1 Introduction

The development of network-based software has been around for many years. Since its conception, there have been numerous communication protocols and programming interfaces put into practice. However, the advent of the Internet has served to popularize one particular suite of protocols and to cement the status of these protocols as the de-facto standard for network programming. This protocol suite is known as the TCP/IP suite (or sometimes just the IP suite), named for its inclusion of the TCP and IP protocols.

The purpose of this lab is to help you familiarize yourself with network programming (in the C programming language), using the *Sockets API*. The Sockets API provides the customary programming interface used by Unix/Linux systems to provide access to the various members of the TCP/IP protocol suite. In this lab, you will be making use of the Sockets API in order to utilize the two primary transport-layer protocols implemented by the TCP/IP suite, namely TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

2 Client-server Model

By far, the most ubiquitous design model used in network programming today is the *client-server* model. In this model, there are two classes of network participants: *clients*, and *servers*. The servers are responsible for providing services; the clients are responsible for contacting the servers to request and consume those services that the servers provide.

Many network-based applications in wide use adhere to this model. For example, web browsing (HTTP), email (POP3/IMAP), and file transfer (FTP) all fit under this category. Figure 1 presents an abstract diagram depicting the communication pattern characteristic of a client-server application; there are a number of isolated clients, each communicating with a single server.

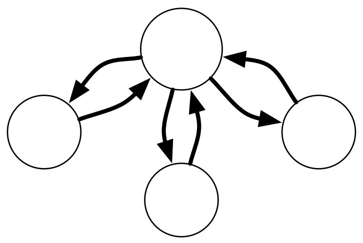

Figure 1: Graphical representation of a client-server application

3 Problem Description

In this lab, you will be implementing **two** client-server pairs; one pair must be implemented using TCP, the other using UDP. Both implementations must satisfy the specification presented below.

3.1 The Queue Server

The **Queue Server** application is designed to operate as a remotely accessible *queue* data structure to the client processes that it serves. Much like the local version of this data structure, it should support *enqueue* and *dequeue* operations, returning elements in FIFO order to the clients. For simplicity, the elements of the queue will be bytes. However, the server should support "batch" versions of the operations in order to improve performance. Conceptually, the server should support the following commands:

- *Enqueue*
  - **Parameters**: b, an array of elements (bytes) to put in the queue; array elements should be queued in order (from the smallest to the largest array index)
  - **Returns**: $n \leq length(b)$, the number of elements of b successfully put in the queue
- *Dequeue*
  - **Parameters**: n, the number of elements (again, bytes) to remove from the queue
  - **Returns**: b (where $length(b) \leq n$), an array containing the elements that were removed from the queue; the array should be filled in order (the smallest array index should contain the element that was added to the queue first)
- *Size*
  - **Returns**: n, the total number of elements currently stored in the queue

The application must be written in the C programming language; and it should take two command-line parameters: the IP address and port number that the server should bind to when listening for incoming requests.

When the server is started, it should wait and service incoming commands issued by the clients. The server need only service a single request at a time, but should be able to process multiple requests in sequence (i.e. after the server processes one request, it should wait for the next one). In order to maintain the queue, the server will need to be able to keep track, internally, of the state of the queue. This state need not be persistent across separate server invocations.

3.2 The Queue Client

The **Queue Client** application is designed to operate as the consumer of the services provided by the **Queue Server**. The client application should be capable of issuing requests to the server corresponding to each of the three commands that server is responsible for processing.

The client application must be able to handle three types of invocations, corresponding to each of these commands. As such, the client application is required to take three parameters: the server IP address, the port the server is bound to, and the type of request the client is issuing (one of $\{e, d, s\}$ for *enqueue*, *dequeue*, and *size* respectively). In order to obtain the byte array for *enqueue* operations, the client application should read from *stdin* until EOF (end of file). Similarly, the client should print any response from the server to *stdout*. All errors, logging, and debug output should go to *stderr*.

3.3 Example

A sample run of your one of your client-server pairs should look something like this:

```
$./queue_server_tcp 127.0.0.1 12345 &
$echo "Hello World" | ./queue_client_tcp 127.0.0.1 12345 e
12
$./queue_clent_tcp 127.0.0.1 12345 s
12
```

```
$echo "12" | ./queue_clent_tcp 127.0.0.1 12345 d
Hello World
$./queue_clent_tcp 127.0.0.1 12345 s
0
```

4 Sockets API

In order to actually implement the networking component of your client-server applications, you will be using the Sockets API. This programming interface consists of a number of function calls designed to create, destroy, and manipulate abstract *socket* structures. Although this is not an exhaustive list, these are the primary functions that you will be using:

- **socket** – Allocate a new socket of a particular protocol type
- **bind** – Attach a socket to a particular address (i.e. an IP-port pair)
- **listen/accept** – Wait for incoming connections (if the protocol supports this)
- **connect** – Initiate a connection (if the protocol supports this)
- **getsockopt/setsockopt** – Get/set options available for the socket
- **send/sendto/sendmsg/write** – Send data out on a socket
- **recv/recvfrom/recvmsg/read** – Receive incoming data from a socket
- **shutdown/close** – Release resources associated with the socket

NOTE: In order to view the full type signature for these functions and determine which header file(s) you must include in the source of your program, I suggest you use the “man” (as in manual) application. For example, issuing the command “man socket” on any of the CS Linux machines should give you the corresponding information about the “socket” function.

4.1 TCP and UDP: Protocol Guarantees

When designing your two client-server implementations, it will be important to consider the differences between the guarantees provided by each of the transport protocols you will be using. Figure 2 provides a high-level overview of the different guarantees offered by TCP and UDP.

	UDP	**TCP**
Connection-orientation	Connectionless	Connection-oriented
Data Presentation	Datagram	Byte stream
Message Ordering	None	In-order
Reliable Delivery	No	Yes
Flow/Congestion Control	No	Yes

Figure 2: Protocol guarantees for TCP and UDP

Given the significant differences between TCP and UDP, it is critical that you pay special attention when designing your application. For example, since UDP provides no message-ordering guarantees, in your implementation of the UDP client-server pair you will probably want to take steps to ensure that “stale” responses (i.e. responses to previous queries) do not get confused with responses to active queries; consider the following scenario:

1. Client A issues a *Dequeue* request to server S
2. Server S receives the request and sends a response to client A
3. The user kills client A before it receives the request
4. The user starts a new client A' bound to the same IP and port
5. Client A' issues a *Dequeue* request to server S
6. Client A' receives the response intended for client A and mistakes it for the real response from the server

Although scenarios like this depend on a specific and *unlikely* sequence of events in order to occur, they are *possible* nevertheless and you should account for them.

4.2 Communication Endpoint Life Cycle

Since the transport-layer protocols offered by the TCP/IP suite offer such radically different guarantees (and since the Sockets API is used as the common interface to all of them), you will need to use the Sockets API in slightly different ways when operating with TCP as opposed to UDP, for example. The following presents the typical sequence of operations required to create, utilize, and then close a communication endpoint for the various protocols/configurations you will be using:

TCP Server:

1. socket
2. bind
3. listen
4. accept
5. send/recv
6. close

TCP Client:

1. socket
2. connect
3. send/recv
4. close

UDP Server/Client:

1. socket
2. bind
3. sendto/recvfrom
4. close

5 Tips

This particular lab is intended to be relatively straightforward. However, network programming using the Sockets API in C can be somewhat tricky simply by virtue of the existence of the many seemingly-arbitrary implementation-specific details of the Sockets API. The purpose of this section is to try to point out some of these common pitfalls. The following table presents a number of common bugs, along with suggestions for what might trigger them and possible solutions:

Symptom	Possible Problem	Solution
No network traffic is getting through	The firewall is in the way	Ensure the firewall rules are correct (and fix, if necessary; use "/sbin/iptables")
	The port you are using requires administrative privileges	Make sure not to use a port in the range 0-1023
	The port numbers are incorrect	Make sure to use the *htons* function on both ends to ensure the proper byte-ordering when specifying a port number
The program dies with a "Broken pipe" message	The remote end closed a connection you were trying to read from, resulting in a SIGPIPE signal being sent to the application	Set up the application to handle or ignore these signals (use the *sigaction/sigprocmask* functions)
A call to *recv/read* never returns	The *recv/read* call was instructed to read more data than is available	Only read data you know is available or use nonblocking I/O (i.e. set the socket to be nonblocking on reads)
A call to *recv/read* returned less data than expected though there is still data to read	The *recv/read* call was interrupted	Be sure to put your call to *recv/read* in a loop to verify that you have read everything you want

6 Lab submission

For this lab, you must submit the source code implementing the **two** client-server pairs that adhere to the specification detailed in section 3 (**NOTE: please do not submit any binary files**). You should also make sure to include a "README" file detailing what you did (i.e. include a high-level description of how you implemented your system) and any problems that you had encountered. You must also include in your README a description of how to compile and run your application (sample output is appreciated).

Although it is not required, it is highly suggested that you use a build tool (such as "make" or "omake") to help automate the compilation of your applications.

6.1 Osaka

In order to submit the labs for this course, you will be using the "Osaka" submission system. The Osaka binaries are available from any of the CS machines, provided you have added the following directory to your "PATH" environment variable: "/cs/software/stow/osaka-0.1/bin".

To submit an assignment, you would perform the following steps:

1. Change to the directory containing all of the files you intend to submit for the lab (i.e. "cd some/path/to/lab1")
2. Run cs145-submit while you are inside that directory and specify the name of the homework/lab you intend to submit (like "cs145-submit lab1")

NOTE: All files in the current directory and all subdirectories (excluding files prefixed with a ".") will be included as part of your submission.

6.2 CS Accounts

To be able to log into the CS department machines, you will need to have a CS account. If you do not already have one, you can request one here:

`http://www.cs.caltech.edu/cgi-bin/sysadmin/account_request.cgi`

6.3 Style Notes

Although the largest component of your lab grade will be based on correctness, style is also an important component. Here are a few notes to ensure you do not lose points on style.

- Comment all major functions, data structures, and complex conditionals
- Provide robust error-handling code; error cases must be handled for all functions that can have an error result
- Compile with all warnings enabled ("-Wall" if you are using gcc) and eliminate them before submission
- I suggest using a tool like "make" (or, better yet, OMake – http:// omake.metaprl.org) to help automate the compilation and testing of your lab projects
- **ALWAYS** provide a "README" file, explaining how to compile and run your code