# 456: Network-Centric Programming

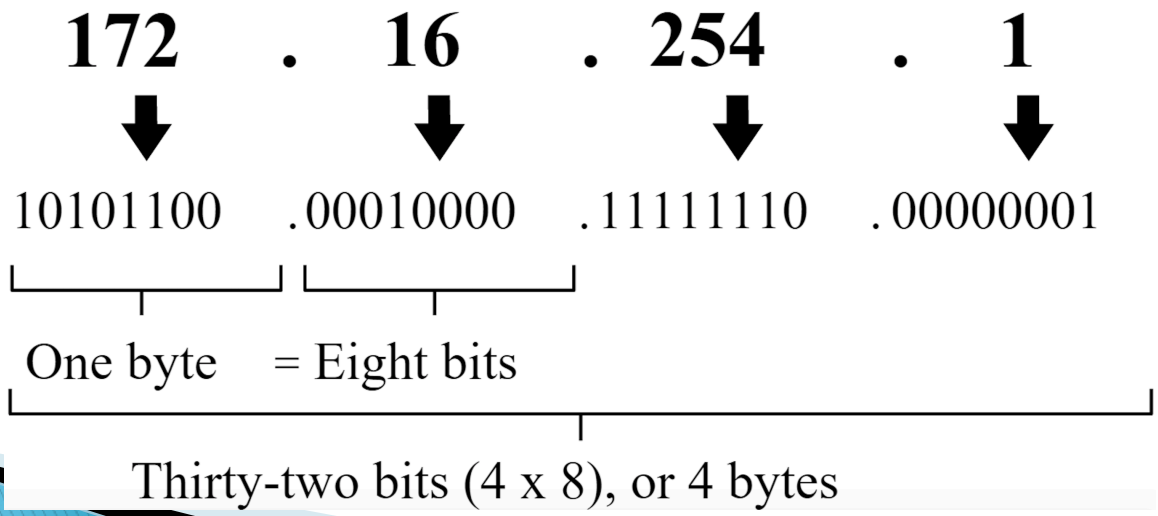
Yingying (Jennifer) Chen

Web: https://www.winlab.rutgers.edu/~yychen/
Email: yingche@scarletmail.rutgers.edu

Department of Electrical and Computer Engineering
Rutgers University

# Address Conversion

- Convert string representation (“172.16.254.1”) into numeric representation
- “String Presentation” to/from “Numeric Presentation”
  - int inet_pton(family, strptr, addrptr)
  - char* inet_ntop(family, addrptr, strptr, len)



# Address Conversion

| |
|---|
| `#include <arpa/inet.h>` |
| `int inet_pton(int`*family*`,const char *`*strptr*`,void *`*addrptr*`);` |
| Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error |
| `const char *inet_ntop(int`*family*`,const void *`*addrptr*`,char *`*strptr*`,size_t`*len*`);` |
| Returns: pointer to result if OK, `NULL` on error |

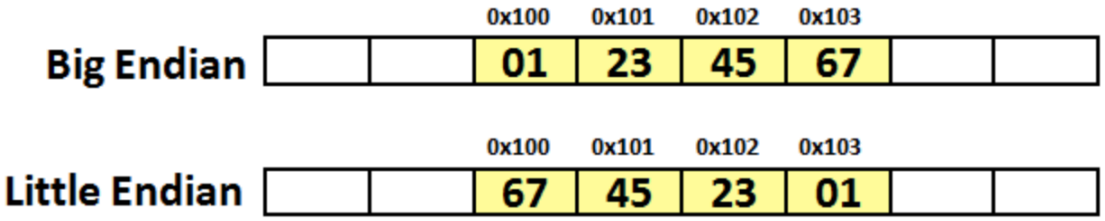
The family could be `AF_INET`.

For instance:

*struct sockaddr_in serv_addr;*
*inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);*

# Byte Ordering

- Internet hosts can have different host byte orders (e.g., little-endian, big-endian), thus TCP/IP needs to define a uniform *network byte order* (big-endian byte order) to make any data item (e.g., IP address) being carried across the network in a packet header.
- IP: '1.35.69.103' → 0x01234567
  - **Network byte order**: The defined uniform *byte order used for network* (big-endian byte order) 0x01234567
  - **Host byte order**: depends on architecture of host (e.g., little-endian, big-endian); little-endian: 0x67452301




Example: How 0x1234567 is stored at memory location 0x100-ox103

**Big endian**: Most significant byte (MSB) is stored at the starting address in memory

**Little endian**: Least significant byte (LSB) is stored at the starting address in memory

# Byte Ordering

The argument of these functions is numerical value, which can be decimal, hex-decimal or binary number, etc.

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
                                        Returns: value in network byte order

unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
                                        Returns: value in host byte order
```

- short 16-bit, usually used for port number conversion
- long 32-bit, usually used for IP address conversion

Example:

```
struct sockaddr_in servaddr;                          // The obtained big endian is
servaddr.sin_addr.s_addr = htonl(0x67452301);         0x01234567
address.sin_port = htons(80); // or htons(0x0050)
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
//INADDR_ANY is used when you don't need to bind a socket to a specific IP.
//INADDR_ANY: 0x00000000
```

# Example (Request from Client and Ack back from Server): lecture_6_code/requestfromclient/server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define PORT 8080                    // no need to use root permission
int main(int argc, char const *argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;      //socket address structure
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    char *ack = "ACK from server";   //acknowledgement message
```

# Example (Request from Client and Ack back from Server): lecture_6_code/requestfromclient/server.c

```
// Creating socket file descriptor
  if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
  {
      perror("socket failed");
      exit(EXIT_FAILURE);
  }

  address.sin_family = AF_INET;
  address.sin_addr.s_addr = htonl(INADDR_ANY); //host byte-order to network byte-order
  address.sin_port = htons( PORT ); //host byte-order to network byte-order

  // bind socket address to socket file descriptor
  if (bind(server_fd, (struct sockaddr *)&address,
                          sizeof(address))<0)
  {
      perror("bind failed");
      exit(EXIT_FAILURE);
  }
```

# Example (Request from Client and Ack back from Server): lecture_6_code/requestfromclient/server.c

```
if (listen(server_fd, 1024) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                    (socklen_t*)&addrlen))<0)
    {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    valread = read( new_socket, buffer, 1024);//receive the msg sent from client
    printf("%s\n",buffer );
    write(new_socket, ack, strlen(ack)); //send msg
    printf("ACK message sent\n");
    return 0;
}
```

# Example (Request from Client and Ack back from Server): lecture_6_code/requestfromclient/client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
 #include <arpa/inet.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *request = "Request from client"; //request msg to be sent
    char buffer[1024] = {0};
```

# Example (Request from Client and Ack back from Server): lecture_6_code/requestfromclient/client.c

```
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr)); //initilize serv_addr

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IP addresses from string presentation to numeric presentation
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
```

# Example (Request from Client and Ack back from Server): lecture_6_code/requestfromclient/client.c

```
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }
    write(sock, request, strlen(request)); //send request msg
    printf("Request message sent\n");
    valread = read(sock, buffer, 1024); //receive the msg sent from server
    printf("%s\n",buffer );
    return 0;
}
```

Terminal 1
$ ./server

Terminal 2
$ ./client

# Example (Hello from Server): lecture_6_code/hellofromserver/server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    char *hello = "Hello from server";
```

# Example (Hello from Server): lecture_6_code/hellofromserver/server.c

```
// Creating socket file descriptor
  if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
  {
      perror("socket failed");
      exit(EXIT_FAILURE);
  }

  address.sin_family = AF_INET;
  address.sin_addr.s_addr = htonl(INADDR_ANY);
  address.sin_port = htons( PORT );

  // bind socket address to socket file descriptor
  if (bind(server_fd, (struct sockaddr *)&address,
                          sizeof(address))<0)
  {
      perror("bind failed");
      exit(EXIT_FAILURE);
  }
```

# Example (Hello from Server): lecture_6_code/hellofromserver/server.c

```
if (listen(server_fd, 1024) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                    (socklen_t*)&addrlen))<0)
    {
        perror("accept");
        exit(EXIT_FAILURE);
    }

    write(new_socket, hello, strlen(hello)); //send hello msg
    printf("Hello message sent\n");
    valread = read( new_socket, buffer, 1024); //receive the msg from client
    printf("%s\n", buffer);
    return 0;
}
```

# Example (Hello from Server): lecture_6_code/hellofromserver/client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
 #include <arpa/inet.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *response = ”Response from client";
    char buffer[1024] = {0};
```

# Example (Hello from Server): lecture_6_code/hellofromserver/client.c

```
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr)); // initialize serv_addr

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IP addresses from string presentation to numeric presentation
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
```

# Example (Hello from Server): lecture_6_code/hellofromserver/client.c

```
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }

    valread = read(sock, buffer, 1024); //receive msg from server
    printf("%s\n", buffer);

    write(sock, response, strlen(response)); //send msg from client
    printf("Response message sent\n");
    return 0;
}
```

Terminal 1
$ ./server

Terminal 2
$ ./client

# Example (Multiple Clients): lecture_6_code/multipleclient/server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    int pid;
    char *response = “Response from server";
```

# Example (Multiple Clients): lecture_6_code/multipleclient/server.c

```
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = htonl(INADDR_ANY);
    address.sin_port = htons( PORT );
    // bind socket address to the socket file descriptor
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))<0){
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 1024) < 0){
        perror("listen");
        exit(EXIT_FAILURE);
    }
```

# Example (Multiple Clients): lecture_6_code/multipleclient/server.c

```
//sequentially use accept() to build the connection
while(1){
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                        (socklen_t*)&addrlen))<0){
            perror("accept");
            exit(EXIT_FAILURE);
         }
        valread = read( new_socket , buffer, 1024); //receive msg from client
        printf("%s\n",buffer );
        write(new_socket, response, strlen(response)); //response from server
        printf("Response msg sent\n");
        close(new_socket);
    }
}
```

# Example (Multiple Clients): lecture_6_code/multipleclient/client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
 #include <arpa/inet.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sock1, sock2, valread;
    struct sockaddr_in serv_addr;
    char *hello1 = "Hello from client 1";
    char *hello2 = "Hello from client 2";
    char buffer[1024] = {0};
```

# Example (Multiple Clients): lecture_6_code/multipleclient/client.c

```
if ((sock1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }
    if ((sock2 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }
    // initialize serv_addr
    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
```

# Example (Multiple Clients): lecture_6_code/multipleclient/client.c

```
if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)  {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
    if (connect(sock1, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
   // send messages via two sockets: sock1
    write(sock1 , hello1 , strlen(hello1)); //send msg via sock1
    printf("Hello message sent from client 1\n");
    valread = read(sock1, buffer1, 1024); //receive msg from server
    printf("%s\n", buffer1);
```

# Example (Multiple Clients): lecture_6_code/multipleclient/client.c

```
    if (connect(sock2, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
        printf("\nConnection Failed \n");
        return -1;
    }
    // send messages via two sockets: sock2
    write(sock2 , hello2 , strlen(hello2)); //send msg via sock2
    printf("Hello message sent from client 2\n");
    valread = read(sock2, buffer2, 1024); //receive msg from server
    printf("%s\n", buffer2);
    return 0;
}
```

Terminal 1
$ ./server

Terminal 2
$ ./client

# Reading/Writing to socket: Subtle Buffering Differences from Files

- Reading/writing to socket may return lower byte count than requested
  - e.g., ask to read 10 bytes, but only read 9 bytes
- *Reason: temporary* buffer empty/full condition
  - Need to call read/write again to solve this problem

# More Robust Function: *readnbytes()*

```
/* Make sure reading "n" bytes from a descriptor. */
int i = 1;
Int readnbytes(int fd, void *vptr, int n) { //n is the number of bytes to be read
    int nleft; //number of bytes left to be read
    int nread; //number of bytes have been read
    char    *ptr;
    ptr = vptr;
    nleft = n;
    while (nleft > 0) {

  // nread is the actual number of bytes that have been read by calling read()

        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR){     /* Interrupted by signal handler*/
                nread = 0;     /* and call read() again */
                printf("Read Failure\n");
            }
            else
                return(-1);
         }
```

# More Robust Function: *readnbytes()*

```
else if (nread == 0)
            break;              /* EOF */
        printf("The %dth read has actually read %d bytes. \n", i, nread);
        nleft = nleft - nread;
        ptr   = ptr - nread;
        i = i + 1;
    }
    return(n - nleft);      /* return >= 0 */
}
```

# Example (readnbytes): lecture_6_code/readnbytes/server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
```

# Example (readnbytes): lecture_6_code/readnbytes/server.c

```
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0){
        perror("socket failed");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = htonl (INADDR_ANY);
    address.sin_port = htons( PORT );

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

# Example (readnbytes): lecture_6_code/readnbytes/server.c

```
if (listen(server_fd, 1024) < 0){
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen))<0){
        perror("accept");
        exit(EXIT_FAILURE);
    }
    //receive message using readnbytes() function
    FILE* fp=fopen("receive.txt","w+");
    while((valread = readnbytes( new_socket , buffer, 1024))>0){
        fwrite(buffer,1,sizeof(buffer),fp);//data will be stored in the file receive.txt
    }
    printf("readnbytes function is called\n");
    printf("Data received\n");
}
```

# Example (readnbytes): lecture_6_code/readnbytes/client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
 #include <arpa/inet.h>
#define PORT 8080

int main(int argc, char const *argv[]){
    struct sockaddr_in address;
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char buffer[1024] = {0};
```

# Example (readnbytes): lecture_6_code/readnbytes/client.c

```
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }
    // initilize serv_addr
    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0) {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
```

# Example (readnbytes): lecture_6_code/readnbytes/client.c

```
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
        printf("\nConnection Failed \n");
        return -1;
    }
 FILE* fp=fopen("data_1MB.txt","r"); //data to be sent, 1MB
 while((valread=fread(buffer, 1, sizeof(buffer), fp))>0) {
      if(write(sock,buffer,valread) < 0)
        return -1;
    }
    printf("Data sent\n");
    return 0;
}
```

Terminal 1
$ ./server

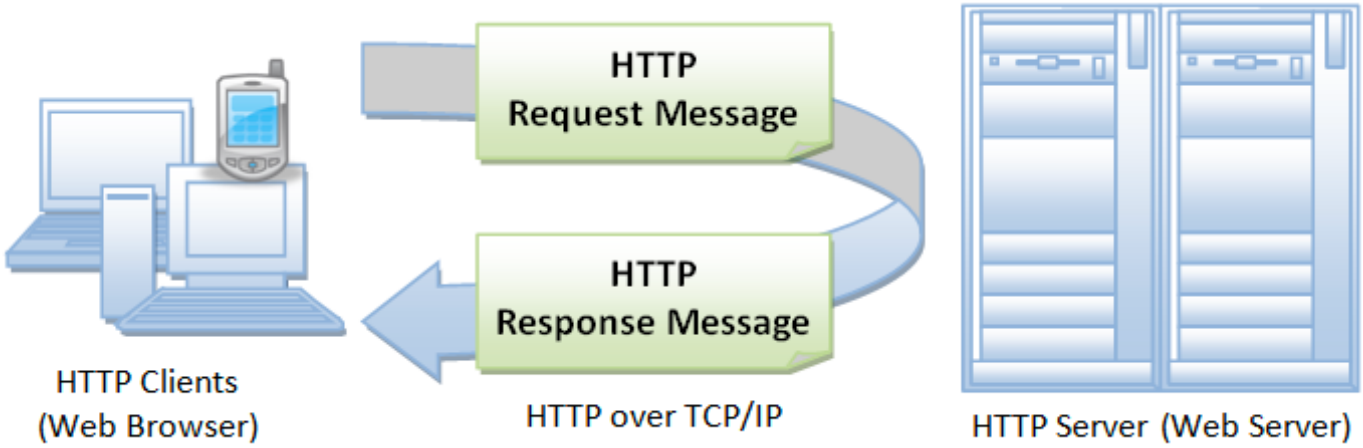
Terminal 2
$ ./client

```
The 957th read has actually read 1024 bytes.
The 958th read has actually read 1024 bytes.
The 959th read has actually read 1024 bytes.
The 960th read has actually read 1024 bytes.
The 961th read has actually read 1024 bytes.
The 962th read has actually read 1024 bytes.
The 963th read has actually read 1024 bytes.
The 964th read has actually read 1024 bytes.
The 965th read has actually read 1024 bytes.
The 966th read has actually read 1024 bytes.
The 967th read has actually read 1024 bytes.
The 968th read has actually read 1024 bytes.
The 969th read has actually read 1024 bytes.
The 970th read has actually read 1024 bytes.
The 971th read has actually read 1024 bytes.
The 972th read has actually read 1024 bytes.
The 973th read has actually read 1024 bytes.
The 974th read has actually read 1024 bytes.
The 975th read has actually read 1024 bytes.
The 976th read has actually read 1024 bytes.
The 977th read has actually read 576 bytes.
readnbytes function is called
Data received
```

# Parsing the HTTP Protocol

# The HTTP protocol

- HTTP is an *asymmetric request-response client-server* protocol.
  - An HTTP client sends a request message to an HTTP server.
  - The server returns a response message.



# The HTTP protocol

- Hypertext Transfer Protocol
  - Designed in 1990 by Tim Berners-Lee at CERN
- Example of ASCII-based protocols
  - The data is sent in a plain-text (ASCII) encoding
  - Others: FTP, SMTP, IMAP …

ASCII Code Table Example

| | | |
|---|---|---|
| ascii code | 65 | A |
| ascii code | 66 | B |
| ascii code | 67 | C |
| ascii code | 68 | D |
| ascii code | 48 | 0 |
| ascii code | 49 | 1 |
| ascii code | 50 | 2 |
| ascii code | 51 | 3 |
| ascii code | 52 | 4 |

# The HTTP protocol

- Stateless request/response protocol
  - Easy to implement
  - Robust against failures
  - Needs extensions (cookies) to allow implementation of advanced applications

Stateless
- No information is retained by the sender or receiver

# Uniform Resource Locator (URL)

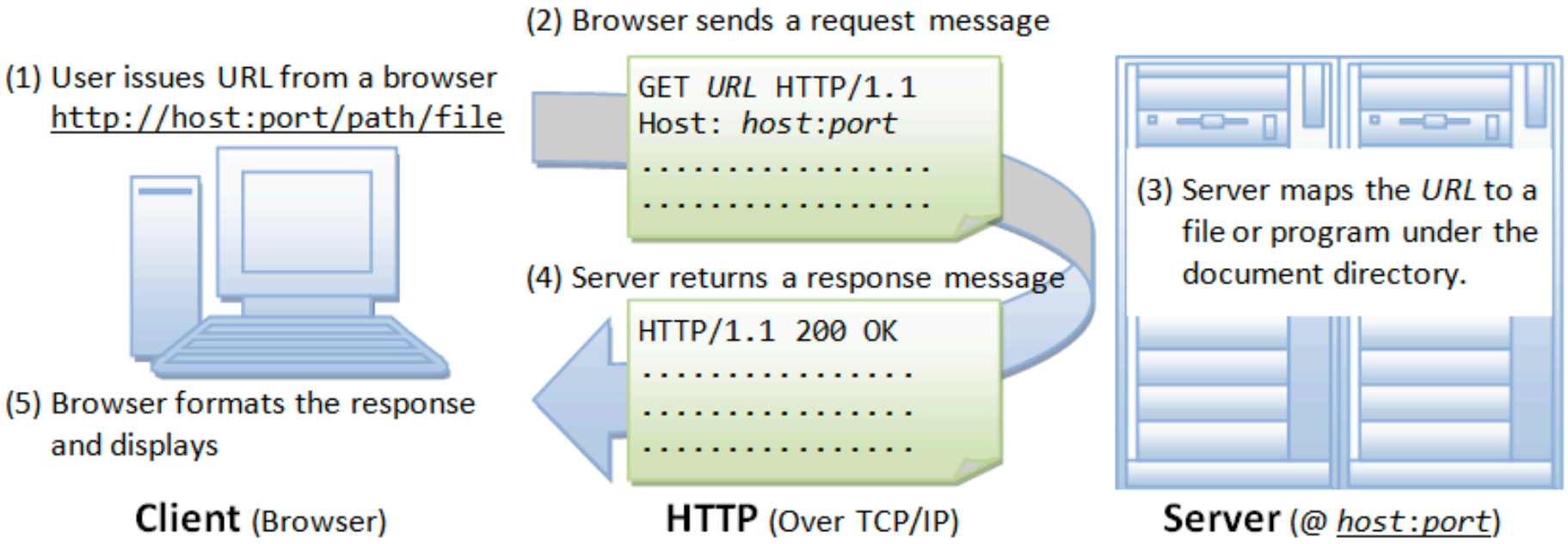
- HTTP protocol utilizes the URL to locate the HTTP resources on the internet
- Protocol://servername:port/path/filename
- String that (world-wide) uniquely identifies web resources
- Example
  - http://www.slashdot.org
  - http://www.rutgers.edu:80/test/index.html
    - The port number is optional and defaults to the well-known HTTP port 80.

# Uniform Resource Locator (URL)

- Protocol://servername:port/path/filename?arguments
- URLs for executable files can include program arguments after the file name.
  - A '?' character separates the file name from the arguments
  - Each argument is separated by an '&' character
- Example:
  http://finance.yahoo.com/quote/AAPL/history?filter=history&frequency=1d
- Apple stock prices provided by yahoo finance: the arguments filter=history and frequency=1d indicate that the user wants to check historical data and the required data frequency is 1day.

# Browser

- Whenever you issue a URL from your browser using HTTP, e.g. http://www.nowhere123.com/index.html, the browser turns the URL into a *request message* and sends it to the HTTP server.
- The HTTP server interprets the request message, and returns you an appropriate response message, which is either the resource you requested or an error message.



# Typical Webbrowser Steps

- http://www.cnn.com:80/test/index.html

1. Check protocol
2. Parse & resolve host name (//,:)
   gethostbyname() invokes DNS and obtain the server's IP address
3. Parse port number (:,/)
4. Open stream socket to host
5. Send HTTP request for path and filename

# Protocol Methods

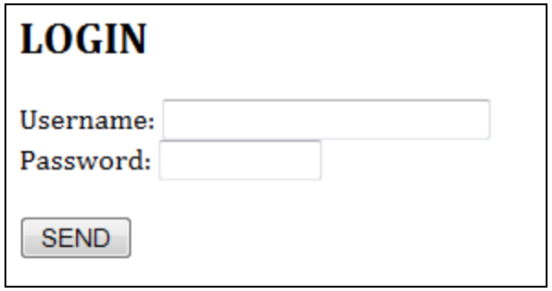
- GET
  - Retrieve content of URL (usually a file)
  - For example, the browser translated the URL http://www.nowhere123.com/docs/index.html into the following request message:

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
(blank line)
```

# Protocol Methods

- POST
  - Post information to URL
  - Example: gather the username and password in a login menu

Suppose the user enters "Peter Lee" as username and "123456" as password:

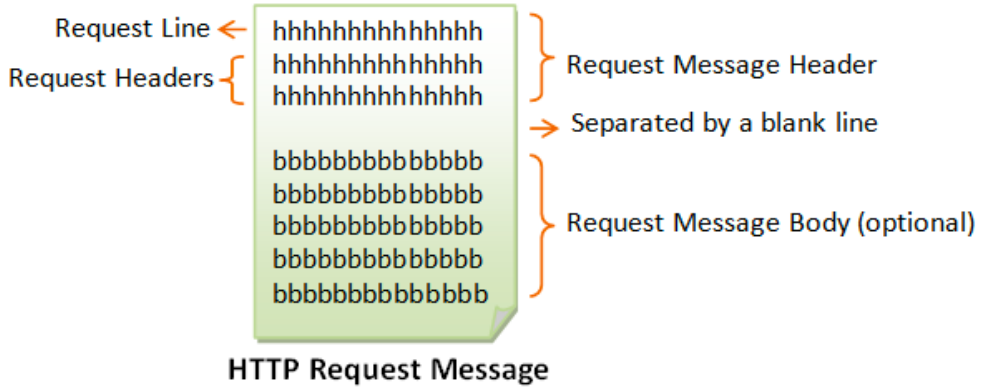


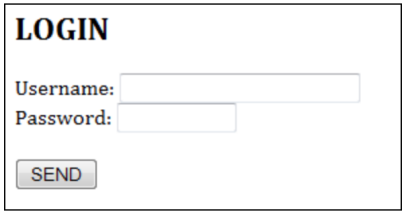

```
POST /bin/login HTTP/1.1
Host: 127.0.0.1:8000
Accept: image/gif, image/jpeg, */*
Referer: http://127.0.0.1:8000/login.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Content-Length: 37
Connection: Keep-Alive
Cache-Control: no-cache

User=Peter+Lee&pw=123456&action=login
```

# The Other HTTP Request Methods

- HEAD
  - Retrieve only header information
- PUT
  - Store data under URL
- DELETE
  - Delete data under URL
- …

# HTTP Request




Suppose the user enters "Peter Lee" as username and "123456" as password:

```
POST /bin/login HTTP/1.1
Host: 127.0.0.1:8000
Accept: image/gif, image/jpeg, */*
Referer: http://127.0.0.1:8000/login.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Content-Length: 37
Connection: Keep-Alive
Cache-Control: no-cache

User=Peter+Lee&pw=123456&action=login
```

Request Line

Request Headers

A blank line separates header & body

Request Message Body

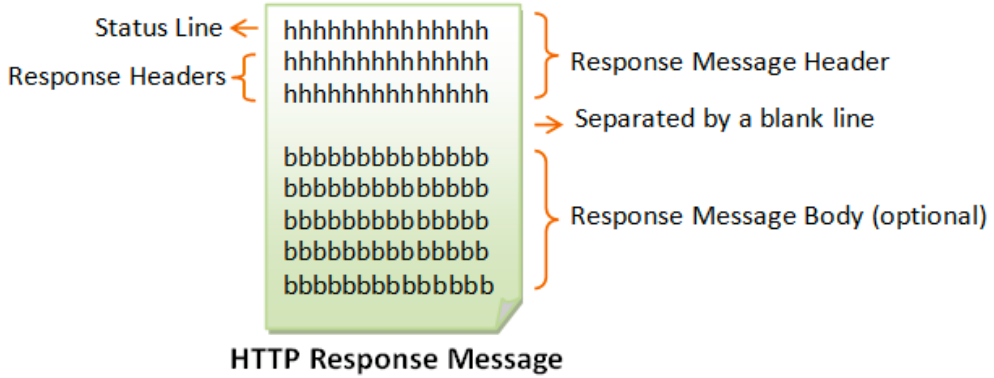
# HTTP Response

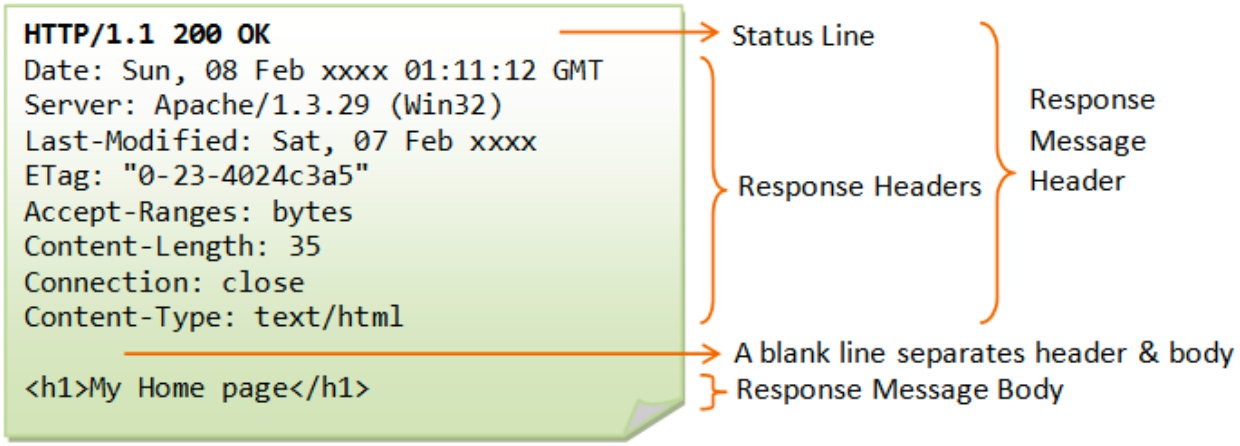
- When this request message reaches the server, the server can take either one of these actions:
  - Returns the *file* requested to the client.
  - Executes the *program*, and returns the output of the program to the client.
  - The request cannot be satisfied, the server returns an error message.

An example of the HTTP response message

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<html><body><h1>It works!</h1></body></html>
```

# HTTP Response



# Response Headers

- Date
  - Timestamp
- Server
  - Vendor name
- Content-length
  - Length of the reply (optional)
- Content-type
  - E.g, plain text, html, mp3 file
- Last Modified
  - Timestamp of last modification of resource
- Cache-control
  - No-cached used to disable web caching in between server and browser
- Expires
  - Timestamp that caches use to discard cached pages

# Return Codes

| Status code | Status message | Description |
|---|---|---|
| 200 | OK | Request was handled without error. |
| 301 | Moved permanently | Content has moved to the hostname in the Location header. |
| 400 | Bad request | Request could not be understood by the server. |
| 403 | Forbidden | Server lacks permission to access the requested file. |
| 404 | Not found | Server could not find the requested file. |
| 501 | Not implemented | Server does not support the request method. |
| 505 | HTTP version not supported | Server does not support version in request. |

# Example: Simple Websever lecture_6_code/webserver/webserver.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <strings.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define LISTENQ 1024
void err_exit() {
  perror("myserver");
  exit(1);
}

int main() {
  int sockfd;
  if((sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0)
    err_exit();
```

# Example: Simple Websever lecture_6_code/webserver/webserver.c

```
struct sockaddr_in serveraddr;
bzero(&serveraddr,sizeof(serveraddr));
serveraddr.sin_family=AF_INET;
serveraddr.sin_addr.s_addr=htonl(INADDR_ANY);
serveraddr.sin_port=htons(8080);

if(bind(sockfd, (struct sockaddr*) &serveraddr, sizeof(serveraddr)) < 0)
  err_exit();

if(listen(sockfd, LISTENQ) <0)
  err_exit();
```

# Example: Simple Websever lecture_6_code/webserver/webserver.c

```
for(;;) {
    int clientfd;
    if((clientfd=accept(sockfd, 0, 0)) < 0)
      err_exit();

    char response[]="HTTP/1.1 200 OK\r\nContent-Type:
text/html\r\n\r\n<HTML><BODY>Hello World!!!!!!!</BODY></HTML>\r\n";
    if(write(clientfd, response, sizeof(response)) < 0)
      err_exit();

    close(clientfd);
  }
}
```

**Terminal:**
./web
**Browser:**
http://127.0.0.1:8080

# Homework Readings

- Readings:
  - Computer Systems – A Programmer‘s Perspective: Section 11.5
  - Unix Network Programing Section 3.4 and 3.7

# Assignment 2

A Basic Sequential Web Proxy

# Problem

- A web proxy is a program that acts as a middleman between a web server and browser.
  - Instead of contacting the server directly to get a web page, the browser contacts the proxy, which forwards the request to the server. When the server replies to the proxy, the proxy sends the reply to the browser.
- You will write a basic sequential web proxy.
  - You will set up the proxy to accept a request, forward the request to the server, and return the result back to the browser, keeping a log of such requests in a disk file.

# Logging

- You will need to log each request to a log file. Your proxy should keep track of all requests in a log file named *proxy.log*. Each log file entry should be in the form:
  - *Date: browserIP URL size*

- browserIP is the IP address of the browser, URL is the URL asked for, size is the size in bytes of the object that was returned. For instance:

- *Sun 27 Oct 2002 02:51:02 EST: 128.2.111.38 http://www.cs.cmu.edu/ 34314*