CSE 123 Fall 2011 Project #2
Instructor: Stefan Savage
TAs: Alex Rasmussen, Nima Nikzad

Due Sunday Dec 4th 12pm (i.e., lunch) by electronic hand in

Overview

In this programming assignment, you will implement a simple, reliable, sequenced message transport protocol (SRMP) on top of UDP. UDP provides point-to-point, unreliable datagram service between a pair of hosts. In this programming assignment, we will develop a more structured protocol which ensures reliable, end-to-end delivery of messages in the face of packet loss, preserves ordering of transmitted messages, and preserves message boundaries.

Unlike TCP, SRMP preserves message boundaries, delivering entire messages to the application, and thus is not a logical bytestream. Also, SRMP will not provide congestion control mechanisms, although it will support window-based flow control using a receiver-advertised window. Finally, whereas TCP allows fully bidirectional communication, our implementation of SRMP will be asymmetric. The two SRMP endpoints will be designated the "sender" and the "receiver" respectively. Data packets will only flow in the "forward" direction from the sender to the receiver, while acknowledgments will only flow in the "reverse" direction from the receiver back to the sender.

To support reliability in a protocol like SRMP, state must be maintained at both endpoints. Thus, as in TCP, connection set-up and connection teardown phases will be an integral part of the protocol.

As we mentioned earlier, congestion control will not be a part of SRMP. However, this does not imply that a sender can transmit as fast as it pleases. SRMP uses window-based flow control whereby the receiver regulates the sender by transmitting an advertised window (as in TCP) which bounds the amount of data SRMP can inject into the network at any point in time.

SRMP

Packet Headers

SRMP packets have 3 fields, as specified in the following structure.

```
typedef struct {
    unsigned short type;
    unsigned short window;
    unsigned short seqno;
  } srmphdr;
```

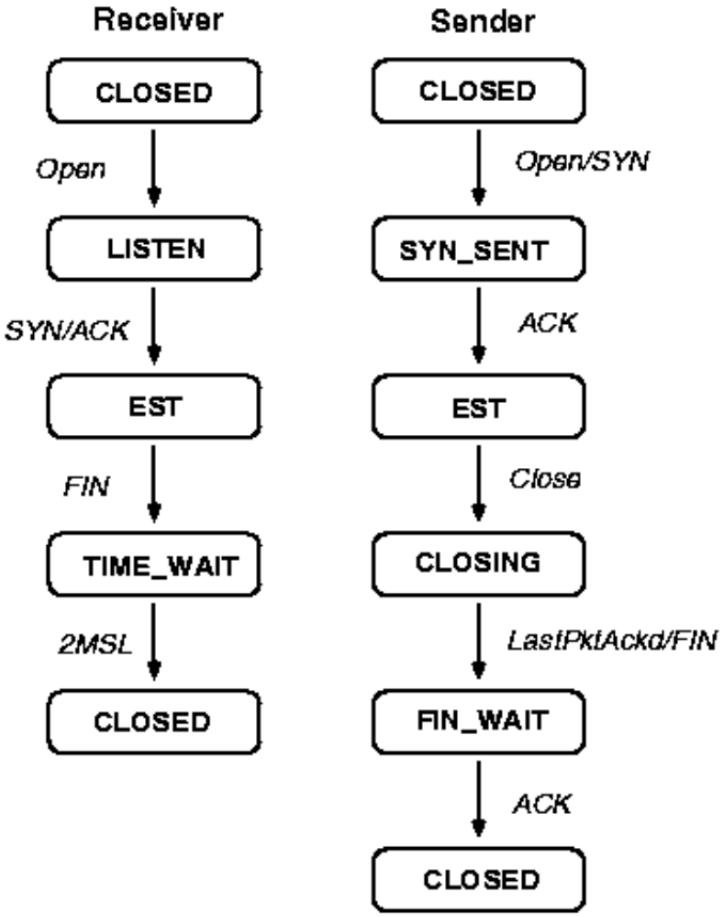
The "type" field takes on 5 possible values (thus in practice, we would allocate far less than a short to this field). DATA = 0, ACK = 1, SYN = 2, FIN = 3, RESET = 4. Unlike TCP, in which similar codes specify individual bits in the "flags" field and multiple bits can be set simultaneously, SRMP packets must be of exactly one of the types specified above.

The "window" field specifies the receiver's advertised window (in bytes) and is used only in packets of type 1 (ACK). The window field must be set to zero for all packets of other types. The maximum window size is $2^{16} - 1$.

The "seqno" field indicates the sequence number of the packet. This field is used in all packets except RESET packets, where it is set to zero. For DATA packets, the sequence number increases by the size (in bytes) of each packet. For ACK packets, the sequence number acts as a cumulative acknowledgment, and indicates the number of the next byte expected by the receiver. For SYN packets, the sequence number is one smaller than the sequence number of the first data packet of the connection. For FIN packets, the sequence number is one larger than the sequence number of the last byte of the last data packet of the connection. The maximum sequence number is $2^{16} - 1$, and the maximum message size in SRMP is 1000 bytes.

State Diagram

The asymmetry between sender and receiver in this protocol leads to somewhat different state diagrams for the two endpoints. The state diagram for SRMP is as follows:



The receiver can be in four possible states: CLOSED, LISTEN, ESTABLISHED and TIME_WAIT. Initially, it is in the CLOSED state. Upon issuing a passive open, it enters the LISTEN state. (Note that the receiver is the passive host in our protocol, while the sender actively opens the connection). While in the LISTEN state, the receiver waits for a SYN packet to arrive on the correct port number. When it does, it responds with an ACK, and moves to the ESTABLISHED state. After the sender has transmitted all data (and received acknowledgments), it will send a FIN packet to the receiver. Upon receipt of the FIN, the receiver moves to the TIME_WAIT state. As in TCP, it remains in TIME_WAIT for two maximum segment lifetimes (MSLs) before re-entering the CLOSED state. Assume for this assignment that the MSL is 2 seconds.

The sender can be in five possible states: CLOSED, SYN_SENT, ESTABLISHED, CLOSING, and FIN_WAIT. Like the receiver, the sender starts in the CLOSED state. It then issues an active open by sending a SYN packet (to the receiver's port), thus entering the SYN_SENT state. This SYN transmission also includes the initial sequence number (ISN) of the conversation. The ISN should be chosen at random from the valid range of possible sequence numbers. Any packet (including the SYN packet) which is not acknowledged in 2 seconds must be retransmitted. If the SYN packet is not acknowledged after three attempts, a RESET packet must be sent to the destination port and the sender moves to the CLOSED state. In the common case in which the SYN is acknowledged correctly (the ACK must have the correct sequence number = ISN + 1), the sender enters the ESTABLISHED state and starts transmitting packets. When the sending application (sitting above SRMP) is finished generating data, it issues a "close" operation to SRMP. This causes the sender to enter the CLOSING state. At this point, the sender must still ensure that all buffered data arrives at the receiver reliably. Upon verification of successful transmission, the sender sends a FIN packet with the appropriate sequence number and enters the FIN_WAIT state. Once the FIN packet is acknowledged, the sender re-enters the CLOSED state. If no ACK comes after three timeouts (which are also 2 seconds each), the sender should send a RESET packet and return to the CLOSED state.

In the event that one party detects that the other is misbehaving, it should reset the connection by sending a RESET packet. For example, one easy way in which a receiver can misbehave is to acknowledge a packet that has not yet been transmitted.

Implementation

We provide an implementation of the SRMP receiver process in C; you must write the sender in C as well. In the skeleton code that we provide, applications will invoke the sender's functionality by means of the following application programmer's interface (API) consisting of the following three routines:

- `int srmp_open (srmp_send_ctrl_blk *srmp_cb,char *dst,int sport,int rport);`
- `int srmp_send (srmp_send_ctrl_blk *srmp_cb, char *data, int len);`
- `int srmp_close (srmp_send_ctrl_blk *srmp_cb);`

Similar to common implementations of TCP, you need a global and central "structure" or "object" to help you manage the things such as state transitions, sliding windows, and buffers. On the receiver's side, we call this structure the "SRMP RECEIVE CONTROL BLOCK". You can use srmp_recv_ctrl_blk defined in srmp.h as a reference for your design of the control block on the sender's side.

The first function (srmp_open) is called once by the application at the sending side to open and establish an SRMP connection to a remote receiver. When it succeeds in establishing a connection with the remote receiver, the srmp_send_ctrl_blk should have been initialized with the corresponding data. The second function (srmp_send) is called repeatedly by the application to transmit messages across an established connection, and the final function (srmp_close) is used by the application when it has no more messages to transmit, and wishes to teardown the connection.

While a typical protocol implementation would consist of many threads of control, we will simplify things by using a single-threaded approach to implementing the sender's functionality. One possible decomposition of srmp_send() might be as follows:

• Copy the message passed to srmp_send() into a packet processing buffer.

• Process any ACKs that are ready immediately (calling readWithTimer in srmp.c with a timeout of zero).

• Then, until the sliding window is full, retransmit any timed-out packets or packets that have newly arrived in the packet processing buffer.

• Return.

By continuously calling srmp_send(), the data transfer will proceed smoothly.

Eventually, when the application has no more data to transmit, it calls srmp_close(). At this point, your SRMP connection must make sure all buffered data is transmitted and acknowledged before tearing down the connection.

The Makefile builds a simple file transfer application which consists of two executables. "SendApp" is an application which takes three command line arguments: a destination address (da), a destination port (dp), and a source port (sp). The application uses the send-side API to call the srmp_ routines that you will write to transfer the contents of a file called "TestInput" to a remote receiver. On the other side of the connection, "ReceiverApp" is an application which expects to be sent a file, and uses the receive-side API to call the srmp_ routines (provided) to get the file, then writes it to a file called "TestOutput".

To recap, you can build "ReceiveApp" with the Makefile and these source files (all available for download from http://cseweb.ucsd.edu/classes/fa11/cse123-a/Project2/)

- `receiver.c`
- `reciever_list.c`
- `wraparound.c`
- `srmp.c`
- `srmp.h`

by typing "make ReceiveApp". Once you've finished creating sender.c (we recommend building it from the template provided at from http://cseweb.ucsd.edu/classes/fa11/cse123-a/Project2/), type "make SendApp".

Testing

To test your code, we provide two receivers for you to test your sender against.

Source code is provided for a basic receiver which has a small, fixed advertised window and does not lose packets. Correct behavior against this receiver will be worth approximately 3/4 of the credit. A more complex receiver which varies its window size and simulates a network which reorders and drops packets is also available for you to test against. You should do all (or almost all) of your testing in a local environment without going over the network. By routing packets to "localhost", those packets are sent via a special software interface known as the loopback interface, which redirects the packets immediately back to the same host. Thus, you can run your receiver in one window:

```
% ./ReceiveApp localhost 4562 4182
```

and then run the sender in another window:

```
% ./SendApp localhost 4182 4562 SendFileName
```

(Note how we reverse the source and destination port numbers so the programs plug together naturally – note that Unix will require you to use port numbers above 1024 since you don't have superuser permissions; SendFileName is the name of file the application will send).

Simplifications

You need not worry about the following aspects of the implementation.

- Round-trip time estimation. Any packet not acknowledged within 2 seconds has been lost and must be retransmitted.
- Congestion control. You only need to worry about overrunning the receiver's advertised window, but ignore network congestion. (Lack of congestion control should motivate you to do your testing on a local host. Try not to get us in too much trouble with the network administrators.)
- The CLOSED state need not be implemented at the sender. Upon start-up, just have the sender immediately send a SYN and enter SYN_SENT.

On the other hand, don't oversimplify. You **will** need to deal with:

- Flow control – make sure you obey the advertised window constraints.
- Multiple data packets in flight. Programs which use the Stop-and-Wait algorithm will at most receive half credit.
- Retransmission of packets which have timed out.

- Sequence number wrap-around. Your senders should choose an ISN at random, and should deal with the problem of the sequence number wrapping around to zero.

How to Proceed

Sometimes assignments such as this can be daunting and it can be hard to figure out where to start. In general, implement the solution piece by piece, testing thoroughly as you go.

Here are some suggestions:

- Start by reading and understanding the main ideas of the receiver code.
- Use the sendpkt and receivepkt routines provided in srmp.c to set up a connection between the sender and the receiver.
- Implement a simple transfer protocol, like Stop-and-Wait, ignoring the complexity of the receiver's advertised window.
- Test the correctness of your basic transfer routine on very small to very large files (megabytes).
- Also make sure sequence number wraparound is working properly.
- Then implement a proper sliding window accounting for the advertised window and retransmission of lost packets.
- Make sure that everything works 1) for large files, 2) against the augmented receiver
- You're done!

Submission and Grading Guidelines

You will be turning in only the sender.c file (you should not depend on changing any other files). Make sure it compiles using the standard Makefile and test your solution Make sure your username and student id is in the header of this file. We will be posting electronic turn in instructions shortly.

The provisional grading (contingent on how the project goes) is:

50% for getting the sender to work with stop and wait

20% for supporting multiple outstanding packets correctly

15% for correct delivery under loss

10% for supporting variable flow control windows

10% for correctly handling reordered packets

5% for managing sequence number wraparound correctly

(Yes, this adds up to 105%.) Original version of this assignment courtesy of John Byers and his Teaching Assistants at Boston University. Good luck!