Accessing the Network

CS439: Principles of Computer Systems
November 11, 2015
Last Time

• Introduction to Networks
  – SAN, LAN, WAN
• OSI Model (7 layers)
  – Layer 1: hardware
  – Layer 2: Ethernet (frames)
    • hardware to hardware
    • ARP, MAC addresses
  – Layer 3: IP (packets)
    • sending machine to receiving machine
    • DNS, IP addresses
  – Layer 4: TCP, UDP (segments)
    • Sending process to receiving process
    • Ports
  – Layers 5, 6: OS stuff
  – Layer 7: Application (soon)
• Network communication
  – Protocols, naming, routing
• TCP/IP congestion control mechanisms
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)
Today’s Agenda

Accessing the Network

– Client-Server Transactions
– Ports
– 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
The Client/Server Model

• One of the most common models for structuring network communication (and now distributed computation)

• A server is a process or collection of processes that provides a service
  – may exist on one or more nodes

• A client is a program that uses the service
  – first binds to the server (locates it in the network and establishes a connection)
  – then sends the server a request to perform some action
Clients

• Examples of client programs
  – Web browsers, ftp, telnet, ssh

• How does a client find the server?
  – The IP address in the server socket address identifies the host (more precisely, an adapter on the host)
  – The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
Servers

• Each server typically waits for requests to arrive on a well-known port associated with a particular service

• A machine that runs a server process is also often referred to as a “server”
Server Examples

• Web server
  – Resource: files/compute cycles (CGI programs)
  – Service: retrieves files and runs CGI programs on behalf of the client

• FTP server
  – Resource: files
  – Service: stores and retrieve files

• Telnet server
  – Resource: terminal
  – Service: proxies a terminal on the server machine

• Mail server
  – Resource: email “spool” file
  – Service: stores mail messages in spool file
Well-Known (TCP) Ports

• Port 21: FTP
• Port 22: SSH
• Port 23: Telnet
• Port 25: SMTP (Email)
• Port 79: Finger
• Port 80: Web

See /etc/services for a comprehensive list of the port mappings on a Linux machine
Fine, But....

Hurry up! We *still* don’t know how to use them!

So...

How do we *USE* them?

...

Funny you should ask....
Sockets!

- Created in the early 80s as a part of the original Berkeley distribution of UNIX that contained an early version of the Internet protocols
- Underlying basis for all Internet applications
- Based on the client/server programming model
What is a Socket?

- 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

![Diagram of client and server communication with socket descriptors](Diagram)

- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors
Internet Connections

• Clients and servers communicate by sending streams of bytes over connections:
  – Point-to-point, full-duplex (2-way communication), and reliable.
• A socket is an endpoint of a connection
  – Socket address is an IP:port pair
• A port is a 16-bit integer that identifies a process:
  – Well-known port: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)
  – Ephemeral port: Assigned automatically on client when client makes a connection request
• A connection is uniquely identified by the socket addresses of its endpoints (socket pair)
  – (client_addr:client_port, server_addr:server_port)
Overview of the Sockets Interface

Client

socket

connect

Connection request

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_INET`: indicates that the socket is associated with Internet protocols
- `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_INET, SOCK_STREAM, 0)) < 0) return -1; /* check errno for cause of error */
...
```
Client: Find Server Using DNS

The client then builds the server’s Internet address

```c
int clientfd;            /* socket descriptor */
struct hostent *hp;      /* DNS host entry */
struct sockaddr_in serveraddr; /* server’s IP address */
...

/* fill in the server's IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
serveraddr.sin_addr.s_addr = htonl(hp->h_addr_list[0]);
```

Note: This is untested code.
Client: `connect()`

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 sockaddr_in serveraddr;            /* server address */
typedef struct sockaddr SA;              /* generic sockaddr */

...                                           

/* Establish a connection with the server */
if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)  
    return -1;
```
Overview of the Sockets Interface

**Client**
- socket
- connect

**Server**
- socket
- bind
- listen
- accept

Connection request
Server: **bind(2)**

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

```c
int listenfd;       /* listening socket */
struct sockaddr_in serveraddr; /* server’s socket addr */
...

/* listenfd will be an endpoint for all requests to port
   on any IP address for this host */
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 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: `accept`

- `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);
  ```

- `accept()` returns a **connected descriptor** (`connfd`) with the same properties as the **listening descriptor** (`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

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

A. True
B. False
Socket Implementation

• 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

Server socket address: 208.216.181.15:80

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

Client host address: 128.2.194.242

Server host address: 208.216.181.15
Overview of Connection Setup

Server
(Running on serverIP)

- Create socket, port=x, for incoming request:
  serverSocket = socket()

- Wait for incoming connection request:
  connectionSocket = serverSocket.accept()

- Read request from connectionSocket

- Write reply to connectionSocket

- Close connectionSocket

TCP connection setup

Client

- Create socket, connect to serverIP, port=x:
  clientSocket = socket()

- Send request using clientSocket

- Read reply from clientSocket

- Close clientSocket
Remote Procedure Calls
Remote Procedure Calls (RPC)

• Servers export procedures for some set of clients to call
• To use the server, the client does a procedure call
• OS manages the communication
• 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
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: 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.
RPC: More Detail

process P1
begin
  call Function(args)
end P1

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

process FunctionServer
begin
  loop
    sender := select()
    receive(sender,params)
    call realFunction(args)
    send(sender,results)
  end loop
end FunctionServer

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

Client

Network

Server
RPC and Regular Procedure Calls

Similarities between procedure call and RPC:

– Parameters ↔ request message
– Result ↔ reply message
– Name of procedure ↔ passed in request message
– Return address ↔ mailbox of the client
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
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
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, 11/13
• Project 4 out Friday
  – Will discuss in discussion section NEXT week
• Homework 9 is posted and due Thursday 9:45a