Computer Networks Practicum, 2008 / 2009

Vrije Universiteit
Amsterdam, The Netherlands

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

1 Introduction

This programming assignment is different from all previous programming assignments. Up till now, you could do the programming while reading the assignment, because up till now, the description of an assignment was a step-by-step description of what to do, this is how you have learned to program.
This practicum, however, is about *networking*, and assumes you have some programming experience. Consequently, you yourself have to create a step-by-step description (i.e., a design), and for that, you will need a deep understanding of the assignment.
You can easily fail this assignment, without really trying. (Read it again, and you will get it.) Reading the handout once, usually, does not lead to the level of understanding it takes. Please read the assignment more than once and ask questions of the assistant if you do not understand. It is highly suggested that you start early and ask questions early. Those who start late usually do not finish or if they do receive a lower grade than those who start early. Don't procrastinate!
You can submit your code (details on the web site `http://www.cs.vu.nl/~cn/`) whenever you want. Even if you are not finished, you still might learn something from the output of the tests. We will test your assignment the first week of each month and give you feedback. Additional test runs will not be performed, so start working early in order to get the most test runs of your code!
In the following, references to pages, figures and chapters, are of Andrew S. Tanenbaum's book *Computer Networks*, fourth edition.

2 The assignment

This practicum consists of two exercises. You are to do both of them. They are to be written in C using the MINIX 3 operating system. Other languages and operating systems are not permitted.

It is recommended that students work in groups of two and turn in one solution to each exercise per group. Working alone is permitted, but working in groups of more than two is not permitted. The first exercise consists of implementing a library for connection-oriented network service. 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 implementing 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 the one provided at http://www.cs.vu.nl/~cn/ip.tar.gz. Interfaces to this IP library layer are provided below.
The second exercise consists of implementing a simple World-Wide Web server on top of your TCP implementation, again in C on MINIX 3. You are to implement a subset of HTTP 1.0, which is described in RFC 1945. This version is obsolete, but it is much simpler than HTTP 1.1 and existing servers browsers generally support it. You should implement a very minimal subset as described below. Although they contain much more information than you actually need, you might have a look at the RFCs 1738 (URLs), 1521 and 1522 (Mime types). See http://www.cs.vu.nl/~cn/.

2.1 How your assignment will be graded

Your assignment will be graded as follows:

- You will get up to 3 points for your source code,
- up to 3 points for your documentation, and
- up to 3 points for passing our extensive-test set,
- up to 1 point for your own test code,
- Your grade is the sum of these points

for a maximum of 10 points.
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.
You should also document your code. Part of your code grade is the internal documentation of your code. This include comments before each function describing what it does as well as comments in the code describing particularly tricky sections of code.

2.2 The latest information

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.

2.3 How you should organize your files.

You will use the following directory structure for the development of your assignments:

```
<YOURDIR>/http
<YOURDIR>/tcp
<YOURDIR>/ip
<YOURDIR>/test
<YOURDIR>/doc
```

Where `<YOURDIR>` usually is: `/home/cn`. But be advised, your `Makefile` should not contain absolute paths like `/home/cn/ip`, use things like `../ip`, instead.
The `tcp` directory will contain all the files for exercise 1. The `http` directory will contain all the files for exercise 2. The `test` directory will contain all the files that you use to test your implementations. The `doc` directory will contain your documentation which MUST be in PDF format.
For exercise 1 your will create a `tcp.c`, a `tcp.h` and a `Makefile` file in the `tcp` directory. The command `make` should create a library `libtcp.a` Note that we will define DEBUG in the environment for test runs of your code when we execute make but the final test run will not define DEBUG. Your implementation must not print anything when DEBUG is not defined in the environment to make.
For exercise 2 your will create a `httpd.c`, a `httpc.c`, and a `Makefile` file in the `http` directory. You can also make a `http.h` for inclusion in both `.c`-files and a `http.c` file containing code that is common to the `httpd` and `httpc` program. The command `make` should create the executables a `httpd` and a `httpc`. Make should link the `httpd.o` and the `httpc.o` to the `http.o` file and the libraries `..\tcp\libtcp.a`, and `..\ip\libip.a`. The same rules as for exercise 1 and the defition of the DEBUG environment variables apply for this exercise.
You may use any additional .c or .h files that you would like in either of these directories.

2.4 How you should prepare yourself

You should do the following as preparation for these exercises:

- 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);
- read Section 7.2.3 and 7.3. of the Computer Networks book by Andrew S. Tanenbaum;
- read RFC 1945: Hypertext Transfer Protocol – HTTP/1.0 by T. Berners-Lee, R. Fielding and H. Frystyk;

- last but not least, check the web site `http://www.cs.vu.nl/~cn/`, and check it regularly.

3 Exercise 1: TCP

You will have to write a TCP implementation. 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.
You only have to support one TCP connection at a time, so no multiplexing and de-multiplexing is needed, however a bonus will be awarded to implementations which can handle this added complexity in our extended test suite. You do not have to handle port number allocation on the client side. Once again bonuses are available for implementations which handle client side port allocation properly but this is very hard. This means you can use a fixed port for client connections (e.g. 8042). The server side has to have a well known port number so that your client can connect to it (e.g., port 80 in case of a Web server). 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). At the TCP level, you have to implement the standard 3-way handshake connection setup as an exchange of TCP packets containing SYN and ACK flags.
You must compute checksums and verify them on receipt. You may assume that no options are ever used.
You are to treat the TCP flags as follows. Ignore URG. Set PSH on all out bound packets. Do not support the RST flag, clear it on all outgoing packets. Use the others flags as required by the TCP protocol. However, since you don't have to handle one connection at a time, if a request for connection comes in while one is already open ou may discard it and the sender will time out.
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 acknowledgment before doing anything else; the receiving side immediately sends on an acknowledgment for each packet it has received. 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 10 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.
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.
Your implementation has to interwork with other student implementations. Since TCP implementations (e.g., Solaris, native Minix) do not follow the stop'n'go protocol, your implementation will not fully inter operate with them. These real-life implementation are, however, well suited to test the checksum and even the complete three-way handshake of your implementation against. Moreover, you will have to set the window-size in the TCP header to the size of one packet. This will allow you to inter operate better with a full blown TCP implementation. You should write a small telnet-like program and test your implementation against a regular (Minix) telnet daemon. You should also find other ways to test your code as this is part of your grade.

3.1 TCP interface

Your TCP implementation must support the following functions that user programs can call:

```
int tcp_socket();
int tcp_connect(ipaddr_t dst, int port);
int tcp_listen(int port, ipaddr_t *src);
int tcp_read(char *buf, int maxlen);
int tcp_write(char *buf, int len);
int tcp_close();
```

where the parameters are:

- `ipaddr_t dst`: IP address of the machine to connect to;
- `ipaddr_t *src`: IP address of the interface to listen on. This may be zero in which case the implementation should select an interface;
- `int port`: port to connect to;
- `char *buf`: the data buffer to be read or written;
- `int maxlen`: the maximal amount of data to receive;
- `int len`: the amount of data sent.

The function `tcp_read` returns the number of bytes read or -1 in case of an error. Reaching the end of a stream is not considered as an error, so in this case `tcp_read` should return 0. The function `tcp_write` returns the number of bytes written or -1 in case of an error. All other functions return 0 for correct execution, -1 in case of an error.
`Tcp_socket()` is a non-blocking function, that initializes the TCP variables (TCP implementation state, global variables).
`Tcp_connect()` is a non-blocking function, that connects a client to a server on host `dst` using port `port`.
`Tcp_listen()` is a blocking function, that waits for a incoming connection on port `port`. 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.
`Tcp_read()` is a blocking function that reads data from the other side. The `tcp_read()` function reads a maximum of `maxlen` bytes using the current open TCP connection. If there is no data, and the other side closed it's output channel, `tcp_read()` will return 0. In case of error, `tcp_read()` returns -1.
`Tcp_write()` is a non-blocking function that writes data to the other side. The `tcp_write` function writes a maximum of `len` bytes using the current open TCP connection. If there is an error, but some data was written, `tcp_write()` returns the number of bytes that the other side acknowledged. If case of any other error, `tcp_write()` returns -1.
`Tcp_close()` is a non-blocking function that closes the current TCP connection for writing. Note that this function almost always returns 0, because possible communication errors will eventually lead to a closed connection. The only time `tcp_close()` does return -1, is when it is called and there is no active current connection.

The above functions will have to be made available as a library called `libtcp.a`. Provide the `tcp.h` include file as well.
Make sure that the `libtcp.a` library and the `tcp.h` c-header-file does not contain any other public functions variables or constants (that would be *name space pollution*). You should check the symbols from your `tcp.o` file by examining the output of (two spaces in `'U  E'`):
`anm -g tcp.o | grep -v 'U  E'`
The output should only contain the six `tcp_...()` functions, the `send_tcp_packet()`, `receive_tcp_packet()` and nothing else. You will not pass our test suite if anm returns anything besides these function.
The `send_tcp_packet()` and `receive_tcp_packet()` are not part of the interface, but are merely suggestions for the interface to the low level functions you may need. We will not test these functions for you and they should not be global symbols in the library.

3.2 How to implement the TCP library

There are two good ways to implement the TCP library. A two tier, and a three tier approach. The two tier approach is straightforward but, in general, leads to a bigger implementation that is harder to test. The three tier approach needs more preparation, but is more elegant and easier to debug.
The two tier approach is the basically divides the implementation in two parts: the connection less part, and on top of that the connection oriented part. Briefly, the lower part deals with creating and exchanging valid TCP/IP packages, and the upper part handles the connectivity (i.e., the state machine as given in the RFC and Tanenbaum's book).
The three tier approach has these two layers, but there is a layer in between these two. This middle layer generically translates TCP/IP packages into state and connection variables changes, greatly simplifying the upper layer.

3.3 The connection less part

Write functions for sending or receiving single TCP/IP packet through the IP layer. They should resemble the following pseudo code:

```
int send_tcp_packet(ipaddr_t dst, u16_t src_port,
     u16_t dst_port, u32_t seq_nb, u32_t ack_nb, u8_t flags,
     u16_t win_sz, const char *data, int data_sz);
int recv_tcp_packet(ipaddr_t *src, u16_t *src_port,
     u16_t *dst_port, u32_t *seq_nb, u32_t *ack_nb, u8_t *flags,
     u16_t *win_sz, char *data, int *data_sz);
```

where the parameters are:

- `dst`: IP address of the machine to send the data to;
- `src`: IP address of the machine we got the data from;

- `src_port`: port of the sender socket;
- `dst_port`: port of the receiver socket;
- `seq_nb`: emission sequence number;
- `ack_nb`: acknowledgment sequence number;
- `flags`: TCP flags;
- `win_sz`: window size;
- `data`: data to send or receive;
- `data_sz`: size in bytes of the data;
- return value: if successful, the size of the sent or received data in bytes; -1 otherwise.

The functions must:

- Encode/decode TCP headers (see Figure 6-29, page 537);
- Compute and verify the checksums;
- Transmit the packets using `ip_send()` and `ip_receive` (see Section 7).

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 `send_tcp_packet()` with.
You can also capture TCP packets from a working application (like `telnet`) and try to decode them with your `recv_tcp_packet` function. You might try to use the -w and -x options of `tcpdump` (see 'man tcpdump').
Since the basic test, does do some basic checking of the send and receive functions, you might get valuable information if you submit your code even if the interface functions (`tcp_...()`) are not yet in working.

3.4 The connection oriented part

To handle connection based communication, your TCP implementation, obviously, needs to keep information about its current state. You should refer to the Chapter 6: The transport layer of the Computer Networks and the Figure 6-33, page 543, in order to define precisely which variables are needed by your library. Given this, you can define a static global compound variable to be used by your TCP library or you can store this information along with the socket if you are going to support multiple connections.
You should have something like this:

```
   ⋮
typedef enum
{
  S_CLOSED, S_LISTEN, S_SYN_SENT, S_SYN_RCVD, S_ESTABLISHED,
  S_FIN_WAIT_1, S_FIN_WAIT_2, S_CLOSE_WAIT, ...
} state_t;
```

```
  ⋮
/* TCP Control Block. */
typdef struct tcb_s {
 ipaddr_t tcb_our_ip_addr;       /* Our IP address. */
 ipaddr_t tcb_their_ip_addr;     /* Their IP address. */
 u16_t    tcb_our_port;          /* Our port number. */
 u16_t    tcb_their_port;        /* Their port number. */
 u32_t    tcb_our_sequence_num;  /* What we know they know. */
 u32_t    tcb_our_expected_ack;  /* What we want them to ack. */
 u32_t    tcb_their_sequence_num;/* Wt we thk they knw we knw. */
 char     tcb_data[TCB_BUF_SIZE];/* Static buffer for recv data. */
 char    *tcb_p_data;            /* The undelivered data. */
 int      tcb_data_left;         /* Undelivered data bytes. */
 state_t  tcb_state;             /* The current connection state .*/
  ⋮
} tcb_t;

static tcb_t g_tcb; /* Global information on the connection. */
  ⋮
```

Note this is pseudo code, you obviously can't cut'n'paste this into your assignment.
Now, write a `tcp_socket()` function to initialize the variables in the `g_tcb` structure.

3.5 Communication between a client and a server

You should now write a stub (i.e., placeholder dummy) `tcp_listen()`, `tcp_connect()` and `tcp_close()`. Work out on paper, what changes these functions would make to the `g_tcb` variables, an make these functions make these changes. It should look a little like this:

```
  ⋮
int tcp_listen(...)
{
  g_tcb.tcb_state = S_ESTABLISHED;
  g_tcb.tcb_our_sequence_num = 10000L;
  g_tcb.tcb_our_expected_ack =
  g_tcb.tcb_their_sequence_num = 20000L;
  ⋮
int tcp_connect(...)
{
  g_tcb.tcb_state = S_ESTABLISHED;
  g_tcb.tcb_our_sequence_num = 20000L;
  g_tcb.tcb_our_expected_ack =
  g_tcb.tcb_their_sequence_num = 10000L;
```

```
   ⋮
int tcp_close()
   ⋮
```

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 functions for reading and writing TCP packets, namely `tcp_read()` and `tcp_write()`. It should look something like this:

```
  ⋮
int tcp_read(...)
{
  while ( <buffer_is_empty> )
    <handle_packets>
  <return_size> = min( <requested_size>, <available_size> );
  <copy_data>;
  ⋮
int tcp_write(...)
{
  while ( <bytes_left> )
    <packet_size> = min( <max_packet_size>, <bytes_left> )
    do
      <write_packet>
      <wait_for_packet>
    while ( <ack> != <what_we_want> )
    <update_bytes_left>
  ⋮
```

Depending on whether you used the three or two tier design, the `<handle_packets>` and `<wait_for_packet>` code are a simple call, or a number of lines of C code. Note that there is a very small amount of code in these psudocode functions. Try to keep it that way in your code by decomposing into functions properly!
If you follow the three tier approach, you should implement a `handle_packet()` method, and use that in the read and write functions. The code should look like this:

```
  ⋮
static int handle_packet()
{
  ⋮
  while (len = ip_receive(...)) {
    if ( <tcp_packet_contains_data> )
      if ( ... && handle_data()) send_ack();
    if ( <tcp_packet_contains_FIN> )
      if ( ... ) handle_fin();
    if ( <tcp_packet_contains_ACK> )
      if ( ... ) handle_ack();
    else
      /* What? no ack?!?!?! */
  ⋮
```

Note that at this point, `handle_packet()` does not yet have to handle SYN, and/or errors. This is just a sketch to help you get started with your design
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.
By now, you should be able to create a library `libtcp.a` that will successfully link to `simpletest.o` and you should be able to run the resulting executable.

3.6 Setting up and terminating a connection

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.
Now, implement the functions `tcp_connect()`, `tcp_listen()` and `tcp_close()`.
The function `tcp_connect()` does a three way handshake with the `tcp_listen()` function. Implement the three-way handshake as described in section 6.5.5. page 539-541. (Again, you do not have to implement the reset, or the simultaneous connect, only simultaneous close.)
Note that if you follow the three tier approach, you now have a choice, you can either handle SYN and SYN/ACK packets in `handle_packet()` or you can have the connect and listen functions use the raw send and receive TCP/IP packet functions. Make sure you document your choice.
Have a client and a server connecting to each other. At the end of the connection procedure, your client and server should be in the `S_ESTABLISHED` state, ready to send or receive packets.
Implement `tcp_close()`. The connection release is symmetric: both parties have to call `tcp_close()` to get a connection closed. This means, that calling `tcp_close()` is like a promise not to call `tcp_write()` anymore. And your implementation should hold the application to that promise. (i.e., `tcp_write()` should return -1 if it is called after the application has called `tcp_close()`. It is not uncommon for application to do something like this:

```
   ⋮
/* Client. */
if (tcp_connect(...) >= 0)
{
  tcp_write(GET_STATS, ...);
  tcp_close();
  while ((len = tcp_read(buf, ...)) > 0)
     window_show(buf, len);
   ⋮
```

Note the `tcp_close()` right after the `tcp_write()` call, after which the application keeps calling `tcp_read()` until there is no more data (or an error). Your implementation should allow this behavior.
When both sides have released their connection using `tcp_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.

3.7 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 what `tcp_...()` functions should be non-blocking. Make sure those functions do not block, by waffling timeout code around potentially blocking calls within their code.

3.8 Handle loss of packets

Now, make a diagram of all possible order of information exchange and check what happens if a packet gets lost. 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 `tcp_write()` and a client doing a `tcp_read()`. On a perfect network, `tcp_write()` would send a data packet, and `tcp_read()` would send an acknowledgment. Suppose the data is lost. Since the client side does not know data has been send, it waits (`tcp_read()` is a blocking call) and `tcp_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 `tcp_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 `tcp_write()` do with the data? If it ignores it, it will be re send. It could also save the data (and send an acknowledgment back). The latter case is more bandwidth friendly. Note that the three trier approach would automatically lead to the second approach.
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 and/or the use of `tcpdump`.

3.9 Handle 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 you 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 to think about what would happen if an attacker, would construct 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 would send 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 full 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.

3.10 Handle 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 in. 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 attend to. Programming, after all, is about details. One particular error has been showing up time and again, so to prevent a lot of discussion about it later, perform the following test. Change your code (temporary) to initialize your counters to `(unsigned long)-10` (try other small negative values as well) to see what happens if one of your counters wraps. Your implementation should be capable of handling such wraps.

3.11 How you should test your implementation

You will have to thoroughly test your implementation.
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!
First of all get the simple test program called `simpletest.c` from the web site, put it on your test directory. Make your TCP library link to it, and get it to work as you would expect it to. `Simpletest.c` sends a small string from a parent process to a child process and back.
Than you should create a new test program, by copying `simpletest.c` to `bigtest.c` and adapt it to send over a big buffer (`char buf[50*1024];`) of at least 50KB. Before sending, fill the buffer with some pattern, check for that pattern at the receiving end. Note that `simpletest`, actually, doesn't handle the `tcp_read()` correctly, the length returned could be less than the length argument. To read a buffer of a specific size, use something like this:

```
  ⋮
tcp_listen();
total = 0;
while (total < sizeof(buf)) {
  len = tcp_read(buf + total, sizeof(buf) - total);
  total += len;
}
tcp_close();
  ⋮
```

Note that this is pseudo code, the real code would contain lots of `printf()` statements and checks. For example, the return value of `tcp_read()` could be less than zero.

You should make further enhancements, for example the child should also send a large buffer, (with an other pattern) back. The size of the array 'buf' could be set to bigger sizes. The size of the array 'buf' could be set to special values, like equal to, or one more or less that the maximum TCP payload, powers of two etc. Also a test program could be constructed to test all values from 1 to 10240. Though this test would take a long time to run.

You should also make many more slightly complex programs to do additional testing. For example, you should construct a test that sends out a 1 byte buffer first, then a 2 byte, 4 byte, 8 byte, etc. The child code reads a buffer of 1024 bytes, and then 512 bytes, etc. After this, the whole process should be repeated, using powers of two minus 1. In general sending packets of one size and receiving buffers of an other, is a tough test.

Read the assignment again, carefully, and make a list of does and do not's. For example if the assignment states that alarm and signal should be restored, copy simpletest.c to alarmtest.c and make changes along these lines:

```
  ⋮
testhand() {} /* Dummy signal handler. */
  ⋮
  /* Parent. */
  alarm(30000);
  signal(SIGALRM, testhand);
  tcp_connect(inet_aton(ip2),1234)
  tcp_write(buf,sizeof(buf))
  tcp_read(buf,16)
  tcp_close()
  al = alarm(0);
  if (al < 20000)
      printf("error")
  if (signal(SIGALRM, NULL) != testhand)
      printf("error")
  ⋮
```

```
/* Child. */
alarm(30000);
signal(SIGALRM, testhand);
tcp_listen(1234,&saddr)
while (tot < sizeof(buf))
    len = tcp_read(buf,sizeof(buf));
tcp_write("foo",4)
tcp_close()
al = alarm(0);
if (al < 20000)
    printf("error")
if (signal(SIGALRM, NULL) != testhand)
    printf("error")
⋮
```

Make a separate test program for each do and don't.
Create a test program to see if your blocking function can be correctly interrupted by the application. Use code like this:

```
⋮
signal(SIGALARM, dummy());

printf("Connecting");
do {
  alarm(1);
  retval = tcp_listen( ... );
  alarm(0);
  printf(".");
} while(retval < 0);
printf("\n");

printf("Reading");
do {
  alarm(1);
  len = tcp_read( ... );
  alarm(0);
  printf(".");
} while(len < 0);
printf("\n");

if (len > 0) { /* No EOS. */
  do_funny_thing_with_data( ... );
}
⋮
```

Now, create a client that waits for input, before connecting and writing. You should be able to see the server waiting, producing output like:

```
Connecting.............
Reading.....................
```

Make a program that tells you how long each `tcp_...()` function took. Check the results against your expectations, as you would with any test.
You should also request the `libtcp.a` from an other couple (**WITHOUT THE SOURCE CODE!**) rename it to `libthem.a` and test your implementation against that. Copy `bigtest.c` (or an other) to `crosstest.c` and change the line:
`if (fork()) {` to: `if (eth[0]=='1') {`
Remove other `fork()` related stuff, like the `wait()` call.
Change the make file to compile the program in to a object file `crosstest.o` and link it to the other students `libtcp.a` in to an executable called `themtest`, and to your own library in to an executable called `ustest`. The Makefile on you test directory should contain something like this:

```
crosstest.o: crosstest.c
cc -c crosstest.c

themtest: crosstest.o libthem.a
cc -o themtest crosstest.o -L. -l them -L ../ip -l ip ...

ustest: crosstest.o libtcp.a
cc -o ustest crosstest.o -L ../tcp -l tcp ../ip -l ip ...
```

This way setting the `ETH` variable to "1" forces the test program to run the parent code, and setting it to "2" will get the child code run. So run `themtest` in one window, and `ustest` in the other.
You should also do additional testing by putting extra statements in the source code. For example:
`send_ip_package(...)`
could be changed to:

```
#if TEST == 4
  if (_cnt++ % 3 != 0) /* Simulate drop. */
#endif
  send_ip_package(...)
#if TEST == 3
  send_ip_package(...) /* Simulate duplicate. */
#endif
```

Simply compiling with the `-DTEST=4` flag would create a lib that will simulate packet drops.
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!*

3.12 How we will test your implementation

To get a grade, your TCP implementation must pass the following tests. We will first check if you provide the files `tcp.h`, `tcp.c` and `Makefile`. Then we will check if your TCP implementation provides all the functions listed in Section 3.1 and only those functions.
The code testing will take place as follow. 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. To do so, we will use a bogus IP layer that looses packets or alters their content (data or control packet). Even with such an IP layer, your client and server programs should not be affected.
We will also check that your implementation is able to inter operate with a reference implementation.
In order for our test scripts to work, your files must have the following names and directory structure in the tar file you submit.

```
tcp/tcp.h
tcp/tcp.c
tcp/Makefile
```

4 Exercise 2: World-Wide Web client and server

You will have to write a HTTP implementation, both a simple World Wide Web server as well as a rudimentary text-based World Wide Web client.

4.1 The server

You are to implement only a subset of HTTP 1.0. Your World-Wide Web server implementation must support the GET request method and return a correctly typed document. When the server receives a GET request, it figures out which object (i.e., file) is requested and sends it back. It has to return one of the following response codes:

- 200 OK;
- 403 Forbidden;
- 404 Not Found.

You do not have to support If-Modified-Since or other request fields. If you get header lines that you do not recognize, ignore them.
Name your server implementation `httpd` and provide the directory where to get the files from as a command line parameter: `httpd <directory>` . Any file present in this directory will be considered as a document ready to be delivered to a World-Wide Web client. You do ignore any subdirectories.
To test your World-Wide Web server, just link it statically with your TCP implementation. The World-Wide Web server should make calls to your TCP implementation, for example, `tcp_listen()` and `tcp_read()`.
Your World-Wide Web server should be attached to port 80 of your TCP program and wait for an incoming connection. When a connection comes in, it sends a response, closes the TCP connection and exits, i.e., the server does not loop to service more than one request like a normal server would. Depending on the file name, your server must determine the Mime type of the document and transmit it in the HTTP response. You only need to support the following Mime types: `text/plain`, `text/HTML`, `image/gif`, `image/jpg`, and `application/postscript`. Use `application/octet-stream` for any other unknown file type.
The protocol has a number of features that are not required for this exercise. You do not have to implement:

- anything related to proxies, gateways, tunnels, or caches;
- HTTP 0.9 or early versions;
- any schemes other than `http://` (e.g., NOT `ftp://` etc);
- character sets other than US-ASCII;
- content encodings;
- sub pahts in the GET request (e.g., `http://192.x.y.z/aap/aap.gif`);
- multi part messages;

- URL en/decoding (e.g. `http://192.x.y.z/the%20file`);
- requests for anything except static files (no cgi-bin);
- the header fields: Allow, Authorization, From, Location, Pragma, Referrer, WWW-Authenticate, If-Modified-Since;
- cookies.

You are allowed to implement some of the extra features as long as you document them.
There is one subtlety concerning file access. Since the cn account you will be using, has root privileges, there are not that many files, your `httpd` can not read. There should however be a sensible way to test "403 Forbidden", so you are left with two options. Either change the effective user id, see the man page for `seteuid()` and `seteguid()`, or use the system call `stat()` and check wither the "other" read bit is set, see the man page for `stat()`. It will not be enough to open the file, as a test for access.

4.2 The client

Your World-Wide Web client implementation must fulfill the following criteria. Your client must be able of retrieving a file (for example HTML) by sending a HTTP request to your server implementation. It has to send a correct HTTP request (GET), wait for the reply from the server and decode it. Once it got a reply, your client has to display the HTTP response header in plain English. You are requested to show the return code, the date, the content length, the content type and the Last-Modified fields. For example, a display could be as follow:

```
The return code is: 200 OK
The date of retrieval is: Fri, Aug 24 22:16:28 2003 GMT
The size of the document is: 42
The Mime type of the document is: text/plain
The document was last modified at: Wed, Dec 27 2002 19:35:56 GMT
```

If the request failed, you have to show the error code returned as well as its meaning.
In addition to displaying the header, your World-Wide Web client will save the document in a file whose name is identical to the end of the URL (e.g., `http://192.xx.xx.xx/bar.html` gives the file name `bar.html`). You do not have to support host to IP-address conversion, though it is very simple, and might help you when it comes to rounding your grade.
Name your client implementation `httpc` and provide a URL as a command line parameter: `httpc <URL>`. Note that the URL is not a fullblown URL, specifically, you do not have to implement sub paths, or host lookup.
You can use the IP address of the server (like: `http://192.x.y.z/aap.html`). Your client contacts the server `httpd` in order to obtain the file. The file is saved in the current directory under the file name found at the end of the URL.
To access a Web page, the client will actually establish a TCP connection to port 80 to which the server is connected to. It sends requests to the server and waits for the replies (documents or error codes).

4.3 How you should test your implementation

You should test your implementation. Write a simple shell script that calls your server (`httpd`) and client (`httpc`) with some arguments, and have check the output against what you would expect. Maybe write a script looking like this pseudo code:

```
#!/bin/sh
( export ETH='1'; httpd datadir; ) &
( export ETH='2'; httpc http://192.xx.xx.xx/non.txt ) > out

if [ -z `grep 404 out` ]; then
  echo "test.sh: Can't get a 404 on non.txt"
fi
```

You should make many more of these tests. One particular important test, is about the headers. Read the documentation very carefully and determine, which headers are obligatory, and which are not. Test if your client and server can handle the minim number of headers. Also test if you client and server can handle extra and/or unknown headers.

4.4 How we will test your implementation

To get a grade, your World-Wide Web server implementation must pass the following tests. We will first check if you provide the files `httpc.c`, `httpd.c` and `Makefile`. You have the option to also include `http.c` and `http.h`.
We will test your client and server implementation together on a few HTTP requests. For example, we will try to fetch a text file, a gif image or a postscript file. The file returned will have to have the correct mime type. The file saved by your client implementation will be compared to the original file to detect any possible corruption. We will also try to retrieve non-existing files or files for which the read permission is not set. In any case, your client implementation is requested to display the response header.
We will also test your implementation against ours. The same kind of test as above applies here. In order for our test scripts to work, your files must have the following names and directory structure. (If you do not submit the http files, for monthly base testing, you will not receive test results for the second exercise. For final submission both the tcp and the http files are required.)

- `tcp/tcp.h`
- `tcp/tcp.c`
- `tcp/Makefile`
- `http/http.h` (optional)
- `http/http.c` (optional)
- `http/httpc.c`
- `http/httpd.c`
- `http/Makefile`

5 Exercise 1: TCP quick list

This is not a replacement of the exercise description, but a brief and incomplete list of things. You do have to implement:

- test your own code;
- comply with the interface described in Section 3.1;
- implement a stop'n'go protocol;
- implement a simultaneous close;
- support a single tcp connection;
- compute and verify checksums;
- support the flags: PSH, ACK, FIN, SYN;
- manage packet loss and retransmission (timeout after 1 second);
- inter operate with other students' implementations;
- pass the tests described in Section 3.12.

You do not have to implement:

- one TCP connection at a time;
- no IP control packets (e.g., ICMP);
- no options;
- no URG, RST flags;
- all packets have flag PSH;
- all packets have flag window size to the size of one packet;
- no slow-start;
- no Nagle's algorithm;
- no IP fragmentation;
- no quality of service support;
- no port number allocation;
- no flow control;
- no crash recovery;
- no simultaneous connect;
- no windowing.

Here are a few tips:

- convert to network order as late as you can, and convert to host order as quick as you can, using `hton()` and `ntoh()` functions;
- use no `thread()`, no `fork()` and `exec()`, you don't need them, since you handle only one connection at a time;
- Do not submit any code that has debug output in it, since it interferes with our test scripts.

6 Exercise 2: HTTP quick list

This is not a replacement of the exercise description, but a brief and incomplete list of things. You do have to implement:

- support the GET request method;
- manage Mime types: `text/plain`, `text/HTML`, `application/postscript`, `image/gif`, `image/jpg`, `application/octet-stream`;
- support response codes: 200, 403, 404;

You do not have to implement:

- anything related to proxies, gateways, tunnels, or caches;
- HTTP 0.9 or early versions;
- any schemes other than http:// (e.g., NOT ftp:// etc);
- character sets other than US-ASCII;
- content encodings;
- multi-part messages;
- requests for anything except static files (no cgi-bin);
- the header fields Allow, Authorization, From, Location, Pragma, Referrer, WWW-Authenticate, If-Modified-Since;
- cookies.

Tip: to see how the browser and server normally interact, use telnet. Activate a shell window and type:

```
telnet www.cs.vu.nl 80
GET /~ast/index.html HTTP/1.0
Host: www.cs.vu.nl
```

followed by a blank line. The server will return the home page of ast. You can also substitute your login name for ast and you will get your home page. In effect, what you are to do is write a program like our server that responds to GET requests by returning the file requested. The Host: command is not in HTTP/1.0 but many modern servers require it. Your server should just ignore it (and other headers it does not understand).

7 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 accessing the Ethernet card directly.

7.1 Minix System Setup

The practicum must be run on Minix 3. You can use the machines in P4.29 or you can install Minix on your own machine or in a virtual machine. See http://www.minix3.org for more information. Note that this is not a supported configuration and your code will be tested on the version of Minix in P4.29 so it must work there. Please do not ask for assistance with installing Minix on your own machine.
Note that the machines in P4.29 may also be used by people working on the operating systems practical, though the two do not run at the same time. In order to ensure that the machine is prestine and has not been altered by another user login as root and issue the `mnx_fresh` command. This will reimage the machine. It is a good idea to reboot after doing this.
Next you need to install the computer networking library on the machine. You can download the libcn.tar archive from the website. Unpack this archive and run make as root. This will install the headers you will need to compile the IP library as well as the library itself you will need for linking. It also installs a cneth script which you may use to help you figure out how to configure the networking on the machines in P4.29.
Each of the machines in P4.29 can be given up to *3 virtual IP addresses* all of which refer to the same Ethernet card. The first IP/Ethernet address set is used for normal network operation by the system. The other two can be used through the IP library, so that you can have two processes on the same machine but with different IP addresses communicating through the library.
In order to configure your minix install to have multiple ip addresses you must add virtual interfaces to /etc/inet.conf. Edit /etc/inet.conf to configure the two virtual ethernet cards. If you are running Minix on your own machine or in a virtual machine the first line should reflect your setup.

```
eth0 rtl8139 0 { default; };
eth1 vlan 1 eth0 {};
eth2 vlan 2 eth0 {};
eth3 vlan 3 eth0 {};
```

Once you have edited this file reboot the machine. If you have configured inet.conf correctly you will have /dev/eth1, /dev/eth2, /dev/eth3, /dev/ip1, /dev/ip2 and /dev/ip3 in addition to the real interface on /dev/eth0 and /dev/ip0.
If your machine is named `minix05.cs.vu.nl`, then the three supplementary addresses should be called `minix05-0.cs.vu.nl`, `minix05-1.cs.vu.nl`, and `minix05-2.cs.vu.nl`
You need to configure the interfaces and add them to the routing tables so that they are properly routed. In order to do that you need to know some information. Execute the cneth command installed when you installed the libcn library which is installed in /usr/local/sbin. This command will provide the information you need to configure the interfaces you have added as well

as the information you need to set the environment variables required by the IP library. If you run it without the ETH variable set it will give you the information for all three available taps. Otherwise it will output the information for a single tap like so:

```
Information for additional ethernet tap 0:
        ETH = 0
        Ethernet address = 76:0:c0:23:c0:1a
        IP address = 192.35.192.51 / 255.255.255.0
        gateway = 192.35.192.1
```

To configure the interface you need to run the ifconfig command and two add_route commands in order to associate an address with the device and set it up in the routing tables on the machine. Based on the sample output above you would issue the following three commands to configure the interface and routing properly.

```
ifconfig -I/dev/ip1 -h 192.35.192.51 -n 255.255.255.0
add_route -i -I/dev/ip1 -g 0 -d 192.35.192.0/24 -m 1
add_route -I/dev/ip1 -g 192.35.192.1
```

You need to repeat these for each of the virtual taps you would like to use.

7.2 Required Environment Variables

The IP library requires several environment variables to function.

ETH The value of the ETH variable tells the IP library which IP address it should consider as its own. A value of `ETH=1` means the first supplementary address (e.g., `minixt05-1`), and a value of `ETH=2` means the second supplementary address (e.g., `minixt05-2`). If you set the `ETH` variable by hand, don't forget to export it, with the shell command `export ETH`.

IPGATEWAY This variable must contain the IP address of the gateway machine. This address is the same as your IP addresses, but with a 0 as the last number (e.g., `130.37.30.0`).

IPNETMASK This variable must contain the network mask.
For the VU setup, that is `255.255.0.0`.

7.3 IP Library Interface

```
int ip_init    (void);
int ip_send    (ipaddr_t dst, unsigned short proto,
                unsigned short id, void *data, int len);
int ip_receive (ipaddr_t *srcp, ipaddr_t *dstp,
                unsigned short *protop,
                unsigned short *idp, char **data);
extern ipaddr_t my_ipaddr;
```

ip_init() initializes the IP library.
It must be invoked once before any `ip_send()` or `ip_receive()` function is called.

ip_send() sends an IP packet. It takes as parameters the destination IP address, the IP higher-level protocol identifier (see RFC 762), the IP packet identification number, a memory chunk to send, and the number of bytes to send. It returns the number of bytes effectively sent, or -1 in case of an error.

Do not send packets larger that 8192 bytes (including the IP and TCP headers and the data): it would cause the IP layer to fragment packets, which is not very well supported.

ip_receive() waits until an IP packet is received. It sets the values of the source IP address, the destination IP address, the IP higher-level protocol number, the IP packet identification number and a memory chunk. *Attention:* `ip_receive` allocates a memory chunk with `malloc()` when a packet is received. You *must* `free()` it yourself as soon as the packet data is no longer needed. The function returns the number of bytes effectively received, or -1 in case of an error.

my_ipaddr contains the value of a program's own IP address. Its value is set automatically by the `ip_init()` function. Note `my_ipaddr` is a global variable, not a function.

You will find the contents of the IP library by looking at

- ip/ip.h;
- ip/libip.a;

You will find the Internet support routines in:

- include: /usr/local/include/cn/
- library: /usr/local/lib/libcn.a

The IP library is provided with two small example programs: `client.c` sends one IP packet and `server.c` receives it. Edit client.c to fit the IP addresses of your machine and verify that you can actually communicate before you start programming.

7.4 Compiling your programs

You must compile using `cc`. To create libraries, use `aal`.

8 Programming hints

Since this assignment classifies as a medium sized, you might benefit from some guidelines, on programming past the trivial.

- **Think before you code.**
  What does this code have to do? Then code it to do that and nothing more!
- **Don't use complex syntax like.**

```
for(;P("\n"),R--;P("|"))
  for(e=C;e--;P("_"+(*u++/8)%2))P("| "+(*u/4)%2);
```

- **Do not make big compounds.** In fact breakup anything between `{` and `}` that is over, say, 20 lines.
- **Do not optimize until it is really needed.** Concentrate on clean code, readable code. Do not save on functions calls do not do loop enrolling, etc.
- **Use a standard debugging habit.**

```
#ifdef DEBUG
# define dprint printf
#else
# define dprint (void)
#endif
```

  Then only use `dprint(...)` for debug output, like so:
  `dprint("Function foo has parameters %s and %d\n",foos,fooi);`

- **Don't be afraid to use status variables.** Use:

```
int found;
int i;

/* Look for arg that is ok.
 */
found = 0;
i = 0;
while (!found && i < N) {
  found = myok(arg[i]);
  i++;
}
```

```
/* If proper arg is found, do stuff to it.
 */
if (found) {
  dostuff(arg[i]);
}
```

and not:

```
int i;
for (i = 0; i < MAX; i++) {
  if (myok(arg[i]) {
    dostuff(arg[i]);
    break;
  }
}
```

or even worse:

```
argp *p = arg - 1;
while(++p < arg + MAX) if(myok(*p)) {dostuff(*p); break;}
```

- **Use proper indentation.** Like 2 or 4 spaces, do not use tabs in your code.
- **Read the man pages.** Especially of `assert`, `alarm` `make`, and `memcpy`.
- **Use** `assert()`**.**
- **Do not put "C-code" in header files.**
- **Do not use funny pre-processor stuff.** Don't do stuff like: `#define cat(a,b) a##b`
- **Don't redefine the truth.**

  Do not (under any circumstances) define `true` as 1. Code like this will print nothing (with most compilers)!

```
#define true 1
#define false 0

x = (4 == 4);
if (x == true) {
  dprint ("4 == 4\n");
}
```

or even worse:

```
typedef enum {false, true} boolean;
boolean b;
b = (i == i);
```

- **Use `{` and `}` on every `if`, `when`, `do`, and `for`.**
- **Use a C-beautifier.** Search for programs like `indent` (see the man page). There are many others available on the Internet.
- **Don't use global variables where you could use parameters.**
- **Don't use `malloc()` unless you have to.** If you do, put a debug counter in your malloc()/free() code like:

```
int _malloc_counter = 0;

void *my_malloc(size_t size\)
{
   _malloc_counter++;
   return malloc(size);
}

void my_free(void *mem)
{
   _malloc_counter--;
   free(mem);
}
```

  Then use: `dprint("Malloc counter is %d\n",_malloc_counter);` in your code where you know the values it should have. Or later on, use:
  `assert(_malloc_counter == 0);` or something like:
  `assert(_malloc_counter == n_blocks + n_fragments);` or whatever. Note, you don't have to use `malloc()` for your assignments.
- **Check return values from all library and system calls.** Do not do:

```
curr_date = strstr(buf, "Date");
if(sscanf(curr_date, "%*[^:]:%[^\n]", current_date)!=1)
{
   printf("Current date not set. [%s]\n", current_date);
   return -1;
}
```

  In fact, do not use `sscanf()`, `gets()`, at all!
- **Use names that have a large psychological distance.** Use (long) names like:

```
int my_first_index;
int my_last_index;
```

Do not use:

```
int indexf, indexl;
```

- **Do not be afraid to use long names and standard prefixes.** use: `i_block`, `n_blocks`, `max_block_size`, `t_block_fragment a_fragment[MAX_FRAGMENTS]` etc. Use `i_` for index, `a_` for array, `t_` for type, `g_` for global, `n_` for number, `max_` for maximum, `p_` for pointer, `my_` for rewrites of "known" functions etc. Or make up your own. But stick to them.
- **Use names constants.**

  Do not put numerical constants like `42` in your code (with exception of "0" and "1"). Use preprocessor defines:

```
#define MSG_MAX_SIZE      1024
#define MSG_ID_OFFSET        0
#define MSG_SIZE_OFFSET      6
#define MSG_COUNT_OFFSET     8
#define MSG_FLAGS_OFFSET    14
#define MSG_OK_FLAG       0x01
#define MSG_FRAGMENT_FLAG 0x02
#define MSG_LAST_FLAG     0x04
#define MSG_RESEND_FLAG   0x08

byte message[MSG_MAX_SIZE];
msg_get (message, MSG_MAX_SIZE);
if (message[MSG_FLAGS_OFFSET] & MSG_RESEND_FLAG) {
    msg_retry_fit(message);
}
```

- **Do not make scores of boolean functions.** Don't make things like this:

```
bool msg_resend_flag_set(byte flag)
{
    return (flag & MSG_RESEND_FLAG) == MSG_RESEND_FLAG);
}
```

  In fact, don't make any of this type.
- **Check your programs with `lint` or `gcc -Wall`.** It will of course not find all your bugs, but at least a few very common ones.

9 How to debug your program

Since this assignment classifies as a medium sized, you might benefit from some general debugging guidelines.
Finding a bug in your Computer Networks assignment, though sometimes difficult, is only the first step. Next you have to get rid of it. Here are a few things you should do as you crush those bugs:

- **Copy your program first.**
  Some bugs are really hard to get rid of. In fact, sometimes a little bug can drive you so crazy that, in the process of eradicating it, you destroy your entire program. Saving your program before you start debugging is the best way to ensure that a bug doesn't get the best of you. Version control like svn and cvs are your friend.
- **The golden rule for debugging programs.** Fix one bug at a time. If you know about several bugs, fix each one and test your fix before you move on to the next bug. Fixing a lot of bugs all at once without checking your work is just an invitation for even more bugs.
- **Lookout for inconsistencies.** Don't mixing bit logic coding with arithmetic coding. The statement `flags = FL_ACK << 8 + FL_SYN` probably doesn't do what you expect it to. Always change code like that to `flags = FL_ACK << 8 | FL_SYN` (the logic coding) or `flags = FL_ACK * 256 + FL_SYN` (the arithmetic coding). Do this with every code fragment like this you encounter, even if it doesn't seem to be the bug at hand. The best way to handle the flags in your TCP implementation is using parenthesis, shift and or, like so:
  ```
  flags = FLAG_ACK | (window_size << SHIFT_WINDOW);
  ```
- **Beware of voodoo coding.** Sometimes you know a bug exists, but you don't really know why. Lets say you have a variable called `index`, and for some reason `index` is always one less than you think it should be. You can do one of two things: sit there for a while and figure out why it's coming up short or just shrug, add one to index before using it, and move on. The latter method is called voodoo programming. When you start thinking "What the hell? why is index two instead of three? Well ... I'll just make it work for now and fix it later," you're slapping a Band-Aid on a potentially serious wound.

  Voodoo programming may work in the short term, but you're looking at long-term doom if you don't understand your code enough to really get rid of a bug, that bug will come back to haunt you. It will either return in the form of yet another weird error that you can't figure out, or it'll make your code extremely hard to understand for the poor soul that has to grade you.
- **Write a test program.** If a bug's appearance is inconsistent, try to write a test program that consistently demonstrates the bug. This test program will make debugging easier and you can easily test the fix (as you should).
- **Look for similar bugs.** In some ways, the ability to cut and paste code is the worst thing for programmers. Often, you'll write some C code in one function and then cut and paste it into another function. And if the first function had a problem, you now have problems

in two functions. I'm not saying you shouldn't cut and paste code. But bugs have a way of multiplying, so if you find one, you should look for others that are similar. (Or just make sure something works before you start creating multiple versions of it.) Misspelled variable names are the kind of thing that crops up several times in one C program. Misspell `the_name` as `the_mane` in one place, chances are you've done it someplace else, search for them.

- **Look for similar bugs II.** If you find a logical bug, like swapped source and destination parameters, find all occurrences of this type and check them.
- **Understand you code.** Sometimes if you just don't know where to start (anymore), look at your code line by line and ask your self "Do I really understand this line, and why it is, like it is?" If the answer is not "I understand every thing and I completely know why it is written this way!" find out the why, what and when. That is find out exactly why it is written like it is, exactly what is is doing and exactly when the code is executed, i.e., under what circumstances.
- **If all else fails...** If you're sitting there, staring at a bug, and you just can't figure out what's going on (or if you can't even find the bug in the first place, but you know it's there because your program isn't working right), the best thing to do is walk away from your computer. Go read a book, take a stroll around the corner, get a tasty beverage do something, anything, but don't think about the program or the problem. This technique is called "incubation" in some circles, and it works amazingly well. After you've had a little break and relaxed, try finding the bug again. You'll approach the problem with a refreshed vision that's often quite illuminating. Incubation works because it breaks you out of a (dare I say it?) disfunctional mind set. If you follow a wrong path for too long, you sometimes get stuck with no room in which to turn around. The best thing to do when this happens is find a way to start down a new path. I know it's touchy-feely, but it works. Really!
- **And only if all this doesn't work...** Ask your "begeleider" for help. Sometimes you wear such a rut in your problem with repetitious thought patterns, only someone else can see the hole in your "logic".
- **The absolute best way to eradicate bugs...** Create bug-free code from the get-go. Win before you fight!