CSS 432
Program 1: Sockets

Professor Dimpsey

---

1. Purpose

This assignment is intended for two purposes: (1) to exercise use of various socket-related system calls and (2) to evaluate dominant overheads of point-to-point communication over 1Gbps networks.

2. Client-Server Model

In all your programming assignments through to the final project, your program will use the client-server model where a client process establishes a connection to a server, sends data or requests, and closes the connection while the server sends back responses or acknowledgments to the client.

3. TCP Communication

HW1 focuses on basic connection-oriented socket (SOCK_STREAM) communication between a client and a server process. To establish such communication, those processes should perform the following sequence of operations:

Client

1. Receive a server's IP port (**server_port**) and IP name (**server_name**) as Linux shell command arguments.
2. Retrieve a **hostent** structure corresponding to this IP name by calling **gethostname()**.

```
struct hostent* host = gethostbyname(server_name);
```

3. Declare a **sockaddr_in** structure, zero-initialize it by calling **bzero**, and set its data members as follows:

```
int port = YOUR_ID;  // the last 5 digits of your student id
sockaddr_in sendSockAddr;
bzero((char*)&sendSockAddr, sizeof(sendSockAddr));
sendSockAddr.sin_family = AF_INET; // Address Family Internet
sendSockAddr.sin_addr.s_addr =
  inet_addr(inet_ntoa(*(struct in_addr*)*host->h_addr_list));
sendSockAddr.sin_port = htons(server_port);
```

4. Open a stream-oriented socket with the Internet address family.

```
int clientSd = socket(AF_INET, SOCK_STREAM, 0);
```

5. Connect this socket to the server by calling **connect** as passing the following arguments: the socket descriptor, the sockaddr_in structure defined above, and its data size (obtained from the **sizeof** function).

```
connect(clientSd, (sockaddr*) &sendSockAddr, sizeof(sendSockAddr));
```

6. Use the **write** or **writev** system call to send data.
7. Use the **read** system call to receive a response from the server.
8. Close the socket by calling **close**.

```
close(clientSd);
```

Server

1. Declare a **sockaddr_in** structure, zero-initialize it by calling **bzero**, and set its data members as follows:

```
int port = YOUR_ID;  // the last 5 digits of your student id
sockaddr_in acceptSockAddr;
bzero((char*)&acceptSockAddr, sizeof(acceptSockAddr));
acceptSockAddr.sin_family = AF_INET; // Address Family Internet
acceptSockAddr.sin_addr.s_addr = htonl(INADDR_ANY);
acceptSockAddr.sin_port = htons(port);
```

2. Open a stream-oriented socket with the Internet address family.

```
int serverSd = socket(AF_INET, SOCK_STREAM, 0);
```

3. Set the SO_REUSEADDR option. **(Note this option is useful to prompt OS to release the server port as soon as your server process is terminated.)**

```
const int on = 1;
setsockopt(serverSd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(int));
```

4. Bind this socket to its local address by calling **bind** as passing the following arguments: the socket descriptor, the sockaddr_in structure defined above, and its data size.

```
bind(serverSd, (sockaddr*) &acceptSockAddr, sizeof(acceptSockAddr));
```

5. Instructs the operating system to listen to up to five connection requests from clients at a time by calling **listen**.

```
listen(serverSd, 5);
```

6. Receive a request from a client by calling **accept** that will return a new socket specific to this connection request.

```
sockaddr_in newSockAddr;
socklen_t newSockAddrSize = sizeof(newSockAddr);
int newSd = accept(serverSd, (sockaddr *)&newSockAddr, &newSockAddrSize);
```

7. Use the **read** system call to receive data from the client. (Use newSd but not serverSd in the above code example.)
8. Use the **write** system call to send back a response to the client. (Use newSd but not serverSd in the above code example.)
9. Close the socket by calling **close**.

```
close(newSd);
```

You need to include the following header files so as to call these OS functions:

```
#include <sys/types.h>    // socket, bind
#include <sys/socket.h>   // socket, bind, listen, inet_ntoa
#include <netinet/in.h>   // htonl, htons, inet_ntoa
#include <arpa/inet.h>    // inet_ntoa
#include <netdb.h>        // gethostbyname
#include <unistd.h>       // read, write, close
#include <string.h>       // bzero
#include <netinet/tcp.h>  // SO_REUSEADDR
#include <sys/uio.h>      // writev
```

4. Statement of Work

Write Client.cpp and Server.cpp that establish a TCP connection between a pair of client and server, repeat sending a set of data buffers, (each with the same size) in three different scenarios from the client to the server, and send back an acknowledgment from the server to the client.

Client.cpp

Your client program must receive the following six arguments:

1. **port**: a server IP port
2. **repetition**: the repetition of sending a set of data buffers
3. **nbufs**: the number of data buffers
4. **bufsize**: the size of each data buffer (in bytes)
5. **serverIp**: a server IP name
6. **type**: the type of transfer scenario: 1, 2, or 3 (see below)

From the above parameters, you need to allocate data buffers below:

```
char databuf[nbufs][bufsize]; // where nbufs * bufsize = 1500
```

The three transfer scenarios are:

1. **multiple writes**: invokes the write( ) system call for each data buffer, thus resulting in calling as many write( )s as the number of data buffers, (i.e., **nbufs**).

```
for (int j = 0; j < nbufs; j++)
{
  write(clientSd, databuf[j], bufsize);    // clientSd: socket descriptor
}
```

2. **writev**: allocates an array of **iovec** data structures, each having its ***iov_base** field point to a different data buffer as well as storing the buffer size in its **iov_len** field; and thereafter calls writev( ) to send all data buffers at once.

```
struct iovec vector[nbufs];
for (int j = 0; j < nbufs; j++)
{
   vector[j].iov_base = databuf[j];
   vector[j].iov_len = bufsize;
}
writev(clientSd, vector, nbufs);           // clientSd: socket descriptor
```

3. **single write**: allocates an **nbufs**-sized array of data buffers, and thereafter calls write() to send this array, (i.e., all data buffers) at once.

```
write(ClientSd, databuf, nbufs * bufsize); // sd: socket descriptor
```

The client program should execute the following sequence of code:

1. Open a new socket and establish a connection to a server.
2. Allocate **databuf[nbufs][bufsize]**.
3. Start a timer by calling **gettimeofday**.
4. Repeat the **repetition** times of data transfers, based on **type** such as **1: multiple writes**, **2: writev**, or **3: single write**
5. Receive from the server an integer acknowledgment that shows how many times the server called read().
6. Stop the timer by calling **gettimeofday**, where stop - start = round-trip time.
7. Print out the statistics as shown below:

```
Test #: round-trip time = yyy usec, #reads = zzz
```

8. Close the socket.

Server

Your server program must receive the following two arguments:

1. **port**: a server IP port
2. **repetition**: the repetition of sending a set of data buffers

The main function should be:

1. Accept a new connection.
2. Set up a signal handler for SIGIO signals. SIGIO signals will come from socket IO calls of the client.

```
signal(SIGIO, your_function); // you need to define your_function a priori
```

3. Change this socket into an asynchronous connection using fcntl commands as follows.

```
fcntl(fd, F_SETOWN, getpid());
fcntl(fd, F_SETFL, FASYNC);
```

4. Let this server sleep forever.

Server.cpp must include **your_function** (whatever the name is) which is called upon an I/O interrupt (the SIGIO signal for the Socket IT). This function takes has an integer as an input paramater and returns void. The input paramater is the signal identifier, (i.e., SIGIO) which you don't have to utilize in the function. This function should:

1. Allocate **dataBuf[BUFSIZE]**, where BUFSIZE = 1500.
2. Repeat reading data from the client into databuf[BUFSIZE]. Note that the **read** system call may return without reading the entire data if the network is slow. You have to repeat calling **read** like the code below. This should be done for each repetition on the server.

```
inRead = 0;
while (nRead < BUFSIZE)
{
    int bytesRead = read(newSD, dataBuf, BUFSIZE - nRead);
    nRead += bytesRead;
    count++;
}
```

   Check the manual page for **read** carefully.
3. Send the number of read() calls made, (i.e., **count** in the above) as an acknowledgment.
4. Close this connection.
5. Terminate the server process by calling **exit(0)**.

You must conduct performance evaluation over 1Gbps network between two different computing nodes of uw1-320-00 ~ uw1-320-15. For each network, your performance evaluation should cover the following nine test cases:

1. **repetition** = 20000
2. Four combinations of **nbufs * bufsize** = 15 * 100, 30 * 50, 60 * 25, and 100 * 15
3. Three test scenarios such as **type** = 1, 2, and 3

You may have to repeat your performance evaluation and to average elapsed times.

5. What to Turn in

Please turn in your program to Canvas as a .zip file.

Criteria	Percentage
Documentation of your algorithm including explanations and illustrations in one or two pages	1pt (5%)
Source code that adheres good modularization, coding style, and an appropriate amount of comments. The source code is graded in terms of (1) correct a tcp socket establishment, (2) three different data-sending scenarios at the client, (3) a signal-driven data-receving code at the server, (4) use of gettimeofday and correct performance evaluating code, (5) coding guidelines, etc...	10pts (50%)
Execution output Submit partial contents of standard output redirected to a file. You don't have to print out all data. Just one page evidence is enough.	1pt (5%)

Performance evaluation showing round trip times and number of reads in an understandable manner. Also clearly show throughput.	4pts (20%)
Discussions should be given for (1) 1Gbps actual throughputs, (2) a discussion in terms of comparing multi-writes, writev, and single-write performance over 1Gbps, and (3) the main reason why we had to use asynchrnoous reads rather than blocking reads at the server.	4pts (20%)
Total	20pts (100%)

6. FAQ

This FAQ page may answer your quetions. Click here