Computer Networks
LECTURE 3
Socket Programming and Application Layer Protocols

Sandhya Dwarkadas
Department of Computer Science
University of Rochester

Assignments

- Lab 2 – socket programming
  - DUE: Wednesday September 14th

Computer Networks: Principles and Architecture

- Network Architecture:
  - Design and Implementation Guide

Principle of Abstraction – layering of protocols

A protocol provides a communication service to the next higher-level layer

- Service interface – e.g., send and receive
- Peer interface – form and meaning of messages exchanged between peers

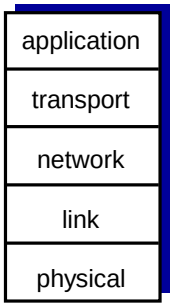
Why layering?

dealing with complex systems:

- explicit structure allows identification, relationship of complex system's pieces
  - layered *reference model* for discussion
- modularization eases maintenance, updating of system
  - change of implementation of layer's service transparent to rest of system
  - e.g., change in gate procedure doesn't affect rest of system
- layering considered harmful?

Internet protocol stack

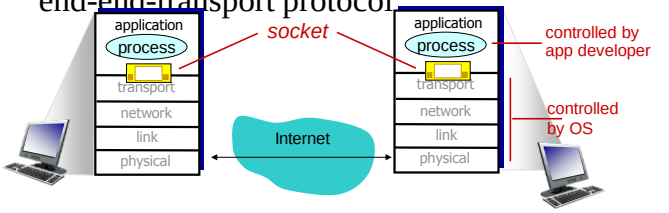
- *application:* supporting network applications
  - FTP, SMTP, HTTP
- *transport:* **process-process data transfer**
  - **TCP, UDP**
- ***network:*** **routing** of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- *physical:* bits "on the wire"



Socket programming

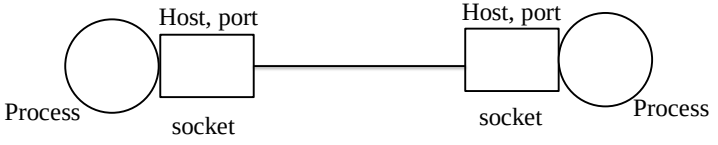
goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Operating Systems Support: Sockets

- A socket is an operating system abstraction in which a port is embedded
- A port is a communication endpoint

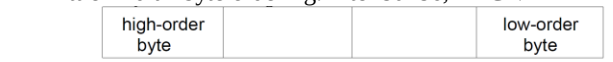


Socket Addresses

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr{
   u_short sa_family;  /*address family: AF_xxx value*/
   char    sa_data[14];/* up to 14 bytes of addr  */
};                  /*   (protocol-specific)   */
struct sockaddr_in {
   short         sin_family; /* AF_INET */
   u_short       sin_port;   /* 16-bit port number */
   struct in_addr sin_addr;  /* 32-bit netid/hostid */
   char          sin_zero[8];/* unused */
}; /*sin_port and sin_addr are network byte ordered*/
```

Network Byte Order

- Two ways to map byte addresses onto words:
  - Little Endian byte ordering: Intel 80x86, DEC VAX



  - Big Endian byte ordering: IBM 360/370, Motorola 68K, MIPS, SPARC, HPPA

- Network byte order (TCP/IP, XNS, SNA protocols): big-endian in protocol headers
- Byte ordering/alignment routines: htonl, htons, ntohl, ntohs

Socket System Call

- Creates an endpoint for communication

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

  - Family (or domain): AF_UNIX, AF_INET, AF_NS, AF_IMPLINK
  - type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, ...
  - protocol: typically 0
  - Returns a socket descriptor, or sockfd

- Methods by which socket options may be changed
  - setsockopt
  - fcntl
  - ioctl

Services Provided by the Transport Layer

- Connection-oriented (virtual circuit) versus connection-less (datagram) protocols
- Sequencing
- Error control
- Flow control
- Byte stream vs. messages
- Full duplex vs. half duplex

- Connection-oriented (virtual circuit)
  - Establish a connection
  - Transfer data
  - Terminate connection
- Connection-less (datagram)
  - Each message or datagram transmitted independently – must contain all information for delivery

The 5-tuple that completely specifies the two processes that make up a connection

```
{protocol, local-addr, local-process,
                    foreign-addr, foreign-process}
```

Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

- Operations:
  - fd = socket(): creates the socket (think OS metadata to manage communication endpoint)
  - bind(fd, port): binds socket to local port (address)
  - sendto(), recvfrom(), write(), read(): operations for sending and receiving data
  - close(fd): close the connection

Socket programming *with UDP*

UDP: no "connection" between client & server

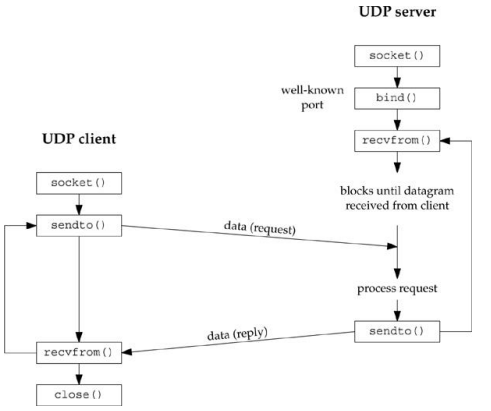
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP addres and port# from received packet

UDP: transmitted data may be lost or received out-of-order

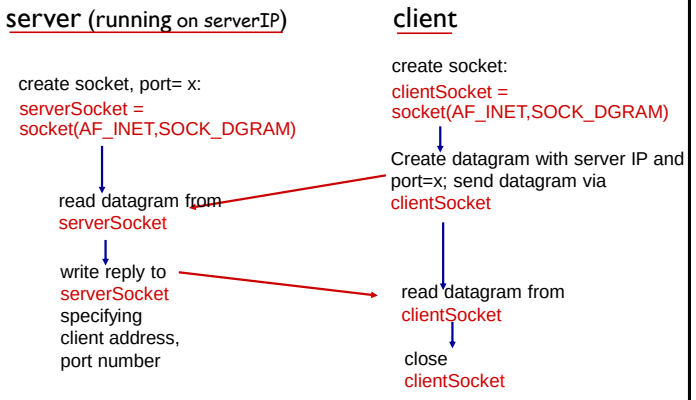
Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Datagram Protocol: UDP



Client/server socket interaction: UDP



Example app: UDP client

Python UDPClient

include Python's socket library →
```
from socket import *
serverName = 'hostname'
serverPort = 12000
```
create UDP socket for server →
```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```
get user keyboard input →
```
message = raw_input('Input lowercase sentence:')
```
Attach server name, port to message; send into socket →
```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```
read reply characters from socket into string →
```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
```
print out received string and close socket →
```
clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

create UDP socket

bind socket to local port number 12000

loop forever

Read from UDP socket into message, getting client's address (client IP and port)

send upper case string back to this client

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chapter 3)

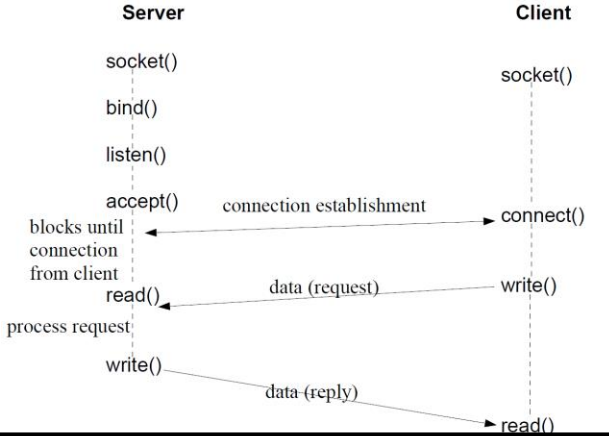
application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

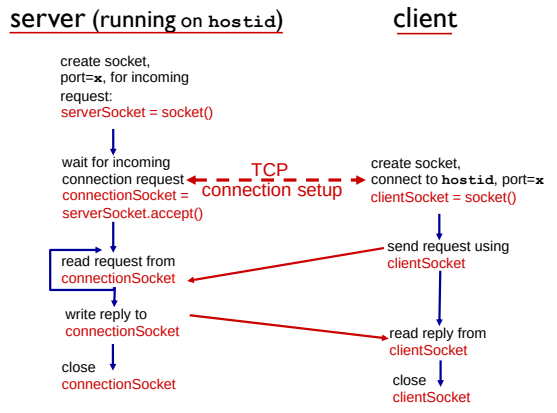
Connection-Oriented System Calls

- Connect: Establishes a connection with a server
  - includes bind - assigns 4 elements of the 5-tuple
- Listen: Indicates that server is willing to accept connections (allows queueing)
- Accept: Waits for actual connection from client process – creates a new socket descriptor
  - assumes a concurrent server

Socket System Calls for Connection-Oriented Protocol: TCP



Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                    encode())
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

Client-Server Model

- Iterative versus concurrent servers
- Role of client and server asymmetric

I/O Multiplexing

- Methods by which I/O may be multiplexed
  - Set socket to non-blocking and poll
    - fcntl(fd, F_SETFL, FNDELAY), where FNDELAY implies non-blocking I/O) non-blocking I/O
  - Fork a process/thread per socket
  - SIGIO - asynchronous I/O with interrupts
  - Select - wait or poll for multiple events

Select System Call

- Waits on several socket descriptors
- int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
- FD_SET(fd, &fdset)
- FD_CLR(fd, &fdset)
- FD_ISSET(fd, &fdset)
- FD_ZERO(&fdset)

Events detected by Select:
Using `readfds`:

- data available for reading
- read-half of connection is established
- listen socket has queued connection
- socket error pending

Using `writefds`:

- space available for writing
- write-half of connection is closed
- socket error pending

select() Example

```
static int Loop(int net_fd)
{
  int ret;
  struct timeval timeout;
  fd_set fdset;

  for (;;) {
    FD_ZERO(&fdset); FD_SET(net_fd, &fdset);
    timeout.tv_sec = 1; timeout.tv_usec = 0;
    ret = select(FD_SETSIZE, &fdset, NULL, NULL, &timeout);
    if (ret < 0) {
      if (errno == EINTR)
        continue; /* signal delivered - restart */
      error(''select failed'');
    }
    else if (ret > 0) {
      if (FD_ISSET(net_fd, &fdset)) {
        FD_CLR(net_fd, &fdset);
        net_process(net_fd); /* process message on socket */
      }
    }
    else ; /* timer expired - restart */
  } /* end forever */
}
```

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from the kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID's
  - The only information in a signal is its ID and the fact that it arrived.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signal Concepts (cont)

- Receiving a signal
  - A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.
  - Three possible ways to react:
    - Ignore the signal (do nothing)
    - Terminate the process.
    - *Catch* the signal by executing a user-level function called a signal handler.
      - Akin to a hardware exception handler being called in response to an asynchronous interrupt.

Signal Concepts (cont)

- A signal is *pending* if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- A process can *block* the receipt of certain signals.
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
- A pending signal is received at most once.

Signal Concepts

- Kernel maintains `pending` and `blocked` bit vectors in the context of each process.
  - `pending` – represents the set of pending signals
    - Kernel sets bit k in `pending` whenever a signal of type k is delivered.
    - Kernel clears bit k in `pending` whenever a signal of type k is received
  - `blocked` – represents the set of blocked signals
    - Can be set and cleared by the application using the `sigprocmask` function.

Default Actions

- Each signal type has a predefined *default action*, which is one of:
  - The process terminates
  - The process terminates and dumps core.
  - The process stops until restarted by a SIGCONT signal.
  - The process ignores the signal.

Receiving Signals

- Suppose kernel is returning from exception handler and is ready to pass control to process *p*.
- Kernel computes `pnb = pending & ~blocked`
  - The set of pending nonblocked signals for process *p*
- If (`pnb == 0`)
  - Pass control to next instruction in the logical flow for *p*.
- Else
  - Choose least nonzero bit *k* in `pnb` and force process *p* to receive signal *k*.
  - The receipt of the signal triggers some *action* by *p*
  - Repeat for all nonzero *k* in `pnb`.
  - Pass control to next instruction in logical flow for *p*.

Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`
- Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum`
  - SIG_DFL: revert to the default action on receipt of signals of type `signum`.
  - Otherwise, handler is the address of a *signal handler*
    - Called when process receives signal of type `signum`
    - Referred to as "*installing*" the handler.
    - Executing the handler is called "*catching*" or "*handling*" the signal.
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

Signals you might use

SIGALRM - can be initiated by a process by calling setitimer and setting the appropriate sigaction

SIGIO - indicates that I/O is possible on a file descriptor. Can use this signal to provide a form of asynchronous I/O for a process

Asynchronous I/O

1. Establish a SIGIO handler (signal driven)
   (e.g., `signal(SIGIO, sigio_func)`, or `sigaction`)
2. Set process ID to receive SIGIO signals
   (`fcntl(int fd, F_SETOWN, pid)`)
3. Enable asynchronous I/O
   (`fcntl(int fd, F_SETFL, FASYNC)`)

Useful System Routines

sigaction - establish a handler (can also use signal)

setitimer - set timer value at which timer interrupt should occur

select - determine whether data has been received on any descriptor

fcntl - set process ID to receive SIGIO signal, enable I/O with F_SETOWN and getpid(), F_SETFL and FASYNC (in order to get interrupted when data arrives)

sigprocmask - block signals in critical sections

Useful Routines to Handle Interrupts

`/usr/include/signal.h`
`/usr/include/sys/signal.h`

sigaction — initialize action
e.g., sigaction(SIGALARM, &sa, &old_sa)
where sa.sa_handler = timer alarm handler, and
sa.sa_mask = signals to be blocked

Example Timer Setting:

```
time_out = { (0, 50000), (0, 50000)}
setitimer(ITIMER_REAL, &time_out, NULL);
```

Example Socket Control:

```
fcntl(fd, F_SETOWN, getpid())
fcntl(fd, F_SETFL, FASYNC)
```

Other Useful Routines

`sigblock`

`sigsetmask`

`sigprocmask` — block signals in critical sections

`select`

`sigpause`

HyperText Transfer Protocol - HTTP

Application-level ASCII protocol used by the World-Wide Web (WWW)

ASCII request/MIME-like response

Request consists of `method, URL, and protocol version`

URL: Uniform Resource Locator - contains information on scheme (http), address of host, and address of page on host

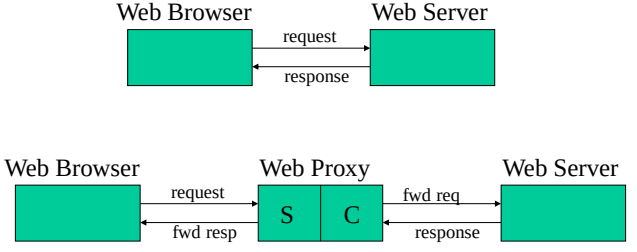
Port number can also be specified
DNS (Domain Name System) used to find IP address

HTML: HyperText Markup Language used to write web pages

HTTP Overview

- Hypertext Transfer Protocol (HTTP).
  - Deliver resources on the World Wide Web
  - HTML files, image files, query results etc
  - HTTP request format: *scheme://host:port/path*
  - Client/Server architecture
    - client: browser, server: web server
  - usually implemented over TCP/IP
  - stateless protocol
  - default port 80

WWW Architecture



HTTP Message Format

- Requests and responses are
  - similar
  - English-oriented readable text

```
<initial line, different for request vs. response>
Header1: value1
Header2: value2
Header3: value3
                                    // blank line (CRLF by itself)
<optional message body>
```

Initial Request Line

- Format: *method path protocol*
- Methods: *GET POST HEAD*
- Protocol HTTP/1.0 or HTTP/1.1
- Example

  GET /path/to/file/index.html HTTP/1.0

Initial Response Line (Status line)

- Format: *protocol status_code message*
- Protocol HTTP/1.0 or HTTP/1.1
- status code and message
  - 200 OK
  - 201 Created
  - 301 Moved Permanently
  - 302 Moved Temporarily
  - 400 Bad Request
  - 404 Not Found
  - 501 Not Implemented
  - 503 Service Unavailable

Header

- Provide information about the request or response
- Format:

  Header-Name: value
- Header name is not case-sensitive
- Any number of spaces or tabs may be between ":" and the value

Important Headers

- **Content-Type**: the MIME-type of the data in the body, e.g. text/html or image/gif.
- **Content-Length**: the number of bytes in the body.
- Others

• Client to server	•Server to client
•From	•Server
•User-Agent	•Last-Modified

Request and Response Example

Steps to access
http://www.somehost.com/path/file.htm

- The client opens a TCP connection to www.somehost.com:80
- The client sends request
- The server sends response
- The browser display file.htm

```
GET /path/file.html HTTP/1.0
From: someuser@jmarshall.com
User-Agent: Netscape/4.7
[blank line here]
```

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>Happy New Millennium!</h1>
(more file contents)
 ...
</body>
</html>
```

Live example

POST Method

- Submit data to servers
- Content-Type: application/x-www-form-urlencoded
- Content is URL-encoded

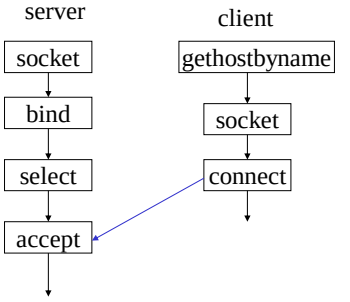
```
POST /path/script.cgi HTTP/1.0
From: frog@jmarshall.com
User-Agent: HTTPTool/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

home=Cosby&favorite+flavor=flies
```

Proxy Specialty

- Requests sent to proxies usually use complete URL, instead of just path
- Example

  GET http://host/path/file.html HTTP/1.0
- Well-behaved proxy should strip ***"http://host"*** before forwarding.

Socket Programming Steps



Example: http://www.ecst.csuchico.edu/~beej/guide/net/html/clientserver.html

Server handle multiple requests

- Multi-process
  - *fork, wait, wait_pid*
- select---Synchronous I/O Multiplexing *(example)*
- Multi-thread:

```
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
pthread_create(&id, &attr, svc_routine, param);
```

- Asynchronous I/O
  - *sigaction and SIGIO*

Disclaimer

- Parts of the lecture slides are adapted from and copyrighted by James Kurose and Keith Ross. The slides are intended for the sole purpose of instruction of computer networks at the University of Rochester. All copyrighted materials belong to their original owner(s).