffer. Since Guaranteed ATM seems to take less time than the Fore API, additional data is ready before the Fore API finishes sending the previous frame. The buffer will always have a ready supply of data waiting for the Fore API during throughput measurements so the Fore API will constantly be kept busy. With latency measurements, however, only one message is sent at a time, so additional data will not be awaiting the Fore API. Thus with latency measurements buffering does not mask the effects of a slow component in the system. Although the two performance variables are highly interrelated, they both provide different information.

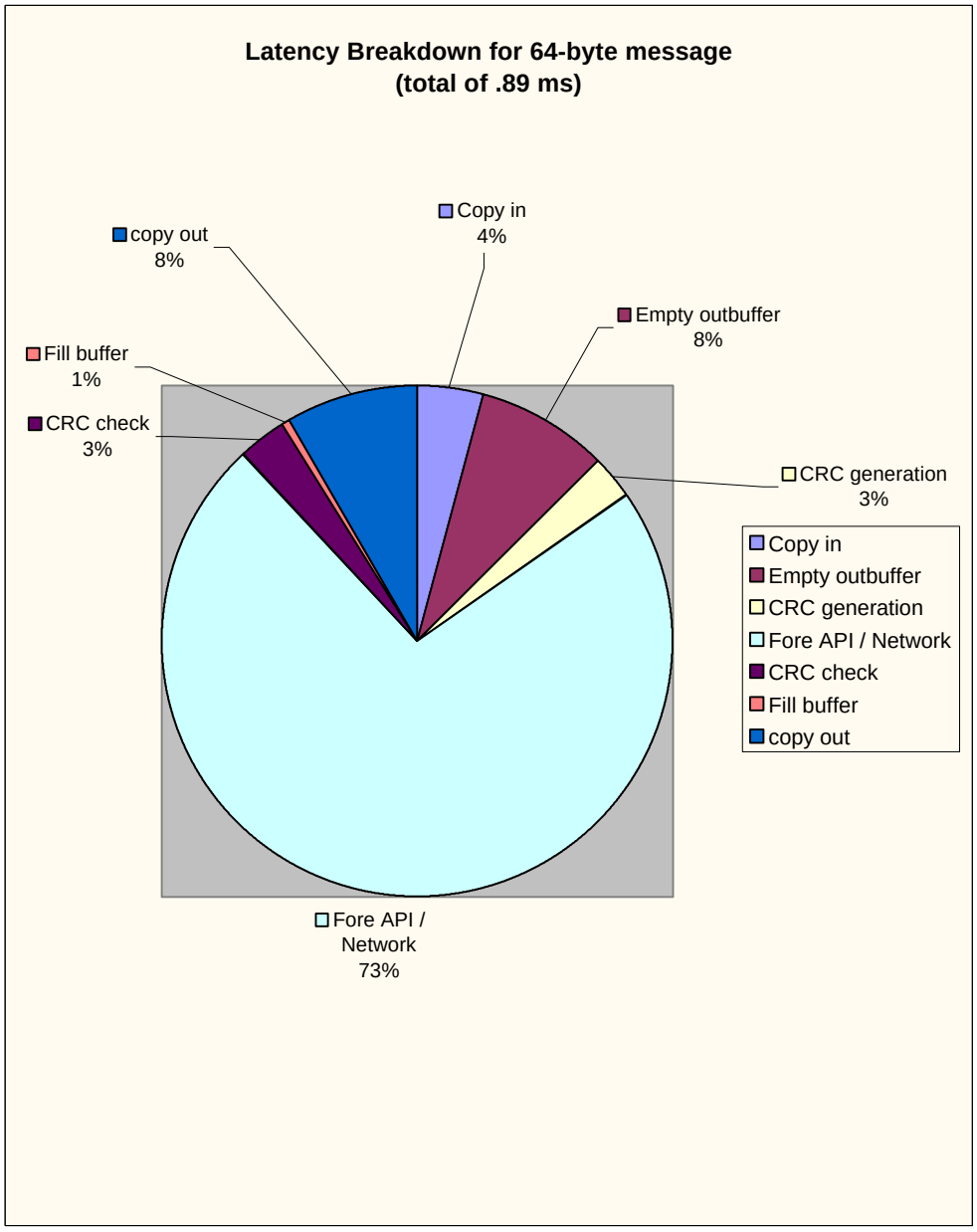


Figure 22: Breakdown of latency for 64-byte message.

Figure 22 shows how a 64-byte message spends its time in transit. Once again, there are three points of note. First, note that even small messages spend a majority of their time in Fore API and network. The work done at the University of Illinois at Urbana-Champaign once again suggests that only 30% of the time should be spent in the network [Karam94, Lauria98]. Integrating the Fore API would enhance performance for small sized messages in a way similar to large messages.

The second point gives further evidence of synchronization loss. The "Empty Output Buffer" function should once again take almost no time, and a 64-byte message should spend a negligible amount of time in the two copy functions. Using techniques

similar to those with the 8-kB message the synchronization loss can be estimated between 12% and 14%.

The final point to note is that the 64-byte message spends about the same proportions of its time in the various places as the 8-kbyte message spends. This similarity indicates that the breakdown latency times scaled with the message size. The significance of this observation is that the processing overhead has not yet become a factor. The processing overhead remains constant regardless of the size of the message, so it should take a larger percentage of time during the transmission of small messages. Comparing Figures 21 and 22 shows that only the checksums took a larger share of time for small sized messages. The scaling of breakdown times implies that performance for small messages has the potential for dramatic improvements. Their time is not yet dominated by overhead processing.

7. Future Integration with SCALE

The SCALE integration goal imposed some requirements on Guaranteed ATM. In addition to being reliable and fast, the ATM interface was to be non-blocking. Non-blocking means that Guaranteed ATM should be able to operate several channels at once. In order to meet this requirement, a structure was created in C so that all the memory needed for a channel could be dynamically allocated for each channel.

The higher layers of SCALE also stipulate that all data passed and received be 64-byte aligned and be multiples of 64 bytes since that improves the memory copying abilities of the main processor. This requirement would have been met whether SCALE was the targeted application or not since the 64-byte alignment also improves the copying ability of Guaranteed ATM. Other than these two stipulations, future integration with SCALE proved to have little impact on the project [George96, Phipps97b].

8. Conclusions and Future Work

The first goal of guaranteed delivery was implemented through error and flow control techniques. The implementation was then verified through an extensive series of torture tests using the half-duplex and full-duplex frameworks. Extreme condition and application anomaly tests identified early software bugs and caught new ones as they developed during performance optimization. Great strides were made towards the second goal of maximized performance, but of course new enhancements are always possible. Performance was optimized through techniques such as the 64-byte memory-to-memory copy, acknowledgement piggybacking, and combined header/data sends. Detailed latency analysis during the project's documentation phase revealed new performance enhancement possibilities for the future. Future integration with SCALE proved to be a minor obstacle, but was planned for nonetheless. All data buffers were 64-byte aligned and Guaranteed ATM was designed to run several channels simultaneously. The remaining paragraphs outline possible future improvements.

Significant performance improvements are achievable in two ways: (1) tightening integration with the layers of code above and below, and (2) reducing synchronization loss. Cornell's work shows that integrating the Fore API has the most promise [U-Net95]. Once that has been achieved, the latency breakdown times indicate that synchronization loss will become significant. After that bottleneck has been relieved, perhaps further performance improvements may be made through tighter integration with the higher layers of SCALE.

Of all of the possibilities, the ones with the most promise lay with modification of the Fore API. After all of the other optimizations, the Fore API emerged as a major latency bottleneck. Modifying the Fore API was not within the scope of this project, but would improve performance dramatically. First of all, the Fore API provides features that Guaranteed ATM does not need, such as Quality of Service guarantees and multicasting. Tighter integration with the Fore API may also be achievable due to the 64-byte stipulation of SCALE. Currently the Fore API will handle any size payload with

any alignment. Perhaps by using the 64-byte atomic memory-to-memory copy, the Fore API could be optimized around 64 bytes. Since the hardware eventually needs the payload to be divided up into sub-64-byte units, this optimization technique may be limited. Finally, tighter integration with the Fore API may be possible through better coordination between the two buffers being employed. Currently, a buffer is used at the Guaranteed ATM layer and at a lower layer.

Once the Fore API has been integrated with Guaranteed ATM, the latency breakdown times indicate that synchronization loss may become a significant bottleneck. Minimizing the time that operations lay idle waiting for access to a variable would enhance performance. Reducing these waits could possibly be accomplished by passing around copies of the variables instead of sharing the same memory location.

Tighter integration with the higher layers of SCALE may also be possible. Perhaps when data is sent, control over that data could be passed instead of just a pointer to it. When Guaranteed ATM receives an acknowledgement, it could then deallocate the memory. This technique would eliminate a buffer copy reducing latency. Only the pointers would need to be passed around. Perhaps on the receive side, a copy could be eliminated through dynamic memory allocation in a similar fashion.

Improvements could probably be made forever. For now, all project goals have been met. After vigorous verification, this layer is worthy of the description “guaranteed delivery.” After thorough optimization, performance within this layer has been maximized. Through advanced planning, integration with SCALE has been eased.

9. References

[Active95] T. von Eicken, Anindya Basu, and Vineet Buch. "Low-Latency Communication Over ATM Networks Using Active Messages." *IEEE Micro*, Feb. 1995.

[U-Net95] T. von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing." *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec 1995.

[Fore93] Fore Systems, Inc., 174 Thorn Hill Road, Warendale, PA 15086-7535. *ForeRunner SBA-200 ATM Sbus Adapter User's Manual*, 1993.

[George96] A.D. George, W. Phipps, R.W. Todd, D. Zirpoli, K. Justice, M. Giacoboni, and M. Sarwar, "SCI and the Scalable Cluster Architecture Latency-hiding Environment (SCALE) Project." *Proceedings of the 6th International Workshop on SCI-based High-Performance Low-Cost Computing*, September 1996.

[Karam94] V. Karamcheti and A. A. Chien. "Software Overhead in Messaging Layers: Where Does the Time Go?" *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[Lauria98] Mario Lauria, Scott Pakin, and Andrew Chien. "Efficient Layering for High Speed Communication: Fast Messages 2.x." University of Illinois at Urbana-Champaign, Urbana, IL, July 1998.

[Phipps97a] W. Phipps. "SCALE Suite API" (Informal Title). Unpublished.

[Phipps97b] W. Phipps. "Channels" (Informal Title). Unpublished.

[SPARC94] SPARC International, Inc., *The SPARC Architecture Manual*, Version 9, Englewood Cliffs, NJ: Prentice Hall, 1994.

[Stallings94] W Stallings, *Data and Computer Communications*, Fourth Edition, Upper Saddle River, NJ: Prentice Hall, 1994.

[Subra] M. Subramanyan, *Protocol Overhead in ATM Networks*, Syracuse University, Technical Report SCCS-678.

10. Appendix: Selected Code

```
/*****************************************************************************************************
*  Header file
*****************************************************************************************************/

#include <stdio.h>                                      /* Include these libraries */
#include <sys/file.h>
#ifdef aix
#include <fcntl.h>
#else /* !aix */
#include <sys/fcntl.h>
#endif /* aix */
#include <fore_atm/fore_atm_user.h>
#include <time.h>
#include <signal.h>
#include "test.h"
#include <synch.h>
#include <thread.h>

#define SERVER_SAP      4096                            /* needed by ATM hardware */
#define TIMEOUT         3                               /* wait for this many seconds before timing out */
#define WINDOW          16                              /* window size */
#define MAX_PACKET_SIZE (9152 + 64)                     /* maximum message size with space for header */
#define BUFFER_SIZE     (WINDOW*MAX_PACKET_SIZE)        /* size of buffers */
#define HEADER_SIZE     11                              /* 0. Frame #, 1. ack, 2-5. length, 11-14. checksum
                                                           data, 15. checksum header */

struct ATM_chan_info {
      long fd;                                  /* file descriptor - used by atm hardware to identify
                                                   channel being used */
      volatile char next_ack_out;               /* Next acknowledgement to be sent */
      volatile char next_nack_out;              /* Next negative acknowledgement to be sent */
      volatile char ack_count;                  /* ack counter – ack is forced when this reaches specified
                                                   value */
      mutex_t send_mutex;                       /* flag that indicates that send stream is being used */
      mutex_t recv_mutex;                       /* flag that indicates that receive stream is being used */
```

```
      char *out_buffer;                         /* points to physical beginning of output buffer */
      char *in_buffer;                          /* points to physical beginning of input buffer */

      volatile long out_frame_length[WINDOW];   /* length of frames in output buffer */

      char cur_in_frame;                        /* next frame to be received */
      volatile long out_buffer_begin;           /* points to beginning of output buffer window */
      volatile long out_buffer_cur;             /* points to data that should be sent next */
      volatile long out_buffer_end;             /* points to end of output buffer window */
      volatile long in_buffer_begin;            /* points to beginning of input buffer window */
      volatile long in_buffer_end;              /* points to end of input buffer window */

      volatile hrtime_t time_sent[WINDOW];      /* time that frames were sent */

}; typedef struct ATM_chan_info ATM_chan_info;

/* #define PRINT_DATA */            /* Debugging option that prints out data at application layer */
/* #define PRINT_RAW_SEND_DATA */   /* Debugging option that prints out raw data just before sending */
/* #define PRINT_RAW_RECV_DATA */   /* Debugging option that prints out raw data just after receiving */
/* #define ADD_ERRORS */            /* Debugging option that simulates network corruptions */
/* #define EAT_PACKETS */           /* Debugging option that simulates lost frames */
/* #define DEBUG_OUTBUFFER */       /* Debugging option that prints output buffer and its variables */
/* #define DEBUG_INBUFFER */        /* Debugging option that prints input buffer and its variables */

#define CONSEC_SENDS    1           /* Number of times each computer sends in a row and receives in a row*/

long PACKETS = 5000;                /* Number of messages to send and/or receive */
long TEST_LENGTH = 32768;           /* Length of messages (greater of next two variables) */
long HOST_SEND = 64;                /* Length of messages that host sends */
long CLIENT_SEND = 64;              /* Length of messages that client sends */

long ACK_FREQ = (WINDOW / 2) - 1;   /* Ack counter – ack is forced when this reaches specified value */

long packets_sent=0;                /* Number of frames sent including resends */
long packets_received=0;            /* Number of frames received including resends */
long progress=1000;                 /* How often to print out progress of test (after this many messages)*/
long errors=0;                      /* Total number of receive errors detected */
```

```
long acks_sent=0;                   /* Total number of acks sent */
long acks_forced = 0;               /* Number of acks forced */
long acks_piggybacked = 0;          /* Number of acks piggybacked */
long acks_received=0;               /* Number of acks received */
long acks_received_forced = 0;      /* Number of forced acks received */
long nacks_sent = 0;                /* Number of nacks sent */
long nacks_received = 0;            /* Number of nacks received */
long frames_ignored_c = 0;          /* Number of frames ignored because of bad checksums */
long frames_ignored_f = 0;          /* Number of frames ignored because of incorrect frames */
long checksums_altered = 0;         /* Number of checksums altered */
long packets_eaten = 0;             /* Number of frames eaten */
long bad_checksums = 0;             /* Number of bad checksums received */
long framing_errors = 0;            /* Number of incorrect frames received */
long timeouts = 0;                  /* Number of timeouts that occurred/

#ifdef ADD_ERRORS
 long error_frequency=1000;         /* Approximately one checksum altered for this many sends */
 long final_error=0;                /* Flag needed for checksum altering */
#endif

#ifdef EAT_PACKETS
 long eat_frequency = 1000;         /* Approximately one frame eaten for this many sends */
 long last_eat = 0;                 /* Flag needed for frame eating */
#endif

long host = 0;                      /* If “1” then host is running, if “0” then client is running */

#define TIME_TESTS      7           /* Number of breakdown times to measure */
char begin = 1;                     /* Flag needed for some breakdown time measurements */
hrtime_t start_time0 = 0;           /* Time at which breakdown time measurement starts */
hrtime_t start_time1 = 0;
hrtime_t start_time2 = 0;
hrtime_t start_time3 = 0;
hrtime_t start_time4 = 0;
hrtime_t start_time5 = 0;
hrtime_t start_time6 = 0;
hrtime_t total_time[TIME_TESTS] = {0,0,0,0,0,0,0};    /* Sum of all measurements for given breakdown time*/
long count_time[TIME_TESTS] = {0,0,0,0,0,0,0};        /* Number of measurements for given breakdown time */
```

```
/*****************************************************************************************
*  Description: “Half-duplex Framework” for testing go back N ARQ control scheme
*  Inputs:  long argc - number of arguments entered at command prompt
*           char *argv[] - pointer to list of argument entered at command prompt
*           ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*                 channel
*  Outputs: Numerous variables printed to screen after test completes
*****************************************************************************************/

long ATM_gobackNARQ_test_code(long argc, char *argv[], ATM_chan_info *ATM_chan_param) {
      long i, j, k, packets = PACKETS;
      long length = TEST_LENGTH;
      char *send_this = (char *) memalign(64, length);
      char *receive_here = (char *) memalign(64, length);
      hrtime_t atm_begin, atm_end;
      double temp;
      long long *send_this_llong = (long long *)send_this;
      long long *receive_here_llong = (long long *)receive_here;

      /* printf("*************************************************************\n");*/

      for (i = 0; i < length / 8; i++) send_this_llong[i]=i;      /* fill buffer */

      if (argc < 2) {                                             /* Host runs this loop */
      /* printf("host\n"); */
            atm_begin = gethrtime();                              /* Throughput time measurement */
            for(i = 0; i < packets / CONSEC_SENDS; i++) {
                  for(k = 0; k < CONSEC_SENDS; k++) {             /* Send this many messages in a row */
                        #ifdef PRINT_DATA
                        printf("Send_this: ");
                        for(j = 0; j < HOST_SEND / 8; j++) printf("%lld ", send_this_llong[j]);
                        printf("\n\n");
                        #endif
                  start_time6 = gethrtime();                /* Total latency time measurment (host) */
                        while(ATM_gobackNARQ_send(ATM_chan_param, send_this, length) < 0) thr_yield();
                  }
                  for(k = 0; k < CONSEC_SENDS; k++) {             /* Receive this many messages in a row */
```

```
                while (ATM_gobackNARQ_recv(ATM_chan_param, receive_here, CLIENT_SEND) < 0)
                        thr_yield();
                #ifdef PRINT_DATA
                printf("Receive_here: ");
                for(j = 0; j < CLIENT_SEND / 8; j++) printf("%lld ", receive_here_llong[j]);
                printf("\n\n");
                #endif
        }
        total_time[6] += gethrtime() - start_time6;
        count_time[6]++;

    }
    atm_end = gethrtime();                              /* Throughput time measurement */
}
else {                                                  /* Client runs this loop */
/* printf("client\n"); */
    for (i = 0; i < length / 8; i++) receive_here_llong[i]=0;   /* clear buffer */
    atm_begin = gethrtime();                            /* Throughput time measurment */
    for(i = 0; i < packets / CONSEC_SENDS; i++) {


        for(k = 0; k < CONSEC_SENDS; k++) {             /* Receive this many messages in a row */
            while (ATM_gobackNARQ_recv(ATM_chan_param, receive_here, HOST_SEND) < 0)
                thr_yield();
        if(begin != 1) {total_time[6] += gethrtime() - start_time6;
        count_time[6]++;}                          /* Total latency time measurement (client) */
            #ifdef PRINT_DATA
            printf("Receive_here: ");
            for(j = 0; j < HOST_SEND / 8; j++) printf("%lld ", receive_here_llong[j]);
            printf("\n\n");
            #endif
        }
        for(k = 0; k < CONSEC_SENDS; k++) {             /* Send this many messages in a row */
            #ifdef PRINT_DATA
            printf("Send_this: ");
            for(j = 0; j < CLIENT_SEND / 8; j++) printf("%lld ", send_this_llong[j]);
            printf("\n\n");
            #endif
```

```
            start_time6 = gethrtime();
            begin = 0;
                  while (ATM_gobackNARQ_send(ATM_chan_param, send_this, CLIENT_SEND) < 0)
                        thr_yield();
            }


      }
      atm_end = gethrtime();                              /* Throughput time measurement */
}

/* printf(" Send_length = %ld, Send_num = %ld\n", HOST_SEND, PACKETS);  /* Print variables to screen
                                                                        after test */
printf(" Recv_length = %ld, Recv_num = %ld\n", CLIENT_SEND, PACKETS);
printf(" Packets_sent = %ld\n", packets_sent - acks_forced);
printf(" Packets_received = %ld\n", packets_received - acks_received_forced);
printf("\n");
printf(" Acks_sent = %ld\n", acks_sent);
printf(" Acks_received = %ld\n", acks_received);
printf(" Acks_received_forced = %ld\n", acks_received_forced);
printf(" Acks_forced = %ld\n", acks_forced);
printf(" Acks_piggybacked = %ld\n", acks_piggybacked);
printf(" Nacks_sent = %ld\n", nacks_sent);
printf(" Nacks_received = %ld\n", nacks_received);
printf("\n");
printf(" Packets_eaten = %ld\n", packets_eaten);
printf(" Checksums_altered = %ld\n", checksums_altered);
printf("\n");
printf(" Receive errors = %ld\n", errors);
printf(" Bad_checksums = %ld\n", bad_checksums);
printf(" Framing_errors = %ld\n", framing_errors);
printf(" Frames_ignored_c = %ld\n", frames_ignored_c);
printf(" Frames_ignored_f = %ld\n", frames_ignored_f);
printf("\n");
printf(" Timeouts = %ld\n", timeouts);
printf(" Ack_timeouts = %ld\n", ack_timeouts);
printf(" Nack_timeouts = %ld\n", nack_timeouts);
printf("\n");
```

```
        printf("Time: %f secs\n", temp = (double)(atm_end - atm_begin) / 1000000000.);
        if (atm_end - atm_begin != 0) printf("Throughput: %f Megabits/sec\n", temp = ((double)HOST_SEND *
              (double)PACKETS * 8. / (1000000. * temp)) + ((double)CLIENT_SEND * (double)PACKETS * 8. /
              (1000000. * temp)));

        temp = (double)(atm_end - atm_begin) / 1000000000.;
        if (host == 1) printf("%f\n", temp = ((double)HOST_SEND * (double)PACKETS * 8. / (1000000. * temp))
               ((double)CLIENT_SEND * (double)PACKETS * 8. / (1000000. * temp)));
*/
}
```

```
/*****************************************************************************************
*  "Full Duplex Framework" – used for full duplex testing
*  This code used for host.  Nearly identical code used for client
*****************************************************************************************/

/* Global variables for testing and constants */

/* #define PRINT_CELLS */          /* Option to print messages or not */
#define CHECK_ERRORS               /* Option to verify whether or not correct data is received */

#define DELAY          0           /* Maximum delay time to insert between sends */

long SEND_LENGTH = 32768;          /* Length to send */
long RECV_LENGTH = 32768;          /* Length to receive */
long SEND_NUM = 1000;
long RECV_NUM = 1000;

long ind_var[10] = {32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64}; /* run test multiple times
                                                                                 with these values (length)*/
long ind_var_two[3] = {300, 1000, 5000};  /* run test multiple times with these values (numner of sends)*/

#define TIME_TESTS_host 1                /* Number of latency breakdown measurements to make */
hrtime_t start_time_host = 0;            /* Time at which breakdown measurment started */
hrtime_t total_time_host[TIME_TESTS] = {0}; /* Total time spent in breakdown piece of code */
long count_time_host[TIME_TESTS] = {0};   /* Number of measurements made */


void *send_test_data_host(void *temp);    /* List of functions and threads*/
void *recv_test_data_host(void *temp);

/*****************************************************************************/

main(long argc, char *argv[]) {
      ATM_chan_info *ATM_chan_param = (ATM_chan_info *) malloc(sizeof(ATM_chan_info));
      char *send_this_main = (char *) memalign(64, 256);
      char *recv_here_main = (char *) memalign(64, 256);
      thread_t thread1;
      thread_t thread2;
```

```
	long i;
	double temp;
	hrtime_t atm_begin, atm_end;
	long loop, in_loop, out_loop, test_loop;

	ATM_efc_host_connect(ATM_chan_param);
	for(test_loop = 0; test_loop < 1; test_loop++) {
		for(out_loop = 0; out_loop < 10; out_loop++) {
			SEND_LENGTH = ind_var[out_loop];
			RECV_LENGTH = ind_var[out_loop];
			for(loop = 0; loop < 3; loop++)  {
				SEND_NUM = ind_var_two[loop];
				RECV_NUM = ind_var_two[loop];
				printf("\n Send_length = %ld, Send_num = %ld\n", SEND_LENGTH, SEND_NUM);
				for(in_loop = 0; in_loop < 10; in_loop++) {

		while (ATM_gobackNARQ_recv(ATM_chan_param, recv_here_main, 64) < 0) thr_yield(); /*
                                                                        Synchronization Receive */
		if(recv_here_main[19] != 75) {printf("ERRRRRRRRRRROR\n"); exit(0);}
		send_this_main[22] = 98;
		while (ATM_gobackNARQ_send(ATM_chan_param, send_this_main, 64) < 0) thr_yield(); /*
                                                                        Synchronization Send */

atm_begin = gethrtime();                                /* Throughput testing time measurement */
	thr_create(NULL, 0, send_test_data_host, (void *)ATM_chan_param, 0, &thread1); /* Spawn send and
                                                                                  receive threads */
	thr_create(NULL, 0, recv_test_data_host, (void *)ATM_chan_param, 0, &thread2);
	thr_join(thread2, NULL, NULL);                      /* Wait for threads to finish and join */
	thr_join(thread1, NULL, NULL);

		while (ATM_gobackNARQ_recv(ATM_chan_param, recv_here_main, 64) < 0) thr_yield(); /*
                                                                        Synchronization receive */
		if(recv_here_main[19] != 75) {printf("ERRRRRRRRRRROR\n"); exit(0);}
		send_this_main[22] = 98;
		while (ATM_gobackNARQ_send(ATM_chan_param, send_this_main, 64) < 0) thr_yield(); /*
                                                                        Synchronization Send */
```

```
atm_end = gethrtime();                          /* Throughput time measurment */

/*    printf("done\n\n");                       /* Print out variables of interest */
      for (i = 0; i < TIME_TESTS_host; i++) {
            printf("avg_time[%d] = %llf usecs\n", i, (double)(total_time_host[i]) /
                  ((double)count_time_host[i] * 1000.));
      }

      printf(" Send_length = %ld, Send_num = %ld\n", SEND_LENGTH, SEND_NUM);
      printf(" Recv_length = %ld, Recv_num = %ld\n", RECV_LENGTH, RECV_NUM);
      printf(" Packets_sent = %ld\n", packets_sent - acks_forced);
      printf(" Packets_received = %ld\n", packets_received - acks_received_forced);
      printf("\n");
      printf(" Acks_sent = %ld\n", acks_sent);
      printf(" Acks_received = %ld\n", acks_received);
      printf(" Acks_received_forced = %ld\n", acks_received_forced);
      printf(" Acks_forced = %ld\n", acks_forced);
      printf(" Acks_piggybacked = %ld\n", acks_piggybacked);
      printf(" Nacks_sent = %ld\n", nacks_sent);
      printf(" Nacks_received = %ld\n", nacks_received);
      printf("\n");
      printf(" Packets_eaten = %ld\n", packets_eaten);
      printf(" Checksums_altered = %ld\n", checksums_altered);
      printf("\n");

      printf(" Receive errors = %ld\n", errors);
      printf(" Bad_checksums = %ld\n", bad_checksums);
      printf(" Framing_errors = %ld\n", framing_errors);
      printf(" Frames_ignored_c = %ld\n", frames_ignored_c);
      printf(" Frames_ignored_f = %ld\n", frames_ignored_f);
      printf("\n");

      printf(" Timeouts = %ld\n", timeouts);
      printf(" Ack_timeouts = %ld\n", ack_timeouts);
      printf(" Nack_timeouts = %ld\n", nack_timeouts);
      printf("\n");

      printf("Time: %f secs\n", temp = (double)(atm_end - atm_begin) / 1000000000.);
```

```
      if (atm_end - atm_begin != 0) printf("Throughput: %f Megabits/sec\n", temp = ((double)SEND_NUM *
            (double)SEND_LENGTH * 8. / (1000000. * temp)) + ((double)RECV_NUM * (double)RECV_LENGTH * 8. /
            (1000000. * temp)));
*/
      temp = (double)(atm_end - atm_begin) / 1000000000.;
      printf("%f\n", temp = ((double)SEND_NUM * (double)SEND_LENGTH * 8. / (1000000. * temp)) +
            ((double)RECV_NUM * (double)RECV_LENGTH * 8. / (1000000. * temp)));
}}}}}


/*****************************************************************************/

void *send_test_data_host(void *temp) {
      ATM_chan_info *ATM_chan_param = (ATM_chan_info *)temp;
      char *send_this = (char *) memalign(64, SEND_LENGTH);
      long long *send_this_llong = (long long *)send_this;
      long i, j;
      long long count = 0;
      hrtime_t last_time, wait_time;

      for(i = 0; i < SEND_NUM; i++) {

            /* SEND_LENGTH = ((random() % 128) + 1) * 64;         /* Random test length option */
            printf("SEND_LENGTH = %d\n", SEND_LENGTH);
            */
            #ifdef CHECK_ERRORS           /* If checking for errors fill data with counting numbers */
                  for (j = 0; j < SEND_LENGTH / 8; j++) send_this_llong[j] = count++;
            #endif

            #ifdef PRINT_CELLS                                    /* Print messages if desired */
                  printf("Sending\n"); for (j = 0; j < SEND_LENGTH / 8; j++) printf("%lld ",
                        send_this_llong[j]); printf("\n");
            #endif

            while (ATM_gobackNARQ_send(ATM_chan_param, send_this, SEND_LENGTH) < 0) thr_yield(); /* Send */
            last_time = gethrtime();                              /* Delay before next send */
            wait_time = (random() % 100) * DELAY;
            do thr_yield();while(gethrtime() - last_time < wait_time);
```

```
	}

	#ifdef PRINT_CELLS
		printf("    \n\n\n*** send done ***\n\n\n");
	#endif
}


/*****************************************************************************/

void *recv_test_data_host(void *temp) {
	ATM_chan_info *ATM_chan_param = (ATM_chan_info *)temp;
	char *recv_here = (char *) memalign(64, RECV_LENGTH);
	long long *recv_here_llong = (long long *)recv_here;
	long i, j;
	long long count = 0;
	hrtime_t last_time, wait_time;

	for(i = 0; i < RECV_NUM; i++) {
		/* RECV_LENGTH = ((random() % 128) + 1) * 64;   /* Random sized receives if desired */
		printf("RECV_LENGTH = %d\n", RECV_LENGTH);
		*/
		while (ATM_gobackNARQ_recv(ATM_chan_param, recv_here, RECV_LENGTH) < 0) thr_yield(); /* Receive
                                                                                                */

		#ifdef PRINT_CELLS                              /* Print messages if desired */
			printf("Receiving\n"); for(j = 0; j < RECV_LENGTH / 8; j++) printf("%lld ",
				recv_here_llong[j]); printf("\n");
		#endif

		#ifdef CHECK_ERRORS                             /* Check for errors if desired */
			for(j = 0; j < RECV_LENGTH / 8; j++) {if (recv_here_llong[j] != count++) {
			printf("****** Bad data received: %lld\n", recv_here_llong[j]);}}
		#endif

		last_time = gethrtime();                        /* Random delay before next receive */
```

```
            wait_time = (random() % 10) * DELAY;
            do thr_yield(); while(gethrtime() - last_time < wait_time);

      }
      #ifdef PRINT_CELLS
            printf("    \n\n\n*** recv done ***\n\n\n");
      #endif
}


/*****************************************************************************/
```

```
/*****************************************************************************************
*  Description:   "Copy In" copies data into the output buffer
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*           char *send_this - points to data that is to be copied into the buffer
*  Outputs: return(-1) - Output buffer is full, no action taken
*           return(0) - Sucessful copy
*****************************************************************************************/

long ATM_gobackNARQ_send(ATM_chan_info *ATM_chan_info_in, char *send_this, long length) {
      long i, j;
      long temp_buffer_end;
      long cur_buffer_length;
      long temp_length, temp_length2, copied;
      long long *out_buffer_llong = (long long *)ATM_chan_info_in->out_buffer;

start_time0 = gethrtime();
      mutex_lock(&ATM_chan_info_in->send_mutex);            /* lock send stream */

      temp_length = MAX_PACKET_SIZE;                        /* Find size rounded up to next packet size */
      temp_length2 = MAX_PACKET_SIZE - 64;
      while (length > temp_length2) {temp_length += MAX_PACKET_SIZE; temp_length2 += MAX_PACKET_SIZE - 64;}

      if (ATM_chan_info_in->out_buffer_end >= ATM_chan_info_in->out_buffer_begin)
            cur_buffer_length = ATM_chan_info_in->out_buffer_end - ATM_chan_info_in->out_buffer_begin;
            /* See if room in buffer */
      else cur_buffer_length = BUFFER_SIZE - ATM_chan_info_in->out_buffer_begin + ATM_chan_info_in-
            >out_buffer_end;
      if (BUFFER_SIZE - cur_buffer_length < temp_length + MAX_PACKET_SIZE) { /* printf("*** Output Buffer
            Full ***\n"); printf("errors = %d\n", errors); */     mutex_unlock(&ATM_chan_info_in-
            >send_mutex); thr_yield(); return(-1);}

      copied = 0;
      while (length - copied > 0) {                         /* copy message into buffer, segment into
                                                            multiple frames if needed */
            if(length - copied > MAX_PACKET_SIZE) {
                  for (i = 0; i < MAX_PACKET_SIZE ; i+=64)
```

```
                        move64m_m(&ATM_chan_info_in->out_buffer[ATM_chan_info_in->out_buffer_end + i + 64],
                              &send_this[copied + i]);      /* copy into buffer */

                  ATM_chan_info_in->out_frame_length[ATM_chan_info_in->out_buffer_end / MAX_PACKET_SIZE] =
                        MAX_PACKET_SIZE - 64;               /* Save length of packet */
                  copied += MAX_PACKET_SIZE - 64;
            }
            else  {
                  for (i = 0; i < length - copied; i+=64) move64m_m(&ATM_chan_info_in-
                        >out_buffer[ATM_chan_info_in->out_buffer_end + i + 64], &send_this[copied + i]);
                                                                  /* copy into buffer */

                  ATM_chan_info_in->out_frame_length[ATM_chan_info_in->out_buffer_end / MAX_PACKET_SIZE] =
                        length - copied;              /* Save length of packet */
                  copied += length - copied;
            }
            ATM_chan_info_in->out_buffer_end = ATM_chan_info_in->out_buffer_end + MAX_PACKET_SIZE;
            if(ATM_chan_info_in->out_buffer_end >= BUFFER_SIZE) ATM_chan_info_in->out_buffer_end -=
                  BUFFER_SIZE;
      }

#ifdef DEBUG_OUTBUFFER
      printf("Out_buffer_begin = %ld, out_buffer_cur = %ld, out_buffer_end = %ld\n",
            ATM_chan_info_in->out_buffer_begin,
            ATM_chan_info_in->out_buffer_cur,
            ATM_chan_info_in->out_buffer_end);
      printf("Just after copy into buffer: Out_buffer = ");
      for (i = 0; i < BUFFER_SIZE / 8; i++) printf("%lld ", out_buffer_llong[i]);
      printf("\n");
#endif

      mutex_unlock(&ATM_chan_info_in->send_mutex);                        /* lock receive stream */
total_time[0] += gethrtime() - start_time0;
count_time[0]++;
      if (ATM_GBN_empty_outbuffer(ATM_chan_info_in) < 0 ) return(-1);
      thr_yield();
      return(0);
}
```

```
/*****************************************************************************************************
*  Description: "Empty Output Buffer" - Sends all unsent data
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*  Outputs: return 0 - OK
*           return -1 - Network error
*****************************************************************************************************/

long ATM_GBN_empty_outbuffer(ATM_chan_info *ATM_chan_info_in) {
      hrtime_t temp_time;
      long temp_frame_cur;
      long temp_frame_begin;
      char *junk;
      char out_header[HEADER_SIZE];

start_time1 = gethrtime();
      mutex_lock(&ATM_chan_info_in->send_mutex);                        /* lock send and receive streams */
      mutex_lock(&ATM_chan_info_in->recv_mutex);

      while (ATM_chan_info_in->out_buffer_cur != ATM_chan_info_in->out_buffer_end) {
                                 /* Send all packets waiting in output buffer */
            temp_frame_cur = ATM_chan_info_in->out_buffer_cur / MAX_PACKET_SIZE;
            temp_frame_begin = ATM_chan_info_in->out_buffer_begin / MAX_PACKET_SIZE;

            temp_time = gethrtime();      /* Save send time of next outputted packet */
            ATM_chan_info_in->time_sent[temp_frame_cur] = temp_time;

            ATM_chan_info_in->out_buffer[ATM_chan_info_in->out_buffer_cur + 64 - HEADER_SIZE] =
                  temp_frame_cur;                                    /* Mark packet with frame number */
            ATM_chan_info_in->out_buffer[ATM_chan_info_in->out_buffer_cur + 64 - HEADER_SIZE + 1] =
                  ATM_chan_info_in->next_ack_out;                    /* Piggyback ack onto outgoing packet */
            if(ATM_chan_info_in->next_ack_out != 127) {acks_sent++; acks_piggybacked++; ATM_chan_info_in-
                  >ack_sent_time = gethrtime(); ATM_chan_info_in->ack_count = 0;}
            ATM_chan_info_in->next_ack_out = 127;
            /* Clear ack to indicate that ack has been sent */
            ATM_chan_info_in->out_buffer[ATM_chan_info_in->out_buffer_cur + 64 - HEADER_SIZE + 2] =
                  ATM_chan_info_in->out_frame_length[ATM_chan_info_in->out_buffer_cur / MAX_PACKET_SIZE] >>
                  24;   /* Send length of packet in header */
```

```
            ATM_chan_info_in->out_buffer[ATM_chan_info_in->out_buffer_cur + 64 - HEADER_SIZE + 3] =
                  ATM_chan_info_in->out_frame_length[ATM_chan_info_in->out_buffer_cur / MAX_PACKET_SIZE] >>
                  16;
            ATM_chan_info_in->out_buffer[ATM_chan_info_in->out_buffer_cur + 64 - HEADER_SIZE + 4] =
                  ATM_chan_info_in->out_frame_length[ATM_chan_info_in->out_buffer_cur / MAX_PACKET_SIZE] >>
                  8;
            ATM_chan_info_in->out_buffer[ATM_chan_info_in->out_buffer_cur + 64 - HEADER_SIZE + 5] =
                  ATM_chan_info_in->out_frame_length[ATM_chan_info_in->out_buffer_cur / MAX_PACKET_SIZE];

total_time[1] += gethrtime() - start_time1;
count_time[1]++;
            if (ATM_send(ATM_chan_info_in->fd, ATM_chan_info_in->out_buffer + ATM_chan_info_in-
                  >out_buffer_cur + 64 - HEADER_SIZE, ATM_chan_info_in->out_frame_length[ATM_chan_info_in-
                  >out_buffer_cur / MAX_PACKET_SIZE]) < 0) {
                        mutex_unlock(&ATM_chan_info_in->recv_mutex); mutex_unlock(&ATM_chan_info_in-
                              >send_mutex); return(-1);}          /* Send Packet */

            ATM_chan_info_in->out_buffer_cur += MAX_PACKET_SIZE;  /* Prepare to send next packet - loop
                                                                     around buffer if needed */
            if (ATM_chan_info_in->out_buffer_cur >= BUFFER_SIZE) ATM_chan_info_in->out_buffer_cur = 0;
      }
      mutex_unlock(&ATM_chan_info_in->recv_mutex);                /*unlock send and receive streams */
      mutex_unlock(&ATM_chan_info_in->send_mutex);
      return(0);
}
```

```
/*****************************************************************************************************
*  Description: "Send Frame" - Generates checksums and sends frame
*  Inputs:  long fd - file descriptor that indicates which channel to use
*           char *buffer - points to data that should be sent
*           long length - amount of data to send
*  Outputs: return 0 - OK
*           return -1 - error
*****************************************************************************************************/

char ATM_send(long fd, char *buffer, long length) {
      register i;
      unsigned long long register sum = 0;
      char sum_char = 0;
      long frac_length = length / 8;
      long long *buffer_llong = (long long *)(buffer + HEADER_SIZE);
      long long j;

start_time2 = gethrtime();
      for (i = 0; i < 6 ; i++) sum_char += buffer[i];                    /* Generate header checksum */
      for (i = 0; i < frac_length; i++) sum += buffer_llong[i];          /* Generate payload checksum */
      sum = (long)sum + (long)(sum >> 32);

#ifdef ADD_ERRORS
        if (!(random() % error_frequency) && (packets_sent - acks_sent) > final_error) {
      /* add errors for testing */
            sum++;
            final_error = packets_sent - acks_sent;
            checksums_altered++;
      /*    printf("\nBad packet inserted\n"); */
        }
#endif
      buffer[6] = sum_char;
      buffer[7] = sum >> 24;                                  /* add checksums to end of buffer */
      buffer[8] = sum >> 16;
      buffer[9] = sum >> 8;
      buffer[10] = sum;

#ifdef PRINT_RAW_SEND_DATA
```

```
      printf("Sending...\n");
for (i = 0; i < HEADER_SIZE; i++) printf("%d * ", (unsigned char)buffer[i]);        /* print out data being
                                                                                    sent with checksum */
for (i = 0; i < length / 8; i++) printf("%lld ", (unsigned long long)buffer_llong[i]);          /* print
                                                                                    out data being sent
                                                                                    with checksum */
      printf("\nChecksum = %ld\n", (long)sum);
#endif

        packets_sent++;

#ifdef EAT_PACKETS
        if ((packets_sent - acks_sent) % eat_frequency == random() % eat_frequency && (packets_sent -
            acks_sent) > last_eat) {                        /* add errors for testing */
            last_eat = packets_sent - acks_sent;
            packets_eaten++;
            /* printf("Sent\n");
            printf("Packet eaten\n"); */
            return(0);
        }
#endif
total_time[2] += gethrtime() - start_time2;
count_time[2]++;

        if (atm_send(fd, buffer, length + HEADER_SIZE) < 0) {     /* Send frame using Fore API */
             atm_error("atm_send");
             return(-1);
        }

#ifdef PRINT_RAW_SEND_DATA
      printf("Sent\n\n");
#endif

      return(0);
}
```

```
/*****************************************************************************************************
*  Description: Sends an ack - If ack is pending, creates header and send ack
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*  Outputs: none
*****************************************************************************************************/
long ATM_send_ack(ATM_chan_info *ATM_chan_info_in) {
      char out_header[HEADER_SIZE];

      out_header[0] = 0;
      out_header[1] = ATM_chan_info_in->next_ack_out;
      out_header[2] = 0;
      out_header[3] = 0;
      out_header[4] = 0;
      out_header[5] = 0;
      ATM_send(ATM_chan_info_in->fd, out_header, 0);
      acks_sent++;
      ATM_chan_info_in->ack_sent_time = gethrtime();
      ATM_chan_info_in->next_ack_out = 127;
      ATM_chan_info_in->ack_count = 0;
      acks_forced++;
}

/*****************************************************************************************************
*  Description: Creates header and sends nack
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*  Outputs: none
*****************************************************************************************************/

long ATM_send_nack(ATM_chan_info *ATM_chan_info_in) {
      char out_header[HEADER_SIZE];


      ATM_chan_info_in->next_ack_out = 127;

      out_header[0] = 0;
      out_header[1] = ATM_chan_info_in->next_nack_out;
```

```
        out_header[2] = 0;
        out_header[3] = 0;
        out_header[4] = 0;
        out_header[5] = 0;

        ATM_send(ATM_chan_info_in->fd, out_header, 0);
        ATM_chan_info_in->nack_sent_time = gethrtime();
        nacks_sent++;

}
```

```
/*****************************************************************************************
*  Description:   "Copy Out" - Copies data out of input buffer for application
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*           char *receive_here - points to location where data should be copied
*           long length - amount of data to copy
*  Outputs: return(-1) - not enough data available
*           return(0) - Successful copy
*****************************************************************************************/

long ATM_gobackNARQ_recv(ATM_chan_info *ATM_chan_info_in, char *receive_here, long length) {
      long i;
      long cur_buffer_length;
      long long *in_buffer_llong = (long long *)ATM_chan_info_in->in_buffer;
      long long *receive_here_llong = (long long *)receive_here;

start_time3 = gethrtime();
      mutex_lock(&ATM_chan_info_in->recv_mutex);

      if (ATM_chan_info_in->in_buffer_end >= ATM_chan_info_in->in_buffer_begin)
            cur_buffer_length = ATM_chan_info_in->in_buffer_end - ATM_chan_info_in->in_buffer_begin;
                                                                  /* See if room in buffer */
      else cur_buffer_length = BUFFER_SIZE - ATM_chan_info_in->in_buffer_begin + ATM_chan_info_in-
            >in_buffer_end;
      if(cur_buffer_length < length) {
            mutex_unlock(&ATM_chan_info_in->recv_mutex);
            thr_yield();
            return(-1);
      }

#ifdef DEBUG_INBUFFER
            printf("In_buffer_begin = %ld, in_buffer_end = %ld\n",
                  ATM_chan_info_in->in_buffer_begin,
                  ATM_chan_info_in->in_buffer_end);
      printf("\nJust before buffer copy: In_buffer = ");
      for (i = -32; i < length / 8 ; i++)
            printf("%lld ",in_buffer_llong[ATM_chan_info_in->in_buffer_begin / 8 + i]);
      printf("\n\n");
```

```
#endif

      for (i = 0; i < length ; i+=64) {                                    /* copy out of buffer */
            if (ATM_chan_info_in->in_buffer_begin + i < BUFFER_SIZE)
                  move64m_m(&receive_here[i], &ATM_chan_info_in->in_buffer[ATM_chan_info_in-
                        >in_buffer_begin + i]);
            else  move64m_m(&receive_here[i], &ATM_chan_info_in->in_buffer[ATM_chan_info_in-
                  >in_buffer_begin + i - BUFFER_SIZE]);
      }

      ATM_chan_info_in->in_buffer_begin += length;
      if (ATM_chan_info_in->in_buffer_begin >= BUFFER_SIZE) ATM_chan_info_in->in_buffer_begin -=
            BUFFER_SIZE;

#ifdef DEBUG_INBUFFER
            printf("In_buffer_begin = %ld, in_buffer_end = %ld, length = %ld\n",
                  ATM_chan_info_in->in_buffer_begin,
                  ATM_chan_info_in->in_buffer_end,
                  length);
      printf("\nJust after buffer copy: In_buffer = ");
      for (i = 0; i < length / 8; i++) printf("%lld ",receive_here_llong[i]);
      printf("\n\n");
#endif

      mutex_unlock(&ATM_chan_info_in->recv_mutex);
total_time[3] += gethrtime() - start_time3;
count_time[3]++;
      thr_yield();
      return(0);
}
```

```
/*****************************************************************************************************
*  Description: Thread that continuously receives data by calling “Fill Input Buffer” function
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*  Outputs: none
*****************************************************************************************************/
void *ATM_recv_thr(void *temp) {
      ATM_chan_info *ATM_chan_info_in = (ATM_chan_info *)temp;
      long status;

      while(1) {
            ATM_GBN_fill_buffer(ATM_chan_info_in);
            thr_yield();
      }
}

/*****************************************************************************************************
*  Description: "Fill Input Buffer" - called continuously by thread to receive data
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*  Outputs: return(0) - OK
*           return(-1) - Error
*****************************************************************************************************/

long ATM_GBN_fill_buffer(ATM_chan_info *ATM_chan_info_in) {
      char status;
      long length_in;
      char in_header[HEADER_SIZE];
      long cur_buffer_length, i;

      if ((status = ATM_recv(ATM_chan_info_in->fd, in_header, ATM_chan_info_in->in_buffer +
            ATM_chan_info_in->in_buffer_end, &length_in, ATM_chan_info_in)) < 0) return(-1);    /* Receive
            data */

start_time4 = gethrtime();
      mutex_lock(&ATM_chan_info_in->send_mutex);            /* Lock send and receive streams */
      mutex_lock(&ATM_chan_info_in->recv_mutex);
```

```
if (status == 1) {                                    /* Verify checksum status */
      if(ATM_chan_info_in->next_nack_out != 127) {
            /* printf("Previous Frame Ignored (bad checksum)\n\n"); */
            frames_ignored_c++;
            mutex_unlock(&ATM_chan_info_in->recv_mutex);
            mutex_unlock(&ATM_chan_info_in->send_mutex);
            total_time[4] += gethrtime() - start_time4;
            count_time[4]++;
            return(0);
      }
      if(length_in != 0 && ATM_chan_info_in->next_nack_out == 127) {
            ATM_chan_info_in->next_nack_out = ATM_chan_info_in->cur_in_frame - 128;
            /* printf("Bad Checksum Received\n");  */
            bad_checksums++;
            ATM_send_nack(ATM_chan_info_in);               /* if bad checksum, send nack */
            errors++;
      }
      mutex_unlock(&ATM_chan_info_in->recv_mutex);
      mutex_unlock(&ATM_chan_info_in->send_mutex);

      total_time[4] += gethrtime() - start_time4;
      count_time[4]++;
      return(1);
}

      if (status == 0 && length_in != 0 && ATM_chan_info_in->cur_in_frame != in_header[0] ) {
                                          /* If frame received not next frame, send nack */
            if(ATM_chan_info_in->next_nack_out == 127) {
                  /* printf("*** Framing error ***\n");
                  printf("Temp_frame_in = %d, In_frame_label = %d\n", ATM_chan_info_in->cur_in_frame,
                  in_header[0]); */
                  framing_errors++;
                  ATM_chan_info_in->next_nack_out = ATM_chan_info_in->cur_in_frame - 128;
                  ATM_send_nack(ATM_chan_info_in);
                  errors++;
            }
            else { frames_ignored_f++; /* printf("Previous Frame Ignored (framing error)\n\n"); */}
      }
```

```
if (ATM_chan_info_in->in_buffer_end >= ATM_chan_info_in->in_buffer_begin) cur_buffer_length =
      ATM_chan_info_in->in_buffer_end - ATM_chan_info_in->in_buffer_begin;          /* find
      buffer length */
else cur_buffer_length = BUFFER_SIZE - ATM_chan_info_in->in_buffer_begin + ATM_chan_info_in-
      >in_buffer_end;

if (status == 0 && length_in != 0 && ATM_chan_info_in->cur_in_frame == in_header[0] &&
      BUFFER_SIZE - cur_buffer_length >= 2 * MAX_PACKET_SIZE) {   /* Prepare for next frame */
      ATM_chan_info_in->next_ack_out = ATM_chan_info_in->cur_in_frame;
      ATM_chan_info_in->ack_count++;
      if (ATM_chan_info_in->next_nack_out != 127) /* printf("Nack cleared\n"); */
      ATM_chan_info_in->next_nack_out = 127;
      if(ATM_chan_info_in->in_buffer_end + length_in > BUFFER_SIZE) {
            for (i = 0; i < ATM_chan_info_in->in_buffer_end + length_in - BUFFER_SIZE; i+=64)
                  move64m_m(&ATM_chan_info_in->in_buffer[i], &ATM_chan_info_in-
                        >in_buffer[BUFFER_SIZE + i]);
            }
      ATM_chan_info_in->in_buffer_end += length_in;
      if (ATM_chan_info_in->in_buffer_end >= BUFFER_SIZE) ATM_chan_info_in->in_buffer_end -=
            BUFFER_SIZE;
      ATM_chan_info_in->cur_in_frame++;
      if(ATM_chan_info_in->cur_in_frame >= WINDOW) ATM_chan_info_in->cur_in_frame = 0;
      if(ATM_chan_info_in->ack_count >= ACK_FREQ) ATM_send_ack(ATM_chan_info_in);
}
if (status == 0 && in_header[1] < 0) {    /* if nack received, set current output frame to
                                             frame indicated by nack */

      /* printf("Received NACK, frame = %d\n\n", in_header[1] + 128); */
      nacks_received++;
      ATM_chan_info_in->out_buffer_cur = (in_header[1] + 128) * MAX_PACKET_SIZE;
      ATM_chan_info_in->out_buffer_begin = (in_header[1] + 128) * MAX_PACKET_SIZE;

      mutex_unlock(&ATM_chan_info_in->recv_mutex);
      mutex_unlock(&ATM_chan_info_in->send_mutex);
      ATM_GBN_empty_outbuffer(ATM_chan_info_in);
      /* printf("Received Nack -  out buffer emptied\n"); */
      total_time[4] += gethrtime() - start_time4;
```

```
            count_time[4]++;
            return(0);
        }

        if (status == 0 && in_header[1] >= 0 && in_header[1] != 127) {    /* if ack received, mark
                                                                            out_buffer_begin */
            acks_received++;
            if(length_in == 0) acks_received_forced++;
            ATM_chan_info_in->out_buffer_begin = in_header[1] * MAX_PACKET_SIZE + MAX_PACKET_SIZE;
            if (ATM_chan_info_in->out_buffer_begin >= BUFFER_SIZE) ATM_chan_info_in->out_buffer_begin
                  -= BUFFER_SIZE;
            mutex_unlock(&ATM_chan_info_in->recv_mutex);
            mutex_unlock(&ATM_chan_info_in->send_mutex);
            total_time[4] += gethrtime() - start_time4;
            count_time[4]++;
            return(0);
        }

        if(status == 0 && in_header[1] == 127) {

            mutex_unlock(&ATM_chan_info_in->recv_mutex);
            mutex_unlock(&ATM_chan_info_in->send_mutex);
            total_time[4] += gethrtime() - start_time4;
            count_time[4]++;
            return(0);
        }

        mutex_unlock(&ATM_chan_info_in->recv_mutex);
        mutex_unlock(&ATM_chan_info_in->send_mutex);
        total_time[4] += gethrtime() - start_time4;
        count_time[4]++;
        return(-1);
}
```

```
/*****************************************************************************************************
*  Description: Retrieves data off of network and determines checksum status
*  Inputs:  long fd - file descriptor that points to channel being used
*           char *header - points to where header should be placed
*           char *buffer - points to where received data should be placed
*           long length_in - length of data received
*           ATM_chan_info *ATM_chan_param - points to structure that contains channel variables
*  Outputs: return 0 - OK
*           return 1 - bad checksum
*           return -1 - error
*****************************************************************************************************/

char ATM_recv(long fd, char *header, char *buffer, long *length_in, ATM_chan_info *ATM_chan_info_in) {

      register i;
      unsigned long long register data_sum = 0;
      char data_sum_char = 0;
      long passed_sum = 0;
      long frac_length;
      long long *buffer_llong = (long long *)buffer;

#ifdef PRINT_RAW_RECV_DATA
      printf("Receiving...\n");
#endif
      if (atm_recv(fd, header, HEADER_SIZE) < 0) {          /* Retrieve header from Fore API */
            atm_error("atm_recv");
           return(-1);
        }

      for (i = 0; i < HEADER_SIZE - 5; i++) data_sum_char += header[i]; /* Verify Header checksum */

      if(data_sum_char != header[HEADER_SIZE - 5]) {
            /* printf("\n*****************************\n********Bad Header
                  Checksum**********\n*****************************\n");
            printf("data_sum_char = %d, passed header sum = %d\n", data_sum_char, header[HEADER_SIZE - 1]);
            */
            length_in = 0;
            return(1);
```

```
      }

        *length_in =    ((long)(unsigned char)header[2] << 24) +        /* Determine length from header */
                  ((long)(unsigned char)header[3] << 16) +
                  ((long)(unsigned char)header[4] << 8) +
                  ((long)(unsigned char)header[5]);

        if (*length_in > 0) {
            if (atm_recv(fd, buffer, *length_in) < 0) {                /* receive payload */
                  atm_error("atm_recv");
                  return(-1);
            }
        }
      packets_received++;
start_time5 = gethrtime();

      frac_length = *length_in / 8;                                    /* Verify payload checksum */
      for (i = 0; i < frac_length; i++) data_sum += buffer_llong[i];
      data_sum = (long)data_sum + (long)(data_sum >> 32);

      passed_sum= ((int)(unsigned char)header[HEADER_SIZE - 4]<<24) +
                  ((int)(unsigned char)header[HEADER_SIZE - 3]<<16) +
                  ((int)(unsigned char)header[HEADER_SIZE - 2]<<8) +
                  ((int)(unsigned char)header[HEADER_SIZE - 1]);

#ifdef PRINT_RAW_RECV_DATA
      for (i = 0; i < HEADER_SIZE; i++) printf("%d * ", (unsigned char)header[i]);        /* print out data
                                                                                            being sent with checksum */
      for (i = 0; i < *length_in / 8; i++) printf("%lld ", (unsigned long long)buffer_llong[i]);
                                                 /* print out data being sent with checksum */
      printf("Length, = %ld, Passed Sum = %ld, Checksum = %ld\n", *length_in, passed_sum, (long)data_sum);
      printf("\nReceived\n\n");

#endif
      if ((long)data_sum != passed_sum) {                              /* See if checksums match */
            /* printf("\n*****************************\n********Bad Data
            Checksum**********\n*****************************\n"); */
            total_time[5] += gethrtime() - start_time5;
```

```
                count_time[5]++;
                return(1);
        }


        total_time[5] += gethrtime() - start_time5;
        count_time[5]++;
        return(0);
}
```

```
/*****************************************************************************************************
*  Description:   "Timeout Thread" - Continuously checks to see if timeouts have occurred
*  Inputs:  ATM_chan_info *ATM_chan_param - points to structure that contains all variables needed by
*           channel
*  Outputs: none
*****************************************************************************************************/

void *ATM_timeout_timer(void *temp) {
      ATM_chan_info *ATM_chan_info_in = (ATM_chan_info *)temp;
      hrtime_t temp_time;
      long temp_frame_begin;
      char out_header[HEADER_SIZE];

      while(1) {
            mutex_lock(&ATM_chan_info_in->send_mutex);
            mutex_lock(&ATM_chan_info_in->recv_mutex);

            temp_time = gethrtime();

            temp_frame_begin = ATM_chan_info_in->out_buffer_begin / MAX_PACKET_SIZE;
            if ((temp_time - ATM_chan_info_in->time_sent[temp_frame_begin] >= (long long)TIMEOUT *
                  1000000000) && (ATM_chan_info_in->out_buffer_cur != ATM_chan_info_in->out_buffer_begin))
                  {
                  /* printf("***Timeout***\n Current Time = %lld, Time_sent = %lld, Out_buffer_begin = %ld,
                        Out_buffer_cur = %ld, Out_buffer_end = %ld\n",
                  temp_time,
                  ATM_chan_info_in->time_sent[temp_frame_begin],
                  ATM_chan_info_in->out_buffer_begin,
                  ATM_chan_info_in->out_buffer_cur,
                  ATM_chan_info_in->out_buffer_end);
                  */
                  timeouts++;
                  ATM_chan_info_in->out_buffer_cur = ATM_chan_info_in->out_buffer_begin;
                  mutex_unlock(&ATM_chan_info_in->recv_mutex);
                  mutex_unlock(&ATM_chan_info_in->send_mutex);
                  ATM_GBN_empty_outbuffer(ATM_chan_info_in);
                  mutex_lock(&ATM_chan_info_in->send_mutex);
                  mutex_lock(&ATM_chan_info_in->recv_mutex);
```

```
            }

            mutex_unlock(&ATM_chan_info_in->recv_mutex);
            mutex_unlock(&ATM_chan_info_in->send_mutex);
            thr_yield();
            sleep(TIMEOUT);
        }
}
```