Murdoch
UNIVERSITY

# Topic 8:Network Programming, Client-Server Systems and the Web

ICT373: Software Architectures

# Recap

- Multithreading.
- Resources synchronization.
- Petri nets and UML can be used to describe thread interaction.

# Outline

- Network Programming Architectures
- Web Application Architectures
- (Very Brief) Database Connectivity and RMI

**Reading:**

- Textbook 11th Edition Chapter 28 (Textbook website) and Chapter 24.

Murdoch UNIVERSITY

# Objectives

- Explain what network programming is.
- Describe layered architecture and explain how network programming is an example.
- Explain the difference between client and server roles in establishing network connections.
- Explain how machines in a network are identified using an IP number and DNS. Explain the terms port and socket.
- Explain the consequences for overall program design of using TCP versus Datagrams.
- Describe the main architectural/design/implementation choices for developing a web-based client server system.
- Explain the problem that a firewall may present.

Murdoch UNIVERSITY

# Objectives

- Describe the structure of a Server program that may have many clients. How can threads be used by a Server to handle Clients?
  - What, if any, is the advantage of using threads.
- Explain what a URL class is used for.
- Explain the basic operation of a CGI based application.
- What is the difference between the GET and POST methods used with CGI.
- Compare Java servlets with CGI.
- Describe the role of JDBC and ODBC in accessing a relational database using SQL from a CGI script.
- Explain the use of RMI.

# Network Programming

- = writing programs which involve communication with other programs on other computers across networks.
- A ***network*** is a system of separate but interconnected computers.
  - There is not usually a big controlling computer in the middle.
- Networks range from privately owned Local Area Networks (LANs), through metropolitan and wide area networks to collections of interconnected networks called internetworks or internets.
  - These include the Internet.

# Network Programming

- Network programming used to be very complex.
  - The programmers needed to be aware of issues to do with programs and operating systems at both ends of the connection and sometimes even the hardware.
  - They usually needed to understand the various layers of the networking protocol.
  - There were a lot of different functions in each different networking library concerned with: connecting, packing, and unpacking blocks of information; sending those blocks back and forth; and handshaking.
- Today, particularly with Java, many of those details are hidden and the programmer can deal with abstractions.

# Network Programming

- The concept of networking is not so difficult.
  - you want to get some information from that machine over there and move it to this machine over here, or vice versa
  - it is similar to reading from and writing to files
  - in this case, the file exists on a remote machine; the remote machine can decide what it wants to do about the information you are requesting or sending.
- The programming model used in Java network programming is that of a file
  - you wrap the network connection (called a **socket**) with stream objects
  - use the same method calls as you do with all other streams
  - Java's built-in multithreading capabilities are used to handle multiple connections at once.

# Layered Architectures

- Java network programming is a good example of a layered architecture.
  - Other good examples are the Windows operating systems (Windows NT, Windows Server 2003/2008/2012, Windows XP, Windows 7), database systems and the Java decorator approach to IO.
- A layered design involves a **hierarchy of layers**.
  - The lowest layers correspond to concrete physical implementations.
  - The higher layers correspond to what facilities are available/visible to clients of the system or software.
  - Each layer is itself a valid way of looking at the system. However, if you need to know how the components, modules, objects, functions, facilities or whatever at a particular level are implemented then you need to look at the layer below.

# Layered Architectures

- Implementation of a particular layer should be totally in terms of the one layer just below. You should not see descriptions involving direct communication between layers which are not adjacent. This is a form of hiding of implementation details.
- This architecture supports incremental design, enhancement, reuse, and implementation choice.

# Java Network Programming Architecture

- Java programs which directly manage network connections; and server and browser programs.
- We can view the Java approach to network programming as a **layered architecture**.
  - At a high level we can just think of **whole objects** being passed from one program on one machine to another on another machine.
  - At a lower level the Java programmer has to manage an **output stream** from one program and an **input stream** in the other. We already know how to send Strings down such streams and also how to make sure that more complicated objects are serializable and so can be sent too.

# Java Network Programming Architecture (cont.)

- The lowest level which the Java programmer has to deal with is the level of **sockets** and **ports**.
  - These are themselves abstractions at quite a high level. However, the Java programmer has to choose which programs on which machines are to communicate (of course) and to specify the sort of communication channel required.
- Then the programmer quickly disguises the connection in terms of IO streams and settles back to work at the higher **byte-stream** or even object-passing level.

# Network Architecture

- Underneath the **byte-stream** level are many more levels which we will only look at briefly.
- Layers such as the **network layer**, the **transport layer** and the **application layer** are defined and given responsibilities. There are many other layers.
  - the **network layer** involves the delivery of small packets (called IP packets) of information step-by-step through unspecified routes across the internet. There is no guarantee of arrival or of the order of arrival.
  - at the **transport layer** a stream is broken up into small packets and, if so desired, checked and put together in the right order. The transport layer can also handle the fact that the consumer of the stream might not be keeping up with the producer. If asked to, the transport layer can alternatively deliver information much faster but less reliably.

# Network Architecture

- the **application layer** is concerned with network applications such as browsers, email, file sharing. In Java almost all network software is written for the application layer, which may call upon the services of lower layers.

- Below the network layer are layers to do with the physical transmission (eg, what voltage counts as a 1 and what counts as a 0) and data transmission (eg, how to specify that a package of information is coming through).
  - Each layer is supported by communication protocols and standards.

# DNS+IP

- In order to communicate between machines across the internet we need to identify the machines. Java uses the IP (= Internet Protocol) address which comes in two forms:
  - DNS (= Domain Name Service) form, eg, it.murdoch.edu.au;
  - dotted quad form, eg, 123.255.28.120
- The quad form can be handled as an **InetAddress** object: this class being defined in **java.net** package.
- Each time your machine is connected to the Internet it might have a different IP address (this is under the control of whoever controls your Internet access, eg your ISP or Murdoch network admin).
- To find your machine's current IP address you can use
- **InetAddress.getByName( NAME )**
- (in a Java program) where NAME is the name of your computer (which, under Windows, you can get from the Control Panel or System Info).

Murdoch UNIVERSITY

# Client and Server Roles

- Suppose that we just want two programs (on different machines) to be able to communicate. We know that in Java we will be able to use the connection as a stream.
- Perhaps there will be another stream back the other way.
- Neither program is “more important” than the other.
- However, at a network connection level, one machine has to act as the server and the other as the client. This just means that:
  - the client has to know the address of the server and try to initiate a connection;
  - the server has to listen for someone trying to connect to it.
  - Once the connection is made then both programs can just deal with IO streams, (as if they were reading and/or writing to a file).

# Client and Server Roles (cont.)

- Note that in testing and development situations it is useful to be able to run such client and server programs on the same machine without using a network connection. In that case you can use

  **InetAddress addr = InetAddress.getByName(null);**

  which will give you a special local loopback IP address

  (also known as "localhost" or "127.0.0.1"). Data sent to this IP address loops around and becomes IP input on the local host.

# Ports

- Before your client program can connect to your server program a little more needs to be agreed. The connection address has to be more specific because there may be several server programs running on the server machine.
- To solve this problem, the client and server have to agree on a ***port***. This is one of many numbered connection locations on each machine with an IP address.
  - They do not physically exist but at the client-server layer of abstraction, the software acts as if they do.
  - Port numbers can be between 0 and 65535.
  - Ports numbered 0 to 1023 are used for system services (e.g., HTTP and FTP and other system services), so do not use those.
  - If a port is already in use then the connection may fail.

# Ports

- Note that it is the port on the server that must be fixed on beforehand. The programmer(s) must arrange for that to be agreed somehow (eg, coding in a fixed port number).
- The server program can then listen to that port on the server machine.
- The client program can then try to establish a connection via that port of the server machine.

# Sockets

- = the software abstraction representing the ends (inside the programs) of a connection between two machines.
- For a given connection, there is a socket on each machine. You create a socket to make a connection to the other machine.
- Java provides ***stream sockets*** and ***datagram sockets***.
- **Stream sockets**
  - a process establishes connection to another process
  - data flows between the processes in continuous streams while the connection is in place
  - enable application to view networking as if it were file I/O
  - use the popular TCP (Transmission Control) protocol
- Java has two stream-based socket classes:
  - a **ServerSocket** that a server uses to 'listen' for incoming connections
  - a **Socket** that a client uses to initiate a connection

# Establishing a connection

- A **ServerSocket** object is created by the server program using a port number. The server program should then call the **accept()** method for that **ServerSocket** object.
- The server machine will then listen on the specified port for a connection request. The **accept** method is blocked until that happens.
- When a connection request arrives at that port then the **accept** method returns a new **Socket** object representing the server end of the connection.
- The client program just tries to make a new **Socket** directly from the server's address and agreed server port number. The construction will succeed if the connection is made.
- Both client and server can then easily turn their sockets into streams. The same socket can be turned into both an input and an output stream.

# Establishing a connection (cont.)

- The sockets represent a dedicated network connection. This puts a substantial load on the network and so care must be taken to release the network resources afterwards by explicitly disconnecting.
- **Steps in building a TCP connection:**
  - Get the server hostname or address
  - Get the service port number
  - Create a socket
  - Create input and output streams that use the socket

# Example: A simple server and client

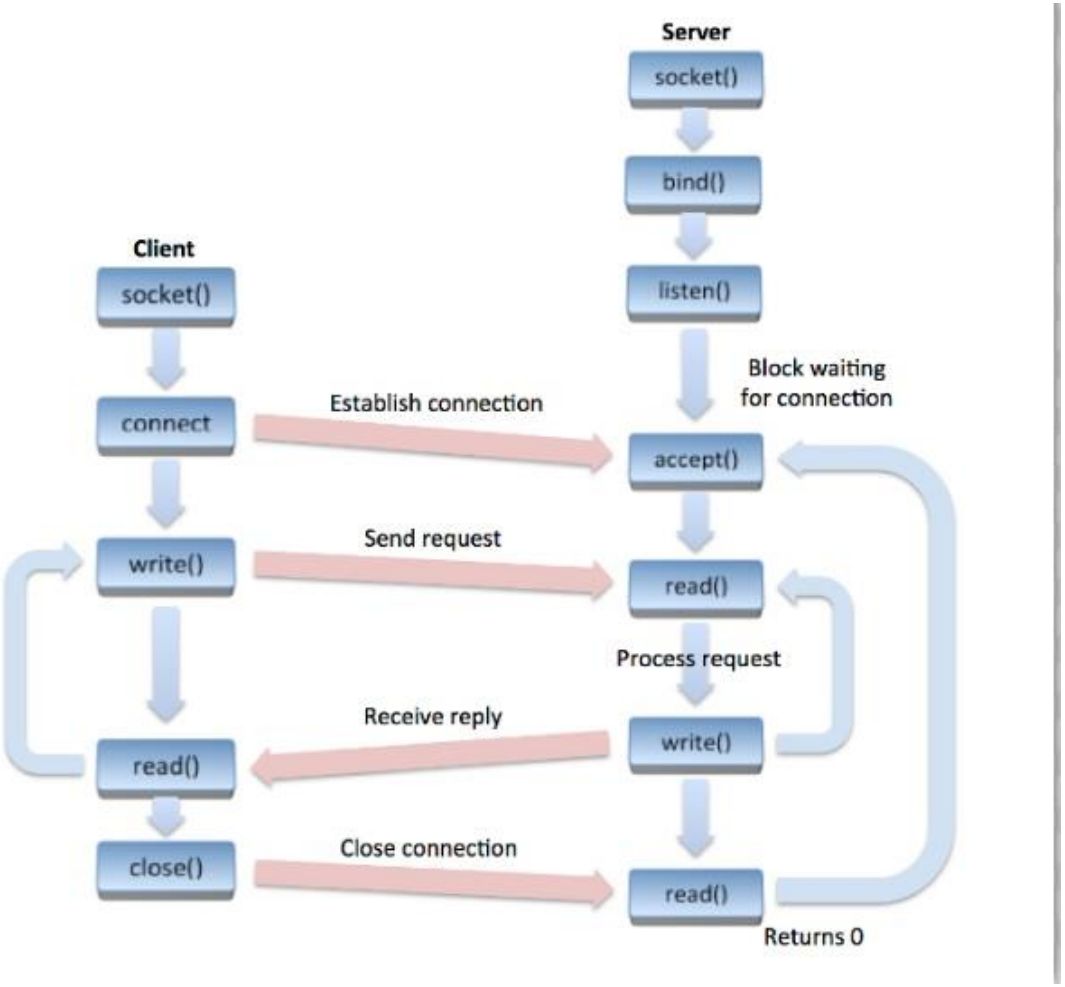
- Creating a Java server involves creating a **ServerSocket** Object.

  **ServerSocket s = new ServerSocket (PORT);**
- All the server does is wait for a connection from clients:

  **Socket socket = s.accept();**
- Then it uses the socket produced by this connection to create **InputStream** (get info from client) and **OutputStream** (send info to client) objects for communication: **socket.getInputStream(); socket.getOutputStream()**
- The server then reads from the **InputStream** and it writes to the **OutputStream** until it receives the line END.
- The client creates a Socket object, using the same port number, to connect to the server **Socket socket = new Socket (serverAddress, PORT);**
- Then the client creates an **InputStream** to hear what the server is saying, and an **OutputStream** to send info to the server.

Murdoch UNIVERSITY

# Example: A simple server and client




Murdoch UNIVERSITY

# Server: example

# JabberServer

- Note that the **ServerSocket** just needs a port number, not an IP address;
- When you call **accept()**, the method blocks until some client tries to connect to it;
- When a connection is made **accept()** returns with a **Socket** object representing that connection;
- The next part of the program looks just like opening files for reading and writing;
- Every time you write to **out**, its buffer must be flushed so that the information goes out over the network – this is done automatically in the above code via the second argument ('true') to the **PrintWriter** constructor. Alternatively use **out.flush()** after the **out.println(str)** statement in the infinite **while** loop.
- The infinite **while** loop reads lines from the **BufferedReader in** and writes information to **System.out** and to the **PrintWriter out**.
- When transmission is complete, connection is closed by calling the method **close()**.

Murdoch UNIVERSITY

# Client: example

# ClientSock

- In the **main** method you can see all three ways to produce the **InetAddress** of the local loopback IP address – using **null**, **localhost**, or the explicit reserved address **127.0.0.1**;
- To connect to a machine across a network you substitute that machine's IP address;
- Note that the **Socket** object called socket is created with both the **InetAddress** and the port number;
- Once the Socket object has been created, the process of turning it into a **BufferedReader** and **PrintWriter** is the same as in the server;
- Note that the buffer must again be flushed which happens automatically via the second argument (true) to the **PrintWriter** constructor;
- If the buffer is not flushed, the whole conversation will hang up because the initial "howdy" will never get sent;
- Each line that is sent back from the server is written to **System.out** to verify that everything is working correctly.

Murdoch UNIVERSITY

# Basic steps: Client

- Open a socket.
- Open an input stream and output stream to the socket.
- Read from and write to the stream according to the server's protocol.
- Close the streams.
- Close the socket.

# Multithreading

- The previous simple server can only deal with one client connection and then it finishes. It is easy to add a loop so that the server can deal with one client and then deal with another. However, for many useful services we need servers which can deal with several clients at once. So we need to use multithreading.
- See the MultiJabberServer example in Eckel.
- Use the same **ServerSocket** stuck in a loop of **accept**s. Each time it accepts it should hand over the resulting Socket to a constructor for a new thread object. This thread object constructor should use the **Socket** to open up I/O streams and then start. In its run method it can deal with the client and then finish.

# Multithreading(cont.)

- Many client-server applications can be built on this basic design. Just change the code for dealing with the I/O streams between the thread and the client.
- It is useful to use Eckel's code as the basis for applications because it involves careful construction of the exception handling. **Socket**s and **ServerSocket**s need to be closed if they are opened even if the program or thread subsequently gives up.
- Otherwise network connections get full.
- Examles: MultiJabberServer/ MultiJabberClient

# Datagrams

- So far we have used the **TCP (Transmission Control Protocol**) at the transport layer of our network communication. This is a reliable connection-based protocol that allows a byte stream to be delivered without error across an internet.
- For various design reasons you might want to use the alternative **UDP (User Datagram Protocol)** which is a connectionless service. Packets containing small messages or parts of messages are submitted which may not arrive (or may arrive in the wrong order) but they generally do arrive and they arrive much faster. No continuous connection is maintained.
- (UDP is often used to send audio and video over the internet).

# Datagrams

- Although this is a transport layer issue, the Java programmer has to convey the decision. Basically you create **DatagramSocket**s at both ends which wait for **DatagramPacket**s to show up and can be used to send the **DatagramPacket**s.
- There is no connection so you need to extract from the DatagramPacket, the sender's address if you want to send an answer.
- Streams are not used. You just put in and retrieve a String from the DatagramPacket.
- Examples

# Stream vs Datagram sockets

- Stream Socket:
  - Dedicated & point-to-point channel between server and client.
  - Use TCP protocol for data transmission.
  - Reliable and Lossless.
  - Data sent/received in the similar order.
  - Long time for recovering lost/mistaken data
- Datagram Socket:
  - No dedicated & point-to-point channel between server and client.
  - Use UDP for data transmission.
  - Not 100% reliable and may lose data.
  - Data sent/received order might not be the same
  - Don't care or rapid recovering lost/mistaken data

# Web Application Architectures

- Many useful programs follow the basic client-server architecture with a server managing a central repository or source of information and one or many clients accessing the information via the server via the Internet.
- Within that model there are several architectural choices.
- Regarding the client:
  - use a browser to run HTML/JavaScript (via HTTP);
  - use an application or a browser to run an applet which communicates via HTTP.
  - have an application program or applet opening a network connection (via TCP or UDP).
- Regarding the server:
  - use a server program;
  - use a CGI script; or
  - use a Java servlet.
  - (JSP, JSF, ASP, ISAP, NSAPI, etc which we don't discuss)

# Java Clients using HTTP

- A Java application can communicate with a server via HTTP as follows.
- It must send a GET or POST message and accept such a message back. This is made easy by classes in the **java.net** package.
  - You can encode strings for sending (eg, space to +) via **URLEncoder.encode(String);**
  - The **URL** class represents URLs.
  - **URL u = new URL (http://www.it.murdoch.edu.au/ + "cgi-bin/madeup?name=fred");**
  - will send a GET message (request) to the madeup program and make a new URL object which corresponds to the resulting connection.

Murdoch UNIVERSITY

# Java Clients using HTTP

- You can turn the reply into an input stream via, eg,

```
DataInputStream s = new
 DataInputStream(u.openStream);
```

- This reply may be just any old string but it is often an HTML page.
- Your application may have to do a little bit of work to extract information from that.

# Server Programs

- We know how to set one up in Java.
- Decide on TCP or UDP. UDP is often easier for smallish messages as it avoids the need for multithreading. If using multithreading and a local file as the data store then be careful to use locks to keep the file being accessed correctly.
- You should be able to leave the server program running constantly on the server machine.
- Unfortunately there is a serious limitation on the use of the architecture involving your own server programs.
- Unless you know the set up of LANs at both server and client ends or you know that there are no LANs involved then the security measures put in place where the LANs meet the outside Internet might stop your client-server communication. LANs often employ ***firewalls*** to guard this junction.

# Server Programs

- These machines (hardware/software) use more or less simple checks to prevent suspicious communication between the LAN and the outside world.  Port 80 is the accepted port for outside HTTP communication with a server. Using other ports is quite likely to make the firewall suspicious.
- However, this is the most general architecture if you can avoid the firewall problem.

# CGI Programming

- Recall that when the server gets a request to apply a method to a page in a directory called "cgi-bin" then it interprets the file name as an executable program (or script) and starts it up. Messages are fed in through standard input and out through standard output.
- It's the traditional and was until recently the most common way of providing a server program.
- The CGI-script, the program which is started up, is usually written in Perl or C (or C++). Java would take too long to start up.
- These languages are chosen because they can easily be used to do text processing, interact with the operating system and other programs and are more or less portable.
    - Perl, in particular, is popular as it is free, uses a Java-like bytecode interpreted approach and a simple and flexible syntax.

# GET and POST

- If the client sends a GET message then information is often appended to the URL after a question mark symbol. This is what an HTML form will do and what we can get a Java client to do easily.
- The server machine puts that information in an environment variable called QUERY_STRING before starting up the specified CGI script. So the script has to be able to get the information out of there. Eg, via

  getenv("QUERY_STRING") in a C/C++ program or

  $in= $ENV{'QUERY_STRING'} in PERL.
- There may be limits on how long a string can fit in QUERY_STRING (eg, 200 characters).

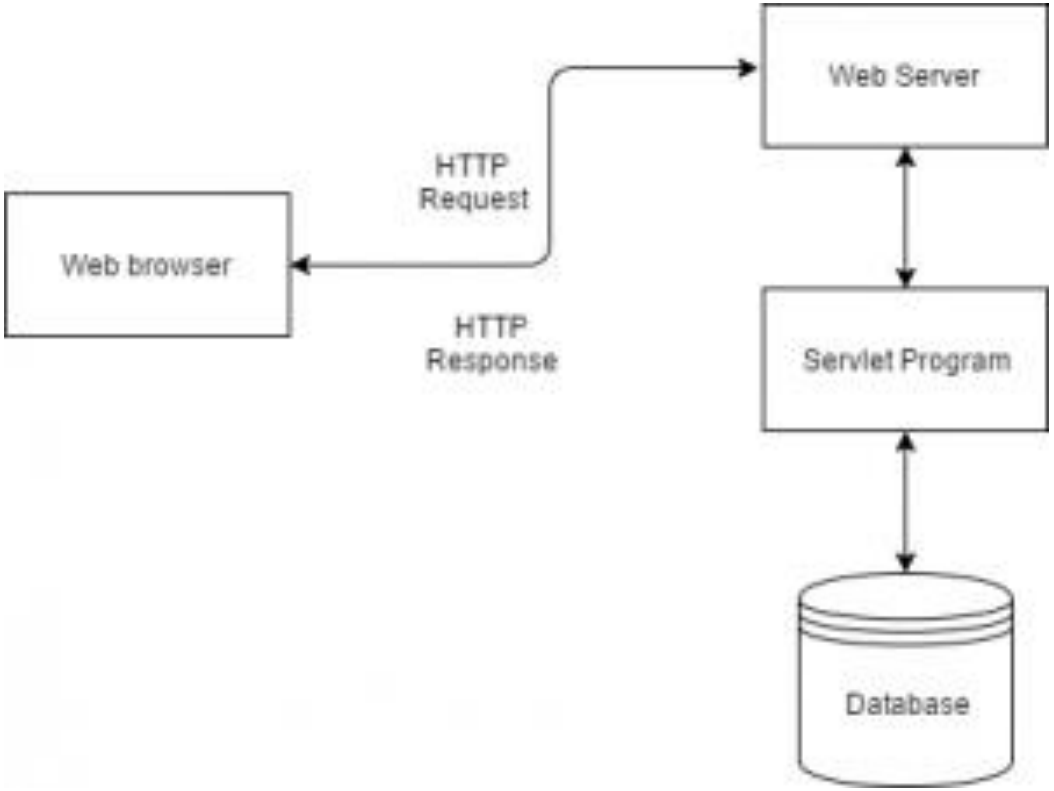
# GET and POST

- In both GET and POST, the server takes any standard output (eg, via printf("x") or print "x") and sends it back to the client. It is essential to start the output with the line "Content-type: text/plain\n\n" so the server knows how to (not) code the message.
- With a POST message the server feeds it in to standard input. The script can look up the environment variable CONTENT_LENGTH to see how long the message is so it can allocate storage and read the right number of bytes.
- GET means that form data is to be encoded (by a browser) into a URL while POST means that the form data is to appear within a message body.
    - "GET" is basically for just getting (retrieving) data whereas "POST" may involve anything, like storing or updating data, or ordering a product, or sending E-mail.

Murdoch UNIVERSITY

# Servlets

- Extend the capabilities of servers that host applications accessed by means of a request-response programming model.




Murdoch UNIVERSITY

# Servlets

- Servlets are Java's answers to CGI scripts. They extend the functionality of a Web server.
- Java packages **javax.servlet** and **javax.servlet.http** provide the classes and interfaces to define servlets.
- The basic idea is that a web server (which contains servlets) constantly runs a JVM which is on the look out for requests to talk to servlets. Servlet programs (written in Java) are kept in a special servlets directory. Each servlet is an extension of say **HTTPServlet** class and overrides the **init** and the **service** method.

# Servlets

- This message and all subsequent messages to fred result in its service method being called. The service method takes two arguments, a request and a response. All the information in the message can be got out of the request object in a few simple method calls. The response can be turned into an output stream.
- Servlets run much quicker than CGI, are easier to write and are platform independent. Other Java Web technologies include Java Server Pages (JSP) which is an extension of the servlet technology and enable web application programmers to create dynamic content.
- See Deitel for further details. Also see Java Server Faces (JSF).

# Databases and JDBC

- By far the most common and most useful client-server systems involve a database at the server end. Connecting the server (or any Java application) to a database is not too difficult.
- JDBC = Java Database Connectivity standard is based on the 1992 ODBC (Open Database Connectivity) standard for connecting to various databases.
- A Java program communicates with databases using the JDBC API **java.sql**.
- In the Java program you start a JDBC **Driver Manager** which manages the various database drivers that your program needs.
- You then open a Connection by specifying the protocol (eg, jdbc:odbc), the database, a user and a password. This gets turned into a Statement on which you execute a query (in SQL) and get back a ResultSet which is just like an iterator. This allows you to go through all the records in the answer to the query in turn.

# Databases and JDBC

- **Connection c = DriverManager.getConnection (dburl, user, password);**
- **Statement s = c.createStatement();**
- **ResultSet r = s.ExecuteQuery("SELECT * FROM authors");** // query example
- Making sure that the database is visible from your program involves some fiddly configuration.
- See Deitel for details.

Murdoch UNIVERSITY

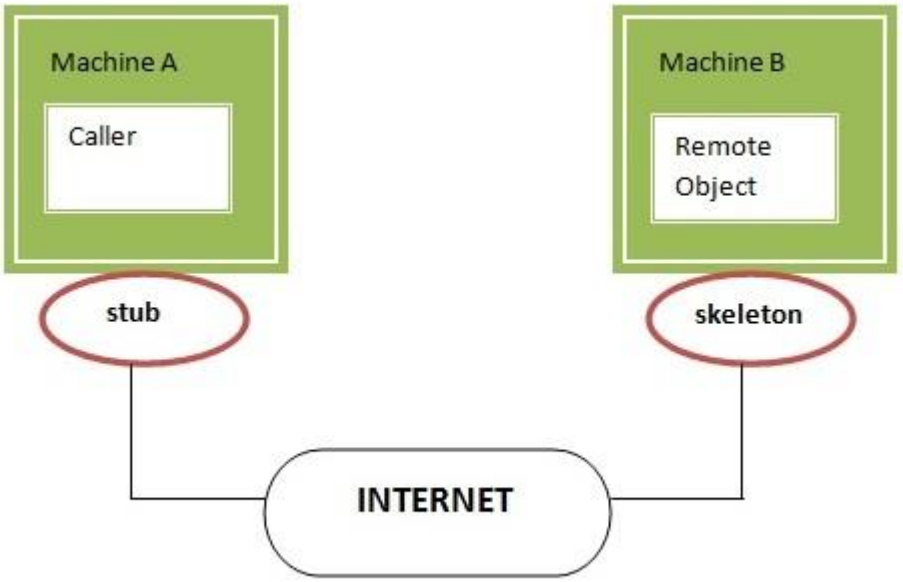
# JDBC Key Steps

- In general, to process any SQL statement with JDBC, you follow these steps:
  - Establishing a connection.
  - Create a statement.
  - Execute the query.
  - Process the ResultSet object.
  - Close the connection.

# Remote method invocation - RMI

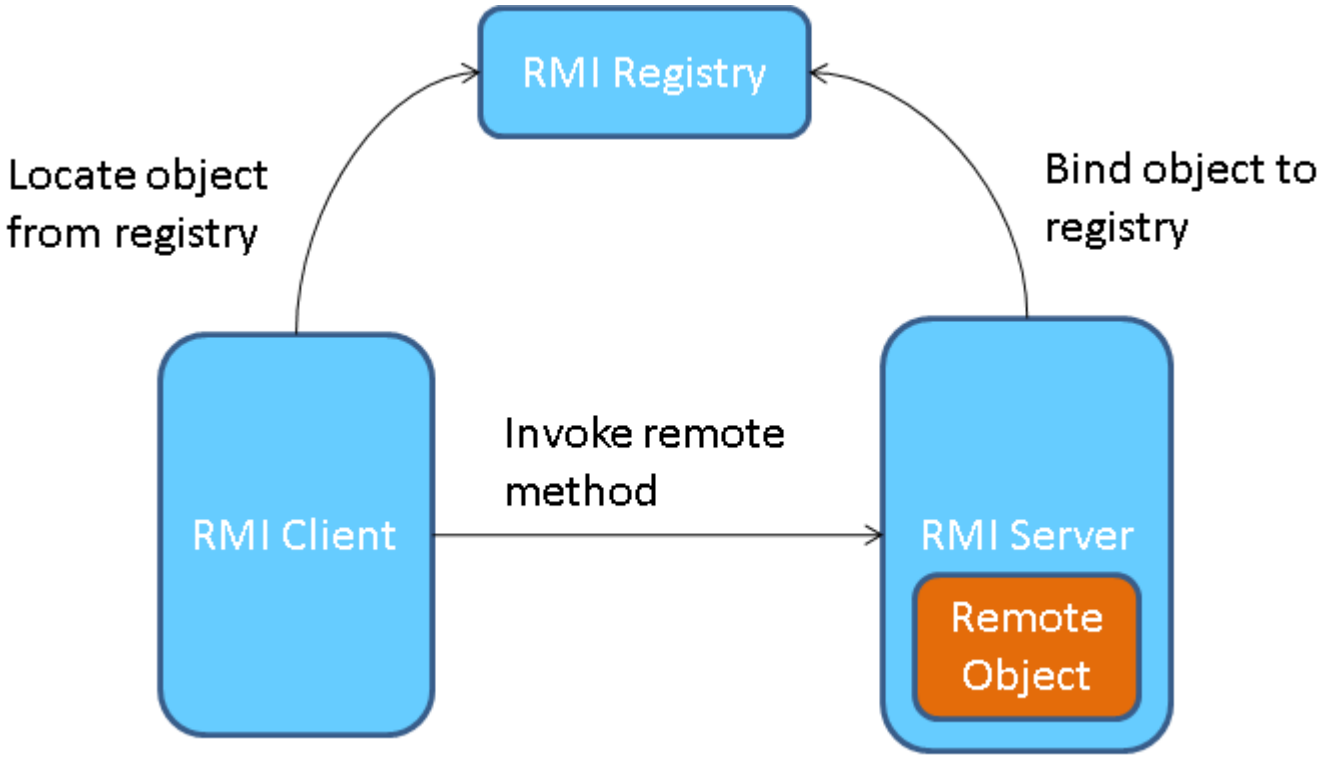
- Getting objects that reside on other machines to work for your programs.
- RMI system allows an object running in one JVM to invoke methods on an object running in another JVM. RMI provides for remote communication between programs written in the Java programming language.
- This is a high-level approach to distributed or client-server architecture. All the network connection detail is hidden. **Java.rmi** packages (SE 5 packages) provide classes and interfaces for RMI.
- Applications: If the service and the clients are Java programs then this is easy to design and implement using RMI.
- The service needs to be attached to some object which can run on the server machine. The client program is essentially just going to send a method to this object and get a return value (as if the object lived in the client program).

# RMI




Murdoch UNIVERSITY

# RMI




Example: RMIServer/RMIClient; server should be run first.

Murdoch UNIVERSITY

# RMI

- To implement it involves a few fiddly technical bits. The client and server have to agree on a remote **interface** (which extends **java.rmi.Remote** interface) for the object.
- A special RMI registry program should be started on the server.
  - A server program should be run which creates the object and registers it with the registry.
  - The object is kept alive by the registry even after the server program exits.
- The client uses one line of code to get a handle to the object from the distant registry.
- See Java API docs for details.

Murdoch
UNIVERSITY

# Summary

Subtitle if required

# Summary

- This topic looked at how parts of a system are distributed over a network and communicate through Sockets.
- It then looked again at Client-Server applications on the Web.
- Earlier we looked at Client-side processing. Here we looked at how the Client and Server communicate using CGI and at the execution of programs on the Server.

Murdoch

UNIVERSITY