Design Document: *multi-threaded HTTP server with Alias*

Nixon Duong
CruzID: niduong

1 Goals

The goal of this program is to add alias functionality to our multi-threaded HTTP server. The simple server responds to GET, PUT, PATCH requests with 27-character ASCII file names or aliases. The server stores files in a directory on the server so that it could be restarted or operate on a directory that already has files.

2 Design

The first procedure performed is to check command line arguments; processing the flags and arguments provided and initializing the hostname, port of our server, number of threads the server will operate on, mapping file and name of log file if enabled. The second procedure is to create network socket if the hostname and port are valid. The third procedure is then to start accepting and processing socket connections. The fourth procedure is to parse the HTTP request and direct the server to the subroutine that handles the specific type of request and response to the client appropriately.

2.1 Check command line arguments and flags

Before we could open a network socket on the server, we need 4 variables initialized. We need hostname/IP address, a port number, specify the number of threads we want our server to operate on and the mapping file to keep track of our alias. These variables are provided to the server through the command line. The server checks if the command line arguments are present, else an error is thrown. If the server receives a port number through the command line, the server will initialize its port number with that port number, or else the server would default to port 80. The most important flags are hostname and mapping file. A server without a mapping file or hostname specified will not be able to start. If the server receives a specified number of threads to operate on, the server will initialize it's number of threads with that number, or else the server would default to 4 threads. Note that the user of the server could also enable logging by specifying on the command line through the -l flag. If the user of the server wants logging, then a log file name must be specified on the command line as well. Once the address, port, number of threads are initialized, mapping file are valid, the server creates a thread poll with the number of threads specified or default 4 threads with each thread pointing to an instance of a state routine(**Algorithm 2**). Then the server opens the mapping file and initializes the global mapping file descriptor with the file descriptor returned from open. Lastly, we

open a network socket by making a call to **Algorithm 3**. If the network socket is valid, the server begins processing socket connections by making a call to **Algorithm 4**.

Input: Argument count: *argc*
Input: Array of arguments: *argv*
Output: 0

```
for (;;)
   option = getopt(argc, argv, "N:l:a:")
   if (option == EOF) break
   switch (option)
      case 'N':
         nThreads = atoi(optarg)
         break
      case 'l':
         logFile = optarg
         break
      case 'a':
         mappingFile = optarg
         break
if (argc == optind)
   stderr("Error: 0 commandline argument given, 1 required\n")
   exit(1)
else if ((argc - optind) > 2)
   stderr("Error: %d commandline argument given, 1 required\n", argc - optind)
   exit(1)
else
   hostname = argv[optind]
   if ((argc - 1) > optind)
      port = argv[optind + 1]
   else
      port = (char*) "80"
if (mappingFile == nullptr)
   fprintf(stderr, "Error: no mapping file specified\n")
   exit(1)
if (nThreads == 0)
   nThread = 4
if hostname and port are valid
   Pthread_t threadPool[nThreads]
   for thread in threadPool
      pthread_create(thread, NULL, threadFunction, NULL)
   main_socket = createNetworkSocket(hostname, port, addrs, &hints)
   if main_socket is valid
      processSocketConnections(main_socket)
```

Algorithm 1: Check command line arguments

Each worker thread is an instance of Algorithm 2. This requires each thread to wait for the dispatcher to produce a connection socket file descriptor. When the dispatcher produces a connection socket file descriptor, the thread consumes a socket file descriptor. While consuming the socket file descriptor, the thread enforces a mutually exclusive zone so that the producer doesn't tamper with what the thread is about to read. Afterward, the thread signals the dispatcher that there is space available. Once the thread consumes the file descriptor, it passes the sockfdItem to **Algorithm 5**. Note, while there are no connections made, the count is 0 so the threads go to sleep and wait for data to be available.

Input: *void **
Output: *void **
Global Variables: *lock, dataAvailable, sharedBuffer, out, count, spaceAvailable*

```
while (1)
   pthread_mutex_lock(&lock)
   while (count == 0)
      pthread_cond_wait(&dataAvailable, &lock)
   sockfdItem = sharedBuffer[out]
   count--
   out = (out + 1) % BUFFER_SIZE
   pthread_cond_signal(&spaceAvailable)
   pthread_mutex_unlock(&lock)
   recv(sockfdItem, buffer, sizeof(buffer), 0)
   parseRequest(buffer, sockfdItem)
   write to log-file
```

Algorithm 2: Thread start routine

2.2 Create a network socket

To create a network socket requires the correct address information. We request the correct address information by making a call to getaddrinfo(). Afterward, we create a network socket by calling socket() and pass the call the required address information. The next step is to then enable the socket by making a call to the setsockopt(). We then need to bind a name to the socket by making a call to bind(). Lastly, we begin listening for connections by calling listen().

Input: Address: *hostname*
Input: Port number: *port*
Input: Address Information: *addrs*
Input: Hints: *hints*
Output: Network socket file descriptor: *main_socket*

```
getaddrinfo(hostname, port, &hints, &addrs)
```

```
main_socket = socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol)
enable = 1
setsockopt(main_socket, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(enable))
bind(main_socket, addrs->ai_addr, addrs->ai_addrlen)
listen(main_socket, BACKLOG_SIZE)
return main_socket
```

Algorithm 3: Accept connection on socket

2.3 Accept connection on socket

Algorithm 4 aims to always be on the lookout for upcoming connections. If there is a connection waiting to be accepted, the server accepts the connection, creates a mutually exclusive zone, places the socket file descriptor of the accepted connection into the shared buffer, signals the threads in the thread pool that a connect has been accepted and then stops enforcing a mutually exclusive zone. When the number of connections accepted exceeds the size of our shared buffer, our server must wait for its worker threads to complete work and make space. Once there is space again, the server could continue dispatching.

Input: Main socket file descriptor: *main_socket*
Output: void
Global Variables: *sockfdItem, count, spaceAvailable, lock, dataAvailable, sharedBuffer*

```
sockfdItem = -1
while (1)
   while (count == BUFFER_SIZE)
      pthread_cond_wait(&spaceAvailable, &lock)
   sockfdItem = accept(main_socket, NULL, NULL)
   if (sockfdItem >= 0)
      pthread_mutex_lock(&lock)
      sharedBuffer[in] = sockfdItem
      count++
      in = (in + 1) % BUFFER_SIZE
      pthread_cond_signal(&dataAvailable)
      pthread_mutex_unlock(&lock)
   sockfdItem = -1
```

Algorithm 4: Accept connection on socket

2.4 Parse HTTP request

Now that we have our request data, we need to make sense of it. We do this by taking a look at the header. We this by scanning the first line of the header for the type of request. Once we know what type of request the client is looking for, we direct the server to the appropriate subroutine to handle the request. We do this by making a call to **Algorithm 6** or **Algorithm 7** or **Algorithm 8**.

```
Input: buffer: buffer
Input: current socket file descriptor: sockfd
Output: void

scanf buffer for type, filename and http
if filename[0] is '/'
   send response to client with 403 error
if type is 'GET'
   handleGetRequest(buffer, sockfd)
if type is 'PUT'
   handlePUTRequest(buffer, sockfd)
if type is 'PATCH'
   handlePatchRequest(buffer, sockfd)
if type not 'GET' or 'PUT' or 'PATCH'
   Send response to client with 500 error
```

Algorithm 5: Parse HTTP request

2.5 Handle GET request

To handle a GET request requires the server to check if the filename of the request is valid. We do so by checking if the filename given was an alias. This is done by passing the filename to **Algorithm 10**. If there is a valid filename or alias, the server returns the data from the requested filename. We first scan the GET request for a filename. Then we check if the resource name is valid by making a call to **Algorithm 11**. If the resource name is valid we first find the size of the data by running through the data one time. We do this so that we could have a content-length for the GET response header. Afterward, we read a byte from the resource and write that byte to the response repeated until there is no more data to be read. Lastly, we close the socket.

```
Input: Request buffer: buffer
Input: Socket file descriptor: sockfd
Output: void

scanf buffer for filename and http
rawFilename = followAliasToObject(filename, "GET")
if (isValidResourceName(rawFilename))
   fileDescriptor = open(filename, O_RDONLY)
   if (fileDescriptor is -1)
      Send response to client with 404 error
   else
      bytesRead = 1
      while (bytesRead)
         bytesRead = read(fileDescriptor, tempBuffer, 1)
         contentLength += bytesRead
         if bytesRead is -1 break
      close fileDescriptor
      Send response to client with 200 status and content length
```

```
    counter = 0
    while (counter < contentLength)
      bytesRead = read(fileDescriptor, readBuffer, sizeof(readBuffer))
      if bytesRead is -1
        break
      else
        send client byte read
      counter += 1
    close(fileDescriptor)
else
  send response to client with 400 error
```

Algorithm 6: Handles GET Request

2.6 Handle PUT request

To handle a PUT request requires the server to check if the filename of the request is valid. We do so by checking if the filename given was an alias. This is done by passing the filename to **Algorithm 10**. If so, the server puts the data provided by the client to a valid resource file on the server. We first scan the PUT request for a filename. Then we check if the resource name is valid by making a call to **Algorithm 11**. We then need to parse the request to see how much bytes of data is the client intending to put into the resource file. Once the server knows the resource file is valid, the server attempts to open the file. If the file exists, then override the file with the number of bytes specified by the client with the data provided. Else, create a file on the server with the resource filename and write to the newly created file with the number of bytes specified by the client with the data provided. Lastly, respond to the client with the appropriate status code and close connection.

Input: Request buffer: *buffer*
Input: Socket file descriptor: *sockfd*
Output: void

```
scanf buffer for filename and http
rawFilename = followAliasToObject(filename, “PUT”)
if (isValidResourceName(rawFilename))
  if file not in server
    status = 201
  fileDescriptor = open(rawFilename, O_WRONLY | O_CREAT | O_TRUNC)
  if fileDescriptor not -1
    if strlen(requestData) == contentLength
      write to file pointed by fileDescriptor with strlen(requestData) bytes of requestData
    if strlen(requestData) < contentLength
      write to file pointed by fileDescriptor with strlen(requestData) bytes of requestData
      contentLength -= strlen(requestData)
      count = 0
      bytesRead = 0
```

```
        while (counter < contentLength)
            bytesRead = read(fileDescriptor, readBuffer, sizeof(readBuffer))
            if (bytesRead == -1)
                break
            else
                send(sockfd, readBuffer, bytesRead, 0)
            counter++
    close(fileDescriptor)
    if (statusMSG == 0)
        responseVal = sprintf(response, "%s %d OK\r\n", http, status)
    if (statusMSG == 1)
        responseVal = sprintf(response, "%s %d Created\r\n", http, status)
    send(sockfd, response, responseVal, 0)
else
   send response to client with 400 error
close(sockfd)
```

Algorithm 7: Handles PUT Request

2.7 Handle PATCH request

To handle a PATCH request requires the server does two things. The first thing the server does is parse the request body for existing_name and new_name. Afterward, the function passes the two values to **Algorithm 10. Algorithm 10** attempts to create a new alias by checking to see if the names meet the requirements and if the existing_name actually exist. The server checks to see if a name is existing by passing the existing_name to **Algorithm 10.** If the existing_name actually exist, then the server hashes the new_name and writes the new alias into the mapping file with an offset of the hash value. **Algorithm 9** and **Algorithm 10** are similar in the sense that 9 produces the aliases while 10 consumes.

```
Input: Request buffer: buffer
Input: Socket file descriptor: sockfd
Output: void

read(sockfd, buffer, BUFFER_SIZE)
sscanf(buffer, "%*s %s %s\r\n", existing_name, new_name)
if (createNewAlias(existing_name, new_name))
    send response that adding alias was a success
else
    send a response that adding the alias was not a success
close(sockfd)
```

Algorithm 8: Handles Patch Request

Input: Existing name: *existing_name*
Input: New name: *new_name*
Output: Creation validity: *status*

```
obj = followAliasToObject(existing_name, (char*)“PATCH”)
status = true
if (obj == nullptr)
   status = false
else
   if (sizeof(existing_name) + sizeof(new_name) > 128)
      status = false
   else
      hashVal = hashFunction(new_name) % 8000
      sprintf(entry, “%s %s”, new_name, existing_name)
      pwrite(mappingFd, entry, entrySize, hashVal)
return status
```

Algorithm 9: Attempts to create a new alias

Input: Alias to follow: *alias*
Input: Type of HTTP request: *type*
Output: actual object: *obj*

```
while (fileTest == -1)
   // Iterate through chain of aliases
   fileTest = open(alias, O_RDONLY)
   obj = nullptr
   if (fileTest == -1 and type == “PUT)
      hashVal = hashFunction(alias) % 8000
      sscanf(entry, “%*s %s”, value)
      obj = value
   else
      obj = alias
return obj
```

Algorithm 10: Server attempts to follow aliases until it reaches an actual object

2.8 Check if resource name is valid

To promote reusability and modularity, we factor out the resource validity check from **Algorithm 6** and **Algorithm 7**.

Input: Resource name: *resourceName*
Output: Resource name validity: *valid*

```
if length(resourceName) not 27
   return false
for character in resourceName
   asciiVal = static_cast<ssize_t>(character)
   if asciiVal not between 48 and 57 or 97 and 122 or 65 and 90 or 95 or 45
      valid = true
   else
      return false
return valid
```

Algorithm 11: Checks if resource name is valid

Citations:

*** This Design Document was based off Ethan Miller's Design Document example ***