Accessing the Network

CS439: Principles of Computer Systems
April 12, 2017
Last Time

• Introduction to Networks
• OSI Model (7 layers)
  – Layer 1: hardware
  – Layer 2: Ethernet (frames)
    • SAN, LAN, WAN
  – Layer 3: IP (packets)
  – Layer 4: TCP, UDP (segments)
• Network communication
  – Protocols, naming, routing
• Client/Server transactions
• TCP/IP congestion control mechanisms
Transferring Data Over an Internet

LAN1

(1) data

internet packet

(2) data PH FH1

LAN1 frame

(3) data PH FH1

LAN1 adapter

Protocol software

Host A

client

Host B

server

LAN2

(4) data PH FH1

LAN1 frame

(5) data PH FH2

Router

LAN1 adapter

Protocol software

LAN2 adapter

LAN2 frame

(6) data PH FH2

(7) data PH FH1

(8) data

PH: Internet packet header

FH: LAN frame header
Transferring Data Over an internet: Text Description

• Example has 2 LANs connected by a router
• Terminology:
  – PH: Internet packet header
  – FH: LAN frame header
• Steps to transfer:
  1: Data is sent from Host A’s client application to protocol software
  2. The PH and FH1 are appended to the data and sent along to the LAN1 adapter
  3. LAN1 adapter sends Data + PH + FH1 along to the router’s LAN1 adapter
  4. Data + PH + FH1 sent to router’s protocol software, which strips off FH1 and creates FH2
  5. Data + PH + FH2 sent to router’s LAN2 adapter
  6. Data + PH + FH2 sent to LAN2 adapter in LAN2
  7. Data + PH + FH2 sent along to protocol software where the PH and FH2 are stripped off
  8. Data goes along to Host B’s server application
Today’s Agenda

Accessing the Network

– Sockets
  • Client-Side Programming
  • Server-Side Programming

– Remote Procedure Calls (RPC)
Accessing the TCP/IP Family From User Code
A Client-Server Transaction

1. Client sends request
2. Server handles request
3. Server sends response
4. Client handles response

Note: clients and servers are processes running on hosts (can be the same or different hosts)

Most network applications are based on the client-server model:
- A server process and one or more client processes
- Server manages some resource
- Server provides service by manipulating resource for clients
- Server activated by request from client
Internet Connections

• Clients and servers communicate by sending streams of bytes over connections

• A socket is an endpoint of a connection
  – Underlying basis for all Internet applications
  – Created in the early 80s as a part of the original Berkeley distribution of UNIX that contained an early version of the Internet protocols
  – Socket address is an IPaddress:port pair

• A connection is uniquely identified by the socket addresses of its endpoints (socket pair)
  – (client_addr:client_port, server_addr:server_port)
Sockets: Implementation Details

• To the kernel, a socket is an endpoint of communication
• To an application, a socket is a file descriptor that lets the application read/write from/to the network
  – Remember: All Unix I/O devices, including networks, are modeled as files
• Clients and servers communicate with each other by reading from and writing to socket descriptors

- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors
Overview of the Sockets Interface

Client

- socket
- connect

Server

- socket
- bind
- listen
- accept

Connection request
Client: `socket(2)`

`socket(2)` creates a socket descriptor on the client

- Allocates and initializes some internal data structures
- `AF_UNSPEC`: indicates that the socket is associated with Internet protocols but either IPv4 or IPv6 is fine
- `SOCK_STREAM`: selects a reliable byte stream connection
  - Bi-directional pipes
  - Gives you TCP
- `SOCK_DGRAM` results in UDP

```c
int clientfd; /* socket descriptor */

if ((clientfd = socket(AF_UNSPEC, SOCK_STREAM, 0) <0)
    return -1; /* check errno for cause of error */
...
```
Domain Name Service (DNS)

Functions for retrieving host entries from DNS:

**getaddrinfo()**: query key is a DNS domain name, returns:

```c
struct addrinfo {
    int ai_flags;
    int ai_family;  /*desired address family*/
    int ai_socktype;  /*desired Layer 4 protocol*/
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
};
```

**getnameinfo()**: query key is an IP address, returns a sockaddr struct (more soon)
Client: Find Server Using DNS

The client then builds the server’s Internet address

```c
int clientfd; /* socket descriptor */

/* setup information*/
struct addrinfo hints;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC; /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */

/* fill in the server's IP address and port */
struct addrinfo *result = NULL; /* DNS host entry */
int s = 0;
if ((s = getaddrinfo(hostname, port, &hints, &result)) != 0)
    return -1;
```

Note: This is untested code.
Finally the client creates a connection with the server

- Client process blocks until the connection is created
- After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls (typically send/recv) on descriptor `clientfd`

```c
int clientfd; /* socket descriptor */
struct addrinfo *result; /* DNS host entry initialized in * call to getaddrbyinfo()*/
typedef struct sockaddr SA; /* generic sockaddr */

... /* Establish a connection with the server */
if (connect(clientfd, result->ai_addr, result->ai_addrlen) < 0) return -1;
```
Overview of the Sockets Interface

Client

socket

connect

Server

socket

bind

listen

Connection request

accept
**Server: bind(2)**

`bind()` associates the socket with the socket address (created similarly to that of the client)

```c
int listenfd;
struct addrinfo *result;
/* listening socket */
/* DNS host entry initialized in */
/* call to getaddrbyinfo()*/

... /* listenfd will be an endpoint for all requests to port */
/* on any IP address for this host */
if (bind(listenfd, rp->ai_addr, rp->ai_addrlen) < 0)
    return -1;
```
Server: `setsockopt(2)`

- Avoid problems with `bind()`... Give the socket some attributes

```c
/* Eliminates "Address already in use" error from bind(). */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
             (const void *)&optval, sizeof(int)) < 0)
    return -1;
```

- Handy trick that allows us to rerun the server immediately after we kill it
  - Otherwise we would have to wait about 15 seconds
  - Eliminates “Address already in use” error from `bind()`
- Strongly suggest you do this for all your servers to simplify debugging
Server: `listen(2)`

- `listen()` indicates that this socket will accept connection (`connect`) requests from clients
- `LISTENQ` is a constant indicating how many pending requests allowed

```c
int listenfd; /* listening socket */

... /* Make it a listening socket ready to accept connection requests */
if (listen(listenfd, LISTENQ) < 0)
    return -1;
```

- We’re finally ready to enter the main server loop that accepts and processes client connection requests.
Server: \texttt{accept}

- \texttt{accept()} blocks waiting for a connection request
  
  ```c
  int listenfd; /* listening descriptor */
  int connfd;   /* connected descriptor */
  struct sockaddr_in clientaddr;
  int clientlen;

  clientlen = sizeof(clientaddr);
  connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
  ```

- \texttt{accept()} returns a \textit{connected descriptor} (\texttt{connfd}) with the same properties as the \textit{listening descriptor} (\texttt{listenfd})
  - Returns when the connection between client and server is created and ready for I/O transfers
  - All I/O with the client will be done via the connected socket

- \texttt{accept()} also fills in client’s IP address
The `listen()` call is listening on a different port than the port the server will eventually use to send data to the client.

A. True
B. False
Socket Implementation in the OS

• Each socket fd has associated socket structure with:
  – Send and receive buffers
  – Queues of incoming connections (on listen socket)
  – A protocol control block (PCB)
  – A protocol handle

• PCB contains protocol-specific information, such as:
  – Pointer to IP TCB with source/destination IP address and port
  – Information about received packets and position in stream
  – Information about unacknowledged sent packets
  – Information about timeouts
  – Information about connection state (setup/teardown)
Putting It All Together: Anatomy of an Internet Connection

Client socket address
128.2.194.242:51213

Connection socket pair (initial contact)
(128.2.194.242:51213, 208.216.181.15:80)

Server socket address
208.216.181.15:80

Client host address
128.2.194.242

Server host address
208.216.181.15
Putting It All Together: Anatomy of an Internet Connection

Client socket address
128.2.194.242:51213

Server ephemeral socket address
208.216.181.15:65398

Client host address
128.2.194.242

Server host address
208.216.181.15

Connection socket pair (established)
(128.2.194.242:51213, 208.216.181.15:65398)
The System Event Notification Service service failed the logon.
The remote procedure call failed.
Remote Procedure Calls
Remote Procedure Calls (RPC)

- Clients execute services on a remote server using a procedure call
- Servers export procedures for clients to call
- Communication details are hidden from the client and handled by the RPC Mechanism
- Is NOT message passing!
  - message passing requires more work for the application, as we saw with TCP/IP and we’ll see again next week
RPC: Lower-Level

The RPC mechanism uses the procedure *signature* (number and type of arguments and return value)

1. to generate a client stub that bundles RPC arguments and sends them off to the server.
2. to generate the server stub that unpacks the message and makes the procedure call

The stubs actually do the work, and the client and server communicate through the stubs.
Remote procedure calls abstract out the *send-wait-reply* paradigm into a “procedure call”

Remote procedure calls can be made to look like “local” procedure calls by using a *stub* that hides the details of remote communication.
RPC: More Detail

process P1
begin
  call Function(args)
end P1

procedure realFunction(args)
begin
  return(results)
end realFunction

procedure Function(args)
begin
  <marshall parameters>
  send(FunctionServer,params)
  receive(FunctionServer,results)
  <unpack results>
  return(results)
end Function

process FunctionServer
begin
  loop
    sender := select()
    receive(sender,params)
    <unpack parameters>
    call realFunction(args)
    <marshall results>
    send(sender,results)
  end loop
end FunctionServer

Client -> Network -> Server
Problems with RPC

• Failure handling
  – A program may hang because of
    • Failure of a remote machine; or
    • Failure of the server application on the remote machine
  – An inherent problem with distributed systems, not just RPC
    • Lamport: “A distributed system is one where you can’t do work because some machine that you have never heard of has crashed”

• Performance
  – Cost of procedure call << same machine RPC << network RPC
More Insidious Problems with RPC

• Cannot pass pointers
  – call by reference becomes copy-restore (but might fail)
• Weakly typed languages
  – client stub cannot determine size
• Not always possible to determine parameter types
• Cannot use global variables
  – may get moved to remote machine
Why does turning every file system operation into an RPC to a server perform poorly?

A. Disk latency is larger than network latency
B. Network latency is larger than disk latency
C. No server-side cache
D. No client-side cache
Summary

• Remote procedure calls enable a client to perform computation on a server
• Support is provided somewhere other than the application (may be language, may be OS)
• RPC is really useful for harnessing power of distributed computation
• RPC can have some drawbacks---particularly if you are not aware you are executing an RPC
Announcements

• Exams are graded, scores are not entered
  – Will be returned in discussion section this week
• Project 3 due Friday
• Project 4 out Friday
  – Will discuss in discussion section NEXT week
  – Group registration is due Wednesday, 4/19
• Problem Set 9 is due Friday in section