Lecture Notes for Lecture 13 of CS 5600 (Computer Systems) for the Fall 2019 session at the Northeastern University Silicon Valley Campus.

System Services

Philip Gust,
Clinical Instructor
Department of Computer Science

These slide highlight content from suggested readings.

Review of Lecture 12

- In lecture 12 we learned about some of the ways that processes can communicate with each other and the mechanisms available for *inter-process communication*.
- We considered two situations: when the processes are running on the same computer, and when the processes are running on different computers.
- Three mechanisms are available when running on the same computer: *signals, shared memory* and *named pipes*.
- When began looking at how processes running on different computers connected by a network can communicate using *ports* and *sockets*, and through *remote procedure calls* (*RPCs*).

System Services

- In this lecture we will continue discussing how to use sockets to communicate between processes on different computers connected by a network,
- We will also briefly discuss the use of *remote procedure calls* (*RPCs*) and the facilities available to implement them in different programming languages.
- Next we will look at the idea of services and how they are implemented using sockets and standardized ports.
- In particular we will discuss web servers (HTTP). In the next lecture, we will consider domain name service (DNS).

System Services

Remote System Services

- Two means of inter-process communication among processes on different computers:
  - *Socket* is one endpoint of a two-way communication link between two processes running on the network. It is bound to a port number so that the networking layer can identify the process that data will be sent to.
  - *Remote procedure call* (*RPC*) is a protocol that one program can use to request a service from a process running in another computer on a network by calling a procedure of the remote process in a way similar to a local procedure call.

System Services

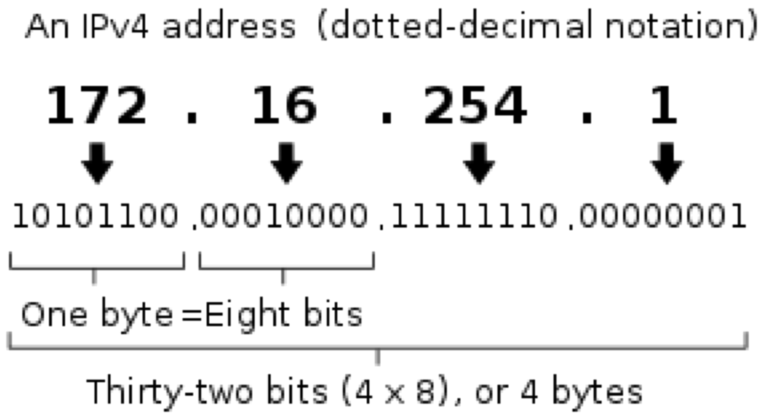
Sockets

- A socket is one endpoint of a two-way communication link between two processes on the network.
- An endpoint is a combination of an *IP address* and a *port number*. Every connection can be uniquely identified by its two endpoints.
- A server runs on a specific computer and listens to the socket for a client running on another server to make a connection request.

System Services

Sockets

- An *IP address* serves two principal functions.
  - It identifies the host, or more specifically its network interface.
  - It provides the location of the host in the network, and thus the capability of establishing a path to that host.


An IPv4 address (dotted-decimal notation)
172 . 16 . 254 . 1
10101100 .00010000 .11111110 .00000001
One byte =Eight bits
Thirty-two bits (4 x 8), or 4 bytes

System Services

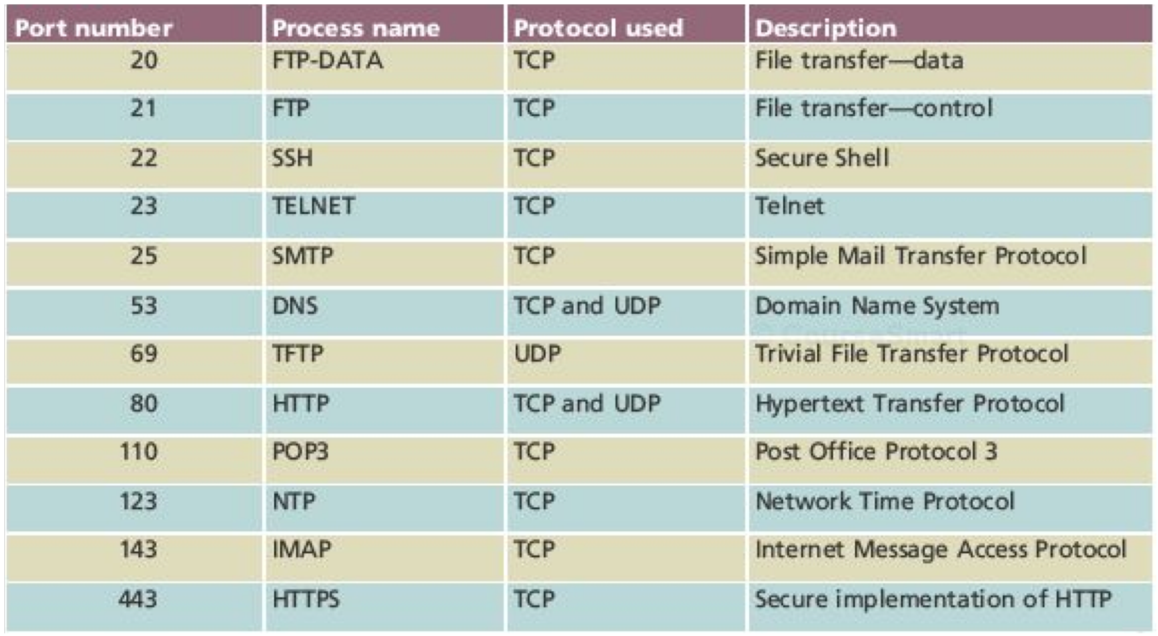
Sockets

- A network *port* identifies one side of a connection between two computers. As network addresses are like street address, port numbers are like room numbers.
- Computers use port numbers to determine to which process a message should be delivered. An application “listens” on a port for incoming data.
- The process of associating a port number with a service is known as “binding.” The operating systems delivers the data on a port to the process bound to that port.

System Services

Sockets

- Commonly available services like email or a web server use standardized port numbers. Here are some well-known ports for some of these services.

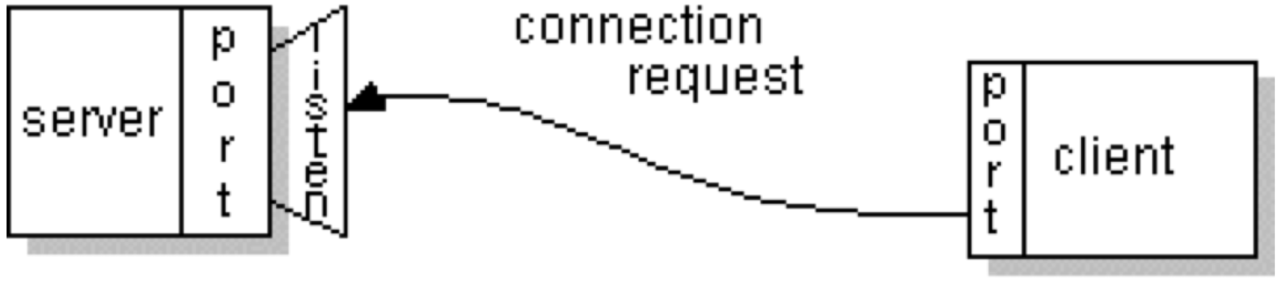


Port number	Process name	Protocol used	Description
20	FTP-DATA	TCP	File transfer—data
21	FTP	TCP	File transfer—control
22	SSH	TCP	Secure Shell
23	TELNET	TCP	Telnet
25	SMTP	TCP	Simple Mail Transfer Protocol
53	DNS	TCP and UDP	Domain Name System
69	TFTP	UDP	Trivial File Transfer Protocol
80	HTTP	TCP and UDP	Hypertext Transfer Protocol
110	POP3	TCP	Post Office Protocol 3
123	NTP	TCP	Network Time Protocol
143	IMAP	TCP	Internet Message Access Protocol
443	HTTPS	TCP	Secure implementation of HTTP

System Services

Sockets

- A client requests a connection using the IP address of the server and the port number on which the server is listening.
- The client identifies itself to the server by binding to a local port number that it uses during this connection (usually assigned by the system).



System Services

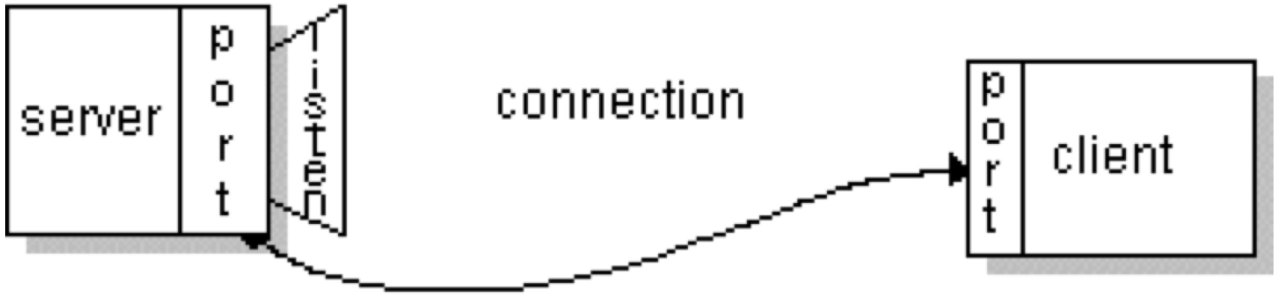
Sockets

- If all goes well, the server accepts the connection.
- Upon acceptance, the server gets a new socket bound to the same local address and port, and also has its remote endpoint set to the address and port of the client.
- It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

System Services

Sockets

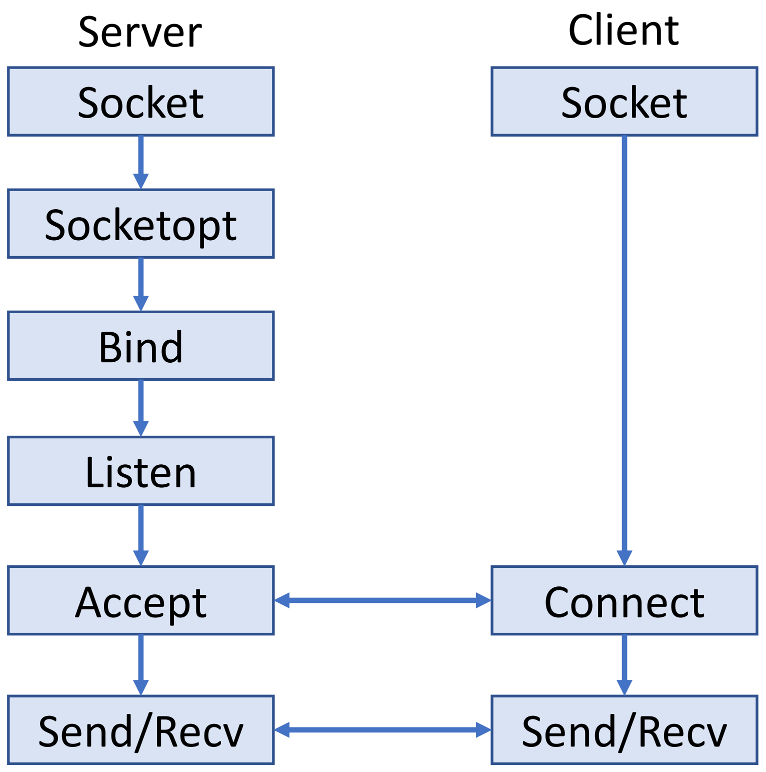
- On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.
- The client and server can now communicate by writing to or reading from their sockets



System Services

Sockets

- Here is a state diagram for server and client



System Services

Example: Simple DateServer and DateClient

```
#import  <stdbool.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 9091
#define MAXBUF 128

int main() {
    int listen_sock_fd = newListenSocket(PORT);
    if (listen_sock_fd == 0) {
            perror("newListenSocket");
            exit(EXIT_FAILURE);
    }
    fprintf(stderr, "listen: port %d\n", PORT);
```

System Services

Example: Simple DateServer and DateClient

```
while (true) {
     // accept client connection
    int peer_sock_fd = acceptPeerConnection(listen_sock_fd);
    fprintf(stderr, "connection?\n");
    if (peer_sock_fd < 0) {
        perror("accept");
        continue;
    }

    char host[MAXBUF];
    int port;
    if (getLocalHostAndPort(peer_sock_fd, host, &port) == 0) {
        fprintf(stderr, "  local: %s:%d\n", host, port);  // show local host:port
    }

    if (getPeerHostAndPort(peer_sock_fd, host, &port) == 0) {
        fprintf(stderr, "  peer:  %s:%d\n", host, port);  // show peer host:port
    }
```

System Services

Example: Simple DateServer and DateClient

```
            // format current time
            time_t timer;
            time(&timer);
            struct tm* tm_info = localtime(&timer);
            char buf[MAXBUF];
            strftime(buf, MAXBUF, "%Y-%m-%d %H:%M:%S", tm_info);

            // send time
            if (write(peer_sock_fd , buf , strlen(buf)+1) < 0) {
                perror("send");
            }
            fprintf(stderr, "  sent:  %s\n", buf);

            // close socket to complete write operation
            close(peer_sock_fd);
        }

        // close server socket
        close(listen_sock_fd);
        return 0;
    }
```

System Services

Example: Simple DateServer and DateClient

```
            // send time
            if (write(socket_fd , buf , strlen(buf)+1) < 0) {
                perror("send");
            }
            // close socket to complete write operation
            close(socket_fd);
        }

        // close listener socket
        close(listen_sock_fd);
        return 0;
    }
```

System Services

Example: Simple DateServer and DateClient

```
/**
 * Create new listen socket on specified port
 *
 * @param port the port
 * @return listen socket or 0 if creation, bind, or listen failed
 */
int newListenSocket(int port) {
    // Creating internet socket stream file descriptor
    int listen_sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_sock_fd == 0) {
        return 0;
    }

    // SO_REUSEADDR prevents the "address already in use" errors when testing servers.
    int optval = 1;
    if (setsockopt(listen_sock_fd, SOL_SOCKET, SO_REUSEADDR, &optval , sizeof(int)) < 0) {
        close(listen_sock_fd);
        return 0;
    }
```

System Services

Example: Simple DateServer and DateClient

```
// listener address and port
struct sockaddr_in listen_addr;
socklen_t listen_size = sizeof(listen_addr);
memset(&listen_addr, 0, listen_size);
listen_addr.sin_family = AF_INET;  // address from internet
listen_addr.sin_port = htons(PORT);   // port in network byte order
listen_addr.sin_addr.s_addr = INADDR_ANY;  // bind to any address

// bind host address and port
if (bind(listen_sock_fd, (struct sockaddr *)&listen_addr, listen_size) < 0) {
        close(listen_sock_fd);
        return 0;
}
```

System Services

Example: Simple DateServer and DateClient

```
        // set up queue for clients connections up to default
        // maximum pending socket connections (usually 128)
        if (listen(listen_sock_fd, SOMAXCONN) < 0) {
                close(listen_sock_fd);
                return 0;
        }

    return listen_sock_fd;
}
```

System Services

Example: Simple DateServer and DateClient

```
/**
 * Accept new peer connection on a listen socket.
 *
 * @param listen_sock_fd the listen socket
 * @return the peer socket fd
 */
int acceptPeerConnection(int listen_sock_fd) {
    while (true) {
        struct sockaddr_in peer_addr;
        socklen_t peer_size = sizeof(peer_addr);
        int peer_sock_fd =
            accept(listen_sock_fd, (struct sockaddr *)&peer_addr, &peer_size);
        if (peer_sock_fd > 0) {
            return peer_sock_fd;
        }
    }
}
```

System Services

Example: Simple DateServer and DateClient

```
/**
 * Get the local host and port for a socket.
 *
 * @param sock_fd the socket
 * @param addr_str buffer for IP address string
 * @param port pointer for port value
 * @return 0 if successful
*/
int getLocalHostAndPort(int sock_fd, char *addr_str, int *port) {
    struct sockaddr_in addr;
    socklen_t size = sizeof(addr);

    int status = getsockname(sock_fd, (struct sockaddr *)&addr, &size);
   if (status == 0) {
    *port = ntohs(addr.sin_port);
    strcpy(addr_str, inet_ntoa(addr.sin_addr));
   }
   return status;
}
```

System Services

Example: Simple DateServer and DateClient

```
/**
 * Get the peer host and port for a socket.
 *
 * @param sock_fd the socket
 * @param addr_str buffer for IP address string
 * @param port pointer for port value
 * @return 0 if successful
 */
int getPeerHostAndPort(int sock_fd, char *addr_str, int *port) {
    struct sockaddr_in addr;
    socklen_t size = sizeof(addr);

    int status = getpeername(sock_fd, (struct sockaddr *)&addr, &size);
    if (status == 0) {
        *port = ntohs(addr.sin_port);
        strcpy(addr_str, inet_ntoa(addr.sin_addr));
    }
    return status;
}
```

System Services

Example: Simple DateServer and DateClient

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

#define PORT 9091
#define MAXBUF 128

int main(int argc, char const *argv[]) {
    printf("Enter IP Address of the date service on port %d:", PORT);
    char host_str[MAXBUF];
    if (fgets(host_str, MAXBUF, stdin) == NULL) {
        perror("fgets");
        return EXIT_FAILURE;
    }
```

System Services

Example: Simple DateServer and DateClient

```
// trim newline
size_t len = strlen(host_str);
if (len > 0 && host_str[len-1] == '\n') host_str[len-1] = 0;
else if (len > 1 && host_str[len-2] == '\r') host_str[len-2] = 0;

// connect to server host and port
int server_sock_fd = connectPeerHostAndPort(host_str, PORT);

char host[MAXBUF];
int port;
if (getLocalHostAndPort(server_sock_fd, host, &port) == 0) {
        fprintf(stderr, "  local: %s:%d\n", host, port); // show local host:port
}

if (getPeerHostAndPort(server_sock_fd, host, &port) == 0) {
        fprintf(stderr, "  peer:  %s:%d\n", host, port); // show peer host:port
}
```

System Services

Example: Simple DateServer and DateClient

```
    char buf[MAXBUF];
    if (read(server_sock_fd , buf, MAXBUF) < 0) {
        perror("read");
    } else {
        fprintf(stderr, "  time:  %s\n", buf);
    }

    close(server_sock_fd);
    return EXIT_SUCCESS;
}
```

System Services

Example: Simple DateServer and DateClient

```
/**
 * Connect to peer at host and port.
 *
 * @param host the host IP address as a string
 * @param port the host port
 * @return the peer socket fd
 */
int connectPeerHostAndPort(const char *host, int port) {
    int sock_fd = 0;
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        return -1;
    }

    struct sockaddr_in host_addr;
    const socklen_t addrlen = sizeof(host_addr);
    memset(&host_addr, 0, addrlen);
    host_addr.sin_family = AF_INET;  // address from internet
    host_addr.sin_port = htons(port);   // port in network byte order
```

System Services

Example: Simple DateServer and DateClient

```
    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, host, &host_addr.sin_addr) <= 0) {
            close(sock_fd);
            return -1;
    }

    // connect to server
    if (connect(sock_fd, (struct sockaddr *)&host_addr, addrlen) != 0) {
            close(sock_fd);
            return -1;
    }

    return sock_fd;
}
```

System Services

Web Server Basics

- A web server is part of a client-server based algorithm that provides access to data that is shared among a group of web browser clients.
- A distributed algorithm enables a web client to send data and requests to the server, and the server to provide access to content stored on the server.

System Services

Web Server Basics

- A web server is a server-based content repository that provides access to static and dynamic content to one or more clients applications.
- Static content is stored within the web server and made available through a fixed request. Dynamic content is generated by the server based on parameters of a query.
- The web server also provides ways for clients to store new information on the server, and to modify existing information.

System Services

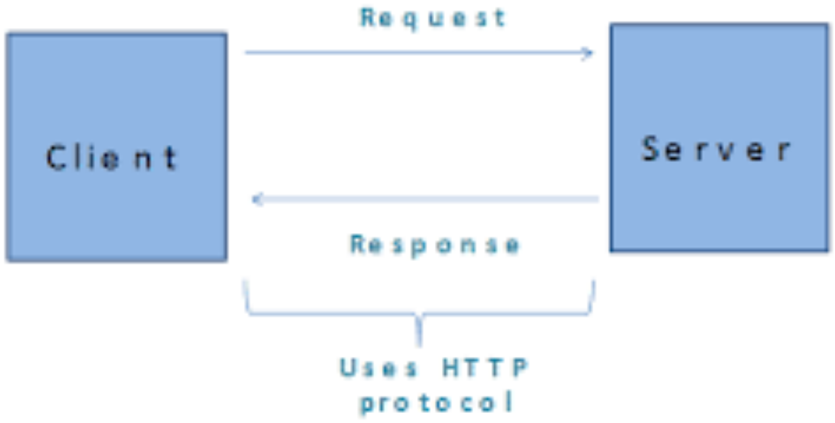
Web Server Basics

- In a typical scenario, the web client initiates communication with the web server by sending a request. The web server responds by sending back the requested information.
- Communication between the client and server follow a protocol known as HyperText Transfer Protocol (HTTP).
- HTTP is a *stateless* protocol that specifies a set of operations, known as *methods*, and required and optional metadata that are exchanged between the client and server.
- With a stateless protocol, the client and server are only connected during a single *request* and *response*. Various techniques are used to maintain *session* state information.

System Services

Web Server Basics

- HTTP protocol is based on a TCP/IP communication channel to transport the client request and server response.



System Services

Web Server Request

- A request sent to the web server uses an addressing scheme known as a *Uniform Resource Locator* (*URL*) that includes the scheme (http), server address and port, and a path.
- An HTTP URL conforms to the syntax of a generic *Uniform Resource Indicator* (*URI*) :

  URI = scheme:[//authority]path[?query][#fragment]
- The authority component has three sub-components:

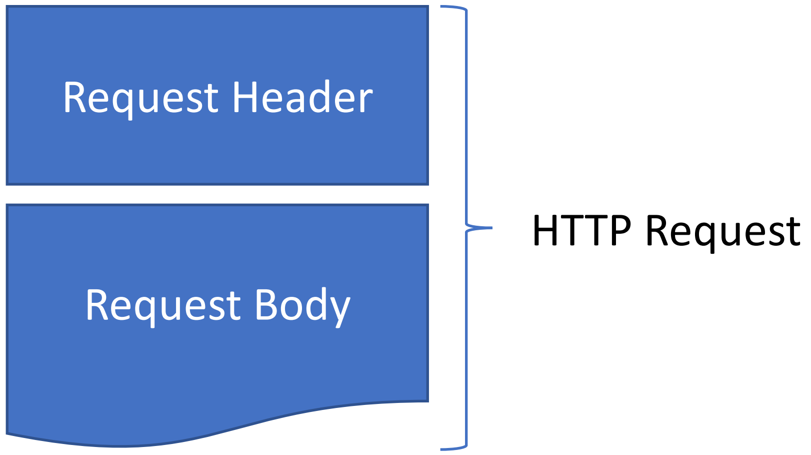
  authority = [userinfo@]host[:port]
- Here is the URL for the instructor’s home page. Port 80 is the default port for HTTP.

  http://ccis.northeastern.edu:80/home/pgust/index.html

System Services

Web Server Request

- The request contains the request header and a request body with the data (if any) for the request. The request header is separated from the request body by a blank line



System Services

Web Server Request

- The request header is a series of text lines. The first line specifies the *method* or operation to perform, the path, and the protocol version being used.
- Other lines provide request metadata in the form of property names and values. The request header ends with an empty line.

```
GET /home/pgust/index.html HTTP/1.1
Host: ccis.neu.edu:80
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15
(KHTML, like Gecko) Version/12.1 Safari/605.1.15
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

System Services

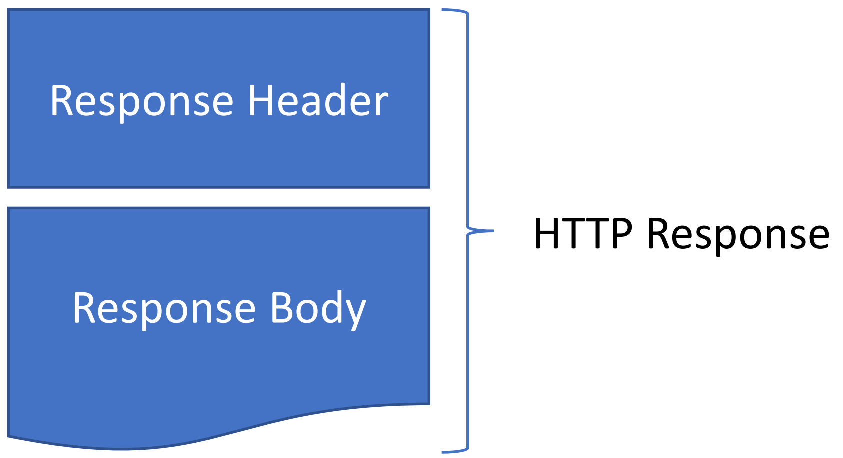
Web Server Request

- There are five standard methods supported by a web server. Some servers support additional methods.
  - **GET**: fetch an existing resource. The URL contains all the necessary information the server needs to locate and return the resource.
  - **HEAD**: like GET but returns only response parameters, not content.
  - **PUT**: update an existing resource. The payload may contain the updated data for the resource.
  - **POST**: create a new resource. POST requests usually carry a payload that specifies the data (e.g. names and values) for the new resource; commonly used to send form data.
  - **DELETE**: delete an existing resource.
- Only PUT and POST have request bodies, and generally must include a Content-Length request parameter.

System Services

Web Server Response

- The web server responds to the request with a response, which includes a response header and a response body



System Services

Web Server Response

- The response header is a series of text lines. The first line specifies the protocol version being used, and a response code and response text.
- Other lines provide response metadata in the form of property names and values. The response header ends with an empty line.

```
HTTP/1.1 200 OK
Server: Tiny Http Server
Date: Mon, 15 Apr 2019 14:26:22 GMT
Content-Length: 826
Last-Modified: Mon, 15 Apr 2019 14:26:22 GMT
Content-type: text/html
```

- The GET method returns a response body if there are no errors. Other methods *may* return response bodies.

System Services

Web Server Response

- The response codes fall into the following categories
  - Informational (1xx). This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line (e.g. 100 “Continue”).
  - Successful (2xx). This class of status code indicates that the client request was successfully received, understood, and accepted. (e.g 200 “Ok”)
  - Redirection (3xx). This class of status code indicates that further action needs to be taken by the user agent to fulfill the request. (e.g. 301 ”Moved Permanently”)
  - Client Error (4xx). This class of status code is intended for cases in which the client seems to have erred (e.g. 404 “Not Found”)
  - Server Error (5xx). This class of status code indicate cases in which the server is aware that it has erred or is incapable of performing the request (e.g. 501 “Not Implemented”).

System Services

Web Server Response

- The type of content returned is described by a *media type*. This indicates the *type* and the *subtype* of the content.
- Web servers sometimes determine the MIME type based on a file extension, but can also “sniff” the file to determine its formats.
- Common MIME types include
  - text/plain, text/html, text/rtf, text/css
  - image/jpeg, image/png, image/gif
  - audio/x-aiff, audio/x-wav
  - video/mp4, video/mpeg
  - application/sql, application/pdf, application/msword