Section 3: Sockets, RPC

CS 162

July 1, 2020

Contents

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);
}
```

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

  char *write_buf = (char *) &a[0];
  int buf_size = size * sizeof(int);
  int bytes_written = 0;

  while (bytes_written < buf_size) {
    bytes_written += write(write_fd, &write_buf[bytes_written], buf_size - bytes_written);
  }
  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);
}
```

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

  char *read_buf = (char *) &a[0];
  int buf_size = size * sizeof(int);

  int bytes_read = 0;
  int total_read = 0;
  while ((bytes_read = read(read_fd, &read_buf[total_read], buf_size - total_read)) > 0) {
    total_read += bytes_read;
  }
  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.

```
Currently, we are reading and writing the contents of memory directly to disk.
This is convinient for us, because we do not have to any parsing of the input.
However, the memory representation of an int array is unlikely to be human readable.
```

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) {
            close(client_socket);
            pthread_exit(NULL);
        }
    }
    if (n == -1) {
        close(client_socket);
        pthread_exit(NULL);
    }

    close(client_socket);
    pthread_exit(NULL);
```

```
}

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 = socket(server->ai_family,
                           server->ai_socktype, server->ai_protocol);
    if (server_socket == -1) {
        return 1;
    }
    if (bind(server_socket, server->ai_addr,
         server->ai_addrlen) == -1) {
            return 1;
    }
    if (listen(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(&handler_thread, NULL,
                serve_client, (void *)connection_socket);
        if (err != 0) {
            printf("pthread_create: %s\n", strerror(err));
            pthread_exit(NULL);
        }
        pthread_detach(handler_thread);
    }
    pthread_exit(NULL);
}
```

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.

We need to provide a unique id for each function and transmit it in at the start of our RPC message. There are many ways to do this, for example if these two functions were a comprehensive list of the rpc calls we needed to support we could provide an enum or we could use defines with unique values for example.

```
enum RPC_ID {
    PRIME_ID = 1,
    COPRIME_ID = 2
};
```

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?

The read is trying to return 8 bytes from the socket or indicate there is no more data. However, when the socket is empty the read will only return due to failure (or if the socket is closed). So long as the connection stays open the reads will never fail and will block instead.

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 ((curr_read = read (sock_fd, ((char *)&id) + bytes_read,
            sizeof (uint32_t) - bytes_read)) > 0) {
        bytes_read += curr_read;
    }
    id = ntohl (id_data);
    get_sizes (id, &args, &arg_bytes, &rets, &ret_bytes);
    bytes_read = 0;
    while ((curr_read = read (sock_fd, &args[bytes_read], arg_bytes - bytes_read)) > 0) {
        bytes_read += curr_read;
    }
    call_server_stub (id, args, arg_bytes, rets, ret_bytes)
    while ((curr_write = write (sock_fd, &rets[bytes_written],
            ret_bytes - bytes_written)) > 0) {
        bytes_written += curr_write;
    }
    close (sock_fd);
}
```

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.

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

uint32_t is_coprime_cstub (struct addrinfo *addrs, uint32_t x, uint32_t y) {
    uint32_t coprime_val;
    uint32_t args[2] = {htonl(x), htonl(y)};
    call_rpc (addrs, 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.

```
void ith_prime_sstub (char *args, char *rets) {
    uint32_t* args_ptr = (uint32_t *) args;
    uint32_t i = ntohl (args_ptr[0]);
    uint32_t result = ith_prime (i);
    result = htonl (result);
    memcpy (rets, &result, sizeof (uint32_t));
}


void is_coprime_sstub (char *args, char *rets) {
    uint32_t* args_ptr = (uint32_t *) args;
    uint32_t x = ntohl (args_ptr[0]);
    uint32_t y = ntohl (args_ptr[1]);
    uint32_t result = is_coprime (x, y);
    result = htonl (result);
    memcpy (rets, &result, sizeof (uint32_t));
}
```

4.5 Handling failure

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

1. Transport failure (i.e. the client can fail to connect to the server, or fail to send data to the server). This can be handled via retrying (the case of dropped packets is handled by reliable transport).
2. The server could crash/error before completing the procedure. *This means that all the client sees is a severed connection.* In this case an ACID compliant transaction will never complete, and it's safe to retry the request.
3. The server could crash have a transport error after completing the procedure, but before the client receives the results. *This means that all the client sees is a severed connection.* If we retry now, we have done the transaction twice! This is a good case for using idempotent operations.
4. (Alternative) The server could send the RPC result back to the client before committing the procedure. The client would have to tell the server when it has received the results. What if the server doesn't receive that message? This is the simple case of the Two General's Problem.

   A distributed design note: Notice that the complicated error handling here comes from stateful modifications. This is why, when designing a distributed system, idempotent operations are preferable when modifying state, and read-only operations are preferable to operations which modify state. Stateless operations (pure functions) are typically preferable to read-only operations (but this is more typically for performance and consistency reasons).