Session 13

Network Programming

Outline

- How data is transmitted across the Internet
- Sockets
- Server Sockets
- UDP

Datagrams

- Before data is sent across the Internet from one host to another using TCP/IP, it is split into packets of varying but finite size called *datagrams*.
- Datagrams range in size from a few dozen bytes to about 60,000 bytes.
- Packets larger than this, and often smaller than this, must be split into smaller pieces before they can be transmitted.

Packets Allow Error Correction

- If one packet is lost, it can be retransmitted without requiring redelivery of all other packets.

- If packets arrive out of order they can be reordered at the receiving end of the connection.

Abstraction

- Datagrams are mostly hidden from the Java programmer.
- The host's native networking software transparently splits data into packets on the sending end of a connection, and then reassembles packets on the receiving end.
- Instead, the Java programmer is presented with a higher level abstraction called a *socket*.

Sockets

- A socket is a reliable connection for the transmission of data between two hosts.
- Sockets isolate programmers from the details of packet encodings, lost and retransmitted packets, and packets that arrive out of order.
- There are limits. Sockets are more likely to throw IOExceptions than files, for example.

Socket Operations

- There are four fundamental operations a socket performs.
- These are:
  1. **Connect to a remote machine**
  2. **Send data**
  3. **Receive data**
  4. **Close the connection**
- A socket may not be connected to more than one host at a time.
- A socket may not reconnect after it's closed.

The java.net.Socket class

- The `java.net.Socket` class allows you to create socket objects that perform all four fundamental socket operations.
- You can connect to remote machines; you can send data; you can receive data; you can close the connection.
- Each `Socket` object is associated with exactly one remote host. To connect to a different host, you must create a new `Socket` object.

Constructing a Socket

- Connection is accomplished through the constructors.
- public **Socket(String host, int port)** throws UnknownHostException, IOException
- public **Socket(InetAddress address, int port)** throws IOException
- public **Socket(String host, int port, InetAddress localAddr, int localPort)** throws IOException
- public **Socket(InetAddress address, int port, InetAddress localAddr, int localPort)** throws IOException

Opening Sockets

- The `Socket()` constructors do not just create a Socket object. They also attempt to connect the underlying socket to the remote server.

- All the constructors throw an `IOException` if the connection can't be made for any reason.

- You must at least specify the remote host and port to connect to.
- The host may be specified as either a string like "utopia.poly.edu" or as an `InetAddress` object.
- The port should be an int between 1 and 65535.

Socket s= new Socket(“www.google.com”, 80);

- You cannot just connect to any port on any host. The remote host must actually be listening for connections on that port.

- You can use the constructors to determine which ports on a host are listening for connections.

Sending and Receiving Data

- Data is sent and received with output and input streams.
- There are methods to get an input stream for a socket and an output stream for the socket.

```
public InputStream getInputStream() throws
  IOException

public OutputStream getOutputStream()
  throws IOException
```

- There's also a method to close a socket.

```
public synchronized void close() throws
  IOException
```

Reading Input from a Socket

- The `getInputStream()` method returns an InputStream which reads data from the socket.
- You can use all the normal methods of the `InputStream` class to read this data.
- Most of the time you'll chain the input stream to some other input stream or reader object to more easily handle the data.

Example

```
try {
  Socket s = new Socket( “192.168.201.20", 9999);
  InputStream in = s.getInputStream();
  InputStreamReader isr = new InputStreamReader(in);
  BufferedReader br = new BufferedReader(isr);
  String data= br.readLine();
  System.out.println(data);
}
catch (IOException e) {
  return (new Date()).toString();
}
```

Writing Output to a Socket

- The `getOutputStream()` method returns an output stream which writes data to the socket.

- Most of the time you'll chain the raw output stream to some other output stream or writer class to more easily handle the data.

Example

```
byte[] b = new byte[128];
try {
  Socket s = new Socket("metalab.unc.edu", 9);
  OutputStream theOutput = s.getOutputStream();
  PrintWriter pw=new PrintWriter(theOutput,true);
  pw.println( “data from client” );
  s.close();
}
catch (IOException e) {}
```

Servers

- There are two ends to each connection: the client, that is the host that initiates the connection, and the server, that is the host that responds to the connection.
- Clients and servers are connected by sockets.
- A serve, rather than connecting to a remote host, a program waits for other hosts to connect to it.

Server Sockets

- A server socket binds to a particular port on the local machine.
- Once it has successfully bound to a port, it listens for incoming connection attempts.
- When a server detects a connection attempt, it *accepts* the connection. This creates a socket between the client and the server over which the client and the server communicate.

Multiple Clients

- Multiple clients can connect to the same port on the server at the same time.
- Incoming data is distinguished by the port to which it is addressed and the client host and port from which it came.
- The server can tell for which service (like http or ftp) the data is intended by inspecting the port.
- It can tell which open socket on that service the data is intended for by looking at the client address and port stored with the data.

Threading

- No more than one server socket can listen to a particular port at one time.
- Since a server may need to handle many connections at once, server programs tend to be heavily multi-threaded.
- Generally the server socket passes off the actual processing of connections to a separate thread.

Queueing

- Incoming connections are stored in a queue until the server can accept them.
- On most systems the default queue length is between 5 and 50.
- Once the queue fills up further incoming connections are refused until space in the queue opens up.

The java.net.ServerSocket Class

- The **java.net.ServerSocket** class represents a server socket.
- A **ServerSocket** object is constructed on a particular local port. Then it calls accept() to listen for incoming connections.
- **accept()** blocks until a connection is detected. Then accept() returns a **java.net.Socket** object that performs the actual communication with the client.

Constructors

- public **ServerSocket(int port)** throws IOException
- public **ServerSocket(int port, int backlog)** throws IOException
- public **ServerSocket(int port, int backlog, InetAddress bindAddr)** throws IOException

Constructing Server Sockets

- Normally you only specify the port you want to listen on, like this:

```
try {
  ServerSocket ss = new ServerSocket(80);
}
catch (IOException e) {
  System.err.println(e);
}
```

- When a ServerSocket object is created, it attempts to *bind* to the port on the local host given by the port argument.
- If another server socket is already listening to the port, then a java.net.BindException, a subclass of IOException, is thrown.
- No more than one process or thread can listen to a particular port at a time. This includes non-Java processes or threads.
- For example, if there's already an HTTP server running on port 80, you won't be able to bind to port 80.

- The `accept()` and `close()` methods provide the basic functionality of a server socket.
  - public Socket accept() throws IOException
  - public void close() throws IOException
- A server socket can't be reopened after it's closed

Reading Data with a ServerSocket

- There are no `getInputStream()` or `getOutputStream()` methods for `ServerSocket`.

- `accept()` returns a Socket object, and its `getInputStream()` and `getOutputStream()` methods provide streams.

Example

```
try {
  ServerSocket ss = new ServerSocket(2345);
  Socket s = ss.accept();
  PrintWriter pw = new PrintWriter(s.getOutputStream());
  pw.println("Hello There!");
  pw.println("Goodbye now.);
  s.close();
}
catch (IOException e) {
  System.err.println(e);
}
```

Interacting with a Client

- More commonly, a server needs to both read a client request and write a response.

Adding Threading to a Server

- It's better to make your server multi-threaded.
- There should be a loop which continually accepts new connections.
- Rather than handling the connection directly the socket should be passed to a `Thread` object that handles the connection.

Adding a Thread Pool to a Server

- Multi-threading is a good thing but it's still not a perfect solution.
- Look at this accept loop:

```
while (true) {
 try {
        Socket s = ss.accept();
        ThreadedEchoServer tes = new ThreadedEchoServer(s)
        tes.start();
 }
catch (IOException e) {}
```

UDP

- Unreliable Datagram Protocol
- Packet Oriented, not stream oriented like TCP/IP
- Much faster but no error correction
- NFS, TFTP, and FSP use UDP/IP
- Must fit data into packets of about 8K or less

The UDP Classes

- Java's support for UDP is contained in two classes:

  `java.net.DatagramSocket`

  `java.net.DatagramPacket`

- A datagram socket is used to send and receive datagram packets.

java.net. DatagramPacket

- a wrapper for an array of bytes from which data will be sent or into which data will be received.

- also contains the address and port to which the packet will be sent.

java.net.DatagramSocket

- A `DatagramSocket` object is a local connection to a port that does the sending and receiving.
- There is no distinction between a UDP socket and a UDP server socket.
- Also unlike TCP sockets, a `DatagramSocket` can send to multiple, different addresses.
- The address to which data goes is stored in the packet, not in the socket.

DatagramPacket Constructors

For receiving

```
public DatagramPacket(byte[] data, int length)
```

For sending

```
public DatagramPacket(byte[] data, int length,
InetAddress iaddr, int iport)
```

DatagramPackets are mutable

public synchronized void setAddress(InetAddress iaddr)

public synchronized void setPort(int iport)

public synchronized void setData(byte ibuf[])

public synchronized void setLength(int ilength)

public synchronized InetAddress getAddress()

public synchronized int getPort()

public synchronized byte[] getData()

public synchronized int getLength()

java.net.DatagramSocket

This is for client datagram sockets; that is sockets that send datagrams before receiving any.

public **DatagramSocket()** throws SocketException

These are for server datagram sockets since they specify the port and optionally the IP address of the socket

public **DatagramSocket(int port)** throws SocketException

public **DatagramSocket(int port, InetAddress laddr)** throws SocketException

Sending UDP Datagrams

- To send data to a particular server
  - Convert the data into byte array.
  - Pass this byte array, the length of the data in the array (most of the time this will be the length of the array) and the `InetAddress` and port to which you wish to send it into the `DatagramPacket()` constructor.
  - Next create a `DatagramSocket` and pass the packet to its `send()` method

example,

```
InetAddress i= new InetAddess(“localhost");
int port= 19;
String s = “This is data from client.";
byte[] b = s.getBytes();
DatagramPacket dp = new DatagramPacket(b, b.length, ia,
chargen);
DatagramSocket sender = new DatagramSocket();
sender.send(dp);
```

Receiving UDP Datagrams

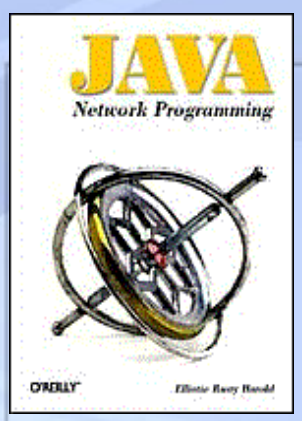
- Construct a `DatagramSocket` object on the port on which you want to listen.
- Pass an empty `DatagramPacket` object to the `DatagramSocket`'s `receive()` method.
  - public synchronized void receive(DatagramPacket dp) throws IOException
- The calling thread blocks until a datagram is received.

example

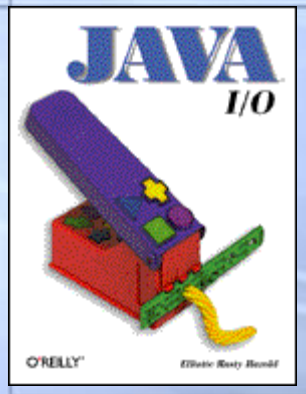
```
try {
  byte buffer = new byte[65536];
  DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2134);
  ds.receive(incoming);
  byte[] data = incoming.getData();
  String s = new String(data, 0, data.getLength());
  System.out.println("Port " + incoming.getPort() + " on " +
   incoming.getAddress() + " sent this message:");
  System.out.println(s);
}
catch (IOException e) {
  System.err.println(e);
}
```

To Learn More

- **Java Network Programming**
  - O'Reilly & Associates, 1997
  - ISBN 1-56592-227-1



- **Java I/O**
  - O'Reilly & Associates, 1999
  - ISBN 1-56592-485-1



Thank You!