Computer Networks Practical 2008-09
Documentation

Anant Narayanan (1824163 / ann340)
Pravin Shinde (1824368 / pse220)

Vrije Universiteit

February 17, 2009

1 Design

1.1 Data Structures

We began with an approach very similar to the method described in the handout itself. Our first step was to create the low level send/receive functions: `send_tcp_packet` and `recv_tcp_packet`, whose only responsibility was to generate a valid TCP from the data it is given and send/receive it. No guarantees about delivery are made at this layer.

At this very stage, we noted that the suggested function prototypes consisted of a long list of arguments. This approach seemed very cumbersome to use. Instead, we chose to create two primary data types for use throughout our library: a `Header` structure that would represent a TCP header (including the fields prescribed in the *pseudo-header*) and a `Data` structure that would represent any piece of data (sequence of bytes), along with it's length so we wouldn't have to worry about buffer overflows and could freely pass data around the different functions of our library. Also, the choice of using such data structures usually results in more readable and succinct code (provided the fields of the structures in question are named properly).

This, of course, meant that we would have to be clear about which parts of the structures are modified by which portions of our library. This was documented at the beginning of every function. Any `Data` structure is not modified by any function (when modifications are to be made, a copy is made first).

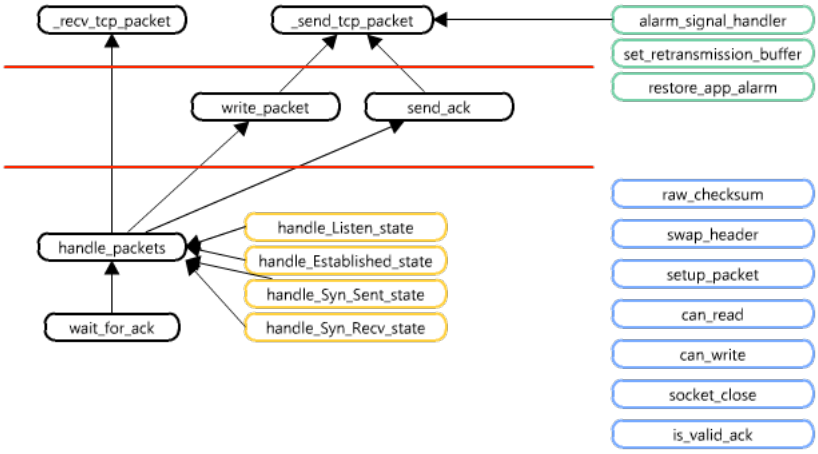
Important constant values were encoded in an `enum`. Constituents of this enumeration were flag values (like `URG` and `FIN`) and states (like `Closed` and `Fin_Wait2`). Using readable constants makes code, well, more readable (as already mentioned in the handbook).

Very early on in the implementation, we had made the decision to code with the intention of supporting multiple TCP connections within one program. Thus, we incorporated another data structure, the `TCPCtl` to represent a single TCP connection. This structure would store information such as the socket number, the source port, the destination, destination port, and so on (again, as indicated by the handbook). Initially, we created a global variable representing the current (single) connection (a `TCPCtl`) - later sections will describe how we extended this to support multiple connections.

1.2 Layers

As recommended by the handbook, using a 3-tiered design would result in a more readable and flexible code base. Hence, we decided to use a 3 layer approach. These connection oriented functions were split into two portions, one that cared about state, and one that didn't.

The following figure demonstrates how our functions are split across the layers (and those that don't fit in any - mainly utility and helper functions):



The public interface functions are skipped from the above figure, but they also belong to the layer which cares about state. These public functions merely set state variables of a connection and let `handle_packets` interpret the rest. This layered design helped greatly in keeping our code as modular as possible. Also, because the interfaces between the layers are clearly defined, it was possible to split the work between the partners easily.

2 Features

For implementing the rest of the library, the approach described in the handbook was followed. In most cases we stuck to the expected behavior. However, there are some caveats (read "features") in our library which are noted in this section.

2.1 Reliable Write

While the general expectation is that `tcp_write` will return as soon as possible, with writing only as many bytes as possible in a short time period, we decided to make our code deliver whatever data is provided, reliably.

Hence our implementation of `tcp_write` will always have a return value of the number of bytes requested to be written (or an error), no matter how long it takes. We implemented this "feature" because almost in all use cases we required this behavior, especially in the HTTP portion of the assignment. Instead of writing our own wrapper function that used `tcp_write` to deliver all of some data, we changed the underlying TCP code itself.

While this approach makes it convenient for users of the library most of the times (all of the time in our test programs, and HTTP implementation), it can also prove to be a serious pitfall. One of the major disadvantages of this approach is that `tcp_write` will block forever if there is no corresponding `tcp_read` called on the other end. Not only does `tcp_read` have to called on the other end, it has to be called enough times to exhaust all of the data that was requested to be written via `tcp_write`.

It is easy to switch behavior to it's former requirement: it is achieved by simply uncommenting one line in `tcp.c`. We considered creating a global variable that would control this, however, because of the additional public interface requirements in the test suite, decided against it.

2.2 Memory Management

We tried to minimize the use of dynamic memory as much as possible. However, we still do make use of it in the following places (listed along with where the memory is freed):

- **_send_tcp_packet**: Temporary memory area allocated in order to store all the data required for the checksum calculation. Calculation of the checksum is easiest when all data elements are in network endian order and in a contiguous memory location. This memory is freed immediately after the checksum is calculated.

- **_recv_tcp_packet**: Once again, in order to verify the checksum (in network endian order), we allocate some memory for temporary use (freed at the end of the operation). Here, we also allocate the contents of a `Data` structure that will hold the body of the TCP packet received. This memory is freed by *handle_packets* after the user has safely copied the data. We also free the memory allocated by the IP library's `ip_receive` method here.

- **tcp_socket**: The incoming and outgoing `Data` buffers for a new TCP connection are allocated here. This memory is freed by `socket_close` when this particular TCP connection is no longer needed.

Because of minimal dynamic memory allocation, we can reasonably confident that our code is free of memory leaks and possible invalid memory accesses.

2.3 Sequence Numbers

Three main points are to be covered on the topic of sequence numbers:

- **Increment for FIN?**: RFC 793 specifies (ref: Page 39, Fig 14) that SYN and FIN packets are treated as packets containing 1 byte of data, and thus, the sequence number is to be increased upon sending such a packet. However, we found that, in doing this, we were deemed incompatible with other TCP implementations as reported by the test suite. The reference TCP implementation does not increment it's sequence number after sending a FIN packet, and we decided to follow the same behavior (in the interest of passing all tests!).
- **Initial Allocation**: We use the `rand` system call in order to generate an initial sequence number. We restrict the number to modulo 1024 (an arbitrary limit, can be any reasonably small number), however, because we do not want the sequence number to be too large. This is because our code handles wrap-around on sequence numbers (explained in the next point).
- **Wrap-around Behavior**: In cases where a large amount of data is transmitted over a single TCP connection, it is possible that sequence numbers wrap-around, because they are 32bit integers. This case is handled (among others) in our `is_valid_ack` function. It is easy to detect a wrap-around, because we are expecting an ACK number greater than that we sent, but receive a number much lesser that the number of bytes we have already sent (and were ACKed).

2.4 Multiple Connection Support

As mentioned before, our code supports creation of multiple TCP connections. We modeled our interface to be as close to existing 'socket' behavior on UNIX systems. We return a unique 'socket number' whenever a new connection is created using the `tcp_socket` function. This number will uniquely identify that TCP connection throughout the subsequent session.

In order to ensure backward compatibility with single connection support required by the original assignment, we simply created a new set of interface functions: `tcp_connect_socket`, `tcp_listen_socket`, `tcp_read_socket`, `tcp_write_socket` and `tcp_close_socket`. Their behavior is exactly the same as the single connection counterparts, except that the application can choose which TCP connection to apply the operation to by specifying it's unique socket number.

During implementation, our initial approach was to create a linked list consisting of `TCPCtl` nodes, each representing a single connection. Each node would additionally store the socket number of that connection. This method would scale well, but proved to be a little cumbersome to implement. Creation and closing of sockets would result in several memory management functions and manipulation of the linked list, which provides a haven for sneaky memory related bugs.

Thus, we chose to adopt a much more simpler approach: we simply create a static array of 256 (defined as `MAX_CONN`) `TCPCtl` structures. The index of the array represents

the unique socket number for that connection. We also maintain a global variable representing the number of currently active connections, so we know what socket number is to be returned on a subsequent call to `tcp_socket`. However, this number is NOT decremented when a socket is closed - the implications of this are discussed in the 'Known Limitations' section.

Overall, we are very pleased with this simple interface and implementation of multiple connections. It provides us the optimum trade-off between code complexity and feature-set. While there are obvious pitfalls with our implementation, they work great for a majority of use cases while retaining a close similarity to the single connection version. Only minimal changes were required when we made the switch from single-connections to multiple-connections: since all functions already expected the global variable `cc` of type `TCPCtl` to represent the current active connection, we only had to set the pointer to an appropriate value (address at the index, which is also the socket number, of the global connection array) at the beginning of every function. Also since only one `tcp_write` is active at any given time (it is a blocking function), only one retransmission buffer is required and alarms are set and handled only once per active connection - none of that code needed any changes to support multiple connections.

The first connection to be created will always have a socket number of 0. Subsequent calls to `tcp_socket` will return socket numbers in incremental order, upto `MAX_CONN`. The single connection interface functions are are simple as calling their '_socket' counterparts with a socket value of 0.

2.5 Port Allocation

We perform port allocation for new TCP connections using a simple global counter. The `START_PORT` constant defines the first port number to be allocated, and subsequent TCP connections are simply assigned a port that is the sum of `START_PORT` and it's socket number. Additionally, we also perform a check in `tcp_listen` to ensure that the port number requested by the user is not already in use by iterating over all the array entries of currently active connections.

2.6 Honoring Window Size

Sometimes when a receiver's buffer is full, it will advertise a window size of 0. The sender is supposed to back off and send data only after waiting for some amount of time. Instead of implementing another alarm (complicated and prone to errors), in order to retransmit after some time of receiving a window size of 0; we simply set the remote window size to 1 byte instead. Thus, the sender will continue to send data packets of 1 byte each, which may or may not be dropped by the receiver depending on if the buffer has more space by then. This approach provides a fine balance between code complexity and behaving 'nice' by not congesting the network too much, while honoring the advertised to a large extent.

2.7 Strictness

Our code is reasonably strict about the types of packets it receives, but not in a way that will disrupt the connection. Since we are not required to implement connection resets (`RST`), we simply ignore any packets that do not match our expectations and log the occurrence. An example of this is receiving a data packet with the `SYN` flag set when in an Established state. We will ignore this packet even though the rest of the packet has perfectly valid data.

2.8 Half-closed Connections

Our code supports half-closed connections, i.e. a client may continue to read out of a connection even after calling `tcp_close` on it. This is very similar to the `shutdown` call in the context of UNIX sockets - closing a socket is only a promise not to write any more data, and doesn't specify anything about reading data out of it. Hence, no memory is released in the `tcp_close` function, but only after we have ensured that both parties do not intend to read or write on that connection.

Incidentally, our code also support simultaneous closes (and opens), as required by the RFC.

3 Testing

We performed extensive testing on our code by writing our own test programs. We implemented dropping and corruption of packets using global variables (`extern int`) that indicate the low level send function to do the action. This code is not included in the final submission because of interface requirements, because our tests also depend on the low level functions being non-static - hence some of the test programs will not run with the submitted code! Following is a list of test programs and a brief description of what they do:

- tcp_ack_drop: Drops an ACK packet and checks if we handle the case correctly.
- tcp_actual_chksum: Tests if our code calculates the checksum of a packer correctly.
- tcp_alarm_test: Tests if the library restores signal handlers set by the application before returning.
- tcp_checksum_test: Tests if corrupted packets are handled correctly.
- tcp_chop: Checks if differing send and receive sizes are handled correctly (also tests the buffers on both ends extensively).
- tcp_close_parallel: Tests if our code supports parallel connections.
- tcp_connect/tcp_listen: Tests simple connect and listen functionality.
- tcp_connect_parallel: Tests if we handle simultaneous 3-way handshakes correctly.

- tcp_listen_multiple: Tests multiple connection support.
- tcp_read_after_close: Tests if we support half-closed connections correctly.
- tcp_syn_ack_drop: Tests if we handle SYN+ACK drops correctly.
- tcp_wrap_around: Tests if we handle wrap-around of very large sequence numbers correctly.

We also have two small tests for testing the HTTP client and server.

4 Known Limitations

- As mentioned before, the global active connection count is not decremented when a connection is closed. This may lead to 'holes' in the static array of connections, because connections can be closed in any order. Thus, the number of actually active connections may be less than `MAX_CONN`. Our only guarantee is that `MAX_CONN` connections may be *created* by an application. This does not affect port number allocation though, because all port numbers for a particular connection are set to 0 whenever it is closed.
- In the case of multiple connections, it is possible for a deadlock to occur. Because our `tcp_read` code ignores all packets that are not meant for the currently active connection, calling `tcp_read` and `tcp_write` in differing orders on two ends of a multi-connection program will result in a permanent deadlock as all packets sent by one end will be ignored by the other until the other end gives up.

5 HTTP

Some useful details about our HTTP implementation:

- MIME types are simply determined using file extensions in our server. This may not be correct all the time as extensions may be misleading.
- Our HTTP server supports a maximum file path length of 256 characters.
- We perform hostname to IP translation using `gethostbyname` if an IP is not provided to the HTTP client.
- Any unknown headers are gracefully handled by both the server and client.

Our HTTP implementation is extremely simple (only a couple of hundred lines), and thus, we believe it does not require much explanation. There are no major design issues tackled in this part of the assignment. Code common to both the server and client is included in the `http.c` file, which contains utility functions mainly for header parsing and reliable reading (simply a loop that calls `tcp_read` until all data requested is read).