TCP Project README/Performance Analysis

Author: Shan Lu (slu5), Jingyiping Zhang (jzhang12)

For better reading experience on this document, please click here to open a rich-formatted PDF file.

Data Structures

- TCP

```
type TCP struct {
    connections map[string]*TransControlBlock // key:localAddr.String()+remoteAddr
.String()
    socketMap   map[int]*TransControlBlock
    verbose     bool
    lock        sync.RWMutex
    ran         *rand.Rand
    network     *NetworkLayer
}
```

Above is the global TCP structure, which contains 2 hashmaps mapping socket number and socket address pair to the corresponding `TransControlBlock` , which stores the specific attributes regarding this connection. Specifically, if a socket is built with address:port pair of (192.168.0.2:4323, 192.168.0.4:5000), and this socket is assigned with socket number 3, then both 3 and string "192.168.0.2:4323192.168.0.4:5000" can get access to the corresponding Transmission Control Block within constant time. This "redundant" design is mainly for time efficiency purpose.

Other attributes include a "verbose" option, which allows user to see what happened in this connection. Since map in Go is not thread-safe, we have to lock TCP when we update `connections` and `socketMap` . Finally, `network` is a pointer pointing to the network layer. Whenever a TCP packet has been wrapped, it will be passed through `network` 's `onRecvTCPData` method.

- Transmission Control Block

```
type TransControlBlock struct {
    sockfd          int
    localAddr       *net.TCPAddr
    remoteAddr      *net.TCPAddr
    state           StateTCP
    iotype          SocketIOType
    send_unack      uint32
    send_next       uint32
    send_window     uint16
    recv_next       uint32
    recv_window     uint16
    read_buf        *CircularBuffer
    write_buf       *CircularBuffer
    retransmit      bool
    dropRate        int
    retransmitQueue *Queue
    lock            sync.RWMutex
    rto             time.Duration
    srtt            time.Duration
    lastSent        time.Time
}
```

The `TransControlBlock` structure follows closely with RFC 793's suggestion. It contains 3 parts. First is the socket pair information(e.g. local and remote address and port, file descriptor, and current state in TCP state machine). Second part is regarding read/write buffer and sliding window control variables. The third part is retransmission queue and retransmission timeout calculation variables.

- Circular Buffer

```
type CircularBuffer struct {
    data               []byte
    size               int
    appReadPtr         int
    window_left        int
    window_right       int
    numRead            int64
    numWritten         int64
    initSeqNo          uint32
    lock               sync.Mutex
    lastGetTime        time.Time
    unorderedPacketMap map[uint32][]byte
}
```

`CircularBuffer` class implements a circular buffer supporting the sliding window algorithm and flow control. It provides two standard buffer I/O methods `Put(data []byte, seqNo uint32)` and `Get(numbyte int)`, and another `PutUnordered` method handling those unordered data. Detail will be specified at later section.

- Queue(lane/queue)

Credits to GitHub user **oleiade**. This thread-safe queue helps me implement retransmission queue easily.

Performance Analysis

I used three different sized file to compute transmission speed.

- **test.mp3** 5185067 bytes (4.94MB)
- **test.pdf** 37629571 bytes (35.88MB)
- **test.zip** 92339236 bytes (88.06MB)

For performance issue, I turned off all printing functions except those related to connection creation/teardown.

Case 1: directly connected node, perfect link(dropRate=0%)

Test File	Test #1	Test #2	Test #3
test.mp3	387ms(12.76MB/s)	427ms(11.56MB/s)	529ms(9.33MB/s)
test.pdf	2.69s(13.33MB/s)	3.25s(11.04MB/s)	3.01s(11.92MB/s)
test.zip	6.46s(13.63MB/s)	7.06s(12.47MB/s)	7.25s(12.11MB/s)

All transmitted files have been `diff` ed with original file and are compeletely and correctly transmitted. The average transmission speed on two nodes connected directly to each other with a perfect link is ~12MB/s, which outperforms the minimum requirement of 8MB/s.

How to simulate:

Node A

```
$ ./bin/node A.lnx
node> recvfile output.pdf 6000
```

Node B

```
$ ./bin/node B.lnx
node> sendfile test.pdf 192.168.0.4 6000
```

where `192.168.0.4` should be node A's IP address.

Case 2: A<->B<->C, where B's dropRate=2%

Test File	Test #4	Test #5	Test #6
test.mp3	425ms(11.62MB/s)	656ms(7.53MB/s)	576ms(8.57MB/s)
test.pdf	8.93s(4.01MB/s)	5.89s(6.09MB/s)	6.58s(5.45MB/s)
test.zip	10.42s(8.45MB/s)	17.72s(4.96MB/s)	17.43s(5.05MB/s)

When intermediate link becomes lossy, the TCP gets slow as data and ACK packets are dropped randomly during the transmission. Moreover, the speed among different test cases has a **larger variance**. This instability may due to the randomness of packet dropping. Function `func (network *NetworkLayer) onRecvIpPacket()` in file `lib/network.go` implements this random packet dropping logic.

How to simulate:

Node A

```
$ ./bin/node A.lnx
node> recvfile output.pdf 6000
```

Node B

```
$ ./bin/node B.lnx
node> lossyforward 2
set faulty node forwarding dropRate to 2 percent
node>
```

Node C

```
$ ./bin/node B.lnx
node> sendfile test.pdf 192.168.0.4 6000
```

where `192.168.0.4` should be node A's IP address.