TCP/IP Project README

Design

The TCP layer is placed under the 'transport' package in the code. This layer is designed to placed on top of our existing 'network' layer. The communication between the two layers is done over Go channels. A new TCP handler is registered with the Network layer as the handler to process the TCP protocol. When TCP packets are received, the network layer delicates the packet to the handler. The handler in turn finds the correct socket to handle the packet and passes the packet along. When sending data outwards, all sockets will encapsulate their packets and write to the IP channel.

Core Components

- SocketManager
  - The SocketManager is a global singleton that manages and perform C-style socket operations on individual TcpSocket objects. It exposes command such as V_socket, V_bind, V_listen etc. Commands are invoked with the Socket FD as primary argument, just like how it is done in C. This is not necessary since we could have alternatively adopt a more object-oriented style with Go, but ultimated decided to folllow the example from the project handout.
  - The primary responsibility of the SocketManager is to locate Socket instances either via Socket FD or their addresses. It tracks all sockets created in the system in two different maps, one keyed by Socket FDs while the other one is keyed using addresses (a pair of Address:port denoting local and remote addresses). The SocketManager essentially does the bookkeeping of sockets for the application and delegates socket API calls to each individual sockets.
- TcpSocket
  - The TcpSocket structure is the basic encapsulation for the notion of TCB. It stores all related information for a given TCP socket.
  - The TcpSocket itself is to handle sending and receiving of data segment. It does so with a separated sliding window for sending and receiving data. The TcpSocket uses the sending and receiving window to determine how much data to send and receive respectively
  - Also, two separate threads are running to process incoming and outgoing data. This allows us to easily shutdown one or both of the reading and writing interface of the socket.
- SenderSlidingWindow
  - The sender sliding window encapsulates all logic and related states regarding transmitting data outward to the receiving socket.
  - When upper layer decides to send data via the socket that the sliding window is attached to, it caller will first attempt to write the outing data into the buffer of the sliding window. The buffer is represented by two arrays, a byte array and a boolean array. The byte array is where the caller writes data into and also where the sending window reads data from. The boolean array tracks whether each slot in the byte array is dirty, in another word, whether it contains data regardless of what is actually stored in that slot in the byte array. This is used since the byte array does not allow storage of nil values.
  - The sliding window closely follows the RFC in terms of calculating its available window size and deciding how much data can be stored into the buffer at any given point. Once the data is successfully written into the buffer, the

socket thread will recognize un-transfered data and starts sending data to the receiving socket. This is done continuously until the caller decides to close the socket.

  - After data has been sent out, a new message is put onto a priority queue ordered by the expiration time. This queue tracks all message in flights, in another word, all packets that we have not received a ACK for. The sending thread will continously monitor this queue and examine expired messages. If an expired message has a smaller expected ACK (its Sequence number plus its payload) than the largest ACK we have received so far, it implies that the receiver has already received this packet and even packets after. Therefore, we simply discard this message and look at the next one in the queue. If an expired message from the queue has a larger expected ACK than the one been received, the expiration time is extended and the message is put back into the queue after a retransmission of the exact same message takes place. This mechanism handles loss and re-order of packets over a lossy node.

- ReceiverSlidingWindow
  - The receiver sliding window closely mirrors the sender sliding window as it uses two arrays to represent its buffer along a few pointers to track where to write and read the data.
  - When data segments arrive at the socket, the socket will attempt to write the content into the receiving buffer only if the content is within the receiving window and can fit into the remaining space. The receiver window then handles the data by placing the bytes into the buffer and updates its pointers accordingly.
  - For handling out-of-order segments, the receiving window calculate where the incoming segment should be placed onto the buffer and writes the data into the buffer accordingly. As a result, the buffer is not filled up sequentially but rather randomly into the buffer over a lossy link. The behavior of out-of-order receiption close follows the RFC as the window will return the next ACK that it is expecting.
  - Also for a lossy environment, the receiving window expects the incoming ACK and ensures that all data has been received before the connection is closed.
- TcpStateMachine
  - The TCP statemachine is implemented as a generalized finite state machine. TCP states and transitions are defined as structures and registered into the statemachine according to the TCP state diagram. The statemachine starts with the initial state of CLOSED, and awaits transition events to be trigger a transition into one of its next states. When a packet or socket command is received, the code asks whether there is a corresponding transition for the said event. If a transition is found, a response object can be obtained from the statemachine, with detailed information regarding what should the resonding action be, so that the proper reply to the event can be triggered. Once the response has been handled, the transition can simply made by calling the #transit method on the statemachine.
  - The statemachine closely follows what was given in the project specifications; however, certain transition has been augmented since the ACK flags is allowed to be on even if not specified by the diagram.
- File Sender & Receiver
  - The file sender and receiver manages file transfer logic by operating on file objects and calling socket level APIs. File Senders and Receivers spawns on their new threads so that the driver can continue to accept commands during the transfer process.
  - The file sender first locates the file to be sent, read the file in small chunks and transfer them to the other end via a newly connected TCP socket. It iteratively puts data read from the file into the socket's sending buffer until all of the file content has been accepted into the sockets. Then it waits until all data has been successfully transferred before closing the soket.

- The file receiver starts by accepting a new connection from a file sender. Once the TCP connection has been established, it iteratively reads received file data from its TCP socket and write them into the receiving file until the corresponding sender signals completion of transfer by closing the sending socket. At this point the file receiver cleans up its own sockets and resources and terminates.

Synchornization

Synchornization is mostly done by using ReadWrite locks where appropriate. For global singletons such as the SocketManager, its internal tracking of sockets are protected by such locks. Also for sending and reciving windows, any buffer and pointer operations are protected by similar locks to eliminate any incorrect intermediate states.

Performance and Known Issues

Although we attempted various optimizations for both sending and receiving data, our performance has been subpar in comparision to the reference node. When used against the reference node, our best performance has been:

- ~1 second to send mediumfile.dat, ~3 seconds to receive mediumfile.dat
- ~14 seconds to send bigfile.dat, ~22 seoncds to receive bigfile.dat
- ~50 seconds to send bigfile.dat over a lossy link with 2% drop rate, ~100 seoncds to receive bigfile.dat over a lossy link with 2% drop rate. With two our of implementations, the non-lossy situation would have the results being slight higher. Over a lossy node, results are actually better since our node does not implement congestion control.

We have tried a few methods to speed up the code. Namely we tried to leverage multi-threading as much as allowed withouth compromising the integrity of the state of the application. We tried placing locks only at absolutely necessary places instead of locking a large chunk of code. We have also removed a few data structures that forces O(n) time pointer adjustments in favors for the new ones that allows us to adjust pointers in O(1) time.

We noticed that most of our performance difference occurs when receiving data, and after some mesurement we have noticed that a lot time is wasted on some of the internal mechanism of Go and our IP layer.

First we observed that handling of a TCP packet and puting its payload into the receiving buffer merely occupies 5000~10000 nanoseconds. With this number we should be able to achieve similar performance as the reference node.

We then looked through our prior implementations of IP and Link layer and identified a few likely bottleneck. We have found a few places where we used the #select mechanism of Go with channels. And after digging around on the internet, we have found discussion on the performance hiccups it can incur. We also spotted a few places where Go interface{} are used non-optimally. This cause the code to interpret arguments and does type conversion on the fly which can seriously affect the performance of our code.

Overall, not knowing some of the internal details of Go has affected our performance. However, we have also observed that different CS lab machines can greatly affect the running time also. We have tested sending and receiving files against the reference node, using exactly the same code and commit, and resulted in running times that are more than 10 seconds apart.