Network Programming Laboratory
Winter 2018/2019

Session 2:
A Simple Webserver

Janne Riihijärvi

RWTH Aachen University - Institute for Networked Systems

Contents

1 Overview

Using a TCP connection with sockets is very similar to using a UDP connection as was done in the previous session. The aim of this session is to program a simple Webserver, which basically means transporting a file over a TCP connection in reply to a HTTP (Hypertext Transfer Protocol) request.

The simplest way to program such a server is the iterative way: accept a connection, process the request and close the connection. This however has the disadvantage that no new request can be processed as long as the old connection is open. To overcome this, there are mainly three possibilities: forking a new process for every connection, using I/O multiplexing for the different connections or using a thread for each connection. We will focus on the first alternative towards the end of this session, and will cover the latter ones in subsequent lab sessions.

2 What a Web Browser Does

2.1 Uniform Resource Locator

When you want your Browser to fetch and display some web document, you give it its URL (Uniform Resource Locator, also known as scheme), for example:

`http://www.google.de/index.html`

The URL consists of three parts:

1. The name of the protocol (here `http`)
2. The DNS name of the server (here `www.google.de`) or the IP address
3. (Usually) the name of the file (here `index.html`) relative to the server's default web directory. If the file name is not specified, the server uses a default value, for example `index.html`.

What the browser then has to do (in this case) is

1. Ask the DNS for the IP address of the server (reply is "173.194.35.183")
2. Open a TCP connection to port 80 ("well-known" port for `HTTP`) to 173.194.35.183
3. Ask for the file by sending a HTTP request
4. Read the file from the socket
5. Display the file (interpret the HTML code)
6. (Fetch and display images in the file)

A general issue with URLs is that they point not only to a file, but also to its location. A way to separate these two aspects are URNs (Universal Resource Names). They were recently specified by the IETF, for information see e.g. RFC 2141 or search on the IETF homepage.

2.2 Hypertext Transfer Protocol

HTTP specifies how web clients (browsers) request web pages from servers, and how the servers transmit pages to the clients. Until 1997 browsers used HTTP 0.9 or HTTP/1.0, which is defined in RFC 1945. From 1998 on HTTP/1.1 was implemented, which is defined in RFC 2616. HTTP/1.1 is backward compatible to version 1.0. Both use TCP as transport protocol.

HTTP does not store information about the clients, it is a stateless protocol. The messages consist of normal ASCII text (except for the message body).

Persistent and Nonpersistent Connections

A nonpersistent connection is a connection that is closed directly after the requested object was transmitted. If a HTML page contains embedded objects (references to them), a new connection must be established for every single object to fetch — several connections for a single page. Persistent connections are kept open after the transmission. HTTP/1.0 works with nonpersistent connections, version 1.1 supports both (and default is persistent). Persistent connections have the advantage of saving overhead (for every connection TCP buffers and variables must be stored at the client and server) and avoiding the TCP slow-start phases. There are some other interesting issues of HTTP like e.g. Pipelining - if you are interested, have a look at the RFC.

Structure of a Request

A typical request looks like:

```
GET /somedir/page.html HTTP/1.0
Host: www.something.de
Connection: close
User-agent: Mozilla/4.0
Accept-language: de
```

A HTTP request consists of one or more lines. After the last line another return (carriage return and line feed) follows. The first line is the request line. The following lines are the (mostly optional) header lines. The request line consists of three fields:

- Method
- URL
- HTTP version

The method field can have several values, among them `GET`, `POST` and `HEAD`. The `GET` method is used to request an object, which is specified in the URL field. The `POST` method is used when the user fills in a form (e.g. entering words for Google to find). In that case the message contains an entity body (the data the user entered) after the header. The `HEAD` method is similar to `GET`, the server gives the same reply header but without the requested object. `HEAD` is often used for debugging. The structure of a HTTP request is shown in figure 1.

Structure of a Reply

A reply has a very similar structure, which is shown in figure 2. It contains a status line, possibly header lines, and the entity body. The first line is the status line. It contains the version, the statuscode and phrase. The most common status codes and phrases together with their meanings are:

- `200 OK`: request was succesful, the requested information is in the entity body.
- `300 Moved Permanently`: the object was moved permanently, the new URL is provided in header line `Location`.

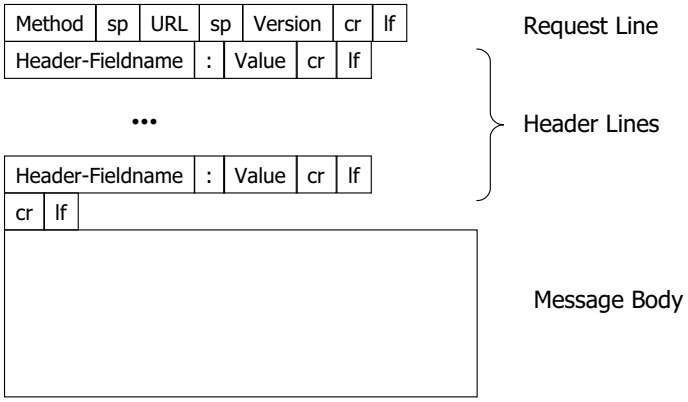

Figure 1: Structure of a HTTP request.

- `400 Bad Request`: Generic error code, server could not interpret the request.
- `404 Not Found`: document is not on the server.
- `505 HTTP Version Not Supported:` the used HTTP version is not supported by the server.

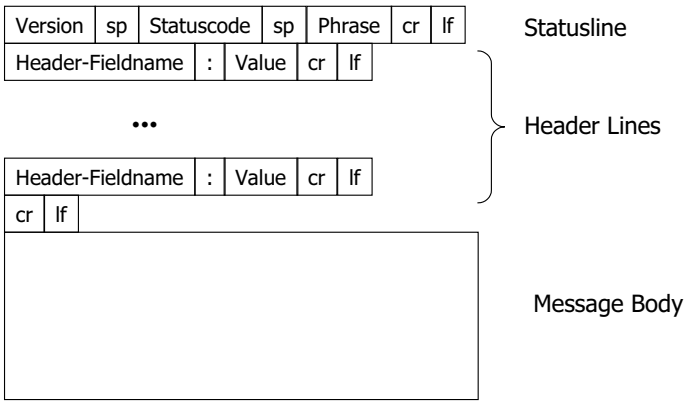

Figure 2: Structure of a HTTP reply.

Example

As HTTP is character based, you can use telnet to talk to a server:

```
gracchus:~ #  telnet caracus.mobnets.rwth-aachen.de 80          // telnet connection to port 80
Trying 134.130.222.55...
Connected to 134.130.222.55.
Escape character is '^]'.
GET /index.html HTTP/1.0  // This is the line I type,
```

```
 // then twice return
HTTP/1.1 200 OK  // Reply from the server
Date: Mon, 26 Apr 2010 16:20:39 GMT  // header lines starting
Server: Apache/2.2.13 (Linux/SUSE)
Last-Modified: Mon, 26 Apr 2010 14:05:43 GMT
ETag: "466334-4913-4852445c99bc0"
Accept-Ranges: bytes
Content-Length: 18707
Connection: close  // non-persistent
Content-Type: text/html  // MIME type

<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitio
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml" lang="en"><head>


..
[SNIP]
..
</body></html>Connection closed by foreign host.

gracchus:~ #
```

2.3 Static Web Documents

Figure 3: The browser interpretation of the HTTP reply.

Static web documents are files lying on the server. They are returned by the server including the MIME type (Multipurpose Internet Mail Extensions) to

describe the file type — e.g. `html`, `jpg`, `mp3` etc. If the browser cannot interpret the file type, it uses a plugin or a helper application.

Plugins run inside the browser and therefore have access to the current page and can modify its appearance. The interface to the browser is a set of (browser-specific) procedures that the plug-in implements.

Another way is to use a helper application which runs as a separate process. Instead of offering/using a browser interface it just accepts the name of a scratch file where the content has been stored.

HyperText Markup Language

HTML is the standard web page description language and allows pages to include text, graphics, hyperlinks, forms for two-way traffic (from HTML 2.0 on), tables, scripting (from HTML 4.0 on), etc.
There are many good tutorials for HTML on the web, so it is not further described here.

XML, XSL and XHTML

HTML does not provide any specific structure for web pages. As e-commerce and other applications became more common, a structure to separate between content and formatting was needed to better support automated processing. Therefore two new languages have been developed by the W3C (WWW consortium): XML (eXtensible Markup Language) describes web content in a structured way and XSL (eXtensible Style Language) describes the formatting. Web pages can be written in XML and then converted to HTML according to an XSL file.

2.4 Dynamic Web Documents

Dynamic web pages are generated on demand. They can be either generated on the server or on the client. A common reason to generate a page on the server on demand is the reply after database access. Reasons to create a page dynamically on the client can be to react to mouse movements (moving over objects in the page) or to interact with the user directly.

Therefore, starting with HTML 4.0, it is possible to have scripts embedded in HMTL pages (with the tag `<script>`), which are executed on the client machine. The most popular scripting language for the client side is JavaScript.

Another possibility for interactive web pages is to use applets - small Java programs that can be embedded in HTML pages using the `<applet>` tag.

Microsoft's answer to Sun's Java applets was allowing web pages to hold ActiveX controls, which are programs compiled to Pentium machine code and directly executed on the hardware (not interpreted like applets by the Java

Virtual Machine). This feature however raises security issues (applets are restricted in what they can do).

3 What a Webserver Does

3.1 Static Web Documents

Static web content means files lying on the server. Providing them essentially is:

- Accept a TCP connection from a client (a browser)
- Get the name of the file requested
- Get the file (from disc)
- Return the file to the client
- Release the connection

Normally the most famous files are kept in a cache to avoid slow disk access. What is not considered here are for example things like access control to certain pages, MIME types to include in the response (determined by file extension, configuration file etc.), logging and TCP connection handoffs in server farms between nodes.

3.2 Dynamic Web Documents

Dynamic generation of content on the server is for example needed when forms are used. When a user fills in a form and sends it, this is not a request for a pre-existing file to return. Instead the data should be given to a program or script to process. Most often a database is to be accessed. Then a HTML page may be generated and returned. Four common ways to generate dynamic content are CGI (Common Gateway Interface), PHP (PHP: Hypertext Preprocessor), JSP (JavaServer Pages) and ASP (Active Server Pages).

CGI is a standardized interface between the Webserver and a backend program or script (often programmed in Perl or Python). These scripts are by convention stored in a directory called `cgi-bin`, which is visible in the URL.

PHP is a language to write small scripts embedded in HTML pages, which are executed by the Webserver to generate the page. Pages containing PHP scripts have the extension `php` .

JSP is similar to PHP, except that the dynamic part is written in Java instead of PHP. Pages using this technique have the extension `jsp` .

ASP is Microsoft's version of PHP and JavaServer Pages. It uses Visual Basic Script for generating the dynamic content. The file extension is `asp` .

4 General Server Design Alternatives

4.1 Iterative Server

The iterative server is the easiest server possible: it accepts a connection, processes the request, sends the answer and terminates the connection. The disadvantage is that no new connection is possible as long as the old one is open. If you program such a server, you can easily test it: open a telnet connection, then open another one (or use a browser). The second connection will be refused. This of course poses a problem, especially with persistent connections of HTTP/1.1.

4.2 Concurrent Server

4.2.1 Fork(): Creating New Processes

`fork()` is the only way in Unix/Linux to create a new process. This system call is called once, but returns twice. It returns once in the calling process (the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence, the return value tells the process whether it is the parent or the child. The child can get the parent's process ID by calling `getppid()`.

Most network servers under Unix are written using `fork()`. The parent process cares for the listening socket, accepts a new connection, forks a child, and the child handles the client. The outline of a server using `fork()` is (without caring about error handling here for clarity):

```
pid_t pid;
                                    // listening on listenfd
for (;;)                            // endless loop
{
    connfd = accept(listenfd, ...);    // accept new connection, probably blocks
    if ((pid = fork())==0)             // this is only true for child
    {
        close(listenfd);               // child closes listen socket
        doclientstuff(connfd);         // process request
        close(connfd);                 // close connection
        exit(0);                       // child terminates
    }
    close(connfd);                     // parent closes connected socket
}
```

To understand this, it must be mentioned that every file descriptor and socket descriptor has a reference count, the number of processes where this file/socket is open. `close()` only terminates a connection when the reference count is zero. Directly after forking, it is 2. When the child calls `close()` for the listening socket, the reference count is reduced to 1, therefore the connection is not terminated and the parent keeps listening (analog for the connected socket).

4.2.2 Further Options

Many functions on sockets and files are by default blocking (read, accept, write when the sending buffer is full,...). If one process handles several sockets/files at once, this is normally not wanted. It could be necessary for example to wait both for input from the user (read from stdin) and from the network (read from socket, accept a new connection). The `select()` call provides this I/O multiplexing by waiting for a list of events. It returns when one or more of the specified events have happened and can tell you which. We will study the use of `select()` for handling multiple clients in a future session.

5 More From the Sockets API

5.1 Connection Termination

5.1.1 shutdown

When the server has sent the requested file, it can terminate the connection. Using `close()`, the connection would be immediately terminated, and data still being in the sending buffer would be discarded. To avoid this, you can use `shutdown()` instead of `close()`. Then the connection will only be terminated after the successful transmission of all the data from the sending buffer.

5.2 Name and Address Conversions

This subsection will be interesting for you especially as a reference for later sessions in which addressing issues and name resolution will be covered in more detail. For an HTTP client you would probably want to use the server's DNS name to reach it instead of it's IP address. Also `'http'` can be automatically resolved to port 80 using the list of well-known ports (`/etc/services`). Additionally, functions are available that work both with IPv4 and IPv6 addresses.

5.2.1 gethostbyname and gethostbyaddr

`gethostbyname` and `gethostbyaddr` convert between IPv4 addresses and hostnames. Both funtions return a pointer to a hostent struct, which looks like:

```
struct hostent {
  char  *h_name;       // official name of the host
  char **h_aliases;    // pointer to array of pointers to alias names
  int    h_addrtype;   // host address type: AF_INET
  int    h_length;     // length of address: 4
  char **h_addr_list;  // ptr to array of ptrs with IPv4 addrs
};
```

5.2.2 getservbyname and getservbyport

`getservbyname` and `getservbyport` convert between service names and port numbers. Both return a pointer to a servent struct, which looks like:

```
struct servent {
  char  *s_name;     // official service name
  char **s_aliases;  // alias list
  int    s_port;     // port number, network byte order
  char  *s_proto;    // protocol to use;
};
```

5.2.3 getaddrinfo and getnameinfo

`getaddrinfo` and `getnameinfo` convert between hostnames and IP addresses (IPv4 and IPv6) and between service names and port numbers.

6 Resources for Further Information

- The main reference for this session is "Unix Network Programming" by W. Stevens.
- Interesting (although slightly outdated) might be "Web Server Technology" by N. Yeager.
- For real things you might want to have a look at the open-source Mozilla Web-Suite or the Firefox standalone-Browser for the client side, and the open-source Webserver Apache (the most often used Webserver).

7 Programming Task

The programming task is a simple Webserver. It should be able to provide a file out of a certain directory in reply to a HTTP request. It is of course not supposed to support scripts and forms. It is therefore enough to only read the request line of an HTTP request, only process a `GET`, identify the requested file and send it back. An iterative server is fine, but in case you have time left, try a concurrent server, first based on forking.

The server would be tested with a standard browser (Mozilla or Konqueror).