

G52APR – Application Programming

# **Networking in Java**

Colin Higgins

G52APR

Application programming

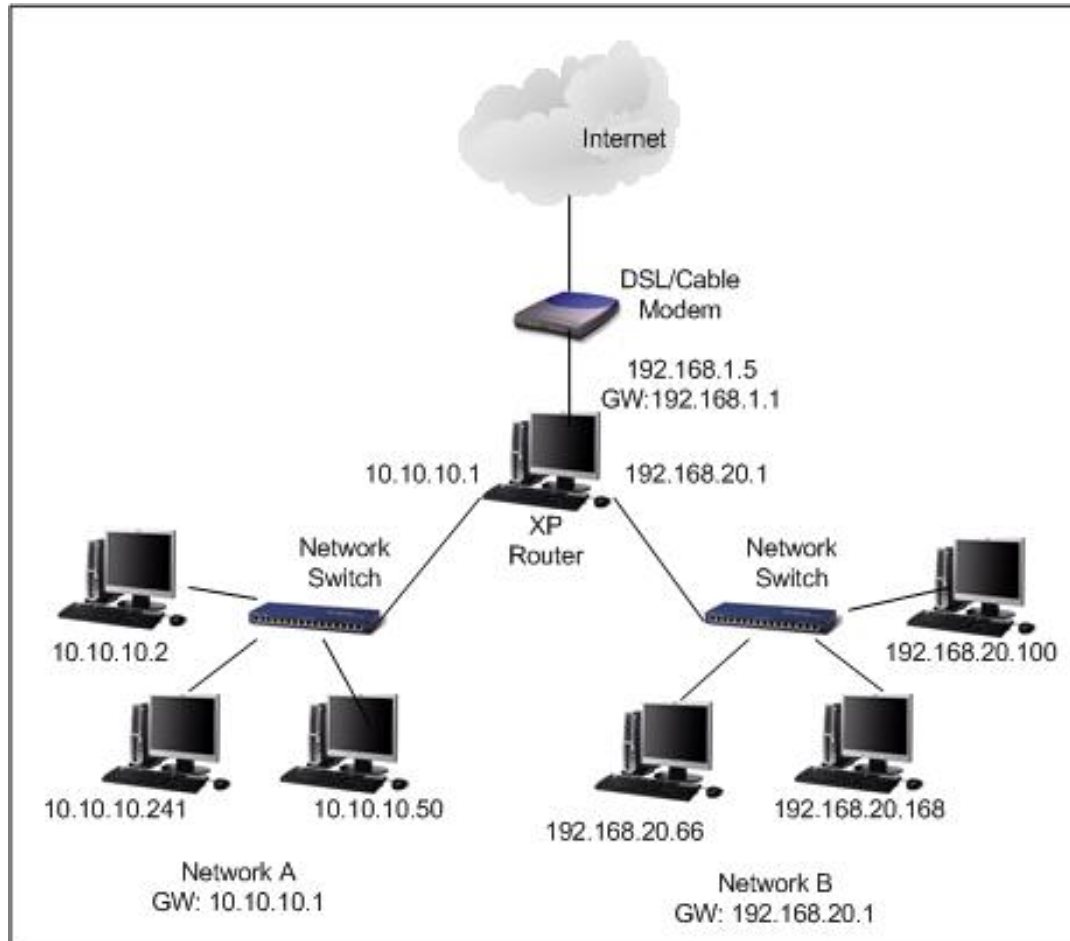
# **Networking Section 1**

Basics & Client Sockets

# Outline

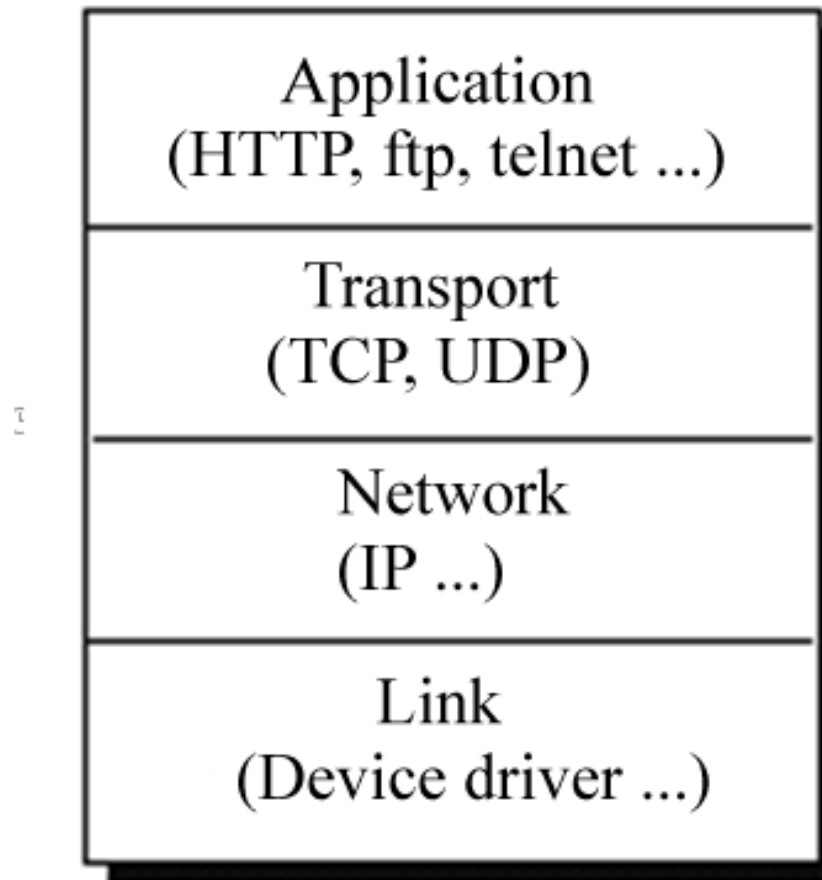
- Networking basics
  - Network architecture
  - IP address and port
  - Server-client model
  - TCP and UDP protocol
- Java Socket
- A simple client program to send email

# Internet hardware structure



- Network cards
- Hubs
- Routers
- Gateways

# Networking Diagram



# Networking in Java

- Java provides lots of networking support
- High-level interfaces
  - Fetch a file via HTTP
  - Execute a method on a remote object
- Low-level interfaces
- We'll start from the bottom and move up

# IP Address

- In Internet Protocol (IP), each machine is given an address
- In IPv4, each address is made up of 4 bytes, e.g. 128.243.80.167
- Certain numbers have special values
- IPv6 is coming online – which expands the address range

# DNS

- Domain Name Service (DNS): associating a textual name with an IP address
- [www.google.com](http://www.google.com)  $\leftrightarrow$  209.85.169.104
- [www.cs.nott.ac.uk](http://www.cs.nott.ac.uk)  $\leftrightarrow$  128.243.80.167
- A DNS server converts a DNS name to an IP address.



# Command Prompt

- Two very useful commands:

- `ipconfig/all`
  - `ping www.google.com`

- `Ipconfig` is an MS-DOS command, can be executed in command prompt.
- `Ping` is MS-DOS and Unix

# ipconfig command

```
Command Prompt

C:\Users\Michael>ipconfig/all

Windows IP Configuration

    Host Name . . . . . : Michael-PC
    Primary Dns Suffix . . . . . :
    Node Type . . . . . : Hybrid
    IP Routing Enabled. . . . . : No
    WINS Proxy Enabled. . . . . : No
    DNS Suffix Search List. . . . . : cable.virginmedia.net

Wireless LAN adapter Wireless Network Connection:

    Connection-specific DNS Suffix . : cable.virginmedia.net
    Description . . . . . : Atheros AR9285 Wireless Network Adapter
    Physical Address. . . . . : E8-39-DF-5A-04-2F
    DHCP Enabled. . . . . : Yes
    Autoconfiguration Enabled . . . . : Yes
    Link-local IPv6 Address . . . . . : fe80::fd1b:630c:372c:7378%12(Preferred)
    IPv4 Address. . . . . : 192.168.0.101(Preferred)
    Subnet Mask . . . . . : 255.255.255.0
    Lease Obtained. . . . . : 18 November 2011 13:25:01
    Lease Expires . . . . . : 26 November 2011 13:33:26
    Default Gateway . . . . . : 192.168.0.1
    DHCP Server . . . . . : 192.168.0.1
    DHCPv6 IAID . . . . . : 317209055
    DHCPv6 Client DUID. . . . . : 00-01-00-01-13-A7-F6-E9-00-24-54-3A-C5-95

    DNS Servers . . . . . : 192.168.0.1
    NetBIOS over Tcpip. . . . . : Enabled

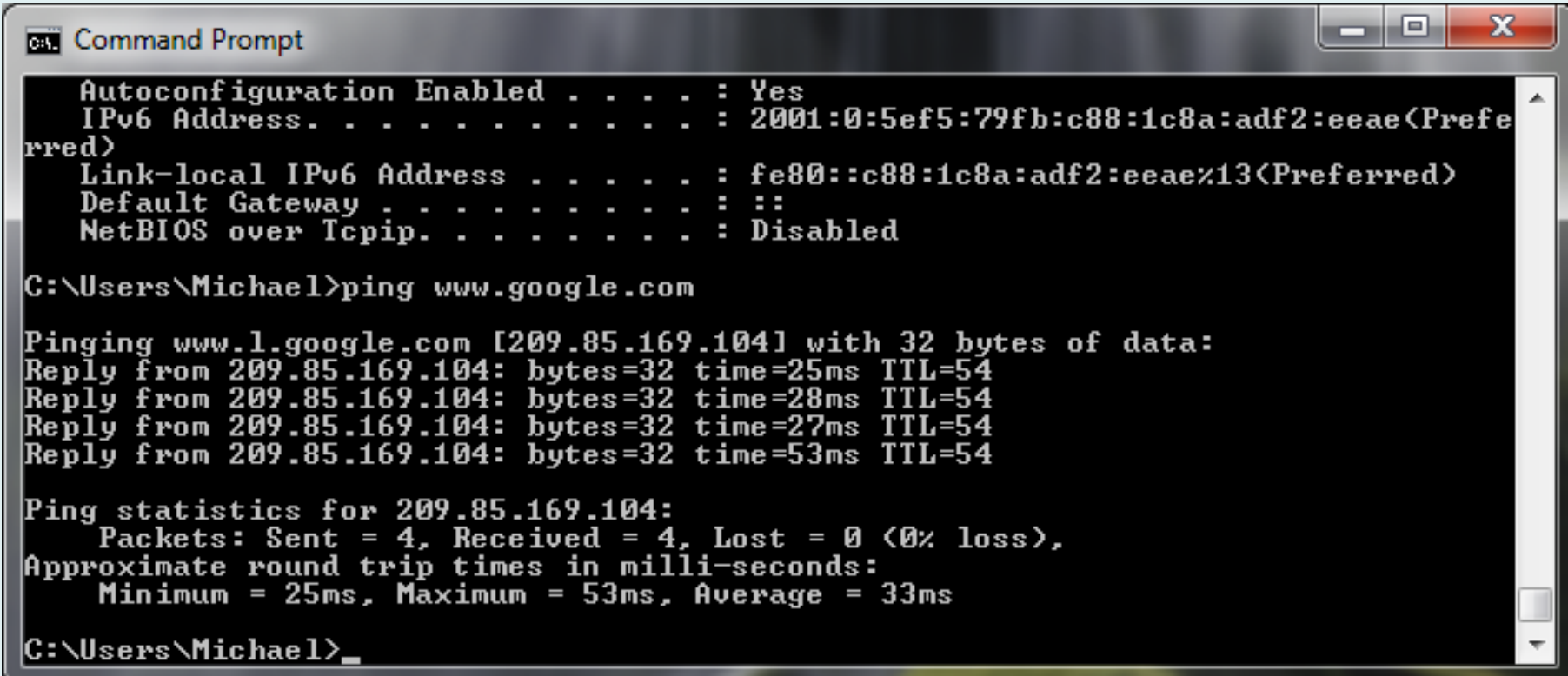
Ethernet adapter Local Area Connection:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix . :
    Description . . . . . : Marvell Yukon 88E8040 Family PCI-E Fast Ethernet Controller
    Physical Address. . . . . : 00-24-54-AE-DF-55
    DHCP Enabled. . . . . : Yes
    Autoconfiguration Enabled . . . . : Yes

Tunnel adapter isatap.{A01204E6-3498-4762-BE85-5AE5592765BC}:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix . :
    Description . . . . . : Microsoft ISATAP Adapter
    Physical Address. . . . . : 00-00-00-00-00-00-E0
    DHCP Enabled. . . . . : No
```

# ping command



```
C:\> Command Prompt

Autoconfiguration Enabled . . . . . : Yes
IPv6 Address. . . . . : 2001:0:5ef5:79fb:c88:1c8a:adf2:eeae(Prefe
rred)
Link-local IPv6 Address . . . . . : fe80::c88:1c8a:adf2:eeae%13(Preferred)
Default Gateway . . . . . : ::
NetBIOS over Tcpip. . . . . : Disabled

C:\Users\Michael>ping www.google.com

Pinging www.l.google.com [209.85.169.104] with 32 bytes of data:
Reply from 209.85.169.104: bytes=32 time=25ms TTL=54
Reply from 209.85.169.104: bytes=32 time=28ms TTL=54
Reply from 209.85.169.104: bytes=32 time=27ms TTL=54
Reply from 209.85.169.104: bytes=32 time=53ms TTL=54

Ping statistics for 209.85.169.104:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 25ms, Maximum = 53ms, Average = 33ms

C:\Users\Michael>_
```

- ping www.google.com
- ping -a 192.168.0.1
- ping -t SERVER
- ping

# Ports

- A port is a 16-bit number, 0-65535
- Some protocols work on specific ports
- Can't create a server on a port below 1024
- Unless you are root (or equivalent)
- Usually the system assign the client's port

# IP/Port examples

- HTTP servers listen on port 80
- SMTP is port 25, POP3 is 110 and IMAP4 143
- [www.bbc.co.uk](http://www.bbc.co.uk) has address 212.58.244.66
- So to connect to the BBC website you need to access port 80 on 212.58.244.66

# InetAddress class

- In java.net package
- Can handle both IPv4 and IPv6 addresses
- No constructor, uses static methods to make objects
- Either from the numeric value or via DNS

# InetAddress class

- `InetAddress.getByAddress(byte[] addr)`♪  
♪//Create an `InetAddress` representing the // address in `addr`♪
- `InetAddress.getName(String host)`♪  
♪//Determines the IP address of a host, given //the host's name♪
- `InetAddress.getAllByName(String host)`♪  
♪//Return an array of `InetAddresses` valid for //host♪

# InetAddress class

- `byte[] getAddress()`♪  
♪//returns the raw IP address♪
- `String getHostName()`♪  
♪//returns the host name for this IP address♪
- `String getCanonicalHostName()`♪  
♪//returns the FQDN for this IP address ♫
- `String.getHostAddress()`♪  
♪//returns a textual presentation of the IP♪

FQDN – fully qualified domain name  
Go try it.



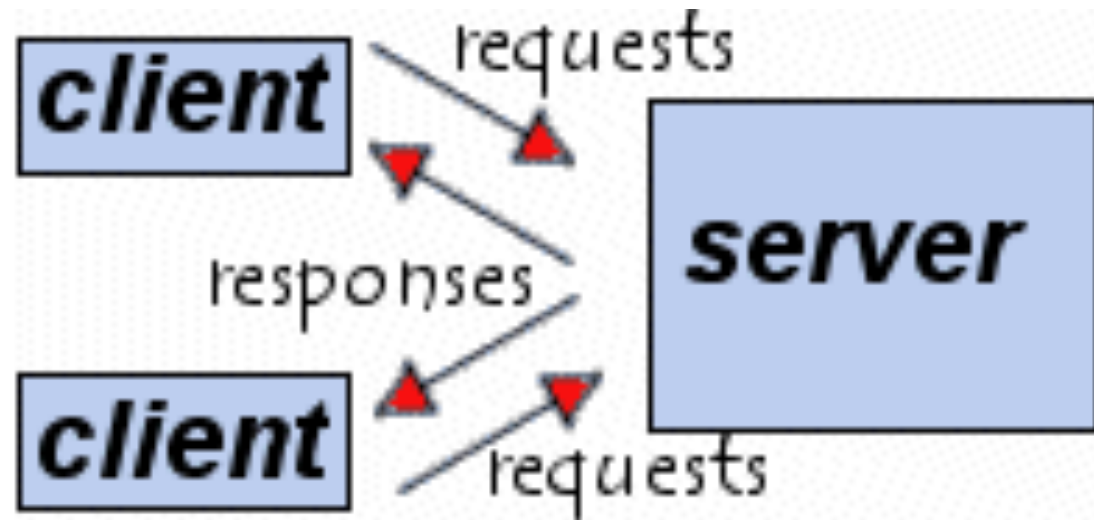
# Internet Protocol

- Internet is built on top of IP – the Internet protocol
- Designed to allow internetworking between different network types
- Mapped on top of other network technologies, e.g. Ethernet, ATM etc.

# IP Networking

- Communication is generally point to point
- One machine (the *client*) connects to another machine (the *server*)
- Data is broken up into packets and sent between the two machines
- Packets can get lost, duplicated or delivered out of order

# Server-Client model



A Server responses requests from the client(s).

# TCP

- TCP
  - Transmission Control Protocol
  - Stateful
  - Accurate delivery
  - But can incur delays
  - Effectively becomes stream based

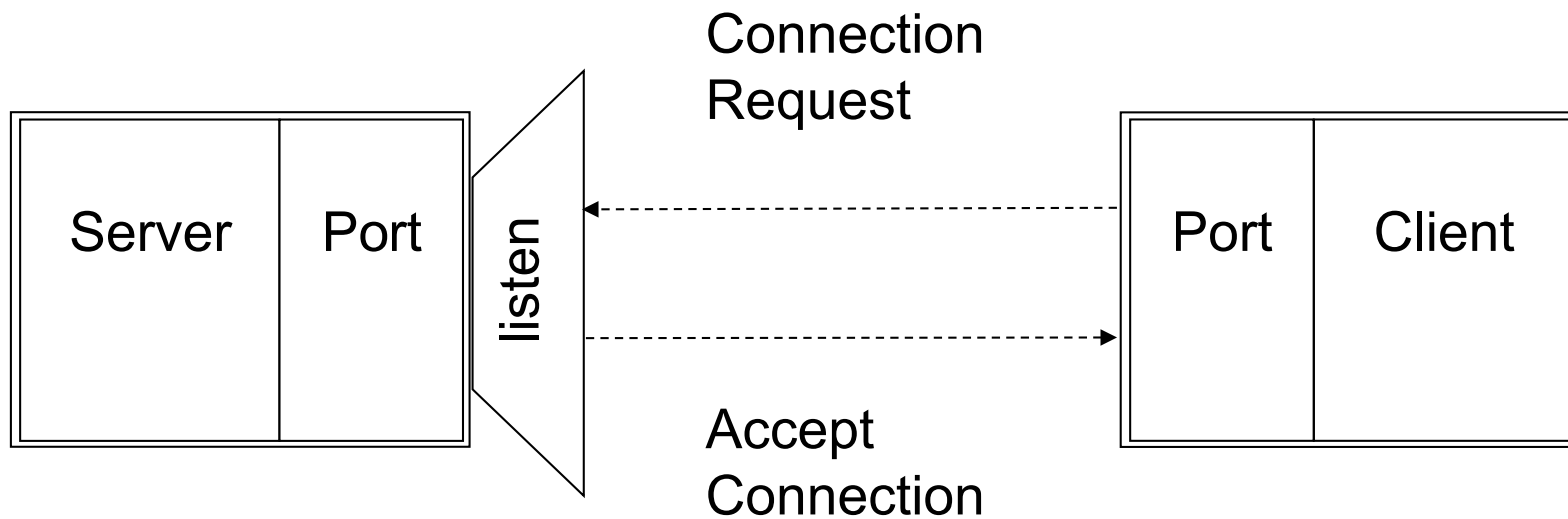
# UDP

- UDP
  - User datagram protocol
  - Stateless
  - Packet (or datagram) based
  - Not guarantee on delivery
  - Good for real-time applications (e.g. streaming media)

# Socket

- A **socket** is one end-point of a two-way communication link between two programs running on the network.
- Socket classes are used to represent the connection between a client program and a server program.
- The java.net package provides two classes--**Socket** and **ServerSocket**--that implement the client side of the connection and the server side of the connection, respectively.

# TCP Server-Client model



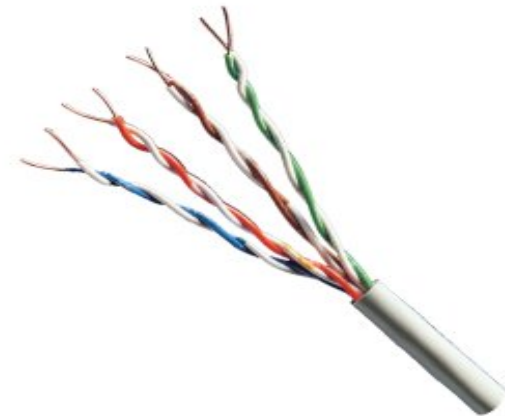
# TCP Server-Client paradigm

- One machine (the server) sets up a socket to listen for incoming connections
- Another machine (the client) attempts to connect one of its sockets to the server's socket
- The server accepts this connection
- And data can be sent and received



# Socket connections

- All sockets have an address and a port number
- To connect to a socket you need to know its address and port
- Data transfer is duplex (both machines can send data at the same time)



# Sockets in Java

- IP connection made using sockets
- Java encapsulates these in objects
- Create a `Socket` object
- `connect()` to a specific IP and port
- Transfer data
- Then `close()` when finished

# Socket methods

- `Socket()`  
♪//New socket♪
- `Socket(InetAddress addr, int port)` throws `IOException`  
♪//Create a new socket and connect to specified address and port♪
- `void close()` throws `IOException`  
♪//Closes this socket♪
- `void connect(SocketAddress endpoint)` throws `IOException`  
♪//Connect socket to `endpoint`, `SocketAddress` is a convenience class♪
- `void close()` throws `IOException`  
♪//Closes this connection♪

Also convenience methods that allow you create directly from a String  
Go check the Java document for detail.

# Socket methods example

```
Socket clientSocket = new Socket();  
clientSocket.connect ("192.168.0.101",1344);
```

```
Socket clientSocket = new Socket("192.168.0.101",1344);
```

```
Socket clientSocket = new Socket("localhost",1566);
```

# Transferring Data

- You already know how to do this ...
- Uses the standard Java I/O classes,  
♪InputStream and OutputStream♪
- Methods provided in Socket object to access the I/O objects
- Can then wrap in BufferedReader and PrintWriters etc.♪

# More Socket methods

- `InputStream getInputStream()` throws `IOException`♪  
♪//Gets an `InputStream` for this `Socket`♪
- `OutputStream getOutputStream()` throws `IOException` ♪  
♪//Gets an `OutputStream` for this `Socket`
- Closing the streams will close the socket  
♪

# A TCP client example

- As a simple example of socket programming we can implement a program that sends email to a remote site
  - This is taken from *Core Java*, vol 2, chapter 3
- Email servers use port number 25 which is the SMTP port
  - Simple mail transport protocol

# An email program

- A client socket to the mail server can be opened on port 25

Socket s=new Socket("engmail.bham.ac.uk",25);

- SMTP uses the following protocol

HELO *sending host*

MAIL FROM: *sender email address*

RCPT TO: *recipient email address*

DATA

*mail message*

...

QUIT



# An email program

```
import java.io.*;
import java.net.*;
public class EmailSender
{
    public static void main() throws Exception{
        Socket s = new Socket("128.243.15.141",25);
        PrintWriter out = new PrintWriter(s.getOutputStream());
        String hostname= InetAddress.getLocalHost().getHostName();
        out.println("HELO " + hostname);
        out.println("MAIL FROM: " + "jiawei.li@cs.nott.ac.uk");
        out.println("RCPT TO: " + "jiawei.li@cs.nott.ac.uk");
        out.println("DATA");
        out.println("This is a test.");
        out.println("QUIT");
    }
}
```

G52APR - Application programming

# **Networking Section 2**

Advanced & Server Sockets

# Outline

- TCP server-client example
- Multiple threading server
- UDP
- UDP example
- HTTP connections
- Remote Method Invocation (RMI)
- Simple Object Access Protocol (SOAP)

# TCP Client Side

- Writing a Java network client isn't that hard
- Create a Socket
- `connect()` it to the server
- And transfer data
- Creating a server isn't that simple

# Where's my server?

- Need to know where our server is (IP address and port)
- Could call `InetAddress.getLocalHost()`
- Returns an `InetAddress` for the local host
- But machines can have more than one IP
- So ask the socket what address it's bound to

# Java Servers

- Need to create a socket that is listening for connections
- Java encapsulates this in the `ServerSocket` class
- Can either give this the port number on what we want the server listen
- Or use 0 to get any free port

# ServerSocket **methods**

- ServerSocket() throws IOException  
new ServerSocket not bound to a port,  
so we use:
- ServerSocket(int port) throws IOException  
Create a new ServerSocket and listen on  
the specified port.  
Throws an exception if unable to create  
the socket

# Socket methods

- `int getLocalPort()`  
Return the port we are listening on
- `InetAddress getAddress()`  
Returns the `InetAddress` the socket is listening on
- `void close()` throws `IOException`  
Closes the socket and stop listening for connections



# Listening, but not Accepting

- The above will create the socket and set it to listen for connections
- But it won't accept any incoming connections
- We have to tell the ServerSocket to `accept()` incoming connections

# ServerSocket methods

- Socket accept() throws IOException  
Listen for a connection to be made to this socket and accept it.
- Will block until connection is made (an Exception does not affect server socket)
- Returns a **Socket** for the active connection
- The ServerSocket is still available to listen for more connections

# A server example

```
import java.io.*;
import java.net.*;
import java.util.*;
public static void main() throws IOException{
    Socket s = null;
    ServerSocket ss = new ServerSocket(1344);
    while (true) {
        s = ss.accept();
        BufferedWriter out = new BufferedWriter(
            new OutputStreamWriter(s.getOutputStream()));
        out.write("Java Daytime server:" +
            (new DatetoString() + "\n");    // Date is deprecated, use Calendar!
        out.close();
        s.close();
    }
}
```

# A client example

```
import java.io.*;
import java.net.*;
public static void main() throws IOException{
    try {
        Socket clientSocket = new Socket("localhost",1344);
        // Socket clientSocket = new Socket();
        // clientSocket.connect ("localhost",1344);
        BufferedReader in = new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()));

        while (!in.ready()) {}
        System.out.println(in.readLine()); // Read one line and output it
        System.out.print("\n");
    } catch(Exception e) {
        System.out.print("Whoops! It didn't work!\n");
    }
}
```

# Test

- Run server program
- Run client program
- Output:  
Java Daytime server:Fri Oct 14 10:58:44 BST 2011
- Run client program
- Output:  
Java Daytime server:Fri Oct 14 10:59:24 BST 2011
- ...
- Manually stop the server program

# Single service

- This will only support one connection
- Once `accept()` returns, it no longer accepts another connection
- Need to call `accept()` again
- But this will block so how to handle multiple connections?
- Create a thread for each connection

# Multiple connections

- Stick the call to `accept()` in a loop
- Once it returns create a new Thread and pass it the Socket
- Loop will call `accept()` again to handle another connection
- Active connection closes and Thread dies
- Each thread has its own protocol object

# A server with multiple clients

```
int portNumber = 8250;

ServerSocket s = new ServerSocket(portNumber);

while(true)    // accept multiple clients
{
    Socket clientSoc = s.accept();
    Thread t = new ThreadedEchoHandler(clientSoc);
    t.start();    // Handle the client communication
}

...

Class ThreadedEchoHandler implements runnable {
// override run()
}
```



# UDP

- Unified Datagrams Protocol
- applications that communicate via datagrams send and receive completely independent packets of information.
- These clients and servers do not have and do not need a dedicated point-to-point channel.
- The delivery of datagrams to their destinations is not guaranteed. Nor is the order of their arrival.

# UDP Advantages

- Less overhead (no connection establishment)
- More efficient (no guaranteed delivery)
- Real-time applications (no error checking or flow-control)
  - E.g., weather, time, video, audio, games
- Data reception from more than one machine

# UDP Sockets

- To create UDP sockets

`java.net.DatagramSocket` class

- Client
- Server

- To send/receive data.

`java.net.DatagramPacket` class

# Create UDP Sockets

// A client datagram socket:

```
DatagramSocket clientSocket =  
new DatagramSocket();
```

// A server datagram socket:

```
DatagramSocket serverSocket =  
new DatagramSocket(port);
```

# Create a UDP packet

// to receive data from a remote machine

```
DatagramPacket packet =
```

```
new DatagramPacket(new byte[256], 256);
```

// to send data to a remote machine

```
DatagramPacket packet = new
```

```
    DatagramPacket( new byte[128], 128,  
        address, port );
```

# UDP Sockets

- A UDP socket can be used for both reading and writing packets.
- Read operations can block.
- Since there is no connections between client and server, it is no need to deal with different clients\servers by multiple threads
- Since there is no guaranteed delivery, a client \server should not wait for a response.

# UDP Server

```
import java.io.*;
import java.net.*;
class UDPServer
{
    public static void main(String args[]) throws Exception
    {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receive = new byte[1024];
        byte[] send = new byte[1024];
        while(true)
        {
            DatagramPacket receivePacket = new DatagramPacket(receive, receive.length);
            serverSocket.receive(receivePacket);
            String sentence = new String( receivePacket.getData());
            System.out.println("RECEIVED: " + sentence);
            InetAddress IPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();
            String capitalizedSentence = sentence.toUpperCase();
            send = capitalizedSentence.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(send, send.length, IPAddress,
port);
            serverSocket.send(sendPacket);
        }
    }
}
```

# UDP client

```
import java.io.*;
import java.net.*;
class UDPClient
{
    public static void main(String args[]) throws Exception
    {
        BufferedReader inFromUser = new BufferedReader(new
InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("localhost");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
IPAddress, 9876);
        clientSocket.send(sendPacket);
        DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}
```



# Higher Level Networking

- Sockets are the low-level interface
- Can handle any network connection type
- Java provides higher level connections
- HTTP connections
- Remote Method Invocation (RMI)
- Simple Object Access Protocol (SOAP)

# HTTP

- Hypertext Transport Protocol
- Protocol behind the web
- Can be used to send/receive files
- Built on top of TCP/IP (usually uses port 80)
- Java provides a few classes that you can use to access data over HTTP
- Also see `org.apache.http` library (in CW)

# URLs

- HTTP is based around a *Uniform Resource Locator (URL)* that specifies where a file is, e.g.

*[http://www.cs.nott.ac.uk/~jwl/G52APR\\_files/lecture5.pdf](http://www.cs.nott.ac.uk/~jwl/G52APR_files/lecture5.pdf)*

- Java encapsulates these in a class `URL`
- In `java.net` package
- Provides methods to access various parts of the `URL`

# URL class

- **URL**(String spec)  
Construct with the URL in spec
- **URL**(String protocol, String host, int port, String file)  
Construct a URL by assembling the URL from the parts
- Can throw **MalformedURLException**

# URL class

- **String `getFile()`**  
Get the file name of the **URL**
- **String `getPath()`**  
Get the path part of the **URL**
- **String `getHost()`**  
Get the hostname of the **URL**, if applicable
- And more...

# Fetching data from URL

- Quick and dirty method  
`InputStream openStream()`
- Returns an `InputStream` that you can read data from
- Fully-featured method

```
URLConnection openConnection()
```

```
URLConnection openConnection(Proxy proxy)
```

# URLConnection class

- Also in `java.net` package
- Represents a communications link between app and the **URL**
- Can read and write data to the referenced resource
- Can also get content as a Java object...

# URLConnection class

- **OutputStream** `getOutputStream()`  
Gets the OutputStream to use to send data
- **InputStream** `getInputStream()`  
gets the InputStream used to receive data
- **void** `connect()` **throws IOException**  
Connect to the resource
- **void** `disconnect()`  
Inform the object that you are unlikely to send any more data
- Many other methods ...



# URL connections

- To send to or retrieve information from a web server, we use the HTTP protocol through a client socket attached to port 80
- We can do this by using normal socket-based programming and sending the correct HTTP commands
- However, Java has specific support for the HTTP protocol through its **URLConnection** class
  - Its very easy to retrieve a file from a web server by simply providing the file's **URL** as a string

```
URL u = new URL("http://www.cs.nott.ac.uk");  
URLConnection c = u.openConnection();  
InputStream in = c.getInputStream();
```

- This sets up an input stream from a **URL** connection
  - Can turn this into a *Scanner* object for text processing
- The **URLConnection** class also has additional functionality related to the HTTP protocol
  - Querying the server for header information
  - Setting request properties

# A URL Client

1. Create a **URL**.
2. Retrieve the **URLConnection** object.
3. Set output capability on the **URLConnection**.
4. Open a connection to the resource.
5. Get an **InputStream(OutputStream)** from the connection.
6. Read/Write.
7. Close the input/output stream.

# Read web page via Java

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

public class ReadWebPage {
    public static void main(String[] args) throws IOException {
        String urltext = "http://www.nottingham.ac.uk/
                           computerscience/index.aspx";
        URL url = new URL(urltext);
        BufferedReader in = new BufferedReader(new
            InputStreamReader(url.openStream()));
        String inputLine;

        while ((inputLine = in.readLine()) != null) {
            // Process each line.
            System.out.println(inputLine);
        }
        in.close();
    }
}
```

# Inter-object communication

- Object orientation is all about objects ‘communicating’ with each other by calling each others’ methods
  - What if the objects are distributed across a network?
  - What if the objects are implemented in different languages? (eg Java and C++)

- There are three main mechanisms for inter object communication across a network
  - Language independent mechanism is called *CORBA*
    - Common Object Request Broker Architecture
    - A separate language neutral *broker object* handles all messages between the objects
    - Generally rather slow and complex because of its generality and ability to handle legacy code
  - If both objects are implemented in Java a much simpler mechanism is *Remote Method Invocation* (RMI)
  - Latest mechanism is SOAP (Simple Object Access Protocol) – XML wrapper

# RMI

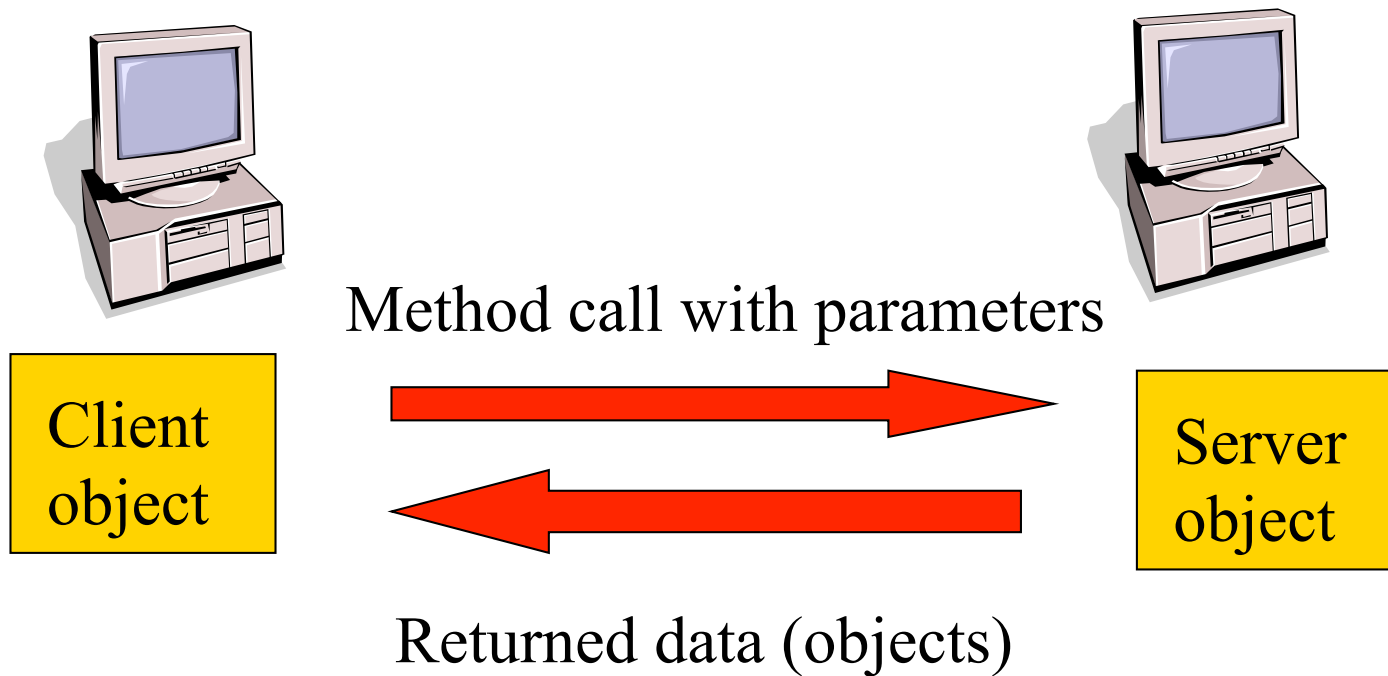
- Distributed Java
- Almost the same syntax and semantics used by non-distributed applications
- Allow code that defines behaviour and code that implements behaviour to remain separate and to run on separate JVMs
- The transport layer is TCP/IP

# Principles of RMI

- On top of TCP/IP, RMI originally used a protocol called *Java Remote Method Protocol* (JRMP). JRMP is proprietary.
- For increased interoperability RMI now uses the *Internet Inter-ORB Protocol* (IIOP). This protocol is language neutral and runs on TCP/IP providing a standard way to make method calls to remote objects.
- RMI is all about remote calls at runtime. It's not about compilation against a remote class.



- In RMI, a *client* object calls a method of a *server* object on a different machine
  - Client/server terminology only applies to the single method call



- All this seems easy but the actual behind the scenes issues are complex
  - Actual inter-object communication is done through separate *stub* objects which package remote method calls and parameters
  - The server registers its objects with a naming service
  - The clients finds the remote objects using a url : *“rmi://servername.com/”*
  - There are complex security issues involved also as calling a remote method locally can introduce viruses
- However, much of this is transparent to the programmer and RMI is relatively straightforward

# Remote interface

- An object becomes remote-enabled by implementing a remote interface, which has these characteristics:
  - A remote interface extends the interface *java.rmi.Remote*
  - Each method of the interface declares *java.rmi.RemoteException* in its throws clause, in addition to any application-specific exceptions

```
interface myInterface extends Remote {  
    public void myMethod();  
}
```

- A class that implements this remote interface can be used as a remote object.

# Steps To Develop An RMI Application

1. Design and implement the components of your distributed application
  - Define the remote interface(s)
  - Implement the remote object(s)
  - Implement the client(s)
2. Compile sources and generate stubs
3. Make required classes network accessible
4. Run the application

# A HelloWorld example

- The classic “Hello, World” Example using RMI!
- First, define the desired remote interface:

```
import java.rmi.*;
/* Hello Interface.
public interface IHello extends Remote {
    public String sayHello() throws RemoteException;
}
```

- A class that implements this remote interface can be used as a remote object. Clients can remotely invoke the `sayHello()` method which will return the string “Hello, World” to the client.

# Client program

```
import java.rmi.*;
public class HelloClient {
    public static void main(String[] args) {
        // Install a security manager!
        System.setSecurityManager(new RMISecurityManager());
        try {
            // Get a reference to the remote object.
            IHello server = (IHello)Naming.lookup("rmi://serverhost/HelloServer");
            System.out.println("Bound to: " + server);
            //Invoke the remote method.
            System.out.println(server.sayHello());
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

# Security Manager

- The default behavior when running a Java application is that no security manager is installed. A Java application can read and write files, open sockets, start print jobs and so on.
- Applets, on the other hand, immediately install a security manager that is quite restrictive.
- A security manager may be installed with a call to the static `setSecurityManager()` method in the `System` class.

# Security Manager

- Any time you load code from another source (as this client might be doing by dynamically downloading the stub class), you need a security manager.
- By default, the **RMI**SecurityManager restricts all code in the program from establishing network connections. But, this program needs network connections.
  - to reach the RMI registry
  - to contact the server objects
- So, Java requires that we inform the security manager through ***a policy file***.



# Naming

- The **Naming** class provides methods for storing and obtaining references to remote objects in the remote object registry.
- Callers on a remote (or local) host can *lookup* the remote object by name, obtain its reference, and then invoke remote methods on the object.
- **lookup()** is a static method of the **Naming** class that returns a reference to an object that implements the remote interface. Its single parameter contains a URL and the name of the object.

# Stub object

- The object references do not actually refer to objects on the server. Instead, these references refer to a stub class that must exist on the client.

```
IHello server =  
    (IHello)Naming.lookup( "rmi://serverhost/  
    HelloServer" );
```

- The stub class is in charge of object serialization and transmission. it's the stub object that actually gets called by the client with the line

```
System.out.println(server.sayHello());
```

# Server program

- We'll implement the remote object as a server
- The remote object server implementation should:
  - Declare and implement the remote method
  - Create and install a security manager
  - Create one or more instances of a remote object
  - Register the remote objects with the RMI remote object registry
- To make things simple, our remote object implementation will extend *java.rmi.server.UnicastRemoteObject*. This class provides for the “exporting” of a remote object by listening for incoming calls to the remote object on an anonymous port.

# Server program

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloServer extends UnicastRemoteObject implements IHello {
    private String name;
    public HelloServer(String name) throws RemoteException {
        super();
        his.name = name;
    }
    public String sayHello() {return "Hello, World!";}
    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            HelloServer obj = new HelloServer("HelloServer");
            Naming.rebind("rmi://serverhost/HelloServer", obj);
            System.out.println("HelloServer bound in registry!");
        }
        catch(Exception e) {
            System.out.println("HelloServer error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

# Compile the programs

- Now we can compile the client and server code:

**javac** IHello.java

**javac** HelloServer.java

**javac** HelloClient.java

- We next use the *rmic* utility to generate the required stub and skeleton classes:

**rmic** HelloServer

- This generates the stub and skeleton classes:

**HelloServer\_Stub.class**

**HelloServer\_Skel.class (Not needed in Java 2)**

# Compile programs

- Our next step would be to make the class files network accessible. For the moment, let's assume that all these class files are available locally to both the client and the server via their CLASSPATH. That way we do not have to worry about dynamic class downloading over the network.
- The files that the client must have in its CLASSPATH are:  
**!Hello.class**  
**HelloClient.class**  
**HelloServer.Stub.class**
- The files that the server must have in its CLASSPATH are:  
**!Hello.class**  
**HelloServer.class**  
**HelloServer\_Stub.class**  
**HelloServer\_Skel.class (Not needed in Java 2)**

# File policy

```
grant
{
    permission java.net.SocketPermission
        "*:1024-65535", "connect";
};
```

- This policy file allows an application to make any network connection to a port with port number at least 1024. (The RMI port is 1099 by default, and the server objects also use ports  $\geq 1024$ .)

# Run programs

- On the server:

//Start the rmiregistry:

**rmiregistry &**

//Start the server:

**java -Djava.security.policy=policy HelloServer**

- On the client:

//Start the client:

**java -Djava.security.policy=policy HelloClient**

- Get this wonderful output on the client:

**Hello, World!**



# SOAP

- Basic message framework
- Works with multiple languages
- Wraps messages in XML