Network Programming using sockets

Distributed Software Systems
Prof. Sanjeev Setia
APIs for TCP/IP

- TCP/IP is a protocol designed to operate in multi-vendor environment
- interface between TCP/IP and applications loosely specified
- application interfaces
  - BSD UNIX: socket interface
  - AT&T: TLI interface
- TCP/IP software inside kernel invoked by system calls
- UNIX I/O facilities extended with TCP/IP specific calls
The Socket Interface

- provides functions that support network communication using many possible protocols
  - PF_INET is one protocol family supported by sockets
  - TCP and UDP are protocols in PF_INET family
- socket is the abstraction for network communication
- a socket is identified by socket descriptor
- system data structure for socket
  - family (e.g., PF_INET)
  - service (e.g., SOCK_STREAM)
  - Local IP address, Local Port
  - Remote IP address, Remote Port
- passive socket: socket used by a server to wait for incoming connections; active socket: socket used by client to initiate a connection
Endpoint Addresses

- TCP/IP protocols define a communication endpoint to consist of an IP address and a protocol port number

- other protocol families have other definitions

- socket abstractions supports the concept of *address family* which allows different protocols to have their own address representations

- TCP/IP protocols use a single address representation with address family denoted by AF_INET
Endpoint Addresses cont’d

• structure for AF_INET addresses

    struct sockaddr_in {
        u_char sin_len; /* total length */
        u_short sin_family; /* type of address */
        u_short sin_port; /* protocol port number */
        struct in_addr sin_addr; /* IP address */
        char sin_zero[8]; /* unused (set to zero) */
    };

• if program using mixture of protocols, programmer must be careful since not all addresses have the same size
System Calls

- socket
  - used to create new socket
  - arguments: protocol family (e.g. PF_INET), protocol or service (i.e., stream or datagram)
  - returns socket descriptor

- connect:
  - client calls connect to establish an active connection to the server
  - argument to connect specifies remote endpoint

- write
  - servers and clients use write to send data across a TCP connection
  - arguments: socket descriptor, address of data, length of data
System Calls cont’d

• read
  – used to receive data from a TCP connection
  – arguments: socket, buffer, length of buffer
  – read blocks if no data; if more data than fits in buffer, it only extracts enough to fill the buffer; if less than buffer length, it extracts all the data and returns number of bytes read

• read and write can also be used with UDP but different behavior

• close: used to deallocate socket; deleted when last process that is using socket does a close

• bind
  – used to specify a local endpoint address for a socket
  – uses sockaddr_in structure
System Calls cont’d

- **listen**
  - used by connection-oriented servers to put socket in passive mode
  - arguments: socket, size of queue for socket connection requests

- **accept**
  - creates a new socket for each connection request
  - returns descriptor of new socket to its caller

- **UDP calls:**
  - send, sendto, sendmsg
  - recv, recvfrom, recvmsg
Integer Conversion

- standard representation for binary integers used in TCP/IP protocol headers: network byte order, MSB first

- e.g. the protocol port field in struct sockaddr_in uses network byte order

- host’s integer representation maybe different

- conversion routines: htons, htonl, ntohl, ntohs should be used for portability
Client Software

conceptually simpler than servers because

- do not have to handle concurrent interactions with multiple servers
- usually not privileged software \( \Rightarrow \) don’t have to be as careful
- no authentication, protection, etc.
Locating the server

server’s IP address and port number needed

• can be specified as a constant in the program

• have the user specify it as an argument when invoking client

• read from a file on disk

• use a protocol to find the server (e.g. a broadcast message to which servers respond)
Parsing address argument

- address argument typically is a hostname like cs.gmu.edu or IP address in dotted decimal notation like 129.174.29.34

- need to specify address using structure sockaddr_in

- library routines inet_addr and gethostbyname used for conversions

```c
struct hostent {
    char *hname;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
#define h_addr h_addr_list[0];
```
EXAMPLE:

```c
struct hostent *hptr;
char *name = "cs.gmu.edu";

if ( hptr = gethostbyname(name) ) {
    /* IP address is in hptr->h_addr */
} else {
    /* handle error */
}
```

- `inet_addr` converts dotted decimal IP address into binary
• looking up a well known port by name

• struct servent defined in *netdb.h* in the same way as struct hostent

```c
    struct servent *sptr;

    if (sptr = getservbyname("smtp","tcp")){
        /* port number is now in sptr->s_port */
    } else {
        /* handle error */
    }
```

• NOTE: `getservbyname` returns protocol port in network byte order
• looking up a protocol by name

• struct protoent defined in *netdb.h*

```c
struct protoent *pptr;

if (pptr = getprotobynamel(‘udp’)){
   /* official protocol number is in pptr->p_proto */
} else {
   /* handle error */
}
```
TCP client algorithm

1. Find IP address and protocol number of server

2. allocate a socket

3. specify that the connection needs an arbitrary, unused protocol port on local machine and allow TCP to select one

4. Connect the socket to the server

5. Communicate with the server using application-level protocol

6. Close the connection
TCP client cont’d

• Allocating a socket

```c
#include <sys/types.h>
#include <sys/socket.h>

int s; /* socket descriptor */

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

• Choosing a local port number
  – conflicts have to be avoided
  – happens as a side-effect to connect call

• choosing a local IP address
  – a problem for hosts connected to multiple networks
  – chosen automatically by TCP/IP at time of connection
Connecting a TCP socket to a server

retcode = connect(s, remaddr, remaddrlen)

- connect performs four tasks
  1. tests specified socket is valid and not already connected
  2. fills in remote address in socket from second argument
  3. chooses a local endpoint address for socket (if it does not have one)
  4. initiates a connection and returns value to the caller
Communicating with the server using TCP: Example

```c
#define BLEN 120
char *req = 'request of some sort';
char buf[BLEN];
char *bptr;
int n;
int buflen;

bptr = buf;
buflen = BLEN;

/* send request */
write(s, req, strlen(req));

/* read response (may come in several pieces) */
while ((n = read(s, bptr, buflen) > 0) {
  bptr += n;
  buflen -= n;
}
```
Closing a TCP connection

- partial close needed because client may not know when all the data from the server has arrived and server may not know if client will send another request

- shutdown call

  ```
  errcode = shutdown(s, direction);
  ```

- direction = 0: no further input, 1: no further output, 2: shutdown in both directions
Programming a UDP client

1. Find IP address and protocol number of server

2. Allocate a socket

3. Specify that the connection needs an arbitrary, unused protocol port on local machine and allow UDP to select one

4. Specify the server to which messages must be sent

5. Communicate with the server using application-level protocol

6. Close the socket
Connected and Unconnected UDP sockets

- with UDP, connected sockets do not mean a “connection” was established
- connected sockets $\Rightarrow$ server specified once
- unconnected sockets $\Rightarrow$ server specified each time
- read and write: message transfer NOT streams
- close does not inform remote endpoint of any actions
- **UDP is unreliable**
Examples

- TCP and UDP clients for services
  - DAYTIME
  - TIME
  - ECHO

- connectTCP and connectUDP procedures invoke connectsock
Issues in Server Design

• Concurrent vs iterative servers: handle multiple requests concurrently or one after the other?

• Connection-oriented vs connection-less servers: TCP or UDP?

• Stateful vs stateless servers
Iterative, connection-oriented server

- Algorithm

1. Create a socket and bind to the well-known address for the service being offered
2. Place the socket in passive mode
3. Accept the next connection request from the socket, and obtain a new socket for the connection
4. Repeatedly read a request from the client, formulate a response, and send a reply back to the client according to the application protocol
5. When finished with a particular client, close the connection and return to step 3 to accept a new connection

- servers should specify INADDR_ANY as internet address while binding

- needed for hosts with multiple IP addresses
Iterative, connection-less servers

- Algorithm
  1. Create a socket and bind to the well-known address for the service being offered
  2. Repeatedly read the next request from a client, formulate a response, and send a reply back to the client according to the application protocol

- cannot use connect (unlike clients)

- use sendto and recvfrom
Concurrent, Connection-less servers

- Algorithm

**Master 1.** Create a socket and bind to the well-known address for the service being offered. Leave the socket unconnected.

**Master 2.** Repeatedly call recvfrom to receive the next request from a client, and create a new slave thread/process to handle the response

**Slave 1.** Receive a specific request upon creation as well as access to the socket

**Slave 2.** Form a reply according to the application protocol and send it back to the client using sendto

**Slave 3.** Exit

- cost of process/thread creation for each client request

- while using threads, use thread-safe functions and be careful while passing arguments to threads
Concurrent, Connection-oriented servers

- Algorithm

**Master 1.** Create a socket and bind to the well-known address for the service being offered. Leave the socket unconnected.

**Master 2.** Place the socket in passive mode.

**Master 3.** Repeatedly call accept to receive the next request from a client, and create a new slave process/thread to handle the response.

**Slave 1.** Receive a connection request (i.e., socket for connection) upon creation.

**Slave 2.** Interact with the client using the connection: read request(s) and send back response(s).

**Slave 3.** Close the connection and exit.

- processes created using fork; can also use execve
Apparent concurrency using a single process

- multiple processes ⇒ need to use shared memory IPC facilities if data structures shared among processes
- creating processes can be expensive
- threads make this easier
- can also achieve the same goal using a single process and asynchronous I/O using select
Apparent concurrency using a single process

• Algorithm

1. Create a socket and bind to the well-known port for the service. Add the socket to the list of those on which I/O is possible
2. Use select to wait for I/O on existing sockets
3. If original socket is ready, use accept to obtain the next connection, and add the new socket to the list of those on which I/O is possible
4. If some socket other than the original is ready, use read to obtain the next request, form a response, and use write to send the response back to the client
5. Continue processing with step 2.
The Problem of Server Deadlock

- iterative server: suppose client creates a connection but does not send any requests

- suppose client does not consume responses

- connection-oriented servers will block on write if local buffer full $\Rightarrow$ deadlock in single process servers
Multi-protocol Server Design

- multiprotocol server handles service requests over both UDP and TCP
- Motivation: allows the use of shared code for service
- asynchronous I/O needed (select system call)
- design can be iterative or concurrent (multi-process or single-process)
Multi-service Server Design

- single server for multiple services

- Motivation: conserve system resources and make maintenance easier

- Design: Iterative, concurrent, or single process concurrent

- Connection-less or Connection-oriented

- Multi-service, Multi-protocol “super servers”, e.g. UNIX inetd

- Static or dynamic server configuration
UNIX inetd super server

- configuration file /etc/inetd.conf

- entries: service name (from /etc/services), socket type, protocol, wait status, userid, server program, arguments
Java sockets API

- TCP socket classes
  - Socket
  - ServerSocket
  - InetAddress

- UDP classes
  - DatagramPacket
  - DatagramSocket
Java Examples
public class EchoClient {
    public static void main(String[] args) throws IOException {

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket("taranis", 7);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for " + "the connection to: taranis.");
            System.exit(1);
        }

        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
        String userInput;
        while ((userInput = stdIn.readLine()) != null) {
            out.println(userInput);
            System.out.println("echo: " + in.readLine());
        }
        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
    }
}
A TCP Client for the Daytime service

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

public class DayClient1 {
    public static final int DAYTIME_PORT = 13;
    String host;
    Socket s;

    public static void main(String args[]) throws IOException {
        DayClient1 that = new DayClient1(args[0]);
        that.go();
    }

    public DayClient1(String host) {
        this.host = host;
    }

    public void go() throws IOException {
        s = new Socket(host, DAYTIME_PORT);
        BufferedReader i = new BufferedReader(
            new InputStreamReader(s.getInputStream()));
        System.out.println(i.readLine());
        i.close();
        s.close();
    }
}
A TCP Server for the Daytime service

import java.io.*;
import java.net.*;
import java.util.*;

public class DayServer1 {
    private ServerSocket ss;
    public static final int DAYTIME_PORT = 13;

    public static void main(String args[]) throws IOException {
        DayServer1 d = new DayServer1();
        d.go();
    }

    public void go() throws IOException {
        Socket s = null;
        ss = new ServerSocket(DAYTIME_PORT, 5);
        for (;;) {
            s = ss.accept();
            BufferedWriter out = new BufferedWriter(
                new OutputStreamWriter(s.getOutputStream(),"8859_1"));
            out.write("Java Daytime server: " +
                (new Date()).toString() + "\n");
            out.close();
            s.close();
        }
    }
}
A Multithreaded TCP server

```java
public class MultiServe implements Runnable {
    private ServerSocket ss;

    public static void main(String args[]) throws Exception {
        MultiServe m = new MultiServe();
        m.go();
    }

    public void go() throws Exception {
        ss = new ServerSocket(DayClient2.DAYTIME_PORT, 5);
        Thread t1 = new Thread(this, "1");
        Thread t2 = new Thread(this, "2");
        Thread t3 = new Thread(this, "3");
        t1.start(); t2.start(); t3.start();
    }

    public void run() {
        Socket s = null;
        BufferedWriter out = null;
        String myname = Thread.currentThread().getName();

        for (;;) {
            try {
                System.out.println("thread " + myname + " about to accept..");
                s = ss.accept();
                System.out.println("thread " + myname + " accepted a connection");
                out = new BufferedWriter(
                        new OutputStreamWriter(s.getOutputStream()));
                out.write(myname + " " + new Date());
                Thread.sleep(10000);
                out.write("\n");
                out.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```
Another Multi-threaded Server Example

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

public class KKMultiServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = null;
        boolean listening = true;

        try {
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Could not listen on port: 4444.");
            System.exit(-1);
        }

        while (listening) {
            new KKMultiServerThread(serverSocket.accept()).start();
        }
        serverSocket.close();
    }
}
```
import java.net.*;
import java.io.*;

public class KKMultiServerThread extends Thread {
    private Socket socket = null;

    public KKMultiServerThread(Socket socket) {
        super("KKMultiServerThread");
        this.socket = socket;
    }

    public void run() {
        try {
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    socket.getInputStream()));

            String inputLine, outputLine;
            KnockKnockProtocol kkp = new KnockKnockProtocol();
            outputLine = kkp.processInput(null);
            out.println(outputLine);

            while ((inputLine = in.readLine()) != null) {
                outputLine = kkp.processInput(inputLine);
                out.println(outputLine);
                if (outputLine.equals("Bye"))
                    break;
            }
            out.close();
            in.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
A UDP Client

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

public class QuoteClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.out.println("Usage: java QuoteClient <hostname>");
            return;
        }

        // get a datagram socket
        DatagramSocket socket = new DatagramSocket();

        // send request
        byte[] buf = new byte[256];
        InetAddress address = InetAddress.getByName(args[0]);
        DatagramPacket packet =
            new DatagramPacket(buf, buf.length, address, 4445);
        socket.send(packet);

        // get response
        packet = new DatagramPacket(buf, buf.length);
        socket.receive(packet);

        // display response
        String received = new String(packet.getData(), 0);
        System.out.println("Quote of the Moment: " + received);

        socket.close();
    }
}
```
import java.io.*;
import java.net.*;
import java.util.*;
public class QuoteServerThread extends Thread {
    protected DatagramSocket socket = null;
    protected BufferedReader in = null;
    protected boolean moreQuotes = true;

    public QuoteServerThread() throws IOException {
        this("QuoteServerThread");
    }

    public QuoteServerThread(String name) throws IOException {
        super(name);
        socket = new DatagramSocket(4445);
    
        try {
            in = new BufferedReader(new FileReader("one-liners.txt");
        } catch (FileNotFoundException e) {
            System.err.println("Could not open quote file. Serving time instead.");
        }
    }
}
public void run() {

    while (moreQuotes) {
        try {
            byte[] buf = new byte[256];

            // receive request
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);

            // figure out response
            String dString = null;
            if (in == null)
                dString = new Date().toString();
            else
                dString = getNextQuote();
            buf = dString.getBytes();

            // send the response to the client at "address" and "port"
            InetAddress address = packet.getAddress();
            int port = packet.getPort();
            packet = new DatagramPacket(buf, buf.length, address, port);
            socket.send(packet);
        } catch (IOException e) {
            moreQuotes = false;
            e.printStackTrace();
        }
    }
    socket.close();
}

protected String getNextQuote() {
    String returnValue = null;
    try {
        if ((returnValue = in.readLine()) == null) {
            in.close();
            moreQuotes = false;
            returnValue = "No more quotes. Goodbye.";
        }
    } catch (IOException e) {
        returnValue = "$IOException occurred in server.";
    }
    return returnValue;
}