

Section 3: Sockets, RPC

CS 162

July 1, 2020

Contents

1	Vocabulary	2
2	Warmup	3
2.1	Storing Ints	3
3	Socket Programming	4
3.1	Multi-threaded Echo Server	4
4	RPC	6
4.1	Selecting Functions	6
4.2	Reading Arguments	7
4.3	Handling RPC	7
4.4	Call Stubs	8
4.5	Handling failure	10

1 Vocabulary

- **Socket** - Sockets are an abstraction of a bidirectional network I/O queue. It embodies one side of a communication channel, meaning that two must be required for a communication channel to form. The two ends of the communication channel may be local to the same machine, or they may span across different machines through the Internet. Most functions that operate on file descriptors like `read()` or `write()` work on sockets. but certain operations like `lseek()` do not.
- **file descriptors** - File descriptors are an index into a file-descriptor table stored by the kernel. The kernel creates a file-descriptor in response to an `open` call and associates the file-descriptor with some abstraction of an underlying file-like object; be that an actual hardware device, or a file-system or something else entirely. Using file descriptors, a process's `read` or `write` calls are routed to the correct place by the kernel. When your program starts you have 3 file descriptors.

File Descriptor	File
0	stdin
1	stdout
2	stderr

- **RPC** - Remote procedures calls are a technique for distributed computation through a client server model. This process effectively calls a procedure on a possibly remote server from a client by wrapping communication over the network with wrapper stub functions. This six steps are:
 1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
 2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshaling.
 3. The client's local operating system sends the message from the client machine to the server machine.
 4. The local operating system on the server machine passes the incoming packets to the server stub.
 5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshaling.
 6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction
- **Endianness** - The order in which the bytes are stored for integers that are larger than a byte. The two variants are big-endian where byte the most significant byte is at the lowest address and the least significant byte is at the highest address and little-endian where the least significant byte is at the lowest address and the most significant byte is at the highest address. The network is defined to be big-endian whereas most of your machines are likely little-endian.

2 Warmup

2.1 Storing Ints

You are working for BigStore and your boss has tasked you with writing a function that takes an array of ints and writes it to a specified file for later use. He also informs you that a major bug has been found in the C file pointer library and wants you to use file descriptors. Fill in the following function:

```
void write_to_file(const char *file, int *a, int size) {
    int write_fd = open(_____, _____);

    char *write_buf = _____
    int buf_size = _____
    int bytes_written = 0;

    // Write a to file.
    _____
    _____
    _____
    close(write_fd);
}
```

Now, write the function that retrieves previously saved integers and places them in a int array.

```
void read_from_file(const char *file, int *a, int size) {
    int read_fd = open(_____, _____);

    char *read_buf = _____
    int buf_size = _____

    // Read a from a file.
    _____
    _____
    _____
    _____
    close(read_fd);
}
```

Your coworker opens up one of the files that you used to store ints on his text editor and complains its full of junk! Explain to him why this might be the case.

3 Socket Programming

3.1 Multi-threaded Echo Server

Please look at the three versions of server code provided with Lecture 5. The first version uses a single process and single thread, the second version sequentially handles each client connection in a child process, and the third version allows child processes to handle connections concurrently.

Write a fourth version of the server implementation that uses multiple threads in a single process. Each connection is handled in its own thread, and threads should be allowed to handle connections concurrently.

```

#define BUF_SIZE 1024

struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int rv = getaddrinfo(NULL, port, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}

void *serve_client(void *client_socket_arg) {
    int client_socket = (int)client_socket_arg;
    char buf[BUF_SIZE];
    ssize_t n;

    while ((n = read(client_socket, buf, BUF_SIZE)) > 0) {
        buf[n] = '\0';
        printf("Client Sent: %s\n", buf);

        if (write(client_socket, buf, n) == -1) {
            -----;
            -----;
        }
    }

    if (n == -1) {
        -----;
        -----;
    }

    -----;
}

```

```

} -----;

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <port>\n", argv[0]);
        return 1;
    }

    struct addrinfo *server = setup_address(argv[1]);
    if (server == NULL) {
        return 1;
    }
    int server_socket = _____(server->ai_family,
                                   server->ai_socktype, server->ai_protocol);
    if (server_socket == -1) {
        return 1;
    }

    if (_____ (server_socket, server->ai_addr,
                server->ai_addrlen) == -1) {
        return 1;
    }
    if (_____ (server_socket, 1) == -1) {
        return 1;
    }

    while (1) {
        int connection_socket = accept(server_socket, NULL, NULL);
        if (connection_socket == -1) {
            perror("accept");
            pthread_exit(NULL);
        }

        pthread_t handler_thread;
        int err = pthread_create(_____);
        if (err != 0) {
            _____;
        }
        pthread_detach(handler_thread);
    }
    _____;
}

```

4 RPC

To explore the process for performing RPC we will consider implement the server end for two procedures. We want to implement the server side of an RPC version of the following code

```
// Returns the ith prime number (0 indexed)
uint32_t ith_prime (uint32_t i);

// Returns 1 if x and y are coprime, otherwise 0.
uint32_t is_coprime(uint32_t x, uint32_t y);
```

Assume the server has already implemented `ith_prime` and `is_coprime` locally.

4.1 Selecting Functions

As a first step we receive data from the client. How do we decide which procedure we are executing? Provide a sample header file addition that could be used to indicate this.

4.2 Reading Arguments

When examining the arguments for your two functions you notice that the arguments require either 8 or 4 bytes, so you believe you can handle either case by attempting to read 8 bytes using the code below.

```
// Assume dest has enough space allocated
void read_args (int sock_fd, char *dest) {
    int byte_len = 0;
    int read_bytes = 0;
    while ((read_bytes = read (fd, dest, 8 - byte_len)) > 0) {
        byte_len += read_bytes
    }
}
```

However when you implement it you notice that for some inputs your server appears to be stuck? Why might this be happening and for which inputs could this happen?

4.3 Handling RPC

Realizing your previous solution was insufficient you decide to implement a slightly more complicated protocol. You settle on the following steps for the client:

1. The client sends an identifier of the function it wants as an integer (0 for `ith_prime`, 1 for `is_coprime`).
2. The client sends all the bytes for all the arguments.

The server then takes the following steps:

1. The server reads the identifier.
2. The server uses the identifier to allocate memory and set the read size.
3. The server reads the remaining arguments.

Complete the following function to implement the server side of handling data. You may find `ntohl` useful.

```

// Function to implement the server side of the protocol.
// Returns whether or not it was successful and closes the socket when finished.
// Assume get_sizes loads all our sizes based on id and call_server_stub selects
// our host function
void receive_rpc (int sock_fd) {
    uint32_t id;
    char *args;
    size_t arg_bytes;
    char *rets;
    size_t ret_bytes;
    int bytes_read = 0;
    int bytes_written = 0;
    int curr_read = 0;
    int curr_write = 0;
    while ((
        -----
        -----) > 0) {
        bytes_read += curr_read;
    }
    -----;
    // Allocates sizes based on our id (not a real library function)
    get_sizes (id, &args, &arg_bytes, &rets, &ret_bytes);
    bytes_read = 0;
    while ((
        -----
        -----) > 0) {
        bytes_read += curr_read;
    }
    // Calls the appropriate server stub based on id (not a real library function)
    call_server_stub (id, args, arg_bytes, rets, ret_bytes)
    while ((
        -----
        -----) > 0) {
        bytes_written += curr_write;
    }
    -----;
}

```

4.4 Call Stubs

Finally we want to implement the call stubs, which are function wrappers between each individual function we support and the generic RPC library. For example these are the client stubs for our functions

After the raw data is processed we need to perform the actual computation and return the result to the client. To do this we introduce a stub procedure which unpacks our arguments, calls the procedure to execute on the server, and finally packs the return results to give to the transport layer.


```

// addr is used for setting up our socket connection
uint32_t ith_prime_cstub (struct addrinfo *addr, uint32_t i) {
    uint32_t prime_val;
    i = htonl(i);
    call_rpc (addr, RPC_PRIME, (char *)&i, 1, (char *) &prime_val, 1);
    return ntohl(prime_val);
}

uint32_t is_coprime_cstub (struct addrinfo *addr, uint32_t x, uint32_t y) {
    uint32_t coprime_val;
    uint32_t args[2] = {htonl(x), htonl(y)};
    call_rpc (addr, RPC_COPRIME, (char *) args, 2, (char *) &coprime_val, 1);
    return ntohl(coprime_val);
}

```

`call_rpc` is our generic rpc handler that our stubs provide an abstraction around. Notice that we marshal arguments with `htonl()` and unmarshal them with `ntohl()`.

Implement `ith_prime_sstub` and `is_coprime_sstub` which should unmarshal the arguments, call the implementation functions, and finally marshal the return data.

```

// Both functions take in args in network order and should place
// return value(s) in rets in network order

void ith_prime_sstub (char *args, char *rets) {
    -----
    -----
    -----
    -----
    -----
}

void is_coprime_sstub (char *args, char *rets) {
    -----
    -----
    -----
    -----
    -----
}

```

4.5 Handling failure

What are some ways a remote procedure call can fail? How can we deal with these failures?