Objectives | Summary | Details | Submission | Grading

Final Project: Client-Server

DUE: August 14, 2019, EOD.

OBJECTIVES

- Experience building a client-server application
- Work with multiple processes

Beej's Guide to Networking is a helpful resource for this

SUMMARY

Make the search engine (movie database) available as a server.

HOW DOES THIS ALL FIT TOGETHER?

In the bigger scheme of things, we're putting pieces together to build a search engine. We're going to allow our indexed to be searched via a client. We won't actually be building a version of Google, but this is similar to what internet search engines do: create an index of the world (wide web), and then when you search for a term, it returns back all the pages that contain it. (Google's a little smarter than that, but it's the general idea-- and definitely where search engines started!).

In our server, we're going to let clients connect, and return back rows of data (movie records) that contain our search term.

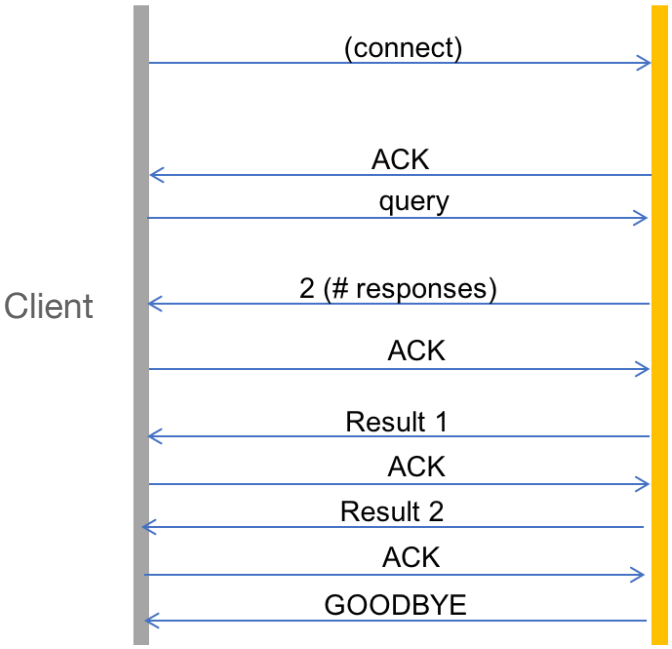
CLIENT/SERVER PROTOCOL

Whenever we have a client and server talking to each other, we define a protocol so both the server and client know what to expect from each other and therefore communicate effectively. The protocol for this project is fairly simple.

Timeline/process:

- Server accepts connection
- Server sends ACK to client (using SendAck) ← function
- Server recieves query term (as a string)
- Server sends client number of responses (as an int)
- Server receives ACK from client (sent using SendAck)
- For each response:
    - Server sends response to client (as a string)
    - Server receives ACK from client (sent using SendAck)
- Server sends GOODBYE (sent using SendGoodbye)
- Server and Client close the connection.

This is represented in the diagram below:



Client

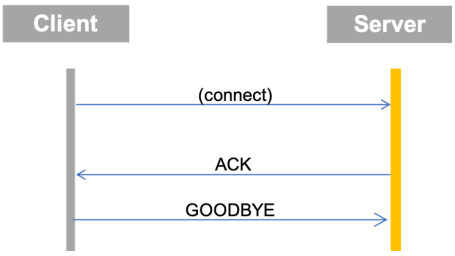
Server

- Server accepts connection
- Server sends ACK to client (using SendAck)
- Server recieves query term (as a string)
- Server sends client number of responses (as an int)
- Server receives ACK from client (sent using SendAck)
- For each response:
  - Server sends response to client (as a string)
  - Server receives ACK from client (sent using SendAck)
- Server sends GOODBYE (sent using SendGoodbye)
- Server and Client close the connection.

The above timeline was written in terms of the server. We can re-write it from the perspective of the client:

- Client calls connect
- Client recieves ACK
- Client sends query (as string)
- Client reads int describing number of results
- Client sends ACK (sent using SendAck)
- For each response:
  - Client reads response from server (as a string)
  - Client sends ACK to server (using SendAck)
- Client reads GOODBYE from server
- Client closes the connection and waits for another query.

Before we do anything though, we want the client to have a way to check that the server is actually alive, without doing anything else. There's a protocol for that too:



Here, the server is waiting for connections. When the client connects, the server sends an ACK back. Then, the client sends "GOODBYE". This represents a branch in the server: the connections start out the same, if the client sends "GOODBYE", it just closes the connection; if it sends something else, we treat that as a query for the server to fulfill. (This does have the downside that you can't query for GOODBYE from the index!! In a real system this would be a bit different, but for various simplification reasons we're doing this for this project.)

specific kind of message and determine if a message is the kind of message expected.

Once again, you're provided with a starting framework, specifically with an indexer that is already working. Once again, you will need to work with the interface provided, but you should be familiar with it at this point. Specifically, you are provided with:

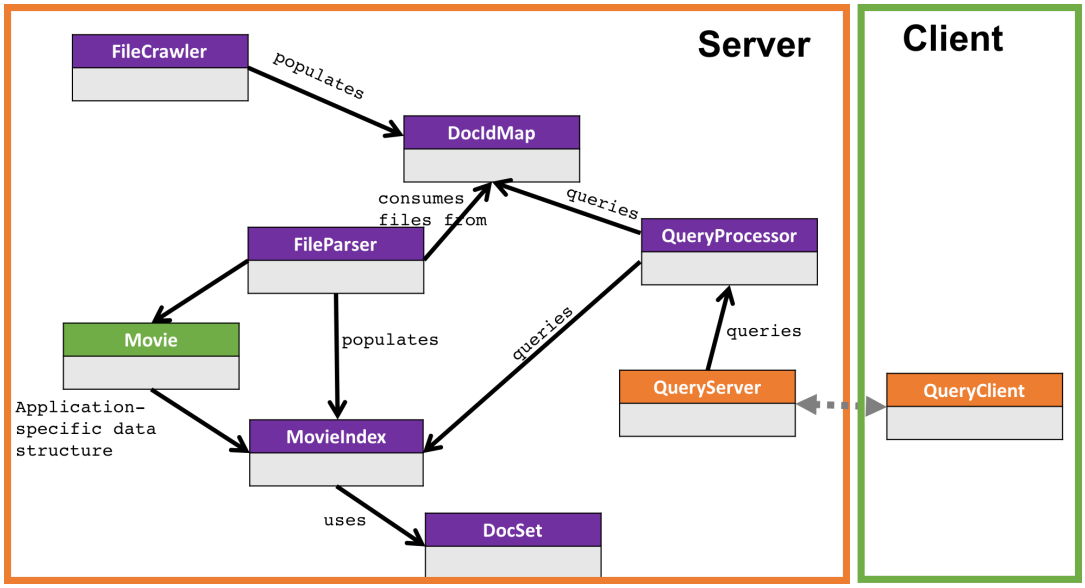
- `QueryClient.c`: Some structure to start the client code (Part 1).
- `QueryServer.c`: Some structure to start the server code (Part 2).
- `MultiServer.c`: Some structure to start the multi-process server code (Part 3).
- `QueryProtocol.h`: The header that encapsulates the protocol between client and server (in the includes folder)
- `libSystem.a`: A library containing all of the functionality that we've already built this semester.
- `includes`: .h files that are the function prototypes/headers for the search engine.
- `binaries`: executables for the server, client and multiserver for you to run to test or explore.

Most of the code (libSystem.a and .h) files are the same as provided for A9, but there are a couple of additions to make it easier for your server implementation:

- `QueryProcessor.h` includes a new function to get the number of results for a query.
- `Util.h` contains a `CopyRowFromFile` function to get the search result.

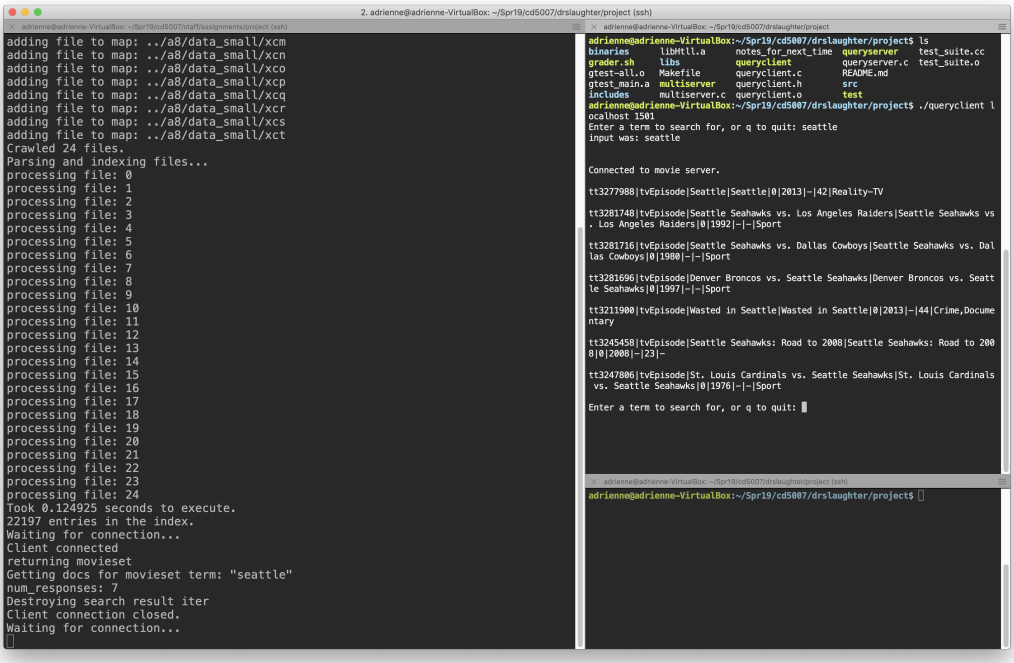
WHAT DOES OUR SYSTEM LOOK LIKE?

The following picture is an overview of the system components:

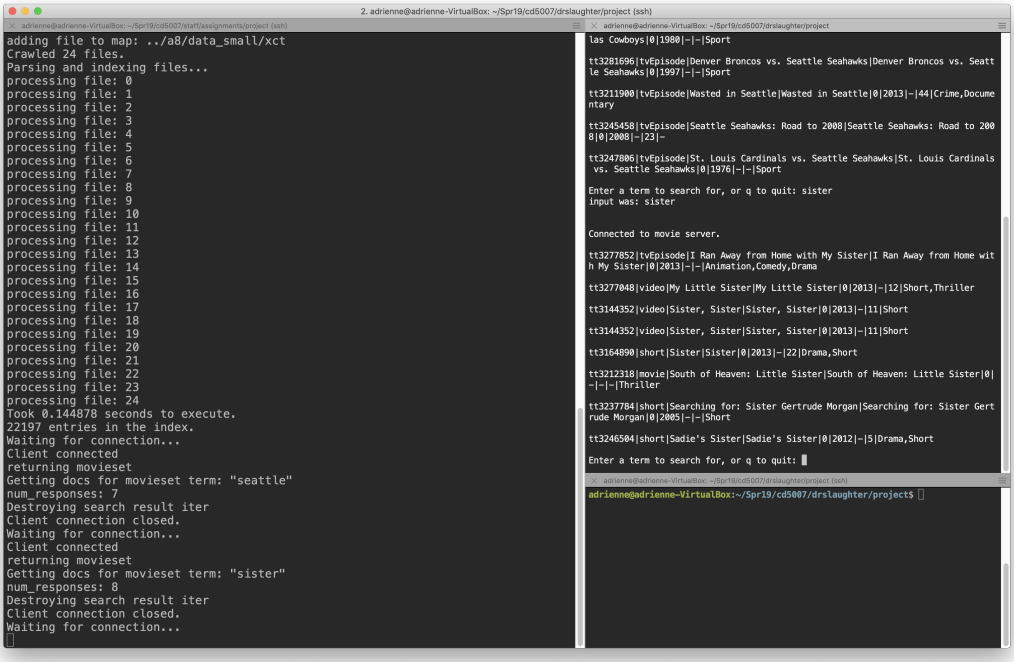


You'll be building the pieces marked in orange: the server and the client.

Below is a sample invocation. The server is on the left (invoked with `./queryserver data_small/ 1501`), and the client is on the right (invoked with `./queryclient localhost 1501`). Once the client is connected, it starts getting queries from the user. The client then connects to the server, sends the query, gets the results, and prints them out to the user. Once the results are returned, the server disconnects the client



The client is still available to get a new query from the user. Below shows when the user made a second query. Again, with each query, the client connects, gets results, and then disconnects.



(Note: the entry "Sister, Sister" is in the results twice because the word 'sister' is in the title twice. A known weakness to this implementation.)

The client should continue until the user enters 'q' to quit. At that point, you can type `Ctrl-C` to shutdown the server.

Clients are easier to write than a server, so let's start with that.

Build your client based on the code provided for you in `QueryClient.c`.

Invoking the client:

```
./queryclient [IP address] [port number]
```

Your client should include `QueryProtocol.h` (in the `includes` folder) to share the protocol interface with the server. You are able to use all of the functions available in the protocol to build your client.

The most relevant files:

- QueryClient.c
- QueryClient.h
- QueryProtocol.h

To start testing your client, you might want to set up a simple echo server. We will also have a sample server running for you to connect against. You can also run the provided binary locally.

Requirements

Your client must satisfy these requirements:

- After started, the client should stay running until the user terminates it.
- Identify and implement a graceful way to with a server that is not accepting connections. This could be very simple or more complex, but your program should not crash. (It can terminate intentionally, with an informative message.)
- The client should make a new connection for every query, and close it properly.
- Limit query from user to 100 characters.
- Makefile has a target `client` which generates an executable `queryclient` (provided; if you need modifications that is fine)
- Client is started by calling `queryclient [ipaddress] [port]`
- Note: `localhost` is the same as IP `127.0.0.1`
- If the wrong input is provided when launching the program, print out instructions to run it properly.
- Provide guidance on how to query to the user.
- To test, you might want to use an echo server or the provided binaries.
- The Makefile includes a target called "runclient". Feel free to modify it to run with your preferred parameters for testing.
- Commit your code! And push it to github, if you haven't done this yet.
- Don't forget to check Valgrind for mem leaks.
- The example output is EXAMPLE. There is lots of room for improvement to make it a better experience.

STEP 2: WRITE A SIMPLE SERVER (SINGLE CLIENT)

Now that we have a client, we can build the server. Just as with the client, the server will follow the protocol provided above. In this step, just build a server that allows one connection, similar to what we did in class.

Requirements:

- There is a target in the Makefile called "server", that builds the server. Modify it as/if necessary.
- Create an executable called "queryserver".
- Run the server like this: `./queryserver -f [datadir] -p [port]`, where the -f flag specifies which directory to index, and the -p flag specifies which port the server should be listening on for connections. We'll continue to use the `data_tiny`, `data_small` and `data` directories for data. Copy them into your project directory if you want, or modify the path to point at your data folders.
- If improper arguments are provided when launching, handle it gracefully and provide a message specifying correct usage.
    - Handling graceful could be either providing defaults or specifying the appropriate usage.
- Run your server with valgrind to ensure you're handling memory properly. There should be no memory in use after the server is killed.
- The server can be killed gracefully by hitting `ctrl+C`. There is a function in `QueryServer.c` (`void sigint_handler(int sig)`) that captures the `ctrl+C` event and does some cleanup. You might want to take advantage of this, but you don't need to worry about the details of it yet.

Finishing Step 2

When you're done with this step:

- Commit your code! And push it to github, if you haven't done this yet.
- Run your server and run your client; test it.
- You can also test it with the provided client binary.
- Note, your server should run flawlessly with the provided client, and your client should run flawlessly with the provided server. It is *NOT* okay if your server and client work well with each other, but not with the provided server and client.
- Run your code on the provided "movies-small" directory.
- You might want to modify the "runserver" target in your Makefile that runs your client with the correct parameters.

STEP 3: MAKE YOUR SERVER HANDLE MULTIPLE CONNECTIONS

If we're looking at a webserver, it doesn't only allow one client to connect at a time. In fact, multiple clients can connect at once, and the server is able to give what each client needs at the same time. How does it do that?

In this part, we're going to take the server you created in the previous step and modify it to allow it to serve multiple clients. To allow multiple clients, we need to either handle each connected client in a new thread or new process. In the previous assignment we explored multi-threading, so in this project we'll explore using processes.

- Some code is provided for you to get started with: `MultiServer.c`
- In `MultiServer.c` you'll see some helper methods:
    - `sigchld_handler` makes sure that any child processes get cleaned up.
    - sigint_handler captures the ctrl+c command to allow us to clean up any mallocs on quit

    - `Cleanup` takes care of the index and such, feel free to add code here to do any other clean up you need.
    - `Setup` creates the movie index from the provided data directory. You shouldn't need to add anything here.
- Beej's Guide to Networking (see the resources) is an excellent resource.

Requirements

Your client must satisfy these requirements:

- Makefile has a target `multiserver` which generates an executable `multiserver`
- Server is started by calling `multiserver -f [dirname] -p [port]`
- There is an optional flag `-d` for debugging. When the `-d` is passed when starting the multiserver, please ask the new process to sleep for 10 seconds before handling the query.
- If the wrong input is provided, print out instructions to run it properly.
- Run your code with valgrind to ensure you handled memory properly.
- You might want to introduce sleeps in your code to help test multiple connections.
- Commit your code! And push it to github, if you haven't done this yet.

SUBMISSION

Submit your assignment by pushing your code to Github and submitting the .zip file (with ONLY the project folder) to the handins server. Tag it with the tag `final-project`.

FAQs

Here are some questions/issues that have come up in previous semesters.

- What happens if the protocol breaks? That is, there is some error, not getting ACK at the right time, etc?
  If the protocol ever breaks, you should terminate the connection (either as the server or client), and prepare for the next connection (either as server to listen for a new one, or as client for a new query). On either side, it might be helpful to print out a descriptive message for trouble-shooting.
- Don't forget to free your address structs.
  See `freeaddrinfo`.
- Close your sockets in both the parent and child processes.
  See https://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html (Enhancements to Server Code) for more info.
- Why does it look like my server is getting `ACKDBYE`?
  Check that you're null-terminating strings you get from the `read` function.
- Why does my server seem to be sending the result plus garbage to the client?
  Be sure to specify the number of bytes to send when you call `write`. `sizeof` != `strlen`.
- Why do I keep getting "address already in use"? The OS isn't sure if the port is still in use, or it's in use by something else. If you run your server once and it

http://beej.us/guide/bgnet/html/multi/syscalls.html#bind). Make sure your code has something like this:

```
int yes=1;

// lose the pesky "Address already in use" error message
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof yes) == -1)
    perror("setsockopt");
    exit(1);
}
```

More Details/Examples