HTTP Server Assignment

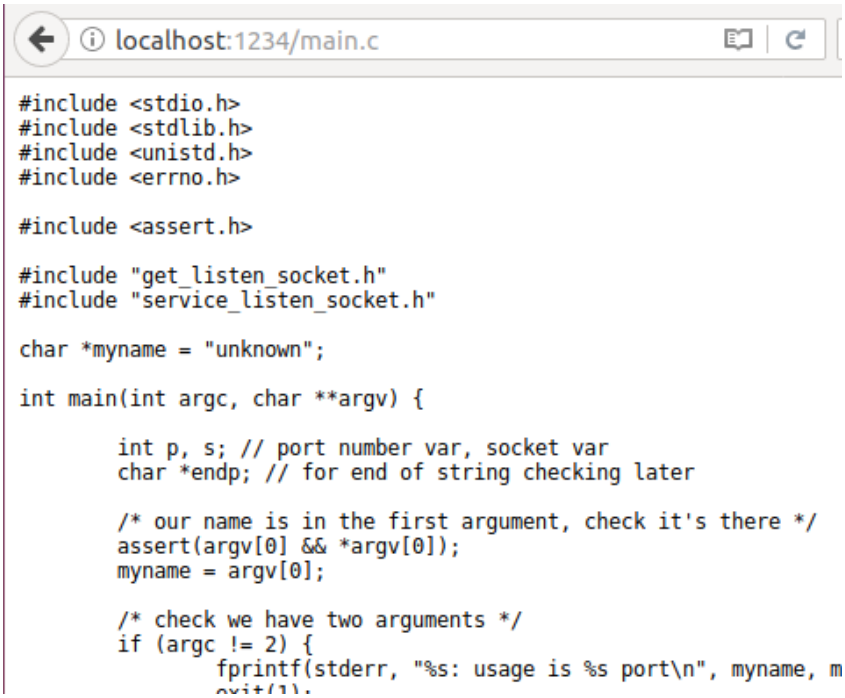
Documentation 1522391

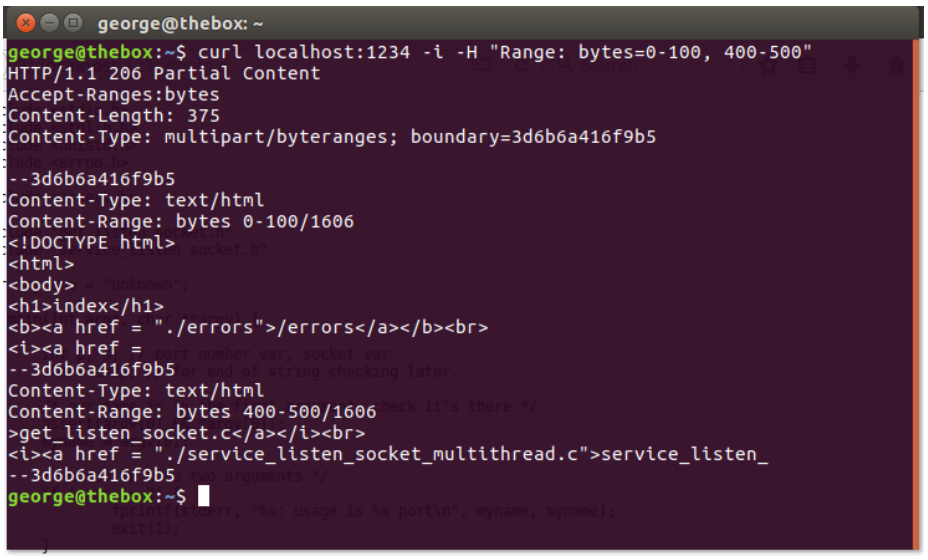
Running the Server

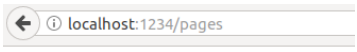
A makefile is provided which uses Clang with flags `-Wall -Werror -std=gnu99 -D_GNU_SOURCE -pthread -g` to compile to executable `server`, which can be run as `./server <port>`

Server Features

- Can handle GET and HEAD requests
- Can serve files and directories from the system
- Can serve text and image files
- A null URI displays an index.html file if one is present or defaults to a directory view otherwise
- Supports byte ranges

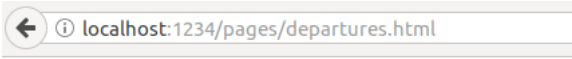






index/pages

Up
home.html
departures.html
trains.html



Departures

1203 Plymouth Plat.3
1205 Longbridge Plat.12B
1206 Nottingham Plat.7B
Back

Source Code

The HTTP server comprises of five source files and associated header files, based off code provided by Ian Batten. They are listed below in order of execution in a typical run of the server program:

- `main.c`
- `get_listen_socket.c`
- `service_listen_socket_multithread.c`
- `make_printable_address.c`
- `service_client_socket.c`

`main.c`

This file contains the entry-point of the server executable. It performs checks on the arguments, ensuring there is only one (in addition to the program name), the port number. Checks on the argument are performed: if it is actually a number, and if it is between 1024-65535, the range of ports accessible to a non-root user. `get_listen_socket()` is then called to obtain a listening socket, which is used in a subsequent `service_listen_socket()` call. If any of these functions or checks fail, the program exits with code 1.

`get_listen_socket.c`

This file contains the function **`int`** `get_listen_socket(`**`const int`** `port)`, which takes in a port number and returns the **int** representing the socket bound to the server's address. It creates a `sockaddr_in6` struct which contains information about the address, which is then used when `bind()`ing the socket to said address, and then set to `listen()` to messages.

`service_listen_socket_multithread.c`

This file contains the function **`int`** `service_listen_socket(`**`const int`** `s)`, which takes an int representing a listening socket and initialises a new thread for each client that connects (allowing concurrency). It contains a loop that continually listens for connections using accept(), then creating a new thread using the function **`static void`** `*client_thread(`**`void`** `*data)`. This function takes a `thread_control_block` struct (which contains client information such as their address and its size), which is used to make a printable address and provide information to `service_client_socket()`. This file also contains basic signal handling, so if SIGINT or SIGTERM are received by the server, it can close gracefully and free memory used.

`make_printable_address.c`

This file contains the function **`char`** `*make_printable_address(`**`const struct`** `sockaddr_in6 *`**`const`** `addr, `**`const`** `socklen_t addr_len, `**`char *const`** `buffer, `**`const size_t`** `buffer_size)`, which takes in an address struct, the length of the address, a buffer and its length, and returns a char buffer containing a printable version of the address contained in the address struct. Conversion is handled by `inet_ntop()`.

`service_client_socket.c`

This file contains the function **`int`** `service_client_socket(`**`const int`** `s, `**`contest char *const`** `tag)`, which takes in a socket and an address tag. This function performs the main brunt of the processing of HTTP requests and their responses.

A loop is created, which continually read()s bytes from the socket, ending only when the stream ends. In each iteration of the loop, the bytes are stored in a buffer which is then parsed with strtok() for the various elements of an HTTP request: most importantly the request URI method, the request URI itself, and the HTTP version. The subsequent headers are also parsed, which opens up the code for future expansion to handle these – only byte ranges are currently supported. Various checks are performed: if any of the three main elements are missing or malformed, a 400 Bad Request flag is set; if the method is not GET or HEAD, a 501 Not Implemented flag is set; and if the HTTP version is not 1.0 or 1.1, a 505 Unsupported Protocol flag is set.

The flags are then processed: if any of the errors are present the appropriate error page is set as the file to be read, otherwise the server searches for the request URI. If a null URI is given, an index.html file is first searched

for, before rolling back to just displaying the directory of the root folder if that is not present. If the URI is found, the response code is set as 200 OK, otherwise the response code is set as 404 Not Found.

Next steps are made to prepare for file loading. A flag is set to indicate whether the URI points to a directory or file (using the functions provided in `sys/stat.h`, such as `S_ISREG(sb.st_mode)`. File extensions are then parsed to ensure that binary files are read in `rb` mode so no data is lost. The relevant MIME type is also stored for use in the HTTP response.

If the URI points to a file, it is opened with `fopen()`, and `fseek()` is used to determine its length. A content buffer is allocated and the file read into it using `fread()`. If a directory is instead specified, it is navigated to using `opendir()` and `chdir()`. An HTML file that lists all files in that folder is then generated using readdir() to read file information. Directories are displayed in bold and files in italics to distinguish them, and they can be clicked on to explore the filesystem more. Every directory except the root directory also contains an ‘up’ button to move up the file tree.

If byte ranges were not specified, all the information gathered is then collated in a response header containing the content length and type, which are then written to the client followed by the content (if a GET request was made). If byte ranges are specified, the ranges are parsed using `strtok()` and stored in a two dimensional int array. If any range is below zero or above the content length, a 416 Request Range Not Satisfiable error is served instead. Each range requested is then sliced from the original content using `strncpy()` and written to the client as specified in HTTP/1.1.

Any error encountered throughout the loop (malloc, read or write errors over sockets) will cause the function to return -1 and exit.