CSE/EE 461
Introduction to Computer-Communication Networks
Autumn 1997
Project 1
Due electronically: 11:30 pm, Tuesday, November 11, 1997.

## Objectives

- Become familiar with the Berkeley sockets API.
- Design part of an application-layer protocol for transferring files.
- Create a network file transfer application using sockets.

The code is located in the directory `/cse/courses/cse461/ftp/`. You must do this assignment individually.

# 1 Preliminaries

This assignment focuses on transport and application layer protocols.

The Berkeley socket interface is a widely-available application program interface to the transport and network layers in Unix. It supports TCP/IP, Xerox NS, Unix domain, and possibly other protocols.

TCP/IP is a suite of protocols consisting of TCP (Transmission Control Protocol, a reliable connection-oriented transport protocol), UDP (User Datagram Protocol, an unreliable connectionless transport protocol), and IP (Internet Protocol, an unreliable connectionless network protocol). TCP and UDP are built on IP. In this assignment, we will use TCP to build a simple file transfer application allowing remote users to copy files to and from a server host. This is similar to the functionality provided by FTP.

## 1.1 Client/Server

Contemporary network applications are often implemented using a client/server model. A server program awaits messages from instances of its client program and provides services for the clients. For example, when you access a web page on the Internet, your local machine runs a Netscape browser client and the remote machine hosts a World Wide Web server. Similarly when you send an email message, you establish a connection between your local machine (the client) and a server process on the destination machine. Server processes in Unix are often run in daemon processes, without being attached to any terminal console.

In the file transfer application that we are going to develop, clients exchange files with the server. Once a client establishes a connection with a server, it may test its connection, request authentication, ask to retrieve one or more files resident on the server's host, or copy one or more files from the client host to the server. Obviously, other file or directory manipulation functions could be added.

## 1.2 Addresses

To communicate, a program must specify the location of the program it wishes to communicate with. Each machine in the Internet has a unique address, represented by a hierarchically structured name, e.g. `orcas.cs.washington.edu`. The name or Internet address by itself is not specific enough, however. For example, there might be both a mail daemon and a finger daemon on orcas. They reside on the same machine and thus have the same Internet address. A mail client program has to specify that it wants to talk with the mail daemon but not the finger daemon on the target machine. Multiple services on a given host are distinguished by *port numbers*.

A number of widely-agreed-upon port numbers, called *well known* ports are listed in the file `/etc/services` on most Unix systems. An API call permits servers and clients to locate well known ports by name rather than by number.

## 1.3 Sockets

We briefly describe some aspects of sockets that are necessary for this assignment. The text outlines the Berkeley socket API beginning on page 486. Complete details can be found in the man pages for the API function calls.

Sockets are an abstraction of communication endpoints. Sending and receiving messages via connected sockets is analogous to writing and reading files in the Unix file system. Each socket in use by a program has a number, or *socket descriptor*, that identifies it. This small integer serves a function identical to that of a file descriptor.

Each Internet socket also has an Internet socket address, more or less analogous to a file name, which is a structure containing an Internet address plus a port number. An Internet socket address has type `struct socketaddr_in`.

Unix provides a number of system calls that enable a programmer to make use of sockets. The `socket()` system call creates a socket, and the `bind()` system call associates an address with the socket. This is *not* like opening a file, where you specify the file name when you ask for a file descriptor. For sockets, this is a two-step process: you get a socket descriptor first, and then you bind the descriptor to a socket address. The `listen()` system call sets a socket to accept incoming service requests. The `connect()` and `accept()` system calls establish a TCP connection between two programs.

You don't need to understand these system calls to do the assignment, so we will skip the details here. You can read the comments in the code that we give you to know more about these system calls. Messages are sent and received by the `send()` and `recv()` system calls respectively. These are the functions that you will be using for this assignment.

If you need more information than the descriptions below provide, see the online manual by typing `man 2 send` and `man recv`.

### 1.3.1 Send a Message

`int send( int` *sockfd*`, char *`*msg*`, int` *len*`, int` *flags*`);`
*sockfd* is the descriptor of the connected socket through which you want to send the message. *msg* is a pointer to the message, *len* is the length of the message in bytes, and *flags* is always 0 for the purpose of this assignment. So the statement

```
send(sender_socket, "hello", 6, 0);
```

sends a string `"hello"` using the socket `sender_socket`. (The length of the message is 6 bytes instead of 5 bytes because of the invisible end-of-string character.)

`send()` returns −1 when an error is detected locally (e.g. *sockfd* is invalid). Note that the TCP protocol guarantees that `send()` will not lose or damage messages.

### 1.3.2 Receive a Message

`int recv( int` *sockfd*`, char *`*msg*`, int` *len*`, int` *flags*`);`
*sockfd* is the descriptor of the connected socket from which you want to receive a message. *msg* points to a message buffer and *len* indicates the maximum number of bytes to receive on this call. Again *flags* is always 0 for our purpose.

The function waits for a message to arrive. That is, the call *blocks* until a message arrives. When the function returns, the buffer pointed to by *msg* contains the received message. `recv()` returns the length of the received message in bytes, which may be less than or equal to the amount specified in the *len* argument.

Note that the TCP protocol guarantees that data received by any two `recv()` calls are delivered in the same order in which they were sent. When the sending process requests shutdown of the connection, `recv()` returns zero.

For example:

```
char msg[8];
int source_len, n;

n = recv(receiver_socket, msg, 8, 0);
```

After the function returns, `msg` will contain `"hello"` (or a prefix of it) and `n` will be 6 (or perhaps less).

### 1.3.3 More on Sockets

Sockets are designed to provide transparent, easy to use communications between computers, even though the computers may use different representations for numeric values. The C standard library provides several data conversion functions for ensuring that an integer or other type of value will be used in the same way on both sides of a client/server application. htons() (host to network short) and ntohs (network to host short) are examples of such functions.

# 2 A File Transfer Application

The file transfer application that you will be building consists of two executables, `server` and `client`. Part of the code has already been written for you, and you are supposed to fill in some functions. The code compiles to form an incomplete application, with some but not all of the functionality. Executables with full functionality are provided in the project directory as well, and they are called `server_demo` and `client_demo`.

The syntax of the server is:

`server` *server_port*

where *server_port* is the port number that the server process will be using. In a real application, the port number might be hardcoded, or (better) retrieved by name from `/etc/services`. For this assignment, you should pick a number between 5001 and 65535. If you get an error message:

```
server: cannot bind local address
```

that means somebody else (probably one of your classmates) is using that port number. Just pick another one. When the server starts, it immediately prompts for an authentication password, which clients must send to enable file transfers. This security provision isn't very robust, but it should prevent casual users from copying or overwriting your files while your program is being tested.

Suppose you have started the server on orcas with port number 6000.

The syntax of the client is:

`client` *server_hostname server_port*

where *server_hostname* is the name of the machine that the server is running, and *server_port* is the port number that the server is using. The port number of the client is automatically selected by the operating system and you don't need to specify it. (The reason for the asymmetry is that the client has to know the port number of the server's listening socket before it can request a connection with the server. However when the server accepts the client connection, it can tell the client's socket address by the `accept()` system call, so the server does not need to know the client's port number in advance.)

As an example, you can run on `sanjuan`:

```
client orcas 6000
```

You can even compile and run the program outside the CS domain, for example on the MSCC machines, on `homer.u`, etc. In those cases, you will need to state that orcas is in the CS domain, for example

```
client orcas.cs 6000
```

or if you run the program outside the UW domain:

```
client orcas.cs.washington.edu 6000
```

The executables `server_demo` and `client_demo` have the same command as `server` and `client`, respectively. They are:

- `hello` sends a message to the server, which the server echos back.
- `login` enables the file transfer commands.
- `quit` terminates the client.
- `help` lists the commands.
- `get` creates a copy of a file from the server on the client.
- `put` creates a copy of a client file on the server.
- `list` lists the current directory of the server.
- `llist` lists the current directory of the client.
- `chdir` changes the current directory of the server.
- `lchdir` changes the current directory of the client.
- `cwd` prints the current directory of the server.
- `lcwd` prints the current directory of the client.

All the commands are implemented in the demo and all but `get`, `put`, `list`, `chdir`, and `cwd` are implemented in the source code you are given.

# 3 The Assignment

## 3.1 Part 1 - Protocol Design

Sending files back and forth across the network involves specifying pathnames, sending arbitrary amounts of data back and forth, and signaling when a file has been completely sent. The agreement between the client and server programs about how this information will be unambiguously shared is the application-layer protocol.

You must design a protocol that enables the client and server to engage in correct file transfers. Take for example, the case of `put`, that takes a file on the client and copies it to a file on the server. You will need to implement several types of messages to enable this to happen.

1. There will be a message from the client to the server indicating that it wishes to write a certain file.
2. The must be a message from the server to the client indicating whether or not it is possible to write the file.
3. If the server accepts the client's request, then the client can send a sequence of messages which contain the file. All of these message are of the same type.
4. When the client has completed sending all the bytes of the file, then it should send a message to the server indicating it is done.

5. Finally, the server can send a message to the client telling it that the transfer is complete.

Figure 1 shows the pattern of communication between the client and server needed for successfully transferring a file from the client to the server. As you can see both the client and server programs will have to be able create and interpret correctly these kinds of messages. Part of your design will be the design of the message formats you will use in your program.

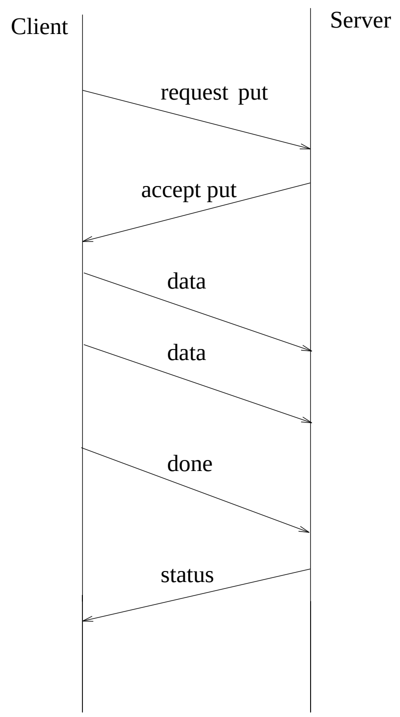


Figure 1: Protocol needed to transfer a file from the client to the server.

### 3.2 Part 2 - Implementation

There are five source files provided, `msgs.h`, `server.h`, `client.h`, `server.c`, and `client.c`.

- `server.c` and `client.c` implement the server and the client respectively.
- The `*.h` files provide constant definitions and prototypes for the functions in the program files.

Provide a working and well documented implementation of the commands `get` and `put`. You will want to make sure that you have done a good job of design before attempting any coding. In addition, you will want to make sure that you understand the implementations of the commands that are already implemented.

### 3.3 Part 3 (Optional for extra credit)

Complete the application by implementing the commands `list`, `chdir`, and `cwd` that list directories, change directories, and print the current directory of the server. Some of these commands are fairly tricky to implement.

### 3.4 Part 4 (Optional for extra credit)

A significant weakness of the present server implementation is that it provides service to *only one* client at a time. Improve the program so that multiple clients can be handled by one server. See the man page for the `fork()` system call for details of a mechanism that makes it possible for the server to make copies of itself that serve individual clients, while the original continues to listen for, and accept, more client connections.

## 4 Criteria for Grading

The main part of the project is implementing `get` and `put`. If these are not done well then there is no extra credit. Your code should be well organized and well presented. Because protocols are notorious for being tricky, your programming style should be impeccable. Your comments in the code preceding the client's functions implementing the commands should explain the design of your solutions. These comments need not be long, but should briefly explain the pattern of communication used by your protocol.

## 5 Turn in

Just type `make turnin` to submit your assignment. You may turnin more than once, but only the last one you turn in will count. The assignment is due by 11:30 pm on Tuesday, November 11, 1997. No late assignments are accepted.