Design Doc: Multithreaded HTTP Server

Ryan Watkins
CruzID: Ryan Watkins

1. **Goal:**

The goal of this project is to create a multithreaded HTTP server that handles GET, PUT, and PATCH requests.

2. **Design**

The design of the project is outlined by the necessary modules outlined below.

2.1 Initialization:

First we need a file descriptor for the socket, we can get a reference descriptor from the return of socket() when we create a socket. Then we need to name the socket so we need to bind the socket to an address so that we can identify it. Lastly we need to listen on our socket for connection requests with listen(). Then if a connection is queued onto our backlog, we accept the connection and get a file descriptor

---

```
/* Starter code provided by the professor on piazza */
char * hostname;
char * port;
struct addrinfo *addrs, hints = {};
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo(hostname, port, &hints, &addrs);
int main_socket = socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
int enable = 1;
setsockopt(main_socket, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(enable));
bind(main_socket, addrs->ai_addr, addrs->ai_addrlen);
listen (main_socket, 16);
socket_FD = accept(main_socket, addrs->ai_addr, addrs->ai_addrlen);
```

2.2 Buffered Socket

In order to properly handle potentially fragmented data we'll need to buffer our socket. We can do this with a buffer struct with member values socketFileDescriptor, inputData[] and validData and member functions read_data(n), getValidData(), peek(), consume(n), and getSocketFileDescriptor(). Read_data will attempt to read n bytes from the socketFileDescriptor into inputData, getValidData returns the number of valid bytes in inputData, peek() returns a read-only pointer to inputData, and consume removes the first n bytes from inputData and shifts the remaining data to the front of the array.

2.3 Parse HTTP Headers

In order to parse a header we first need to obtain a valid header from our buffer. To do this we get a pointer to the first four elements of the inputData and compare it to the end of header HTTP sequence. If it doesn't match we increment the ptr and check again. If we run out of validData then there must not be a valid header in the buffer so we return null.

In order to parse headers first we use sscanf to break down the header file into its components. Then we use the first string in the header to call handle_Get() or handle_Put().

```
char* get_header( Buffer *buf ) {

        ptr = buf->peek();
        input;

        for( int i=0; i <= buf->length()+4; ++i ){
                memcpy(input, ptr, 2 );
                if( strncmp( input, "\r\n", 2 ) == 0 ){
                        ptr += 2;
                        memcpy(input, ptr, 2);
                        if( strncmp( input, "\r\n", 2 ) == 0 ){
                                memcpy( input, buf->peek(), i+4 );
                                buf->consume(i+4);
                                return input;
                        }
                        ptr -= 2;
                }
                ++ptr;
        }
        return NULL;
}
```

```
Int parse_header(Buffer* buf, char* header ){
        char* requestType, char* fileName, char* contentLength;
        int bytesRead;
        sscanf( header, "%s %s HTTP/1.1\r\n%n", requestType, fileName, %bytesRead );
        sscanf( header+bytesRead, "Content-Length: %d", contentLength );
        if( strncmp( requestType, "GET" ) ){
                handle_Get( buf, fileName );
        } else {
```

```
            handle_Put( buf, fileName, contentLength );
        }
        Return 1;
}
```

2.3 handle_get

handle_get() is given a fileName and a Buffer. It will then find the file-length of the requested file and return an OK header with the length of the requested file. Then we have to open the file and loop through its contents reading from the file and writing to the socket. We can loop through this data with a while loop that exits when contentLength is < 0 and decrementing contentlength by the amount of bytes read by read() each iteration.

```
Int handle_Get( Buffer* buf, char* fileName ){
        Struct stat fileStat;
        stat( fileStat, &fileName );
        Int contentLength = stat.st_size;
        char* ok;
        sprintf( ok, “HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n”, contentLength );
        send( buf->getSocketFileDescriptor(), ok, strlen( ok ) );

        Int fd = open( fileName );
        char* buf;
        While( contentLength > 0 ){
            Int bytesRead = read( fd, buf, 1024 );
            write( buf->getSocketFileDescriptor(), buf, bytesRead );
            contentLength -= bytesRead;
        }
        close(fd);
        Return 1;
}
```

2.4 Handle_put()

handle_Put() is very similar to handle_get() except we don’t have to find the contentLength because it is provided. So we simply have to create a new file using open() and copy the data from the socket file descriptor to the new file descriptor. We can loop through this data with a while loop that exits when contentLength is < 0 and decrementing contentlength by the amount of bytes read by recv() each iteration.

```
Int handle_Put( Buffer* buf, char* fileName, contentLength ){
        char* buff;
```

```
        Int fd = open(fileName);
        char* ok = “HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\r\n”;
        send( buf->getSocketFileDescriptor(), ok, strlen( ok ) );
        while( contentLength > 0 ){
                Int bytesRead = recv( buf->getSocketFileDescriptor(), buff, 1024 );
                Write( fd, buff, bytesRead );
                contentLength -= bytesRead;
        }
        close(fd);
        Return 1;
}
```

2.5 Send Response Code

Our function will take the response number, response name, and contentLength as arguments. It will then use sprintf to generate the return value as a character array.

```
int send_response(int sockFD, int responseNum, char* responseName, int contentLength ){
        char* response;
        sprintf( response, “HTTP/1.1 %d %s\r\nContent-Length: %d\r\n\r\n”,
                responseNum, responseName, contentLength );
        write( sockFD, response, strlen(response) );
        Return 1;
}
```

2.6 Putting it together to handle a single connection

In order to handle a single connection we have to first initialize our main socket and wait for a connection. On accept(), we create a new buffer and pass the newly returned file descriptor to its constructor. Then we call handleConnection(Buffer) which first calls read_data(n) and then enters a while loop on the condition that there is still valid data in the buffer. In the loop, we will try to remove a valid header from the buffer. If none is returned but there is still data in the buffer then we keep attempting to read from the socket until a valid header is present. Once we receive a valid header we pass it to parse_header which will call handle_get or handle_put based on the type of request.

2.7 MultiThreading

In order to make the server handle multiple requests at once we'll need to create N threads using pcreate at the start of our program. We'll need two condition variables and two accompanying mutex locks. A thread_cond that the threads wait on when they're not filing a request and a dispatcher_cond for the dispatcher thread to wait on when there are no available workers threads. We'll need to synchronize two global variables, a fifo queue of socketFDs and an integer freeThreads, using the dispatcher mutex. The dispatcher will signal the thread_cond when a job occurs, decrement freeThreads and then while freeThreads equals zero it will wait on the dispatcher_cond. When a thread is woken up it gets the socketFD from the queue, it will then handle the connection. When it's done the thread will increment freeThreads and signal the dispatcher_cond before waiting on the thread_cond for it's next job.

Global Synchronized Assets

```
Pthread_cond worker_cond
Pthread_cond dispatcher_cond
Pthread_mutex worker_mutex
Pthread_mutex dispatcher_mutex
Int freeThreads = 0;
Queue<socketFD> jobs
```

Worker Thread

```
handleThread(){
while(true){
pthread_mutex_lock(worker_mutex)
        pthread_mutex_lock(dispatcher_mutex)
                freeThreads++           /* one more free worker */
        pthread_cond_signal(dispatcher_signal) /* Wake dispatcher if it's waiting */
        pthread_mutex_unlock(dispatcher_thread)
pthread_cond_wait( worker_cond, worker_mutex) /* Wait for job */
pthread_mutex_lock( dispatcher_mutex )
Buffer* buf = new Buffer( jobs.pop() ) /* Get socketFD */
pthread_mutex_unlock( dispatcher_mutex )
pthread_mutex_unlock( worker_mutex )
handleConnection( buf ) /* Handle Job */
}
}
```

Dispatcher Thread

```
dispatcherThread(){
while(True){
        Fd = accept() /* Wait for connection */
        pthread_mutex_lock(dispatcher_mutex)
                while( freeThreads == 0 ) /* Wait for available worker */
                        { pthread_cond_wait( dispatcher_cond, dispatcher_mutex ) }
                jobs.push(fd) /* Add connection to queue */
                freeThreads-- /* decrement available workers */
        pthread_mutex_unlock(dispatcher_mutex)

        pthread_cond_signal( worker_cond ) /* Wake up one worker */
}
}
```

2.8 Logging and Input Handling

In order to parse flags we'll use getopt to check for -N and -l flags. A value is set to the default number of threads to be created at the beginning of my program, however if the -N flag is used then that value will be overwritten with the requested value. In order to handle logging when the -l flag is used, I'll use two global variables and a writer_thread mutex, the logFileName and logFileOffset. Whenever a request is handled by handleGet and handlePut if the logFileName is set, they will lock the writer mutex, set a local offset variable equal to logFileOffset and then increment logFileOffset by the number of bytes they need and then unlock the writer mutex. Then when data is read from read() or recv() it will be converted to hex with sprintf and written to the logfile at the local offset using pwrite. In order to log error headers, in my sendResponseMessage() function if the response code is an error message and if the logFileName is set then I'll lock the writer mutex, reserve space, and unlock the writer mutex. I will also need a helper function that can determine the number of bytes I need to reserve given a files content length called calculate_offset(contentLength);

```
Int entryLength = calculate_offset( contentLength );
pthread_mutex_lock( writer_thread );
Int offset = logFileOffset;
logFileOffset += offset;
char* logData;
pthread_mutex_unlock( writer_thread );
…
while( contentLength < 0 ){
Data = read()
sprintf( logData, “%0x”, Data )
```

```
pwrite( logFileDescriptor, logData, offset )
Offset += strlen(logData)
}
```

2.9 Patch

In order to support persistent aliases we'll need to be able to read and write the key value tuple from and to disk and be able to query a hashtable for the alias. My program will use an alias struct for all operations relating to the alias file. It will need a hashtable, a counter for the number of aliases and a reference to the alias file. It will also need member functions to add an alias and resolve an alias. This struct will take an alias filename as a constructor argument, check if it exists and is valid, then parse the file and create a hashtable to map aliases to addresses.

2.9.1 Alias File:

Since we need to know if the alias file is valid we'll need to add a magic value to the beginning of the file and check if that value is present before using the file. The magic value I will use is "InCaffeineWeTrust/n". Key value pairs will be space separated in 128 byte chunks in the alias file.

2.9.2 HashTable:

The hashtable will be implemented within the alias struct. It will use google's cityhash as it's hashing function. It will also need a get(key), put(key, value), has(key) and remove(key) member function.

```
Data[8000];
FOR i IN 7999 { Data[i] = -1; }

get(key){ return Data[ cityhash( key ) % 8000 ]; }

put(key, value){ Data[ cityhash( key ) % 8000 ] = value; }

has(key, value){
        if( Data[ cityhash( key ) % 8000 ] != -1 ){ Return true; }
        Else{ return false; }
}

remove( key ) {
        Data[ cityhash(key)%8000 ] = -1;
}
```

2.9.3 Write:

Operations on the alias file are dependent on maintaining a hashtable that maps aliases to their byte offset in the alias file. When writing to the alias file there are two cases, either the alias already exists or it's a new alias. In the first case we query the alias in the hashtable for a byte offset, use memset to clean the existing 128 bytes, and then write to the reserved location. In the second case we write the new alias to the number of aliases*128 + len(magicNumber), increment the number of aliases and add the (alias, offset) pair to our hashtable.

```
addAlias( alias, filename ){
        hashCode = cityhash( alias );
        char* entry;
        sprintf( "%s %s\r\n", alias, filename );
        if( has( hashCode ) ) {
                /* Clear previous entry */
                char* nothing;
                memset( nothing, NULL, 128 )
                pwrite( aliasFD, nothing, 128, get( hashCode ) );

                /* Write new entry */
                pwrite( aliasFD, entry, 128, get( hashCode ) );
        } else {
                pwrite( aliasFD, entry, 128, aliasFileSize )
                add( alias, aliasFileSize )
                aliasFileSize += 128;
}
```

2.9.4 Read:

When reading from the alias file we look for an entry in the hashtable, if it exists we read at the returned byte address. If the mapping is to a valid file name we return the file name. If the mapping is not a valid filename or opening the file failed we query the mapping in our hashtable and try again. If the mapping isn't our hashtable we return 404.

```
/* Helper Function to Read One Alias */
readAlias( alias ){
        if( !has( alias ) ) { return NULL; }
        offset = get( alias );
        char* entry; char* key; char* value;
        pread( aliasFD, entry, 128, offset )
        sscanf( entry, "%s %s\n", key, value);
        Return value;
}
```

```
resolveAlias( alias ){
        while( ! isValidFileName( alias ) ){
                Alias = readAlias( alias );
                if( alias == NULL ) { return NULL; }
        }
        Return alias;
}
```

2.9.5 Synchronization:
We'll add an alias mutex that threads must request before operating on the alias file to avoid race conditions

handle_patch() will accept an alias and a filename. Lock the alias mutex, call addAlias(), release the alias mutex, and send an OK response code if successful.

```
handle_patch(alias, filename){
        lock( &alias )
        alias->addAlias(alias, filename)
        release(&alias)
        send_response( SUCCESS )
}
```

---

3. Error Checking and Handling:
In order to ensure that my code always operates correctly, I'll need to check the return value of every system call for errors. Additionally, I'll need to add checks to ensure that all ingress headers are formatted correctly and contain valid request. In order to ensure that the server doesn't continue with a request all relevant functions I created will return -1 if they were unsuccessful, I can check these return values to halt my program on failure.