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).

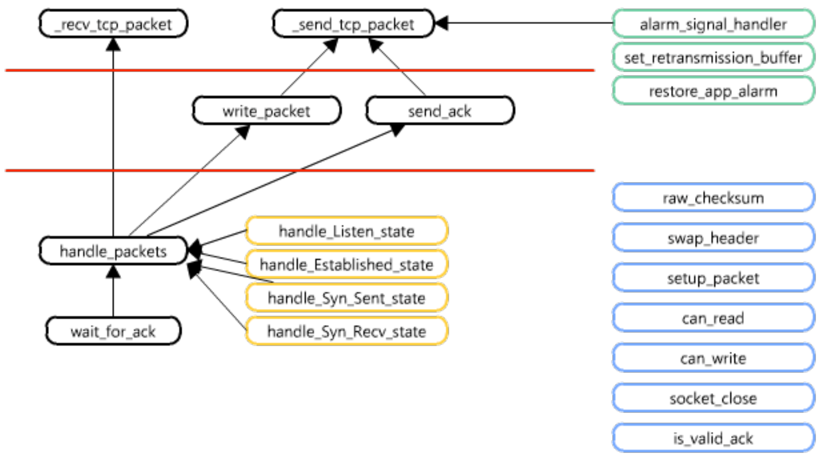
Important constant values were encoded in an `enum`. Constituents of this enumeration were flag values (like `URG` and `FIN`) and states (like `Closed` and `Fin_Wait2`). Using readable constants makes code, well, more readable (as already mentioned in the handbook).

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 Implementation

2.1 Interface methods

The approach described in the handbook was followed. No major departures from design decisions were made.

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.
- **send_ack**: WTF?

3 Testing

4 Challenges

5 Known Limitations

6 HTTP