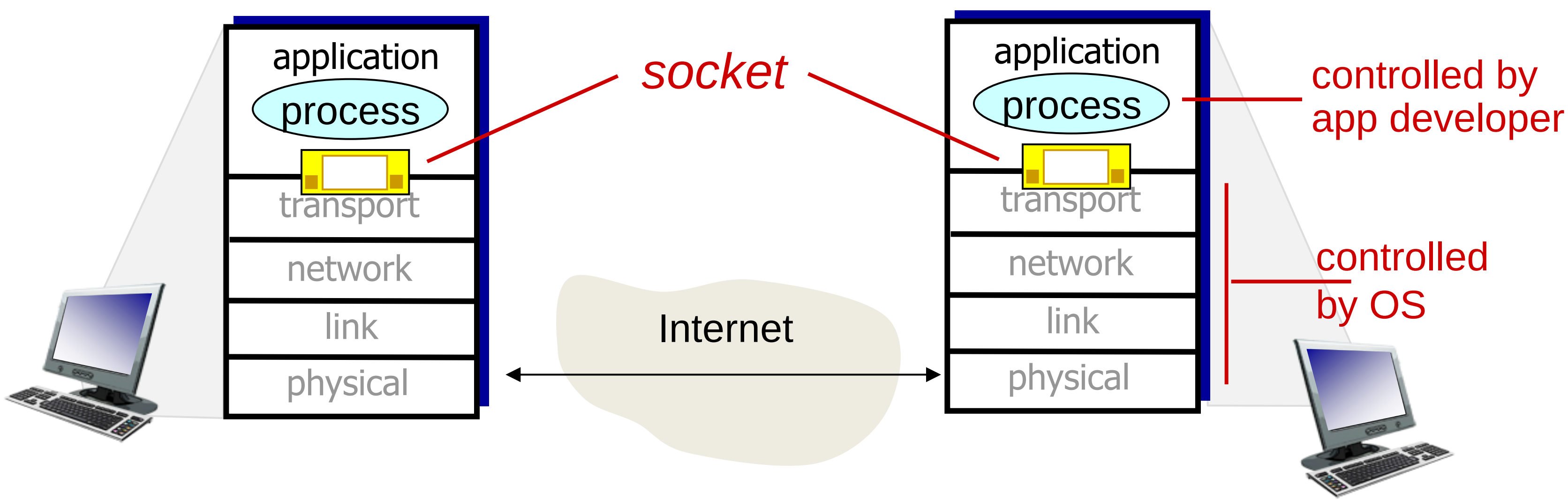
4. Socket Programming

2018 Spring

Yusung Kim
yskim525@skku.edu

Socket programming

- **goal:** learn how to build client/server applications that communicate using sockets
- **socket:** door between application process and end-end-transport protocol



Addressing processes

- To receive messages, process must have ***identifier***
- Host device has unique 32-bit IP address
- ***Identifier*** includes both **IP address** and **port numbers** associated with process on host.
  - Web server process : port number 80
  - Mail server process : port number 25

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

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 address 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

Client/server socket interaction: UDP

server

create socket, port= x:

serverSocket =
socket(AF_INET, **SOCK_DGRAM**)

read datagram from
serverSocket

write reply to
serverSocket
specifying client
address, port number

client

create socket:

clientSocket =
socket(AF_INET, **SOCK_DGRAM**)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: UDP client

```
from socket import *
serverIP = '1.1.1.1'
serverPort = 12000
```

create UDP socket

```
clientSocket = socket(PF_INET, SOCK_DGRAM)
msg = input('Input lowercase sentence: ')
```

Attach server IP, port to message; send into socket

```
clientSocket.sendto( msg.encode(), (serverIP, serverPort) )
```

read reply characters from socket into string

```
newMsg, serverAddress = clientSocket.recvfrom( 2048 )
```

print out received string and close socket

```
print ( newMsg.decode() )
clientSocket.close()
```

Example app: UDP server

```
from socket import *
serverPort = 12000
serverSocket = socket(PF_INET, SOCK_DGRAM)
serverSocket.bind( ('1.1.1.1', serverPort) )
print ('The server is ready to receive')

while True:
    message, clientAddress = serverSocket.recvfrom( 2048 )

    newMsg = message.decode().upper()

    serverSocket.sendto( newMsg.encode(), clientAddress )
```

create UDP socket

bind socket to local port number 12000

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*

- **server listens first**
  - server process must first be running
  - server must have created socket (door) to listen client's contact
- **client contacts server by**
  - Creating TCP socket with IP address, port number of server process
  - *when client creates socket:* client TCP establishes connection to server TCP
- ***server creates a new socket***
  - a new socket to communicate with an incoming client
  - still need to listen with the existing socket
  - may need to talk with multiple clients

Client/server socket interaction: TCP

server

create socket,
ip=**x,** port=**y**, 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 ip=**x**, port=**y**
clientSocket = socket()

clientSocket.connect(x, y)

send request using clientSocket

read reply from clientSocket

close clientSocket

Example app: TCP client

create TCP socket for server, remote port 12000

No need to attach server name, port

```
from socket import *
serverIP = '1.1.1.1'
serverPort = 12000
clientSocket = socket(PF_INET, SOCK_STREAM)
clientSocket.connect( (serverIP, serverPort) )

msg = input( 'Input lowercase sentence: ' )
clientSocket.send( msg.encode() )

newMsg = clientSocket.recv(1024)
print ('From Server: ', newMsg.decode())
clientSocket.close()
```

Example app: TCP server

create TCP listening socket

server begins listening for incoming TCP requests

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

read bytes from socket

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

```
from socket import *
serverPort = 12000
serverSocket = socket(PF_INET, SOCK_STREAM)
serverSocket.bind( ('1.1.1.1', serverPort) )
serverSocket.listen( 5 )
print( 'The TCP server is ready to receive.' )
while True:
    connectionSocket, addr = serverSocket.accept()
    msg = connectionSocket.recv(1024).decode()
    newMsg = msg.upper()
    connectionSocket.send( newMsg.encode() )
    connectionSocket.close()
```

Socket Programming in C

Creating a socket

```
int sockfd = socket(domain, type, protocol);
```

– The socket number returned is the socket descriptor for the newly created socket.

- `int sockfd = socket (PF_INET, SOCK_STREAM, 0); // TCP`
- `int sockfd = socket (PF_INET, SOCK_DGRAM, 0);  // UDP`

Client-Serve Model with TCP

Server

- Passive open
- Prepares to accept connection, does not actually establish a connection.

Server invokes

```
int bind (int socket, struct sockaddr *address, int addr_len)
int listen (int socket, int backlog)
int accept (int socket, struct sockaddr *address, int *addr_len)
```

Client-Serve Model with TCP

Bind

- Binds the newly created socket to the specified address. i.e. the network address of the server
- Address is a data structure which combines IP and port.

Listen

- Defines how many connections can be pending on the specified socket.

Client-Serve Model with TCP

Accept

- Blocking operation : does not return until a remote participant has established a connection.
- Returns a new socket that corresponds to the new established connection and the address argument contains the remote peer’s address .

Client-Serve Model with TCP

Client

- Application performs active open
- It says who it wants to communicate with

Client invokes

```
int connect (int socket, struct sockaddr *address, int addr_len)
```

Connect

- Does not return until TCP has successfully established a connection at which application is free to begin sending data.

Client-Serve Model with TCP

In practice

- The server listens on a well-known port.
- The client specifies the server's address and port number.
- A client does not care which port it uses for itself,
  the OS simply selects an unused one.

Client-Serve Model with TCP

Once a connection is established, the application process invokes two operation.

```
int send (int socket, char *msg, int msg_len, int flags)

int recv (int socket, char *buff, int buff_len, int flags)
```

Example Application: Server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main(int argc, char *argv[])
{
    int serv_sock;
    int clnt_sock;

    struct sockaddr_in serv_addr;
    struct sockaddr_in clnt_addr;
    int clnt_addr_size;

    char message[] = "hello World!";

    if(argc !=2 )
    {
        printf("Usage : %s <port>\n", argv[0]);
        return -1;
    }
```

Example Application: Server

```
    serv_sock = socket(PF_INET, SOCK_STREAM, 0);

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(atoi(argv[1]));

    if (bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1)
    {
       puts("bind error");
       return -1;
    }

    listen(serv_sock, 5);
    puts("wait to accept");
    clnt_addr_size = sizeof(clnt_addr);
    clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size);

    write(clnt_sock, message, sizeof(message));
    close(clnt_sock);
    close(serv_sock);

    return 0;
}
```

Example Application: Client

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main(int argc, char *argv[])
{
    int clnt_sock;
    struct sockaddr_in serv_addr;

    char message[100];
    int str_len;

    if(argc !=3 )
    {
        printf("Usage : %s <IP> <port>\n", argv[0]);
        return -1;
    }

   clnt_sock = socket(PF_INET, SOCK_STREAM, 0);
```

Example Application: Client

```
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    serv_addr.sin_port = htons(atoi(argv[2]));

    puts("start to connect");
    if ( connect( clnt_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr) ) < 0 ) {
        puts("connect error");
        return -1;
    }

    puts("wait to read");
    str_len = read(clnt_sock, message, sizeof(message)-1);

    if(str_len == -1)
        return -1;

    printf("Message from server : %s\n", message);
    close(clnt_sock);

    return 0;
}
```

How to handle multiple connections?

- Where do we get incoming data?
  - stdin (typically keyboard input)
  - network stream, datagram from sockets
  - asynchronous arrival, program doesn’t know when data will arrive

- Basically, socket related functions are in blocking mode.
  - `recv()` to read from a stream, control isn't returned to your program until data is read from the remote site.
  - The same is true for `connect()`, `accept()`, etc. When you run them, the program is "blocked" until the operation is complete.

Multiplexing Network I/O

	fork / thread	non-blocking function ex) select( )
Approach	Create a new parallel context for every new connection	monitor multiple file descriptors/sockets
Advantages	Simple to write programs using blocking calls	Explicit control flow
Disadvantages	Context switching overheads & High memory consumption	not scale to large number of file descriptors/sockets

More Advanced Approaches

- epoll is a Linux kernel system call for a scalable I/O event notification mechanism.

- IOCP (Input/output completion port) is an API for processing multiple asynchronous I/O requests on Windows.