CS118 Discussion 1B

Week 2

Zhiyi Zhang (zhiyi@cs.ucla.edu)

Contents

- Socket programming (Cont’d)
- Application Layer
- Review HTTP
- Review DNS

Socket Programming (cont’d)

Socket Programming

Continue our programming.

Plan:

- First review the server code
- Then work on the client code

Next two pages

- Page 5: TCP server sample code
- Page 6: TCP client sample code

```
// *** Author: Zhiyi Zhang for CS118, Time: 04/10/2020, TCP Server Sample Code ***
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main()
{
 // *** Initialize socket for listening ***
 int sockfd;
 if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
   perror("socket");
   exit(1);
 }

 // *** Initialize local listening socket address ***
 struct sockaddr_in my_addr;
 memset(&my_addr, 0, sizeof(my_addr));
 my_addr.sin_family = AF_INET;
 my_addr.sin_port = htons(5678);
 my_addr.sin_addr.s_addr = htonl(INADDR_ANY); // INADDR_ANY allows to connect to any one of the host’s IP address

 // *** Socket Bind ***
 if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
   perror("bind");
   exit(1);
 }

 // *** Socket Listen ***
 if (listen(sockfd, 10) == -1) {
   perror("listen");
   exit(1);
 }
```

```c
// *** Author: Zhiyi Zhang for CS118, Time: 04/10/2020, TCP Client Sample Code ***
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>

int main()
{
 // *** Initialize Socket ***
 int sockfd;
 if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
   perror("Cannot create socket");
   exit(1);
 }

 // *** Initialize the server socket address ***
 struct sockaddr_in server_addr; // server socket address struct
 server_addr.sin_family = AF_INET; // protocol family
 server_addr.sin_port = htons(5678); // port number
 struct hostent *host_name = gethostbyname("localhost"); // get IP from host name
 server_addr.sin_addr = *((struct in_addr *)host_name->h_addr); // set IP address
 memset(server_addr.sin_zero, '\0', sizeof server_addr.sin_zero); // make the rest bytes zero

 // *** Connect to the server ***
 if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1) {
   perror("Cannot connect");
   exit(1);
 }

 // *** Socket Read & Write ***
 int sin_size, recvline_size;
 char sendline[1024], recvline[1024];
 while (fgets(sendline, 1024, stdin) != NULL) {
   write(sockfd, sendline, strlen(sendline));
   if (memcmp(sendline, "bye", strlen("bye")) == 0) {
     printf("Will close the connection\n");
     close(sockfd);
```

Application Layer

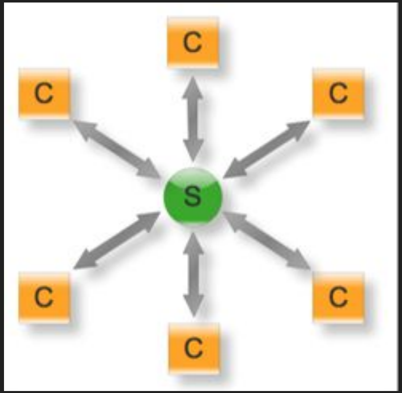
Fat layer with so many different applications

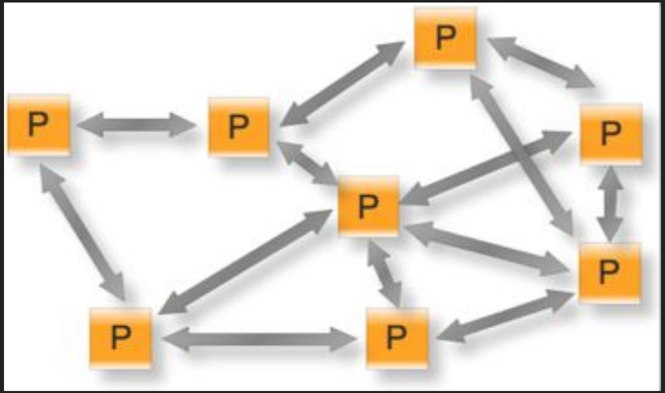
- Web: HTTP
- Email: SMTP, POP
- DNS
- P2P Applications
- Video streaming
- … so many others

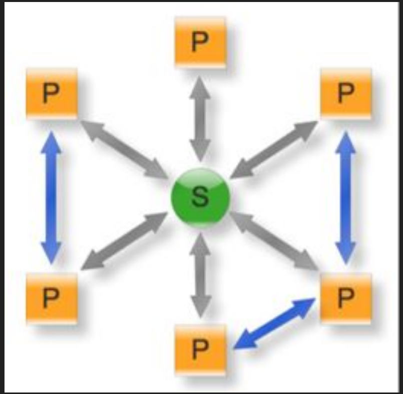
They are all enabled by a thin transport layer (TCP, UDP) and a even thinner layer (IP)

Models

- As we mentioned in previous discussion
  - Client-server: Web, DNS
  - P2P: BitTorrent, BitCoin, Onion Routing
  - Hybrid of two? Skype, BitTorrent + BT Site

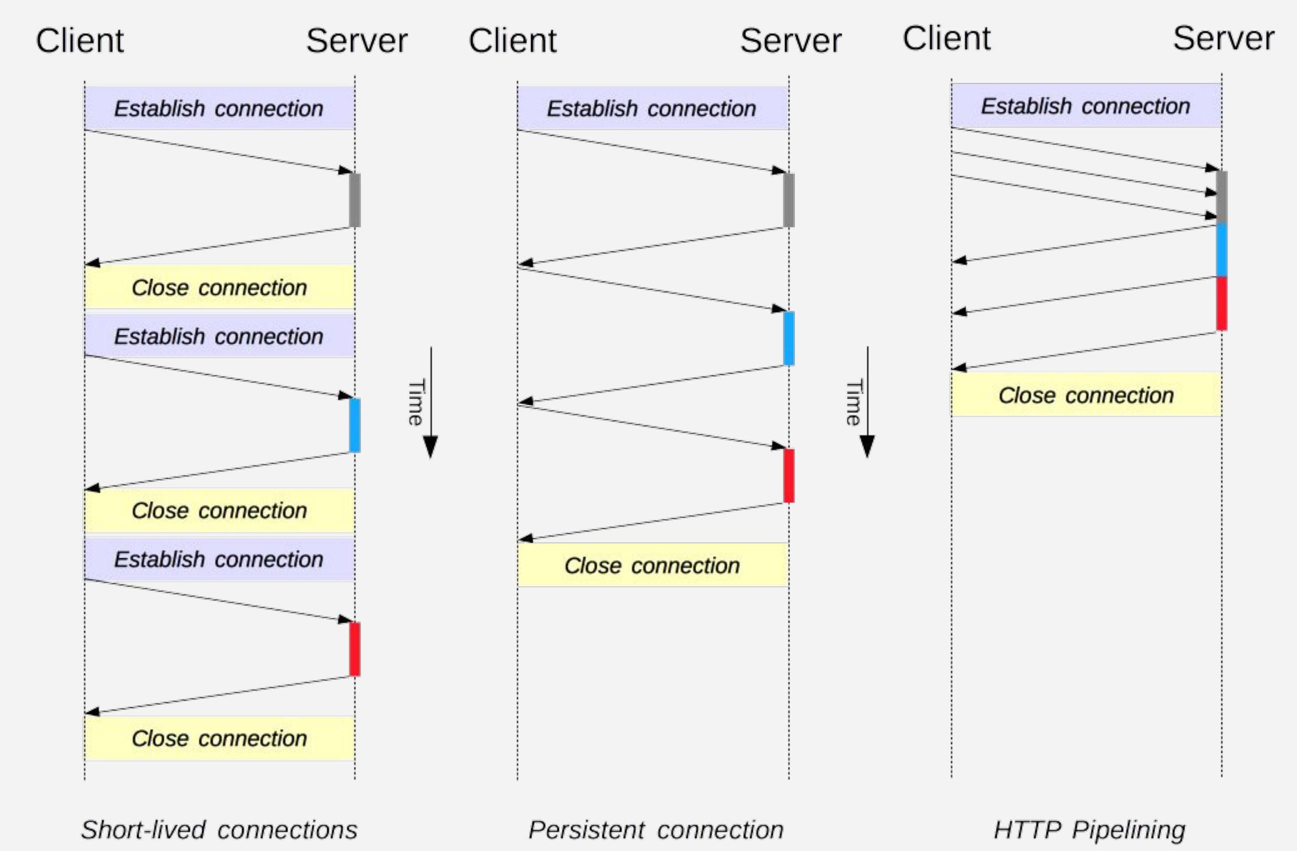






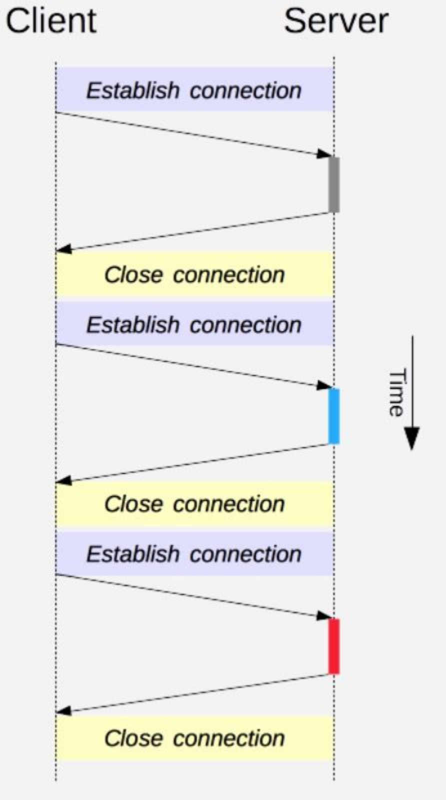
Review HTTP

Non-persistent v.s. Persistent v.s. Pipelining



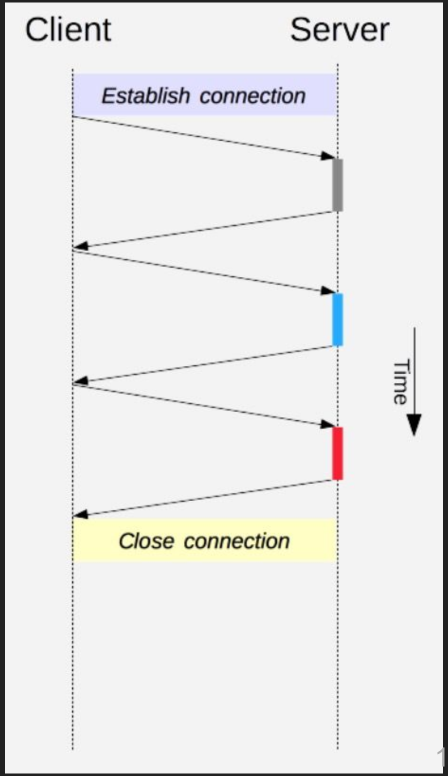
Response Time: Non-persistent HTTP

- To fetch each object, HTTP requires
  - 1 RTT to set up the TCP connection
  - 1 RTT to get the data back
  - Data transmission time
- Therefore: one html with N objects
  - 2 RTTs + N*2 RTTs + Transmission Time
- Example question
  - One HTTP html page with 2 objects? (ignoring tx time)
    - HTTP page: 2 RTTs
    - 2 objects: 2*2 RTTs = 4 RTTs
    - In total: 6 RTTs



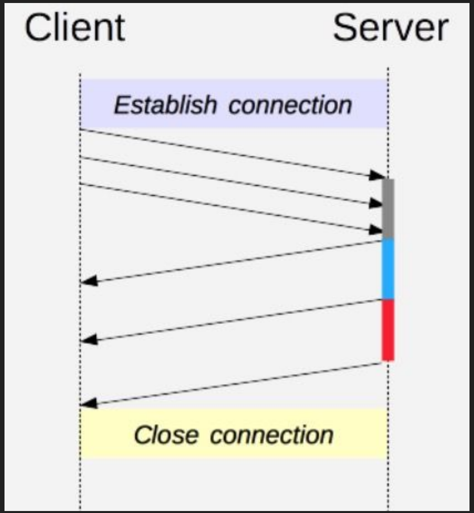
Response Time: Persistent HTTP without Pipeline

- Use the same TCP connection for other objects
- To fetch all objects, HTTP requires
  - 1 RTT to set up the TCP connection
  - 1 RTT to get the HTML file back
  - N RTT to get N objects back, e.g., CSS, JS, pictures
  - Data transmission time
- Therefore
  - (2 + N) RTTs + Transmission Time
- Example question
  - One HTTP page with 2 objects on the same server? (ignoring tx time)
    - HTTP page: 2 RTTs
    - 2 objects: 2 RTTs = 2 RTTs
    - In total: 4 RTTs



Response Time: Persistent HTTP with Pipeline

- Use the same TCP connection for other objects
- To fetch each object, HTTP requires
  - 1 RTT to set up the TCP connection
  - 1 RTT to get the HTML file back
  - 1 RTT to get N objects back, e.g., CSS, JS, pictures
  - Data transmission time
- Therefore
  - 3 RTTs + Transmission Time
- Example question
  - One HTTP page with 2 objects on the same server? (ignoring tx time)
    - HTTP page: 2 RTTs
    - 2 objects: 1 RTTs
    - In total: 3 RTTs

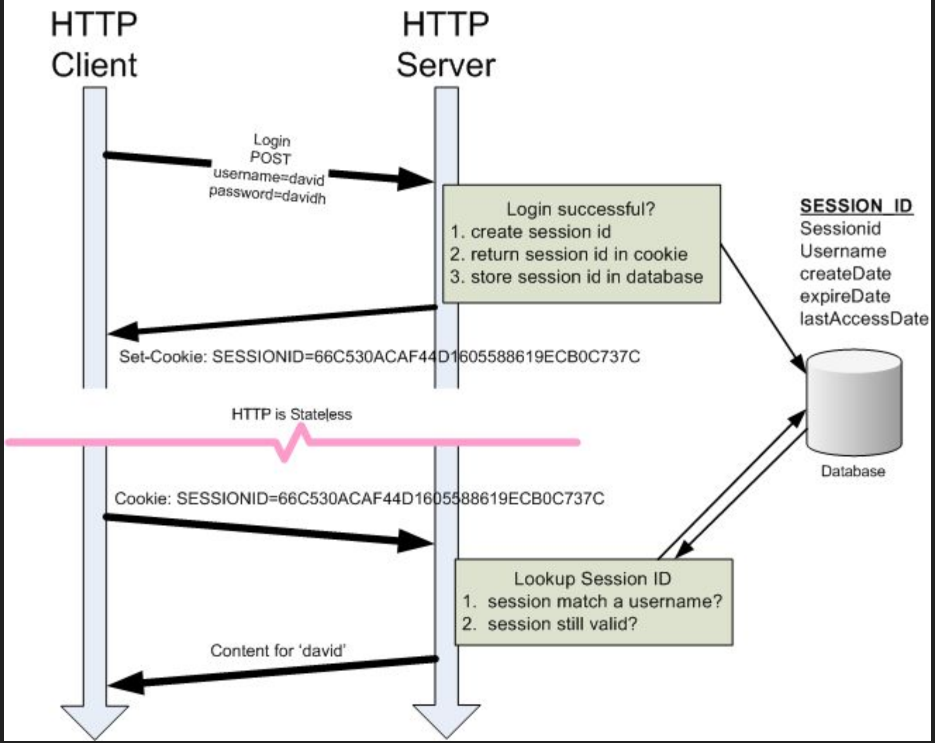


Cookie makes HTTP stateful

- Components of Cookie
  - Cookie header line in response: server initializes the cookie for the client
  - Cookie header line in request: client specifies its cookie
  - Cookie file on the browser
  - Cookie stored in server database
- Main Purposes of Cookie
  - Session management (e.g., you shopping cart keeps the same when you navigate the website, you are kept login for some time)
  - Personalization (e.g., the website remember your preferences)
  - Tracking (e.g., advertisements with third-party cookie)

Cookie: A Typical Workflow

Cookie is kept by your browser



HTTP Client
HTTP Server
Login POST username=david password=davidh
Login successful?
1. create session id
2. return session id in cookie
3. store session id in database
SESSION_ID
Sessionid
Username
createDate
expireDate
lastAccessDate
Set-Cookie: SESSIONID=66C530ACAF44D1605588619ECB0C737C
HTTP is Stateless
Database
Cookie: SESSIONID=66C530ACAF44D1605588619ECB0C737C
Lookup Session ID
1. session match a username?
2. session still valid?
Content for 'david'

Where is our current Internet? (FYI)

- HTTP/1.1

- HTTP/2
- QUIC <= HTTP/3

First need to understand the limitations of 1.1 (FYI)

- High page load latency
  - Head of line blocking
    - The response must be returned in full, in the order of request
  - Large http header
  - Responses are replied only after requesting them first

HTTP/2 (FYI)

- New features
  - Multiplexing over a single TCP connection
    - Responses can come back in arbitrary order
  - HTTP header compression
  - Server push
    - Server can push objects to the client without requests (pre-loading)
- Limitations
  - HTTPS = HTTP over TCP + TLS: When used with TLS, the handshake time is long (TCP handshake + TLS handshake)
  - Cannot work with mobility -- when client changes it IP address, TCP connection is broken

QUIC (Quick UDP Interaction Connections) (FYI)

- Over UDP
- QUIC realizes their own reliability, in-order packet delivery, congestion control plus
  - Combined TLS and QUIC handshake: 0 RTT
    - When QUIC finishes its handshake, so does TLS
  - Connection ID to identity connection instead of <src socket, dest socket>
    - When IP changes, your connection ID won't
- QUIC is in user space (kernel space)
  - Allow easy version update (evolvability)
  - Plug and play congestion control (modularity)

Review DNS

DNS components

- End hosts
  - Who wants to know the IP address of www.ucla.edu
- Local DNS server / DNS Resolver
  - Who helps remember existing DNS responses for the local network
- Authoritative Name servers
  - Who has the knowledge of Domain<=>IP mappings of its own zone

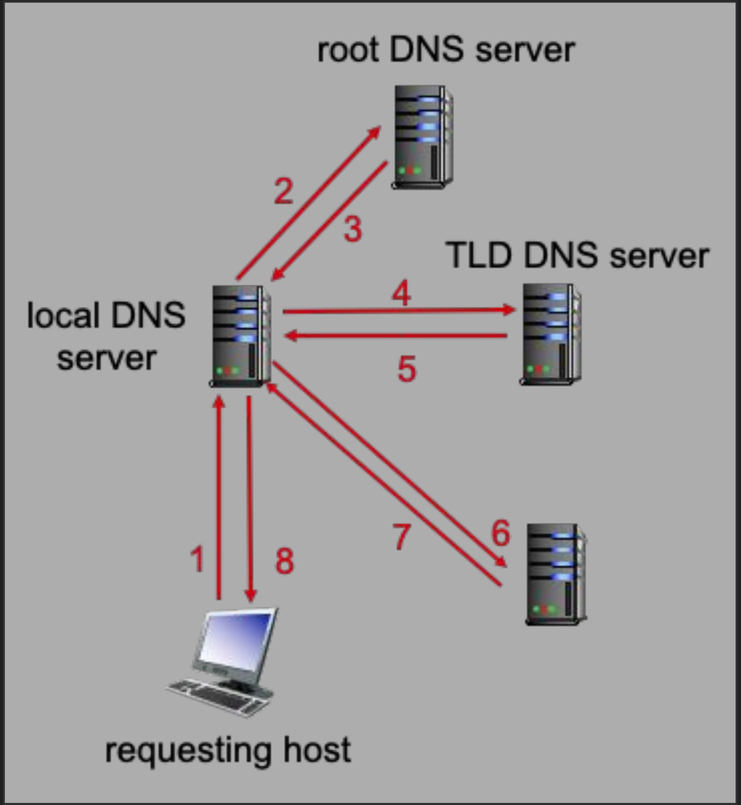
Types of Query

Transport layer used:

- Mainly on UDP

Iterative v.s. Recursive query

- Iterative query
  - Step 2-7
- Recursive query
  - Step 1 and 8



How does it scale for global use?

- Hierarchical structure
- Each authoritative name server cares their own business

dig command and nslookup command

See the DNS query and answer of a domain name.