File Transfer Protocol Design and Implementation using the ACE Networking Framework

### Ryan Foster, Brian Lockwood and Jordan Scherer

**Computer Science Department**

### University of Wisconsin-Eau Claire

#### Eau Claire, WI 54702

#### {lockwobr, fosterrr, schereje}@uwec.edu

#### Abstract

This suite was designed to meet the specifications of a simple file transfer protocol system using datagram sockets and the ACE networking framework. The program functionally implements error detection using the internet checksum algorithm. It has the option of using several protocols for datagram transfer; such as selective repeat, go-back-n, and stop-and-wait. Evaluation of each transfer protocol is analyzed under a variety of simulated network conditions.

#### Introduction

The goal of this project was to implement a simple file transfer protocol using datagram sockets. The ACE networking framework was chosen for designing the program to ease cross-platform compatibility. The main goals for the program included:

* Error detection
* Implementation of Stop and Wait, Go-Back-N, and Selective Repeat
* Timeouts
* Instances capable of handling multiple connections

##### Limitations

The main requirements for this project made the design slightly more complicated that a trivial file transfer protocol implementation.

First and foremost, datagram sockets were required. The use of datagram sockets implies that packets may arrive out of order. Therefore sequencing and packet ordering needed to be accounted for.

Another significant constraint was that the program would be required to run on both x86 architectures, and SPARC architectures. This specification was influential in choosing to design our program with the ACE networking framework.

##### Design

Since the ACE networking framework was chosen for implementation, we were required to get intimate with the reactor design pattern.

The reactor design pattern is a concurrent programming pattern for handling service requests delivered concurrently to a service handler by one or more inputs[1]. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.

Using this reactor pattern, we designed our program such that each client and server would contain a reactor, and each client and server would be service handler.

The reactor on the server side is responsible for monitoring timer events for packet timeouts, as well as monitoring input for a particular directed datagram connection.

The client side too implemented a reactor, which was responsible for listening for input connections on an established datagram connection and delegating appropriate reactions upon such event triggering.

##### Implementation

By choosing the ACE networking framework, many low level details that are often meticulously designed were overlooked. For instance, using the reactor pattern eliminated the need for threading. Also, many low-level memory operations and file I/O operations were simplified using the ACE networking framework.

Since we were to develop the program on multiple architectures as well, using the ACE framework ensured smooth cross platform compatibility.

* 1. **Sender implementation**

Using the reactor necessitated a server endpoint implementation, which would inherit from the ACE\_Event\_Handler class. The ACE\_Event\_Handler class implements a variety of functions that are called by the ACE\_Reactor when the an event is triggered that is associated with a registered ACE\_Event\_Handler.

The server endpoint implemented to the following classes virtually from ACE\_Event\_Handler.

* handle\_input()
* handle\_close()
* handle\_timeout()

**4.1.a handle\_input()**

Upon registering an event handler with the reactor, a certain mask is associated with a registered event such that the reactor will trigger the event handler if a certain phenomenon is observed by the reactor. For instance, the server endpoint was registered with a READ\_MASK, notifying the reactor that it would be interested in *reading* data.

The handle\_input() function is the default function that is called by the reactor for an event handler when a READ is triggered.

The implementation of handle\_input() at the server endpoint was responsible for identifying the peer and closing action based on resuming a peers request or starting a new request. An STL map using containing the string of the ip and port number as the key mapped to a *window* object containing all of the peer’s current connection status information was created. The object was then passed to another function that was responsible for creating frames with the appropriate binary data for the file transfer, and transmitting frames based on the conditions of the chosen algorithm (i.e, selective repeat).

**4.1.b handle\_close()**

The handle\_close() function polls for exit conditions and is necessitated to close the program cleanly. Little extraneous memory management was necessary in this function.

**4.1.c handle\_timeout()**

The handle\_timeout() function is called when a registered timer event expires. Per our implementation, each client-session has a timer on each packet necessary for managing timeouts. Upon registering a reactor, a void\* can be associated with each timer, and passed to the handle\_timeout() upon timer expiration. This allowed us to hold important user data about timer and frame association upon each timer registration.

##### 4.2 Receiver implementation

The receiver also inherits virtual members from ACE\_Event\_Handler, so it is required to implement handle\_input(), and handle\_close().

**4.2.a handle\_input()**

Similarly, handle\_input() on the client side polls a datagram socket and listens for inputting data. Upon receiving data from the datagram socket, handle\_input() passes the packet to a *processPacket()* function which determines if the packet is expected based on the underlying protocol chosen. If appropriate, the packet is dissected and the payload is appended the file for writing.

**4.2.b handle\_close()**

As in the implementation of the server endpoint, the handle\_close() member is required to clean up any necessary allocated memory so the program can exit gracefully.

* 1. **Common implementation**

Several classes were created that were implemented on both the client-side, and the server-side. The backbone of our implementation pivoted on the functionality of the *window* class. Each connection requires a window object on both the sending and receiving side.

**4.3.a Window**

Each window contains a number of frames that is equal to the maximum sequence number. In selective repeat, this is equal in both the sending and receiving sides. Depending on the protocol, the window is responsible for sliding and shifting, and determining what packets should be stored in the window. The window class is also responsible for reading and writing payload data from/to a file depending on if it is of a receiving or sending endpoint.

**4.3.b Packet**

The packet class is also common to both the sender and receiver endpoint implementations. It’s layout was inspired by similar formatting in Michael LeMay’s implementation. [5] The packet is constructed as follows.

class packet{

public:

packet(const packet &p);

…

u\_int16 seqNum;

u\_int8 type;

u\_int32 len;

u\_int8\* payload;

u\_int16 CRC;

…

};

Each data contains a sequence number, necessary for re-ordering packets sent using UDP. The sequence number is unsigned and has a maximum of two bytes.

A type, which is only 1 byte of the packet, is used for a variety of internal commands. Many types are used for calibrating and configuring the connection. Such types are the handshake type, and the ping type. There is also an acknowledgement type, and a simple data type.

The payload is variable length, and the CRC is the internet checksum that is tagged on the end of every packet for error detection.

* 1. **Protocol Implementations**

Various protocols are available for implementation, such as Stop and Wait, Go-Back-N, and Selective repeat.

**4.4.a Stop and Wait**

Various protocols are available for implementation, such as Stop and Wait, Go-Back-N, and Selective repeat. Stop-and-wait ARQ is the simplest kind of automatic repeat-request (ARQ) method. A stop-and-wait ARQ sender sends one frame at a time. After sending each frame, the sender doesn't send any further frames until it receives an ACK (acknowledgement) signal. After receiving a good frame, the receiver sends an ACK. If the ACK does not reach the sender before a certain time, known as the timeout, the sender sends the same frame again.[2]  For Stop and Wait, the window size on each side is one.

**4.4.b Go-Back-N**

Go-Back-N is another protocol in which the sending process continues to send a number of frames specified by a window size even without receiving an ACK packet from the receiver.[3] In this protocol, the sender’s window size may be any positive integer *N*, however the receivers window size is always one.

**4.4.c Selective Repeat**

In selective repeat, the sender and the receiver window side are always equal to preconfigured integer *N*. Since the receiver has a window of *N* frames, it is significantly more efficient that Go-Back-N because it can retain a certain amount of frames that are out of order. The efficiency will be discussed later.

##### Testing

For the testing of our program we decided to split the test we did into different types. The two different types of testing were running the code normally and then running the code while forcing error into the code such as purposely dropping and corrupting the packet. Also since the Stop and Wait algorithm is just the Selective Repeat and Go-Back-N algorithm with a window size of 1, we decided that it was trivial to test it and didn’t include it in our testing.

**5.1 Non-Error Testing**

There are countless different parameters that could be adjusted when testing the different types of packet sending algorithms. The three major parameters that we decided to test were packet size, window size, and timeout length. All three of the different parameters will be measured by their effective throughput.

**5.1a Packet Size**

For testing throughput times for varying packet sizes a 5MB file was sent if a timeout 0.5s and a window size of 10. The packet size varied from 1kB to 64kB for both the Selective Repeat and Go-Back-N algorithms and this yielded the follow results.

For all of the different packet sizes Selective Repeat had a higher throughput than Go-Back-N. Another thing that is noticeable in the data collected is that the throughput seems to go up as the packet size goes up until it a certain point and then it starts to drop off. This is because as the packet size becomes larger, the chances of packets dropping increases and they have to be resent.

**5.1b Window Size**

Once again for this testing the same 5MB file was sent with a packet size of 8kB and a timeout of .25s. The window size tested ranged for 2 to 256 and this yielded the following results.

There is a very noticeable correlation between window size and throughout. As the window size goes up so does the throughout for both Go-Back-N and Selective Repeat. Also the Selective Repeat algorithm seems to be superior to the Go Back algorithm for all window sizes.

**5.1c Timeout Length**

For testing how a timeout length affects the throughput of a file 20kB packets were sent with a window size of 10. The same 5MB file was sent with timeout lengths varying from 0.25s to 1.5s and this gave the following results.

For the third time in the row Selective Repeat has come out on top for the two algorithms. For the most part as the timeout time goes up the throughput goes down. The one exception to rule is from 0.25s to 0.5s the throughput goes up for selective repeat. This is probably explain by if the timeout is to small then the packet is timing out before the ack can even return to the server side and because of this the server floods the network with unnecessary packets. As the timeout gets to big the then the server is waiting far too long for the packet to send it again. If you were to extend the graph out farther you would really see a vey bell shaped graph.

**5.2 Error Testing**

For testing how the different protocols handle errors we introduced to major ways of creating error. The first way of doing this is corrupting the packet so that the CRC check sum fails. The second way is to just loose packets. This is simulated by when a packet is received it is just ignored like it never got there.

**5.2a Corrupting Packets**

For the testing of the how the different protocols work when packets are corrupted we setup a test where a 10% of the packets are corrupted on the server side so that the CRC check sum fails on the client side. This was down with a window size of 5 and a timeout length of 0.5s. The test was then done with varying packet sizes ranging from 1kB to 65kb packets.

The graph shows the absolute throughput (ATP) and effective throughput (ETP) for both Selective Repeat and Go-Back-N at the varying packet size length. For the smaller packet sizes the ATP and ETP are fairly close, but as the packet size increase the figures these significantly differ. The ATP becomes extremely high compared to the ETP. This is probably due to the fact that when the smaller packets are corrupted you don’t have to resend as much data so the throughput are basically the same, but with the large packets large amounts of data is corrupted and must be resent so you get a huge difference.

**5.2b Loosing Packets**

For the testing of loosing packet 5% of the total packets on the client side and the 5% of the ack on the server side are lost. Like the previous test a 2MB file was sent with a window size of 5 and a timeout length of 0.5s

The data collected for this is turned out to be a lot like the data from the last test, where the ATP is basically the same as the ETP at small packet sizes and significantly higher at larger packet size. This further proves that if a packet becomes lost or corrupted, then the bigger the packet is the bigger the penalty there is to throughput.

##### Summary

There are many things that can be taken from doing this project. The major thing that everyone in the group can agree on is how nice it is to use a framework to complete a project. While the ACE framework was rather difficult to get use to in the beginning due mostly to lack of documentation, overall it made the project easy to implement. As far as the different protocols goes we have found that the Selective Repeat is by far the best protocol to using in almost all situations.

##### References

1. Reactor Pattern, http://en.wikipedia.org/wiki/Reactor\_pattern
2. Stop-and-wait ARQ , http://en.wikipedia.org/wiki/Stop-and-wait\_ARQ.
3. Go-Back-N ARQ <http://en.wikipedia.org/wiki/Go-Back-N_ARQ>
4. Douglas C. Schmidt. The ADAPTIVE Communications Environment (ACE). <http://www.cs.wustl.edu/~schmidt/ACE.html>.
5. Michael LeMay and Jack Tan. Abstracting Low-Level Network Programming with ACE, a Pattern-Oriented Network Programming Framework. <http://lemaymd/pubs/ftpr.pdf>, 2004.