S–38.3610 Network Programming
Assignment 2: HTTP Server
Filip Nybäck (79808K)

Assignment 2: HTTP Server

Overview

In this assignment a basic HTTP server is implemented. The implementation separates the functionality into a few C files. The following table shows the main functionality of each C file:

http-server.c	– converting a socket address to a C string – checking command line options – generating a local filename from the root path and request path – assembling and sending a reply header – sending an error reply – serving GET and PUT requests – general request handling and dispatching – starting threads to serve requests – creating the server (listening) socket – daemonizing the server – the main function
threads.c	– initializating and shutting down the thread handling – allocating and releasing an entry in the thread table
parse.c	– reading a line of the HTTP header – parsing a status line – parsing a request line – parsing a header line to a name and value pair – parsing a HTTP request header – freeing a parsed header – decoding and validating a request path
pstring.c	– initialization, clearing and freeing PSTRINGs (see below) – converting PSTRINGs to C strings
string.c	– extensions to the standard C library string functions
transfer.c	– copying binary data from one stream to another, one buffer at a time
log.c	– writing messages to the stdout, stderr and log file
discovery.c	– announcing to the course specific server discovery system that the server is starting or shutting down
connection.c	– making a TCP connection – getting the hostname string of the server itself
directory.c	– listing a directory to a buffer

Key functions and operation logic

The *main* function in http-server.c goes through a number of steps to set up the server, then calls *server_loop,* where the server spends most of its execution time and finally, when the *server_loop* function has finished, shuts down the server.

The setup phase

During the first step of the setup phase, the server program goes through the command line arguments, with which the server was started. These arguments, or options, are documented in the ‘User instructions’ section. The *check_options* function sets the runtime parameters (struct CONFIG) based on the options given on the command line. The second step during the setup phase is to open the log file. This implementation of the HTTP server uses its own private log file (*http-server-log.txt*).

The third step is to daemonize the server, unless the *--foreground* options has been given. The daemonization means that the server is completely detached from the terminal, from which the server was started. In more detail, the server *fork*s, the parent process exits and the child continues to run, but has now a new process id. The child creates a new session and becomes the new session leader. The process forks again, the parent exits, the child continues and has a new process id, but is not a session leader. The daemonization continues by reopening *stdin, stdout* and *stderr* as */dev/null*. All other file descriptors, except for the file descriptor of the log file, are closed. The daemonization does not change the current directory of the server, because the log file is still there and the default directory of the file storage for the GET and PUT requests is also inside the current directory. If the server is not daemonized (by starting it with the *--foreground* option), log messages are output to the terminal and the server can be shut down using Ctrl–C (the SIGINT signal).

The fourth step is indeed to setup signal handling. The server reacts to three signals: SIGINT, SIGTERM and SIGPIPE. SIGINT is triggered in foreground mode by pressing Ctrl–C in the controlling terminal. SIGTERM is triggered by the default signal of the kill command. SIGPIPE is triggered when a connection has been closed from the remote side. When the server has been daemonized, its new process id is written to the log. The log can most easily be followed using the *tail* command: *tail -f http-server-log.txt*. Another way to find out the process id of the server is to use the *ps* command: *ps x.* When the process id of the server is known (e.g. 12345), the server can be shutdown by using the *kill* command: *kill 12345.*

The fifth step in the initialization phase is to set up thread handling. This implementation of the HTTP server uses Posix threads to implement concurrency. A limited number of concurrent threads can execute at once. This is a deliberate choice in order to prevent a DOS attack to allocate a very large number of threads and other resources. More details about the thread handling are found in the 'Thread handling' section below.

The sixth step is to create a server (listening) socket. If the server is started with the *--ipv4* option, the server will only listen to connections over IPv4. Similarly, if the server is started with the *--ipv6* option, the server will only listen to connections over IPv6. If none or both options are given, the server will listen to connections over both protocols. The server does not use the *getaddrinfo* to get a listening socket, but uses the more old-fashioned way of filling a *sockaddr* structure. The main reason for doing it this way is that the server can listen to any address (INADDR_ANY and in6addr_any) using only one server (listening) socket.

The seventh and last step before the server is ready to receive connections is to announce to the discovery service (provided by the course) that the server has started. The server acts temporarily as a specialized HTTP client by PUTting a file named servers-*username*.txt (in this case servers-fnyback.txt) to the discovery service (nwprog1.netlab.hut.fi:3000). The content (body) of the file is a line with the follwing syntax: *hostname:port.*

The shutdown phase

The shutdown phase consists of fewer steps than the setup phase. The first step is to announce to the discovery service that the server will shutdown. This works in a very similar manner as in the setup phase; the difference is that the body of the PUT request is empty.

The second step is to close the server (listening) socket. After this, threads can be shutdown. However, if threads are running, the main thread has to wait for the requests threads to finish, so that connections are not abruptly broken. More details about the shutdown process of thread handling are covered in the 'Thread handling' section below. As a fourth and final step, the log is closed.

Thread handling

The thread handling is based on a thread table. Each entry in the table has a flag indicating if the thread is reserved (in use), a thread id, which is assigned when the thread is started, and arguments for the thread. The thread table is protected from concurrent access by a mutex in order to keep the table consistent. When a new connection has been received, a free entry is allocated from the thread table and the thread counter is incremented. If there is not immediately a free entry in the thread table, the main thread waits for the condition variable *thread_table_not_full.* Any other pending connection is kept in the queue of connections having the length specified by the *listen* function. The main purpose of the thread table is to deliver multiple arguments, such as the client socket file descriptor and socket address, to the new thread, because the thread started with *pthread_create* can only take one argument. The thread table has a fixed size, which means that only a limited number of clients can do concurrent requests. The main reason for this is to limit the harm of a DOS attack, so that new threads are not created uncontrolled. When a thread finishes execution, it releases its thread table entry, decrements the thread count and signals the condition variable *thread_table_not_full*. When the last thread has been released, the condition variable *thread_table_empty* is signaled. This condition variable is used as a synchronization mechanism when the server is shutting down and waits for all threads to finish without accepting new requests.

A mutex is also used for memory management (in *safer_malloc* and *safer_free*) in order to prevent inconsistencies in the memory management of the standard library, which is not guaranteed to be thread safe. Another mutex needs to be used when writing log messages, so that different threads writing log messages do not mix the messages up.

Handling requests

When a new connection is received by the *accept* function, an entry in the thread table is allocated, the client socket file descriptor and socket address is put in the thread entry and a new thread is created using *pthread_create*. The new thread runs (through a wrapper function) the *serve_request*, which does the main high level task of serving and dispatching a request. In *serve_request,* the IP address and port of the client is logged, the request header is received and parsed (*parse_header*), the request path is decoded (*decode_request_path*) and validated (*validate_request_path*). After this a local filename is generated from the file storage root path (from the CONFIG struct and indirectly from the command line option --root <*root-dir*>) and the request path (from the request line). Finally the request can be dispatched based on the requested method: GET or PUT.

The GET requests are dispatched to the *serve_get* function. In this function a new header struct is reserved for the reply, the local file is opened (based on the local filename described above), *fstat*-ed to get the file size and modification time and the reply header is filled in. If the requested file is a directory, a buffer is allocated for the directory listing, which is done in *list_directory.* The header is sent, and then, if the file is a regular file (and not a directory) the content of the local file is sent using the *send_binary_data* function (the same function as in assignment 1). If the local file is a directory, the buffer with the directory listing is sent.

The PUT requests are dispatched to the *serve_put* function. In this function the validity of the Content-Length header line in the request is checked, the local file is opened for writing, *fstat*-ed to check if the file is a directory. In this server implementation it is not possible to PUT to a directory. Then the request body is received using the *receive_binary_data* function (the same function as in assignment 1). A reply header struct is reserved, filled in and sent.

Parsing

Parsing headers is done in a similar way as in assignment 1. A status code has been added (as an output parameter) to distinguish different types of failure when parsing. The *parse_header* function must also be able to parse request headers; request lines are parsed in *parse_request_line.* Otherwise header lines follow the same name–value pair syntax as in assignment 1. The *parse_status_line* is the same as in the first assignment.

Decoding and validation of request paths

Characters in the request path (in the request line) can be encoded using the %XY syntax, where X and Y are hexadecimal digits. In his way URLs can contain spaces and non-ASCII characters in the request path. The request path is decoded in the *decode_request_path* function using a finite state machine (see figure 1). A transition in the figure without a label is a wildcard label matching any other character.

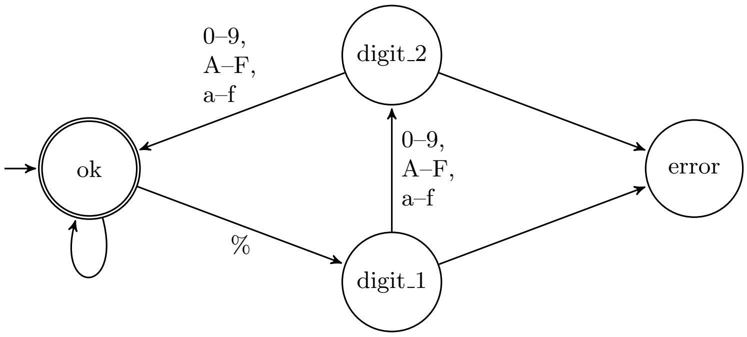

Figure 1. The request path decoding finite state machine.

When a request path has been decoded, it is validated (in *validate_request_path)*. A request path comprises segments separated by slashes (/), e.g. the request path /dir/file consists of the segments *dir* and *file.* A request path is not considered valid if any segment is . (dot) or .. (dotdot). The dotdot is considered invalid for security reasons, so that a request cannot break out of the file storage root directory and for consistency the other special directory entry (dot) is also considered invalid (apart for being quite useless in a request path). Paths with subsequent slashes are considered invalid (e.g. /dir///file). A single slash is valid; this is the file storage root directory. Directories can be requested with or without a trailing slash (/dir is the same as /dir/).

Build and installation instructions and requirements

The HTTP server is most easily built using the provided makefile, simply by typing *make*.

User instructions

The HTTP server is started using the following syntax:
./http-server *options*
where *options* can be any of the following:

--foreground

The server is not daemonized and writes log messages to stdout and stderr (in addition to the log file).

--port *port-number*

Sets the port number the server is listening to. The same port number is used for both IPv4 and IPv6. The default port number is 54321.

-- root *root-directory*

Sets the root directory for the file storage. Files are *GOT* from and *PUT* to this directory. The default root directory is the *files* directory inside the current directory.

--ipv4
--ipv6

Listen only to connections over IPv4 or IPv6. If none or both of the --ipv4 and --ipv6 options are given, the server listens to connections over both protocols.

--no-discovery

Do not announce the server to the discovery service when starting and shutting down.

--threads *max-threads*

Sets the maximum number of concurrent threads. Default is 5.

--backlog *backlog-length*

Sets the length of the backlog queue of connections. Default is 10.

Testing and known limitations

Tests

The tests presented here are also performed on two other students' implementations in answer 5 in the 'Answers to specific questions' section and the results are presented there. The results of my own implementation are presented here.

I. **Basic PUT and GET test.** A local file is PUT to the server using the HTTP client from assignment 1. After that the same file is GOT from the server. The file sizes and contents should match before and after. This can be ensured using *ls* and *diff.* Result: the server accepted both requests and the sent and received file had the same size and contents. The server replied with status 200 OK for both requests.

II. **Replace with smaller file.** A smaller file than in test I is PUT and GOT but with the same name. The new smaller file should replace the old. File sizes and contents are checked using *ls* and *diff.* Result: the server accepted both requests and the sent and received file had the same size and contents. The server replied with status 200 OK for both requests.

III. **Get nonexistent file.** Try to get a file with a filename that most likely does not exist (this can be confirmed for my own implementation, but not for the ones done by other students). Result: the file could not be downloaded and the server replied with status 404 Not Found.

IV. **Gibberish in request line.** An invalid request line consisting entirely of gibberish is sent to the server using *netcat.* Result: the server replied immediately after the request line with status 400 Bad Request.

V. **Gibberish method name.** An invalid method is requested in an otherwise valid HTTP request line using *netcat*. Result: the server replied immediately after the request line with status 501 Not Implemented.

VI. **Malformed header line.** An invalid header line that does not conform to the 'Name: value' syntax is sent to the server. Result: the server replied with status 400 Bad Request.

VII. **PUT without Content-Length.** A PUT request without a Content-Length header is done using *netcat*. Result: the server replied with status 411 Length Required.

VIII. **PUT to root.** A PUT request is made to upload a file to the root (the request path is /). Result: the server replied with status 405 Method Not Allowed.

IX. **PUT and GET 2MB file.** A 2 MB file is uploaded and downloaded. The file sizes and contents are compared using *ls* and *diff*. Result: the sent and received file had the same size and contents. The server replied with status 200 OK for both requests.

X. **Aborted download.** A 2 MB file is downloaded, but the download is aborted before it is finished. If the server does not ignore the SIGPIPE signal, it might terminate. A subsequent request is made to make sure that the server is still responding. Result: the server logged that there was an error when sending and freed the resources and thread used for the request. No status could be sent to the client, because the connection was already closed at that point. The server responded to subsequent requests.

Limitations

- Currently there is no timeout mechanism for receive and send operations.
- Directory listings are limited to one buffer size (default is 8192 bytes, defined in config.h), because the content (the directory listing) is not on disk, but created on request.
- Query strings (the '?' and every thing after that in a relative path) are not detected.
- When a file is created for the first time using PUT, the status returned is 200 OK, but a more appropriate status would be Create 201. Currently there is no distinction between creating a file and updating it.

Answers to specific questions

1. Multiple clients are served concurrently by a limited number of threads (5 in the default server configuration). The main reason to keep the number of concurrent threads limited is to limit the harm of a DOS attack, so that new threads are not created uncontrolled. For a more detailed description of how threads are managed and used, see section 'Thread handling' above. Currently, if a client keeps a connection open for an indefinite time, the server does not close the connection. There is no timeout mechanism for receiving or sending.
2. By default the server supports connection over both IPv4 and IPv6 on the tested systems (Linux and Mac OS X). An IPv6 socket accepts both IPv4 and IPv6 connections if the IPV6_V6ONLY option is not set on the socket (this is done using the *--ipv6* option). This means that a single socket is used for receiving both IPv4 and IPv6 connections. If the server is started with the *--ipv4* option, a single IPv4 socket is created instead.
3. To test the behavior of the server under high load, the client and server machines were directly connected using gigabit Ethernet in order to really test the server and not the (possibly slow) network link. The *ab* (Apache Bench) tool was used on the client computer to generate requests in a controlled way. The client computer ran Mac OS X 10.6.8 and the server computer ran Linux 3.5.0–23 as the operating system. In each of these tests, *ab* sent 20 concurrent requests and 3000 requests in total. Initially the HTTP server ran with 5 concurrent threads and a backlog length of 10 (the default values). This gave minimum reply times of 2–8 ms, mean reply times of 14–

48 ms and maximum reply times of 1921–2805 ms. When the thread concurrency level was raised on the server to 10 (concurrent threads, using the *--threads 10* option), similar reply times were obtained: minimum 2–8 ms, mean 20–51 ms and maximum 929–3081 ms. When the backlog length was raised to 20 (connections waiting) and the thread concurrency level was kept at 10 (using the options *--threads 10 --backlog 20),* the following reply times were obtained: minimum 2–9 ms, mean 11–40 ms and maximum 24–70 ms). The conclusion that can be drawn from this experiment is that just raising the thread concurrency from 5 to 10 did not improve the performance much, but when the backlog length also was raised from 10 to 20, the performance increased significantly.

4. This can most easily be tested by downloading a big file (I tried with a 100 MB file) and then terminating the client (e.g. wget or the HTTP client from assignment 1) with Ctrl-C. I did have problems with this case; the server terminated when a download was aborted. Then I found out that the SIGPIPE signal is triggered when the connection is lost. The return value of every receive and send (fread and fwrite) call is checked anyway, so there is no need to use the SIGPIPE signal in the server. Hence, the SIGPIPE signal is configured to be ignored in the *init_signals* function. Now when a download is aborted, the server notices this as an error condition and frees allocated resources and terminates the thread serving the request.
5. The same ten tests described in the 'Tests' section were performed on two other students' implementations. The result of my own implementation is included for comparison.

Test	jacquov1's implementation	tvkamara's implementation	my own implementation
I. Basic PUT and GET test.	File size and contents are the same. Server replied: PUT: *201 Created* GET: *200 OK*	File size and contents are the same. Server replied: PUT: *200* GET: *200 OK* No reason phrase in put reply status line.	File size and contents are the same. Server replied: PUT: *200 OK* GET: *200 OK*
II. Replace with smaller file.	File size and contents are the same. Server replied: PUT: *201 Created* GET: *200 OK*	File size and contents are the same. Server replied: PUT: *200* GET: *200 OK* No reason phrase in put reply status line.	File size and contents are the same. Server replied: PUT: *200 OK* GET: *200 OK*
III. Get non-existent file.	Server replied: *404 Not Found* Iam header not included.	Server replied: *404 Not Found*	Server replied: *404 Not Found*
IV. Gibberish in request line.	Server replied: *501 Not Implemented*	Server closed connection, no reply.	Server replied: *400 Bad Request*
V. Gibberish method name.	Server replied: *501 Not Implemented*	Server closed connection, no reply.	Server replied: *501 Not Implemented*
VI. Malformed header line.	Server replied: *200 OK* Content delivered.	Server replied: *200 OK* Content delivered.	Server replied: *400 Bad Request* Content not delivered.
VII. PUT without Content-Length.	Server replied: *204 No Content*	Server closed connection, no reply.	Server replied: *411 Length Required*
VIII. PUT to root.	Server replied: *403 Forbidden*	Server replied: *Failed to create the file.* Not a valid HTTP status line.	Server replied: *405 Method Not Allowed*
IX. PUT and GET 2MB file.	No reply from server. Even with a 100 kB file, the server gives no reply.	File size and contents are the same. Server replied: PUT: *200*	File size and contents are the same. Server replied: PUT: *200 OK*

		GET: *200 OK*	GET: *200 OK*
X. Aborted download.	No reply. Subsequent requests OK.	No reply. Subsequent requests OK.	No reply. Subsequent requests OK.

Diary

Week 7: 8 hours

Implemented a very basic server with limited parsing and validation and serving a fixed file (not the file requested). Some code reused and updated from the HTTP client.

Week 8: 8 hours

Improved parsing, validation and error recognition and handling.

Week 9: 10 hours

Implemented request path decoding and validation. Implemented the PUT method.

Week 10: 24 hours

Implemented daemonization and threading. Refactorized and commented the code. Started with the documentation.

Week 11: 24 hours

Implemented server discovery announcements and directory listings. Did some testing and finished the documentation. I had some problems with the SIGPIPE signal being triggered, but got it fixed (see answer 4 in the 'Answers to specific questions' section).