

TCP Client/Server Programming

First Name: _____ Last Name: _____

ID: _____

1 Introduction

We know that many Internet applications use a client/server model for communication: a server listens for connections; and a client initiates connections to the server. How are these client and server programs implemented?

In this lab you will learn the basic programming constructs, called *sockets*, to create a client and server program. You will use these programming constructs to implement your own client/server application in following labs.

2 Programming with Sockets

Sockets are programming constructs used to communicate between processes. There are different types of systems that sockets can be used for, the main one of interest to us are Internet-based sockets (the other commonly used socket is Unix sockets).

Sockets for Internet programming were created in early versions of Unix (written in C code). Due to the popularity of Unix for network computing at the time, these Unix/C based sockets become quite common. Now, the same concept has been extended to other languages and other operating systems. So although we use C code and a Unix-based system (Ubuntu Linux), the principles can be applied to almost any computer system.

There are two main Internet socket types, corresponding to the two main Internet transport protocols:

1. Stream sockets use the Transmission Control Protocol (TCP) to communicate. TCP is stream-oriented, sending a stream of bytes to the receiver. It is also a reliable transport protocol, which means it guarantees that all data arrives at the receiver, and arrives in order. TCP starts by setting up a connection (we have seen the 3-way handshake in other labs), and then sending data between sender and receiver. TCP is used for most data-oriented applications like web browsing, file transfer and email.
2. Datagram sockets use the User Datagram Protocol (UDP) to communicate. UDP is an unreliable protocol. There is no connection setup or retransmissions. The sender simply sends a packet (datagram) to the receiver, and hopes that it arrives. UDP is used for most real-time oriented applications like voice over IP and video conversations.

In this lab we are dealing only with Stream (TCP) sockets.

The basic procedure is shown in Figure 1. The server must first create a socket, then associate or bind an IP address and port number to that socket. Then the server listens for connections.

The client creates a socket and then connects to the server. The `connect()` system call from the client triggers a TCP SYN segment from client to server.

The server accepts the connection from the client. The `accept()` system call is actually a blocking function – when the program calls `accept()`, the server does not return from the function until it receives a TCP SYN segment from a client, and completes the 3-way handshake.

After the client returns from the `connect()` system call, and the server returns from the `accept()` system call, a connection has been established. Now the two can send data.

Sending and receiving data is performed using the `write()` and `read()` functions. `read()` is a blocking function – it will only return when the socket receives data. You (the application programmer) must correctly coordinate reads and writes between the client and server. If a client calls the `read()` function, but no data is sent from the server, then the client will wait forever!

2.1 Servers Handling Multiple Connections

It is common for a server to be implemented such that it can handle multiple connections at a time. The most common way to do this is for a main server process to listen for connections, and when a connection is established, to create a child process to handle that connection (while the parent process returns to listening for connections). In our example, we use the `fork()` system call.

The `fork()` system call creates a new process, which is the child process of the current process. Both the parent and child process execute the next command following the call to `fork()`. `fork()` returns a process ID, which may be:

- Negative, meaning the creation of the child process was unsuccessful
- 0 is returned to the child process
- A positive value is returned to the parent process – this is the process ID of the child.

Hence we can use the process ID returned from `fork()` to determine what to do next – the parent process (`pid > 0`) will end the current loop and go back to waiting for connections. The child process (`pid = 0`) will perform the data exchange with the client.

2.2 Further Explanation

You should read the source code for the `server.c`, and then the source code for `client.c`. The comments contain further explanations of how the sockets communication is performed.

The example code for `client.c` and `server.c` came from:

<http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html>

You may read through the details on this web page.

Most of the socket system calls are described in detail in their individual `man` pages. You should use the `man` pages for finding out further details of each function.

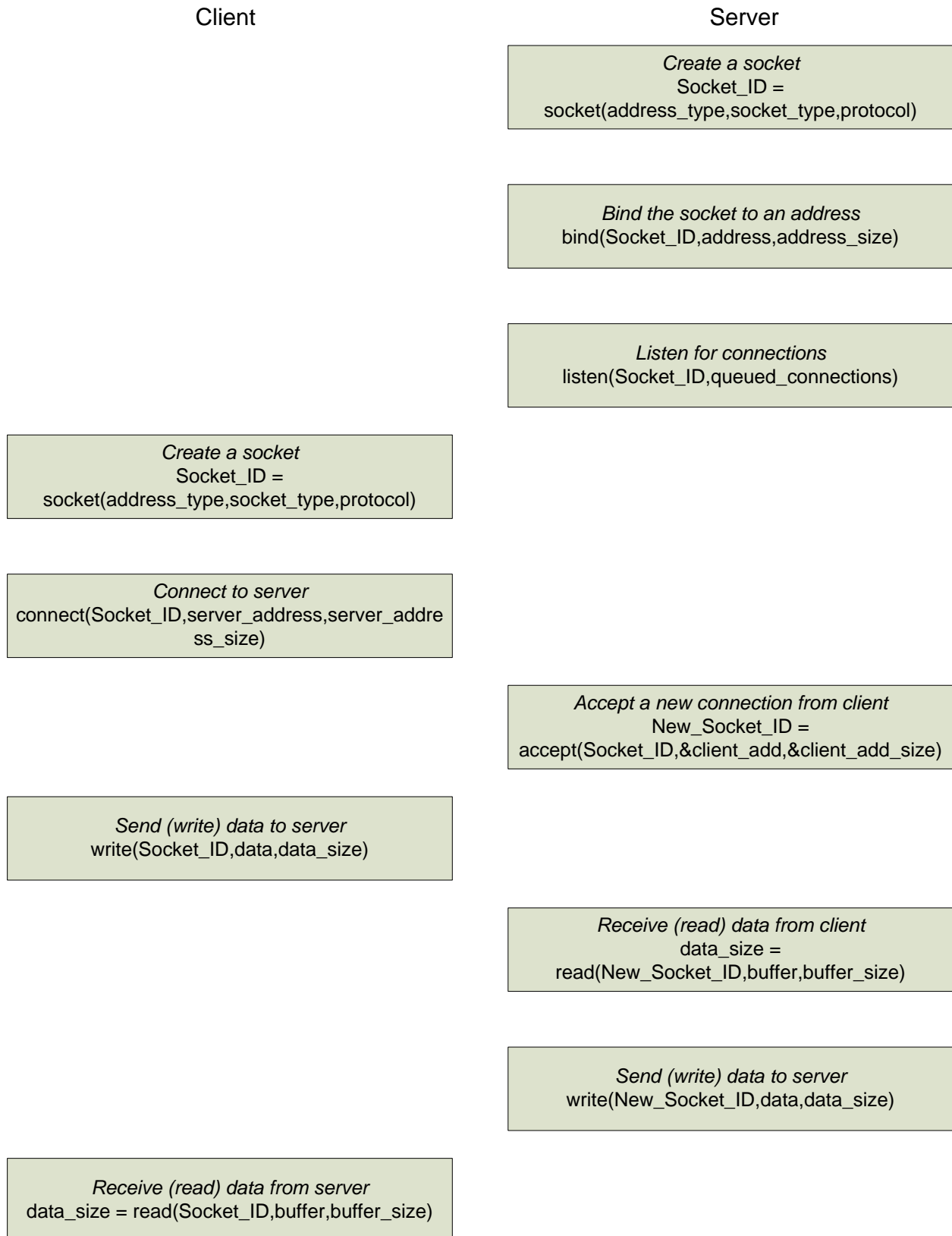


Figure 1: Socket communication

3 Tasks

In this lab you will work in pairs again (one client computer and one server computer). All tasks must be completed using Ubuntu Linux.

You can use the switched Ethernet for this lab. In other words, you do not have to configure the two computers in a peer-to-peer mode. Instead, make note of the IP address assigned to your computer and use that to communicate with your partner's client/server.

For the following instructions we will assume the IP address of the client is ClientIP and the IP address of the server is ServerIP. You should replace these IP addresses with the IP addresses of the computers you are using. Also, when showing commands:

- `client>` means the command line prompt on the client machine
- `server>` means the command line prompt on the server machine
- `>` means the command line prompt on either or both machines

For example:

```
client> ls
```

means you should execute the `ls` command on the client machine.

3.1 Download the Client and Server programs

One (or several) of the computers in the lab will store the files needed for this lab exercise. We will tell you the IP address of this/these computers at the start of the lab. Go to the web server and download all of the files to your computer.

The files will be automatically saved on your Desktop (which is the directory `/home/student/Desktop` or `/home/network/Desktop`).

3.2 Compile the client and server programs

Copy the files into a new directory:

```
> cd ~
> mkdir sockets
> cp Desktop/* sockets
> cd sockets
```

Now compile the programs (you only need the client program on the client computer and the server program on the server computer).

```
client> gcc -o client client.c
# or
server> gcc -o server server.c
```

The programs should compile without any warnings or errors.

3.3 Using the Client and Server Programs

Start the server (you can choose any port number larger than 1024) :

```
server> ./server 40000
```

The server does nothing – it is waiting for a client connection.

Now start the client and enter a message:

```
client> ./client ServerIP 40000
Please enter the message: hello
I got your message
```

You should see on your server:

```
Here is the message: hello
```

3.4 Exercise 1 – Capture the Exchange

Using Wireshark, capture the messages sent when running the server program and connecting and sending a “hello” message from the client.

Hint: It is recommended you go to the Wireshark Edit -> Preferences menu, and under Protocols select TCP and make sure only the following three options are selected:

Show TCP summary in protocol tree

Analyze TCP sequence numbers

Relative sequence numbers and window scaling

Hint: Sometimes when you use Wireshark to capture packets, you capture many packets (e.g. HTTP, ARP, MSN), but are only interested in a selection of those (e.g. HTTP to a specific web server only). You can use the display filter (shown above the list of captured packets) to display only those packets of interest. A brief explanation is given in the Help -> Contents -> Display Filters menu. As an example, if you were using port number 40000 for your server, you could show only packets set to and from the server using:

```
tcp.dstport == 40000 || tcp.srcport == 40000
```

or simply

```
tcp.port == 40000
```

Question: What port number did your client use? Why?

Question: Draw a time sequence diagram showing the packet exchange. You do not have to show the times, sequence number or ACK numbers. Simply show the information (message types, protocol information) that is exchanged between sender and receiver.

Question: In the packets that contain the messages (e.g. “hello” and “I got your message”), what TCP flags are set? What do the flags mean?

Question: What do you think the PSH (or Push) flag is used for in TCP? What would happen if it was not set?

3.5 Exercise 2 – Capture a Reset

This is a simple task (should take you 5 minutes):

- Close the server program.
- Start capturing in Wireshark
- Start the client program, trying to connect to the server
- View and compare the output of the client program and the Wireshark capture.

Question: What happened? Why?

3.6 Exercise 3 – Modify the Data Exchange

In this task you should create a new (modified) client and server program that allows them to exchange multiple messages.

Create new copies of the client and server programs:

```
client> cp client.c client_2msgs.c
server> cp server.c server_2msgs.c
```

Edit the files `client_2msgs.c` and `server_2msgs.c` to create a very simple login mechanism. The server should ask the client for a username by sending a “username” message, and then the client will send the username to the server. Then the server will prompt for a password by sending the “password” message, and then the client will send the password to the server. Finally, the server will check if the username and password match an already known username/password

pair (you can hardcode this into your server code), and send a response back to the client. Then both the client and server can finish (of course the server should still handle more connections).

The output of the interaction should look like this:

Client	Server
1. Username:	
2. <user types in username, eg. "X">	
3.	Client login with username "X"
4. Password:	
5. <user types in password, eg. "Y">	
6.	Client entered password "Y"
7.	Username and password are correct.
8. You are now logged in.	

Alternatively, the last two lines may be:

7.	Username and password are incorrect.
8. Incorrect username and password.	

Hint: In the server program, you need to modify the function `dostuff()` to perform the correct read and write operations. The first operation from the server should be to write the message "Username:" to the socket. In the client program the first operation should be a `read()`.

Question: Demonstrate your new program to the instructors.

4 Cleaning Up

At the end of the Lab class you must reset your computer back to the default settings, which include:

- Make sure you copy your source code to a USB drive – you will need it in future labs.
- Delete your source code from the PC.