Communication

Sockets (Haviland – Ch. 10)
Simple Web Request
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 four 8-bit numbers separated by dots.

www.eecg.toronto.edu = 128.100.10.235
Domain Name Servers

Diagram showing the flow of a DNS query from a browser to the root name server, and how the local name server, "ca" name server, and "utoronto" name server are involved in the process.
This is getting complicated!

Number of messages?

10-12
## Protocols

**Invoice:**
Customer: John Doe  
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

---

CPUS are us  
John Doe  
Dept. of Computer Science  
University of Toronto

---

John Doe  
0 College Street  
Toronto Ontario M5S 3G4

---

*We deliver!*  
*Courier*
**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>
</tbody>
</table>

| data |
TCP Connection

Hi 128.100.10.25
Connection port 1?

Hi 128.100.10.128
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
Putting it together

browser

local name server

root name server

“ca” name server

“utoronto” name server

“eecg” name server

HSE-Toronto-ppp.sympatico.ca

dis1-toronto-64-pos2.bellnexxia.net

“utoronto” gw

onel-gw.dis1-toronto.bellnexxia.net

utorgw-border-if.onet.on.ca

sf2-bbup.gw.utoronto.ca

server
How many messages?

- It depends on the size of the web page we retrieve.
- If the web page is 75 Kbytes (small!) it will be broken up into 103 IP packets.
- Remember DNS took 10 messages

$$10 + 103 \times 7 \text{ hops} = 731 \text{ messages!}$$
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
Sockets

• One form of communication between processes.
• Similar to pipes, except sockets can be used between processes on different machines.
• Use file descriptors to refer to sockets.
• Built on top of TCP layer
TCP: Three-way handshake

**client**

- sequence number = \( J \)
- socket
- connect (blocks)
- connect
- returns

**server**

- sequence number = \( K \)
- socket, bind, listen
- accept (blocks)
- SYN \( J \)
- SYN \( K \), ack \( J+1 \)
- ack \( K+1 \)
- accept returns
TCP Server

- socket()
- bind()
- listen()
- accept()
- read()
- write()
- close()

TCP Client

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

Connection establishment (3-way handshake)
block until connection from client
data transfer
end-of-file notification
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:
  - 128.100.31.200 = werewolf
  - 128.100.31.201 = seawolf
  - 128.100.31.202 = skywolf

- Ports
  - because multiple processes can communicate with a single machine we need another identifier.
More on Ports

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

- **Registered ports: 1024-49151**
  - 2709 = supermon
  - 26000 = quake

- **Dynamic (private) ports: 49152-65535**
  - You should pick ports in this range to avoid overlap
**TCP Server**

- **socket()**
- **bind()**
- **listen()**
- **accept()**
- **read()**
- **write()**
- **close()**

**TCP Client**

- **socket()**
- **connect()**
- **write()**
- **read()**
- **close()**

**Connection establishment**

(3-way handshake)

- **block until connection**
- **from client**

**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

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

- **sockfd** – returned by socket
- **struct sockaddr_in** {
  ```
  short          sin_family; /*AF_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 with any host
Set up queue in kernel

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 *cliaaddr,  
            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
- `cliaaddr` 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 addrrlen);
```

- 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.
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\n", 16);
read(soc, buf, sizeof(buf));
printf("SERVER SAID: %s\n", buf);
close(soc);
struct sockaddr_in peer;
struct sockaddr_in self;
int soc, ns, k;
int peer_len = sizeof(peer);

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

peer.sin_family = AF_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);
...
/* ... 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

\[ 91,329 = \]

<table>
<thead>
<tr>
<th>A</th>
<th>A+1</th>
<th>A+2</th>
<th>A+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>64</td>
<td>C1</td>
</tr>
</tbody>
</table>

• Little-endian

\[ 91,329 = \]

<table>
<thead>
<tr>
<th>A+3</th>
<th>A+2</th>
<th>A+1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>64</td>
<td>C1</td>
</tr>
</tbody>
</table>

• 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`