What makes the world wide web work?

Karen Reid
Simple Web Request

MarkUs

MarkUs is a web application for giving high quality feedback to students. The administration project.
The Request

• How do we tell the web server what we want?
• How do we even find the web server?
• How do the web server and browser talk to each other?
How do we find the server?

• Every computer on the Internet has an Internet address.
• Called an IP address (Internet Protocol)
• An IP address is 4 numbers separated by dots.

markusproject.org = 69.164.221.145
This is getting complicated!

Number of messages?

8 (or so)
Now what?

• Okay, we have the address.
• What do we do with it?
• Let’s look at how two computers communicate.
• HTTP is a high-level protocol
• HTTP is specific to the web.
• Computers communicate for many reasons.
Protocols

• Computers use several layers of general protocols to communicate.
• To understand why these layers are important, think about how a company sends you an invoice for a purchase.
Protocols

Invoice:
Customer: Karen Reid
Order No: 5379

<table>
<thead>
<tr>
<th>Qty</th>
<th>Item</th>
<th>Unit Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Athalon</td>
<td>219.00</td>
<td>219.00</td>
</tr>
<tr>
<td>2</td>
<td>128 MB</td>
<td>149.95</td>
<td>299.90</td>
</tr>
</tbody>
</table>

Subtotal 518.90
Tax 77.84
TOTAL 596.74

Payable to: CPUS are us $596.74
Five hundred ninety six 74/100

Karen Reid
Feb 18, 2001

CPUS are us
Karen Reid
Dept. of Computer Science
University of Toronto

We deliver!

CPUS are us
0 College Street
Toronto Ontario M5S 3G4
TCP/IP

- Transmission Control Protocol.
- Tells us how to package up the data.

<table>
<thead>
<tr>
<th>source address</th>
<th>dest. address</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>ack</td>
</tr>
<tr>
<td></td>
<td>port</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
</tbody>
</table>
TCP Connection

3-way handshake
SYN

Hi 207.99.53.46
Connection port 80?

Hi 192.168.1.1
Let’s talk

okay

Send me a file

Here’s some data

Got it

Here’s some more

Got it

I’m done

I’m done too
Packaging up the data

- Each TCP packet is given a header
  - sequence number
  - checksum

- Make packets

- Put in an IP envelope with another header

<table>
<thead>
<tr>
<th>IP</th>
<th>IP</th>
<th>IP</th>
<th>IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>To 207.99.53.46</td>
<td>To 207.99.53.46</td>
<td>To 207.99.53.46</td>
<td>To 207.99.53.46</td>
</tr>
</tbody>
</table>
The Big Picture

- **Client-Server model**: a client process wants to talk to a server process
- Client must find server - DNS lookup
- Client must find process on server - ports
- Finally establish a connection so two processes can talk
Routing (15 hops)
7 cities, 5 states/prov, 2 countries
Putting it together

browser

markusproject.org

delegate server

local name server

root name server

“org” name server

192.168.1.1

10.126.51.129

67.231.222.81

bloor6.cable.teksavvy.com

tge11-3.fr4.yyz.llnw.net

bloor1.cable.teksavvy.com

tge8-1.fr3.lga.llnw.net

bloor1.cable.teksavvy.com

vian804.esd2.mmun.ac.net

ve5.fr3.yyz.llnw.net

tge8-1.fr3.lga.llnw.net

eqix.e-2-3.tbr2.ewr.nac.net

0.e1-2.tbr2.mmu.nac.net

207.99.53.42

li136-145.members.linode.com
How many messages?

- It depends on the size of the web page
- The web page that appears for markusproject.org is less than 30 Kbytes
- If the web page is 30 Kbytes (small!) it will likely be broken up into ~20 IP packets.

\[
8 \text{ (DNS)} + 20 \times 15 \text{ hops} = 308 \text{ messages}
\]
When something goes wrong

• A packet might not arrive
  – traffic overload
  – bit corruption
• Receiver asks for missing packets to be resent.
• Want to send data as fast as possible.
• But sending too fast wastes resources.
TCP Congestion Control

• Window-based:
  – some number of packets allowed to be sent and not ack’d
  – as successful ack’s arrive, grow window
  – if packet loss is detected, cut window size
TCP Congestion Control

- Time
- Window size
- Packet losses
All we did was click on a link...
Take aways

• The web today is made up of complex layers of software
• No one person, organization, or company could have created it in isolation
• We can understand it because we can study one layer at a time
• We can create new things by building on top of existing layers
TCP: Three-way handshake

**Client**
- socket
- connect (blocks)
- connect returns

**Server**
- sequence number = $K$
- socket, bind, listen
- accept (blocks)

Sequence numbers:
- client sequence number = $J$
- server sequence number = $K$

Handshake:
1. SYN $J$
2. SYN $K$, ack $J+1$
3. ack $K+1$
TCP Server

1. `socket()`
2. `bind()`
3. `listen()`
4. `accept()`

   Block until connection from client

TCP Client

1. `socket()`
2. `connect()`

   Connection establishment (3-way handshake)

3. `write()`

   Data transfer

4. `read()`

   End-of-file notification

5. `close()`
Connection-Oriented

Server
• Create a socket: `socket()`
• Assign a name to a socket: `bind()`
• Establish a queue for connections: `listen()`
• Get a connection from the queue: `accept()`

Client
• Create a socket: `socket()`
• Initiate a connection: `connect()`
Socket Types

• Two main categories of sockets
  – UNIX domain: both processes on the same machine
  – INET domain: processes on different machines

• Three main types of sockets:
  – SOCK_STREAM: the one we will use
  – SOCK_DGRAM: for connectionless sockets
  – SOCK_RAW
Addresses and Ports

- A **socket pair** is the two endpoints of the connection.
- An endpoint is identified by an **IP address and a port**.
- IPv4 addresses are 4 8-bit numbers:
  - 128.100.31.198 = greywolf
  - 128.100.31.203 = redwolf
- **Ports**
  - because multiple processes can communicate with a single machine we need another identifier.
More on Ports

- **Well-known ports: 0-1023**
  - 80 = http
  - 22 = ssh
  - 23 = telnet
  - 21 = ftp
  - 25 = smtp (mail)
  - 194 = irc

- **Registered ports: 1024-49151**
  - 2709 = supermon
  - 26000 = quake
  - 3724 = world of warcraft

- **Dynamic (private) ports: 49152-65535**
TCP Server

socket()
bind()
listen()
accept()
block until connection from client
read()
write()
close()

TCP Client

socket()
connect()
write()
read()
close()

Connection establishment (3-way handshake)
data transfer
end-of-file notification
Server side

```c
int socket(int family, int type, int protocol);
```

- **family** specifies protocol family:
  - `PF_INET` – IPv4
  - `PF_LOCAL` – Unix domain
- **type**
  - `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`
- **protocol**
  - set to 0 except for RAW sockets
- **returns a socket descriptor**
bind to a name

int bind(int sockfd,
          const struct sockaddr *servaddr,
          socklen_t addrlen);

- sockfd – returned by socket
- struct sockaddr_in {
  short sin_family; /*PF_INET */
  u_short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8]; /*filling*/
};

- sin_addr can be set to INADDR_ANY to communicate on any network interface.
int listen(int sockfd, int backlog)

• **after calling** `listen`, **a socket is ready to accept connections**
• prepares a queue in the kernel where partially completed connections wait to be accepted.
• `backlog` is the maximum number of partially completed connections that the kernel should queue.
Complete the connection

```c
int accept(int sockfd,
           struct sockaddr *cliaddr,
           socklen_t *addrlen);
```

- blocks waiting for a connection (from the queue)
- returns a new descriptor which refers to the TCP connection with the client
- `sockfd` is the listening socket
- `cliaddr` is the address of the client
- reads and writes on the connection will use the socket returned by `accept`
Client side

- `socket()` – same as server, to say “how” we are going to talk

```c
int connect(int sockfd,
            const struct sockaddr *servaddr,
            socklen_t addrlen);
```

- the kernel will choose a dynamic port and source IP address.
- returns 0 on success and -1 on failure setting `errno`.
- initiates the three-way handshake.
```c
int soc;
struct hostent *hp;
struct sockaddr_in peer;

peer.sin_family = AF_INET;
peer.sin_port = htons(PORT);
/* fill in peer address */
hp = gethostbyname(argv[1]);
peer.sin_addr = *((struct in_addr *)hp->h_addr);
/* create socket */
soc = socket(PF_INET, SOCK_STREAM, 0);
/* request connection to server */
if (connect(soc, (struct sockaddr *)&peer, sizeof(peer)) == -1) {
    perror("client:connect"); close(soc); exit(1);
}
write(soc, "Hello Internet
", 16);
read(soc, buf, sizeof(buf));
printf("SERVER SAID: %s
", buf);
close(soc);
```
struct sockaddr_in peer;
struct sockaddr_in self;
int soc, ns, k;
int peer_len = sizeof(peer);

self.sin_family = PF_INET;
self.sin_port = htons(PORT);
self.sin_addr.s_addr = INADDR_ANY;
bzero(&(self.sin_zero), 8);

peer.sin_family = PF_INET;
/* set up listening socket soc */
soc = socket(PF_INET, SOCK_STREAM, 0);
if (soc < 0) {
    perror("server:socket"); exit(1);
}

if (bind(soc, (struct sockaddr *)&self, sizeof(self)) == -1) {
    perror("server:bind"); close(soc); exit(1);
}
listen(soc, 1);
...
inetserver.c (concluded)

/* ... repeated from previous slide ... */
soc = socket(PF_INET, SOCK_STREAM, 0);
bind(soc, (struct sockaddr *)&self, sizeof(self))== -1){
    perror("server:bind"); close(soc); exit(1);
}
listen(soc, 1);
... and now continuing ... */

/* accept connection request */
ns = accept(soc, (struct sockaddr *)&peer, &peer_len);
if (ns < 0) {
    perror("server:accept"); close(soc); exit(1);
}
/* data transfer on connected socket ns */
k = read(ns, buf, sizeof(buf));
printf("SERVER RECEIVED: %s\n", buf);
write(ns, buf, k);

close(ns);
close(soc);
Byte order

- **Big-endian**

  \[ 91,329 = \begin{array}{cccc}
  A & A+1 & A+2 & A+3 \\
  00 & 01 & 64 & C1 
  \end{array} \]

- **Little-endian**

  \[ 91,329 = \begin{array}{cccc}
  A+3 & A+2 & A+1 & A \\
  00 & 01 & 64 & C1 
  \end{array} \]

- **Intel is little-endian, and Sparc is big-endian**
Network byte order

• To communicate between machines with unknown or different “endian-ness” we convert numbers to network byte order (big-endian) before we send them.

• There are functions provided to do this:
  – unsigned long htonl(unsigned long)
  – unsigned short htons(unsigned short)
  – unsigned long ntohl(unsigned long)
  – unsigned short ntohs(unsigned short)
Sending and Receiving Data

- **read and write calls** work on sockets, but sometimes we want more control
  - `ssize_t send(int fd, const void *buf, size_t len, int flags);`
    - works like `write` if `flags==0`
    - **flags**: `MSG_OOB`, `MSG_DONTROUTE`, `MSG_DONTWAIT`
  - `ssize_t recv(int fd, void *buf, size_t len, int flags);`
    - **flags**: `MSG_OOB`, `MSG_WAITALL`, `MSG_PEEK`