COMP3331/9331 Computer Networks and Applications

Programming Tutorial for Term 2, 2024

Version 1.0

Table of Contents

1. Objectives

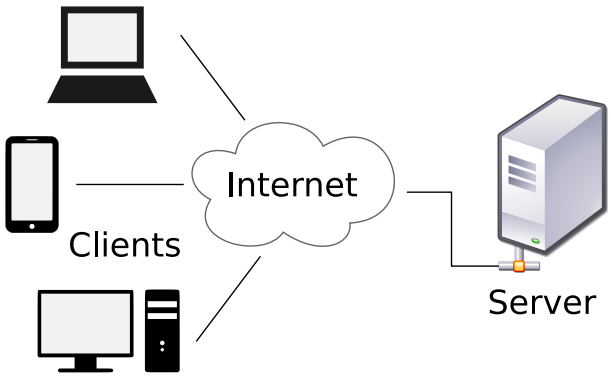
- Lay the foundational building blocks for the programming assignment.
- Review the client-server paradigm.
- Discuss the program flow of typical client and server applications.
- Explore the socket API.
- Build a client-server application.
- Introduce multi-threading and locking primitives to handle concurrency.
- Have some fun!

2. Prerequisites

- Chapter 2.7 on Socket Programming from Computer Networking - A Top-Down Approach.
- Completion of the socket programming exercises in Labs 2 and 3.
- A basic understanding of Linux. Some good resources include:
    - UNIX Tutorial for Beginners (introductory)
    - The Missing Semester of Your CS Education (more advanced)

3. Client-Server Model

The client-server model is one of the most used paradigms in networked and distributed systems. Servers typically provide some resource or service, while clients request that resource or service.



[1]

Figure 1: The Client-Server Model

Clients initiate communication and typically only communicate with one server at a time. Meanwhile, servers wait to be contacted, and at any point in time, may be communicating with multiple clients.

Questions

1. Do you think it's necessary for the client to know the server's address prior to communicating?
2. Do you think it's necessary for the server to know the client's address prior to communicating?

Answer

Given that clients initiate communication, the client needs to know in advance that the server exists and its address. The server, however, does not require the same advanced knowledge of the client. The server can learn of the client's existence, and the client's address, when the client first makes contact.

4. Typical Client-Server Applications

4.1. TCP

A typical sequence of socket API function calls for a client and a server participating in a TCP connection is presented in Figure 2. Note the two `recv()` calls in the server are actually one and the same. A second call is shown only to improve the clarity of the diagram regarding connection termination.

[1] Source: https://commons.wikimedia.org/wiki/File:Client-server-model.svg

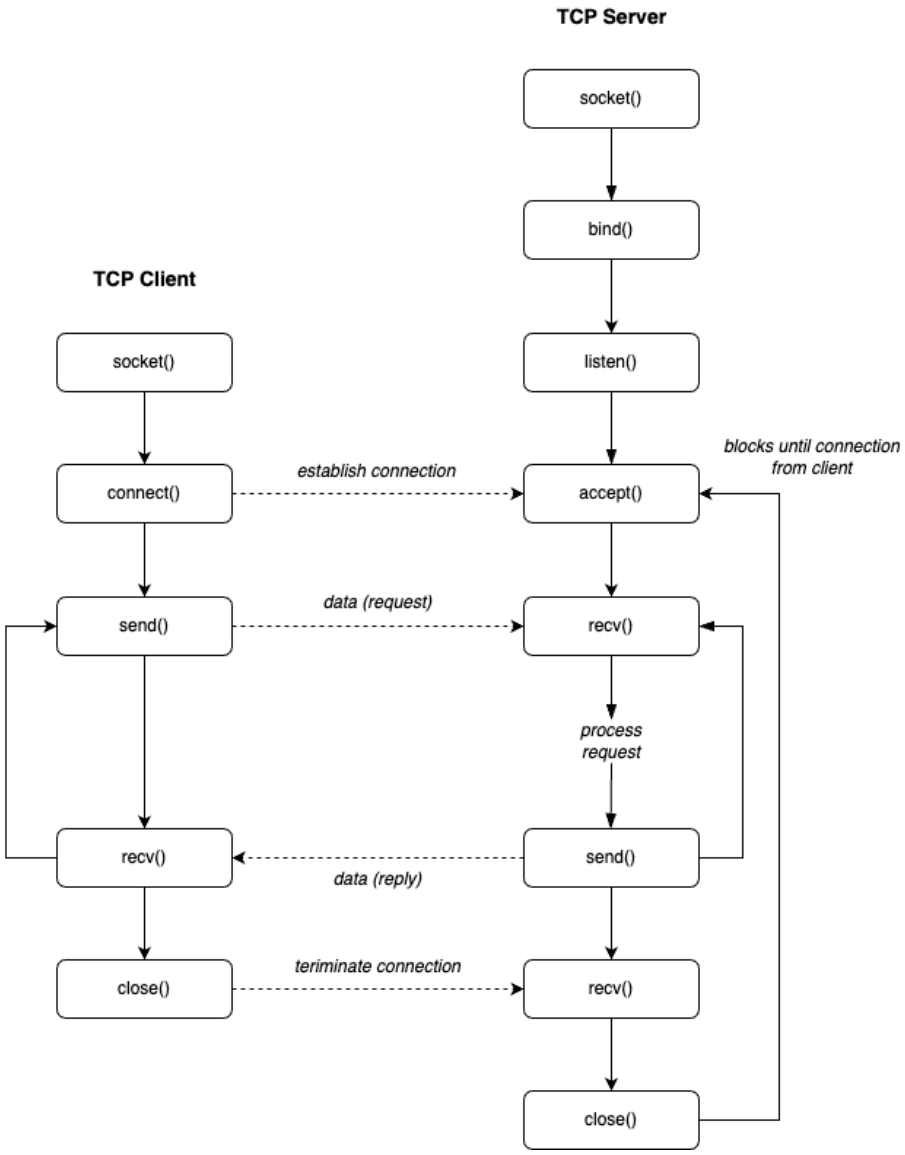


Figure 2: A typical TCP program flow.

Questions

1. What purpose does each of the function calls serve?
2. Why doesn't the client need to bind? What is the consequence of not binding?
3. What is the difference between a blocking socket and a non-blocking socket?
4. The diagram depicts a single-threaded iterative server, where the server handles one client at a time. What might be a logical way to introduce multi-threading and turn it into a concurrent server, so the server can handle multiple clients at the same time?

Answer

For the TCP client:

- `socket()` creates a communication end-point. In setting this up we specify a protocol family and a socket type. We would typically set the family to `AF_INET` to indicate IPv4, and the type to either `SOCK_STREAM` (for TCP) or `SOCK_DGRAM` (for UDP).

- `connect()` initiates a three-way handshake and establishes a connection with the TCP server. It takes the address of the server as an argument. If the client socket hasn't been bound (with `bind()`), then this will also cause an address to be assigned to the client. The client's IP address will be based on the outgoing interface needed to communicate with the server, and the operating system will assign an available port number. The port number will be selected from the dynamic port range, 49152 to 65535. These ports are also known ephemeral ports.
- `send()` and `recv()` are used to send and receive data through the socket. If the socket is blocking, then the call to `recv()` will suspend execution of the program until data is available to be read. If the socket is non-blocking, the call will return immediately regardless of whether any data is available. We can also configure a timeout on the socket, so any blocking function calls will fail after a specified period of time.
- `close()` terminates the TCP connection and closes the socket.

For the TCP server:

- `socket()`, `send()`, `recv()`, and `close()` serve the same purpose as for the TCP client.
- `bind()` assigns an address to the socket. This is important for the server, as we need it to be listening at a known location. We typically nominate a port number from the dynamic port range for our server applications, to minimise the risk of clashing with other network applications.
- `listen()` converts an unconnected socket into a passive socket or "welcoming socket", indicating that the operating system should accept incoming connection requests directed to this socket. We can pass a backlog argument, which specifies how many pending connections can be queued before the system will refuse any new requests.
- `accept()` extracts the first connection request from the queue and creates a new connected socket, which is returned. If the socket is blocking, execution of the program will suspend on the call to `accept()` until a connection request has been received. The new socket has the same family and type as the original "welcoming" socket, but the original socket is not associated with the connection. The original socket remains available to receive additional connection requests.

  This is a typical point in the server execution to leverage multi-threading. A common approach is for the main thread to sit in an infinite loop, listening for connection requests on the welcoming socket. When a request is received, the main thread spawns a child thread to service the client, passing it the new connected socket.

4.2. UDP

Figure 3 shows the socket API function calls for a typical UDP client and server.

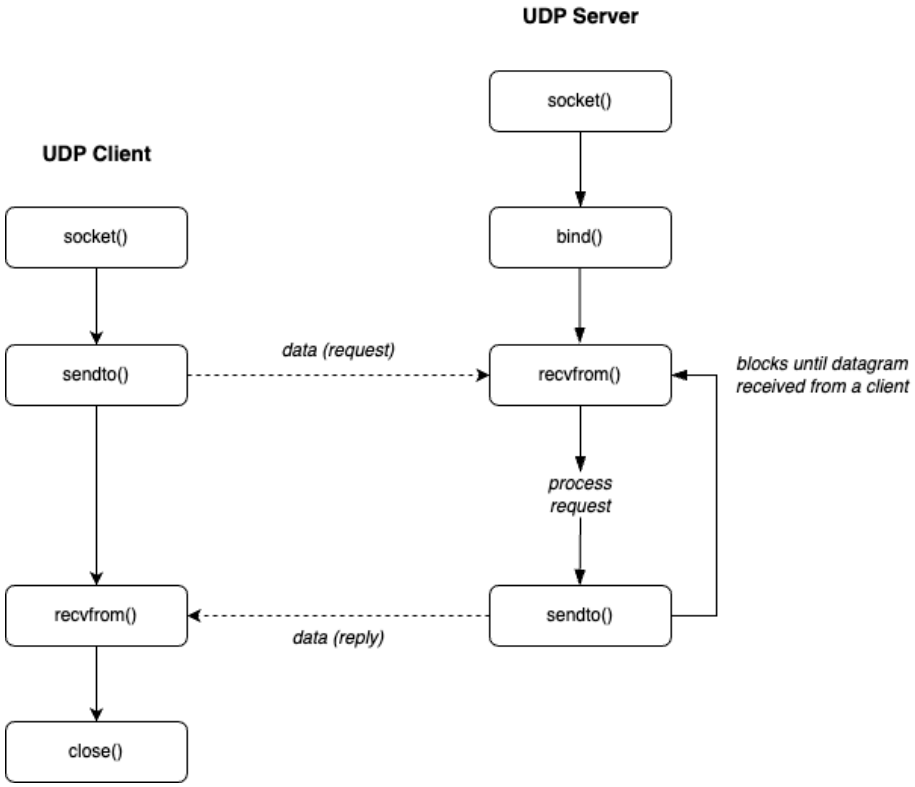


Figure 3: A typical UDP program flow.

Questions

1. How does Figure 3 compare to Figure 2? What are some of the fundamental differences between TCP and UDP?
2. Why does neither figure show the server socket (in the case of TCP, the server's "welcoming socket") being closed?

Answer

TCP is connection-oriented, reliable, and stream based. By contrast, UDP is connection-less, unreliable, and datagram based.

As we can see from the figure, the client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto()` function, which requires the address of the destination as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the `recvfrom()` function, which waits until data arrives from some client. `recvfrom()` returns the address of the client (IP and port number), along with the datagram, so the server can send a response to the client.

Neither figure shows the server socket being closed to merely indicate that servers are typically "always-on". In principle, once it's running, it should run forever. Of course, this isn't true in practice, and indeed the server code would include a call to `close()` the socket whenever it's taken offline.

5. Building Our Own Client-Server Application

5.1. The Scenario

The fictional CSEBank is a bricks-and-mortar bank used to store the discretionary budgets for each course. CSE has decided to modernise and migrate the bank online.

Each account held by CSEBank has three associated attributes:

1. `account`: A unique name that identifies the account. For example, `comp3331` has an account.
2. `password_hash`: A 40 character hex-string, representing a 20 byte SHA-1 cryptographic hash of the account's password. It's generally a bad idea to ever store passwords directly. For example, the password for `comp3331` is `TheForceIsStrongInThisOne`, which translates to `62d8447a4724bb8aa5a3c92a901d7e43773b108c`.
3. `balance`: The current balance of the account.

Fortunately developers have already built the banking system itself, and have provided an API that allows us to initialise the online bank, by reading in existing account information from a tab-separated file. Each line of the file contains the `account`, `password_hash`, and `balance` of an account. Additionally, functionality is provided to:

- Open a new account.
- Get the balance of an existing account.
- Transfer funds between two accounts.

CSE has tasked us with exposing this functionality by building a server application for the banking system, and a client application to interact with that system. Both applications are already partly written, it's mainly the socket programming that remains to be done.

5.2. The Protocol

The applications are to run over UDP, using a simple line-based, ASCII text protocol. Unlike HTTP, which uses Windows style `\r\n` as an end-of-line sequence, CSE favours the Unix philosophy, and wishes to use a single `\n`.

The first three lines of any request follow the format:

```
operation
account
password_hash
```

Where `operation` is one of:

- `open`
- `balance`
- `transfer`

All responses are a single line, indicating the outcome of the request.

Open Account

For example, for Alice to open an account with password “AliceLovesBob!”:

```
open
alice
3df8fb541e9cc6f202cbc6e813a910102d2bedba
```

Client code is already provided to handle the password hashing, but a quick way to check the SHA-1 hash of a string is to open a terminal and run:

```
$ echo –n 'AliveLovesBob!' | sha1sum
```

If the account was opened successfully, the server response should be:

```
successful
```

However, if an account with the same name already exists, the response should be:

```
account already exists
```

Get Balance

For example, for Alice to check the balance her new account:

```
balance
alice
3df8fb541e9cc6f202cbc6e813a910102d2bedba
```

If the account exists, and the password hash matches, the server response should be:

```
0.00
```

Otherwise, the response should be:

```
not authorised
```

Transfer Funds

The client request includes two additional lines, and should follow the format:

```
transfer
source_account
source_password_hash
destination_account
amount_to_transfer
```

For example, for Alice to send $100.00 to Bob (assuming she had $100.00):

```
transfer
alice
3df8fb541e9cc6f202cbc6e813a910102d2bedba
bob
100.00
```

If both accounts exists, the password hash for the source account matches, and the source account has sufficient funds, then the server response should be:

```
successful
```

Otherwise, in the case that one of the accounts doesn't exist or the password hash doesn't match:

```
not authorised
```

And in the case of insufficient funds:

```
insufficient funds
```

5.3. The Initial Task

Most of the client and server code is already written. Download the starter code in your preferred language, as well as the accounts file, and complete each of the tasks below. You'll find corresponding `TODO` comments in the starter code, where you might implement each task.

Server

1. Create a UDP socket and bind it to the server port.
2. In the main server loop, (wait to) receive an incoming client request.
3. Once received, call the function responsible for processing requests, passing it the data and the client address.
4. Within the function responsible for processing requests:
   4.1. Decode the data into string form (not really necessary in C).
   4.2. Parse the request to extract the first three lines, i.e. the operation, account, and password.
   4.3. Call the appropriate bank function, based on the operation, which in the case of a transfer will require further parsing of the request.
   4.4. Encode the bank's response and send it back to the client.
5. To provide an audit trail (and assist with debugging), add some server logging. In particular, before sending each response, log (to the terminal) a single line that includes a timestamp, the client address, the operation requested, the associated account, and outcome of the operation (i.e. the response).

Client

The structure of the client is more like a command line tool than a traditional client. Rather than being interactive, it takes the necessary command-line arguments for a particular operation, and given the operation, calls the appropriate function. Within each function:

- Construct and encode the appropriate request message.
- Create a UDP socket.
- Send the message to the server.
- Wait for a response.
- Decode and print the response.

Note, there's a lot of code repetition here. This is mainly to become familiar with socket programming, though you can write one function, copy and paste into the next, and make the necessary adjustements. A better structure would probably be to refactor it so the only difference is in request construction, and there's a single code path to encode the request, create the socket, send the request, receive, decode and print the response.

Once you have completed these tasks, using the client you should:

- Check the balance of the `comp3331` account.
- Open your own account.
- Transfer funds from the `comp3331` account to your account.
- Check the balance of your account.

5.4. Handling Multiple Clients

Rather than process each client iteratively:

6. When a request is received, spawn a new thread to process the request.
7. To provide some modicum of security while handling multiple concurrent clients, we'd like to rate limit requests. Add a short sleep of 50 milliseconds before sending each response.
8. Concurrency introduces possible race conditions. To ensure the bank remains in a consistent state:
   8.1. Add a lock primitive to the bank
   8.2. Minimally use it around the critical section of code in the function that transfers funds.

5.5. Account Security (Optional)

While the bank has the good sense to only store hashes of passwords, CSE has concerns that some account holders might be using weak passwords that may be subject to brute-force dictionary attacks. Mind you, the fact that we're transmitting these password hashes unencrypted is also a major security concern, but we'll leave that for another course (COMP4337/9337) to worry about.

With the permission of account holders, CSE has asked us to perform an attack to try and recover one or more passwords. **Of course, without such express permissions, anything of this nature would be a serious crime!**

A code stub is already provided in the client for a function that takes an account and a path to a wordlist file. The idea is to iterate over the file, hash each line, and send it to the server with a balance request for the account. Should one of the requests succeed, the password will have been found, and the loop can break.

Download the wordlist file that's provided with the starter code. This wordlist is a subset of the RockYou wordlist, which is well known in the security community. RockYou was a company that developed widgets and applications for social media platforms. The company used an unencrypted database to store user account data, including plaintext passwords, and following a data breach in 2009, millions of accounts were exposed. Because of the size and availability of the RockYou wordlist, it's commonly used in dictionary attacks.

Disclaimer: this is a very large and publicly available data set, and we are not responsible for any passwords that you may consider vulgar or offensive. This is purely a learning exercise.

A script is also provided to split the wordlist into multiple subsets. This will allow you to perform the attack more efficiently, and test out how well the server handles concurrency. Split the wordlist into as many files as clients you'd like to run (say 3-5), select a particular bank account to try and crack, then run each client with a different chunk of the original wordlist.

Tip: if you're feeling impatient, consider reducing or removing the artificial delay that was added to the server.