CS 456 Assignment 1
Design Documentation

Siwei Yang

June 21, 2013

1 Overview

The goal of this assigment is to implement programs that can reliably transfer data over unreliable communication channels. The reliable transfer protocol is designed with client-server model for the sake of simplicity. Both flavor of the protocol, namely `Go-Back-N` and `Selective Repeat`, are provided for inspection.

The programs delivered are configured to transfer a file from client side to server side over a provided, most likely unreliable, channel to demonstrate the correctness of the protocol, and performance evaluation. But the instruments and modules developed for the programs can be packaged with other programs for rapid adoption of the reliable transfer protocol.

2 Design Philosophy

2.1 Layered Design

The complexity of this program lies in the mixture of dispatch/processing logic and transmission facility. So the strategy of layered design is applied where the transmission is taken care by a data structure:

```
struct Stack {
  unsigned int size;
  unsigned int window_low;
  unsigned int window_high;
  Packet * packets;
  unsigned int * timeouts;
};
typedef struct Stack Stack;
```

Thus, the program can sequantially process dispatching and transmission in different spaces.

2.2 State Centric

To manage transmission effectively, a suit of the operations are created with a focus on the state of transmission. For instance:

```
Stack updateStackWindow (Stack stack, Packet p) {
  if (p.sn < stack.window_low || p.sn > stack.window_high || p.sn > stack.size)

  if (p.pt == DAT) {
    // receiver can choose to not have a sending window
    /*
#ifdef GBN
    unsigned int adv = p.sn - stack.window_low + 1;
    stack.window_low += adv;
    stack.window_high += adv;

#endif

#ifdef SR
    stack = updateStackWindow(stack);
#endif
    */
  }

  if (p.pt == ACK) {
#ifdef GBN
    unsigned int adv = p.sn - stack.window_low + 1;
    stack.window_low += adv;
    stack.window_high += adv;
#endif

#ifdef SR
    stack = passiveUpdateStackWindow(stack);
#endif
  }

  return stack;
}
```

The method operate the transmission stack on receiving a new packet as if it operates a state machine. Thus, cleanly encapsulate the inner working of the

stack. Let the user interact with the stack only based on its state.

2.3 Self-managing Data Structure

Related to the state machine centric approach is the adoption of self-managing data structure. This is usually taken care of by container types in the standard library. However, C is no normal language in which the programmer has to spend effort creating self-managing data structure:

```
Stack expandStack (Stack stack, unsigned int margin) {
  unsigned int size = stack.size + margin;
  printf("Expanding Stack by size of %d\n", margin);
  Stack s = createStack(size);
  s.window_low = stack.window_low;
  s.window_high = stack.window_high;

  memcpy(s.packets, stack.packets, stack.size * sizeof(Packet));
  memcpy(s.timeouts, stack.timeouts, stack.size * sizeof(unsigned int));

  return s;
}
```

The importance of self-managing data structure is often over looked. But here it is key to support potentially unlimited state created while receiving packets.

3 Testing and Quality Assurance

3.1 Black Box Testing

The program has been tested with two black box testing techniques: random data attack and integration testing. The data attack concerns the robustness of the program: will the program crash on any possible input? Not only out of order packet but also random message that does not deserialize to packet are thrown at both the client and the server to test their robustness. And integration testing is simply transferring data files, big and small, with a check on file content after. Different timeout configurations are tested on each cases.

3.2 White Box Assurance

While black-box testing concerns program behavior on typical use case, to provide quality assurance, analysis of program correctness has to be done. To help with that, before anything else, the program is written with defensive programming in mind and trades performance with provability sometime:

```
unsigned int timeoutStackWindow(Stack stack) {
  //printf("Find Free Packet between %d and %d\n", stack.window_low, stack.wind
  unsigned int timeout = UINT_MAX;
  unsigned int cursor = stack.window_low;
  while (cursor <= stack.window_high && cursor <= stack.size) {
    Packet * p_ptr = stack.packets+cursor;
    unsigned int t = *(stack.timeouts+cursor);
    //printf("Packet %d timeout at %d\n", p_ptr->sn, t);
    if ((p_ptr->pt == DAT || p_ptr->pt == EOT) && t > 0 && t < timeout)timeout
    cursor++;
  }
  printf("Next timeout is %d\n", timeout);

  if (timeout == UINT_MAX)failure("Requesting next timeout without active trans
  return timeout;
}
```

This method and the previous one for expanding stack are written in such way that they are easy to be proven correct. This solid guarantee on correctness reduce the workload for unit testing as well. The rest if unit testing on various methods operate with stack. After making sure the stack is working up to its state machine design, the author is fairly confident that the program will behave correctly on all cases.