Program 4: Peer-to-Peer Registry

Due: By 23:59 Sunday, May 8

Introduction

In this program you will accomplish several goals:

- Expand your knowledge of the socket API
- Continue development in the peer-to-peer (P2P) application paradigm
- Design and implement a simple P2P registry application

All programming assignments must be done in pairs.

Requirements

In this program, you will implement the complement of the peer-to-peer (P2P) file sharing application you developed in Program 2 and Program 3. There are two roles for applications in a P2P network: the registry and the peer. Within each P2P network there are multiple peers, but only one registry, which keeps track of available peers and the files located at each peer (a centralized index). Peers that wish to share files contact the registry to: join the P2P network, announce the files available at the peer, find available peers, and locate files at other peers.

Your task is to create a registry application that interacts with existing peer applications according to the protocol defined in the Program 2 and Program 3 handouts. Your registry must be able to accept and respond to, when required:

- JOIN messages from peers wanting to join the P2P network.
- PUBLISH messages from peers in the P2P network.
- SEARCH messages from peers in the P2P network.

Unlike the peer from Program 2 and Program 3, your registry is **not** an interactive application. Users enter commands through peer applications only. The registry only responds to messages it receives from the network.

The next sections provide any clarifying details about the protocol developed as part of Program 2 and Program 3. All message formats and semantics are the same for Program 4.

A peer application is provided on Blackboard Learn for you to use when testing your registry application. If you successfully completed your Program 3 assignment, you could use that peer program as well. The peer application runs on `jaguar` and the virtual machine image used in lab; it might run on other Linux-based systems, but use it on these systems at your own risk. Provide the peer the same arguments as defined in Program 3. The peer source code will not be released.

Handling Multiple Sockets

Your registry must handle multiple sockets during its execution: one socket upon which to accept new peer connections and one socket for every connected peer. *This is a significant new step compared to Program 2 and Program 3.* Since `recv` is a blocking system call, you cannot call `recv` on a socket unless you know data is available. Otherwise, your registry would wait for data that might never arrive. `select` is the common way to determine which, if any, sockets have data available for processing. Note that `select` is useful for all your sockets (the accept socket and the peer sockets). `select` **Requirements**

- Do not use `FD_SETSIZE` in your program.
- Use only one call to select in your program.
- Do not use a timeout value (`timeout` should be NULL).

Keep in mind you may have a connected peer without knowing its ID. This happens between the time a peer connects to the registry and when it sends a JOIN message.

P2P Protocol Clarifications

Clarification about individual messages are provided in the following sub-sections.

JOIN

Your registry needs to track peers that JOIN the network so that the registry can handle SEARCH commands. For each peer that joins the network, you will need to track the files PUBLISHed by the peer, the peer's IP address, the peer's port number, and the socket descriptor for the socket connected to the peer. A `struct` will likely be helpful in storing this information for each peer. One example struct *might* be:

```
struct peer_entry {
    uint32_t id;                              // ID of peer
    int socket_descriptor;                    // Socket descriptor for connection to peer
    char files[MAX_FILES][MAX_FILENAME_LEN]; // Files published by peer
    struct sockaddr_in address;               // Contains IP address and port number
};
```

Refer to the Identifying Remote Addresses section for information on how to determine a peer's IP address and port number.

*Your registry only needs to support at most **5** peers, but you should strive to support an arbitrary number of peers.*

Each peer must send a JOIN for each connection. Your registry should not maintain state for a peer that disconnects.

PUBLISH

Your registry needs to track the files available at each peer by saving the filenames PUBLISHed by each peer.

Your registry may make the following assumptions, but you should strive to gracefully handle all scenarios. *You may assume that:*

- *peers will JOIN the network before sending a PUBLISH message,*
- *peers will only send one PUBLISH message,*
- *peers will PUBLISH at most 10 filenames,*
- *and filenames will not be longer than 255 characters (including NULL).*

SEARCH

Your registry must respond to SEARCH messages from peers and send back the correct response.

All values in the SEARCH response must be in network byte order.

Additional Requirements

Additional requirements for this program include:

- You may not use any form of sleep function, or equivalent, in this assignment and you may not use any socket flags (i.e., no `MSG_WAITALL`). You may not use `ioctl(2)` or equivalent functions. You may not use `FD_SETSIZE`.
- Do not use any timeout value with `select`. Your registry must provide NULL as the `timeout` argument. Your registry has nothing to do without a socket to handle.
- Use at most one call to `select` in your program.
- All submissions must pass compilation checks before they will be graded. Run the script `/user/home/kkredo/public_bin/program4_check` on `jaguar` to check your submission. Run the script without arguments or with the argument `help` for directions.
- Submit your program source files through Blackboard Learn. Do not submit archive (zip/tar) files or files provided to you.
- You must include a Makefile with your submission, which builds the program by default and removes all generated files with the target `clean`. The registry executable must be named `registry`.
- The registry must accept one command line argument: the port number to use.
- Your program must compile cleanly on `jaguar` or the machines in OCNL 340 and you must use the gcc/g++ argument `-Wall`. You may not use `-w` to disable warnings.
- Check all function calls for errors. Points will be deducted for poor code.
- Put both group member names, the class, and the semester in the comments of all files you submit.

Output

Your registry must print out summary messages that indicate what command and arguments it received. *Your registry should print **only** these messages.* The general format for these summary messages are `TEST] <cmd> <args>`, where `TEST]` is printed at the start of every summary message, `<cmd>` is an indication of what command the registry received, and `<args>` are the arguments for that command. Each summary message must: be on one line, have spaces separating each part, and include fields printed in host byte order. The program outputs below provide an example, where the peer PUBLISHes the files 'a.txt', 'b.pdf', and 'c.tgz'. You may also run the `p4_peer` program, available on Blackboard Learn, with the `-t` argument for further summary message examples.

Peer Input/Output

```
$ ./peer my.server 5000 16
Enter a command: JOIN
Enter a command: SEARCH
Enter a file name: found.pdf
File found at
 Peer 34
 201.18.192.56:8225
Enter a command: PUBLISH
Enter a command: SEARCH
Enter a file name: missing.txt
File not indexed by registry
Enter a command: EXIT
$
```

Registry Output

```
TEST] JOIN 16
TEST] SEARCH found.pdf
TEST] PUBLISH 3 a.txt b.pdf c.tgz
TEST] SEARCH missing.txt
```

Evaluation

Your program will be evaluated based on:

- 10 points: Registry accepts peer connection
- 10 points: JOIN command accepted and printed correctly
- 10 points: PUBLISH command accepted and printed correctly
- 25 points: Registry handles multiple simultaneous peers
- 10 points: SEARCH command accepted and printed correctly
- 25 points: SEARCH response sent correctly
- 10 points: All responses sent as single packets

Identifying Remote Addresses

Your registry needs to determine the IP address and port number for peers within the P2P network. This is an operation performed by the `getpeername` function. The code segment below retrieves the connection information into the variable `addr` for the *already connected* socket `s`.

```
struct sockaddr_in addr;
socklen_t len = sizeof(addr);
int ret = getpeername(s, (struct sockaddr*)&addr, &len);
```

Those using the C++ `net_socket` class should explore the available functions.

Hints

- Beej's Guide to Network Programming[1] is a valuable resource.
- If the registry produces an "Address in use" error, then pick a different port number or wait for a couple minutes and try again.
- Ensure you pay attention to byte ordering. The byteorder(3) man page will be useful.
- Use Wireshark to help debug!
- Data type sizes vary across machines and operating systems in C and C++. Consider using defined size types, such as `uint32_t` for unsigned integer of 32 bits.
- Test and debug in steps. Start with a subset of the program requirements, implement it, test it, and then add other requirements. Performing the testing and debugging in steps will ease your efforts. In general, the Evaluation requirements are ordered in a way to ease this implementation process.
- You may have an optional command line argument that enables debugging output. The debug argument must be optional (meaning that your program must run when it is not specified) and the debug output in your code must be cleanly organized.

[1] https://beej.us/guide/bgnet/