1. **Introduction**
   1. **Objectives**

      The program will function as a multi-threaded http server that implements redundancy. It will respond to multiple requests simultaneously. Each file will have three copies when the program is run with the redundancy flag. The program will run on Ubuntu 18.04 installation.

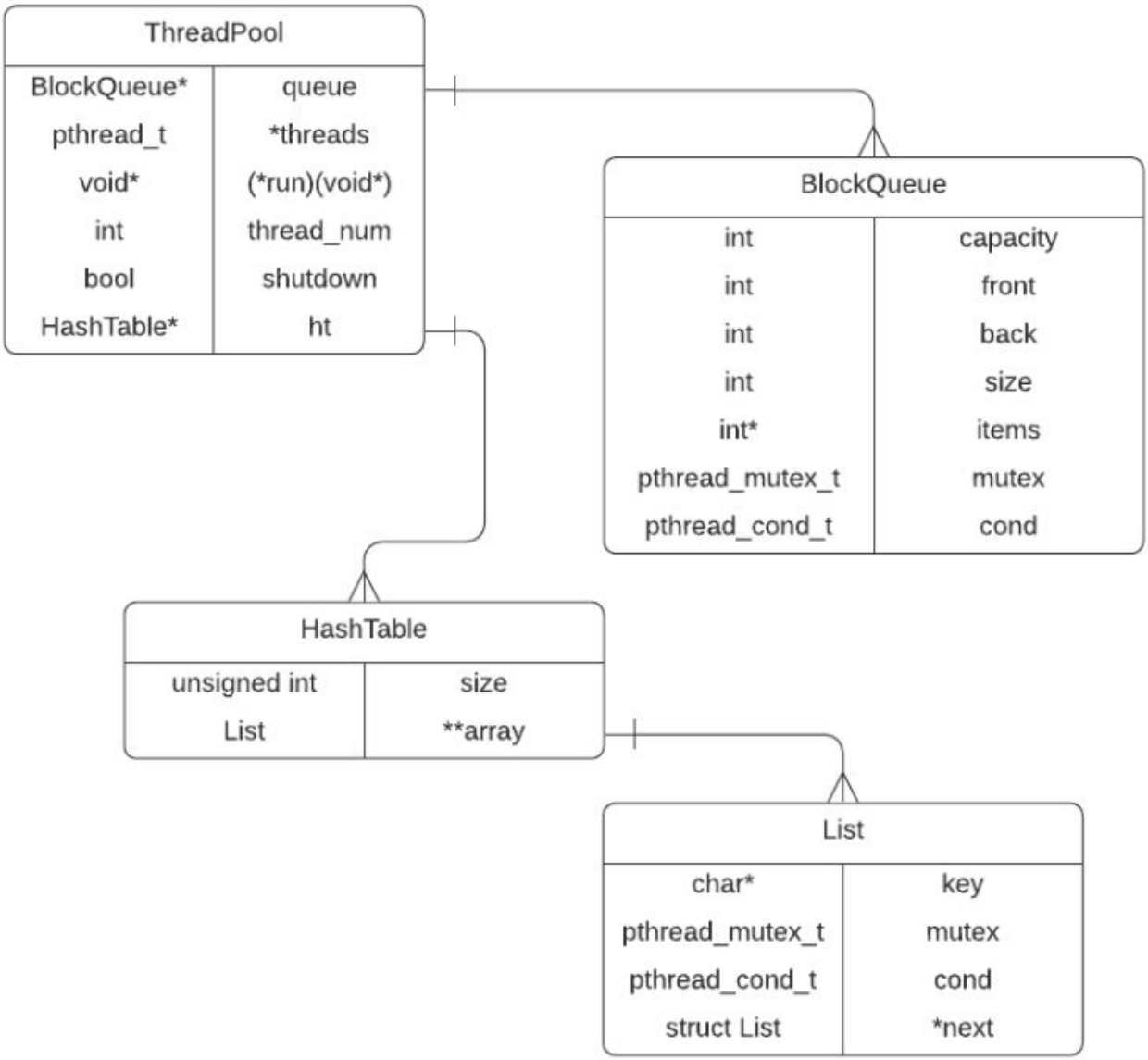
   2. **Constraints**

      All source files must have `.cpp` suffix and be compiled with no warnings using the flags: `-std=gnu++11 -Wall -Wextra -Wpedantic -Wshadow`. Only standard networking system calls may be used, and no FILE * calls can be used to handle reads/writes. Only POSIX threads libraries may be used.

2. **Data Design**
   1. **Struct**

      HttpObject: stores information from incoming requests. Contains the following attributes:

      - `char method[5]`: access method of the request, spec only mentions PUT and GET, so length of 5 should be more than enough.
      - `char filename[11]`: the name of the file, max length is 10 and it seems like names should be exactly 10 characters. Only alphanumerics may be used.
      - `char httpversion[9]`: the version of http being used, default is HTTP/1.1.
      - `char path[17]`: the path that includes the folder name and file name,
      - `ssize_t request_header_length`: length of the request header, helps to parse the http request.
      - `ssize_t content_length`: length of the file.
      - `int status_code`: http status code for the response.
      - `uint8_t buffer[]`: the buffer used to transfer files, default to 16KiB.
      - `ssize_t total_len`: size of the data read, helps to parse data being received.
      - `int exists`: check if “Content-Length” is present in the header.
      - `int client_status`: boolean that checks client’s connectivity.

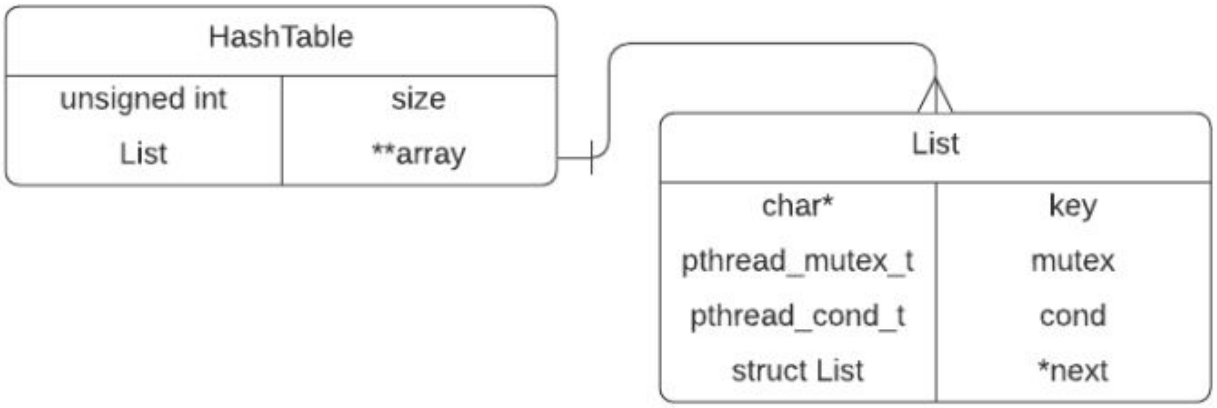



BlockQueue: FIFO queue that stores the client file descriptor passed by the dispatcher.

- `int capacity`: the maximum number of clients that can be stored in the queue
- `int front`: index of the front element of the queue
- `int back`: index of the back element of the queue
- `int size`: number of clients currently in queue
- `int* items`: an int array that stores the client file descriptors.
- `pthread_mutex_t mutex`: mutex for when modifying the queue
- `pthread_con_t cond`: conditional variable for the queue

ThreadPool: struct that managers worker threads.

- `BlockQueue* queue`: pointer to the block queue that the worker threads are working off of.
- `HashTable* ht`: pointer to the hashtable initialized in `main()`. It is included in ThreadPool so it can be accessed in the `(*run)` sequence.
- `pthread_t *threads`: array of threads with size `thread_num`.
- `void* (*run)(void*)`: pointer to the code sequence that the threads are executing.
- `int thread_num`: number of threads
- `bool shutdown`: signal to shutdown the thread pool.




List: an element in the hashtable that stores the filename, mutex, conditional variable, and a pointer to the next element.

- `char* key`: filename being stored.
- `pthread_mutex_t mutex`: file lock for the file, it will be locked when the file is being accessed.
- `pthread_cond_t cond`: conditional variable for the lock.
- `struct List *next`: pointer to the next element in the array.

HashTable: a table that is used to store files based on the hash value of the filenames.

- `unsigned int size`: size of the hashtable array. The `**array` will have `size` number of slots for linked lists.
- `List **array`: an array where each element is a linked list. Each file will be hashed according to its filename, then it will fall into one of the linked lists.

### 2.2 Functions // modify for multi-threading

- `void read_http_response( int client_socket_fd, HttpObject *message )`: reads HTTP request from client. Parse the http header, then extract information and store it in the HttpObject message.

  creates a buffer to read the http header.

recv() from client.
checks for client_status.
parse for http method, filename, http version.
check for valid method, filename, and http version, sets status code to 400 if any of the three is invalid and returns.
find Content-Length and mark its existence.
*100 continue.*

- `void process_request( int client_socket_fd, HttpObject *message, HashTable* ht )`: process the request. Checks for client status, checks status code, then checks for the request type. PUT will create and store the file being uploaded to the server. GET will search for the requested file on the server. 404 Status code will be created if the file does not exist, 403 if the client has no access right to the file. 500 Internal Server Error occurs when read/write or access file fails.

```
if client_status is 0:
        do nothing and skips
if status_code >= 400:
        error occurred, skip to next function
if method is GET:
        if redundancy:
                create paths to folders: copy1, copy2, copy3
                attach filename to paths
                compare file contents:
                        if compare(path1, path2):
                                message->path = path1
                        else if compare(path1, path3):
                                message->path = path1
                        else if compare(path2, path3):
                                message->path = path2
                        else:   // all three files are different
                                message->status_code = 500
                                return
                check for existing files (404 Not Found)
                check file stats:
                        if stat < 0: // stat() error
                                message->status_code = 500
                                return
                        else:
                                if errno == EACCES:  (403 Forbidden)
                                else:
                                        200 OK
                                        content_length = filesize;
        else: // not redundant
                check for existing files (404 Not Found)
                check file stats:
                        if stat < 0: // stat() error
                                message->status_code = 500
```

```
                    return
                else:
                    if errno == EACCES:  (403 Forbidden)
                    else:
                        200 OK
                        content_length = filesize;

if method is PUT:
    if -r flag is true:
        create paths to folders: copy1, copy2, copy3
        attach filename to paths

        acquire file lock from ht
        if lock does not exist in ht, then use global lock
        lock the mutex

        open file in all the paths, with
        open(...,O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU): flags:
        write only, create it if file not exists, or overwrite if file exists
        If failed: (500 Internal Server Error), skips to next function

        if( message->exists ): // Content-Length is provided in the http
        header
            while( i < content-length)
                check recv() value from client
                if recv() <= 0:
                    set client_status to 0
                    skips
                write to file, to all three paths
                if write() failed: 500 Internal Server Error

                i += content written
        else:
            while( i = recv() > 0 ): // keeps receiving until client drops
                write to file, all three of them
                if data written to each files are not equal:
                    500 Internal Server Error

            /* doesn't need to check client status because writing to
            file only stops when client terminates the connection.

        add new file lock to hashtable if it doesn not already exist, then
        unlock either the existing lock, or the global lock

    else:
        same procedure as above, except only one copy of the file is
        created and it's in the same directory as the server, not in
        copy1...
```

else: // not a valid request
status_code = 400

- `void construct_http_response( HttpObject *message )`: constructs a http response to send to the client. Stores the constructed message in *message->buffer*.

  if client_status is 0:
  do nothing and skips
  allocate string buffer *msg*
  set *msg* and *message->buffer* to 0
  if method is PUT or status_cose >= 400:
  message->content_length = 0
  sprintf() response header to msg
  memcpy( message->buffer, msg, strlen( msg ))
  message->request_header_length = strlen( msg )

- `void send_message( int client_socket_fd, HttpObject *message, HashTable* ht )`: sends the http response to the client. Sends the requested file if successfully processed a GET request.

  if client_status is 0:
  do nothing and skips
  send http response header to client:
  send( client_sockd, message->buffer,
  message->request_header_length, 0 )

  if method is GET and status_code < 400:
  if redundancy:
  acquire file lock from `ht`
  lock mutex

  open file with `open(message->path, )_RDONLY)`, flag: read only

  i = 0
  while( `i < message->content_length` ):
  read file content to buffer
  send buffer to client
  i += size written

  unlock mutex
  else:
  acquire file lock from `ht`
  lock mutex

  open file with `open(message->filename, O_RDONLY)`, flag: read only

i = 0
while( bytes = read() > 0 ):
read file content to buffer
send buffer to client
i += size written

/* we don't check for failed sends because the http header is already sent, there is no way to retract and change the status code
*/
unlock mutex

- `void* run(void* pool)`: code sequence for worker threads to operate on. pool is the ThreadPool that maintains the threads.

ThreadPool* tp = (ThreadPool*) pool
while( `!tp->shutdown` ): `// repeat the process of accepting clients when the shutdown signal is inactive. According to spec, it should never shutdown`
struct httpObject message
client_socket_fd = front element of `tp->queue`
message.client_status = 1

while( `message.client_status > 0` ):
read_http_response()
process_request()
construct_http_response()
send_message()
close( `client_socket_fd` )

### 2.2.1 Helpers

- `char *message_for_code( int status_code )`: returns corresponding message for the given http status code. Codes implemented: 200, 201, 400, 403, 404, 500. Default is 100 Continue.
  switch( status_code )
- `unsigned long getaddr( char* name )`: returns the numerical representation of the address identified by *name*. Code from example code from section.
- `ssize_t next_line_index( char* buffer, ssize_t start, ssize_t total_len)`: finds the line break in the buffer, and returns the index of the starting position of the next line.
  for i in range( start, total_len - 1 ):
  if character in buffer[ i ] == '\r' and the next char is '\n'
  break
  i += 2 // need to move 2 positions
  return i

- `int is_al_num( char* name )`: returns 1 if all characters in name are alphanumeric, else 0.

```
int r = 1 // boolean if the string only contains alnum, default to 1,
for char in name:
    if char is not alnum:
        r = 0
return r
```

- `int is_num( char* str )`: checks if `str` consists of only numbers. It's used in `main()` to differentiate between hostname and port number. Returns 1 if only numbers exists in `str`, else 0.

```
for i in range( len(str) ):
    if !isdigit( str[i] ):
        return 0
return 1
```

- `int compare( char* path1, char* path2 )`: compare the contents of two files. Return 1 if they are the same, else 0.

```
if access( path1, F_OK) != access( path2, F_OK)
    return 0

buffer1, buffer2
while (b1 = read( path1, buffer1, BUF_SIZE )) &&  (b2 = read( path1, buffer1, BUF_SIZE )):
    if b1 != b2:
        return 0
    if buffer1 != buffer2:
        return 0

return 1
```

### 2.3 Constants

- BUFFER_SIZE: 16 KiB. The http header being received will be no longer than 16 KiB, so it might as well be the buffer size.

### 2.4 Global Variables

- `int thread_num`: number of threads for the server to use, default to 4.
- `int redundancy`: flag for redundancy, default to 0.
- `pthread_mutex_t global_mutex`: file lock for new files. It will be locked when a PUT request sends a new file to the server.

## 3. Structure

```
Check for the number of arguments, exit if less than 2.
Set option variable for getopt().
While (option = getopt() != -1):
    switch( option ):
        Case 'N':
```

If option N is < 1, return EXIT_FAILURE

Case ‘r’:

Set `redundancy` flag to 1

If remaining argument is more than 2 or less than 1: `// only hostname and optional port number should be left`

return EXIT_FAILURE

Configure socket:

int server_sockd = socket(AF_INET, SOCK_STREAM, 0)

error if server_sockd < 0

for elem in optind:

if is_num( elem ):

port = elem

server_addr.sin_port = htons(port)

else:

server_addr.sin_addr.s_addr = getaddr( elem )

// avoid: 'Bind: Address Already in Use' error

```
    int enable = 1;
    int ret = setsockopt(server_sockd, SOL_SOCKET,
SO_REUSEADDR, &enable, sizeof(enable));
```

ret = bind( server_sockd, (struct sockaddr *) &server_addr, adddrlen ) : bind server address to socket address

ret = listen( server_sockd, 5 ) : listen for connection

Create file lock for existing files:

HashTable *ht

ht = ht_create(512) `// create HashTable with size 512`

if redundancy == 1:

dirs = {“copy1”, “copy2”, “copy3”}

for dr in dirs:

opendir(dr)

for file in dr:

if `len(file.name)` == 10:

`ht_put( ht, file.name) // tries to add file lock to hashtable, does nothing if it already exists`

else:

```
for file in cwd:
    if len( file.name ) == 10:
        ht_put( ht, file.name )
```

Initialize ThreadPool:

ThreadPool pool

```
thread_pool_init(&pool, ht,  thread_num, 1024, run)
/*
 initialize pool with thread_num number of threads,
maximum queue size of 1024, and process sequence run
for the worker threads.
*/
```

thread_pool_start(&pool)

// structs and data needed for comms
struct soddaddr client_addr
Socklen_t client_addrlen

while( true ): // start server, keeps it running

```
// connects to client socket
```

client_sockd = accept(server_sockd, &client_addr, &client_addrlen)

check for successful accept, warn and skips if unsuccessful

add `client_sockd` to ThreadPool

## 4. Tests

**Test Cases used:**

1. GET file that doesn't exist
2. GET file with invalid name
3. PUT file with invalid name
4. PUT file with incomplete body (closed early); (empty file will be created)
5. Send 2-4 requests of the same resource at the same time using 1-4 worker threads.
6. Send 2-4 requests of different resources at the same time using 1-4 worker threads.
7. Send multiple requests in the same connection
8. PUT large files
9. GET large files

did not test cases where folders (ie. copy1) might be missing, they are always provided according to spec.

**Question:**

- If we do not hold a global lock when creating a new file, what kind of synchronization problem can occur? Describe a scenario of the problem.
- As you increase the number of threads, do you keep getting better performance/scalability indefinitely? Explain why or why not.

**Answer:**

1. If we don't hold a global lock when creating a new file, then there is nothing to stop other clients from sending PUT requests with the same filename. So the first thread might not be finished writing when other threads finished creating a new file, with the same name as the first thread but different content, then the first thread might overwrite the file created by another thread. This incoherence is a big problem when multiple clients are trying to update the same source.
   Also, the threads might access each other's memory when writing, corrupting the file with mixed data from one thread to another, rendering the file unreadable.

2. No, because the hardware only has a limited amount of resources. As the number of threads increases, they also send more I/O requests to the system. When creating a new file, the threads are limited by the system's read/write speed, and they must wait for the global mutex. Same goes for reading a file, other threads will have to wait until the one that is currently reading and sending this particular file to finish, before they can have their turn at the mutex.