RDT 3.0 - A reliable large datagram protocol

Steven Clark

May 13, 2017

1 Introduction

Our class was assigned to create a reliable transport protocol running as an applpication layer API over UDP. Rather than cloning the bidirectional streaming conneciton oriented TCP this protocol emulated a message transmission protocol like UDP itself, but capable of much larger transmissions more reliably. To craft my own implementation I drew features from across the network stack. The fragmentation system from IPv4 was used instead of the sequence numbers. Multiple flags were used mildly resembling TCP. Overall the protocol meets the limited stop-and-wait case required by the assignment.

2 Methodology

RDT 3.0 was written in C mostly to make the creation of a C static function library practical, and because there were no difficult concepts to warrant object oriented design or the use of the STL. The API consists of the following functions:

```
int rdt_socket(int addr_family, int type, int protocol);
```

Equivalent to the socket system call with the same syntax.

```
int rdt_bind(int sockd, const struct sockaddr* localaddr, socklen_t addrlen);
```

Equivalent to the bind system call.

```
int rdt_recv(int sockd, char* buffer, int bufflen, int flags,
                        struct sockaddr* addrfrom, socklen_t* addrlen);
```

This function replaces the recvfrom sytem call. It receives a UDT 3.0 connection and fills a buffer of arbitrary length with the message sent.

```
int rdt_send(int sockd, char* buffer, int bufflen, int flags,
                        struct sockaddr* addrto, socklen_t addrlen);
```

Replaces the sendto system call providing the ability to send one contiguous buffer, of length up to the limits of a 32bit address reliably to a destination socket.

```
int rdt_close(int fd);
```

Behaves identically to the close system call.

```
int set_drop_q(int mils);
```

Set a probability that rdt_send will drop any one of its' packets. Measured in mils.

```
int set_corrupt_q(int mils);
```

Set a probability that rdt_send will corrupt any one of its' packets. Measured in mils.

```
uint16_t checksum_of(void * buffer, size_t length);
```

The checksum function used by rdt_send and rdt_receive. Provided as a useful utility.

2.1 Packet format

Every datagram sent by RDT 3.0 contains a header with the following fields:
A 16bit unsigned payload length
A 16bit checksum of the entire packet with the checksum field set to 0
A 32bit unsigned offset indicating where the data should be written in the buffer, also serving as a sequence number.
29 bits of currently unutilized padding
3 flags for further fragments, transmission packet, and acknowledgement packet.

2.2 Transmission algorithm

When called the rdt_send function sets a reading pointer at the beginning of the buffer. It will then build a packet and copy the minimum of one full payload length or the remaining data in the buffer into the packet, initialize its' length, and set offset of the packet to the current location of

the read pointer within the buffer. The packet is then checksummed and sent to the destination socket. If an ack packet with an offset inidicating all sent data was accepted is received the read pointer will be incremented and a retry counter zeroed. Otherwise the retry counter is incremented. In both cases the next packet is sent or resent until an acknowledgement past the end of the buffer is received. If the retry counter reaches a preset limit or another unrecoverable error occurs the function will terminate early returning 0 minus the number of bytes that were succesfully sent. Otherwise the function returns the number of bytes sent.

2.3 Receipt Algorithm

When called the rdt_receive function sets a write pointer to the start of the buffer. It then blocks waiting for a RDT packet on the given socket. When an RDT packet is received, it's checksum is verified, and its' offset is checked against the position of the write pointer within the package. If either is incorrect the response packet offset is set using the current write pointer index. Otherwise the payload is copied into the buffer at the write pointer, the write pointer is incremented, and the ACKs offset is written using the new write pointer's index. For both cases length and cehcksum are caclulated and filled in and the package is sent. This repeats until a valid packet is received with a zeroed fragment flag upon which the function returns the number of bytes written to the buffer on exiting the loop. If an unrecoverable error occurs the function will return 0 minus the number of bytes that were written before the error occured.

Debug outputs were written to both these functions so if a packet was dropped at the sender, or a corrupted packet received at the reciever it would print that out to standard error. In a production library this information would be invisible unless requested.

2.4 Testing Programs

2.4.1 RDTrecv

A simple server was written to test the protocol. RDTrecv binds a port number provided on the command line and repeatedly listens for RDT3.0 messages. Upon receiving a message it prints it to the console error stream. If the call returns 0 or negative it will notify thr user there was an error receiving the message and continue.

2.4.2 RDTsend

Another simple program was used to send the test messages to RDTrecv. RDTsend takes a file path, an IP address, a port number, a drop rate, and a corruption rate from the command line. The file is then memmory mapped and the resulting buffer sent to the destination socket using rdt_send. If rdt_send returns 0 or negative the user is informed there was an error sending the file. A long and short message file were created for testing, consisting of one run through the lowercase English alphabet and amny such runs respectively. This was so that duplications or omissions would be easily visible.

3 Results

Under testing the library performed well. Corrupt and out-of-order packets were handled by retransmission so swiftly and silently that debug messages had to be added to the library function to see that they were working. Dropped packets were noticeably significantly slower, which fits expectations as it took a timeout to find them. The debug message for a complete failure to transmit only ever occurred when the server was not running to receive packets or the error rates were set unreasonably high, in excess of 30 percent. Overall the library behaved exactly as expected.

The library strictly meets the requirements of the assignment, but does not exceed them. The output on receiving side is the same as the input on the sending side, even with corruption or dropping enabled. An offset of 0 is used to start the transport session which doesn't complete until a datagram with a 0 fragment flag is received and acknowledged. The offset ensures that packets are simpoly not accepted out of order, since the protocol is stop-and-wait duplicate, missing, or delayed packets are the only ways packets are likely to arrive out of order and this protocol handles them. A select call timeout is used when waiting for acknowledgements ensuring that missed packets or acknowledgements result in a timely retransmission. Packets retry a fixed number of times and then the send process emits an error.

There are still sevral ways the library could have been improved. Firstly messages have no ID field and the offsets are not randomly started like TCP sequence numbers. Given the fixed maximum payload, multiple sessions will generate valid packets with identical offsets which if heavily delayed could be received in the middle of a subsequent session in error. Also, if the acknowledgement of the final fragment were to be lost it will never be resent, the receiver having left the session while the sender waits for a response that

won't come. A critical error in the sender will not terminate the connection at the receiver side and vice versa. Both of these could be handled with some sort of closing handshake system and there are more than enough unused bits in the header to implement this. There is also low hanging fruit for potential upgrades. A trasmission window could be implemented relatively easily with little or no changes to the protocol. The offset and fixed payload length would allow a map of received and unreceived buffer space to be created. This would allow for accepting out of order packets if valid instead of simply ignoring them.

4 Conclusion

The library has its flaws, but it works. The implementation is needlessly primitive, and it would never really replace TCP as a way of sending messages reliably. That same primitve implementation serves as an interesting demonstration: It is actually possible to implement a reliable protocol like this one very simply if performance is not a goal. The actual design, from the packet structure to the send and receive loops was designed to be implemented with as little additional complexity as possible. As a proof of concept it works very well.