5: Socket Programming

Last Modified:
2/10/2003 2:38:37 PM

Socket programming

Goal: learn how to build client/server application that communicate using sockets

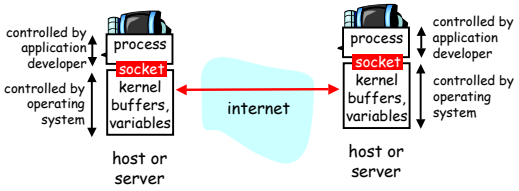
Socket API
- introduced in BSD4.1 UNIX, 1981
- Sockets are explicitly created, used, released by applications
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

socket

a *host-local, application-created/owned, OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process

Sockets

Socket: a door between application process and end-end-transport protocol (UCP or TCP)



Languages and Platforms

Socket API is available for many languages on many platforms:
- C, Java, Perl, Python,…
- *nix, Windows,…

Socket Programs written in any language and running on any platform can communicate with each other!

Writing communicating programs in different languages is a good exercise

Socket Programming is Easy

- Create socket much like you open a file
- Once open, you can read from it and write to it
- Operating System hides most of the details

Decisions

- Before you go to write socket code, decide
  - Do you want a TCP-style reliable, full duplex, connection oriented channel? Or do you want a UDP-style, unreliable, message oriented channel?
  - Will the code you are writing be the client or the server?
    - Client: you assume that there is a process already running on another machines that you need to connect to.
    - Server: you will just start up and wait to be contacted

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 client-local 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 client
  - Frees up incoming port
  - allows server to talk with multiple clients

application viewpoint

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

Pseudo code TCP client

Create socket, connectSocket

Do an active connect specifying the IP address and port number of server

Read and Write Data Into connectSocket to Communicate with server

Close connectSocket

Pseudo code TCP server

Create socket (doorbellSocket)

Bind socket to a specific port where clients can contact you

Register with the kernel your willingness to listen that on socket for client to contact you

Loop

Accept new connection (connectSocket)

Read and Write Data Into connectSocket to Communicate with client

Close connectSocket

End Loop

Close doorbellSocket

Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

```
        BufferedReader inFromServer =
          new BufferedReader(new
          InputStreamReader(clientSocket.getInputStream()));

        sentence = inFromUser.readLine();

        outToServer.writeBytes(sentence + '\n');

        modifiedSentence = inFromServer.readLine();

        System.out.println("FROM SERVER: " + modifiedSentence);

        clientSocket.close();

    }
}
```

Create input stream attached to socket

Send line to server

Read line from server

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
  {
    String clientSentence;
    String capitalizedSentence;

    ServerSocket welcomeSocket = new ServerSocket(6789);

    while(true) {

      Socket connectionSocket = welcomeSocket.accept();

      BufferedReader inFromClient =
        new BufferedReader(new
        InputStreamReader(connectionSocket.getInputStream()));
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '\n';
outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

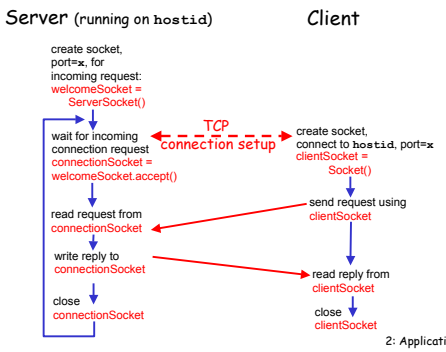
Create output stream, attached to socket

Read in line from socket

Write out line to socket

End of while loop, loop back and wait for another client connection

Client/server socket interaction: TCP (Java)



Example: C client (TCP)

Warning: Should check return codes of major functions!! Omitted for space here

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

Int main(int argc, char **argv) {

    int connectionSocket;
    char sentence[MAX_LINE];
    char modifiedSentence[MAX_LINE];
    struct hostent *hp;

    connectionSocket = socket (PF_INET, SOCK_STREAM,0);

    /* translate host name into peer's IP address */
    hp = gethostbyname("hostname");
```

Create client socket

Resolve hostname Of server to IP adresss

Example: C client (TCP), cont.

```
    bzero((char *) &sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(6789r);
    bcopy(hp->h_addr, (char *) &sin.sin_addr,
        hp->h_length);

    connect(connectSocket,
        (struct sockaddr *) &sin, sizeof(sin))

    fgets(sentence, MAXLINE, stdin);
    buff[MAXLINE-1] = '\0';
    write(connectSocket, sentence, strlen(sentence)+1, 0);

    read(connectionSocket, modifiedSentence,
        sizeof(modifiedSentence), 0);
    fprintf(stderr, "FROM SERVER: %s \n",
                modifiedSentence);
    close(connectSocket);
}
```

Connect to server

Send line to server

Read line from server

Example: C server (TCP)

Warning: Should check return codes of major functions!! Omitted for space here

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

Int main(int argc, char **argv) {
    int welcomeSocket, connectionSocket;
    char clientSentence[MAX_LINE];;
    struct sockaddr_in    servaddr;

    welcomeSocket = socket(AF_INET, SOCK_STREAM, 0);

    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(6789);

    bind(lwelcomeSocket, (struct sockaddr *) &servaddr, \
        sizeof(servaddr));
    listen(welcomeSocket , LISTENQ);
```

Create welcoming socket at port 6789

Example: C server (TCP), cont

```
    for ( ; ; ) {

    connectionSocket =
        accept(welcomeSocket , (struct sockaddr *) NULL, NULL);

    bytesRead = read(connectionSocket, clientSentence, MAXLINE);

    //would have to write the capitalize procedure
    capitalize(clientSentence);

    write(connectionSocket,  clientSentence, MAXLINE);

    close(connectSocket);
    }
  close(welcomeSocket);
}
```

Wait, on welcoming socket for contact by client

Read in line from socket

Write out line to socket

End of while loop, loop back and wait for another client connection

TCP Server vs Client

- Server waits to accept connection on well known port
- Client initiates contact with the server
- Accept call returns a new socket for this client connection, freeing welcoming socket for other incoming connections
- Read and write only (addresses implied by the connection)

Concurrent TCP Servers

- What good is the doorbell socket? Can't accept new connections until call accept again anyway?
- Benefit comes in ability to hand off processing to another process
  - Parent process creates the "door bell" or "welcome" socket on well-known port and waits for clients to request connection
  - When a client does connect, fork off a child process to handle that connection so that parent process can return to waiting for connections as soon as possible
- Multithreaded server: same idea, just spawn off another thread rather than a full process
  - Threadpools?

Pseudo code concurrent TCP server

```
Create socket doorbellSocket
Bind
Listen
Loop
   Accept the connection, connectSocket
   Fork
   If I am the child
        Read/Write connectSocket
        Close connectSocket
        exit
EndLoop
Close doorbellSocket
```


Backlog

- Many implementations do allow a small fixed number (~5) of unaccepted connections to be pending, commonly called the backlog
- This helps avoid missing connections while process not sitting in the accept call

Socket programming with UDP

UDP: very different mindset than TCP

- no connection just independent messages sent
- no handshaking
- sender explicitly attaches IP address and port of destination
- server must extract IP address, port of sender from received datagram to know who to respond to

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

application viewpoint

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

Pseudo code UDP server

Create socket

Bind socket to a specific port where clients can contact you

Loop

(Receive UDP Message from client x)+

(Send UDP Reply to client x)*

Close Socket

Pseudo code UDP client

Create socket

Loop

(Send Message To Well-known port of server)+

(Receive Message From Server)

Close Socket

Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
   public static void main(String args[]) throws Exception
   {
Create input stream
     BufferedReader inFromUser =
       new BufferedReader(new InputStreamReader(System.in));
Create client socket
     DatagramSocket clientSocket = new DatagramSocket();
Translate hostname to IP address using DNS
     InetAddress IPAddress = InetAddress.getByName("hostname");

     byte[] sendData = new byte[1024];
     byte[] receiveData = new byte[1024];

     String sentence = inFromUser.readLine();
     sendData = sentence.getBytes();
```


Example: Java client (UDP), cont.

```
Create datagram with data-to-send, length, IP addr, port
     DatagramPacket sendPacket =
       new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
Send datagram to server
     clientSocket.send(sendPacket);
     DatagramPacket receivePacket =
       new DatagramPacket(receiveData, receiveData.length);
Read datagram from server
     clientSocket.receive(receivePacket);
     String modifiedSentence =
       new String(receivePacket.getData());

     System.out.println("FROM SERVER:" + modifiedSentence);
     clientSocket.close();
     }
}
```


Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
   {
Create datagram socket at port 9876
     DatagramSocket serverSocket = new DatagramSocket(9876);

     byte[] receiveData = new byte[1024];
     byte[] sendData  = new byte[1024];

     while(true)
      {
Create space for received datagram
       DatagramPacket receivePacket =
         new DatagramPacket(receiveData, receiveData.length);
Receive datagram
       serverSocket.receive(receivePacket);
```


Example: Java server (UDP), cont

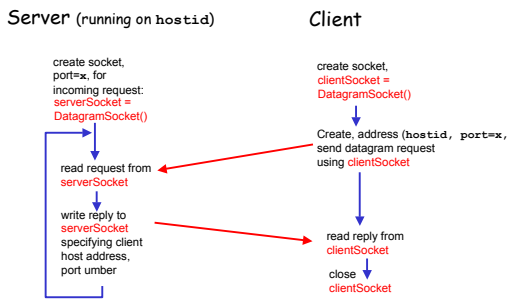
```
                String sentence = new String(receivePacket.getData());
Get IP addr port #, of sender
                InetAddress IPAddress = receivePacket.getAddress();
                int port = receivePacket.getPort();

                String capitalizedSentence = sentence.toUpperCase();

                sendData = capitalizedSentence.getBytes();
Create datagram to send to client
                DatagramPacket sendPacket =
                  new DatagramPacket(sendData, sendData.length, IPAddress,
                                     port);
Write out datagram to socket
                serverSocket.send(sendPacket);
             }
          }
       }
```

End of while loop, loop back and wait for another datagram

Client/server socket interaction: UDP



Example: C client (UDP)

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

Int main(int argc, char **argv) {

     int socket;
     char sentence[MAX_LINE];
     char modifiedSentence[MAX_LINE];
     struct hostent *hp;
     struct sockaddr_in cliAddr, remoteServAddr;

     socket= socket(AF_INET,SOCK_DGRAM,0);

     /* translate host name into peer's IP address */
     hp = gethostbyname("hostname");
```

Create socket

Translate hostname to IP address using DNS

Example: C client (UDP), cont.

```
     /* bind any port */
     cliAddr.sin_family = AF_INET;
     cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
     cliAddr.sin_port = htons(0);

     bind(socket, (struct sockaddr *) &cliAddr, sizeof(cliAddr));

     remoteServAddr.sin_family = h->h_addrtype;
     memcpy((char *) &remoteServAddr.sin_addr.s_addr,
              h->h_addr_list[0], h->h_length);
     remoteServAddr.sin_port = htons(9876);

     sendto (socket, sentence, MAX_LINE, 0,
              (struct sockaddr *) &remoteServAddr,
              sizeof(remoteServAddr));

     recvfrom(socket, modifiedSentence, MAX_LINE, 0,
              (struct sockaddr *) &remoteServAddr, &remoteServLen);

     fprintf(stderr, "FROM SERVER: %s \n",  modifiedSentence);
     close(socket);
}
```

Register to receive datagrams on this socket

Send datagram to server

Read datagram from server

Example: C server (UDP)

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

Int main(int argc, char **argv) {

     int socket;
     char clientSentence[MAX_LINE];
     struct sockaddr_in svrAddr, cliAddr;

     socket= socket(AF_INET,SOCK_DGRAM,0);
```

Create UDP socket

Example: C server (UDP), cont.

```
     /* bind any port */
     svrAddr.sin_family = AF_INET;
     svrAddr.sin_addr.s_addr = htonl(INADDR_ANY);
     svrAddr.sin_port = htons(9876);

     bind(socket, (struct sockaddr *) &svrAddr, sizeof(svrAddr));

     for (;;){
          recvfrom(socket, clientSentence, MAX_LINE, 0,
                   (struct sockaddr *) &cliAddr, &cliLen);

          //would have to write the capitalize procedure
          capitalize(clientSentence);

          sendto (socket, clientSentence, MAX_LINE, 0,
                   (struct sockaddr *) &cliAddr,
                   sizeof(cliAddr));
     }
     close(socket);
}
```

Register to receive datagrams on this socket

Read datagram from client
Extract return address

Reply to the client

End of for loop, loop back and wait for another datagram

UDP Server vs Client

- Server has a well-known port number
- Client initiates contact with the server
- Less difference between server and client code than in TCP
  - Both client and server bind to a UDP socket
  - Not accept for server and connect for client
- Client send to the well-known server port; server extracts the client's address from the datagram it receives

TCP vs UDP

- TCP can use read/write (or recv/send) and source and destination are implied by the connection; UDP must specify destination for each datagram
  - Sendto, recevfrom include address of other party
- TCP server and client code look quite different; UDP server and client code vary mostly in who sends first

Java vs C

- Java hides more of the details
  - new ServerSocket of Java = socket, bind and listen of C
  - new Socket hides the getByName (or gethostbyname) of C; Unable to hide this in the UDP case though
  - Socket API first in C for BSD; more options and choices exposed by the interface than in Java ?

Note

- Examples were simple code snippets
- To fit on a slide, I omitted important things like:
  - Testing each connect, sendto and recvfrom for errors
  - In UDP case, handling the case of packet loss
- The behavior of many of these functions can be "customized" with various socket options
  - In C, use setsockopt/getsockopt
  - In Java, use setOption/getOption

Socket Programming in the Real World

- Download some open source implementations of network applications
  - Web browsers (Mosaic, Jazilla)
  - DNS Servers and resolvers (BIND)
  - Email clients/servers (sendmail, qmail, pine)
  - telnet
- Can you find the socket code? The protocol processing? What percentage of the code is it? What does the rest of the code do?

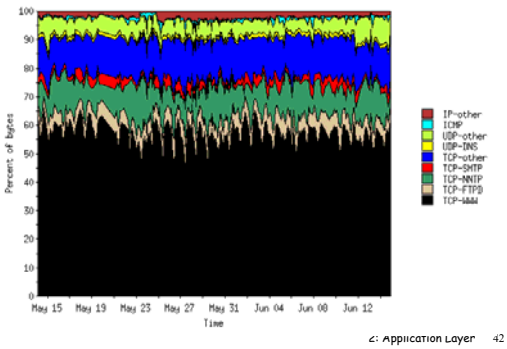
On to the transport layer…

- Important to remember that we build transport services to support applications
- Transport services are a means to an end

Outtakes

Real Internet Traffic Analysis

Credit: CAIDA (1999)



Transport service requirements of common apps

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no?
e-mail	no loss	elastic	no
Web documents	loss-tolerant	elastic	no?
real-time audio/video	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps up	yes, 100's msec
news	No loss ?	elastic	no

Internet apps: their protocols and transport protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	smtp [RFC 821]	TCP
remote terminal access	telnet [RFC 854]	TCP
Web	http [RFC 2068]	TCP
file transfer	ftp [RFC 959]	TCP
streaming multimedia	RTP, proprietary (e.g. RealNetworks)	UDP
remote file server	NFS	TCP or UDP
Internet telephony	proprietary (e.g., Vocaltec)	typically UDP
DNS	DNS	typically UDP, TCP