CS434/534 Programming Assignment 2: Network Transport

This document serves as the report for this assignment.

Author: Ruichun Ma (ruichun.ma@yale.edu)

File Descriptions

```
|-- src
|    |--node.rs: The executable program. Run this to start a transport
node.
|    |--lib.rs: The lib for all modules needed for my transport protocol
implementation.
|    |--core: The folder holds all modules of the lib.
|         |--socket.rs: The non-blocking socket API used by node.rs.
|         |--manager.rs: The manager struct to govern all the sockets, the
timer and udp sender and receiver.
|         |--packet.rs: The transport packet format and some helper
functions.
|         |--timer.rs: The timer struct for setting up timeout.
|         |--udp_utils.rs: The underlaying udp sender and receiver.
|-- target
|    |--debug: The compiled program. This one only wroks for macos x86_64.
|    |--logs: The logs for each socket, recording packets sent or recv.
```

To Run

There are threes exec modes for `./target/debug/node_bin`

```
// this will compile the code (if you installed rust)
cargo build

// run this to see all the arguments available
./target/debug/node_bin --help

// basic functionality test running on 127.0.0.1
./target/debug/node_bin --ex local_test

// A test for flow control. A fast sender and a slow receiver. Also
running with localhost.
./target/debug/node_bin --ex window_test

// Transfer 40KB testing data to the server side. The server program needs
to be running first on another machine.
./target/debug/node_bin --ex transfer --da 'dest ip addr' --la 'local ip
addr'
```

```
// set up a server socket listener, which with continously accept new
connections.
./target/debug/node_bin --ex server --la 'local ip addr'
```

To simulate some packet loss, go to `./src/core/udp_utils.rs` and change `SIM_LOSS_RATE`. This defines the probability of a packet loss.

Does it wroks?

For local_test and window_test, the program runs with no error even when packet loss rate equals 5%.

When tested on two real machines, one runs the client program and one runs the server program. I used two distant cloud machines, which have a 100+ ms delay. Two concurrent connections are created and 40KB data are sent for each connection. The test completes successfully if no simulated packet loss.

Part 1

High Level Design

From the perspective of multi-thread programming, I have four threads running when I start the node. All the threads send messages to each other through channels/pipes.

First, we have *the user application thread*, i.e. the main thread, created by the `node.rs`. Inside this thread, it will bring up *the manager thread* in `manager.rs`, which will provide transport service. The `socket.rs` is only an API and does not do any work. When the user thread calls the socket API, it only sends a task message to the manager. The manager holds a task queue, which is the most important component of it. All it does during execution is to process next task in the queue.

Second, when the manager thread starts, it will also bring up *the timer thread* and *the udp-utils thread*. The manager can tell the timer thread to set up or cancel timeout timer. The timer will put a task into the queue when timeout happens. The udp thread does two things. It unpacks packets and hands them over to mananger thread when new datagrams arrived using UDP. It also packs packets in its queue and send them to remote host using UDP.

Disscussions of design

1. Diss1a: Your transport protocol implementation picks an initial sequence number when establishing a new connection. This might be 1, or it could be a random value. Which is better, and why?

   I chose a random number as the initial sequence number. It seems better than 1 because it makes sure that the ack from the server side is fresh.

2. Diss1b: Our connection setup protocol is vulnerable to the following attack. The attacker sends a large number of connection request (SYN) packets to a particular node, but never sends any data. (This is called a SYN flood.) What happens to your implementation if it were attacked in this way? How might you have designed the initial handshake protocol (or the protocol implementation) differently to be more robust to this attack?

If all the SYNs are from the same port, one connection will be established and that is all. All the following SYN will be ignored. If SYN are from different ports, only limited number of new connections will be established, because the number of backlog is limited. To prevent the attacker from using up the backlog, we can limit the number of SYN accepted from the same ip addr at a certain period of time, assuming that the attacker can not easily get a large number of ip addresses.

3. Diss1c: What happens in your implementation when a sender transfers data but never closes a connection? (This is called a FIN attack.) How might we design the protocol differently to better handle this case?

   Given my current design, this will leave a connection open infinitely. This is undesriable. We could potentially use time out to regulate the max duration of an idle connection.

Implementation Details

1. `The socket manager` struct mantains a task queue to hold all the tasks to handle. I reused the inter-thread communication channel/pipe as the queue. Since the message in the channel is also FIFO. Rust enum can hold various types of values, so I use this to hold all the task arguments for socket APIs and other tasks.

```
// Task Message from socket api and timer sent to socket manager
#[derive(Clone)]
pub enum TaskMsg {
    // (enum also can hold args)
    // ==== for socket API
    /// (local_addr)
    New(String),
    /// (sock_id, local_port)
    Bind(SocketID, u8),
    /// (sock_id, backlog)
    Listen(SocketID, u32),
    Accept(SocketID),
    /// (sock_id, dest_addr, dest_port)
    Connect(SocketID, String, u8),
    /// (sock_id, buf, pos, len)
    Write(SocketID, Vec<u8>, u32, u32),
    /// (sock_id, len)
    Read(SocketID, u32),
    /// (sock_id)
    Close(SocketID),
    Release(SocketID),

    // ==== UDP packets related
    // a new packet is received
    OnReceive(TransportPacket),
    /// schedule a sending task
    /// (sock_id, trans_type, seq_start, len, retrans_flag)
    SendNow(SocketID,TransType, u32, u32, bool),
}
```

2. `The socket manager` also matains a hashmap containing pairs of `SocketID` and `SocketContens`. `SocketID` contains a four-element tuple, (local_addr, local_port, remote_addr, remote_port). The `SocketContents` holds all the buffers and sliding windows. As described above, the socket API does not have any info and does not do any work. And all APIs are non-blocking.

3. The sending and recving buffers are implemented suing `VecDeque<u8>`, which is a ring buffer of bytes. Several 32-bit values are used to define the sliding window. For sending window, send_base, send_next, send_wind are used. For recv window, recv_base, recv_next, recv_wind are used.

```
// ==== sliding window pos ====
/// the start of sliding window, the first byte that is not acked
send_base: u32,
/// the start of range to be filled with new data
send_next: u32,
/// send window size
send_wind: u32,

/// the start of bytes to be read by user
recv_base: u32,
// the start of bytes to be filled
recv_next: u32,
/// recv window size
recv_wind: u32,
```

4. I use a special timer thread to schedule the retransmission task. The socket manager can send command to the timer thread to set up or tear down a timeout timer. The socket manager will also provide a callback, which is a task for the manager. When the timer is triggered, the timer thread will put the callback into the manager's task queue to perform retransmission.

```
type TimeoutCallback = TaskMsg;
pub enum TimerCmd {
    New(time::Instant, TimeoutCallback),
    Cancel(TimerToken),

}
struct TimerEntry {
    time_lim: time::Instant,
    callback: TimeoutCallback,
}
```

5. For each socket, there are seven states.

```
pub enum SocketState {
    CLOSED,
    LISTEN,
    SYN_SENT,
    ESTABLISHED,
```

```
    /// We need to close, but still work to do.
    /// When the receiver reveived FIN, but has still data not read by
the user.
    SHUTDOWN,
    /// wait for all the packets to be acked, so that we can send FIN
    FIN_WAIT,
    /// a FIN has been sent
    FIN_SENT,
}
```

Test Outputs

Here shows the test output of `lcoal_test`. In this test, one client and one server socket are created. They communate over UDP on localhost. Frist, the client sends 100 packets to the server. The sever will read the data right after one packet is sent. Then, the client sends 20 packets to fill the server buffer. And the sever reads 20 packets. This uses the buffer but does not involve flow control. The packt loss rate is 5% **Client Outputs**

```
S:.:.:..!:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:
.:.:.:.:.:.:.:..!:.:.:.:.:.:.:.:..!:.:.:..!:.:.:.:.:..!:.:.:..!:.:.:.:.:
..!:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:..!:.:.:.:..!:.:.:.:.:.:..!:.:
....:::..::.:.:..::.::...::..:.:.:.:.:::?F:
```

Server Outputs

```
.....................................................!...!.....!...
...............................................f
```

My notion is slight different from the assignment document. In my output, `.` is a normal data packet, `.!` is a retransmited data packet.

As the outputs show, all the data packet will get an ACK, and retransmit if it is lost. The received data is examined at the server side.

Part 2 : Flow Control and Congestion Control

Disscussions of design

1. Diss2: Your transport protocol implementation picks the size of a buffer for received data that is used as part of flow control. How large should this buffer be, and why?

   The buffer should be at least larger than the max size of one packet. So the packet delivery can move forword. Ideally, the larger the buffer is, the more efficient the transport would be. Because a large buffer can endure bursts of received packets without telling the sender to slow down.

Details fo design

Several varibales inside the socket content are used for flow and congestion control.

```
/// flow control, recv window left
send_flow_ctl: usize,
/// conjestion control, num of bytes
send_cong_ctrl: usize,
dup_ack_record: u32,
dup_ack_num: u32,
```

`send_flow_ctl` will be updated when new ACK arrives. And `send_cong_ctrl` will be updated accroding to AIMD. When three duplicate ACKs are recorded, it will be cut to half. And it will additively increase when ACK arrives normally.

There are some cases to consider when we use flow control. If the server's buffer is full and the sender stops sending packets, how to restart to transportation? To solve this issue, the server socket will send a reduntant ACK carrying new window size when the full recv buffer has been read and gets room again. But this updated ACK may also get lost. So I also allow the client to ignore the flow control window and send a data packet to get a ACK from the server. This will only happens when client is limited by the window after many timeouts.

Test Outputs

Here shows the results of `./target/debug/node_bin --ex window_test`. In this test, the client is sending packets in a much faster speed than the server read the data. So the flow control will work. And considering simulated packet loss, congestion control will also work. **Client Outputs**

```
S:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:?.?.?..:?.:?.:.??.!:?....:??.?
?...!..:???.??..:?.?.??.!:.?.??.!:.!?.??..!.:??..!?.?.!:.?....:???.?.?.?.?
?.:...?.:.?.??....!.:???..?.!?.?.!:.?.?.?.??...!.........!..!:???????????.
?.?.?.?.?.!.!:??.?..?.??.....!:?????f
```

Server Outputs

```
.............................!!!..!!!!..!.........!!.....!!!............!
!!!.........!!!!!!!!!!.....!!.....!!!!F:
```

Part 3 : Design Extensions

Your design can be extended in multiple directions to integrate modern features. Please pick 3 of 4 the following directions.

1. Multipath TCP API and protocol: Your current API is the basic socket API. In class, we discussed multi-path TCP. How may you extend your socket to allow multi-path API? Please (1) specify the issues of existing API to support multipath TCP, (2) list the new API, clearly marking modifications and/or additions to the API, (3) give an example of client and server programs using your multi-path TCP API, and (4)

briefly describe how your implementation (protocol format, server, client) will be changed to support multipath TCP. For this design exercise, you can assume bi-directional transport.

(1)(2) First, we would need to keep socket interface to provide the same service to the applications that are unaware of multipath tcp or still want to access the socket interface. Then, we need to provide extra APIs for MPTCP-aware applications. Based on RFC6897, we need at least these socket operations for MPTCP.

- TCP_MULTIPATH_ENABLE: This value should only be set before the establishment of a TCP connection. Its value should only be read after the establishment of a connection.
- TCP_MULTIPATH_ADD: This operation can be applied both before connection setup and during a connection. If used before, it controls the local addresses that an MPTCP connection can use. In the latter case, it allows MPTCP to use an additional local address, if there has been a restriction before connection setup.
- TCP_MULTIPATH_REMOVE: This operation can be applied both before connection setup and during a connection. In both cases, it removes an address from the list of local addresses that may be used by subflows.
- TCP_MULTIPATH_SUBFLOWS: This value is read-only and can only be used after connection setup.
- TCP_MULTIPATH_CONNID: This value is read-only and should only be used after connection setup.

```
+------------------------+-----+-----+------------------------------+
| Name                   | Get | Set |           Data type          |
+------------------------+-----+-----+------------------------------+
| TCP_MULTIPATH_ENABLE   |  o  |  o  |           boolean            |
| TCP_MULTIPATH_ADD      |     |  o  |      list of addresses       |
|                        |     |     |         (and ports)          |
| TCP_MULTIPATH_REMOVE   |     |  o  |      list of addresses       |
|                        |     |     |         (and ports)          |
| TCP_MULTIPATH_SUBFLOWS |  o  |     |  list of pairs of addresses  |
|                        |     |     |         (and ports)          |
| TCP_MULTIPATH_CONNID   |  o  |     |           integer            |
+------------------------+-----+-----+------------------------------+
```

To provide the setting above, I need to add the following API.

```
fn set_multipath_enable (&self, mp_enable: bool);
fn add_multipath (&self, path: IpAddr, port: u32);
fn remove_multipath (&self, path: IpAddr, port: u32);
fn get_multipath_flows (&self) -> Vec<SubFlow>; // this returns a list
pairs of addresses
fn get_multipath_connid (&self) -> u32; // returns the unique mptcp id
```

(2) example of using the API

```
let mut server_sock = Socket::new();
server_sock.set_multipath_enable(true); // turn on MPTCP
server_sock.add_multipath(192.168.10.10, 80);
server_sock.add_multipath(192.168.0.10, 80); // add multiple paths
server_sock.listen(args.backlog).expect("Can not listen to port!"); //
set backlog

let mut client_sock = Socket::new();
client_sock.set_multipath_enable(true); // turn on MPTCP
client_sock.add_multipath(192.168.10.11, 80);
client_sock.add_multipath(192.168.0.11, 80); // add multiple paths
client_sock.connect(args.remote_addr, args.remote_port).expect("Can
not establish connection.");

let server_recv = server_sock.accept();

client_sock.write(...);
server_recv.read(...);

// close all
client_sock.close();
server_sock.close();
server_recv.close();
```

(4) I would need to add new data structures to represent different subflows. Inside each socket struct, it will contain multiple subflows. Server and client program would need algorithms to select the optimal flow to deliver data.

2. Secure Transport API and protocol: One direction of modern transport design is the integration (e.g., in QUIC) of basic transport (TCP) and security (e.g., TLS). Please provide a basic, high-level API and protocol design which integrates basic transport and TLS security.

   The main reference for my answer is `The Secure Socket API: TLS as an Operating System Service` in NSDI'18. It seems a good fit for what we are looking for.

   We can build all TLS functionality directly into the POSIX socket API. The `socket()` API accepts a argument to specify the protocol. So we can add special protocol to integrate TLS by specifying the protocol as `IPPROTO_TLS`. Then we can define special behaviors under IPPROTO TLS for all the socket APIs.

   - For `connect()`, it perform a connection using underlying transport protocol (e.g., TCP handshake), and perform the TLS hand-shake (client-side) with the specified remote address.
   - For `accept()`, it gets a connection request from the pending connections, perform the TLS handshake (server-side) with the remote endpoint, and creates a new socket descriptor.
   - For `read()` and `recv()`, it encrypts/decrypts and transmit/receive data to a remote endpoint.
   - For `close()`, it closes a socket, sends a TLS close notify, and tears down connection, if applicable.

An additional configuration file can be used to gain control over TLS parameters. Some options are listed below.

- TLS Version: Select which TLS versions to enable
- Cipher Suites: Select which cipher suites to enable, in order of preference
- Certificate Validation: Select active certificate validation mechanisms and strengthening technologies.
- And so on

3. Congestion Control: Please describe the modification of your code (as concrete as you can) to implement TCP Cubic congestion control. Please describe briefly how your code can be extended to use Google's BBR v1 congestion control.

   To implement TCP Cubic congestion control,

   - Basic reduction, when three duplicate ACKs

     ```
     self.w_max = self.cwnd; // save window size before reduction
     self.ssthresh = max(2, self.cwnd * Self::beta_cubic); // new slow
     start threshhold
     self.cwnd = Self::beta_cubic * self.cwnd; // multiplicative
     decrease
     self.t = 0; // reset time
     ```

   - Fast Convergence, to update w_max

     ```
     if self.w_max < self.w_last_max {
         self.w_last_max = self.w_max; // remember last w max
         self.w_max = self.w_max * (1 + Self::beta_cubic ) / 2; //
     further reduce w_max
     } else {
         self.w_last_max = self.w_max;
     }
     ```

   - Congestion avoidance, after receiving an ACK

     ```
     self.t += self.rtt / self.cwnd; // update time, this is an
     inaccurate estimation
     let K = cubic_root( self.w_max * (1 - Self::beta_cubic) / C );

     w_cubic = C * (self.t - K) ^ 3 + self.w_max; // new cubic window
     w_cubic_rtt = C * ((self.t + self.rtt) - K) ^ 3 + self.w_max;
     self.w_est = self.w_max * Self::beta_cubic +
                  3 * (1-Self::beta_cubic) / (1+Self::beta_cubic) * (t
     / self.rtt);

     // update window
     if self.w_cubic < self.w_est {
     ```

```
    self.cwnd = self.w_est;
} else {
    self.cwnd += (w_cubic_rtt - self.cwnd) / self.cwnd; // update
window size
}
```

To use google BBR v1, we need to find the optimal point with maximal BW and minimal RTT, which will reduce the queue usage. First, we need a startup process to do exponential BW search. Then we need to drain the queue to get a lower RTT. Then we continuously explore max BW, seeking the optimal point.

4. Delivery Flexibilities: Some major networks (e.g., Amazon Scalable Reliable Datagram) propose that the network does not enforce in-order delivery. Please describe how you may design a flexible API and protocol so that the transport can provide flexibilities such as delivery order (segments can be delivered to applications not in order) and reliability semantics/flexibilities (e.g., some packets do not need reliability, and one can use FEC to correct errors instead of using retransmissions).