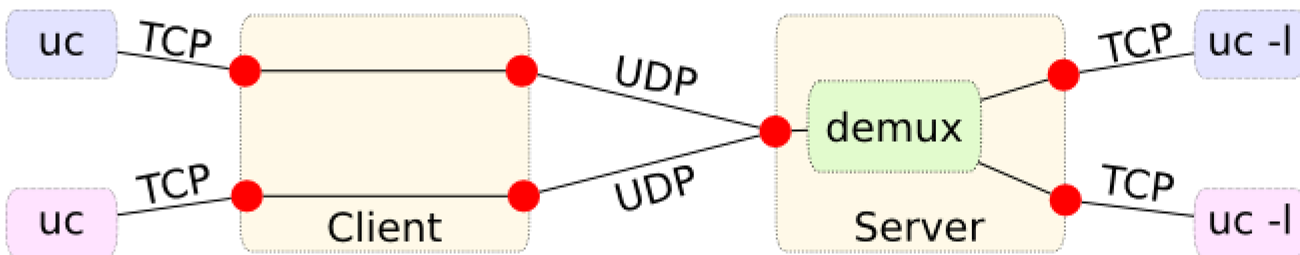
Labs 1 & 2: Reliable Transport

Due date: Thursday, September 30 @ the beginning of class (part 1).
Due date: Thursday, October 7 @ the beginning of class (part 2).

Introduction

Your task is to implement a reliable, stop and wait (Lab 1) and sliding window (Lab 2) transport layer on top of the user datagram protocol (UDP). You will use IP addresses and UDP port numbers to demultiplex trafic, but not otherwise rely on UDP--in particular you should not rely on UDP's checksum to detect bit errors in packets.

The assignment is split into two labs. In the first lab, you just need to support a single direct connection between two UDP ports, one on the server and the other on the client; you can use the stop and wait protocol for this. In Lab 2, you extend the functionality to support demultiplexing of several connections at the server as well as to support a sliding window with sizes > 1. (Recall that stop and wait is equivalent to sliding window with a window size of 1). The picture below shows how the system should look after Lab 2 is implemented:



In this assignment, you are provided with a library (rlib.h and rlib.c) and you have to implement some functions and data structures for which skeletons are provided (in reliable.c). You will probably find it useful to look through rlib.h, as several useful helper functions have been provided.

In general your implementation should:

- Handle packet drops.
- Handle packet corruption.
- Provide trivial flow control.
- Provide a stream abstraction.
- Allow multiple packets to be outstanding at any time (using a limit given to your program as a run-time parameter, via the -w option).
- Handle packet reordering.
- Detect any single-bit errors in packets.

You will implement both the client and server component of a transport layer. The client will read a stream of data in (either from STDIN, in the first lab, or from a reliable TCP connection for Lab 2), break it into fixed-sized packets suitable for UDP transport, prepend a control header to the data, and write this packet to the server. The server will read these packets and write the corresponding data, in order, to a reliable stream (STDOUT in Lab 1, and a TCP connection in Lab 2).

Packet types and fields

There are two kinds of packets, Data packets and Ack-only packets. You can tell the type of a packet by its length. Ack packets are 8 bytes, while Data packets vary from 12 to 512 bytes. The packet format is defined in rlib.h:

```
struct packet {
  uint16_t cksum; /* Ack and Data */
  uint16_t len;   /* Ack and Data */
  uint32_t ackno; /* Ack and Data */
  uint32_t seqno; /* Data only */
  char data[500]; /* Data only; Not always 500 bytes, can be less */
};
typedef struct packet packet_t;
```

Every Data packet contains a 32-bit sequence number as well as 0 or more bytes of payload. The `length`, `seqno`, and `ackno` fields are always in big-endian order (meaning you will have to use `htonl`/`htons` to write those fields and `ntohl`/`ntohs` to read them). Both Data and Ack packets contain the following fields:

`cksum`

> 16-bit IP checksum (you can set the cksum field to 0 and use the `cksum(const void *, int)` function on a packet to compute the value of the checksum that should be in there). Note that you *shouldn't* to call htons on the checksum value produced by the `cksum` function--it is already in network byte order.

`len`

> 16-bit total length of the packet. This will be 8 for Ack packets, and 12 + payload-size for data packets (since 12 bytes are used for the header). An end-of-file condition is transmitted to the other side of a connection by a data packet containing 0 bytes of payload, and hence a len of 12. Note: You must examine the length field, and should not assume that the UDP packet you receive is the correct length. The network might truncate or pad packets.

`ackno`

> 32-bit cumulative acknowledgment number. This says that the sender of a packet has received all packets with sequence numbers earlier than ackno, and is waiting for the packet with a seqno of ackno. Note that the ackno is the sequence number you are waiting for, that you

have not received yet. The first sequence number in any connection is 1, so if you have not received any packets yet, you should set the ackno field to 1.

The following fields only exist in a data packet:

`seqno`

Each packet transmitted in a stream of data must be numbered with a seqno. The first packet in a stream has seqno 1. Note that in TCP, sequence numbers indicate bytes. By contrast, this protocol just numbers packets. That means that once a packet is transmitted, it cannot be merged with another packet for retransmission. This should simplify your implementation.

`data`

Contains (len - 12) bytes of payload data for the application.

To conserve packets, a sender should not send more than one unacknowledged Data frame with less than the maximum number of bytes, 500. (This behavior is somewhat akin to TCP's Nagle algorithm, which we will discuss in lecture.)

Requirements

Your transport layer must support the following:

- Each side's output should be identical to the other side's input, regardless of a lossy, congested, or corrupting network layer. You will ensure reliable transport by having the recipient acknowledge packets received from the sender; the sender will detect missing acknowledgments and resend the dropped or corrupted packets.
- Your server should handle multiple client connections in lab 2.
- As reliable transport is inherently a stateful protocol, your transport layer should handle simple connection establishment. In this assignment, the server can detect a new connection when it receives a packet with a sequence number of 1 (which should always be the sequence number of the first packet in a new connection).
- You should handle connection teardown properly. When you read an EOF, you should send a zero-length payload (12-byte packet) to the other side to indicate the end of file condition. When you receive a zero-length payload (and have written the contents of all previous packets), you should send an EOF to your output by calling `conn_output` with a `len` of 0.
- For Lab 1, you can have the window size to be just one packet (the default). For Lab 2, you have to support larger window sizes. The window size is supplied by the `-w` command-line option, which will show up as the `window` field in the `config_common` data structure passed to the `rel_create` and `rel_demux` functions you implement.
- Your server and client should ensure that data is written in the correct order, even if the network layer reordered packets. Your receiver should buffer as many packets as the client may send concurrently. In other words, the sender window size (SWS) should equal the

receiver window size (RWS), and both should be the same as the `window` field in the `config_common` structure.

- The sender should resend a packet if the receiver does not acknowledge it within an appropriate time period. You need not implement any backoff like TCP, and can instead merely send packet(s) whenever a sent packet has gone unacknowledged for the timeout period. The timeout period in milliseconds is supplied to you by the `timeout` field of the `config_common` structure. The default is 2000 msec, but you may change this with the `-t` command-line option.
- Again, acknowledgements should be cumulative rather than selective. Remember that like TCP, you acknowledge the next sequence number you are expecting to receive, which is 1 more than the largest in-order sequence number you have received. You don't have to handle sequence number overflowing and wrapping in the lifetime of a connection.
- You can retry packets infinitely many times, and should make sure you retry at least FIVE times, after which, if you want, the client can terminate the connection with an error. You can call `rel_destroy` to destroy the state associated with a connection when you give up on retransmitting.
- Note: For debugging printfs you should use the Standard Error `fprintf (stderr, ...)` and not print on standard output. This is because standard output is being used for the actual program output and it will be confusing for the grader as well as the tester.

Implementation Details

There are two modes of operation of the reliable transport protocol:

The first mode is single-connection mode, and connects standard input and output of the two processes together. The second is multi-connection mode, in which a client accepts TCP (or unix-domain) socket connections and relays them over UDP to a server that connects to a TCP port or unix-domain socket.

You are provided with a library (rlib.h/rlib.c) and your task is to implement the following seven functions: `rel_create`, `rel_destroy`, `rel_recvpkt`, `rel_demux` (Lab 2), `rel_read`, `rel_output`, `rel_timer`:

- **rel_create**: The `reliable_state` structure encapsulates the state of each connection. The structure is typedefed to `rel_t` in rel.h, but the contents of the structure is defined in reliable.c, where you should add more fields as needed to keep your per-connection state. A `rel_t` is created by the `rel_create` function. When running in single-connection or client mode, the library will call `rel_create` directly for you. When running as a server, you will need to invoke `rel_create` yourself from within `rel_demux` when you notice a new connection, which will show up as a packet with sequence number 1 from a `sockaddr_storage` that you have not seen before (you can test for whether you have seen a connection before by using `addreq(const struct sockaddr_storage *, const struct sockaddr_storage *)` to compare a packet's source address to addresses you have seen before).
- **rel_destroy**: A `rel_t` is deallocated by `rel_destroy()`. The library will call `rel_destroy` when it receives an ICMP port unreachable (signifying that the other end of the connection has died). You should also call `rel_destroy` when all of the following hold:
    - You have read an EOF from the other side (i.e., a Data packet of len 12, where the payload field is 0 bytes).

    - You have read an EOF or error from your input (`conn_input` returned -1).
    - All packets you have sent have been acknowledged.
    - You have written all output data with `conn_output`.

  Note that to be correct, at least one side should also wait around for twice the maximum segment lifetime in case the last ack it sent got lost, the way TCP uses the FIN_WAIT state, but this is not required.

- **rel_recvpkt** and **rel_demux**: When a packet is received, the library will call either `rel_recvpkt` or `rel_demux`. `rel_recvpkt` is called when running in single-connection or client mode. In that case, the library already knows what `rel_t` to use for the particular UDP port receiving the packet, and supplies you with the `rel_t`. In the case of the server, all UDP packets go to the same port, so you must demultiplex the connections in rel_demux.

- **rel_read**: To get the data that you must transmit to the receiver, call `conn_input`. `conn_input` reads from standard input when running in single-connection mode, and from a TCP connection when running in client or server mode (thus abstracting away which mode you are in from the protocol implementation). If no data is available, `conn_input` will return 0. At that point, the library will call `rel_read` once data is again available again, so that you can once again call `conn_input`. **Do not loop calling `conn_input` if it returns 0**; simply return and wait for the library to invoke `rel_read`!

- **rel_output**: To output data you have received in decoded UDP packets, call `conn_output`. This function outputs data either to STDOUT or to a TCP connection, depending on the mode of operation. You may find the function `conn_bufspace` useful--it tells you how much space is available for use by `conn_output`. If you try to write more than this, `conn_output` may return that it has accepted fewer bytes than you gave it. You must flow-control the sender by not acknowledging packets if there is no buffer space available for `conn_output`. The library calls `rel_output` when output has drained, at which point you can call `conn_bufspace` to see how much buffer space you have and send out more Acks to get more data from the remote side.

- **rel_timer**: The function rel_timer is called periodically, currently at a rate 1/5 of the retransmission interval. You can use this timer to inspect packets and retransmit packets that have not been acknowledged. Do not retransmit every packet every time the timer is fired! You must keep track of which packets need to be retransmitted when.

Lab 1

To get started on Lab 1, you should log in to an ITSS Unix machine and obtain the assignment software via `git`, as described below. (You can alternatively download and untar the assignment package from this link, then copy reliable.c-dist to reliable.c.)

The functions you need to implement are all in the file `reliable.c`, so that is the only file you need to modify for the assignment.

You should be able to run the command `make` to build the reliable program. When you are done with Lab 1, two instances of reliable should be able to talk to one another. An example of the working program is given here (with what you type in green).

On machine myth15, run:

```
myth15:~/test/reliable> ./reliable 6666 myth14:5555
[listening on UDP port 6666]
Hello I am typing this on myth14.
```

On machine myth14, run:

```
myth14:~/test/reliable> ./reliable 5555 myth15:6666
[listening on UDP port 5555]
Hello I am typing this on myth14.
```

Now anything typed on myth14 will show up on myth15 and vice versa.

For debugging purposes, you may find it useful to run `./reliable` with the `-d` command-line option. This option will print all the packets your implementation sends and receives.

For testing purposes, you may wish to test your code against our reference implementation of the source code. The reference is an x86_64 linux binary, included as a program called `reference` in the assignment package. If you wish to test on other architectures, you will need to build the reference implementation and tester from source code. Because the reference implementation is a solution to the assignment, we implemented it in Haskell rather than C. Moreover, the `reference` does not support client/server mode. To build `reference`, you will need a recent version of the Haskell Platform on your machine. See the file README in our source distribution for other steps.

Lab 2

For Lab 2, you will extend your solution to Lab 1 to support two additional features:

1. A sliding send and receive window larger than one packet, and
2. Connection demultiplexing.

The first feature is relatively straight-forward. When you run the reliable program with the `-w` argument, it should set the sender and receiver window sizes to be whatever the supplied argument is. For example, the following command should select a window size of 5:

```
myth15:~/test/reliable> ./reliable -w 5 6666 myth14:5555
[listening on UDP port 6666]
```

The value specified for the `-w` argument is stored in the `window` field of the `config_common` data structure. You should access it as `cc->window` in the `rel_create` function, and store the value somewhere in the `reliable_state` structure so you have access to it in other functions.

Connection demultiplexing is used when running the reliable program in server mode, which is selected by the `-s` switch. For example, the following command:runs reliable in server mode:

```
myth15:~/test/reliable> ./reliable -s -w 5 1111 localhost:2222
[listening on UDP port 1111]
```

Unlike single-connection mode, which you've been using up until this point, in server mode the argument `localhost:2222` specifies a **TCP**, rather than UDP port. At this point reliable may accept multiple connections from different clients on different client UDP ports, all sending packets to port 1111 on the server. The reliable program will get all of these packets, but since they are all destined to the same UDP port, the rlib code doesn't know which connection they belong to. Therefore, received packets will be passed to the function `rel_demux`.

In server mode, the library never calls `rel_recvpkt`. Instead, you must look up the `rel_t` structure for a packet based on the client's UDP `sockaddr_storage`. You will find the `addreq` function that compares two `sockaddr_storage` structures for equality useful here.

In server mode, reliable input and output no longer come from standard input and output. Instead, for each new connection set up, the library creates a TCP connection to the TCP port specified (`localhost:2222` in the example above). There is a utility `uc` that came with the `reliable.tar.gz` bundle that allows you to listen to a particular TCP port, so that you can test your library. Just run, e.g., `./uc -l 2222` to listen for one connection on a particular TCP port. (You'll have to run it again in a different terminal if you want to accept more than one connection.)

There is also a client mode, selected by `-c`. You shouldn't need any special support in your software for client mode, as long as you are using the `rel_t` structure correctly. Client mode allows you to accept TCP connections and relay them to a reliable server on a particular UDP port. For instance:

```
myth15:~/test/reliable> ./reliable -c -w 5 3333 localhost:4444
[listening on TCP port 3333]
```

The above command accept connections on TCP port 3333, and for each connection, allocates a new UDP port and uses that port to talk to a reliable server listening on port 4444. The `uc` command without the `-l` flag allows you to connect to a TCP port. For instance, to test the above, run `./uc localhost 3333`.

Getting Started

To get started you have to log into a machine managed by Stanford ITSS as described on the ITSS webpage.You may login remotely or use one of the computer labs, for example, the myth machines located in Gates B08. We will test your code on the myth cluster, and the instructions given here assume this environment. To log in remotely from Unix/Linux, use the `ssh` command. For windows, you may need to download an SSH client such as Putty. Also you can develop remotely using VNC Server/Viewer. To use VNC just download and install VNC viewer. When you have done this run the command `vncserver` on a myth machine (after logging in via SSH) and then use the new 'X' desktop it returns to connect to the machine via the VNC viewer on your local machine.

The best way to download the assignment source code is to use git, by executing the following command:

```
git clone http://www.scs.stanford.edu/10au-cs144/repos/reliable
```

Git is a powerful version control system that will make it easy for you to checkpoint your work and later browse your history to track down problems if you have introduced a bug. Using `git` will also make it easy for you to update your source tree should the course staff need to make corrections to the lab assignment. While use of `git` is not required for this class, if you invest the time to learn `git` now, you will likely benefit far into the future. The Git User's Manual is a good resource for learning git.

If we update the assignemt in any way, you can update your source tree to merge in our changes as follows. First, you need to commit a record of any changes you have made, to ensure they are not lost. Then you need to merge our changes. Do these two operations by running the following commands:

```
git commit -a -m "Save work before merging updated assignment"
git pull
```

Testing

Your program must interoperate with the reference implementation, `reference`, included in the assignment package. There is also a tester program called `tester`, which is the same program we will use to grade your assignment. As discussed above, source code for both is available but requires the Haskell Platform to build.

Run the tester giving it your `./reliable` program as an argument. By default the tester program will run all tests, not use server mode, and set a window size of one. The following options may be useful to you:

- `-s` tests server mode.
- `-w` *N* sets the window size to *N* (also passing the `-w` option to reliable).
- `-v` shows the stderr output of your reliable program. You will almost certainly want to use this option when doing any debugging. It is only disabled by default so you can get a clean summary of how you are doing on all of the tests.
- `-T` *N* runs test number *N* instead of running all of them. Useful for debugging one particular test.
- `--gdb` With this option, every time the tester spawns a copy of your reliable program, it prints the process ID and waits for you to press return. This is useful if you want to attach to the process in the gdb debugger (using the command "attach PID").

Submitting

To submit the assignment, you must do two things:

- Run the command `make submit`, this should create a file called reliable.tar.gz.
- Submit reliable.tar.gz to this web page.

Collaboration policy

You should direct most questions to Piazzza, but should not post source code there.

You must write all the code you hand in for the programming assignments, except for code that we give you as part of the assignment and system library code. You are not allowed to show your code to anyone else in the class or look at anyone else's solution. You also must not look at solutions from previous years. You may discuss the assignments with other students, but do not copy each others' code.