

Java sockets 101

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Tutorial tips	2
2. Socket basics	3
3. An undercover socket	7
4. A simple example	11
5. A multithreaded example	18
6. A pooled example	21
7. Sockets in real life	27
8. Summary	31
9. Appendix	33

Section 1. Tutorial tips

Should I take this tutorial?

Sockets, which provide a mechanism for communication between two computers, have been around since long before the Java language was a glimmer in James Gosling's eye. The language simply lets you use sockets effectively without having to know the details of the underlying operating system. Most books that focus on Java coding either fail to cover the topic, or leave a lot to the imagination. This tutorial will tell you what you really need to know to start using sockets effectively in your Java code. Specifically, we'll cover:

- * What sockets are
- * Where they fit into the structure of programs you're likely to write
- * The simplest sockets implementation that could possibly work -- to help you understand the basics
- * A detailed walkthrough of two additional examples that explore sockets in multithreaded and pooled environments
- * A brief discussion of an application for sockets in the real world

If you can describe how to use the classes in the `java.net` package, this tutorial is probably a little basic for you, although it might be a good refresher. If you have been working with sockets on PCs and other platforms for years, the initial sections might bore you. But if you are new to sockets, and simply want to know what they are and how to use them effectively in your Java code, this tutorial is a great place to start.

Getting help

For questions about the content of this tutorial, contact the authors, Roy Miller (at rmiller@rolemodelsoft.com) or Adam Williams (at awilliams@rolemodelsoft.com).

Roy Miller and Adam Williams are Software Developers at RoleModel Software, Inc. They have worked jointly to prototype a socket-based application for the TINI Java platform from Dallas Semiconductor. Roy and Adam are currently working on porting a COBOL financial transaction system to the Java platform, using sockets.

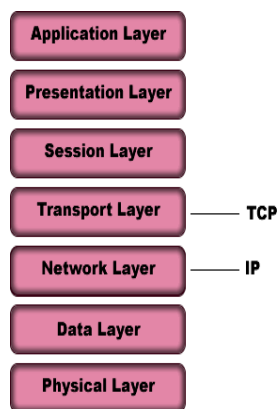
Prior to joining RoleModel, Roy spent six years with Andersen Consulting (now Accenture) developing software and managing projects. He co-authored *Extreme Programming Applied: Playing to Win* (Addison-Wesley XP Series) scheduled for publication in October 2001.

Section 2. Socket basics

Introduction

Most programmers, whether they're coding in the Java language or not, don't want to know much about low-level details of how applications on different computers communicate with each other. Programmers want to deal with higher-level abstractions that are easier to understand. Java programmers want objects that they can interact with via an intuitive interface, using the Java constructs with which they are familiar.

Sockets live in both worlds -- the low-level details that we'd rather avoid and the abstract layer we'd rather deal with. This section will explore just enough of the low-level details to make the abstract application understandable.



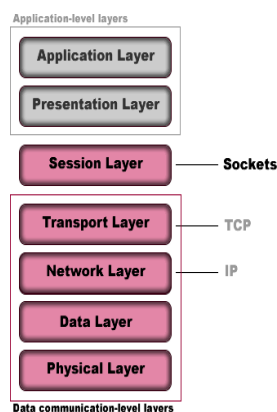
Computer networking 101

Computers operate and communicate with one another in a very simple way. Computer chips are a collection of on-off switches that store and transmit data in the form of 1s and 0s. When computers want to share data, all they need to do is stream a few million of these bits and bytes back and forth, while agreeing on speed, sequence, timing, and such. How would you like to worry about those details every time you wanted to communicate information between two applications?

To avoid that, we need a set of packaged protocols that can do the job the same way every time. That would allow us to handle our application-level work without having to worry about the low-level networking details. These sets of packaged protocols are called *stacks*. The most common stack these days is TCP/IP. Most stacks (including TCP/IP) adhere roughly to the International Standards Organization (ISO) Open Systems Interconnect Reference Model (OSIRM). The OSIRM says that there are seven logical layers in

a reliable framework for computer networking (see the diagram). Companies all over have contributed something that implements some of the layers in this model, from generating the electrical signals (pulses of light, radio frequency, and so on) to presenting the data to applications. TCP/IP maps to two layers in the OSI model, as shown in the diagram.

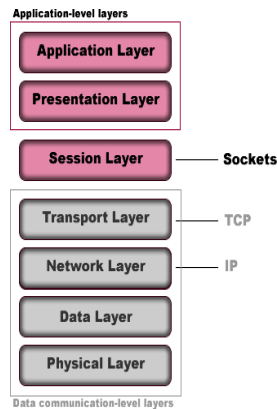
We won't go into the details of the layers too much, but we want you to be aware of where sockets fit.



Where sockets fit

Sockets reside roughly at the *Session Layer* of the OSI model (see the diagram). The Session Layer is sandwiched between the application-oriented upper layers and the real-time data communication lower layers. The Session Layer provides services for managing and controlling data flow between two computers. As part of this layer, sockets provide an abstraction that hides the complexities of getting the bits and bytes on the wire for transmission. In other words, sockets allow us to transmit data by having our application indicate that it wants to send some bytes. Sockets mask the nuts and bolts of getting the job done.

When you pick up your telephone, you provide sound waves to a sensor that converts your voice into electrically transmittable data. The phone is a human's interface to the telecommunications network. You aren't required to know the details of how your voice is transported, only the party to whom you would like to connect. In the same sense, a socket acts as a high-level interface that hides the complexities of transmitting 1s and 0s across unknown channels.



Exposing sockets to an application

When you write code that uses sockets, that code does work at the *Presentation Layer*. The Presentation Layer provides a common representation of information that the *Application Layer* can use. Say you are planning to connect your application to a legacy banking system that understands only EBCDIC. Your application domain objects store information in ASCII format. In this case, you are responsible for writing code at the Presentation Layer to convert data from EBCDIC to ASCII, and then (for example) to provide a domain object to your Application Layer. Your Application Layer can then do whatever it wants with the domain object.

The socket-handling code you write lives only at the Presentation Layer. Your Application Layer doesn't have to know anything about how sockets work.

What are sockets?

Now that we know the role sockets play, the question remains: What is a socket? Bruce Eckel describes a socket this way in his book *Thinking in Java*:

The socket is the software abstraction used to represent the "terminals" of a connection between two machines. For a given connection, there's a socket on each machine, and you can imagine a hypothetical "cable" running between the two machines with each end of the "cable" plugged into a socket. Of course, the physical hardware and cabling between machines is completely unknown. The whole point of the abstraction is that we don't have to know more than is necessary.

In a nutshell, a socket on one computer that talks to a socket on another computer creates a communication channel. A programmer can use that channel to send data between the two machines. When you send data, each layer of the TCP/IP stack adds appropriate header information to wrap your data. These headers help the stack get your data to its destination. The good news is that the Java language hides all of this from you by providing the data to your code on streams, which is why they are sometimes called *streaming sockets*.

Think of sockets as handsets on either side of a telephone call -- you and I talk and listen on our handsets on a dedicated channel. The conversation doesn't end until we decide to hang up (unless we're using cell phones). And until we hang up, our respective phone lines are busy.

If you need to communicate between two computers without the overhead of higher-level mechanisms like ORBs (and CORBA, RMI, IIOP, and so on), sockets are for you. The low-level details of sockets get rather involved. Fortunately, the Java platform gives you

some simple yet powerful higher-level abstractions that make creating and using sockets easy.

Types of sockets

Generally speaking, sockets come in two flavors in the Java language:

- * TCP sockets (implemented by the `Socket` class, which we'll discuss later)
- * UDP sockets (implemented by the `DatagramSocket` class)

TCP and UDP play the same role, but they do it differently. Both receive transport protocol packets and pass along their contents to the Presentation Layer. TCP divides messages into packets (*datagrams*) and reassembles them in the correct sequence at the receiving end. It also handles requesting retransmission of missing packets. With TCP, the upper-level layers have much less to worry about. UDP doesn't provide these assembly and retransmission requesting features. It simply passes packets along. The upper layers have to make sure that the message is complete and assembled in correct sequence.

In general, UDP imposes lower performance overhead on your application, but only if your application doesn't exchange lots of data all at once and doesn't have to reassemble lots of datagrams to complete a message. Otherwise, TCP is the simplest and probably most efficient choice.

Because most readers are more likely to use TCP than UDP, we'll limit our discussion to the TCP-oriented classes in the Java language.

Section 3. An undercover socket

Introduction

The Java platform provides implementations of sockets in the `java.net` package. In this tutorial, we'll be working with the following three classes in `java.net`:

- * `URLConnection`
- * `Socket`
- * `ServerSocket`

There are more classes in `java.net`, but these are the ones you'll run across the most often. Let's begin with `URLConnection`. This class provides a way to use sockets in your Java code without having to know *any* of the underlying socket details.

Using sockets without even trying

The `URLConnection` class is the abstract superclass of all classes that create a communications link between an application and a URL. `URLConnections` are most useful for getting documents on Web servers, but can be used to connect to any resource identified by a URL. Instances of this class can be used both to read from and to write to the resource. For example, you could connect to a servlet and send a well-formed XML `String` to the server for processing. Concrete subclasses of `URLConnection` (such as `HttpURLConnection`) provide extra features specific to their implementation. For our example, we're not doing anything special, so we'll make use of the default behaviors provided by `URLConnection` itself.

Connecting to a URL involves several steps:

- * Create the `URLConnection`
- * Configure it using various setter methods
- * Connect to the URL
- * Interact with it using various getter methods

Next, we'll look at some sample code that demonstrates how to use a `URLConnection` to request a document from a server.

The `URLClient` class

We'll begin with the structure for the `URLClient` class.

```
import java.io.*;
import java.net.*;
public class URLClient {
    protected URLConnection connection;
    public static void main(String[] args) {
    }
    public String getDocumentAt(String urlString) {
    }
}
```

The first order of business is to import `java.net` and `java.io`.

We give our class one instance variable to hold a `URLConnection`.

Our class has a `main()` method that handles the logic flow of surfing for a document. Our class also has a `getDocumentAt()` method that connects to the server and asks it for the given document. We will go into the details of each of these methods next.

Surfing for a document

The `main()` method handles the logic flow of surfing for a document:

```
public static void main(String[] args) {
    URLClient client = new URLClient();
    String yahoo = client.getDocumentAt("http://www.yahoo.com");
    System.out.println(yahoo);
}
```

Our `main()` method simply creates a new `URLClient` and calls `getDocumentAt()` with a valid URL String. When that call returns the document, we store it in a `String` and then print it out to the console. The real work, though, gets done in the `getDocumentAt()` method.

Requesting a document from a server

The `getDocumentAt()` method handles the real work of getting a document over the Web:

```
public String getDocumentAt(String urlString) {
    StringBuffer document = new StringBuffer();
    try {
        URL url = new URL(urlString);
        URLConnection conn = url.openConnection();
        BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        String line = null;
        while ((line = reader.readLine()) != null)
            document.append(line + "\n");
        reader.close();
    } catch (MalformedURLException e) {
        System.out.println("Unable to connect to URL: " + urlString);
    } catch (IOException e) {
        System.out.println("IOException when connecting to URL: " + urlString);
    }
    return document.toString();
}
```

The `getDocumentAt()` method takes a `String` containing the URL of the document we want to get. We start by creating a `StringBuffer` to hold the lines of the document. Next, we create a new `URL` with the `urlString` we passed in. Then we create a `URLConnection` and open it:

```
URLConnection conn = url.openConnection();
```


Once we have a `URLConnection`, we get its `InputStream` and wrap it in an `InputStreamReader`, which we then wrap in a `BufferedReader` so that we can read lines of the document we're getting from the server. We'll use this wrapping technique often when dealing with sockets in Java code, but we won't always discuss it in detail. You should be familiar with it before we move on:

```
BufferedReader reader =
    new BufferedReader(new InputStreamReader(conn.getInputStream()));
```

Having our `BufferedReader` makes reading the contents of our document easy. We call `readLine()` on `reader` in a `while` loop:

```
String line = null;
while ((line = reader.readLine()) != null)
    document.append(line + "\n");
```

The call to `readLine()` is going to *block* until it reaches a line termination character (for example, a newline character) in the incoming bytes on the `InputStream`. If it doesn't get one, it will keep waiting. It will return `null` only when the connection is closed. In this case, once we get a line, we append it to the `StringBuffer` called `document`, along with a newline character. This preserves the format of the document that was read on the server side.

When we're done reading lines, we close the `BufferedReader`:

```
reader.close();
```

If the `urlString` supplied to a `URL` constructor is invalid, a `MalformedURLException` is thrown. If something else goes wrong, such as when getting the `InputStream` on the connection, an `IOException` is thrown.

Wrapping up

Beneath the covers, `URLConnection` uses a socket to read from the `URL` we specified (which just resolves to an IP address), but we don't have to know about it and we don't care. But there's more to the story; we'll get to that shortly.

Before we move on, let's review the steps to create and use a `URLConnection`:

1. Instantiate a `URL` with a valid `URL String` of the resource you're connecting to (throws a `MalformedURLException` if there's a problem).
2. Open a connection on that `URL`.
3. Wrap the `InputStream` for that connection in a `BufferedReader` so you can read lines.
4. Read the document using your `BufferedReader`.

5. Close your `BufferedReader`.

You can find the complete code listing for `URLClient` at [Code listing for URLClient](#) on page 33

Section 4. A simple example

Background

The example we'll cover in this section illustrates how you can use `Socket` and `ServerSocket` in your Java code. The client uses a `Socket` to connect to a server. The server listens on port 3000 with a `ServerSocket`. The client requests the contents of a file on the server's C: drive.

For the sake of clarity, we split the example into the client side and the server side. At the end, we'll put it all together so you can see the entire picture.

We developed this code in IBM VisualAge for Java 3.5, which uses JDK 1.2. To create this example for yourself, JDK 1.1.7 or greater should be fine. The client and the server will run on a single machine, so don't worry about having a network available.

Creating the RemoteFileClient class

Here is the structure for the `RemoteFileClient` class:

```
import java.io.*;
import java.net.*;
public class RemoteFileClient {
    protected String hostIp;
    protected int hostPort;
    protected BufferedReader socketReader;
    protected PrintWriter socketWriter;
    public RemoteFileClient(String aHostIp, int aHostPort) {
        hostIp = aHostIp;
        hostPort = aHostPort;
    }
    public static void main(String[] args) {
    }
    public void setUpConnection() {
    }
    public String getFile(String fileNameToGet) {
    }
    public void tearDownConnection() {
    }
}
```

First we import `java.net` and `java.io`. The `java.net` package gives you the socket tools you need. The `java.io` package gives you tools to read and write streams, which is the only way you can communicate with TCP sockets.

We give our class instance variables to support reading from and writing to socket streams, and to store details of the remote host to which we will connect.

The constructor for our class takes an IP address and a port number for a remote host and assigns them to instance variables.

Our class has a `main()` method and three other methods. We'll go into the details of these methods later. For now, just know that `setUpConnection()` will connect to the remote

`server`, `getFile()` will ask the remote server for the contents of `fileNameToGet`, and `tearDownConnection()` will disconnect from the remote server.

Implementing main()

Here we implement the `main()` method, which will create the `RemoteFileClient`, use it to get the contents of a remote file, and then print the result:

```
public static void main(String[] args) {
    RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
    remoteFileClient.setUpConnection();
    String fileContents =
        remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt");
    remoteFileClient.tearDownConnection();
    System.out.println(fileContents);
}
```

The `main()` method instantiates a new `RemoteFileClient` (the client) with an IP address and port number for the host. Then, we tell the client to set up a connection to the host (more on this later). Next, we tell the client to get the contents of a specified file on the host. Finally, we tell the client to tear down its connection to the host. We print out the contents of the file to the console, just to prove everything worked as planned.

Setting up a connection

Here we implement the `setUpConnection()` method, which will set up our `Socket` and give us access to its streams:

```
public void setUpConnection() {
    try {
        Socket client = new Socket(hostIp, hostPort);
        socketReader = new BufferedReader(
            new InputStreamReader(client.getInputStream()));
        socketWriter = new PrintWriter(client.getOutputStream());
    } catch (UnknownHostException e) {
        System.out.println("Error setting up socket connection: unknown host at " + hostIp);
    } catch (IOException e) {
        System.out.println("Error setting up socket connection: " + e);
    }
}
```

The `setUpConnection()` method creates a `Socket` with the IP address and port number of the host:

```
Socket client = new Socket(hostIp, hostPort);
```

We wrap the `Socket`'s `InputStream` in a `BufferedReader` so that we can read lines from the stream. Then, we wrap the `Socket`'s `OutputStream` in a `PrintWriter` so that we can send our request for a file to the server:

```
socketReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
socketWriter = new PrintWriter(client.getOutputStream());
```

Remember that our client and server simply pass bytes back and forth. Both the client and the server have to know what the other is going to be sending so that they can respond appropriately. In this case, the server knows that we'll be sending it a valid file path.

When you instantiate a `Socket`, an `UnknownHostException` may be thrown. We don't do anything special to handle it here, but we print some information out to the console to tell us what went wrong. Likewise, if a general `IOException` is thrown when we try to get the `InputStream` or `OutputStream` on a `Socket`, we print out some information to the console. This is our general approach in this tutorial. In production code, we would be a little more sophisticated.

Talking to the host

Here we implement the `getFile()` method, which will tell the server what file we want and receive the contents from the server when it sends the contents back:

```
public String getFile(String fileNameToGet) {
    StringBuffer fileLines = new StringBuffer();
    try {
        socketWriter.println(fileNameToGet);
        socketWriter.flush();
        String line = null;
        while ((line = socketReader.readLine()) != null)
            fileLines.append(line + "\n");
    } catch (IOException e) {
        System.out.println("Error reading from file: " + fileNameToGet);
    }
    return fileLines.toString();
}
```

A call to the `getFile()` method requires a valid file path `String`. It starts by creating the `StringBuffer` called `fileLines` for storing each of the lines that we read from the file on the server:

```
StringBuffer fileLines = new StringBuffer();
```

In the `try{}catch{} block`, we send our request to the host using the `PrintWriter` that was established during connection setup:

```
socketWriter.println(fileNameToGet);
socketWriter.flush();
```

Note that we `flush()` the `PrintWriter` here instead of closing it. This forces data to be sent to the server without closing the `Socket`.

Once we've written to the `Socket`, we are expecting some response. We have to wait for it on the `Socket's InputStream`, which we do by calling `readLine()` on our `BufferedReader` in a `while` loop. We append each returned line to the `fileLines` `StringBuffer` (with a newline character to preserve the lines):

```
String line = null;
while ((line = socketReader.readLine()) != null)
    fileLines.append(line + "\n");
```

Tearing down a connection

Here we implement the `tearDownConnection()` method, which will "clean up" after we're done using our connection:

```
public void tearDownConnection() {
    try {
        socketWriter.close();
        socketReader.close();
    } catch (IOException e) {
        System.out.println("Error tearing down socket connection: " + e);
    }
}
```

The `tearDownConnection()` method simply closes the `BufferedReader` and `PrintWriter` we created on our `Socket`'s `InputStream` and `OutputStream`, respectively. Doing this closes the underlying streams that we acquired from the `Socket`, so we have to catch the possible `IOException`.

Wrapping up the client

Our class is done. Before we move on to the server end of things, let's review the steps to create and use a `Socket`:

1. Instantiate a `Socket` with the IP address and port of the machine you're connecting to (throws an `Exception` if there's a problem).
2. Get the streams on that `Socket` for reading and writing.
3. Wrap the streams in instances of `BufferedReader/PrintWriter`, if that makes things easier.
4. Read from and write to the `Socket`.
5. Close your open streams.

You can find the complete code listing for `RemoteFileClient` at [Code listing for RemoteFileClient](#) on page 33.

Creating the RemoteFileServer class

Here is the structure for the `RemoteFileServer` class:

```
import java.io.*;
import java.net.*;
public class RemoteFileServer {
    protected int listenPort = 3000;
    public static void main(String[] args) {
```

```
    }  
    public void acceptConnections() {  
    }  
    public void handleConnection(Socket incomingConnection) {  
    }  
}
```

As with the client, we first import `java.net` and `java.io`. Next, we give our class an instance variable to hold the port to listen to for incoming connections. By default, this is port 3000.

Our class has a `main()` method and two other methods. We'll go into the details of these methods later. For now, just know that `acceptConnections()` will allow clients to connect to the server, and `handleConnection()` interacts with the client `Socket` to send the contents of the requested file to the client.

Implementing main()

Here we implement the `main()` method, which will create a `RemoteFileServer` and tell it to accept connections:

```
public static void main(String[] args) {  
    RemoteFileServer server = new RemoteFileServer();  
    server.acceptConnections();  
}
```

The `main()` method on the server side is even simpler than on the client side. We instantiate a new `RemoteFileServer`, which will listen for incoming connection requests on the default listen port. Then we call `acceptConnections()` to tell the server to listen.

Accepting connections

Here we implement the `acceptConnections()` method, which will set up a `ServerSocket` and wait for connection requests:

```
public void acceptConnections() {  
    try {  
        ServerSocket server = new ServerSocket(listenPort);  
        Socket incomingConnection = null;  
        while (true) {  
            incomingConnection = server.accept();  
            handleConnection(incomingConnection);  
        }  
    } catch (BindException e) {  
        System.out.println("Unable to bind to port " + listenPort);  
    } catch (IOException e) {  
        System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);  
    }  
}
```

The `acceptConnections()` method creates a `ServerSocket` with the port number to listen to. We then tell the `ServerSocket` to start listening by calling `accept()` on it. The

`accept()` method blocks until a connection request comes in. At that point, `accept()` returns a new `Socket` bound to a randomly assigned port on the server, which is passed to `handleConnection()`. Notice that this accepting of connections is in an infinite loop. No shutdown supported here.

Whenever you create a `ServerSocket`, Java code may throw an error if it can't *bind* to the specified port (perhaps because something else already has control of that port). So we have to catch the possible `BindException` here. And just like on the client side, we have to catch an `IOException` that could be thrown when we try to accept connections on our `ServerSocket`. Note that you can set a timeout on the `accept()` call by calling `setSoTimeout()` with number of milliseconds to avoid a really long wait. Calling `setSoTimeout()` will cause `accept()` to throw an `IOException` after the specified elapsed time.

Handling connections

Here we implement the `handleConnection()` method, which will use streams on a connection to receive input and write output:

```
public void handleConnection(Socket incomingConnection) {
    try {
        OutputStream outputToSocket = incomingConnection.getOutputStream();
        InputStream inputFromSocket = incomingConnection.getInputStream();
        BufferedReader streamReader =
            new BufferedReader(new InputStreamReader(inputFromSocket));
        FileReader fileReader = new FileReader(new File(streamReader.readLine()));
        BufferedReader bufferedFileReader = new BufferedReader(fileReader);
        PrintWriter streamWriter =
            new PrintWriter(incomingConnection.getOutputStream());
        String line = null;
        while ((line = bufferedFileReader.readLine()) != null) {
            streamWriter.println(line);
        }
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

As with the client, we get the streams associated with the `Socket` we just made, using `getOutputStream()` and `getInputStream()`. As on the client side, we wrap the `InputStream` in a `BufferedReader` and the `OutputStream` in a `PrintWriter`. On the server side, we need to add some code to read the target file and send the contents to the client line by line. Here's the important code:

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
BufferedReader bufferedFileReader = new BufferedReader(fileReader);
String line = null;
while ((line = bufferedFileReader.readLine()) != null) {
    streamWriter.println(line);
}
```

This code deserves some detailed explanation. Let's look at it bit by bit:


```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
```

First, we make use of our `BufferedReader` on the `Socket`'s `InputStream`. We should be getting a valid file path, so we construct a new `File` using that path name. We make a new `FileReader` to handle reading the file.

```
BufferedReader bufferedFileReader = new BufferedReader(fileReader);
```

Here we wrap our `FileReader` in a `BufferedReader` to let us read the file line by line.

Next, we call `readLine()` on our `BufferedReader`. This call will block until bytes come in. When we get some bytes, we put them in our local `line` variable, and then write them out to the client. When we're done reading and writing, we close the open streams.

Note that we closed `streamWriter` and `streamReader` after we were done reading from the `Socket`. You might ask why we didn't close `streamReader` immediately after reading in the file name. The reason is that when you do that, your client won't get any data. If you close the `streamReader` before you close `streamWriter`, you can write to the `Socket` all you want but no data will make it across the channel (it's closed).

Wrapping up the server

Before we move on to another, more practical example, let's review the steps to create and use a `ServerSocket`:

1. Instantiate a `ServerSocket` with a port on which you want it to listen for incoming client connections (throws an `Exception` if there's a problem).
2. Call `accept()` on the `ServerSocket` to block while waiting for connection.
3. Get the streams on that underlying `Socket` for reading and writing.
4. Wrap the streams as necessary to simplify your life.
5. Read from and write to the `Socket`.
6. Close your open streams (and remember, *never* close your `Reader` before your `Writer`).

You can find the complete code listing for `RemoteFileServer` at [Code listing for RemoteFileServer](#) on page 34.

Section 5. A multithreaded example

Introduction

The previous example gives you the basics, but that won't take you very far. If you stopped here, you could handle only one client at a time. The reason is that `handleConnection()` is a blocking method. Only when it has completed its dealings with the current connection can the server accept another client. Most of the time, you will want (and need) a multithreaded server.

There aren't too many changes you need to make to `RemoteFileServer` to begin handling multiple clients simultaneously. As a matter of fact, had we discussed backlogging earlier, we would have just one method to change, although we'll need to create something new to handle the incoming connections. We will show you here also how `ServerSocket` handles lots of clients waiting (backing up) to use our server. This example illustrates an inefficient use of threads, so be patient.

Accepting (too many?) connections

Here we implement the revised `acceptConnections()` method, which will create a `ServerSocket` that can handle a backlog of requests, and tell it to accept connections:

```
public void acceptConnections() {
    try {
        ServerSocket server = new ServerSocket(listenPort, 5);
        Socket incomingConnection = null;
        while (true) {
            incomingConnection = server.accept();
            handleConnection(incomingConnection);
        }
    } catch (BindException e) {
        System.out.println("Unable to bind to port " + listenPort);
    } catch (IOException e) {
        System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
    }
}
```

Our new server still needs to `acceptConnections()` so this code is virtually identical. The highlighted line indicates the one significant difference. For this multithreaded version, we now specify the maximum number of client requests that can backlog when instantiating the `ServerSocket`. If we don't specify the max number of client requests, the default value of 50 is assumed.

Here's how it works. Suppose we specify a backlog of 5 and that five clients request connections to our server. Our server will start processing the first connection, but it takes a long time to process that connection. Since our backlog is 5, we can have up to five requests in the queue at one time. We're processing one, so that means we can have five others waiting. That's a total of six either waiting or being processed. If a seventh client asks for a connection while our server is still busy accepting connection one (remember that 2-6 are still in queue), that seventh client will be refused. We will illustrate limiting the number of clients that can be connected simultaneously in our pooled server example.

Handling connections: Part 1

Here we'll talk about the structure of the `handleConnection()` method, which spawns a new `Thread` to handle each connection. We'll discuss this in two parts. We'll focus on the method itself in this panel, and then examine the structure of the `ConnectionHandler` helper class used by this method in the next panel.

```
public void handleConnection(Socket connectionToHandle) {
    new Thread(new ConnectionHandler(connectionToHandle)).start();
}
```

This method represents the big change to our `RemoteFileServer`. We still call `handleConnection()` after the server accepts a connection, but now we pass that `Socket` to an instance of `ConnectionHandler`, which is `Runnable`. We create a new `Thread` with our `ConnectionHandler` and start it up. The `ConnectionHandler`'s `run()` method contains the `Socket` reading/writing and `File` reading code that used to be in `handleConnection()` on `RemoteFileServer`.

Handling connections: Part 2

Here is the structure for the `ConnectionHandler` class:

```
import java.io.*;
import java.net.*;
public class ConnectionHandler implements Runnable{
    Socket socketToHandle;
    public ConnectionHandler(Socket aSocketToHandle) {
        socketToHandle = aSocketToHandle;
    }
    public void run() {
    }
}
```

This helper class is pretty simple. As with our other classes so far, we import `java.net` and `java.io`. The class has a single instance variable, `socketToHandle`, that holds the `Socket` handled by the instance.

The constructor for our class takes a `Socket` instance and assigns it to `socketToHandle`.

Notice that the class implements the `Runnable` interface. Classes that implement this interface must implement the `run()` method, which our class does. We'll go into the details of `run()` later. For now, just know that it will actually process the connection using code identical to what we saw before in our `RemoteFileServer` class.

Implementing run()

Here we implement the `run()` method, which will grab the streams on our connection, use them to read from and write to the connection, and close them when we are done:

```
public void run() {
    try {
```

```
        PrintWriter streamWriter = new PrintWriter(socketToHandle.getOutputStream());
        BufferedReader streamReader =
            new BufferedReader(new InputStreamReader(socketToHandle.getInputStream()));
        String fileToRead = streamReader.readLine();
        BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));
        String line = null;
        while ((line = fileReader.readLine()) != null)
            streamWriter.println(line);
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

The `run()` method on `ConnectionHandler` does what `handleConnection()` on `RemoteFileServer` did. First, we wrap the `InputStream` and `OutputStream` in a `BufferedReader` and a `PrintWriter`, respectively (using `getOutputStream()` and `getInputStream()` on the `Socket`). Then we read the target file line by line with this code:

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
BufferedReader bufferedFileReader = new BufferedReader(fileReader);
String line = null;
while ((line = bufferedFileReader.readLine()) != null) {
    streamWriter.println(line);
}
```

Remember that we should be getting a valid file path from the client, so we construct a new `File` using that path name, wrap it in a `FileReader` to handle reading the file, and then wrap that in a `BufferedReader` to let us read the file line by line. We call `readLine()` on our `BufferedReader` in a `while` loop until we have no more lines to read. Remember that the call to `readLine()` will block until bytes come in. When we get some bytes, we put them in our local `line` variable, and then write them out to the client. When we're done reading and writing, we close the open streams.

Wrapping up the multithreaded server

Our multithreaded server is done. Before we move on to the pooled example, let's review the steps to create and use a multithreaded version of the server:

1. Modify `acceptConnections()` to instantiate a `ServerSocket` with a default 50-connection backlog (or whatever specific number you want, greater than 1).
2. Modify `handleConnection()` on the `ServerSocket` to spawn a new `Thread` with an instance of `ConnectionHandler`.
3. Implement the `ConnectionHandler` class, borrowing code from the `handleConnection()` method on `RemoteFileServer`.

You can find the complete code listing for `MultithreadedRemoteFileServer` at [Code listing for MultithreadedRemoteFileServer](#) on page 35, and the complete code listing for `ConnectionHandler` at [Code listing for ConnectionHandler](#) on page 35.

Section 6. A pooled example

Introduction

The `MultithreadedServer` we've got now simply creates a new `ConnectionHandler` in a new `Thread` each time a client asks for a connection. That means we have potentially a bunch of `Threads` lying around. Creating a `Thread` isn't trivial in terms of system overhead, either. If performance becomes an issue (and don't assume it will until it does), being more efficient about handling our server would be a good thing. So, how do we manage the server side more efficiently? We can maintain a pool of incoming connections that a limited number of `ConnectionHandlers` will service. This design provides the following benefits:

- * It limits the number of simultaneous connections allowed.
- * We only have to start up `ConnectionHandler` `Threads` one time.

Fortunately, as with our multithreaded example, adding pooling to our code doesn't require an overhaul. In fact, the client side of the application isn't affected at all. On the server side, we create a set number of `ConnectionHandlers` when the server starts, place incoming connections into a pool and let the `ConnectionHandlers` take care of the rest. There are many possible tweaks to this design that we won't cover. For instance, we could refuse clients by limiting the number of connections we allow to build up in the pool.

Note: We will not cover `acceptConnections()` again. This method is exactly the same as in earlier examples. It loops forever calling `accept()` on a `ServerSocket` and passes the connection to `handleConnection()`.

Creating the `PooledRemoteFileServer` class

Here is the structure for the `PooledRemoteFileServer` class:

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledRemoteFileServer {
    protected int maxConnections;
    protected int listenPort;
    protected ServerSocket serverSocket;
    public PooledRemoteFileServer(int aListenPort, int maxConnections) {
        listenPort = aListenPort;
        this.maxConnections = maxConnections;
    }
    public static void main(String[] args) {
    }
    public void setUpHandlers() {
    }
    public void acceptConnections() {
    }
    protected void handleConnection(Socket incomingConnection) {
    }
}
```

Note the `import` statements that should be familiar by now. We give our class the following instance variables to hold:

- * The maximum number of simultaneous active client connections our server can handle
- * The port to listen to for incoming connections (we didn't assign a default value, but feel free to do that if you want)
- * The `ServerSocket` that will accept client connection requests

The constructor for our class takes the port to listen to and the maximum number of connections.

Our class has a `main()` method and three other methods. We'll go into the details of these methods later. For now, just know that `setUpHandlers()` creates a number of `PooledConnectionHandler` instances equal to `maxConnections` and the other two methods are like what we've seen before: `acceptConnections()` listens on the `ServerSocket` for incoming client connections, and `handleConnection` actually handles each client connection once it's established.

Implementing main()

Here we implement the revised `main()` method, which will create a `PooledRemoteFileServer` that can handle a given number of client connections, and tell it to accept connections:

```
public static void main(String[] args) {
    PooledRemoteFileServer server = new PooledRemoteFileServer(3000, 3);
    server.setUpHandlers();
    server.acceptConnections();
}
```

Our `main()` method is straightforward. We instantiate a new `PooledRemoteFileServer`, which will set up three `PooledConnectionHandlers` by calling `setUpHandlers()`. Once the server is ready, we tell it to `acceptConnections()`.

Setting up the connection handlers

```
public void setUpHandlers() {
    for (int i = 0; i < maxConnections; i++) {
        PooledConnectionHandler currentHandler = new PooledConnectionHandler();
        new Thread(currentHandler, "Handler " + i).start();
    }
}
```

The `setUpHandlers()` method creates `maxConnections` worth of `PooledConnectionHandlers` (three) and fires them up in new `Threads`. Creating a `Thread` with an object that implements `Runnable` allows us to call `start()` on the `Thread` and expect `run()` to be called on the `Runnable`. In other words, our `PooledConnectionHandlers` will be waiting to handle incoming connections, each in its own `Thread`. We create only three `Threads` in our example, and this cannot change once the server is running.

Handling connections

Here we implement the revised `handleConnections()` method, which will delegate handling a connection to a `PooledConnectionHandler`:

```
protected void handleConnection(Socket connectionToHandle) {
    PooledConnectionHandler.processRequest(connectionToHandle);
}
```

We now ask our `PooledConnectionHandlers` to process all incoming connections (`processRequest()` is a static method).

Here is the structure for the `PooledConnectionHandler` class:

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledConnectionHandler implements Runnable {
    protected Socket connection;
    protected static List pool = new LinkedList();
    public PooledConnectionHandler() {
    }
    public void handleConnection() {
    }
    public static void processRequest(Socket requestToHandle) {
    }
    public void run() {
    }
}
```

This helper class is very much like `ConnectionHandler`, but with a twist to handle connection pooling. The class has two single instance variables:

- * `connection`, the `Socket` that is currently being handled
- * A static `LinkedList` called `pool` that holds the connections that need to be handled

Filling the connection pool

Here we implement the `processRequest()` method on our `PooledConnectionHandler`, which will add incoming requests to the pool and tell other objects waiting on the pool that it now has some contents:

```
public static void processRequest(Socket requestToHandle) {
    synchronized (pool) {
        pool.add(pool.size(), requestToHandle);
        pool.notifyAll();
    }
}
```

This method requires some background on how the Java keyword `synchronized` works. We will attempt a short lesson on threading.

First, some definitions:

- * **Atomic method.** Methods (or blocks of code) that cannot be interrupted mid-execution
- * **Mutex lock.** A single "lock" that must be obtained by a client wishing to execute an atomic method

So, when object A wants to use `synchronized` method `doSomething()` on object B, object A must first attempt to acquire the mutex from object B. Yes, this means that when object A has the mutex, no other object may call *any* other `synchronized` method on object B.

A `synchronized` block is a slightly different animal. You can synchronize a block on any object, not just the object that has the block in one of its methods. In our example, our `processRequest()` method contains a block `synchronized` on the `pool` object (remember it's a `LinkedList` that holds the pool of connections to be handled). The reason we do this is to ensure that nobody else can modify the connection pool at the same time we are.

Now that we've guaranteed that we're the only ones wading in the pool, we can add the incoming `Socket` to the end of our `LinkedList`. Once we've added the new connection, we notify other `Threads` waiting to access the pool that it's now available, using this code:

```
pool.notifyAll();
```

All subclasses of `Object` inherit the `notifyAll()` method. This method, in conjunction with the `wait()` method that we'll discuss in the next panel, allows one `Thread` to let another `Thread` know that some condition has been met. That means that the second `Thread` must have been waiting for that condition to be satisfied.

Getting connections from the pool

Here we implement the revised `run()` method on `PooledConnectionHandler`, which will wait on the connection pool and handle the connection once the pool has one:

```
public void run() {
    while (true) {
        synchronized (pool) {
            while (pool.isEmpty()) {
                try {
                    pool.wait();
                } catch (InterruptedException e) {
                    return;
                }
            }
            connection = (Socket) pool.remove(0);
        }
        handleConnection();
    }
}
```

Recall from the previous panel that a `Thread` is waiting to be notified that a condition on the connection pool has been satisfied. In our example, remember that we have three

`PooledConnectionHandlers` waiting to use connections in the pool. Each of these `PooledConnectionHandlers` is running in its own `Thread` and is blocked on the call to `pool.wait()`. When our `processRequest()` method called `notifyAll()` on the connection pool, all of our waiting `PooledConnectionHandlers` were notified that the pool was available. Each one then continues past the call to `pool.wait()`, and rechecks the `while(pool.isEmpty())` loop condition. The pool will be empty for all but one handler, so all but one handler will block again on the call to `pool.wait()`. The one that encounters a non-empty pool will break out of the `while(pool.isEmpty())` loop and will grab the first connection from the pool:

```
connection = (Socket) pool.remove(0);
```

Once it has a connection to use, it calls `handleConnection()` to handle it.

In our example, the pool probably won't ever have more than one connection in it, simply because things execute so fast. If there were more than one connection in the pool, then the other handlers wouldn't have to wait for new connections to be added to the pool. When they checked the `pool.isEmpty()` condition, it would fail, and they would proceed to grab a connection from the pool and handle it.

One other thing to note. How is the `processRequest()` method able to put connections in the pool when the `run()` method has a mutex lock on the pool? The answer is that the call to `wait()` on the pool releases the lock, and then grabs it again right before it returns. This allows other code synchronized on the pool object to acquire the lock.

Handling connections: One more time

Here we implement the revised `handleConnection()` method, which will grab the streams on a connection, use them, and clean them up when finished:

```
public void handleConnection() {
    try {
        PrintWriter streamWriter = new PrintWriter(connection.getOutputStream());
        BufferedReader streamReader =
            new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String fileToRead = streamReader.readLine();
        BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));
        String line = null;
        while ((line = fileReader.readLine()) != null)
            streamWriter.println(line);
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("Could not find requested file on the server.");
    } catch (IOException e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

Unlike in our multithreaded server, our `PooledConnectionHandler` has a `handleConnection()` method. The code within this method is exactly the same as the code in the `run()` method on our non-pooled `ConnectionHandler`. First, we wrap the

`OutputStream` and `InputStream` in a `PrintWriter` and a `BufferedReader`, respectively (using `getOutputStream()` and `getInputStream()` on the `Socket`). Then we read the target file line by line, just as we did in the multithreaded example. Again, when we get some bytes, we put them in our local `line` variable, and then write them out to the client. When we're done reading and writing, we close our `FileReader` and the open streams.

Wrapping up the pooled server

Our pooled server is done. Let's review the steps to create and use a pooled version of the server:

1. Create a new kind of connection handler (we called it `PooledConnectionHandler`) to handle connections in a pool.
2. Modify the server to create and use a set of `PooledConnectionHandlers`.

You can find the complete code listing for `PooledRemoteFileServer` at [Code listing for PooledRemoteFileServer](#) on page 36, and the complete code listing for `PooledConnectionHandler` at [Code listing for PooledConnectionHandler](#) on page 37.

Section 7. Sockets in real life

Introduction

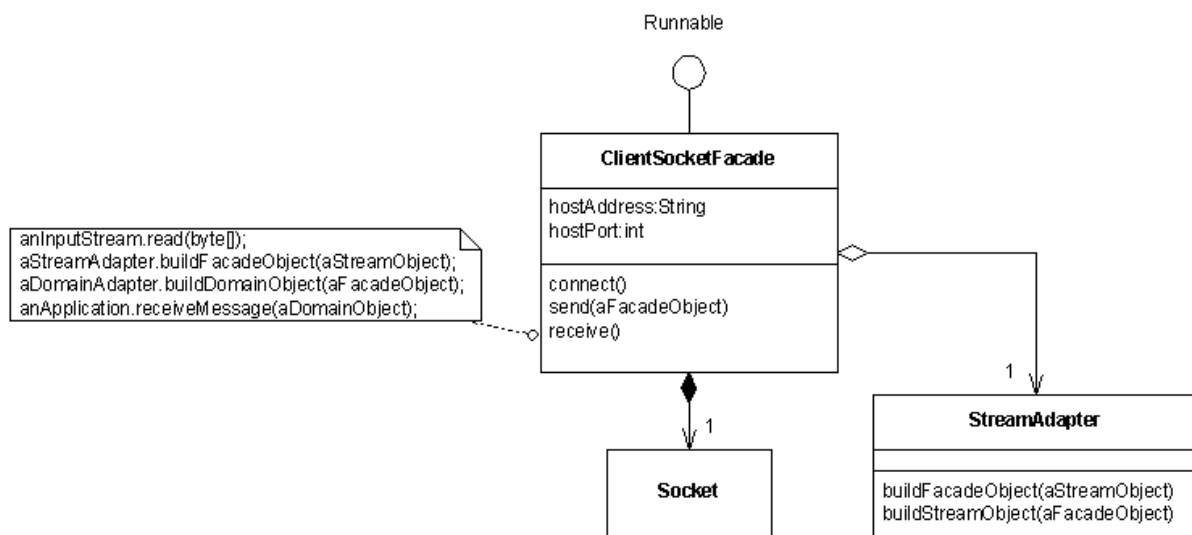
The examples we've talked about so far cover the mechanics of sockets in Java programming, but how would you use them on something "real?" Such a simple use of sockets, even with multithreading and pooling, would not be appropriate in most applications. Instead, it would probably be smart to use sockets within other classes that model your problem domain.

We did this recently in porting an application from a mainframe/SNA environment to a TCP/IP environment. The application's job is to facilitate communication between a retail outlet (such as a hardware store) and financial institutions. Our application is the middleman. As such, it needs to communicate with the retail outlet on one side and the financial outlet on the other. We had to handle a client talking to a server via sockets, and we had to translate our domain objects into socket-ready stuff for transmission.

We can't cover all the detail of this application in this tutorial, but let us take you on a tour of some of the high points. You can extrapolate from here to your own problem domain.

The client side

On the client side, the key players in our system were `Socket`, `ClientSocketFacade`, and `StreamAdapter`. The UML is shown in the following diagram:



We created a `ClientSocketFacade`, which is `Runnable` and owns an instance of `Socket`. Our application can instantiate a `ClientSocketFacade` with a particular host IP address and port number, and run it in a new `Thread`. The `run()` method on `ClientSocketFacade` calls `connect()`, which lazily initializes a `Socket`. With `Socket` instance in hand, our `ClientSocketFacade` calls `receive()` on itself, which blocks until the server sends some data over the `Socket`. Whenever the server sends some data, our `ClientSocketFacade` will wake up and handle the incoming data. Sending data is just as direct. Our application can simply tell its `ClientSocketFacade` to send data to its server by

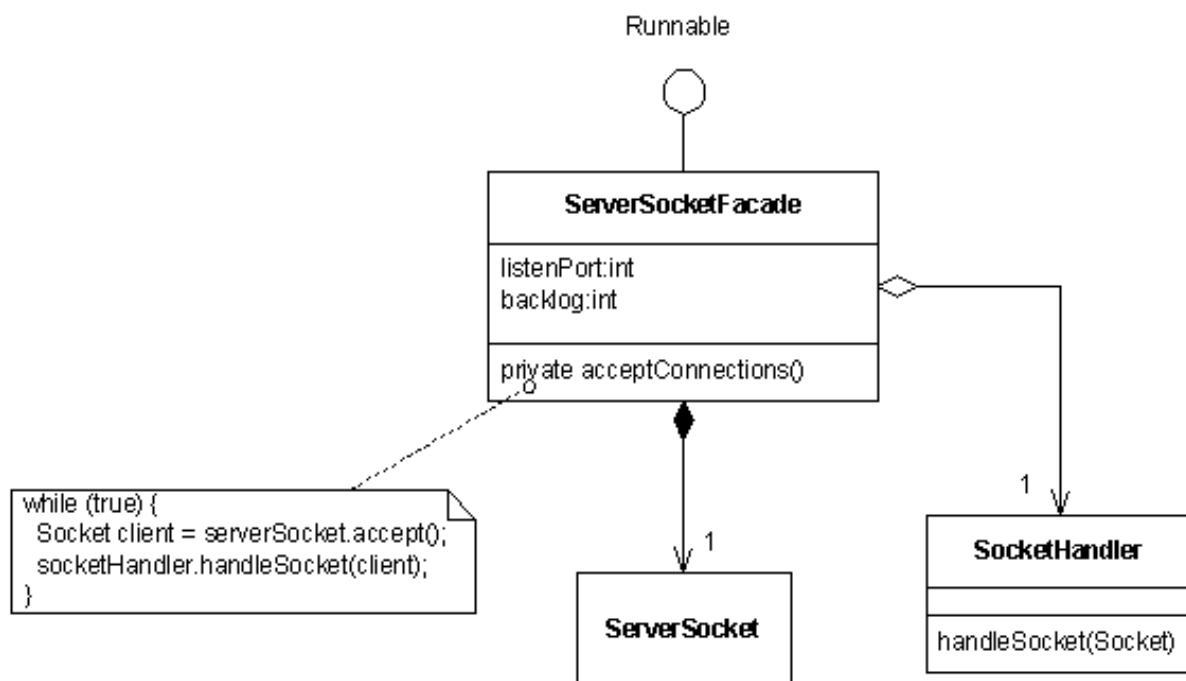
calling the `send()` method with a `StreamObject`.

The only piece missing from the discussion above is `StreamAdapter`. When an application tells the `ClientSocketFacade` to send data, the Facade delegates the operation to an instance of `StreamAdapter`. The `ClientSocketFacade` delegates receiving data to the same instance of `StreamAdapter`. A `StreamAdapter` handles the final formatting of messages to put on the `Socket`'s `OutputStream`, and reverses the process for messages coming in on the `Socket`'s `InputStream`.

For example, perhaps your server needs to know the number of bytes in the message being sent. `StreamAdapter` could handle computing and prepending the length to the message before sending it. When the server receives it, the same `StreamAdapter` could handle stripping off the length and reading the correct number of bytes for building a `StreamReadyObject`.

The server side

The picture is similar on the server side:



We wrapped our `ServerSocket` in a `ServerSocketFacade`, which is `Runnable` and owns an instance of a `ServerSocket`. Our applications can instantiate a `ServerSocketFacade` with a particular server-side port to listen to and a maximum number of client connections allowed (the default is 50). The application then runs the Facade in a new `Thread` to hide the `ServerSocket` interaction details.

The `run()` method on `ServerSocketFacade` calls `acceptConnections()`, which makes a new `ServerSocket` and calls `accept()` on it to block until a client requests a connection. Each time that happens, our `ServerSocketFacade` wakes up and hands the new `Socket` returned by `accept()` to an instance of `SocketHandler` by calling

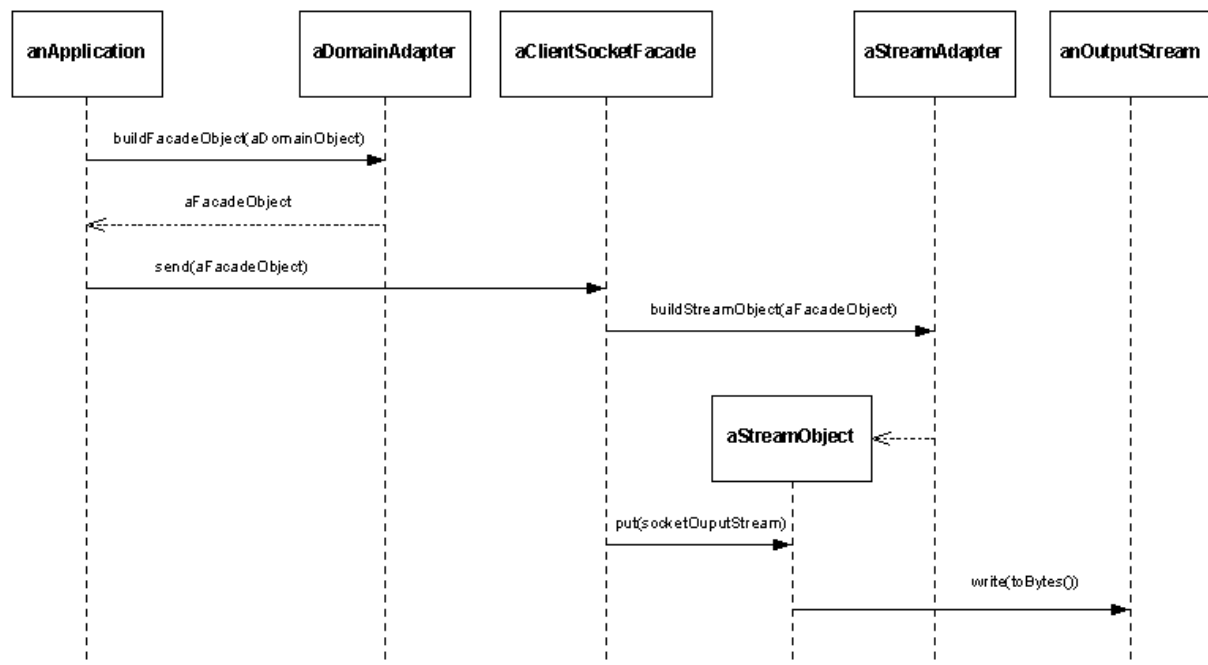
`handleSocket()`. The `SocketHandler` does what it needs to do in order to handle the new channel from client to server.

The business logic

Once we had these `Socket Facades` in place, it became much easier to implement the business logic of our application. Our application used an instance of `ClientSocketFacade` to send data over the `Socket` to the server and to get responses back. The application was responsible for handling conversion of our domain objects into formats understood by `ClientSocketFacade` and for building domain objects from responses.

Sending messages to the server

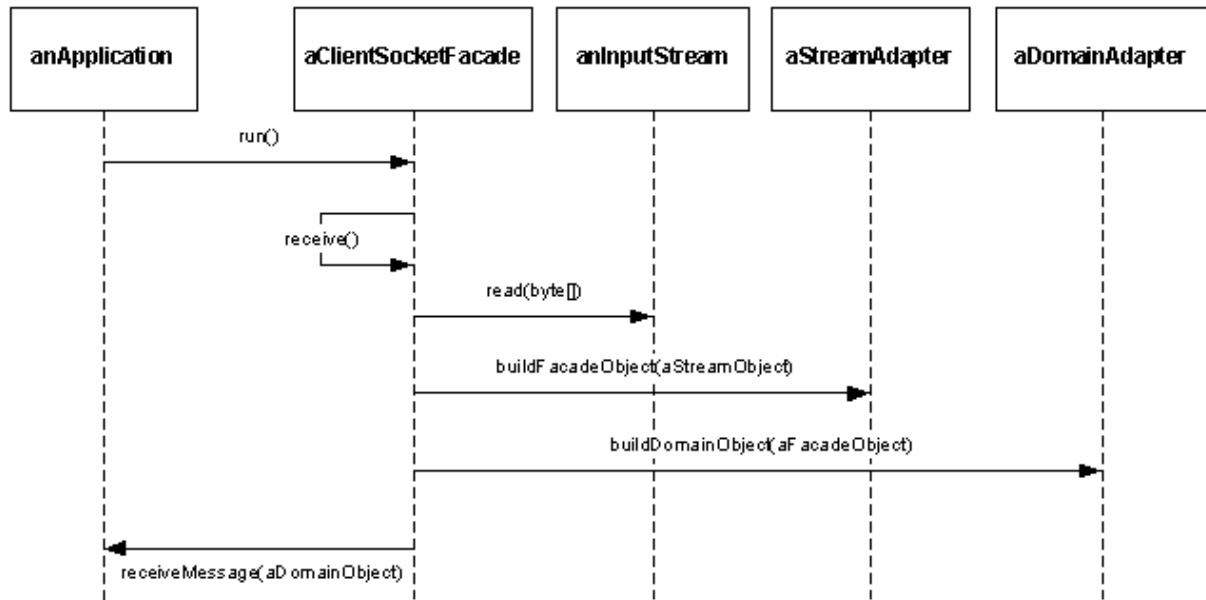
The following diagram shows the UML interaction diagram for sending a message in our application:



For simplicity's sake, we didn't show the piece of the interaction where `aClientSocketFacade` asks its `Socket` instance for its `OutputStream` (using the `getOutputStream()` method). Once we had a reference to that `OutputStream`, we simply interacted with it as shown in the diagram. Notice that our `ClientSocketFacade` hid the low-level details of socket interaction from our application. Our application interacted with `aClientSocketFacade`, not with any of the lower-level classes that facilitate putting bytes on `Socket OutputStreams`.

Receiving messages from the server

The following diagram shows the UML interaction diagram for receiving a message in our application:



Notice that our application runs `aClientSocketFacade` in a Thread. When `aClientSocketFacade` starts up, it tells itself to `receive()` on its `Socket` instance's `InputStream`. The `receive()` method calls `read(byte[])` on the `InputStream` itself. The `read([])` method blocks until it receives data, and puts the bytes received on the `InputStream` into a byte array. When data comes in, `aClientSocketFacade` uses `aStreamAdapter` and `aDomainAdapter` to construct (ultimately) a domain object that our application can use. Then it hands that domain object back to the application. Again, our `ClientSocketFacade` hides the lower-level details from the application, simplifying the Application Layer.

Section 8. Summary

Wrapup

The Java language simplifies using sockets in your applications. The basics are really the `Socket` and `ServerSocket` classes in the `java.net` package. Once you understand what's going on behind the scenes, these classes are easy to use. Using sockets in real life is simply a matter of using good OO design principles to preserve encapsulation within the various layers within your application. We showed you a few classes that can help. The structure of these classes hides the low-level details of `Socket` interaction from our application -- it can just use pluggable `ClientSocketFacades` and `ServerSocketFacades`. You still have to manage the somewhat messy byte details somewhere (within the Facades), but you can do it once and be done with it. Better still, you can reuse these lower-level helper classes on future projects.

Resources

- * Download the [source code](#) for this article.
- * "[Thinking in Java, 2nd Edition](#)" (Prentice Hall, 2000) by Bruce Eckel provides an excellent approach for learning Java inside and out.
- * Sun has a good tutorial on [Sockets](#). Just follow the "All About Sockets" link.
- * We used VisualAge for Java, version 3.5 to develop the code in this tutorial. Download your own copy of [VisualAge for Java](#) (now in release 4) or, if you already use VAJ, check out the [VisualAge Developer Domain](#) for a variety of technical assistance.
- * Now that you're up to speed with sockets programming with Java, this article on the Visual Age for Java Developer Domain will teach you to [set up access to sockets through the company firewall](#).
- * Allen Holub's [Java Toolbox column](#) (on *JavaWorld*) provides an excellent series on Java Threads that is well worth reading. Start the series with "[A Java programmer's guide to threading architectures](#)." One particularly good article, "[Threads in an object-oriented world, thread pools, implementing socket 'accept' loops](#)," goes into rather deep detail about `Thread` pooling. We didn't go into quite so much detail in this tutorial, and we made our `PooledRemoteFileServer` and `PooledConnectionHandler` a little easier to follow, but the strategies Allen talks about would fit nicely. In fact, his treatment of `ServerSocket` via a Java implementation of a callback mechanism that supports a multi-purpose, configurable server is powerful.
- * For technical assistance with multithreading in your Java applications, visit the [Multithreaded Java programming discussion forum](#) on developerWorks, moderated by Java threading expert Brian Goetz.
- * Siva Visveswaran explains connection pooling in detail in "[Connection pools](#)" (developerWorks, October 2000).

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Section 9. Appendix

Code listing for URLClient

```
import java.io.*;
import java.net.*;
public class URLClient {
    protected HttpURLConnection connection;
    public String getDocumentAt(String urlString) {
        StringBuffer document = new StringBuffer();
        try {
            URL url = new URL(urlString);
            URLConnection conn = url.openConnection();
            BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream()));
            String line = null;
            while ((line = reader.readLine()) != null)
                document.append(line + "\n");
            reader.close();
        } catch (MalformedURLException e) {
            System.out.println("Unable to connect to URL: " + urlString);
        } catch (IOException e) {
            System.out.println("IOException when connecting to URL: " + urlString);
        }
        return document.toString();
    }
    public static void main(String[] args) {
        URLClient client = new URLClient();
        String yahoo = client.getDocumentAt("http://www.yahoo.com");
        System.out.println(yahoo);
    }
}
```

Code listing for RemoteFileClient

```
import java.io.*;
import java.net.*;
public class RemoteFileClient {
    protected BufferedReader socketReader;
    protected PrintWriter socketWriter;
    protected String hostIp;
    protected int hostPort;
    public RemoteFileClient(String aHostIp, int aHostPort) {
        hostIp = aHostIp;
        hostPort = aHostPort;
    }
    public String getFile(String fileNameToGet) {
        StringBuffer fileLines = new StringBuffer();
        try {
            socketWriter.println(fileNameToGet);
            socketWriter.flush();
            String line = null;
            while ((line = socketReader.readLine()) != null)
                fileLines.append(line + "\n");
        } catch (IOException e) {
            System.out.println("Error reading from file: " + fileNameToGet);
        }
        return fileLines.toString();
    }
    public static void main(String[] args) {
```

```

        RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
        remoteFileClient.setUpConnection();
        String fileContents = remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt");
        remoteFileClient.tearDownConnection();
        System.out.println(fileContents);
    }
    public void setUpConnection() {
        try {
            Socket client = new Socket(hostIp, hostPort);
            socketReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
            socketWriter = new PrintWriter(client.getOutputStream());
        } catch (UnknownHostException e) {
            System.out.println("Error setting up socket connection: unknown host at " + hostIp);
        } catch (IOException e) {
            System.out.println("Error setting up socket connection: " + e);
        }
    }
    public void tearDownConnection() {
        try {
            socketWriter.close();
            socketReader.close();
        } catch (IOException e) {
            System.out.println("Error tearing down socket connection: " + e);
        }
    }
}

```

Code listing for RemoteFileServer

```

import java.io.*;
import java.net.*;
public class RemoteFileServer {
    int listenPort;
    public RemoteFileServer(int aListenPort) {
        listenPort = aListenPort;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
        }
    }
    public void handleConnection(Socket incomingConnection) {
        try {
            OutputStream outputToSocket = incomingConnection.getOutputStream();
            InputStream inputFromSocket = incomingConnection.getInputStream();
            BufferedReader streamReader = new BufferedReader(new InputStreamReader(inputFromSocket));
            FileReader fileReader = new FileReader(new File(streamReader.readLine()));
            BufferedReader bufferedFileReader = new BufferedReader(fileReader);
            PrintWriter streamWriter = new PrintWriter(incomingConnection.getOutputStream());
            String line = null;
            while ((line = bufferedFileReader.readLine()) != null) {
                streamWriter.println(line);
            }
        }
    }
}

```

```
    }
    fileReader.close();
    streamWriter.close();
    streamReader.close();
} catch (Exception e) {
    System.out.println("Error handling a client: " + e);
}
}
public static void main(String[] args) {
    RemoteFileServer server = new RemoteFileServer(3000);
    server.acceptConnections();
}
}
```

Code listing for MultithreadedRemoteFileServer

```
import java.io.*;
import java.net.*;
public class MultithreadedRemoteFileServer {
    protected int listenPort;
    public MultithreadedRemoteFileServer(int aListenPort) {
        listenPort = aListenPort;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort, 5);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
        }
    }
    public void handleConnection(Socket connectionToHandle) {
        new Thread(new ConnectionHandler(connectionToHandle)).start();
    }
    public static void main(String[] args) {
        MultithreadedRemoteFileServer server = new MultithreadedRemoteFileServer(3000);
        server.acceptConnections();
    }
}
```

Code listing for ConnectionHandler

```
import java.io.*;
import java.net.*;
public class ConnectionHandler implements Runnable {
    protected Socket socketToHandle;
    public ConnectionHandler(Socket aSocketToHandle) {
        socketToHandle = aSocketToHandle;
    }
    public void run() {
        try {
            PrintWriter streamWriter = new PrintWriter(socketToHandle.getOutputStream());
        }
    }
}
```

```
        BufferedReader streamReader = new BufferedReader(new InputStreamReader(socket));
        String fileToRead = streamReader.readLine();
        BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));
        String line = null;
        while ((line = fileReader.readLine()) != null)
            streamWriter.println(line);
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
}
```

Code listing for PooledRemoteFileServer

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledRemoteFileServer {
    protected int maxConnections;
    protected int listenPort;
    protected ServerSocket serverSocket;
    public PooledRemoteFileServer(int aListenPort, int maxConnections) {
        listenPort = aListenPort;
        this.maxConnections = maxConnections;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort, 5);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
        }
    }
    protected void handleConnection(Socket connectionToHandle) {
        PooledConnectionHandler.processRequest(connectionToHandle);
    }
    public static void main(String[] args) {
        PooledRemoteFileServer server = new PooledRemoteFileServer(3000, 3);
        server.setUpHandlers();
        server.acceptConnections();
    }
    public void setUpHandlers() {
        for (int i = 0; i < maxConnections; i++) {
            PooledConnectionHandler currentHandler = new PooledConnectionHandler();
            new Thread(currentHandler, "Handler " + i).start();
        }
    }
}
```

Code listing for PooledConnectionHandler

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledConnectionHandler implements Runnable {
    protected Socket connection;
    protected static List pool = new LinkedList();
    public PooledConnectionHandler() {
    }
    public void handleConnection() {
        try {
            PrintWriter streamWriter = new PrintWriter(connection.getOutputStream());
            BufferedReader streamReader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
            String fileToRead = streamReader.readLine();
            BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));
            String line = null;
            while ((line = fileReader.readLine()) != null)
                streamWriter.println(line);
            fileReader.close();
            streamWriter.close();
            streamReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("Could not find requested file on the server.");
        } catch (IOException e) {
            System.out.println("Error handling a client: " + e);
        }
    }
    public static void processRequest(Socket requestToHandle) {
        synchronized (pool) {
            pool.add(pool.size(), requestToHandle);
            pool.notifyAll();
        }
    }
    public void run() {
        while (true) {
            synchronized (pool) {
                while (pool.isEmpty()) {
                    try {
                        pool.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                connection = (Socket) pool.remove(0);
            }
            handleConnection();
        }
    }
}
```

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial [Building tutorials with the Toot-O-Matic](#) demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.

Chapter 2: Java Sockets

As we saw in the last chapter *Socket* is an abstraction of an *IP Port*. Sockets are a concept that have been around in programming languages for some time. They first appeared in early Unix systems in the 1970s and are now the 'standard' low-level communication primitive. Prior to Java, they were fairly painful to use - but now you can build applications using them quite straightforwardly.

Actually, there are two kinds of sockets - *connection-oriented* sockets, almost always based on TCP, and *connectionless* sockets, usually based on *User Datagram Protocol* (UDP). The difference is that TCP-type sockets guarantee data arrives, and in the correct order whereas UDP-type ones do not.

In addition to distinguishing TCP-type and UDP-type sockets, we must also distinguish - in Java - between *client* and *server* sockets. It might seem obvious that 'server sockets run on servers and client sockets on clients' but this is not always true - servers can be both servers *and* clients, and clients can legitimately run server sockets. The real difference is that server sockets wait for connections to be initiated by client sockets - the naming is not particularly helpful.

2.1. Client Sockets

We will start off by looking at how we can create a simple client socket. The class that handles client sockets is *Socket* in the *java.net* package. You would generally include the line:

```
import java.net.Socket;
```

in your code to allow you to use the 'short' names of the package methods. A complete description of the methods and constructors of *Socket* can be found *here*. As you can see, there are a range of constructors that can be used to create a new [client] socket - the one we would use would depend on our particular circumstances. Probably the simplest from our point of view is

```
Socket(String host, int port)
```

which simply connects to port on machine host. For example,

```
Socket s = new Socket("this.doesnt.exist.com", 1024);
```

Note that *you must not use socket number less than 1024* in your own applications. These are reserved for well-known services (though if you are actually trying to connect to a specific well-known service, then you should of course use the appropriate port number). On systems with a well-developed notion of access priviledges (e.g. linux), 'ordinary' users - i.e. you - cannot create server sockets (below) on well-known ports (you need *superuser* - e.g. 'root' - access).

How do you read and write from/to the socket? You simply use standard Java I/O methods. For example, the following code opens a socket and attaches appropriate input/output streams to it:

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

Socket s = new Socket("this.doesnt.exist.com", 1024);
BufferedReader in = new BufferedReader(
    new InputStreamReader(s.getInputStream()));
PrintStream out =
    new PrintStream(s.getOutputStream());
```

(Lines of codes between the `import` statement and the `Socket` declaration have been omitted.) The key bits are (a) `s = new Socket(...)`, which creates a new socket `s`; (b) `s.getInputStream()`, which returns the *input stream* associated with the socket `s`; and (c) `s.getOutputStream()` which returns the corresponding *output stream*. The other stuff (`BufferedReader`, `InputStreamReader`, `PrintStream`) are exactly the same methods we would use if we wished to do line-based IO from files: they 'wrap' the raw input stream with something more convenient and useable. So we can make socket-based communication look more-or-less exactly like file-based communication.

We can now (say) input (`inString = in.readLine();`) and output (`out.println("blah");`) lines of text in the normal way. (It has to be said though, that Java's stream-based IO, though very flexible and capable, is not the easiest thing to come to grips with initially.)

2.2. Server Sockets

A key question you should have after the previous section is: who is listening to the other end of our socket? The answer is, unless we set up a *server socket* or are connecting to a pre-existing service, nobody, and the connection attempt will typically fail. (We must be a little careful when trying to open sockets because if there is nobody listening, an *exception* can be *thrown*. We are generally required to *catch* and deal with such exceptions - not arranging to catch most exceptions causes compilation errors). We will look at exceptions some more in the *example*.) The class that deals with server sockets is (no surprise) `ServerSocket` and you can find API details *here*. The principal constructor from our point of view is

```
ServerSocket(int port)
```

which creates a server socket that *listens* on the specified port. To set up a server socket and establish a connection, we might use code like this:

```
import java.net.*;

ServerSocket sSoc = new ServerSocket(1024);

Socket in = sSoc.accept();
```

We create a new server socket called `sSoc` and then call its `accept()` method. At this point, our code will *wait* until some other process (probably on another machine, though this does not have to be the case) connects to the server socket, by automatically creating a new *client* socket. The `accept` method *blocks* - that is, any program invoking it will wait until a communication attempt occurs (there are other versions of `accept` that specify a maximum waiting time). Once again, we should not use socket numbers less than 1024 because they are reserved for well-known services.

2.2.1. Simple Example

Here is an example program that repeatedly waits until it is contacted on its server socket. When it is, it starts up a new *thread* that outputs the first 100 *Fibonacci Numbers* (if you don't know what a Fibonacci number is, it doesn't particularly matter).

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

public class Fib1 {

    public static void main(String argv[]) {

        try {
            ServerSocket sSoc = new ServerSocket(2001);

            while(true) {
                Socket inSoc = sSoc.accept();

                FibThread FibT = new FibThread(inSoc);
                FibT.start();}
        }
        catch (Exception e) {
            System.out.println("Oh Dear! " +
                e.toString());
        }
    }
}

class FibThread extends Thread {
```



```

Socket threadSoc;

int F1 = 1;
int F2 = 1;

FibThread(Socket inSoc) {
    threadSoc = inSoc;
}

public void run() {
    try {
        PrintStream FibOut = new
            PrintStream(threadSoc.getOutputStream());

        for (int i=0; i < 100; i++) {
            int temp;

            temp = F1;
            FibOut.println(F1);

            F1 = F2;
            F2 = temp + F2;
        }
    }
    catch (Exception e) {
        System.out.println("Whoops! " +
            e.toString());
    }

    try {
        threadSoc.close();
    }
    catch (Exception e) {
        System.out.println("Oh no! " +
            e.toString());
    }
}
}

```

There are a few things to note about this code.

- **The chosen socket number.** We have picked socket 2001 at random - in principle, it doesn't matter which socket we pick, provided it is greater than 1023. Of course, there is a possibility that 2001 is in use already - sockets above 1023 are free for anyone to use.
- **Exceptions** We have used `try - catch` to make sure that any exceptions that are thrown are caught and handled. In this case, 'handling' is a little basic - we have just specified that any exception should result in a message being printed, which will include the actual details of the exception. In practice, we might want several different `catch` clauses to deal with the various different *classes* of exception that can be thrown. For example, both the server socket constructor `ServerSocket` and the method `accept` can throw exceptions of class `IOException`, which is a sub-class of `Exception`, and which in turn has its own subclasses - the most likely of which in this case is `SocketException` (which *itself* has subclasses). Different sub-classes of exception may need to be handled in different ways - for example, it is likely that not all exceptions will be terminal, and the `catch` clause can attempt some form of recovery. This might happen if socket 2001 is in use - we could try another one and continue. However, this is too subtle for us at the moment.
- **Threads.** When `Fib1` actually receives some input, and hence creates a new socket, it passes off all further work to a new thread called `FibThread`. It is `FibThread` that outputs the Fibonacci numbers.

Notice that `FibThread` also has to deal with exceptions - both `getOutputStream` (which returns an output stream connected to a socket) and `close` (which closes a socket connection) can throw an `IOException`. Why have we used a separate thread? Because our server then goes back to listening for more communication attempts, and it is perfectly possible for it to serve multiple clients at once. That is because each client will be served by a separate thread and socket, and multiple sockets are able to 'share' a port.

2.3. Client-Side Code

Now we have seen what a server must do to communicate over sockets, we need to look at the client-side code. The following is a simple client for Fib1.

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

public class FibReader {

    Socket appSoc;
    BufferedReader in;
    String message;

    public static void main(String argv[]) {
        try {
            appSoc = new Socket("wherever",2001);
            in = new BufferedReader(new
                InputStreamReader(appSoc.getInputStream()));
            for (int i = 0; i < 100; i++) {
                message = in.readLine();
                System.out.println(message); }

        }
        catch (Exception e) {
            System.out.println("Died... " +
                e.toString());
        }
    }
}
```

This code creates a socket connection to some host (on which an appropriate server must be running), and sets up an input stream connected to that socket. It then prints out 100 lines read from the server, using the socket. Again, this is not particularly good code - for example, we are relying on the server to send us 100 lines: what if it doesn't? However, it does illustrate the point.

2.4. Exercise

Here is a simple exercise you can try to illustrate that socket communication can work. The java source code file for the server is [here](#). Download it to your machine. Then download the *client*. You will probably have to edit the hostname in the client code - try localhost to start with. Start up the server in a terminal, or command, window. You might want to do this as a background task - on windows type:

```
start java Fib1
```

and on linux:

```
java Fib1 &
```

Then type:

```
java FibReader
```

and you should see Fibonacci numbers appearing (they will start to go horribly wrong eventually because of arithmetic overflow).

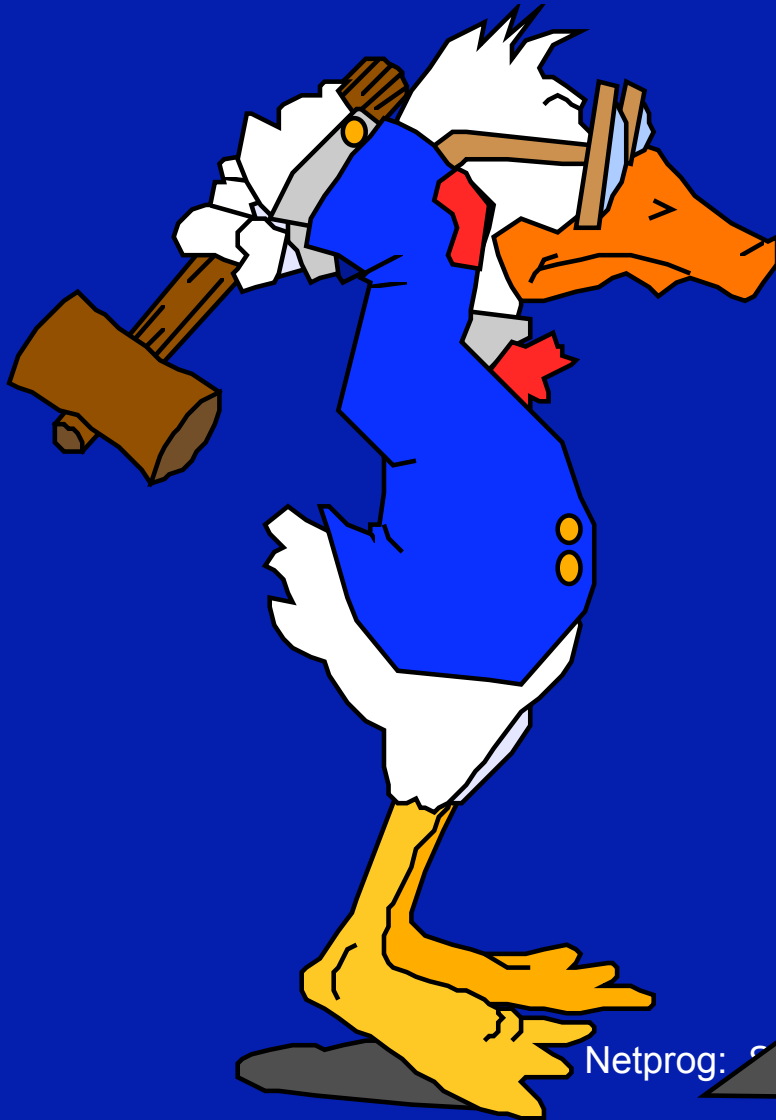
For experiment two, try doing the same as above, but this time open an extra terminal/command window and start the client up in both - you should see the Fibonacci numbers appearing more-or-less simultaneously on both.

The next experiment to try is to use two machines. Start the server up on one, and the client on the other - remember to edit the hostname in the client (or change the code so it takes command line parameters allowing you to specify a host. How exactly you manage to use two machines will depend on your circumstances - if you have access to the Linux laboratory, it is very straightforward. It might also be easy if you have access to

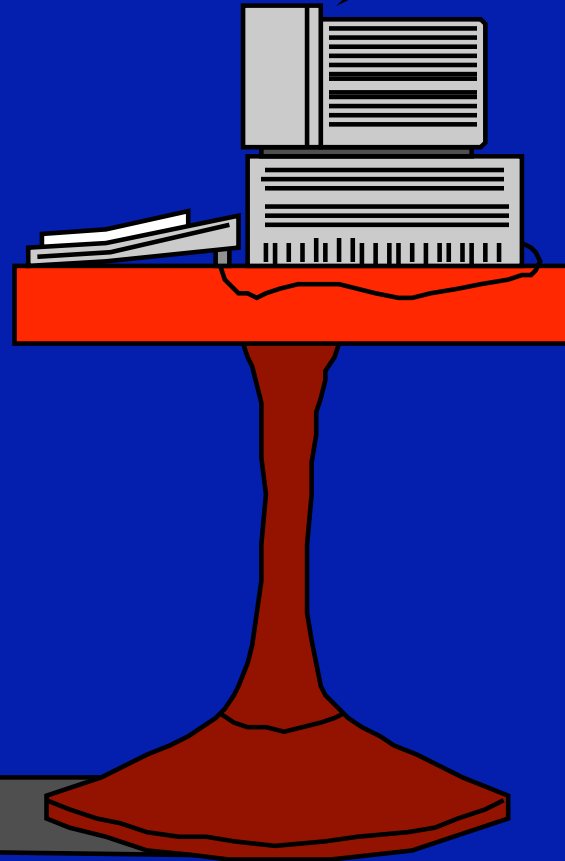
multiple machines at home (though beware that a firewall will almost certainly stop it working - in this case you can either disable the firewall or open up the port you are using - the safest option).

If you have internet access from outside the university, try the experiment again. (*However, you will need to start up the server process remotely, e.g. via TelNet, or better ssh - don't start it up in the lab and then go home!* This may seem obvious but it's been done.) You may find that it doesn't work this time. (I would like to say that this is a consistent security policy by the University to protect machines from suspicious communication attempts - but it appears to be purely random.)

Sockets Programming



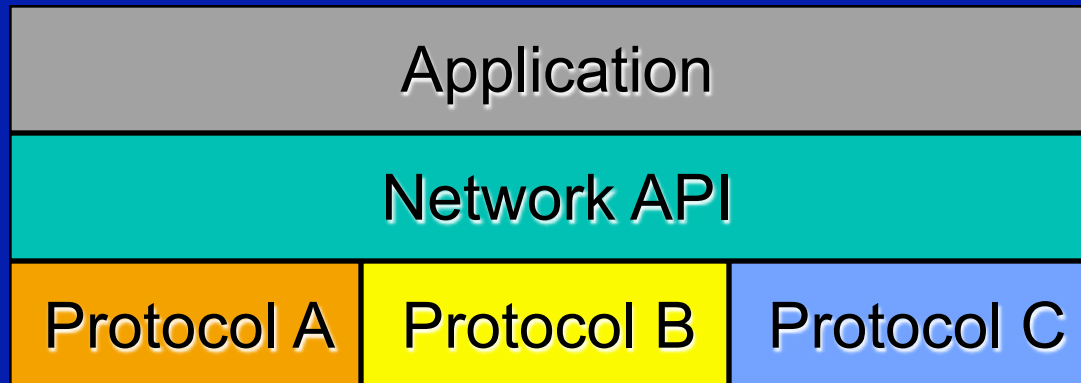
Socket to me!



Netprog: 5

Network Application Programming Interface (API)

- The services provided (often by the operating system) that provide the interface between application and protocol software.



Network API wish list

- Generic Programming Interface.
- Support for message oriented and connection oriented communication.
- Work with existing I/O services (when this makes sense).
- Operating System independence.
- Presentation layer services

Generic Programming Interface

- Support multiple communication protocol suites (families).
- Address (endpoint) representation independence.
- Provide special services for Client and Server?

TCP/IP

- TCP/IP does not include an API definition.
- There are a variety of APIs for use with TCP/IP:
 - Sockets
 - TLI, XTI
 - Winsock
 - MacTCP

Functions needed:

- Specify local and remote communication endpoints
- Initiate a connection
- Wait for incoming connection
- Send and receive data
- Terminate a connection gracefully
- Error handling

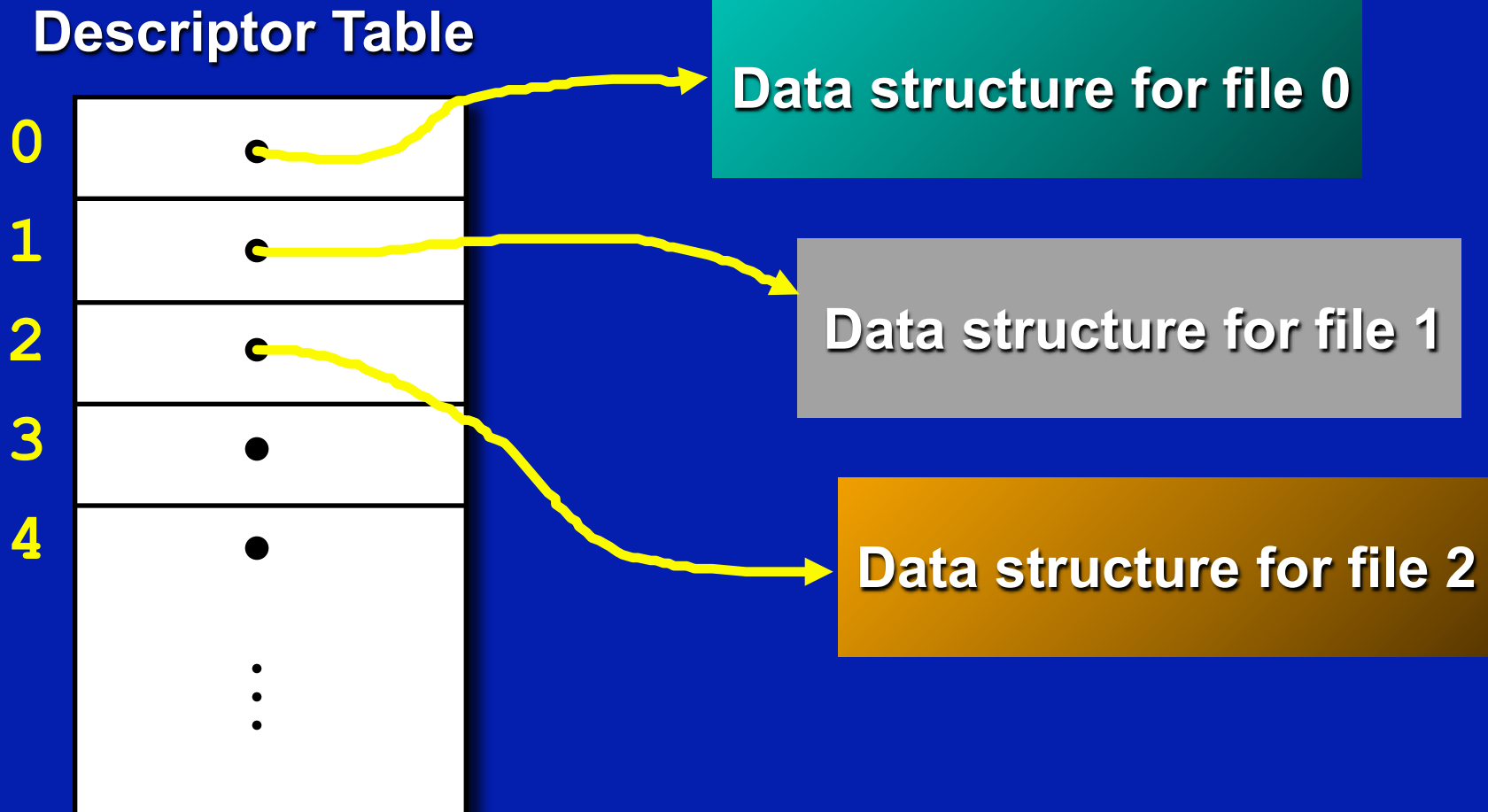
Berkeley Sockets

- Generic:
 - support for multiple protocol families.
 - address representation independence
- Uses existing I/O programming interface as much as possible.

Socket

- A socket is an abstract representation of a communication endpoint.
- Sockets work with Unix I/O services just like files, pipes & FIFOs.
- Sockets (obviously) have special needs:
 - establishing a connection
 - specifying communication endpoint addresses

Unix Descriptor Table



Socket Descriptor Data Structure

Descriptor Table

0	•
1	•
2	•
3	•
4	•
	•
	•
	•

Family: PF_INET
Service: SOCK_STREAM
Local IP: 111.22.3.4
Remote IP: 123.45.6.78
Local Port: 2249
Remote Port: 3726

Creating a Socket

```
int socket(int family, int type, int proto);
```

- `family` specifies the protocol family (**PF_INET** for TCP/IP).
- `type` specifies the type of service (**SOCK_STREAM**, **SOCK_DGRAM**).
- `protocol` specifies the specific protocol (usually 0, which means *the default*).

socket ()

- The `socket ()` system call returns a socket descriptor (small integer) or `-1` on error.
- `socket ()` allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.

Specifying an Endpoint Address

- Remember that the sockets API is generic.
- There must be a generic way to specify endpoint addresses.
- TCP/IP requires an IP address and a port number for each endpoint address.
- Other protocol suites (families) may use other schemes.

Necessary Background Information: POSIX data types

<code>int8_t</code>	signed 8bit int
<code>uint8_t</code>	unsigned 8 bit int
<code>int16_t</code>	signed 16 bit int
<code>uint16_t</code>	unsigned 16 bit int
<code>int32_t</code>	signed 32 bit int
<code>uint32_t</code>	unsigned 32 bit int

`u_char, u_short, u_int, u_long`

More POSIX data types

`sa_family_t`

address family

`socklen_t`

length of struct

`in_addr_t`

IPv4 address

`in_port_t`

IP port number

Generic socket addresses

```
struct sockaddr {  
    uint8_t    sa_len;   
    sa_family_t sa_family;  
    char       sa_data[14];  
};
```

Used by kernel



- `sa_family` specifies the address type.
- `sa_data` specifies the address value.

sockaddr

- An address that will allow me to use sockets to communicate with my kids.
- address type **AF_DAVESKIDS**
- address values:

Andrea	1	Mom	5
Jeff	2	Dad	6
Robert	3	Dog	7
Emily	4		

AF_DAVESKIDS

- Initializing a sockaddr structure to point to Robert:

```
struct sockaddr robster;
```

```
robster.sa_family = AF_DAVESKIDS;
```

```
robster.sa_data[0] = 3;
```

Really old picture! →

Netprog: Sockets API



AF_INET

- For AF_DAVESKIDS we only needed 1 byte to specify the address.
- For AF_INET we need:
 - 16 bit port number
 - 32 bit IP address

IPv4 only!



struct sockaddr_in (IPv4)

```
struct sockaddr_in {
    uint8_t          sin_len;
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

A special kind of sockaddr structure

struct in_addr

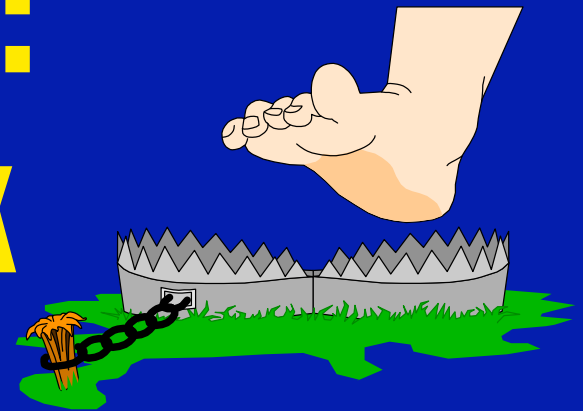
```
struct in_addr {  
    in_addr_t    s_addr;  
};
```

`in_addr` just provides a name for the 'C' type associated with IP addresses.

Network Byte Order

- All values stored in a `sockaddr_in` must be in network byte order.
 - `sin_port` a TCP/IP port number.
 - `sin_addr` an IP address.

**Common Mistake:
Ignoring Network
Byte Order**



Network Byte Order Functions

'h' : host byte order

'n' : network byte order

's' : short (16bit)

'l' : long (32bit)

```
uint16_t htons (uint16_t) ;  
uint16_t ntohs (uint16_t) ;
```

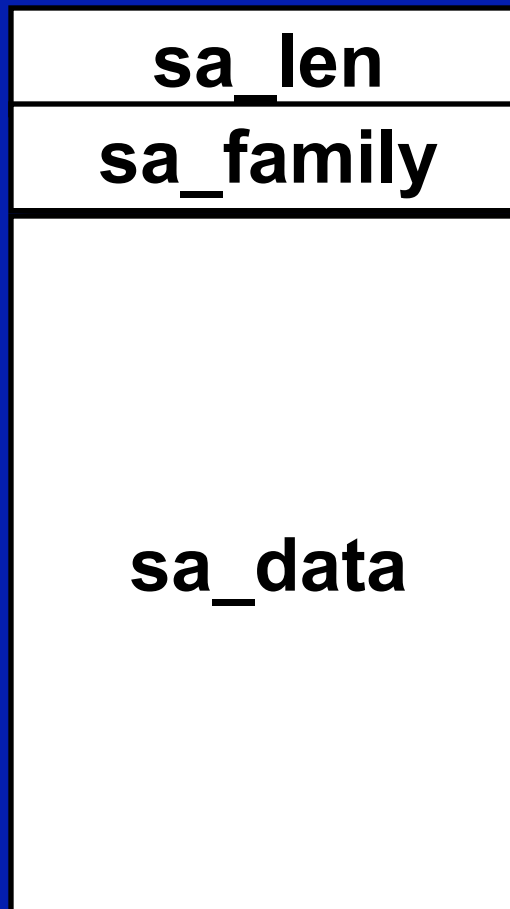
```
uint32_t htonl (uint32_t) ;  
uint32_t ntohl (uint32_t) ;
```

TCP/IP Addresses

- We don't need to deal with `sockaddr` structures since we will only deal with a real protocol family.
- We can use `sockaddr_in` structures.

BUT: The C functions that make up the sockets API expect structures of type `sockaddr`.

sockaddr




sockaddr_in



Assigning an address to a socket

- The `bind()` system call is used to assign an address to an existing socket.

```
int bind( int sockfd,  
          const struct sockaddr *myaddr,  
          int addrlen);  
const! 
```

- `bind` returns 0 if successful or -1 on error.

bind()

- calling `bind()` assigns the address specified by the `sockaddr` structure to the socket descriptor.
- You can give `bind()` a `sockaddr_in` structure:

```
bind( mysock,  
      (struct sockaddr*) &myaddr,  
      sizeof(myaddr) );
```

bind() Example

```
int mysock, err;
struct sockaddr_in myaddr;

mysock = socket(PF_INET, SOCK_STREAM, 0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons( portnum );
myaddr.sin_addr = htonl( ipaddress);

err=bind(mysock, (sockaddr *) &myaddr,
        sizeof(myaddr));
```


Uses for `bind()`

- There are a number of uses for `bind()`:
 - Server would like to bind to a well known address (port number).
 - Client can bind to a specific port.
 - Client can ask the O.S. to assign *any available* port number.

Port schmort - who cares ?

- Clients typically don't care what port they are assigned.
- When you call bind you can tell it to assign you any available port:

```
myaddr.port = htons(0);
```

What is my IP address ?

- How can you find out what your IP address is so you can tell `bind()` ?
- There is no realistic way for you to know the right IP address to give `bind()` - what if the computer has multiple network interfaces?
- specify the IP address as: `INADDR_ANY`, this tells the OS to take care of things.

IPv4 Address Conversion

```
int inet_aton( char *, struct in_addr *);
```

Convert ASCII dotted-decimal IP address to network byte order 32 bit value. Returns 1 on success, 0 on failure.

```
char *inet_ntoa(struct in_addr);
```

Convert network byte ordered value to ASCII dotted-decimal (a string).

Other socket system calls

- General Use

- `read()`
- `write()`
- `close()`

- Connection-oriented (TCP)

- `connect()`
- `listen()`
- `accept()`

- Connectionless (UDP)

- `send()`
- `recv()`

Concurrent Programming using Threads

Threads are a control mechanism that enable you to write concurrent programs. You can think of a thread in an object-oriented language as a special kind of “system object” that contains information about the state of execution of a sequence of function calls that are said to “execute as a thread”. Usually, a special “run” or “start” procedure starts a separate thread of control.

Normally, when you call a function or procedure, the compiler sets-up a stack frame (also called an activation frame) on the run-time procedure call stack, pushes arguments (or puts them into registers), and calls the function. The stack is also used as temporary storage for locally allocated objects declared in the scope of a procedure.

In a sequential program, there is only one run-time stack and all activation frames are allocated in a nested fashion on the same run-time stack, corresponding to each nested procedure call. In a multi-threaded application, each “thread” represents a separate run-time stack, so you can have multiple procedure call chains running at the same time, possibly on multiple processors. Java on Solaris supports multi-processor threads.

In a sequential program, the main run-time stack is allocated at program start and all procedure calls, including the initial call to “main” are made on this single run-time stack. In a multi-threaded program, a program starts on the system run-time stack where the main procedure runs. Any functions/procedures called by the main procedure have their activation frames allocated on this run-time stack.

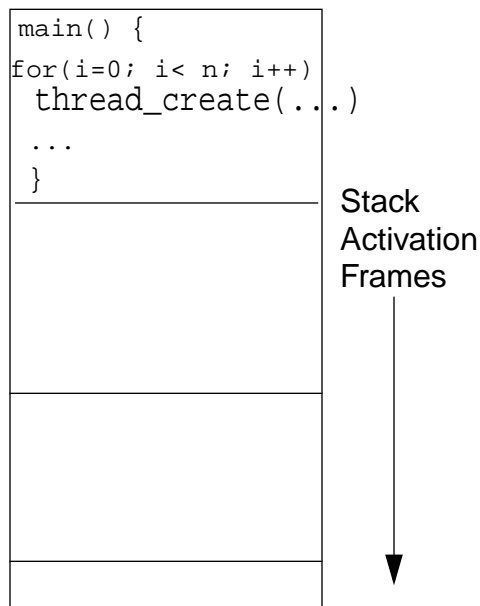
If the main procedure creates a new thread for run some procedure (usually calling a special “thread creation or construction” procedure/method), then a new run-time stack is dynamically allocated from the heap and the activation frames for the procedures are allocated on this new stack.

Thread Stacks

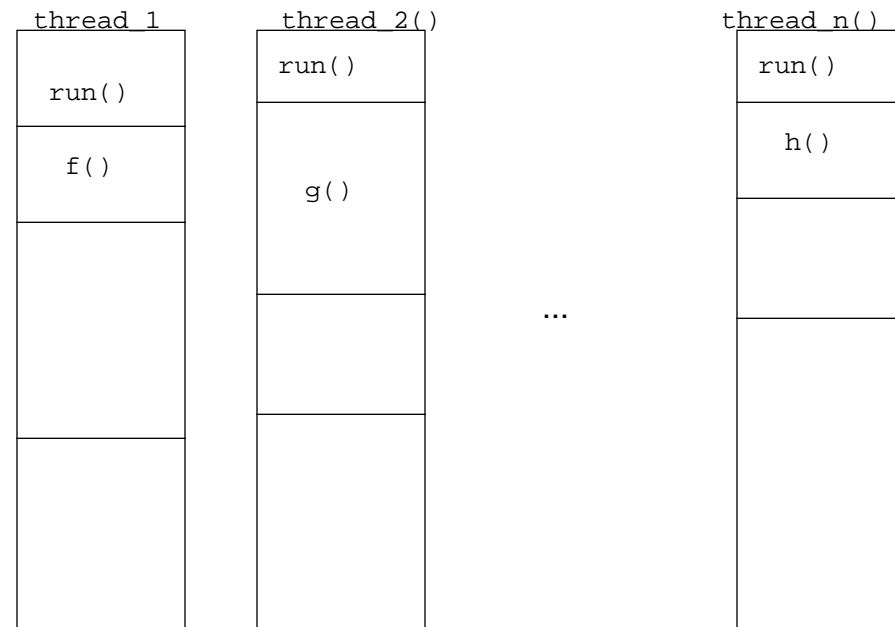
Question: How large should the heap allocated thread stack be?

A thread stack will contain the activation record of the “starting” thread procedure (e.g., called the “run” method in Java), as well as any procedures that are called by the procedure that was first started in the new thread of control. So, the thread stack needs to be large enough to hold the maximum number of bytes required to hold all the activation records of the deepest procedure call chain, as well as storage for all local variables allocated on the stack.

Main thread and run-time stack

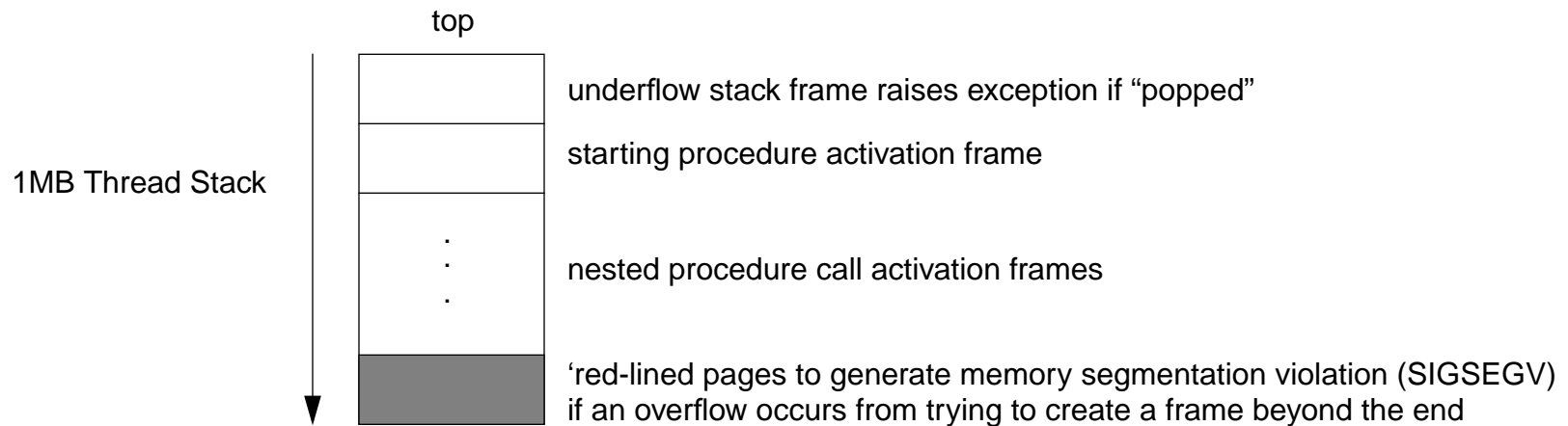


Multiple thread run-time stacks, each a separate “thread of execution”



Thread Stack Size

On operating systems that support processes with multiple threads of control, threads stacks are typically set at 1MB, consisting of contiguous virtual memory pages, that are allocated incrementally at run-time by the system. There may also be some extra pages at the “top” or “bottom” for some thread book-keeping. There is also usually an extra pages allocated “above” the top and “below” the bottom of the stack to detect overflow:



Java Threads

In Java, the same concepts about threads, threads stacks, and starting threads applies. The primary difference is that in Java, a thread is defined as a special class in the `java.lang` package. The thread class implements an interface called the “Runnable” interface, which defines a single abstract method called “run”.

```
// in the java.lang.Runnable file you will find the following interface
package java.lang;
```

```
public interface Runnable {
    public void run(); // just like a pure virtual function in C++
}
```

Since we are defining an interface, `run` is implicitly a “abstract” method. As we have seen, interfaces in Java define a set of methods with NO implementation. Some class must “implement” the `Runnable` interface and provide an implementation of the `run` method, which is the method that is started by a thread. Java provides a “Thread” class, that implements the `Runnable` interface, but does not implement the `run`

```
public class ConcurrentReader implements Runnable {
    ...
    public void run() { /* code here executes concurrently with caller */ }
    ...
}
```

Defining a Thread in Java

To start this thread you need to first create an object of type `MyOwnThreadObjects`, bind it to a new `Thread` object, and then start it. Calling `start` creates the thread stack for the thread, and then invoked the `run()` method as the first procedure on that new thread stack.

```
ConcurrentReader readerThread = new ConcurrentReader();
Thread t = new Thread(readerThread); // create thread using a Runnable object
readerThread.start();
```

The `java.lang.Thread` class has a constructor that takes an object of type `Runnable`:

```
Thread(Runnable object); // must provide an object that implements run
```

Alternatively, we can define a subclass of the class `Thread` directly.

```
class ConcurrentWriter extends Thread {
    public void run() {
        // you provide the code here to run as a separate thread of control
    }
}
```

To start this thread you just need to do the following:

```
ConcurrentWriter writerThread = new ConcuirrentWriter();
writerThread.start(); // start calls run()
```

java.lang.Thread

```
public class Thread implements Runnable {
    private char name[];
    private Runnable target;
    ...
    public final static int MIN_PRIORITY = 1;
    public final static int NORM_PRIORITY = 5;
    public final static int MAX_PRIORITY = 10;

    private void init(ThreadGroup g, Runnable target, String name) {...}

    public Thread() { init(null, null, "Thread-" + nextThreadNum()); }
    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum());
    }
    public Thread(Runnable target, String name) { init(null, target, name); }

    public synchronized native void start();

    public void run() {
        if (target != null) {
            target.run();
        }
    }
    ...
}
```

java.lang.Thread

```
public class Thread implements Runnable {
    ...
    public static native Thread currentThread();
    public static native void yield();
    public static native void sleep(long millis) throws InterruptedException;
    public static int enumerate(Thread tarray[])

    public static boolean interrupted() { ... }
    public boolean isInterrupted() { ... }
    public final native boolean isAlive();
    public String toString() {
    public void interrupt() { ... }
    public void interrupt() { ... }
    public final void stop() { ... }
    public final void suspend() { ... }
    public final void resume() { ... }
    public final void setPriority(int newPriority) {
    public final int getPriority() {
    public final void setName(String name) { ... }
    public final String getName() { return String.valueOf(name); }
    public native int countStackFrames();
    public final synchronized void join() throws InterruptedException {...}
    public void destroy() { throw new NoSuchMethodError(); }
}
}
```

Extending Class Thread vs Implementing Interface Runnable

Q: Why does Java allow two different ways to provide thread objects? How do you decide when to extend the Thread class versus implementing the Runnable interface?

Java only allows single class inheritance, so if we have a class that needs to inherit from another class, but also needs to run as a thread, then we extend the other class and implement the Runnable interface. So, it is quite common for a class X to extend some class Y and implement the Runnable interface:

```
class X extends Y implements Runnable {  
  
    public synchronized void do_something() { ... }  
    public void run() { do_something(); } // can be run a thread if needed  
}
```

By implementing the Runnable interface, rather than extending the Thread class, you are communicating to the user of the class X that you expect that an object of type X will run as a thread, but it does not HAVE TO run as a thread. Since all the run() method does, in this case, is call another public method that could be called without running a thread, it gives the user the option of either having an object of type X run concurrently, or sequentially. A **synchronized** method is one that “locks” an object so that no other thread can execute inside the object while the method is active.

```
X obj = new X();  
obj.do_something(); // runs sequentially in the current thread  
Thread t = new Thread(new X()); // create an X and run as a thread  
t.start(); // start() calls run() which calls do_something() as
```

Synchronized Methods, Wait, Notify, and NotifyAll

An very interesting features of Java objects is that they are all “lockable” objects. That is, the `java.lang.Object` class implements an implicit locking mechanism that allows any Java object to be locked during the execution of a **synchronized method** or **synchronized block**, so that the thread that holds the lock gains exclusive access to the object for the duration of the method call or scope of the bloc. No other thread can “acquire” the object until the thread that holds the lock “releases” the object. This *synchronization policy* provides **mutual exclusion**.

Synchronized methods are methods that lock the object on entry and unlock the object on exit. The `Object` class implements some special methods for allowing a thread to explicitly release the lock while in the method, **wait** indefinitely or for some time interval, and then try to reacquire the lock when some condition is true. Two other methods allow a thread to signal waiting thread(s) to tell them to wakeup: the **notify** method signals one thread to wakeup and the **notifyAll** method signals all threads to wakeup and compete to try to re-acquire the lock on the object. This type of synchronized object is typically used to protect some shared resource, using two types of methods:

```
public synchronized void consume() {
    while (!consumable()) {
        wait();    // release lock and wait for resource
    }
    ... // have exclusive access to resource to consume
}
public synchronized void produce() {
    ... // change of state must result in consumable condition being true
    notifyAll(); // notify all waiting threads to try consuming
}
```

Synchronized Method vs Synchronized Block

The synchronized method declaration qualifier is syntactic sugar for the fact that the entire of scope of the procedure is to be governed by the mutual exclusion condition obtained by acquiring the object's lock:

```
public void consume () {
    synchronized(this) {
        // code for consuming
    }
}
```

A synchronized block allows the granularity of a lock to be finer-grained than a procedure scope. The argument given to the synchronized block is a reference to an object.

What about recursive locking of the same Object?

```
public class Foo {
    ...
    public void synchronized f() { ... }
    public void synchronized g() { ...; f(); ... }
}
```

If g() is called, and it then calls f(), what happens? What happens in the following case?

```
Foo f = new Foo;
synchronized(f) { ...; synchronized (f) { ... } ... }
```

Wait, Notify, and NotifyAll

Every object has access to wait, notify, and notify methods, which are inherited from class Object.

```
public class Object {  
    ...  
    public final native void notify();  
    public final native void notifyAll();  
  
    public final native void wait(long timeout) throws InterruptedException;  
    public final void wait() throws InterruptedException { wait(0); }  
    public final void wait(long timeout, int nanos)  
        throws InterruptedException { ... }  
}
```

The `Object.wait()` method implicitly releases the object's lock and the thread then waits on an internal queue associated with each object. The thread waits to be notified of when it can try to re-acquire the lock and test the condition again.

The `Object.notify()` method signals the highest priority thread closest to the front of the wait queue to wakeup. `Object.notifyAll()` wakes up all waiting threads, and they compete for the lock. The thread actually gets the lock is **non-deterministic** and not necessarily "fair". E.g., high priority threads in the wait queue could always win-out over lower priority threads, resulting in starvation since low priority threads never get access to the resource.

A Shared Queue Example

This is an example of a “Producer-Consumer” shared resource. Note that the `wait()`, `wait(timeout)`, `notify()`, and `notifyAll()` method can only be called from a synchronized method, or a method called by a synchronized method. I.e., the object must be in the locked state.

```
class SharedQueue {
    private Element head, tail;

    public boolean empty() { return head == tail; }

    public synchronized Element remove() {
        try { while (empty()) wait(); } // wait for an element in the queue
        catch (InterruptedException e) { return null; }
        Element p = head; head = head.next;
        if (head == null) tail = null;
        return p;
    }

    public synchronized void insert(Element p)
        if (tail == null) head = p;
        else tail.next = p;
        p.next = null;
        tail = p;
        notify(); // let one waiter know something is in the queue
    }
}
```

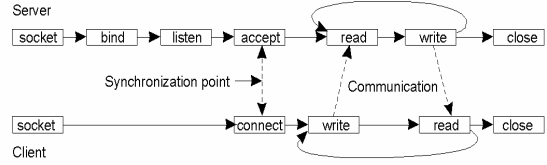
Berkeley Sockets (1)

□ Socket primitives for TCP/IP

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections (queue)
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Transport Layer 2

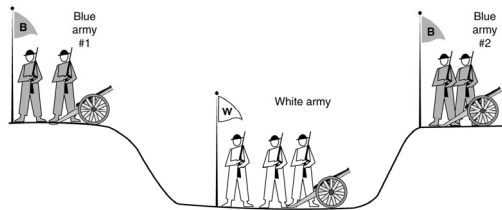
Berkeley Sockets (2)



□ Connection-oriented communication pattern using sockets

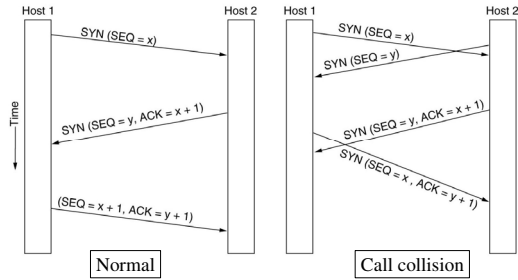
Transport Layer 3

2-army problem



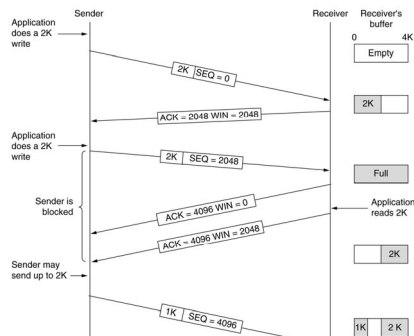
Transport Layer 4

3-way handshake



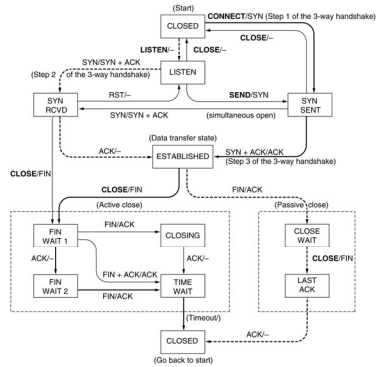
Transport Layer 5

TCP data transfer



6

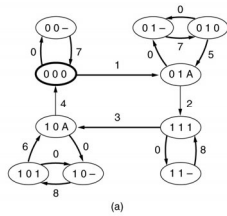
TCP FSM



7

Stop-and-wait

- Notation SRC: S = no. of frame being sent by sender, R = no. of frame being expected by receiver, C = state of channel (frame 0 or 1, or A = ACK, or "-" empty)



Transition	Who runs?	Frame accepted	Frame emitted	To network layer
0	-	(frame lost)	-	-
1	R	0	A	Yes
2	S	A	1	-
3	R	1	A	Yes
4	S	A	0	-
5	R	0	A	No
6	R	1	A	No
7	S	(timeout)	0	-
8	S	(timeout)	1	-

Transport Layer 8

Reliable delivery in ARPAnet: Concurrent logical channels

- Multiplex 8 logical channels over a single link
- Run stop-and-wait on each logical channel
- Maintain three state bits per channel
 - channel busy
 - current sequence number out
 - next sequence number in
- Header: 3-bit channel # and 1-bit sequence #
 - 4 bits in total
 - same as sliding window protocol
- Separates reliability from order
 - no errors and no packet loss, but not in order

Transport Layer 9

Berkeley Socket

Introduction

The following are some general information about Berkeley Sockets

1. Developed in the early 1980s at the University of California at Berkeley. There are no longer any major alternatives. Other major alternative was TLI (Transport Layer Interface). There are communications tools that are built on tool of Berkeley sockets. (E.g. RPC)
2. It is an API.
3. Its implementation is usually requires kernel code.
4. It is the defacto standard for communications programming.
5. There are higher level tools for programs that span more than one machine. RPC, DCOM and windows remoting are examples.
6. Used for point-to-point communications between computers through an inter-systems pipe. Namely can use the UNIX read, write, close, select, etc. system calls.
7. Supports broadcast. This is where the same message may be delivered to multiple systems on a network without additional overhead.
8. Available on every UNIX system that I know of and somewhat available in WIN32.
9. Build for client/server development. That is having one system provide a service to other systems.

Berkeley sockets support two types of communications. These sit on top of the TCP Internet datagrams.

TCP - connection oriented, stream, reliable.

UDP - connectionless, record oriented, unreliable.

Question: Why would anyone use a form of communication that is not reliable? Answer: speed. Answer: ability to broadcast.

Programming Aspects of Berkeley Sockets

Uses the TCP/IP protocol.

What are sockets? They represent end points for communications. In the UNIX world, they are file descriptors. Thus we can use system calls like read and write to receive and send data.

In order to communicate with a program on another computer, we have to identify the computer and specify the program. We specify the computer by giving its IP address. The program that we want to communicate with is identified by a port number. The port number is a positive integer that is advertised by the program that is waiting for a connection. More on this later.

The following are the major system calls supplied by Berkeley sockets and UNIX to perform TCP communications. There are other calls that we will use and calls to support UDP. These will be presented later.

1. socket - creates a socket for network communications. Returns a file descriptor for the socket.
2. connect - called by the client process to establish a connection between it and a server process. Need the server's address and port number. The file descriptor from the socket call is one of the arguments. Once the connection is made, this file descriptor may be used for communications.
3. write - to send data on the connection. If connection broken your program will receive a SIGPIPE signal or an error is returned. (The SIGPIPE signal does not occur immediately. What is the implication of this?)
4. read - to get data that was sent on the connection. It returns the number of bytes read. 0 is returned if the connection broken. (NOTE: WIN32

gives an error return.).

5. close - to de-allocate the socket.
6. bind - used by server process to associate an end-point address with a socket. Must include the port and server address with this call. This is like advertising the service. This socket is called the listening socket. Note: there are some port numbers that are referred to as well-known ports. See /etc/services for a list. A good habit is to always select port numbers that are greater than 7000.
7. listen - used by server process to indicate that it is ready to receive connections. The listening socket must be specified in this call. Another parameter is the queue length for those . Parameter values are not always guaranteed to be used. At one time, regardless of the value of this parameter SUN uses 5. Note: the select system call will consider a listening socket with a connection waiting to be ready to read.
8. accept - called by the server process to accept a connection. If no client is trying to connect the call will block.
9. select - used to determine if there is data available on a socket or if there is a client queued up for a connection to a server. Can also be used to determine if a write is possible.

Use of socket calls in a program.

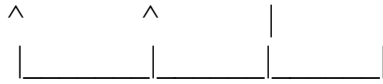
Client:

socket → connect → write → read → close



Server:

socket → bind → listen → accept → read → write → close



Design idea

Talk of concurrent servers vs. Multiplex servers.

Talk of state vs. stateless servers. Stateless safer.

System Calls for Clients

We will start out by learning how to build a client. This is less complex and has fewer issues than writing server code. Before looking at an example, I will give you a list of some relevant system calls and their definitions.

getservbyname

Many standard services such as ECHO have predefined port numbers. This function determines the port number for a service given its name.

```
#include <netdb.h>
```

```
struct servent *getservbyname( const char *name, const char *proto);
```

```
struct servent {  
  
    char   *s_name; /* official service name */  
    char   **s_aliases; /* alias list */  
    int     s_port; /* port # */  
    char   *s_proto; /* protocol to use */  
  
};
```

name - is the name of the service.

protocol - the protocol we are using. Choice between "TCP" or "UDP". Most services have both a TCP and UDP version.

Returns - NULL if fails and a pointer to a servent struct is successful. From this we can get the port number of the service.

NOTE: this function is not thread safe. Why? There is a thread safe version. How do you think they fixed the problem?

htons, htonl, ntohs, and ntohl

These are functions to convert to and from network byte ordering. n means network, h means host, s means short and l means long. Why do you think we need this?

gethostbyname

The gethostbyname function allows you to get the IP address of a computer given its full name. It has the following prototype:

```
#include <netdb.h>
```

```
struct hostent *gethostbyname( const char *name );
```

```
struct hostent {
```

```
    char *h_name; /* official name of host */
```

```
    char **h_aliases; /* alias list */
```

```
    int h_addrtype; /* host address type */
```

```
    int h_length; /* length of address */
```

```
    char **h_addr_list; /* list of addresses from name server */
```

```
#define h_addr h_addr_list[0] /* address, for backward compatibility */
```

```
};
```

name - the name of the host.

Returns NULL if fails. Otherwise, it returns the hostent structure. What we care most about in this structure is the address list. It contains the addresses for the host as a set of bytes in network order.

NOTE: this function is not thread safe. There is a thread safe version.

socket

This function returns a file descriptor that will eventually become the end of a pipe between client and server or a point at which we listen for connections.

Note: the location of the include files may differ on various UNIX systems.

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

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

domain - the family of addresses. The only one we care about is:

AF_INET ARPA Internet addresses

There is also PF_INET6. What does this mean.

PF_UNIX for UNIX internal protocols.

type - specifies the semantics of communication. The sys/socket.h file defines the socket types. The following types are the ones that we care about:

SOCK_STREAM

Provides sequenced, reliable, two-way byte streams with a transmission mechanism for out-of-band data. We use this for TCP/IP communications.

SOCK_DGRAM

Provides datagrams, which are connectionless messages of a fixed maximum length. We use this for UDP communications.

protocol - specifies the protocol. Since there is usually only one choice, we use 0 for the default.

connect

The connect call establishes a link between to the client that calls it and a server application. It has the following prototype.

```
int connect ( int socket, const struct sockaddr *address, size_t address_len );
```

socket - a socket that we created using the socket call.

address - a pointer to a address that contains the port and IP address of the host and our port and IP address. See [example](#) for details of how to fill in the address.

Returns -1 if error and 0 if successful.

Notes on example:

1. There is a service provided by most computers called the DAYTIME service. If you connect to a computer, it will return the date/time and disconnect.
2. This is an easy way to allow us to test client code without writing a server.
3. Not all computer will supply this service. In fact fewer supply it now than 5 years ago.

The code requires the address/name of server computer and the port number/service name as command-line arguments. User will be able to enter the IP address in decimal notation, the domain name (e.g. phobos.ramapo.edu) or if no entry the default address will be our computer. See /etc/hosts for local list of hosts. User will be able to enter the name of the service (see /etc/services), the port number, or if no entry, defaults to the DAYTIME service.

Lab6.

Time how long it takes to connect to an echo server and to send and receive 20,000 bytes in 100 byte packets. You should pick a far away site. We will discuss this in class.

Elementary TCP Sockets

UNIX Network Programming

Vol. 1, Second Ed. Stevens

Chapter 4

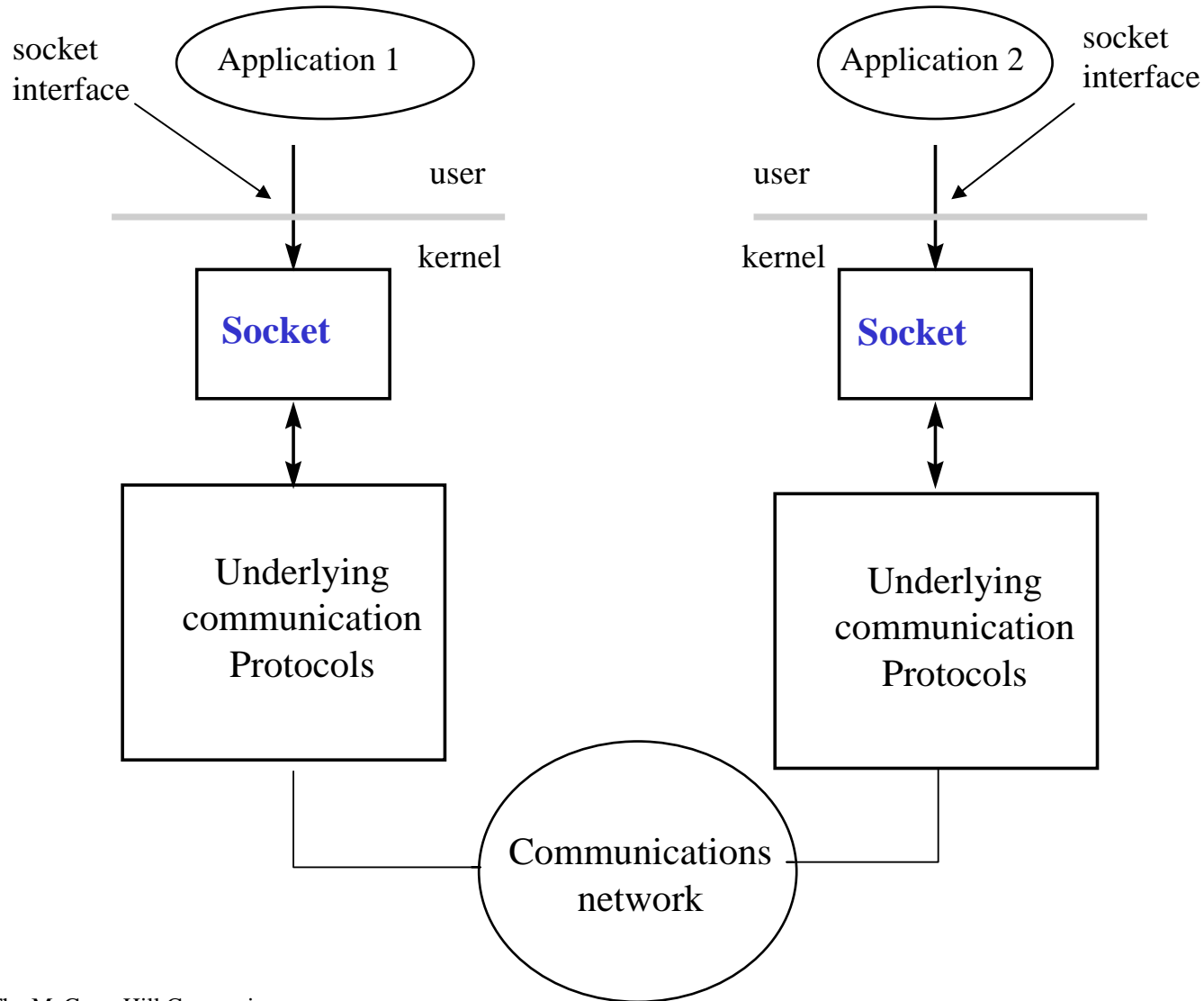
IPv4 Socket Address Structure

*The Internet socket address structure is named **sockaddr_in** and is defined by including `<netinet/in.h>` header.*

```
struct in_addr {
    in_addr_t  s_addr          /* 32-bit IP address */
};                               /* network byte ordered */

struct sockaddr_in {
    uint8_t    sin_len;        /* length of structure (16) */
    sa_family_t sin_family;    /* AF_INET */
    in_port_t  sin_port;       /* 16-bit TCP or UDP port number */
                                   /* network byte ordered */
    struct in_addr sin_addr;    /* 32-bit IPv4 address */
                                   /* network byte ordered */
    char        sin_zero[8];    /* unused */
};
```

The Socket Interface

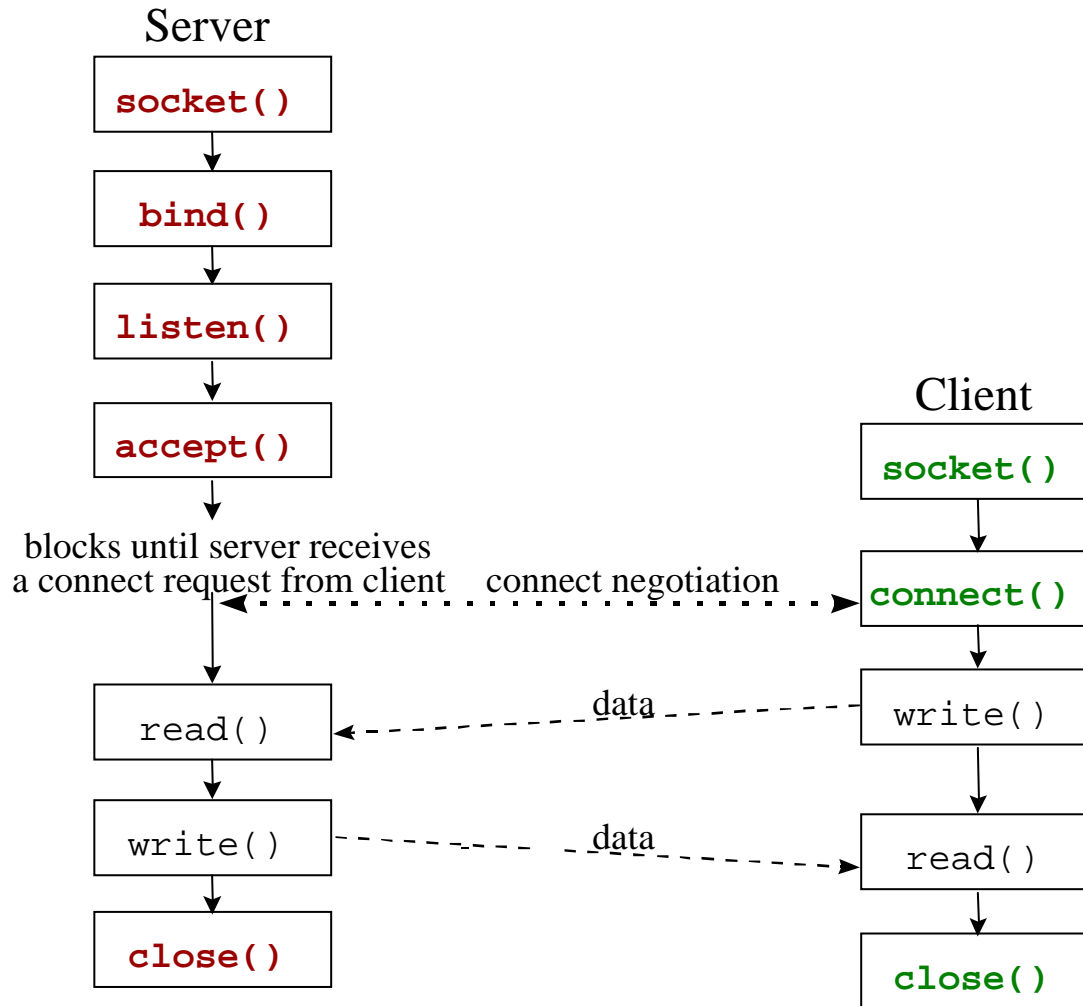


Copyright ©2000 The McGraw Hill Companies

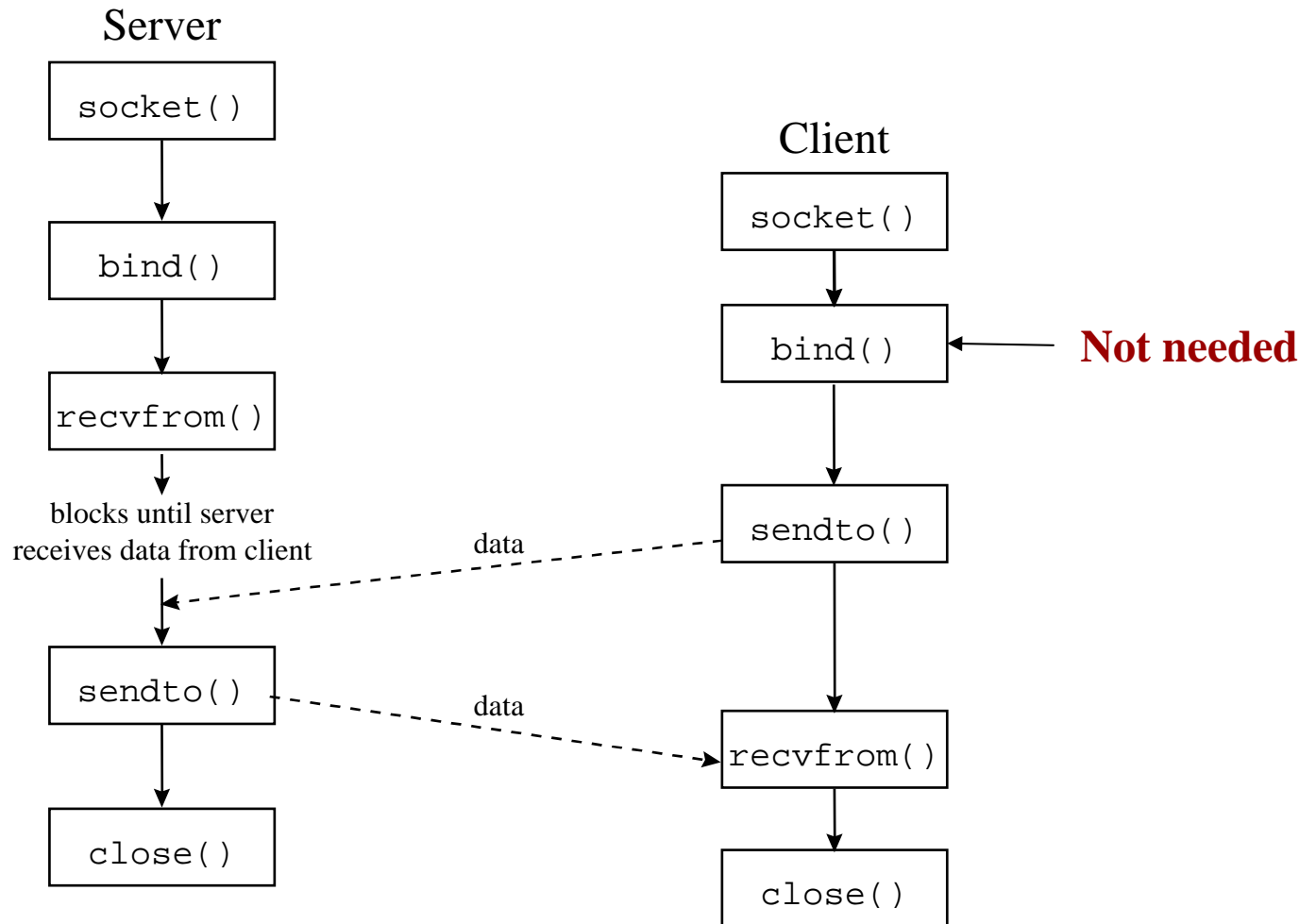
Leon-Garcia & Widjaja: *Communication Networks*

Figure 2.16

TCP Socket Calls



UDP Socket Calls



Copyright ©2000 The McGraw Hill Companies

Leon-Garcia & Widjaja: *Communication Networks*

Figure 2.18

System Calls for Elementary TCP Sockets

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

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

socket Function

```
int socket (int family, int type, int protocol);
```

family: specifies the protocol family {AF_INET for TCP/IP}

type: indicates communications semantics

SOCK_STREAM	stream socket	TCP
SOCK_DGRAM	datagram socket	UDP
SOCK_RAW	raw socket	

protocol: set to 0 except for raw sockets

returns **on success**: **socket descriptor** {a small nonnegative integer}

on error: -1

Example:

```
if ((sd = socket (AF_INET, SOCK_STREAM, 0)) < 0)  
    err_sys (“socket call error”);
```


connect Function

```
int connect (int sockfd, const struct sockaddr *servaddr,  
socklen_t addrlen);
```

sockfd: a socket descriptor returned by the socket function

**servaddr*: a pointer to a socket address structure

addrlen: the size of the socket address structure

The socket address structure must contain the *IP address* and the *port number* for the connection wanted.

In TCP **connect** initiates a three-way handshake. **connect** returns only when the connection is established or when an error occurs.

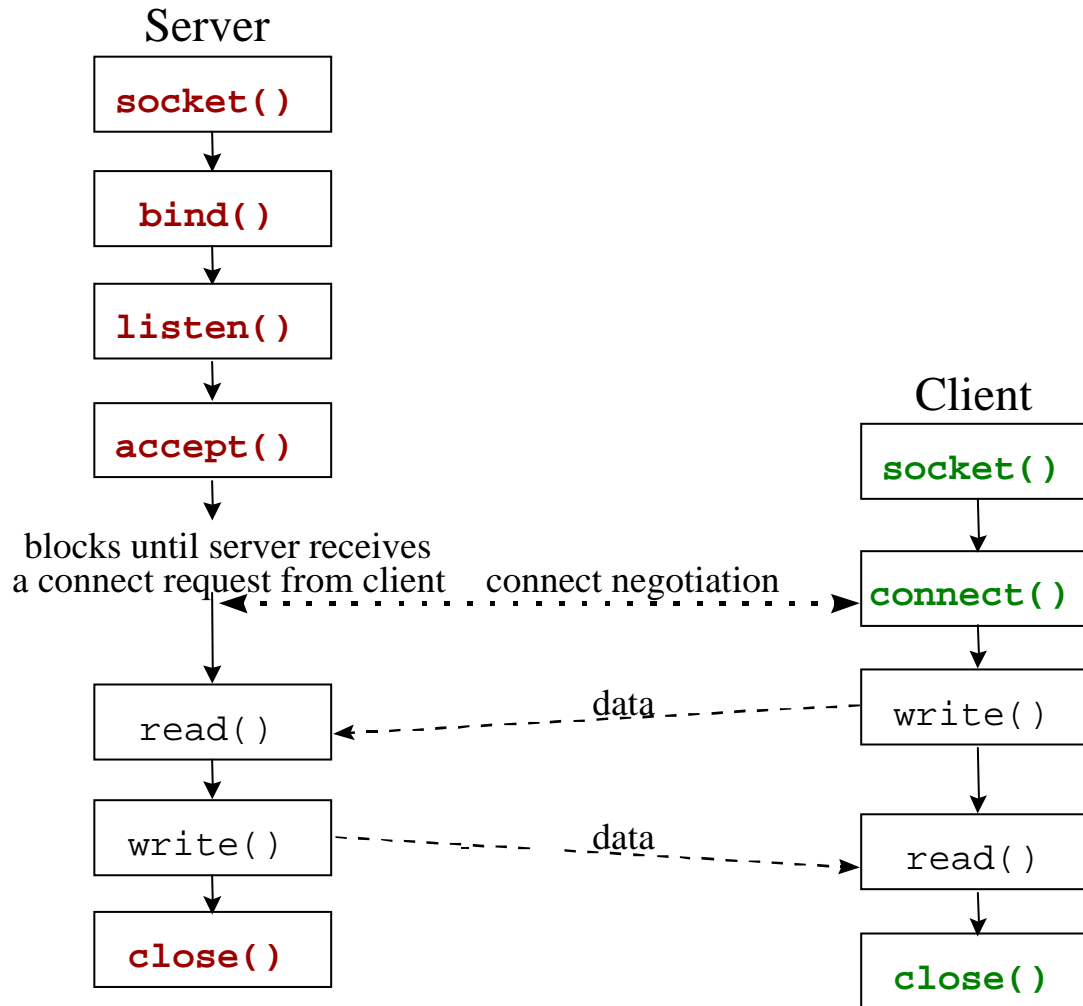
returns on success: 0

on error: -1

Example:

```
if ( connect (sd, (struct sockaddr *) &servaddr, sizeof (servaddr)) != 0)  
err_sys(“connect call error”);
```

TCP Socket Calls



bind Function

```
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

bind assigns a local protocol address to a socket.

protocol address: a 32 bit IPv4 address and a 16 bit TCP or UDP port number.

sockfd: a socket descriptor returned by the socket function.

**myaddr*: a pointer to a protocol-specific address.

addrlen: the size of the socket address structure.

Servers **bind** their “well-known port” when they start.

returns on success: 0

on error: -1

Example:

```
if (bind (sd, (struct sockaddr *) &servaddr, sizeof (servaddr)) != 0)  
    errsys (“bind call error”);
```

listen Function

```
int listen (int sockfd, int backlog);
```

listen is called **only** by a TCP server and performs two actions:

1. Converts an unconnected socket (*sockfd*) into a passive socket.
2. Specifies the maximum number of connections (*backlog*) that the kernel should queue for this socket.

listen is normally called before the **accept** function.

returns on success: 0
on error: -1

Example:

```
if (listen (sd, 2) != 0)  
    errsys (“listen call error”);
```

accept Function

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

accept is called by the TCP server to return the next completed connection from the front of the completed connection queue.

sockfd: This is the same socket descriptor as in **listen** call.

**cliaddr*: used to return the protocol address of the connected peer process (i.e., the client process).

**addrlen*: { this is a value-result argument }

before the accept call: We set the integer value pointed to by **addrlen* to the size of the socket address structure pointed to by **cliaddr*;

on return from the accept call: This integer value contains the actual number of bytes stored in the socket address structure.

returns on success: a new socket descriptor

on error: -1

accept Function (cont.)

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

For **accept** the first argument *sockfd* is the listening socket and the returned value is the connected socket.

The server will have one connected socket for each client connection accepted.

When the server is finished with a client, the connected socket must be closed.

Example:

```
sfd = accept (sd, NULL, NULL);
```

```
if (sfd == -1) err_sys (“accept error”);
```

close Function

```
int close (int sockfd);
```

close marks the socket as closed and returns to the process immediately.

sockfd: This socket descriptor is no longer useable.

Note – TCP will try to send any data already queued to the other end before the normal connection termination sequence.

Returns **on success:** 0

on error: -1

Example:

```
close (sd);
```

TCP Echo Server

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), and connect() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXPENDING 5   /* Maximum outstanding connection requests */
void DieWithError(char *errorMessage); /* Error handling function */
void HandleTCPClient(int clntSocket); /* TCP client handling function */
```

D&C


```

int main(int argc, char *argv[])
{
    int servSock;          /*Socket descriptor for server */
    int clntSock;         /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort;    /* Server port */
    unsigned int clntLen;           /* Length of client address data structure */

    if (argc != 2)    /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage: %s <Server Port>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]); /* First arg: local port */

    /* Create socket for incoming connections */
    if ((servSock = socket (PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

```

D&C

TCP Echo Server

```
/* Construct local address structure */  
memset(&echoServAddr, 0, sizeof(echoServAddr));      /* Zero out structure */  
echoServAddr.sin_family = AF_INET;                  /* Internet address family */  
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */  
echoServAddr.sin_port = htons(echoServPort);        /* Local port */  
  
/* Bind to the local address */  
if (bind (servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)  
    DieWithError("bind() failed");  
  
/* Mark the socket so it will listen for incoming connections */  
if (listen (servSock, MAXPENDING) < 0)  
    DieWithError("listen() failed");
```

D&C

TCP Echo Server

```
for (;;) /* Run forever */
{
    /* Set the size of the in-out parameter */
    clntLen = sizeof(echoClntAddr);    /* Wait for a client to connect */
    if ((clntSock = accept (servSock, (struct sockaddr *) &echoClntAddr, &clntLen)) < 0)
        DieWithError("accept() failed");

    /* clntSock is connected to a client! */
    printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));
    HandleTCPClient(clntSock);
}
/* NOT REACHED */
```

D&C

TCP Echo Client

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>    /* for memset() */
#include <unistd.h>    /* for close() */

#define RCVBUFSIZE 32 /* Size of receive buffer */

void DieWithError(char *errorMessage); /* Error handling function */
```

D&C

```
int main(int argc, char *argv[])
```

```
{
```

```
    int sock; /* Socket descriptor */  
    struct sockaddr_in echoServAddr; /* Echo server address */  
    unsigned short echoServPort; /* Echo server port */  
    char *servIP; /* Server IP address (dotted quad) */  
    char *echoString; /* String to send to echo server */  
    char echoBuffer[RCVBUFSIZE]; /* Buffer for echo string */  
    unsigned int echoStringLen; /* Length of string to echo */  
    int bytesRcvd, totalBytesRcvd; /* Bytes read in single recv()  
                                     and total bytes read */
```

```
    if ((argc < 3) || (argc > 4)) /* Test for correct number of arguments */  
    {  
        fprintf(stderr, "Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n",  
                argv[0]);  
        exit(1);  
    }
```

D&C

```

servIP = argv[1];          /* First arg: server IP address (dotted quad) */
echoString = argv[2];     /* Second arg: string to echo */

if (argc == 4)
    echoServPort = atoi(argv[3]); /* Use given port, if any */
else
    echoServPort = 7; /* 7 is the well-known port for the echo service */

/* Create a reliable, stream socket using TCP */
if ((sock = socket (PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

/* Construct the server address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr)); /* Zero out structure */
echoServAddr.sin_family = AF_INET; /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
echoServAddr.sin_port = htons(echoServPort); /* Server port */

```

D&C

TCP Echo Client

```
/* Establish the connection to the echo server */
if (connect (sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");

echoStringLen = strlen(echoString);      /* Determine input length */

/* Send the string to the server */
if (send (sock, echoString, echoStringLen, 0) != echoStringLen)
    DieWithError("send() sent a different number of bytes than expected");

/* Receive the same string back from the server */
totalBytesRcvd = 0;
printf("Received: ");                    /* Setup to print the echoed string */
```

D&C

TCP Echo Client

```
while (totalBytesRcvd < echoStringLen)
{
    /* Receive up to the buffer size (minus 1 to leave space for
       a null terminator) bytes from the sender */
    if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0)) <= 0)
        DieWithError("recv() failed or connection closed prematurely");
    totalBytesRcvd += bytesRcvd; /* Keep tally of total bytes */
    echoBuffer[bytesRcvd] = '\0'; /* Terminate the string! */
    printf("%s", echoBuffer); /* Print the echo buffer */
}
printf("\n"); /* Print a final linefeed */
close (sock);
exit(0);
}
```

D&C

Timekeeping

UNIX timekeeping is an untidy area, made more confusing by national and international laws and customs. Broadly, there are two kinds of functions: one group is concerned with getting and setting system times, and the other group is concerned with converting time representations between a bewildering number of formats.

Before we start, we'll define some terms:

- A *time zone* is a definition of the time at a particular location relative to the time at other locations. Most time zones are bound to political borders, and vary from one another in steps of one hour, although there are still a number of time zones that are offset from adjacent time zones by 30 minutes. Time zones tend to have three-letter abbreviations (TLAs) such as *PST* (*Pacific Standard Time*), *EDT* (*Eastern Daylight Time*), *BST* (*British Summer Time*), *AET* (*Australian Eastern Time*), *MEZ* (*Mitteleuropäische Zeit*). As the example shows, you should not rely on the combination *ST* to represent *Standard Time*.
- *UTC* is the international base time zone, and has the distinction of being one of those abbreviations which nobody can expand. It means *Universal Coordinated Time*, despite the initials. It obviously doesn't stand for the French *Temps Universel Coordonné* either. It corresponds very closely, but not exactly, to Greenwich Mean Time (GMT), the local time in England in the winter, and is the basis of all UNIX timestamps. The result is that for most of us, UTC is *not* the current local time, though it might be close enough to be confusing or far enough away to be annoying.
- From the standpoint of UNIX, you can consider the *Epoch* to be the beginning of recorded history: it's 00:00:00 UTC, 1 January 1970. All system internal dates are relative to the Epoch.
- Daylight Savings Time is a method of making the days appear longer in summer by setting the clocks forward, usually by one hour. Thus in summer, the sun appears to set one hour later than would otherwise be the case.

Even after clarifying these definitions, timekeeping remains a pain. We'll look at the main problems in the following sections:

Difficult to use

The time functions are not easy to use: to get them to do anything useful requires a lot of work. You'd think that UNIX would supply a primitive call upon which you could easily build, but unfortunately there isn't any such call, and the ones that are available do not operate in an intuitively obvious way. For example, there is no standard function for returning the current time in a useful format.

Implementations differ

There is no single system call that is supported across all platforms. Functions are implemented as system calls in some systems and as library functions in others. As a result, it doesn't make sense to maintain our distinction between system calls and library functions when it comes to timekeeping. In our discussion of the individual functions, we'll note which systems implement them as system calls and which as library calls.

Differing time formats

There are at least four different time formats:

- The system uses the *time_t* format, which represents the number of seconds since the Epoch. This format is not subject to time zones or daylight savings time, but it is accurate only to one second, which is not accurate enough for many applications.
- The *struct timeval* format is something like an extended *time_t* with a resolution of 1 microsecond:

```
#include <sys/time.h>

struct timeval
{
    long    tv_sec;           /* seconds since Jan. 1, 1970 */
    long    tv_usec;        /* and microseconds */
};
```

It is used for a number of newer functions, such as *gettimeofday* and *setitimer*.

- Many library routines represent the calendar time as a struct *tm*. It is usually defined in */usr/include/time.h*:

```
struct tm
{
    int    tm_sec;          /* seconds after the minute [0-60] */
    int    tm_min;         /* minutes after the hour [0-59] */
    int    tm_hour;        /* hours since midnight [0-23] */
    int    tm_mday;        /* day of the month [1-31] */
    int    tm_mon;         /* months since January [0-11] */
    int    tm_year;        /* years since 1900 */
    int    tm_wday;        /* days since Sunday [0-6] */
    int    tm_yday;        /* days since January 1 [0-365] */
    int    tm_isdst;       /* Daylight Savings Time flag */
};
```

```

long  tm_gmtoff;          /* offset from UTC in seconds */
char  *tm_zone;          /* timezone abbreviation */
};

```

Unlike `time_t`, a *struct tm* does not uniquely define the time: it may be a UTC time, or it may be local time, depending on the time zone information for the system.

- Dates as a text string are frequently represented in a strange manner, for example `Sat Sep 17 14:28:03 1994\n`. This format includes a `\n` character, which is seldom needed—often you will have to chop it off again.

Daylight Savings Time

The support for Daylight Savings Time was rudimentary in the Seventh Edition, and the solutions that have arisen since then are not completely compatible. In particular, System V handles Daylight Savings Time via environment variables, so one user's view of time could be different from the next. Recent versions of BSD handle this via a database that keeps track of local regulations.

National time formats

Printable representations of dates and times are very much a matter of local customs. For example, the date *9/4/94* (in the USA) would be written as *4/9/94* in Great Britain and *04.09.94* in Germany. The time written as *4:23 pm* in the USA would be written *16.23* in France. Things get even worse if you want to have the names of the days and months. As a result, many timekeeping functions refer to the *locale* kept by ANSI C. The locale describes country-specific information. Since it does not vary from one system to the next, we won't look at it in more detail—see *POSIX Programmer's Guide*, by Donald Lewine, for more information.

Global timekeeping variables

A number of global variables define various aspects of timekeeping:

- The variable *timezone*, which is used in System V and XENIX, specifies the number of minutes that the standard time zone is west of Greenwich. It is set from the environment variable `TZ`, which has a rather bizarre syntax. For example, in Germany daylight savings time starts on the last Sunday of March and ends on the last Sunday of September (not October as in some other countries, including the USA). To tell the system about this, you would use the `TZ` string

```
MEZ-1MSZ-2;M3.5,M9.5
```

This states that the standard time zone is called *MEZ*, and that it is one hour ahead of UTC, that the summer time zone is called *MSZ*, and that it is two hours ahead of UTC. Summer time begins on the (implied Sunday of the) fifth week in March and ends in the fifth week of September.

The punctuation varies: this example comes from System V.3, which requires a semicolon in the indicated position. Other systems allow a comma here, which works until you try to move the information to System V.3.

- The variable `altzone`, used in SVR4 and XENIX, specifies the number of minutes that the Daylight Savings Time zone is west of Greenwich.
- The variable `daylight`, used in SVR4 and XENIX, indicates that Daylight Savings Time is currently in effect.
- The variable `tzname`, used in BSD, SVR4 and XENIX, is a pointer to two strings, specifying the name of the standard time zone and the Daylight Savings Time zone respectively.

In the following sections we'll look at how to get the current time, how to set the current time, how to convert time values, and how to suspend process execution for a period of time.

Getting the current time

The system supplies the current time via the system calls `time` or `gettimeofday`—only one of these is a system call, but the system determines which one it is.

`time`

```
#include <sys/types.h>
#include <time.h>

time_t time (time_t *tloc);
```

`time` returns the current time in `time_t` form, both as a return value and at `tloc` if this is not NULL. `time` is implemented as a system call in System V and as a library function (which calls `gettimeofday`) in BSD. Since it returns a scalar value, a call to `time` can be used as a parameter to functions like `localtime` or `ctime`.

`ftime`

`ftime` is a variant of `time` that returns time information with a resolution of one millisecond. It originally came from 4.2BSD, but is now considered obsolete.

```
#include <sys/types.h>
#include <sys/timeb.h>

typedef long time_t;                /* (typically) */

struct timeb
{
    time_t    time;                /* the same time returned by time */
    unsigned short millitm;        /* Milliseconds */
    short    timezone;             /* System default time zone */
    short    dstflag;              /* set during daylight savings time */
};
```

```
};

struct timeb *ftime (struct timeb *tp);
```

The `timezone` returned is the system default, possibly not what you want. System V.4 deprecates* the use of this variable as a result. Depending on which parameters are actually used, there are a number of alternatives to `ftime`. In many cases, `time` supplies all you need. However, `time` is accurate only to one second. On some systems, you may be able to define `ftime` in terms of `gettimeofday`, which returns the time of the day with a 1 microsecond resolution—see the next section. On other systems, unfortunately, the system clock does not have a finer resolution than one second, and you are stuck with `time`.

gettimeofday

```
#include <sys/time.h>

struct timeval
{
    long    tv_sec;           /* seconds since Jan. 1, 1970 */
    long    tv_usec;        /* and microseconds */
};

int gettimeofday (struct timeval *tp,
                 struct timezone *tzp); /* (BSD) */
int gettimeofday (struct timeval *tp); /* (System V.4) */
```

`gettimeofday` returns the current system time, with a resolution of 1 microsecond, to `tp`. The name is misleading, since the `struct timeval` representation does not relate to the time of day. Many implementations ignore `tzp`, but others, such as SunOS 4, return time zone information there.

In BSD, `gettimeofday` is a system call. In some versions of System V.4 it is emulated as a library function defined in terms of `time`, which limits its resolution to 1 second. Other versions of System V appear to have implemented it as a system call, though this is not documented.

* The term *deprecate* is a religious term meaning “to seek to avert by prayer”. Nowadays used to indicate functionality that the implementors or maintainers wish would go away. This term seems to have come from Berkeley. To quote the “New Hackers Dictionary”:

:deprecated: adj. Said of a program or feature that is considered obsolescent and in the process of being phased out, usually in favor of a specified replacement. Deprecated features can, unfortunately, linger on for many years. This term appears with distressing frequency in standards documents when the committees writing the documents realize that large amounts of extant (and presumably happily working) code depend on the feature(s) that have passed out of favor. See also {dusty deck}.

Setting the current time

Setting the system time is similar to getting it, except that for security reasons only the superuser (*root*) is allowed to perform the function. It is normally executed by the *date* program.

adjtime

```
#include <sys/time.h>

int adjtime (struct timeval *delta, struct timeval *olddelta);
```

`adjtime` makes small adjustments to the system time, and is intended to help synchronize time in a network. The adjustment is made gradually—the system slows down or speeds up the passage of time by a fraction of a percent until it has made the correction, in order not to confuse programs like *cron* which are watching the time. As a result, if you call `adjtime` again, the previous adjustment might still not be complete; in this case, the remaining adjustment is returned in `olddelta`. `adjtime` was introduced in 4.3BSD and is also supported by System V. It is implemented as a system call in all systems.

settimeofday

```
#include <sys/time.h>

int gettimeofday (struct timeval *tp, struct timezone *tzp);
int settimeofday (struct timeval *tp, struct timezone *tzp);
```

`settimeofday` is a BSD system call that is emulated as a library function in System V.4. It sets the current system time to the value of `tp`. The value of `tzp` is no longer used. In System V, this call is implemented in terms of the `stime` system call, which sets the time only to the nearest second. If you really need to set the time more accurately in System V.4, you can use `adjtime`.

stime

```
#include <unistd.h>

int stime (const time_t *tp);
```

`stime` sets the system time and date. This is the original Seventh Edition function that is still available in System V. It is *not* supported in BSD—use `settimeofday` instead on BSD systems.

Converting time values

As advertised, there are a large number of time conversion functions, made all the more complicated because many are supported only on specific platforms. All are library functions. Many return pointers to static data areas that are overwritten by the next call. Solaris attempts to solve this problem with versions of the functions with the characters `_r` (for *reentrant*)

appended to their names. These functions use a user-supplied buffer to store the data they return.

strftime

```
#include <sys/types.h>
#include <time.h>
#include <string.h>

size_t strftime (char *s, size_t maxsize, char *format, struct tm *tm);
```

`strftime` converts the time at `tm` into a formatted string at `s`. `format` specifies the format of the resultant string, which can be no longer than `maxsize` characters. `format` is similar to the format strings used by `printf`, but contains strings related to dates. `strftime` has a rather strange return value: if the complete date string, including the terminating NUL character, fits into the space provided, it returns the length of the string—otherwise it returns 0, which implies that the date string has been truncated.

`strftime` is available on all platforms and is implemented as a library function. System V.4 considers `asctime` and `cftime` to be obsolete. The man pages state that `strftime` should be used instead.

strptime

```
#include <time.h>

char *strptime (char *buf, char *fmt, struct tm *tm);
```

`strptime` is a library function supplied with SunOS 4. It converts the date and time string `buf` into a `struct tm` value at `tm`. This call bears the same relationship to `scanf` that `strptime` bears to `printf`.

asctime

```
#include <sys/types.h>
#include <time.h>

int asctime (char *buf, char *fmt, tm *tm);
```

`asctime` converts the time at `tm` into a formatted string at `buf`. `format` specifies the format of the resultant string. This is effectively the same function as `strftime`, except that there is no provision to supply the maximum length of `buf`. `asctime` is available on all platforms and is implemented as a library function.

asctime and asctime_r

```
#include <sys/types.h>
#include <time.h>

char *asctime (const struct tm *tm);
```

```
char *asctime_r (const struct tm *tm, char *buf, int buflen);
```

`asctime` converts a time in `struct tm*` format into the same kind of string that is returned by `ctime`. `asctime` is available on all platforms and is always a library function.

`asctime_r` is a version of `asctime` that returns the string to the user-provided buffer `res`, which must be at least `buflen` characters long. It returns the address of `res`. It is supplied as a library function on Solaris systems.

cftime

```
#include <sys/types.h>
#include <time.h>
```

```
int cftime (char *buf, char *fmt, time_t *clock);
```

`cftime` converts the time at `clock` into a formatted string at `buf`. `format` specifies the format of the resultant string. This is effectively the same function as `strftime`, except that there is no provision to supply the maximum length of `buf`, and the time is supplied in `time_t` format. `cftime` is available on all platforms and is implemented as a library function.

ctime and ctime_r

```
#include <sys/types.h>
#include <time.h>
extern char *tzname[2];
```

```
char *ctime (const time_t *clock);
char *ctime_r (const time_t *clock, char *buf, int buflen);
```

`ctime` converts the time `clock` into a string in the form `Sat Sep 17 14:28:03 1994\n`, which has the advantage of consistency: it is not a normal representation anywhere in the world, and immediately brands any printed output with the word *UNIX*. It uses the environment variable `TZ` to determine the current time zone. You can rely on the string to be exactly 26 characters long, including the final `\0`, and to contain that irritating `\n` at the end. `ctime` is available on all platforms and is always a library function.

`ctime_r` is a version of `ctime` that returns its result in the buffer pointed to by `buf`. The length is limited to `buflen` bytes. `ctime_r` is available on Solaris platforms as a library function.

dysize

```
#include <time.h>
```

```
int dysize (int year);
```

`dysize` return the number of days in `year`. It is supplied as a library function in SunOS 4.

gmtime and gmtime_r

```
#include <time.h>

struct tm *gmtime (const time_t *clock);
struct tm *gmtime_r (const time_t *clock, struct tm *res);
```

`gmtime` converts a time in `time_t` format into `struct tm*` format, like `localtime`. As the name suggests, however, it does not account for local timezones—it returns a UTC time (this was formerly called Greenwich Mean Time, thus the name of the function). `gmtime` is available on all platforms and is always a library function.

`gmtime_r` is a version of `gmtime` that returns the string to the user-provided buffer `res`. It returns the address of `res`. It is supplied as a library function on Solaris systems.

localtime and localtime_r

```
#include <time.h>

struct tm *localtime (const time_t *clock);
struct tm *localtime_r (const time_t *clock, struct tm *res);
```

`localtime` converts a time in `time_t` format into `struct tm*` format. Like `ctime`, it uses the time zone information in `tzname` to convert to local time. `localtime` is available on all platforms and is always a library function.

`localtime_r` is a version of `localtime` that returns the string to the user-provided buffer `res`. It returns the address of `res`. It is supplied as a library function on Solaris systems.

mktime

```
#include <sys/types.h>
#include <time.h>
time_t mktime (struct tm *tm);
```

`mktime` converts a local time in `struct tm` format into a time in `time_t` format. It does not use `tzname` in the conversion—it uses the information at `tm->tm_zone` instead. In addition to converting the time, `mktime` also sets the members `wday` (day of week) and `yday` (day of year) of the *input* `struct tm` to agree with day, month and year. `tm->tm_isdst` determines whether Daylight Savings Time is applicable:

- if it is `> 0`, `mktime` assumes Daylight Savings Time is in effect.
- If it is `0`, it assumes that no Daylight Savings Time is in effect.
- If it is `< 0`, `mktime` tries to determine whether Daylight Savings Time is in effect or not. It is often wrong.

`mktime` is available on all platforms and is always a library function.

timegm

```
#include <time.h>

time_t timegm (struct tm *tm);
```

`timegm` converts a `struct tm` time, assumed to be UTC, to the corresponding `time_t` value. This is effectively the same thing as `mktime` with the time zone set to UTC, and is the converse of `gmtime`. `timegm` is a library function supplied with SunOS 4.

timelocal

```
#include <time.h>

time_t timelocal (struct tm *tm);
```

`timelocal` converts a `struct tm` time, assumed to be local time, to the corresponding `time_t` value. This is similar to `mktime`, but it uses the local time zone information instead of the information in `tm`. It is also the converse of `localtime`. `timelocal` is a library function supplied with SunOS 4.

difftime

```
#include <sys/types.h>
#include <time.h>

double difftime (time_t time1, time_t time0);
```

`difftime` returns the difference in seconds between two `time_t` values. This is effectively the same thing as `(int) time1 - (int) time0`. `difftime` is a library function available on all platforms.

timezone

```
#include <time.h>

char *timezone (int zone, int dst);
```

`timezone` returns the name of the timezone that is `zone` minutes west of Greenwich. If `dst` is non-0, the name of the Daylight Savings Time zone is returned instead. This call is obsolete—it was used at a time when time zone information was stored as the number of minutes west of Greenwich. Nowadays the information is stored with a time zone name, so there should be no need for this function.

tzset

```
#include <time.h>

void tzset ();
```

`tzset` sets the value of the internal variables used by `localtime` to the values specified in the environment variable `TZ`. It is called by `asctime`. In System V, it sets the value of the global variable `daylight`. `tzset` is a library function supplied with BSD and System V.4.

tzsetwall

```
#include <time.h>

void tzsetwall ();
```

`tzsetwall` sets the value of the internal variables used by `localtime` to the default values for the site. `tzsetwall` is a library function supplied with BSD and System V.4.

Suspending process execution

Occasionally you want to suspend process execution for a short period of time. For example, the `tail` program with the `-f` flag waits until a file has grown longer, so it needs to relinquish the processor for a second or two between checks on the file status.

Typically, this is done with `sleep`. However, some applications need to specify the length of time more accurately than `sleep` allows, so a couple of alternatives have arisen: `nap` suspends execution for a number of milliseconds, and `usleep` suspends it for a number of microseconds.

nap

`nap` is a XENIX variant of `sleep` with finer resolution:

```
#include <time.h>

long nap (long millisecs);
```

`nap` suspends process execution for at least `millisecs` milliseconds. In practice, the XENIX clock counts in intervals of 20 ms, so this is the maximum accuracy with which you can specify `millisecs`. You can simulate this function with `usleep` (see page 281 for more details).

setitimer

BSD systems and derivatives maintain three (possibly four) interval timers:

- A *real time* timer, `ITIMER_REAL`, which keeps track of real elapsed time.
- A *virtual* timer, `ITIMER_VIRTUAL`, which keeps track of process execution time, in other words the amount of CPU time that the process has used.
- A *profiler* timer, `ITIMER_PROF`, which keeps track of both process execution time and time spent in the kernel on behalf of the process. As the name suggests, it is used to implement profiling tools.

- A *real time profiler* timer, `ITIMER_REALPROF`, used for profiling Solaris 2.X multi-threaded processes.

These timers are manipulated with the system calls `getitimer` and `setitimer`:

```
#include <sys/time.h>

struct timeval
{
    long tv_sec;           /* seconds */
    long tv_usec;        /* and microseconds */
};

struct itimerval
{
    struct timeval it_interval; /* timer interval */
    struct timeval it_value;   /* current value */
};

int getitimer (int which, struct itimerval *value);
int setitimer (int which, struct itimerval *value, struct itimerval *ovalue);
```

`setitimer` sets the value of a specific timer `which` to `value`, and optionally returns the previous value in `ovalue` if this is not a `NULL` pointer. `getitimer` just returns the current value of the timer to `value`. The resolution is specified to an accuracy of 1 microsecond, but it is really limited to the accuracy of the system clock, which is more typically in the order of 10 milliseconds. In addition, as with all timing functions, there is no guarantee that the process will be able to run immediately when the timer expires.

In the `struct itimerval`, `it_value` is the current value of the timer, which is decremented depending on type as described above. When `it_value` is decremented to 0, two things happen: a signal is generated, and `it_value` is reloaded from `it_interval`. If the result is 0, no further action occurs; otherwise the system continues decrementing the counter. In this way, one call to `setitimer` can cause the system to generate continuous signals at a specified interval.

The signal that is generated depends on the timer. Here's an overview:

Table 16-1: `setitimer` signals

Timer	Signal
Real time	<code>SIGALRM</code>
Virtual	<code>SIGVTALRM</code>
Profiler	<code>SIGPROF</code>
Real-time profiler ¹	<code>SIGPROF</code>

¹ Only Solaris 2.x

The only timer you're likely to see is the real time timer. If you don't have it, you can fake it with `alarm`. In System V.4, `setitimer` is implemented as a library function that calls an undocumented system call. See *The Magic Garden explained: The Internals of UNIX System*

V Release 4, by Berny Goodheart and James Cox, for more details.

`setitimer` is used to implement the library routine `usleep`.

sleep

```
#include <unistd.h>

unsigned sleep (u_int seconds);
```

The library routine `sleep` suspends program execution for approximately `seconds` seconds. It is available on all UNIX platforms.

usleep

`usleep` is a variant of `sleep` that suspends program execution for a very short time:

```
#include <unistd.h>
void usleep (u_int microseconds);
```

`usleep` sleeps for at least `microseconds` microseconds. It is supplied on BSD and System V.4 systems as a library function that uses the `setitimer` system call.

select and poll

If your system doesn't supply any timing function with a resolution of less than one second, you might be able to fake it with the functions `select` or `poll`. `select` can wait for nothing if you ask it to, and since the timeout is specified as a `struct timeval` (see page 270), you can specify times down to microsecond accuracy. You can use `poll` in the same way, except that you specify its timeout value in milliseconds.

For example,

```
void usleep (int microseconds)
{
    struct timeval timeout;
    timeout.tv_usec = microseconds % 1000000;
    timeout.tv_sec = microseconds / 1000000;
    select (0, NULL, NULL, NULL, &timeout);
}

or

void usleep (int microseconds)
{
    poll (0, NULL, microseconds / 1000);
}
```