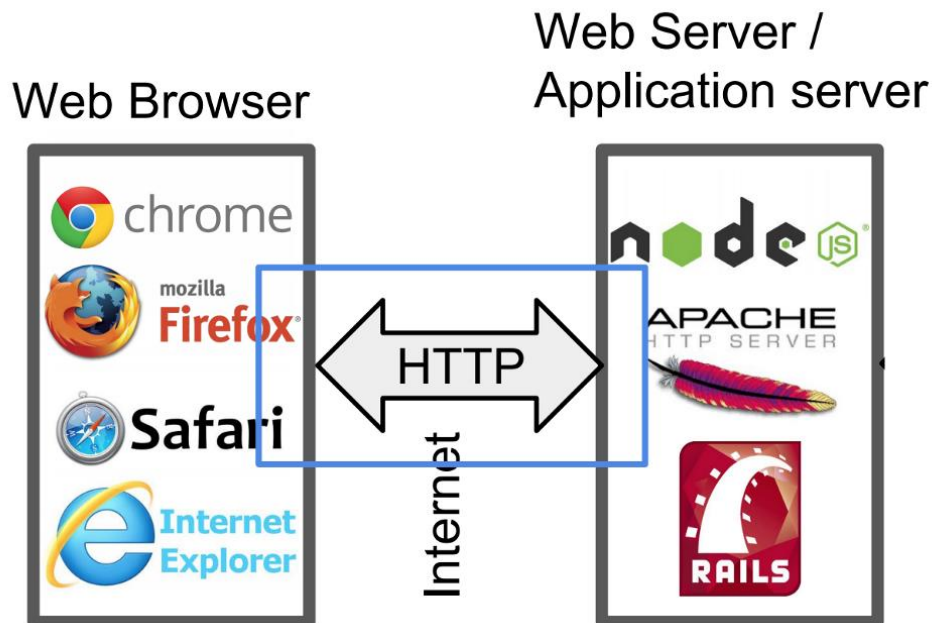


HTTP Server Design and Implementation from Scratch

HTTP Server: Introduction

- An HTTP Server is software that understands URLs (web addresses) and HTTP (the protocol browser uses to view pages).
- An HTTP Server can be accessed through the domain names of the websites it stores and it delivers the content of these hosted websites to the end user's device.



HTTP Server: Introduction

- Lets have a look at the interaction between Server and Web Browser.

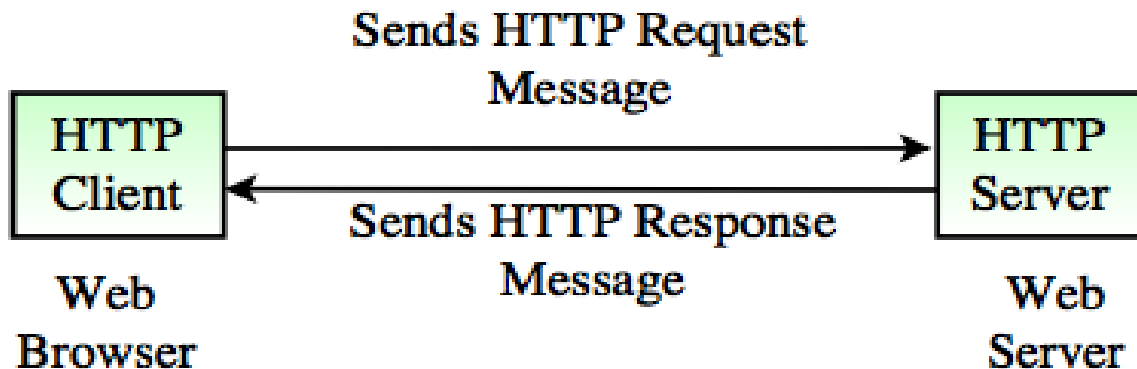


Fig. HTTP Protocol

Outline of client-server interaction

HTTP Client Request

- Client needs to connect to the server every time.
- When we want to connect to the server, we type some URL/Address of the website in the browser
- To display the page, browser fetches the file index.html from a web server.
- For example, www.example.com (Defaults: port 80, file index.html, http protocol).
- On entering link www.example.com in the web browser, the web browser re-constructs the URL/Address as:
<http://www.example.com:80>
- This reconstructed URL web-browsers send to the servers every time we navigate the internet pages.
- If the server is configured to certain default pages. Like, server has a default web page, it is displayed when we visit a website on that server.
- That web page is decided by the name of the file. Some servers have **public.html** and some will have **index.html**.

HTTP Client Request

- Lets consider **index.html** as default page.
- When browser Sends HTTP request to server, a HTTP header is sent.

Send HTTP Request - Write lines to socket

The diagram illustrates the structure of an HTTP request. It shows the following components:

- Method:** GET
- URL:** /index.html
- Protocol Version:** HTTP/1.1
- Header:** A bracket groups the following lines:
 - Host: www.example.com
 - User-Agent: Mozilla/5.0
 - Accept: text/html, */*
 - Accept-Language: en-us
 - Accept-Charset: ISO-8859-1,utf-8
 - Connection: keep-alive
- blank line:** A line separating the header from the body.
- Body (optional):** A bracket groups the following text:
 - For POST and PUT method*

- This HTTP Request is received at Server's end, which is processed by the server

HTTP Server Response

- The client sent some headers to HTTP server and expects same in-return.
- This is the HTTP response format the web-browser is expecting from Server

HTTP Response - Read lines from socket

The diagram illustrates the structure of an HTTP response. It shows a sequence of lines: the status line, followed by several header lines, a blank line, and then the body content. Red arrows and brackets are used to label parts of the response. Three arrows at the top point to 'HTTP/1.1', '200', and 'OK' in the status line, labeled 'Version', 'Status', and 'Status Message' respectively. A bracket on the left groups the status line and the four header lines under the label 'Header'. Another bracket on the left groups the HTML tags under the label 'Body'. The text '*Your server name*' is highlighted in orange.

```
Version Status Status Message
  ↓      ↓      ↓
HTTP/1.1 200 OK
Date: Fri, 16 Mar 2018 17:36:27 GMT
Server: *Your server name*
Content-Type: text/html; charset=UTF-8
Content-Length: 1846
blank line
<?xml ... >
<!DOCTYPE html ... >
<html ... >
...
</html>
```

HTTP Server Using Socket Programming

Step 1: Creating socket file descriptor

```
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)  
{  
    perror("In socket");  
    exit(EXIT_FAILURE);  
}
```

```
// This is a TCP socket
```

HTTP Server Using Socket Programming

Step 2: Create a struct `sockaddr_in` address;

#define PORT 80 //define with header files

address.sin_family = AF_INET;

address.sin_addr.s_addr = INADDR_ANY;

address.sin_port = htons(PORT);

HTTP Server Using Socket Programming

Step 3: Bind *server_fd* and *address*

```
if (bind(server_fd, (struct sockaddr *)&address,  
sizeof(address) ) < 0)  
{  
    perror("In bind");  
    exit(EXIT_FAILURE);  
}
```

HTTP Server Using Socket Programming

Step 4: Listen on *server_fd*

```
if (listen(server_fd, 10) < 0)  
{  
    perror("In listen");  
    exit(EXIT_FAILURE);  
}
```

- Here 10 is **queue_limit**, i.e. number of active participants that can “wait” for a connection

HTTP Server Using Socket Programming

Step 5: Accept client request using *accept()*

```
int addrlen = sizeof(address);
```

```
while(1) // infinite while loop for accepting request of different clients  
{
```

```
if ((new_socket = accept(server_fd, (struct sockaddr *)&address,  
(socklen_t*)&addrlen))<0)
```

```
{  
    perror("In accept");  
    exit(EXIT_FAILURE);  
}
```

```
} //end while(1)
```

- Note that “*new_socket*” will be used for further communication

HTTP Server Using Socket Programming

Step 6: Read and Print data received from client

```
while(1)
{
if ((new_socket = accept(server_fd,.....
.....

char buffer[30000] = {0};
valread = read( new_socket , buffer, 30000);
printf("%s\n",buffer );

//end while(1)
```

HTTP Server Using Socket Programming

Step 7: Create HTTP Response in a String

```
char *server_response = "HTTP/1.1 200 OK\nContent-Type:  
text/html\nContent-Length: 72\n\n<!DOCTYPE  
HTML><HTML><body><h1>Hello Thapar  
Institute</h1></body></HTML>";  
  
while(1)  
{  
  
.....  
  
}
```

- Here, we are sending HTTP response header appended with HTML content <html><body><h1>Hello Thapar Institute</h2></body></html>

HTTP Server Using Socket Programming

Step 7: Send the Response to client and close new_socket

```
char *server_response = "HTTP/1.1 200 OK\nContent-Type:  
text/html\nContent-Length: 72\n\n <!DOCTYPE  
HTML><HTML><body><h1>Hello Thapar Institute  
</h1></body></HTML>";
```

```
while(1)
```

```
{
```

```
.....accept().....
```

```
.....print client data....
```

```
write(new_socket , server_response , strlen(server_response));
```

```
printf("-----Response sent to client-----\n");
```

```
close(new_socket);
```

```
//end while(1)
```