

Computer Networks Practicum 2011-2012

Vrije Universiteit
Amsterdam, The Netherlands

<http://www.cs.vu.nl/~cn/>

1 Overview

This practicum consists of two parts. The first is to build a TCP implementation for Android smartphones. The second is to develop a Chat application over the TCP stack of the first part. Both parts are to be written in Java using the Android operating system. Other languages and operating systems are not permitted. Working alone is permitted, but working in groups of more than two is not permitted.

The exercise consists of implementing a library for connection-oriented network service and a small application which uses this library. This service is a simplified version of TCP on top of IP as described below. The TCP standard is defined in RFCs 793, 1122, and 1323, available at <http://www.cs.vu.nl/~cn/>. Note: Since you are expected to implement a simplified version of the TCP protocol you will be able to skip a lot of details provided by these RFCs. However, reading the RFCs will help you significantly in the implementation so please read them!

Your TCP implementation will need IP primitives that are usually not directly available to user programs. You are not asked to implement your own IP layer but you will use a special IP implementation included in the `cn.tar`.

1.1 Book

In the following, references to pages, figures and chapters, are of Andrew S. Tanenbaum's book *Computer Networks*, fourth edition.

1.2 Staying up to date

The latest information on this practicum can always be found on <http://www.cs.vu.nl/~cn>. Read all the documentation carefully. You have to check back with this site regularly for changes. It is

understood that if it is on the site then you should know it.

1.3 Grading

Your assignment will be graded as follows:

- Source code quality: **2.5 points**
- Test code (yours) quality: **2.5 points**
- Passing our extensive testing: **2.5 points**
- Documentation in PDF: **2.5 points**

Your grade is the sum of these points for a maximum of 10 points.

Source code quality Your source code should be well structured. It should have a clear flow, meaningful function names, variables, etc. It should necessarily have a lot of comments that help a third person quickly understand what is happening. This includes comments before each function describing what it does, as well as comments in the code describing particularly tricky sections of your code.

Documentation From this you can tell that it is equally important to write good documentation. You should work on your documentation from the beginning for two reasons, 1) to prevent a rush job at the end, and 2) so you will not forget to document important stuff. You should document features of your design as well as why you made the decision to design those features in the first place. Do not tell the story of how you built your system. Documentation is not a novel of what you did, it is an explanation of how the system works and why.

1.4 How to organize your files

Please download the file `cn.tar` from the course's web page. Expand it in an empty directory, and it will create three top-level folders: `cn`, `doc`, and `cntest`:

cn/ → This is the actual project to implement. In `src/nl/vu/cs/cn` you will find three classes:

- **IP:** Do not alter this class. It is the IP layer, that you will use in your TCP implementation.
- **TCP:** This is where you will implement your TCP stack, which will use the IP layer.
- **Chat:** This is where you will implement your Chat application, which will use the TCP stack.

doc/ → This is where you should place your documentation, in **PDF** format.

cntest/ → This is where you will create an Android test project, and place your test cases.

Of course, you are free to create additional `.java` files that you would like as a portion of your implementation and testing.

1.5 Code Submission

You can submit your code (details on the course web page) whenever you want. Even if you are not finished, you still might learn something from the output of the tests. For your convenience, you can submit your work for base-testing up to three times every month. You can submit any time of the month . Your code will be tested and results will be sent back as soon as possible. Since the assistants might be busy with their work at times, it might take up to a week for the assignment to be tested.

1.6 Developing for Android

Developing for Android is easiest with Eclipse, ADT, and the Android-SDK. Documentation of how to setup Eclipse for Android development is found here: <http://developer.android.com/guide/developing/index.html> It is possible to test your software on the emulator, which Eclipse will launch for you if you run your project. Alternatively, if you own an Android phone you can also test directly on a phone.

1.7 How to prepare yourself

You should do the following as preparation for the exercise:

- read Chapter 6 of the Computer Networks book by Andrew S. Tanenbaum;
- read RFC 793: Transmission Control Protocol. J. Postel. Sep-01-1981;
- read and understand the code of the provided IP library (many of the problems you will encounter while programming TCP have already been solved in the IP code);
- last but not least, check the web site <http://www.cs.vu.nl/~cn/>, and check it regularly.

2 Part I: Building a TCP stack

As *Part I* of this practicum, you will have to write a TCP implementation. At the application level, a connection between a client and a server is established using the socket primitives (see Figure 6-5, page 487 and Figure 6-6, page 490-491).

Phases Your implementation must follow the three phases of connection-oriented protocols: connection setup, data transfer, connection termination. You must handle the standard three-way handshake connection setup, containing SYN and ACK flags.

Multiplexing You only have to support one TCP connection at a time, so no multiplexing and de-multiplexing is needed.

Options You may assume that no options are ever used.

Checksums You must compute checksums and verify them on receipt.

Flags Ignore URG. Set PSH on all outbound packets. Do not support the RST flag, clear it on all outgoing packets. Use the other flags as required by the TCP protocol. However, since you don't have to handle more than one connection at a time, if a request for connection comes in while one is already open you may discard it and the sender will time out.

Window Management You do not have to implement a window management scheme. The window size will always be *one packet*. This means you are asked to implement a *stop'n'go* protocol: the sending side transmits one packet and waits for the acknowledgement before doing anything else; the receiving side immediately sends an acknowledgment for each packet it receives. You have to make the distinction between data and control packets. Do not send TCP packets of more than 8KB (including the IP and TCP headers). If an acknowledgment has not been received within *one second*, the packet is assumed to have been lost. You must retransmit the packet and wait for the acknowledgment again (with a maximum of *ten times*). Since you are to implement the stop'n'go protocol, ignore the window size, set it to the maximum size of one packet on all outgoing packets.

Advanced Features You do not have to implement things like the Van Jacobson slow-start algorithm or the Nagle's algorithm. You are not asked to implement quality of service features, flow control, or crash recovery.

Interoperability Your implementation has to work seamlessly with our reference implementation.

2.1 TCP interface

The TCP interface consists of two classes: `TCP` and `Socket`. The former (`TCP`) is used to instantiate a new TCP stack as well as new `Socket` objects. It exports one public constructor and two public methods, explained in Table 1

<code>TCP(int addr)</code>	Constructor. Creates a new TCP stack, at internal address <code>192.168.0.addr</code> .
<code>public Socket socket()</code>	Returns a new instance of the <code>Socket</code> class, intended to be used as a client socket.
<code>public Socket socket(int port)</code>	Returns a new instance of the <code>Socket</code> class, intended to be used as a server socket.

Table 1: Methods of TCP class

Your `Socket` class implementation must support *no* public constructor (it can only be instantiated by a `TCP` instance), but should support five public methods, listed in Table 2, with the respective parameters shown in Table 3.

<code>public boolean connect(IPAddress dst, int port)</code>
<code>public void accept()</code>
<code>public int read(byte[] buf, int offset, int maxlen)</code>
<code>public int write(byte[] buf, int offset, int len)</code>
<code>public boolean close()</code>

Table 2: Methods of Socket class

<code>IPAddress dst</code>	IP address of the machine to connect to
<code>IPAddress src</code>	IP address of the interface to accept on. This may be zero in which case the implementation should select an interface
<code>int port</code>	port to connect to
<code>byte[] buf</code>	the data buffer to be read or written
<code>int offset</code>	the offset within <code>buf</code> to operate on
<code>int maxlen</code>	the maximal amount of data to receive
<code>int len</code>	the amount of data sent

Table 3: Parameters of Socket API

The descriptions of the `Socket` class' methods are shown in Table 4.

<code>connect()</code>	Non-blocking method that connects a client socket to a server on host <code>dst</code> using port <code>port</code> .
<code>accept()</code>	Blocking method, that waits for an incoming connection on port <code>port</code> . It is used by a server process (e.g., a Web server) to wait for a remote machine to try to connect to that port. It blocks until a connection is established.
<code>read()</code>	Blocking method that reads data from the other side. Reads a maximum of <code>maxlen</code> bytes using the current open TCP connection, and stores them at <code>buf</code> starting from position <code>offset</code> . If there is no data, and the other side closed its output channel, it returns 0. In case of error, it returns -1.
<code>write()</code>	Non-blocking method that writes data to the other side. Writes a maximum of <code>maxlen</code> bytes from <code>buf</code> starting at position <code>offset</code> , using the current open TCP connection. If there is an error, but some data was written, it returns the number of bytes that the other side acknowledged. In case of any other error, it returns -1.
<code>close()</code>	Non-blocking method that closes the current TCP connection for writing. Note that this method almost always returns true, because possible communication errors will eventually lead to a closed connection. The only time it does return true, is when it is called and there is no currently active connection.

Table 4: Methods of Socket class (with description)

2.2 Development Roadmap

Implementing the TCP library can be a tricky process. Here we give a high level suggestion on the roadmap to follow, without it being the one and only way.

2.2.1 Basic IP packets

First write functions for sending and receiving (e.g., `send_tcp_packet()` and `recv_tcp_packet()`) single packets through the IP layer. These functions should be internal to your implementation, i.e., they should not be made public. They should:

- Encode/decode TCP headers (see Figure 6-29, page 537);
- Compute and verify the checksums;
- Transmit the packets using `IP.ip_send()` and `IP.ip_receive` (see Section 2.5). `IP.ip_receiv_timeout` (see Section 2.5).

You should write a test program that performs some test on your functions. To test your send and receive functions, try to encode/decode well-known data. Check that your receive function gets the same data you fed your send function with.

2.2.2 Simplified TCP

TCP operates based on a finite state machine, shown in Figure 6-33, page 543. Given this, you can define a class that holds the state of a TCP connection, which should be stored on a per socket basis.

It would help, for instance, to define an enumeration of the states:

```
public enum ConnectionState
{
    S_CLOSED, S_LISTEN, S_SYN_SENT, S_SYN_RCVD, S_ESTABLISHED,
    S_FIN_WAIT_1, S_FIN_WAIT_2, S_CLOSE_WAIT, ...
}
```

and a class that stores the various variables regarding a TCP connection (typically called the TCP Control Block):

```
/**
 * TCP Control Block
 */
public class TcpControlBlock
{
    IPAddress      tcb_our_ip_addr;          // Our IP address
    IPAddress      tcb_their_ip_addr;       // Their IP address
    short          tcb_our_port;            // Our port number
    short          tcb_their_port;         // Their port number
    int            tcb_our_sequence_num;    // What we know they know
    int            tcb_our_expected_ack;    // What we want them to ack
    int            tcb_their_sequence_num;  // What we think they know we know
    byte           tcb_data[TCB_BUF_SIZE]; // Static buffer for recv data
    byte           tcb_p_data;             // The undelivered data
    int            tcb_data_left;          // Undelivered data bytes
    ConnectionState tcb_state;            // The current connection state
}
```

Note this is pseudocode to serve as an example. It is not advisable to cut it and paste it into your assignment. You might, for instance, need to add more fields or to use different data types for some of the listed fields.

Now, you should start writing the implementation of the Socket class methods `accept()`, `connect()`, and `close()`. Think carefully what changes these functions would make to the `TcpControlBlock` variables. These three functions should not do any communication. The true three-way handshake, `connect()`, and `close()` will be implemented later on. For now, they just simulate the start and finish of the communication.

Now, fully implement the Socket methods for reading and writing TCP packets, namely `read()` and `write()`. You can test your implementation by checking that your client and server are actually able to communicate, that they send and get back the correct information.

2.2.3 Complete TCP

At this point, your TCP implementation is actually able to maintain all the state information, correctly encode/decode TCP headers, calculate and verify the checksums, send and receive TCP

packets. However, you still have to manage the connection setup and termination automatically, as well as the state transitions shown Figure 6-33, page 543. This is the time to complete the implementation of `connect()`, `accept()` and `close()`.

The `connect()` method does a three-way handshake with the `accept()` method. Implement the three-way handshake as described in section 6.5.5. page 539-541. You do not have to implement the reset, or the simultaneous connect, only simultaneous close. At the end of the connection procedure, both the client and the server should be in the `S_ESTABLISHED` state, ready to send or receive packets. Test it with a simple client and server.

The last method remaining is `close()`. The connection release is symmetric: both parties have to call `close()` to get a connection closed. This means, that calling `close()` is like a promise not to call `write()` anymore. And your implementation should hold the application to that promise. (i.e., `write()` should return -1 if it is called after the application has called `close()`).

It is not uncommon for applications to do something like this:

```
// Client
if (socket.connect(...))
{
    socket.write(...);
    socket.close();
    while ((len = socket.read(buf, ...)) > 0)
        window_show(buf, len);
}
```

Note the `close()` right after the `write()` call, after which the application keeps calling `read()` until there is no more data (or an error). Your implementation should allow this behavior. When both sides have released their connection using `close()`, then and only then the connection is closed.

At this point, your implementation should be able to handle connection establishment and termination. You can test it by having a client and a server establishing and releasing a TCP connection. Then, extend your test by adding packet transmission and by checking the correctness of the data received.

2.3 Error handling

The TCP protocol is mainly about two things. Offering connection-oriented connections and handling (hiding) network errors.

By now, your implementation is able to offer connection-oriented communication to the application layer. So now you will have to make it robust. Your TCP implementation should now be fully working as long as the environment is reliable.

So start thinking about what can go wrong. Think about packet loss and extend your TCP implementation with the management of packet timeouts and retransmissions. Use a timeout of one second. Figure out which methods should be non-blocking. Make sure these functions do not block, by waffling timeout code around potentially blocking calls within their code.

2.3.1 Handling packet loss

Now, consider all scenarios of packet loss during some information exchange and check what happens. Figure out what will happen, and whether your implementation should do something about it. Implement the result, and make some notes you can later use in your documentation.

Often, it will be very clear how to handle packet loss. For example, consider a server doing a `write()` and a client doing a `read()`. On a perfect network, `write()` would send a data packet, and `read()` would send an acknowledgment. Suppose the data is lost. Since the client side does not know data has been sent, it waits (`read()` is a blocking call) and `write()` waits for the proper ACK number. After a while the server side will time out and send the data again, we are now back at the beginning of the cycle.

Sometimes, however, there is more than one solution. Imagine `write()` waiting for a correct ACK number, and a packet comes in with that ACK number, but the packet also contains data. This situation can arise if an ACK packet was lost, and the other side sends over some data. What should `write()` do with the data? If it ignores it, it will be resent. It could also save the data (and send an acknowledgment back). The latter case is more bandwidth friendly.

For each type of error, add a test program to check if your implementation handles it, as it should. Don't forget to document your choices.

To test the packet loss, create a bogus function `send_tcp_packet()`, that ignores every, say, third packet. This will simulate packet loss. You should then check that the timeouts actually occur and observe packet retransmissions, using debug output.

2.3.2 Handling corrupted packets

After this, consider what can go wrong when packets get corrupted. For each field in the header decide what to do with a wrong value. Consider what to do when a packet is longer or shorter than expected. Make sure your implementation handles these corruptions as you planned. Documenting your decisions is very important. Write test programs to corrupt packets and test if your implementation handles them as you specified. You could replace calls to `ip_receive()` with calls to `my_ip_receive()` and have that function make changes. Ditto for `ip_send()`.

Now consider what happens if a bogus packet arrives. If you handled the corrupted packets well, a bogus packet should be identified as corrupted. However, it can pay off to think about what would happen if an attacker constructed a packet to hinder your communication. Take a look at your source code to see if there are any holes to exploit. Play a lot of “what if” games.

Something else that can go wrong is packet ordering. However, since you are not requested to implement windowing, you would be hard pressed to think of a situation where your implementation sends packets out of order. Because of the stop'n'go protocol, you can assume that out of order packets can be regarded as corrupted, or bogus packets. However, a fully blown TCP implementation, could very well send several packets, of which your implementation could only handle the first. You might want to design a test based on this.

2.3.3 Handling other problems

Your implementation will have to handle packet loss and corruption. You are not asked to implement quality of service features, flow control, crash recovery, and so on. In case you do feel you need to implement extra things, please do not forget to document your extra features.

There are many, many details you have to pay attention to. Programming, after all, is about details. One particular error has been showing up time over time, so to prevent a lot of discussion about it later, perform the following test. Change your code (temporarily) to initialize your counters to very high values, to see what happens if one of your counters wraps. Your implementation should be capable of handling such wraps.

2.4 Testing

Testing should be done by two main actors: *you* and *us*!

2.4.1 How you should test your implementation

You will have to thoroughly test your implementation using the Android test project you will create in the `cntest` directory. (Eclipse → New... → Project... → Android Test Project). This project should include test cases which extend `AndroidTestCase`. If you run this project using Eclipse all tests should pass.

Testing is a simple process: Think of a test... predict the outcome... do the test... check the results. Sounds simple, it *is* simple, but in practice it also is hard!

A portion of your grade will be based on the tests you write for your implementation. Begin by writing simple tests, for example, open a connection in one thread and accept the connection in another. Add additional tests which transmit and receive data, tests for various buffer sizes and corner cases, etc. The more thorough your test suite, the more likely you are to pass our extensive test suite, and the better your grade for the tests.

Read the assignment again, carefully, and make a list of do's and don't's. Make a separate test case for each do and don't and turn this in with your project.

Needless to say that the above is just the beginning of real testing. There are many more ways to test code. And indeed you should do more testing.

Remember: *What can go wrong will!*

2.4.2 How we will test your implementation

To get a grade, your TCP implementation must pass the following tests. We will first check if your TCP implementation conforms to the provided interface and does not expose additional public methods.

We will check that your TCP implementation is actually able to correctly send and receive packets, to handle TCP flags, calculate and verify checksums. We will perform simple message exchanges. After checking the behavior of your TCP implementation in a reliable environment, we will test its capabilities to handle packet loss and corruption. Your client and server programs

should not be affected. We will also check that your implementation is able to interoperate with a reference implementation.

2.5 Utilization of the IP Library

Your TCP library will rely on the IP library that is provided to you. This is a user-level library that can send/receive IP packets by encapsulating them in UDP packets sent over the local interface.

Note that it is possible to have more than one “virtual address” by constructing more than one IP stack. Conceptually you can think of this as either having multiple hardware devices, each bound to the same network, or alternatively as having one hardware device which is bound to multiple IP addresses. In fact, in order to properly construct the Chat Application described in the next section you will have to instantiate two TCP stacks with different underlying IP addresses.

3 Part II: The Chat Application

In *Part II* of this practicum, you are requested to build a very simple Chat application that uses the TCP stack you developed in Part I. This application will offer two “windows” so you can chat with yourself. Each “window” should have a text input below it and a send button to send the line of text to the other window, as well as clear the input area. The sending of messages between the two must be handled using your TCP stack. You can see an example of what the Graphical User Interface (GUI) for your application should look like in Figure 1. You are not required to match this GUI exactly, but the overall functionality should be the same. You are free to construct this GUI using code, or with an XML layout.

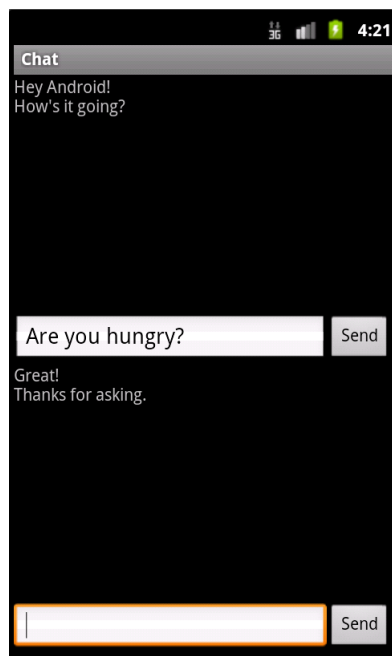


Figure 1: Sample Chat GUI