Programming Assignment #3: Reliable Data Transfer

Logistics

- The due date is 10:30am on **Thu, May. 10th**.

Overview

We learned in Chapter 3 that TCP has the following four key features:

1) segment structure
2) connection management
3) reliable data transfer
4) flow control

In PA #2, you have already dealt with 1) segment structure and 2) connection management: implemented necessary data structures to handle TCP segments and 3-way handshake. You have implemented connection establishment and teardown mechanisms of TCP in PA #2. In PA #3, you will implement 3) reliable data transfer and 4) flow control, first over a reliable channel and then over an unreliable channel. The testing script is in `testtransfer.cpp`.

Note: *You will not implement congestion control. That is, you will not use the congestion window size in your implementation.* In this assignment, the sender-side window size (i.e., the maximum allowed number of bytes which are sent but unacked) is simply set to the advertised rwnd.

3-1. Reliable Data Transfer over Reliable Channel

The main task of this assignment is to implement **read()** and **write()** in KENSv3, assuming that **the underlying channel is reliable** (i.e., there is **no bit corruption nor packet loss**).

```
case READ:
    // this->syscall_read(syscallUUID, pid, param.param1_int,
param.param2_ptr,
    // param.param3_int);
    break;
```

The **read()** call receives three parameters from the application layer: a socket (file) descriptor, a pointer to the application's read buffer, and the number of bytes to read from the socket. It should return the number of bytes read. Details about the **read()** call are available here: https://man7.org/linux/man-pages/man2/read.2.html

```
case WRITE:
    // this->syscall_write(syscallUUID, pid, param.param1_int,
param.param2_ptr,
    // param.param3_int);
    break;
```

The **write()** call receives three parameters from the application layer: a socket (file) descriptor, a pointer to the application's data to write, the number of bytes to write to the socket. It should return the number of bytes written. Details about the write() call are available here: https://man7.org/linux/man-pages/man2/write.2.html

When an application calls **read()**, the TCP layer is in one of the following two situations:

(1) If there is already received data in the corresponding TCP socket's receive buffer, the data is copied to the application's buffer and the call returns immediately.
(2) If there is no received data, the call blocks until any data is received from the sender. When data arrives, the data is copied to the application's buffer and the call returns.

When an application calls **write()**, the TCP layer is in one of the following two situations:

1) If there is enough space in the corresponding TCP socket's send buffer for the data, the data is copied to the send buffer. Then,
    a) if the data is sendable (i.e., the data lies in the sender's window), send the data and the call returns.
    b) if the data is not sendable (i.e., the data lies outside the sender's window), the call just returns.
2) If there is not enough space, the call blocks until the TCP layer receives ACK(s) and releases sufficient space for the data. When sufficient space for the given (from application) data becomes available, the data is copied to the send buffer and then,
    a) if the data is sendable (i.e., the data lies in the sender-side window), send the data and the call returns.
    b) if the data is not sendable (i.e., the data lies outside the sender-side window), the call just returns.

When a data packet arrives (to the TCP layer), you should:

- copy the payload to the corresponding TCP socket's receive buffer
- acknowledge received packet (i.e., send an ACK packet)

When an ACK packet arrives (to the TCP layer), you should:

- free the send buffer space allocated for acked data
- move the sender window (the number of in-flight bytes should be decreased)
- adjust the sender window size (from advertised receive buffer size)
- send data if there is waiting data in the send buffer and if the data is sendable (i.e., there is room in sender's window)

For the sake of convenience, set the MTU size to 1500. In TCP, it translates to MSS of 1460. The minimum buffer size should be at least 2MB.

Note: The description above is based on the assumption that you allocate and use fixed-length send buffer and receive buffer for each socket. You may use packet queues or dynamically allocated memory. These implementation details are up to you.

3-2. Reliable Data Transfer over Unreliable Channel

Now you will extend your implementation of **read()** and **write()** to work when **the underlying channel is unreliable** (i.e., **bit corruption** or **packet loss** may **occur**).
You will encounter the same test cases as in 3-1. However, the KENSv3 framework will randomly drop or corrupt some packets. Use `unreliable` versions of binaries (e.g. `kens-part3-unreliable`) to test over unreliable channels. You have to pass unreliable versions of parts 1, 2, and 3 to get full credit for PA #3.

What should you do more when the channel becomes unreliable?

In order to address bit corruption, you have to validate the checksum at the receiver. If a received packet is corrupted, simply discard (drop) it. KENSv3 provides a utility function that computes checksum. Take a look at
https://github.com/ANLAB-KAIST/KENSv3/wiki/Tip:-Networking-Utilities

In order to address packet loss, you need timers and implement retransmission at the sender. In computation of RTT, you need the system clock. Call HostModule::getCurrentTime() and get the current KENS system clock time.

For timers, use the following RTT estimation:

$$\text{EstimatedRTT} = (1 - \alpha)*\text{EstimatedRTT} + \alpha*\text{SampleRTT}$$

$$\text{DevRTT} = (1 - \beta)*\text{DevRTT} + \beta*|\text{SampleRTT} - \text{EstimatedRTT}|$$

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4*\text{DevRTT}$$

$$\alpha = 0.125, \ \beta = 0.25$$

If an expected ACK packet does not arrive on time (i.e., the timer expires) at the sender, you should retransmit the corresponding data packet (whose timer is expired).

Note that the sender should remember every sent packet for the case of retransmission. Also, the receiver should be aware of (possible) duplicate packets (i.e., the same packets may arrive more than once). Therefore, the receiver should be able to determine which packet to ignore and not.

Start with the initial RTT of 100ms.

More Resources

https://github.com/ANLAB-KAIST/KENSv3/wiki/Misc:-External-Resources

Tips

- Managing File Descriptors
- Returning a System Call
- Sending and Receiving a Packet
- Figuring out the Source IP Address to Use
- Using a Timer
- Networking Utilities
- Using pcap files
- Debugging Memory Bugs (Only for macOS Linux)

More Resources

https://github.com/ANLAB-KAIST/KENSv3/wiki/Misc:-External-Resources

Submission

- The test code is in the following files. Your code will be graded according to the test results from them
    - ./app/kens/testtransfer.cpp (`test-kens-transfer`)
    - ./app/kens/testbind.cpp (`test-kens-bind-unreliable`)
    - ./app/kens/testhandshake.cpp (`test-kens-handshake-unreliable`)
    - ./app/kens/testclose.cpp (`test-kens-close-unreliable`)
    - ./app/kens/testtransfer.cpp (`test-kens-transfer-unreliable`)
- You should submit only three files: **readme.txt**, **TCPAssignment.cpp**, **TCPAssignment.hpp**. TCPAssignment.cpp and TCPAssignment.hpp should contain your implementation.
- Upload the files on KLMS.
- There is no designated template for readme.txt; just briefly describe how you have progressed to complete this assignment. It does not have to be long and detailed. A brief summary will suffice.