Computer Networks Practical

Champilomatis, Stavros
schampilomatis@gmail.com

Sarris, Nikolaos
sarris.nikos68@gmail.com

June 30, 2015

1 Introduction

In this project we were asked to build a simplified **TCP Implementation** for Android smartphones on top of a provided IP stack and a Chat application on top of the TCP stack of the first part. In brief, our TCP interface implements the Connection Setup phase (standard three-way handshake) providing `connect()` and `accept()` methods, Data Transfer providing `write()` and `read()` methods and Connection Termination providing the `close()` method.

The **Testing Phase** of our code is based on a basic test Suite provided to us and on our own implemented tests. Our tests cover an extended variety of cases, for all the three stages, namely problems that may occur during setting up the connection, transferring data or releasing the connection.

The **Chat Application** is a basic Android Application that uses two windows in order to permit users communicate with themselves. This application uses the TCP stack implemented by us to exchange messages. Both windows are able to send and receive messages.

2 Implementation

2.1 TCP Stack

We do not implement any kind of port management because according to the specification of the assignment we only need one socket per `TCP` stack.

2.1.1 Connection Establishment

As described in the Introduction our implementation provides an `Interface` containing five methods for setting up a connection, data exchange and release a connection.

Firstly, the `connect` method implements the **active** connection setup by sending a `SYN` segment to the server and waiting the relevant `SYNACK` response. When, this latter segment is received `connect` sends an `ACK` back to the server, changes the state to `ESTABLISHED` and returns **true**. Otherwise it returns **false**.

The **passive** part of the connection is implemented by the `accept` method. A server calls `accept` which blocks until a valid `SYN` segment arrives. By that time, the state is changed to `SYN_RCVD` and a `SYNACK` segment is returned. Finally, the state is changed to `ESTABLISHED` and the method returns.

During the connection setup each part (client,server) sets the values for its own variables (IP,port,initial sequence number) and updates the relative variables of the other part upon the receipt of the first response segment.

2.1.2 Data Exchange

The `read` method is only applicable when the state is one of `ESTABLISHED`, `FIN_WAIT_1` or `FIN_WAIT_2`. The two last states represent the scenario when we have closed our part of the connection but we have to read until the other part closes the connection as well.

The `read` method maintains a buffer in which stores data, that have been received but not consumed yet. Accordingly, when invoked, first checks the buffer. If there are more data in the buffer than the application asked `read` to

read, then no receipt of a new segment is required. If not, we wait for a new segment to arrive, we return the data of the required length and store the rest data for future consumption. The `read` method returns the number of bytes received.

On the other hand, the `write` method is applicable only if the state is one of `ESTABLISHED` or `CLOSE_WAIT`. This latter case is when we have received the `FIN` segment from the other part.

Furthermore, the `write` method takes care of the fragmentation of the data to be sent. If the data exceeds 8172 bytes, we have to send multiple segments. The `window` is set to one and hence for each segment we send we require to receive an acknowledgment before we send a new one. The `write` method returns the number of bytes sent.

2.1.3 Closing Connection

Closing a connection can be either `active` or `passive`. In the first case either of the parts can decide that wants to release the connection by invoking the `close` method. This can take place either in `ESTABLISHED` state or in `CLOSE_WAIT` state. In the first case, the closing part sends a `FIN` segment and changes its state to `FIN_WAIT_1`. At this point there are two possible scenarios. In the first scenario, an acknowledgment is received, the state is changed again to texttttFIN_WAIT_2 in which we have to keep the connection open until the `FIN` segment is received from the other part. In the second scenario instead of an acknowledgment the `FIN` is received directly (simultaneous close). In this case the initiator is able to close the socket after waiting a certain time.

Furthermore, the **active** close can be invoked in the `CLOSE_WAIT` state in which the initiator sends a `FIN` segment and waits for the last acknowledgment from the other part in order to terminate (close socket, release resources).

The **passive** is handled by the method `receivedFIN` method which is invoked when we receive a `FIN` segment from the other part either when we read or when we wait an acknowledgment for a segment we sent.

Accordingly, when we receive such a segment in the `ESTABLISHED` state, we change the state to `CLOSE_WAIT` and we acknowledge the `FIN`. If on the other hand we receive the `FIN` in the `FIN_WAIT_2` we change the state to `TIME_WAIT` and terminate the connection after a certain time period. The final case is when we receive a `FIN` segment in the `FIN_WAIT_1` state (simultaneous close). In this last case the termination of the connection is handled from the `close` method.

2.1.4 Auxiliary Methods

The five basic methods of the Interface described above rely on a number of auxiliary methods which handle sending a segment (`send`, `sendSegment`, `sendSYN`, `sendACK`, `sendFIN`), receiving a segment (`receiveSegment`, `receiveDataSegment`) and identifying a segment (`isPreviousNonSYN`, `isPreviousSYN`) . These are the methods that are in charge of receiving and sending data and control segments changes by updating the values of the sequence numbers according to the protocol.

Methods `send` and `receiveDataSegment` take into account possible packet loss and try to send or receive a new segment ten times respectively.

2.1.5 TCP Segment

In this class all the necessary variables regarding a `TCP` connection are defined. Two constructors are provided, one for constructing a new segment from an `IP Packet` receipt and one for creating a new segment to send. Furthermore, we provide a method for computing the checksum (`ComputeChecksum`) of the segment and a method for checking the segment validity (`isValid`). Finally, five more methods are defined to check if the received segment is a delayed segment (`isPreviousData`, `isPreviousFin`,`isPreviousSYNACK`,`isPreviousSYN`).

2.1.6 TCP Control Block

In this class all the necessary variables regarding a `TCP` connection are stored.

2.2 Testing

In testing we tried to test all the scenarios that may come up when using the TCP library. We only tested public functions and the results that they give to the user of the library. Each of the tests is performed with packet corruption and packet loss by changing the system variables `PACKET_LOSS` and`PACKET_CORRUPTION` to 4%.

Checksum Test

1. Check the result of the checksum function

Connect Test

1. Normal connect
2. connect to non existent socket
3. connect to an already bound server
4. connect to a wrong port
5. perform the same tests in an unstable environment

ReadWrite Test

1. read-write one segment of 10 bytes
2. read-write two segments of 5 bytes each
3. read-write using the buffered data
4. read-write expecting more data than written
5. read without connection
6. write without connection
7. read-write 30000 bytes
8. write after close
9. read after close
10. perform the same tests in an unstable environment

Close Test

1. normal close (simultaneous close)
2. close unopened connection
3. close a closed connection
4. perform the same tests in an unstable environment

3 Chat App

The Chat App is implemented with an Android Activity. Its main components are the `Upper Class` and the `Lower Class`, which implement the `Runnable` interface. Both are wrappers around the two sockets that initialize new threads for reading and exposing writing functions. The `Upper Class` is responsible for accepting connections and Lower for connecting. The `run()` of the two classes perform the three-way handshake and initialize a new thread of the `Class Reader` that reads constantly through the socket. Every time new data are read `Reader` changes the values of the TextViews by adding the received message using the `runOnUiThread` function. We chose to use messages of fixed size, `50`, filling up smaller messages with 0s or reducing the size of larger ones, so that we know what size we expect to read from each `TCP Segment`. The life-cycle of the App is:

onCreate Initialize visual elements, set listeners and create the initialize the `Upper` and `Lower` Runnables.

onResume Run the two Runnables.

onPause Start two threads that perform `close()` on both the sockets.

4 Problems

4.1 Lost last ACK

Our Implementation successfully passes all the tests with some exceptions. The main problem is when the last ACK is lost during writing. The sending socket then assumes that the last segment did not arrive and the receiving segment stops reading, so it is impossible for it to detect that the sending one repeatedly tries to resend it. The test ends up failing as the writing bytes are less than expected.

In the case of lost last ACK in connect and close procedures the socket assumes that the procedure finished correctly after trying to resend last Segment 10 times. We think that this is the desired approach, because there is no way to detect such failure. In the case of lost ACK in the three way handshake we can trust that the failure will be detected in the following `read()` calls.

4.2 Reference close incompatibility

We found that our implementation is a bit incompatible with the reference close from the basic test suite and we are not sure if the problem is on our end. We note as `A` the reference TCP and as `B` our implementation. The first number is

the sequence number and the second the acknowledge number The succession of messages in close procedure is:

Messages from Logcat

1. B sends FIN [1,5]
2. A receives FIN[1,5]
3. A sends FIN[5,1]
4. A sends ACK[5,2], which is detected as an invalid segment, expected [6,2]
5. B receives FIN[5,1]
6. B sends ACK[2,6]
7. A receives ACK[2,6] and closes
8. B receives ACK[5,2] and cannot accept it.

We are using these numbers to make the issue more visible. The implementation proceeds to close normally, because it stops waiting for ACK after 10 tries.

References

[1] A. Tanenbaum, *Computer Networks*, 4th ed. Prentice Hall Professional Technical Reference, 2002.

[2] J. F. Kurose and K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt