CSCI 415 Computer Networks
Homework 2
Solution

Saad Mneimneh
Computer Science
Hunter College of CUNY

Problem 1
Consider the following server and client C++ code that we saw in class:

server.c

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;
const int max_pending = 10;
const int max_len = 256;

int main() {

  sockaddr_in address; //address
  sockaddr_in client_address; //client address
  char message[max_len];

  int s;
  int new_s;
  int len;

  //build address
  memset(&address, 0, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = htonl(INADDR_ANY);
  address.sin_port = htons(port);

  //setup passive open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }
```

```
  //bind socket to address
  if (bind(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in bind";
    return 0;
  }

  if (listen(s, max_pending) < 0) {
    cout<<"error in listen";
    return 0;
  }

  //wait for connection, then receive message
  socklen_t size = sizeof(sockaddr_in);
  while (1) {
    if ((new_s = accept(s, (sockaddr *)&client_address, &size)) < 0) {
      cout<<"error in accept";
      return 0;
    }

    while (len = recv(new_s, message, sizeof(message), 0)) {
      cout<<message<<"\n";
    }
    close(new_s);
  }
}
```

client.c

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;

int main() {
  int s;
  sockaddr_in address; //address to connect to

  memset(&address, 0, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_port = htons(port);
  address.sin_addr.s_addr = htonl(INADDR_ANY); //my IP address

  //active open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }
```

```
  //connect
  if (connect(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in connect";
    close(s);
    return 0;
  }

  char message[256];
  while(cin.getline(message, 256, '\n')) {
    if (strlen(message) == 0)
      break;
    send(s,message,strlen(message)+1,0);
  }
  close(s);
  return 0;
}
```

(a) Obtain the code for both the server and the client (you can cut and paste from pdf) and compile it using a C++ compiler (e.g. g++ on Unix). Start one server and one client in separate windows. Start the server first; otherwise, the client will not be able to connect to the server and will report an error and exit. The client should accept messages from the keyboard (you) and sends them to the server. The server just echoes what you send. To exit the client just type an empty message (hit Enter).

(b) While the client is running, start 10 other clients. What happens to these clients? Do their connect()s fail, or time out, or succeed? Do any other calls block? Now let the first client exit. What happens? Try this with the server value max_pending set to 1 instead.

ANSWER: Generally, the server will queue max_pending clients for acceptance, and will accept messages only from the first client. The other clients will be ignored, they will eventually time out. You probably cannot see this if you don't allow enough time; for instance, have the client send messages indefinitely, then abort the first few clients. The rest of the clients won't succeed.
After the active client exits, the next one start communicating with the server. All messages previously sent by that client, however, are ignored, except for the first one, which I think is sent in the ACK packet (but I need to do more research on this issue).

(c) Modify the server and the client such that each time the client sends a line to the server, the server sends the line back to the client. The client (and server) will now have to make alternating calls to recv() and send().

ANSWER:
server.c

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
```

```
using std::cin;

const unsigned short int port = 5432;
const int max_pending = 1;
const int max_len = 256;

int main() {

  sockaddr_in address; //address
  sockaddr_in client_address; //client address
  char message[max_len];

  int s;
  int new_s;
  int len;

  //build address
  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = htonl(INADDR_ANY);
  address.sin_port = htons(port);

  //setup passive open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }

  //bind socket to address
  if (bind(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in bind";
    return 0;
  }

  if (listen(s, max_pending) < 0) {
    cout<<"error in listen";
    return 0;
  }

  //wait for connection, then receive message

  socklen_t size=sizeof(sockaddr_in);
  while (1) {
    if ((new_s = accept(s, (sockaddr *)&client_address, &size)) < 0) {
      cout<<"error in accept";
      return 0;
    }

    while (len = recv(new_s, message, sizeof(message), 0)) {
      cout<<message<<"\n";
      send(new_s,message,strlen(message)+1,0);
    }
```

```
    close(new_s);
  }
}
```

client.c

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;

int main() {
  int s;
  sockaddr_in address;

  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_port = htons(port);
  address.sin_addr.s_addr = htonl(INADDR_ANY); //my IP address


  //active open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }

  //connect
  if (connect(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in connect";
    close(s);
    return 0;
  }

  char message[256];
  while(cin.getline(message, 256, '\n')) {
    if (strlen(message) == 0)
      break;
    send(s,message,strlen(message)+1,0);
    int len = recv(s, message, sizeof(message), 0);
     cout<<message<<"\n";
  }
  close(s);
}
```

Now when a client exits and a next one starts communicating with the server, you will notice that all messages are being delivered. That's because the client waits to receive a message from the server before sending the next one.

(d) Modify the server and client so that they use UDP as the transport protocol, rather than TCP. You will have to change SOCK_STREAM to SOCK_DGRAM in both client and server. Then, in the server, remove the calls to listen() and accept(), and replace the two nested loops at the end with a single loop that calls recv() with socket **s**. Finally, see what happens when two such UDP clients simultaneously connect to the same UDP server, and compare this to the TCP behavior.

ANSWER:
server.c

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;
const int max_pending = 1;
const int max_len = 256;

int main() {

  sockaddr_in address; //address
  sockaddr_in client_address; //client address
  char message[max_len];

  int s;
  int new_s;
  int len;

  //build address
  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = htonl(INADDR_ANY);
  address.sin_port = htons(port);

  //setup passive open
  if ((s=socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }

  //bind socket to address
  if (bind(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in bind";
    return 0;
```

```
  }

  while (len = recv(s, message, sizeof(message), 0)) {
    cout<<message<<"\n";
  }

}
```

client.c

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;

int main() {
  int s;
  sockaddr_in address;

  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_port = htons(port);
  address.sin_addr.s_addr = htonl(INADDR_ANY); //my IP address


  //active open
  if ((s=socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }

  //connect
  if (connect(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in connect";
    close(s);
    return 0;
  }

  char message[256];
  while(cin.getline(message, 256, '\n')) {
    if (strlen(message) == 0)
      break;
    send(s,message,strlen(message)+1,0);
  }
  close(s);
}
```

While TCP requires an advance connection, UDP accepts a packet of data from any source at any time. Thus, two clients can now talk simultaneously; their messages will be interleaved on the server.

Problem 2: Framing by doubling
Consider the following approach for framing. Each bit in the original packet is doubled, and the frame is ended with 01. For example, if the original string of bits is:

011100

then the frame will be:

00111111000001

The problem with this technique is that we've just doubled the length of every packet. Using a combination of this technique and a header, suggest a way to reduce the amount of doubling.

ANSWER: One way is to use a variable size length field, and use doubling only on the length field to determine where it ends. For instance, in the above example, the length is 6, i.e. 110. We will have:

11110001011100

Of course, we don't see a reduction in the size of the frame for this particular example, but if the message is longer, doubling the header is more economical than doubling the entire message.

The same idea can be used recursively. We can use another length field to denote how big the length field is, and double only that. For instance, the length field 110 has length 3, which is 11.

111101110011100

Again, the advantage of this will be seen if the message is long enough. So the idea is the following: add more and more headers, each of which tells the size of the next header, and only the first one is doubled.

Problem 3

(a) Apply the bit stuffing rules we discussed in class to the following frame:

01101111100111111010111111111101111010

ANSWER: The stuffed zeros are shown in bold.

011011111**0**0011111**0**101011111**0**11111**0**01111010

(b) Suppose the following string of bits is received:

0111111011111011001111100111110111110110001111111010111110

Remove the stuffed bits and show where the actual flags are.

ANSWER: A dot indicates a deleted 0. The flags are shown in bold.

0111111011111.110011111.011111.11111.1100**01111110**1011111.

(c) Suppose that the bit stuffing rule, which is set to stuff a zero after the occurrence of 5 consecutive 1's, is modified to stuff a 0 only after the appearance of 01^5 in the original data. Carefully describe how the destuffing rule at the receiver must be modified to work with this change. Show how you would destuff the following string:

0110111110111111011111010111110

If your destuffing rule is correct, you should remove only two 0's and find only one actual flag.

ANSWER: The modified destuffing rule starts at the beginning of the string and destuffs bit by bit. A zero is removed from the string if the previous six bits in the already destuffed portion of the string have the value 01^5. For the particular example:

011011111.111111011111.1**01111110**

Problem 4
Errors can cause the flag 01^60 to appear within a transmitted frame. In this problem, you are asked to show that the expected number of such flags is $(1/32)Kp$, where

- K is the size of the frame before stuffing
- p is the probability of bit error (binary symmetric channel)

Assume that the bits of the original frame are IID (independent and identically distributed) with equal probability of 0 and 1.

You can choose to do this problem either by simulation or analytically.

Simulation

1. Generate a large enough random string of bits (that's your original frame), e.g. 10000 bits
2. Apply the bit stuffing rule to obtain a new string of bits (at this point, the flag 01^60 does not appear)
3. To simplify the stuffing operation, stuff while generating the bits
4. Introduce error, i.e. flip each bit with a probability p
5. Count the number of times the flag 01^60 appears
6. Repeat the experiment many times and find the average of your count

Analytically

This is a bit tricky since the bits after stuffing are no longer IID. So analyze the stuffed bits and the original bits separately.

- Find the probability that a stuffed bit in error causes a flag (i.e. 01111110) to appear, and then find the expected number of flags created in this way, simply by multiplying this probability by p (the probability of error) and by the expected number of stuffed bits ($K2^{-6}$ as argued in class).
- Find the probability of a flag appearing due to errors in the original bits of the frame. *Hint*: Only a 0 flipping to 1 can cause the appearance of the flag. A 1 flipping to a 0 cannot (try it). So given that a data bit is 0, what is the probability that an error will cause the appearance of the flag? Once you find this probability, you need to multiply it by 1/2 (the probability that the bit is indeed 0) and by p (the probability that there is in deed an error), and finally by K to find the expected number of flags created in this way.
- Add both numbers, you should get $(1/32)Kp$

ANSWER: This is the analytical solution. First, a stuffed zero in error (i.e. it is actually received as 1) will cause the appearance of a flag only if it is followed by a zero (011111**0**0 perceived as 011111**1**0). That's a probability of 1/2. We know that the expected number of stuffed zeros is $K2^{-6}$. Therefore, the expected number of flags due to an error in a stuffed bit is $K2^{-6} \times 1/2 \times p = Kp \times 2^{-7}$.

Second let us look at the original bits. We argue that we only need to worry about the zeros in the original data. A bit that flips from 1 to 0 due to error cannot cause the flag to appear. That's because this bit has to be the start or the end of the flag, and therefore, we have to have either **1**1111110 or 0111111**1** in the original data (with the 1 in bold being the bit of interest). In either case, a zero is stuffed after the 5^{th} 1 and the pattern it avoided.

A bit that flips from 0 to 1 due to error can cause the flag to appear. Let's look at all possible cases for this 0 bit (shown in bold and is assumed to flip to a 1).

						↓						
x	x	x	x	x	0	**0**	1	1	1	1	1	x
x	x	x	x	0	1	**0**	1	1	1	1	0	x
x	x	x	0	1	1	**0**	1	1	1	0	x	x
x	x	0	1	1	1	**0**	1	1	0	x	x	x
x	0	1	1	1	1	**0**	1	0	x	x	x	x
0	1	1	1	1	1	**0**	0	x	x	x	x	x

The last case can be eliminated because because a zero will be stuffed in front of the erroneous bit, and therefore, the flag cannot be created due to the error.

					↓						
x	x	x	x	0	**0**	1	1	1	1	1	x
x	x	x	0	1	**0**	1	1	1	1	0	x
x	x	0	1	1	**0**	1	1	1	0	x	x
x	0	1	1	1	**0**	1	1	0	x	x	x
0	1	1	1	1	**0**	1	0	x	x	x	x

The first case is slightly different than the rest of the four cases in that the last bit can be either 0 or a 1 (instead of being required to be zero). The reason for this is that bit stuffing will stuff the required zero.

Each of the four cases have a probability of $\frac{2^4}{2^{12}} = 2^{-8}$, and the first case has a probability of $\frac{2^5}{2^{12}} = 2^{-7}$. Therefore, the probability that the flag will appear is $(4 \times 2^{-8} + 2^{-7})p = (2 \times 2^{-7} + 2^{-7})p = 3 \times 2^{-7}p$. The expected number of such flags is $3Kp \times 2^{-7}$.

Adding both we get $4Kp \times 2^{-7} = (1/32)Kp$.