Socket Programming

Rohan Murty
Hitesh Ballani

Last Modified:
2/8/2004 8:31:51 AM

Slides adapted from Prof. Matthews’ slides from 2003SP
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)

controlled by application developer
controlled by operating system

controlled by application developer
controlled by operating system

host or server
host or server
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
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 (serverSocket)
- 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 serverSocket
Example: Java client (TCP)

```java
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());

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

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

        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```
Example: Java client (TCP), cont.

Create input stream attached to socket

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

sentence = inFromUser.readLine();

Send line to server

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

Read line from server

modifiedSentence = inFromServer.readLine();

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

clientSocket.close();
Example: Java server (TCP)

```java
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()));

            ServerSocket welcomeSocket = new ServerSocket(6789);

            while(true) {
                Socket connectionSocket = welcomeSocket.accept();
                BufferedReader inFromClient =
                    new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
            }
        }
    }
}
```
Example: Java server (TCP), cont

Create output stream, attached to socket

```
DataOutputStream outToClient =
new DataOutputStream(connectionSocket.getOutputStream());
```

Read in line from socket

```
clientSentence = inFromClient.readLine();
```

Write out line to socket

```
capitalizedSentence = clientSentence.toUpperCase() + '\n';
outToClient.writeBytes(capitalizedSentence);
```

End of while loop, loop back and wait for another client connection
Client/server socket interaction: TCP (Java)

Server (running on hostid)

1. create socket, port=x, for incoming request:
   welcomeSocket = ServerSocket()

2. wait for incoming connection request
   connectionSocket = welcomeSocket.accept()

3. read request from connectionSocket

4. write reply to connectionSocket

5. close connectionSocket

Client

1. create socket, connect to hostid, port=x
   clientSocket = Socket()

2. send request using clientSocket

3. read reply from clientSocket

4. close clientSocket
Queues

• We just saw a simple example, with one socket on the server handling incoming connections
• While the server socket is busy, incoming connections are stored in a queue until it can accept them
• Most systems maintain a queue length between 5 and 50
• Once the queue fills up, further incoming connections are refused until space in the queue opens up
• This is a problem in a situation where our server has to handle many concurrent incoming connections. Example: HTTP servers
  – Solution? Use concurrency
Concurrent TCP Servers

• 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?
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

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)

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

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        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

```java
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```java
clientSocket.send(sendPacket);
```

Read datagram from server

```java
DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence = new String(receivePacket.getData());

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

clientSocket.close();
```
```
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while(true) {
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
        }
    }
}
Example: Java server (UDP), cont

```java
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
```

Get IP addr port #, of sender
Create datagram to send to client
Write out datagram to socket
End of while loop, loop back and wait for another datagram
Client/server socket interaction: UDP

Server (running on **hostid**)

Client

create socket, port=x, for incoming request:
serverSocket = DatagramSocket()

create socket, clientSocket = DatagramSocket()

Create, address (hostid, port=x), send datagram request using clientSocket

read request from serverSocket

write reply to serverSocket specifying client host address, port number

read reply from clientSocket

close clientSocket
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, recvfrm include address of other party
- TCP server and client code look quite different; UDP server and client code vary mostly in who sends first
Byte ordering

- Big Endian byte-order

The byte order for the TCP/IP protocol suite is big endian.
Byte-Order Transformation

Host byte order

16-bit → 32-bit

16-bit

htonhs

ntohl

32-bit

htonl

Network byte order

u_short  htons ( u_short  host_short );

u_short  ntohs ( u_short  network_short );

u_long  htonl ( u_long  host_long );

u_long  ntohl ( u_long  network_long );
Some Definitions

• Internet Address Structure

```c
struct in_addr
{
    in_addr_t s_addr;
};
```

`in_addr_t` is defined as a long on Linux machines, implying 32 bit addresses!
Socket address structure

```
struct sockaddr_in
{
    u_char sin_len;
    u_short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
} ;
```
Address Transformation

```c
int    inet_aton ( const char *strptr , struct in_addr *addrptr );
char   *inet_ntoa ( struct in_addr inaddr );
```
Socket Types

Application program

Stream socket interface
TCP

Datagram socket interface
UDP

Raw socket interface

IP

Physical and data link layers
**Server**

1. Create transport endpoint for incoming connection request: `socket()`

2. Assign transport endpoint an address: `bind()`

3. Announce willing to accept connections: `listen()`

4. Block and Wait for incoming request: `accept()`

5. Wait for a packet to arrive: `read()`

6. Formulate reply (if any) and send: `write()`

7. Release transport endpoint: `close()`

**Client**

1. Create transport endpoint: `socket()`

2. Assign transport endpoint an address (optional): `bind()`

3. Determine address of server

4. Connect to server: `connect()`

4. Formulate message and send: `write()`

5. Wait for packet to arrive: `read()`

6. Release transport endpoint: `close()`

**CONNECTION-ORIENTED SERVICE**
Connectionless Service (UDP)

Server

1. Create transport endpoint: `socket()`

2. Assign transport endpoint an address: `bind()`

3. Wait for a packet to arrive: `recvfrom()`

4. Formulate reply (if any) and send: `sendto()`

5. Release transport endpoint: `close()`

Client

1. Create transport endpoint: `socket()`

2. Assign transport endpoint an address (optional): `bind()`

3. Determine address of server

4. Formulate message and send: `sendto()`

5. Wait for packet to arrive: `recvfrom()`

6. Release transport endpoint: `close()`
Procedures That Implement The Socket API

Creating and Deleting Sockets

- **fd=socket**(protofamily, type, protocol)
  Creates a new socket. Returns a file descriptor (fd). Must specify:
  - the protocol family (e.g. TCP/IP)
  - the type of service (e.g. STREAM or DGRAM)
  - the protocol (e.g. TCP or UDP)

- **close**(fd)
  Deletes socket.
  For connected STREAM sockets, sends EOF to close connection.
Procedures That Implement The Socket API
Putting Servers “on the Air”

- **bind**(fd,laddress,laddresslen)
  Used by server to establish port to listen on. When server has >1 IP addr, can specify “IF_ANY”, or a specific one

- **listen** (fd, queuesize)
  Used by connection-oriented servers only, to put server “on the air”
  Queuesize parameter: how many pending connections can be waiting
• afd = accept (lfd, caddress, caddresssl)  
Used by connection-oriented servers to accept one new connection

• There must already be a listening socket (lfd)
• Returns afd, a new socket for the new connection, and
• The address of the caller (e.g. for security, log keeping, etc.)
Procedures That Implement The Socket API
How Clients Communicate with Servers?

- **connect** (fd, saddress, saddreslen)

  Used by connection-oriented clients to connect to server
  - There must already be a socket bound to a connection-oriented service on the fd
  - There must already be a listening socket on the server
  - You pass in the address (IP address, and port number) of the server.

  Used by connectionless clients to specify a “default send to address”
  - Subsequent “sends” don’t have to specify a destination address
Procedures That Implement The Socket API

How Clients Communicate with Servers? (TCP)

• int \texttt{write} (fd, data, length)
  
  Used to send data
  
  • write is the “normal” write function; can be used with both files and sockets

• int \texttt{read} (fd, data,length)
  
  Used to receive data… parameters are similar!

NOTE: both functions can return a value less than the length
Procedures That Implement The Socket API
How Clients Communicate with Servers (UDP)

• **int sendto** (fd, data, length, flags, destaddress, addresslen)
  Used to send data.
  • Connectionless socket, so we need to specify the dest address

• **int recvfrom** (fd, data, length, flags, srcaddress, addresslen)
  Used to receive data… parameters are similar, but in reverse
Concurrent Server: TCP (C/C++)

Server (running on hostid)

- create socket, port=x, for incoming request:
  - socket(), bind(), listen()

  
  wait for incoming connection request
  - accept()

  
  read and process
  - read()
  - reply
  - write()

  
  close
  - close()

TCP connection setup

client

- create socket, connect to hostid, port=x
  - socket(), connect()

  
  send request
  - write()

  
  read reply from
  - read()

  
  close
  - close()
Non-blocking I/O

• By default, `accept()`, `recv()`, etc block until there’s input

• What if you want to do something else while you’re waiting?

• We can set a socket to not block (i.e. if there’s no input an error will be returned)

• … or, we can tell the kernel to let us know when a socket is ready, and deal with it only then
non-blocking/select

- The host uses \texttt{select()} to get the kernel to tell it when the peer has sent a message that can be \texttt{recv()}’d

- Can specify multiple sockets on which to wait
  -- \texttt{select} returns when one or more sockets are ready
  -- operation can time out!
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?
AIM: Write a program (referred to as the IP box) that opens four sockets, two TCP and two UDP

2 TCP SOCKETS:

1. A receive-config socket: IP BOX acts as a Server (must be bound to a port you have to find, and the interface IP address)

2. A send-config socket: IP BOX acts as a receiever
• 2 UDP SOCKETS

• App -- acts as the interface between the IP layer and the application

• Iface – represents the network interface

• Both must be bound to an used port and the interface address
IP BOX OPERATION

- Send-config sockets connects to the Test Box and sends a “ready-to-test” command

- The Test Box then connects to recv-config socket and send a ‘\n’ terminated command which must be echoed

- The Test Box then sends UDP packets to app and iface sockets which must be echoed (Note: If the Test Box does not receive your echo, it retransmits the packet)
• On receiving both the echoes, the Test Box sends a “send-stat” command to the send-config socket

• The IP box sends a “list-of-stats”

• The Test Box then sends an exit message (during final test, this will have a 40 character hex string representing a hashed timestamp, which your program must RECORD!)