FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Project documentation

HTTP bulletin board

Brno
13. October 2019

Marek Varga
xvarga14

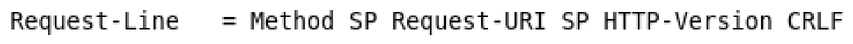
Content

1 Message format

HTTP message can be either a request or a response. Request is a message from client to server. Response is a message from server to client. Messages of HTTP/1.1 version have strict format. All messages begin with start-line, followed by header fields, blank line and message body. There may be zero or more header fields and message body is optional. Blank line is there to separate header from the body. Start-line differs, depending whether message is request or response.

1.1 Request message

The start-line of request message is called request line. R. Fielding (2014a) Request line consists of method, space character, request-target, space charecter, HTTP-version and CRLF.
Request method can be one of the following: GET, HEAD, POST, PUT, DELETE, POST, PUT, CONNECT, OPTIONS and TRACE all case sensitive. For the purpose of this project only GET, POST, PUT and DELETE methods were used. The GET method is used for retrieving information from the server. „The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.“ R. Fielding (1999) The DELETE method requests the server to delete the entity specified in Request-URI. „The PUT method replaces all current representation of the target resource with the request payload.“ R. Fielding (2014b)



```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

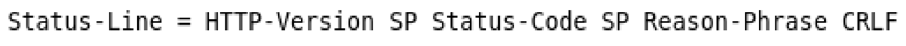
Fig. 1: Request line, from: https://tools.ietf.org/html/rfc2616

Request-target can also be called Request-URI. „The request-target identifies the target resource upon which to apply the request.“ R. Fielding (1999) HTTP-version used in this project is HTTP/1.1. Last of the request line are characters representing CR and LF.
Folling request line are other header fields. Only header field required in request message is Host. In this field host and potentionally port can be specified. If port is present then host address and port are separated by colon. If port is not present then port is set default value 80.
Request method and header fields together make the request header. Following the request header is a blank line, simply written as CRLF. Message body is optional and is only used in request message when the request method is POST or PUT.
Example request header may look like this:

GET /boards HTTP/1.1
Host: localhost:2019

1.2 Response message

The start-line of response message is called status line. R. Fielding (2014a) Status line consists of HTTP-version, space character, status code, space character, reason phrase and CRLF.



```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Fig. 2: Status line, from: https://tools.ietf.org/html/rfc2616

HTTP-version used in this project is HTTP/1.1.

Status code is a three digit number. Status codes of format 1xx are informative, 2xx are successful, 3xx are redirection, 4xx are client errors and 5xx are server errors. For the purpose of this project only 200 (OK), 201 (Created), 400 (Bad request), 404 (Not found) and 409 (Conflict) codes were used.
Reason phrase provides textual description of the status code, e.g. Created for response code 201.

When the response message contains message body then the response header also contains header fields specifying type and lenght of the body. „Content-type indicates the media type of the entity-body sent to the recipient.“ R. Fielding (1999) Value of the Content-type parameter is in the form type/subtype. For the purpose of this project Content-type: text/plain was used. Content-Lenght is a decimal number indicating number of octets present in body.
Following response header is the blank line written as CRLF. Message body is present in some of the response messages. It is typically present after GET request.
Example response header may like this:

HTTP/1.1 200 OK

2 Implementation specifications

This section provides some details of my implementation.

2.1 Communication

„HTTP communication usually takes place over TCP/IP connections.“ R. Fielding (1999) Therefore the client-server communication is done through TCP. There are a few steps int TCP communication. First a socket needs to be created with *socket()* function. Then the socket needs to be assigned port number and transport address via call to the *bind()* function. Then server waits for incoming connection by calling *listen()* and all pending connections will be queued up until server satisfies their demand. By calling *accept()* function, server takes the first pending connenction from the queue. The number of pending connections is set to ten. After server takes the first pending connection it calls *read()* function to obtain the message from client and then calls *write()* to send a message to client. In the end the socket is closed via *close()* function. The communication is always initiated by client, client always sends requests, and server is passive, server sends responses.
Based on client's command line, a request message is built according to the format specified in the section *Request message* above. Appropriate method is chosen and URI is loaded from the command line arguments. If the command line arguments do not correspond to those specified in the task then client is exited with exit code 1, no communication is initiated. Otherwise the request is built and sent to the server. Request also contains HTTP-version, Host header fields. It the request method is POST or PUT then the request header also contains Content-Type and Content-Length fields.
Server is iterative therefore can satisfy only one client's demand at a time. When a request message is received header is parsed and checked. HTTP-version, request method and request-URI are checked. When unknown request method, wrong HTTP-version or bad request-URI are received server is not doing any action and sends response message containing status code 404 and appropriate error in the response body, e.g. client receives a response contating status code 404 and body will hold „Unknown method for /boards manipulation“ when request is sent with the PUT method and */boards/*$< name >$ request-URI.
Server's bulletin board is a linked list with each element a structure holding the board's name, board's content and a pointer to the next board. Board's content is a linked list with each element a structure holding id, some text and a pointer to the next content.
Once header is parsed, server processes the request according to task. The GET method with */boards* URI loads the names of all the boards. The POST method with */boards/name* URI creates new board with name *name*. The DELETE method with */boards/name* URI deletes the board with the name *name*. The GET method with

/board/name URI loads the content of the board *name*. The POST method with */board/name* URI creates new content for the board *name*, content is located in the request body. The PUT method with */board/name/id* URI overwrites the content at *id* at the board *name* with the content located in the request body. The DELETE method with the */board/name/id* URI deletes the content at *id* at the board *name*. Any other of the request method and request URI leads to status code 404. Also POST and PUT methods with content length of 0 lead to status code 400.
After request is processed, server builds the response message. Depending on the result of the request processing, the response may or may not contain the body and body can be an error, names of the boards or content of the board. For example, after successful processing of POST request method, the response will not contain any body. On the other hand the POST request method, with an attempt to create an already created board will result in the response with the body carrying an error „Board already exists“. The response is then sent to client. Client parses the response message to header and body. The header is printed to *stderr* and the body is printed to *stdout*.

2.2 Program termination

Since server is implemented as an infinite loop, one way of terminating the program is generating *SIGINT*. There is function called *signalHandler* registered, via a call to *signal()* function, to handle *SIGINT*. There also is a function called *atExitFunction* registered to normal process termination. This function is registered via a call to *atexit()* function.
When *SIGINT* signal is generated, the *signalHandler* is called. It prints to *stderr* that *SIGINT* was detected and calls *exit()*. Function *exit()* is a call to normal process termination and therefore *atExitFunction* is called, which disposes all the boards, frees allocated memory and closes socket descriptor. I did not think of a better way to free all allocated memory that this.

2.3 Further restrictions

How to run server and client is written in the *Usage* section. However port number must be greater than 1024 because lower ports are system reserved.
The example run was run with valgrind and no memory leaks or errors were found.

3 Usage

The server part needs to be running when the client sends a request. The server program is called *isaserver* and the client program is called *isaclient*. Both programs take *-h* command line argument to display info about usage. The server takes *-p* $< portnumber >$ command line argument which specifies the port number. The client takes *-H* $< hostname >$ *-p* $< portnumber >$ $< command >$ command line arguments where hostname is address of the server, portnumber is portnumber on which the server is running and command is one of the following: *boards*, *board add* $< name >$, *board delete* $< name >$, *board list*, *item add* $< name >< content >$, *item delete* $< name >< id >$, *item update* $< name >< id >$.

3.1 Example run

Run server from the command line: *./isaserver -p 2019*
Now run client with various ¡command¿ arguments and watch output
Run client from the command line: *./isaclient -H localhost -p 2019 board add nastenka1* to create new board with name *nastenka1*
Output:
HTTP/1.1 201 CREATED

Run client from the command line: *./isaclient -H localhost -p 2019 board add nastenka2* to create new board with name *nastenka2*
Output:
HTTP/1.1 201 CREATED

Run client from the command line: *./isaclient -H localhost -p 2019 item add nastenka2 prispevok1* to add content *prispevok1* to board *nastenka2*
Output:
HTTP/1.1 201 CREATED

Run client from the command line: *./isaclient -H localhost -p 2019 item add nastenka1 prispevok1* to add content *prispevok1* to board *nastenka1*
Output:
HTTP/1.1 201 CREATED

Run client from the command line: *./isaclient -H localhost -p 2019 item add nastenka prispevok1* to add content *prispevok1* to board *nastenka*

Output:
HTTP/1.1 404 NOT FOUND
Content-Type: text/plain
Content-Length: 19

No such board exists

Run client from the command line: *./isaclient -H localhost -p 2019 boards* to display all boards
Output:
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 20

```
nastenka1
nastenka2
```

Run client from the command line: *./isaclient -H localhost -p 2019 item add nastenka2 prispevok2* to add content *prispevok2* to board *nastenka2*
Output:

```
HTTP/1.1 201 CREATED
```

Run client from the command line: *./isaclient -H localhost -p 2019 board list nastenka2* to display all content of board *nastenka2*
Output:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 42

nastenka2
1. prispevok1.
2. prispevok2.
```

Run client from the command line: *./isaclient -H localhost -p 2019 board delete nastenka1* to delete board with name *nastenka1*
Output:

```
HTTP/1.1 200 OK
```

Run client from the command line: *./isaclient -H localhost -p 2019 boards* to display all boards
Output:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 10

nastenka2
```

Run client from the command line: *./isaclient -H localhost -p 2019 item update nastenka2 2 novy prispevok2* to update content located at position *2* at board with name *nastenka2* with content *novy prispevok2*
Output:

```
HTTP/1.1 200 OK
```

Run client from the command line: *./isaclient -H localhost -p 2019 board list nastenka2* to display all content of board *nastenka2*
Output:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 47

nastenka2
1. prispevok1.
2. novy prispevok2.
```

Run client from the command line: *./isaclient -H localhost -p 2019 item delete nastenka2 2* to delete content located at position *2* at board with name *nastenka2*

Output:

```
HTTP/1.1 200 OK
```

Run client from the command line: *./isaclient -H localhost -p 2019 board list nastenka2* to display all content of board *nastenka2*
Output:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 26

nastenka2
1. prispevok1.
```

Run client from the command line: *./isaclient -H localhost -p 2019 board add nastenka2* to create new board with name *nastenka2*
Output:

```
HTTP/1.1 409 CONFLICT
Content-Type: text/plain
Content-Length: 19

Board already exists
```

Run client from the command line: *./isaclient -H localhost -p 2019 item delete nastenka2 2* to delete content located at position *2* at board with name *nastenka2*
Output:

```
HTTP/1.1 404 NOT FOUND
Content-Type: text/plain
Content-Length: 28

No such board content exists
```

Run client from the command line: *./isaclient -H localhost -p 2019 item add nastenka2* to add empty content to board with name *nastenka2*
Output:

```
HTTP/1.1 400 BAD REQUEST
Content-Type: text/plain
Content-Length: 45

Content-Length of POST request cannot be zero
```

4 Sources

R. FIELDING, e. a. *Hypertext Transfer Protocol – HTTP/1.1* [online]. The Internet Society, 1999. [cit. 13. 10. 2019]. Dostupné z: <https://tools.ietf.org/html/rfc2616>.

R. FIELDING, J. R. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing* [online]. Internet Engineering Task Force, 2014a. [cit. 13. 10. 2019]. Dostupné z: <https://www.rfc-editor.org/rfc/pdfrfc/rfc7230.txt.pdf>.

R. FIELDING, J. R. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [online]. Internet Engineering Task Force, 2014b. [cit. 13. 10. 2019]. Dostupné z: <https://tools.ietf.org/html/rfc7231>.