

RDT: Reliable data transport protocol on an unreliable network

The goal of this assignment is to develop a reliable data transfer protocol, RDT, for reliably transferring data over an underlying unreliable channel. RDT is somewhat similar to TCP, but it doesn't implement flow and congestion control. The unreliable transmission channel can drop packets, but it can't corrupt them. RDT assumes that both the sender and receiver have sufficient space to store all out-of-order packets, i.e., there is no restriction on the window size. RDT uses a 63-bit sequence number (one bit is used for a validity check). The starting sequence number must be zero. The sequence number is for every byte, as in TCP. The wrapping of sequence numbers is not handled, as 63 bits are sufficient for a very large amount of data transfer. The acknowledgments are sent in a cumulative manner, similar to TCP.

1 Implementation

We have already provided you with a skeleton code. You can use the existing APIs, and you will need to implement some APIs.

Below are the API you can use in the sender and receiver.

1.1 Sender

- `udt send(pkt, len)`: Send a packet, `pkt`, of length `len` to the unreliable channel.
- `make_pkt(buf, seqno)`: Embed 63-bit sequence number, `seqno`, in the buffer, `buf`. The size of `buf` remains the same.
- `get seqno(pkt)`: Fetch sequence number from the packet, `pkt`.
- `start_timer()`: Start the timer.
- `stop_timer()`: Stop the timer.

1.2 Receiver

- `get seqno(pkt)`: Takes the address of a packet, `pkt`, as input. Fetch sequence number from the `pkt`.

- `get_data(pkt)`: Takes the address of a packet, `pkt`, as input. Returns a pointer to the data computed from `pkt`.

- `send ack(ackno)`: Send an acknowledgment number, `ackno`, to the sender.
- `notify app()`: Notifies the application that some data is available for the application's consumption.

You are to implement the following APIs.

1.3 Send

- `rdt send(buf, len)`: Transmits the input buffer, `buf`, of length `len` bytes to the transmission channel. `len` must be greater than `PACKET HEADER LEN`, which is already ensured by the caller. Use `make_pkt` to add a per-byte sequence number. `make_pkt` doesn't change the length of the buffer. It embeds the sequence number in the `buf`. The initial sequence number must be zero. Start the timer if necessary.
- `rdt recv base(ackno)`: Called when the sender receives an acknowledgment number, `ackno`. `ackno` must be a cumulative acknowledgment, as in the case of TCP. Update `send_base` and start or stop timer, as needed.
- `timeout()`: If needed, retransmit the segment at `send_base`, as TCP does. Don't transmit more than one segment.

1.4 Receive

- `rdt recv(pkt, len)`: Called when the receiver receives a packet, `pkt`. Use `get_seqno` and `get_data` to fetch the sequence number and a pointer to the data field from `pkt`. The length of data is `len - PACKET HEADER LEN`. Call `notify app` if the app can consume some data at this point. Send a cumulative acknowledgment similar to TCP using `send ack`.
- `app recv(buf, len)`: App requested data of length up to `len` bytes. If `len` bytes are available, copy them to `buf`; otherwise, copy `x` bytes, where `x` is the length of data that is available for application use. The return value is the number of bytes copied to `buf`. After delivering the data to the application, the receiver can throw away the data. For subsequent calls to `app recv`, the data after the last call will be copied to the `buf`.

2 Environment

For this assignment, you need to clone the project repository. To clone, run:

```
git clone https://github.com/Systems-IIITD/rdt
```

To build the tools, run make in the rdt folder. It will create two applications: sender and receiver.

Run sender using: ./sender
Run receiver using: ./receiver

Ensure that you run the sender before the receiver. You need to run both sender and receiver on the same machine.

You need to modify two files: sender helper.c and receiver helper.c. You are not allowed to change any other files for the final submission. You can find the prototype of all relevant functions in the server.h and client.h files.

You are not allowed to use custom APIs to send and receive data. Use udt send and send ack APIs for data transmission. You can't use network related system calls, such as select, poll, epoll, socket, bind, listen, send, recv, sendto, recvfrom, connect, accept, etc., in your implementation of client helper.c and server helper.c.

The default test case, i.e., the test routine in server.c, would be slightly harder to debug initially. Instead, you can write and debug with your own simple test cases first, before making the given test case work. In your test case, use the create pkt API to create a packet. create pkt creates a packet of size pkt sz. pkt sz must be greater than PACKET HEADER LEN. The seqno argument in create pkt is one more than the expected sequence number of the first data byte in the packet. To embed the sequence number in the packet, you must use the make pkt API. You can read the sequence number of the first data byte in the packet using get seqno. After sending a bunch of packets, the sender poll API waits for an acknowledgment. sender poll also handles the timeout.

3 Test cases and their weightage

There are two test cases. By default, the sender and receiver run the first test case. You can run the second test case using: ./sender 1

The receiver doesn't take additional parameters, and is invoked in the same way for both test cases. The test cases must finish within 60 seconds. Both test cases are of 5 marks. If the test is successful, the sender prints "test PASSED" after a bunch of checkpoint messages. For a failed test case, you'll receive zero. If a test case takes between 60 seconds and 120 seconds, you'll receive half marks if the test case passes. If it takes more than 120 seconds, you'll receive zero, even if the test case passes.

4 How to submit.

Don't leave any print statements in your code. You will receive zero if your implementation prints any additional information.

Create a design document in the PDF format and answer the following questions in your design documentation.

1. Does test case 1 pass? Write the output of the sender for test case
1. 2. Does test case 2 pass? Write the output of the sender for test case 2.
3. What is the runtime of test case 1?
4. What is the runtime of test case 2?
5. Invoke the sender as /usr/bin/time -v ./sender 1 and the receiver as /usr/bin/time -v ./receiver. What is the value of Elapsed (wall clock) time and the Maximum resident set size at both sender and receiver?
6. Invoke the sender as /usr/bin/time -v ./sender and the receiver as /usr/bin/time -v ./receiver. What is the value of Elapsed (wall clock) time and the Maximum resident set size at both sender and receiver?
7. Write the names and roll numbers of all group members.

Submit sender helper.c, receiver helper.c, and your design documentation. Use the naming convention for the assignments and homework. Your assignment will not be evaluated if you don't follow the submission guidelines.

