Java Network Programming

Study Guide

Professor Wanlei Zhou
School of Information Technology
Deakin University

Contents

0. Unit Outline

0.1. Objective

The objective of this unit is to provide an up-to-date knowledge of network programming in Java to students. It covers most, if not all, aspects of Java network programming facilities. First, the lower level Java networking is overviewed which includes communication with sockets, Web URLs, and datagrams. Second, higher-level object-oriented networking is addressed which includes communication with homogenous RMI (remote method invocation) and heterogeneous CORBA (common object request broker architecture) in IDL (interface definition language). The unit also covers topics related to JDBC, concurrent programming, security, servlets, and Java Beans.

At the end of this unit, students should be familiar with the basic concepts, the architectures, the protocols, the advantages and limitations, and the recent development of various Java network programming technologies. The knowledge gained from the study of this unit will enable students to create network applications using the Java language. Students will also have adequate knowledge in tracking the current and future development in network programming in Java.

The Study Guide is divided into three parts. The first part (Sessions 1 and 2) overviews the fundamental knowledge required for this unit, including Java basics, communication systems, and the client-server model for network computing. The second part (Sessions 3, 4, 5, 6, 7, and 8) introduces the techniques used in Java network computing, including Java streams, connection-oriented communication in Java, developing Java programs for clients and servers, connectionless communication in Java, and multithreading in Java.. The third part (Sessions 9, 10, 11 and 12) addresses some advanced issues in Java networking, such as JDBC, Java RMI and CORBA, Java Servlets and Java Beans, and Java security.

0.2. Text Books

0.2.1. Main Text Book

Couch, Justin.

“Java 2 networking,” Justin Couch, New York : McGraw-Hill, c1999 ([JC]).

0.2.2. Reference Books

- “Beginning Java Networking,”, C. Darby et al., Wrox Press Ltd., 2001, ISBN: 1-861005-60-1 ([BJN]).
- “Java Network Programming.” 2nd ed., E. R. Harold, O’Reilly, 2000. ISBN: 1-56592-870-9 ([JNP]).
- “Java Network Programming”, 2nd ed., M. Hughes, M. Shoffner, and D. Hamner, Manning Publishing Co., 1999, ISBN: 1-884777-49-X [HSH]

- “Java Distributed Computing”, J. Farley, O’Reilly, 1998, ISBN: 1-56592-206-9 [FAR].
- “Java 2 Developer’s Handbook”, P. Heller and S. Roberts, Sybex, 1999, ISBN: 0-7821-2179-9 [Java2H]
- “Java 1.2 Unleashed”, J. Jaworski,: Macmillan Computer Publishing, 1998, ISBN: 1575213893 [Java2U]

0.3. Assessment

- Progressive assessment 40%
- Examination 60%

1. Java Language Basics

1.1. Study Points

- Understand the basic principles of the Java programming language.
- Be able to complete small programs in Java.
- Be familiar with object-oriented programming using Java.
- Be familiar with Java Applets.
- Be able to design simple user interfaces using Java AWT.

Reference: (1). Reader 1. "Learning Java Programming by Examples", by Wanlei Zhou, Lessons 1 through 9. (2). [Java2H]: Part 1, Chapters 1-5. (3). [Java2U] Chapters 1-10.

1.2. The Java Development Kit

Although this unit assumes that students have a basic knowledge in Java, students without Java knowledge can still study this unit without much trouble. I have assembled a special document (Reader 1) for those who have little knowledge in Java. Experiences show that most students with a basic programming knowledge in any other languages such as C or C++ can have a good understanding in Java in a week through the help of the document. Students with a solid knowledge in Java can skip this session. One comment about the example programs in Reader 1: they were accumulated over a period where Java versions have been fast evolving. Therefore, some of the programs may include deprecated methods for later Java versions.

1.2.1. The JDK

The Java Development Kit™ (JDK), created by Sun's JavaSoft subsidiary, is Java's base software environment. It includes all the software and documentation necessary to create Java applications or applets, and commands used to compile, run, package, and debug Java programs. Figure 1.1 shows how a Java program is created.

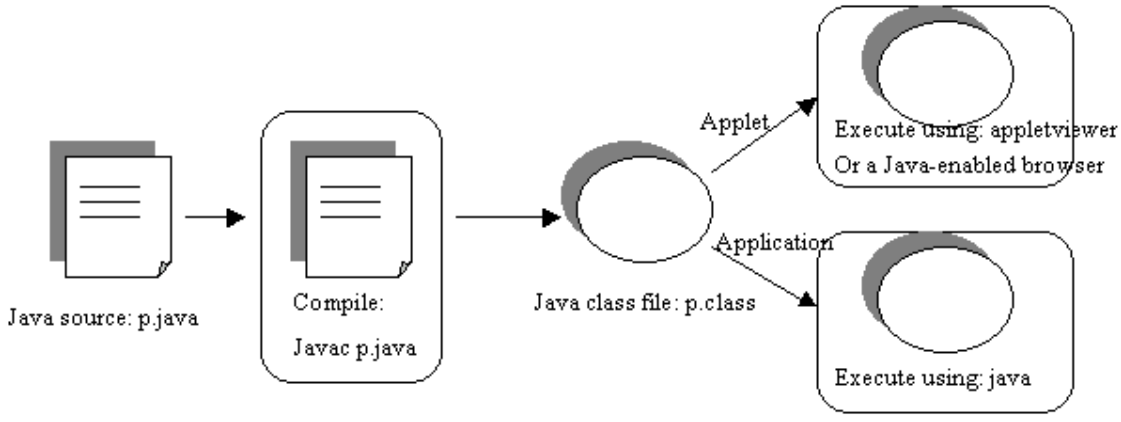

Figure 1.1. Creation of Java programs

The JDK is available on various operating systems in which the Java Core Application Program Interface (API), a standard library of classes for Java applications and applets, is guaranteed to execute on all Java implementations.

Java programming language has gone through a number of versions. Many examples of this unit are based on Java 2, aka JDK 1.2, the currently widely used version. Some features introduced in this unit need JDK 1.3. Currently Sun makes a Java Development Kit (JDK) freely available for Windows operating systems and Solaris. You can download it from http://java.sun.com/products/jdk/1.2/.

The JDK 1.2 API consists of 57 packages, all of which are in the Core API. Of the 53 Core API packages, 22 are part of the Java Foundations Classes (JFC). The Core API is the minimum subset of the Java API that must be supported by the Java Platform. All classes and interfaces that are part of the JDK but are not in the Core API are in the Standard Extension API. The JFC are Core API classes and interfaces that support GUI development for applets and applications.

1.2.2. The Java Virtual Machine

All compiled Java programs are Java classes. The Java virtual machine (JVM) is a software implementation of a CPU designed to run Java programs. Since JVM is typically not implemented in hardware, hence the term of *virtual*.

The JVM is a software layer that enables Java to be independent of physical CPU type. Therefore, Java programs can be executed on any platform as long as the JVM is installed.

Java programs can be applets or pure Java applications. Java applications are Java programs that do not require a WWW browser. These applications run under the JDK's *java* command and are started from a user's command line or shell script. A Java application should contain a *public static void main* method (equivalent to a *main()* function in C).

Java applets are Java programs that are downloaded from WWW sites and executed in web browsers. This is the most popular use of Java programs to deliver network-based applications through WWW sites and web browsers. Instead of a *main()* method, applets use methods suited to downloaded programs that display output in a web browser window.

1.3. Java Basics

1.3.1. Introductory Lessons on Java

Students are encouraged to read lessons 1, 2, 3, and 4 of Reader 1 and test run all the programs in these four lessons. In lesson 1, we introduce a number of "first" Java programs, including a few simple examples of Java application and applet. We then introduce the concepts of object-oriented programming using Java and basic font in Java.

In lesson 2, we introduce some basic components of the Java language, including the dates, strings, arrays, loops. We also introduce some mechanisms for parameter passing, argument handling, and type converting.

In lesson 3, we outline the principles of object-oriented programming in Java, including the constructors and inheritance. In object-oriented programs data is represented by objects. Objects have two sections, fields (instance variables) and methods. Fields tell you what an object is. Methods tell you what an object does. These fields and methods are closely tied to the object's real world characteristics and behavior. When a program is run messages are passed back and forth between objects. When an object receives a message it responds accordingly as defined by its methods. Most Java programs are organised using the object-oriented concepts.

In lesson 4, we emphasis on the principles of applet design since applets are the major mechanism of network programming in Java. We concentrate on topics of the HTML pages related to applets, the alignment and passing parameters.

Each topic of a lesson is illustrated using simple examples. Students are encouraged to test run these examples and understand the Java components and statements illustrated in these examples.

1.3.2. Applets

According to Sun, "An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application....The Applet class provides a standard interface between applets and their environment." There are four definitions of applet:

- A small application
- A secure program that runs inside a web browser
- A subclass of java.applet.Applet
- An instance of a subclass of java.applet.Applet

Applets are executed in the “sandbox” of the Java virtual machine of a web browser. An applet can:

- Draw pictures on a web page
- Create a new window and draw in it.
- Play sounds.
- Receive input from the user through the keyboard or the mouse.
- Make a network connection to the server from which it came and can send to and receive arbitrary data from that server.

Anything you can do with these abilities you can do in an applet. However, an applet cannot:

- Write data on any of the host's disks.
- Read any data from the host's disks without the user's permission. In some environments, notably Netscape, an applet cannot read data from the user's disks even with permission.

- Delete files.
- Read from or write to arbitrary blocks of memory, even on a non-memory-protected operating system like the MacOS. All memory access is strictly controlled.
- Make a network connection to a host on the Internet other than the one from which it was downloaded.
- Call the native API directly (though Java API calls may eventually lead back to native API calls).
- Introduce a virus or trojan horse into the host system.
- An applet is not supposed to be able to crash the host system. However in practice Java isn't quite stable enough to make this claim yet.

1.4. Further Topics in Java

We use five lessons (lessons 5, 6, 7, 8, and 9) in the Reader 1 document to introduce some further topics in Java programming.

1.4.1. Graphics and Colors

In lesson 5, we introduce the topic on graphics and colours, in particular, we illustrate the use of lines and rectangles, the creation of polygons, ellipses, and arcs, and the use of different fonts and colours. Apart from examples on each feature, we also use an example to illustrate the use of all these features.

1.4.2. Animation, Image, Threads, Sounds and the Use of Swing

In lesson 6, we introduce the use of animation, images, threads, sound and the use of Swing. This lesson consists of examples of a digital clock, the colurs swirl, the use of images and the adjustment of images, some simple animation mechanisms, such as a bouncing circle and an animated cat, and also some techniques in using sound.

1.4.3. Events Handling

In lesson 7, we introduce the use of events and interactivity in Java. We use examples to illustrate the techniques to draw spots, normal lines and scribble lines. We then introduce the mechanism of keyboard control and event handling in different versions of Java JDK.

1.4.4. Abstract Windowing Toolkit

In lesson 8, we introduced the use of the abstract windowling toolkit (AWT), a graphics tool package provided by the JDK. In particular, we introduced the use of buttons, check boxes, radio buttons, choice menus, and text fields. We then introduce the mechanisms for flow layout, grid layout, and border layout. In addition to examples for each topic, we finally use an example of a color switcher to illustrate the use of all these features.

In lesson 9, we introduce some advanced usage of the AWT. In particular, we illustrate the use of text areas, the mechanism to scroll a list of text, the use of sliding bar, simple

pop-up windows, pop-up windows with dialogue, and pop-up windows with dialogu and menu. We also introduce the technique to create links inside applets in this lesson.

Reader 1 also contains other lessons related to networking programming. You can skip these lessons since they will be covered in more details in the following sessions.

1.5. Java Versions and Network APIs

1.5.1. Java Versions

Java 1.0 gave the programmer the basic networking capability, such as connecting to other computers and listening to connection requests from other computers. On top of the basic capability, some APIs, such as the Universal Resource Locator (URL) handlers, for high-level Internet related protocols were provided. However, using the Java 1.0 to create large-scale applications was difficult since it was hard to customize connections..

Java 1.1 added more control over the connections to the APIs. These options allowed the Java programmer to build networked programs with a similar quality as programs built using native code. It also added the ability to create customized network handlers while still retaining the high-level URL handler capabilities. The third useful part added to the Java 1.1 version is the ability to create multicast connections within Java code.

Java 1.2 (Java 2) introduced few changes in terms of network computing capability. The only real new change was the new fine-grain security model, but in day-to-day application programming, it is not much of an issue. In Java 2, the changes moved to the supporting APIs. APIs that relied on the core networking capabilities had a lot more functionality added to them.

Java version 1.3 is an enhancement to Java 2, Standard Edition (J2SE). It includes a new Java virtual machine (VM)--the Java HotSpot Client--which is aimed at improving client performance in an enterprise desktop.. The Java HotSpot Client VM is totally new, the emphasis is having it tuned for client performance--reducing application start-up time, how quickly an application comes to life on a desktop. Java 1.3 also added support for lightweight directory access protocol (LDAP), which gives a desktop client directory connectivity to access information across the enterprise, said Connell. J2SE 1.3 also has new Java applet caching functionality, which will enable a developer to deploy a Web application more quickly by downloading and storing Java applets on a local hard drive for immediate access.

1.5.2. Java Networking APIs

Java networking APIs are provided in a series of layered abstractions on which the programmer can decide to work. Java's basic networking capabilities are split between the *java.net* and the *java.io* packages. Programmers are able to use these packages to establish and communicate over a network connection and pass moderately complex data over it.

Once a connection has been established, there are a number of opportunities for the programmer, e.g., defining custom protocols, using existing protocols, or building a

higher-level API on top of the basic services. Java provides a number of these APIs. At the bottom of the heap are the URL content handlers contained in the *java.net* package.

Next in the line is the Remote Method Invocation (RMI), which uses *java.net* and *java.io* to implement its functionality as well as a good deal of native code. RMI comes in a number of packages: *java.rmi*, *java.rmi.dgc*, *java.rmi.registry*, and *java.rmi.server*. RMI uses the network facilities to pass references to Java classes over the network.

Another two Java techniques, the Java servlets and Java Beans, can be regarded as function-oriented mechanisms. Servlets can be viewed as server-side applets in which they responds to client requests via the HTTP protocol. Java Beans are based on the component-based system concepts, in which reusable components can be distributed across the network.

2. Network Communication and the Client-Server Model

2.1. Study Points

- Understand the Internet protocol architecture.
- Understand the basic concepts of the TCP/IP protocol suit.
- Understand the IPv6 concepts.
- Understand the basic client-server model.
- Understand the extensions to the basic client-server model and the service discovery mechanisms.
- Be familiar with the mechanisms in client-server communication
- Understand the different semantics of message passing.
- Understand the RPC mechanism.

Reference: (1). Any book on computer networks. (2). A. Goscinski and W. Zhou, "The Client-Server Model and Systems", Technical Report, Deakin University, TR C97/04, 1997.

2.2. Communication Protocol Architectures

2.2.1. The OSI Protocol Architecture

The OSI (Open System Interconnection) Reference Model was developed by ISO (International Standards Organisation) as a model for implementing data communication between cooperating systems. It has seven layers.

- Application: providing end-user applications.
- Presentation: translating the information to be exchanged into terms that are understood by the end systems.
- Session: for the establishment, maintenance, and termination of connections.
- Transport: for reliable transfer of data between end systems.
- Network: for routing the data to its destination and for network addressing.
- Data link: preparing data in frames for transfer over the link and error detection and correction in frames.
- Physical: defining the electrical and mechanical standards and signaling requirements for establishing, maintaining, and terminating connections.

Figure 2.1 depicts the OSI process and peer-to-peer communication, where SDU represent service data unit, Hi (i = 2 … 7) headers of layer i, and T2 is the trailer for layer

2.

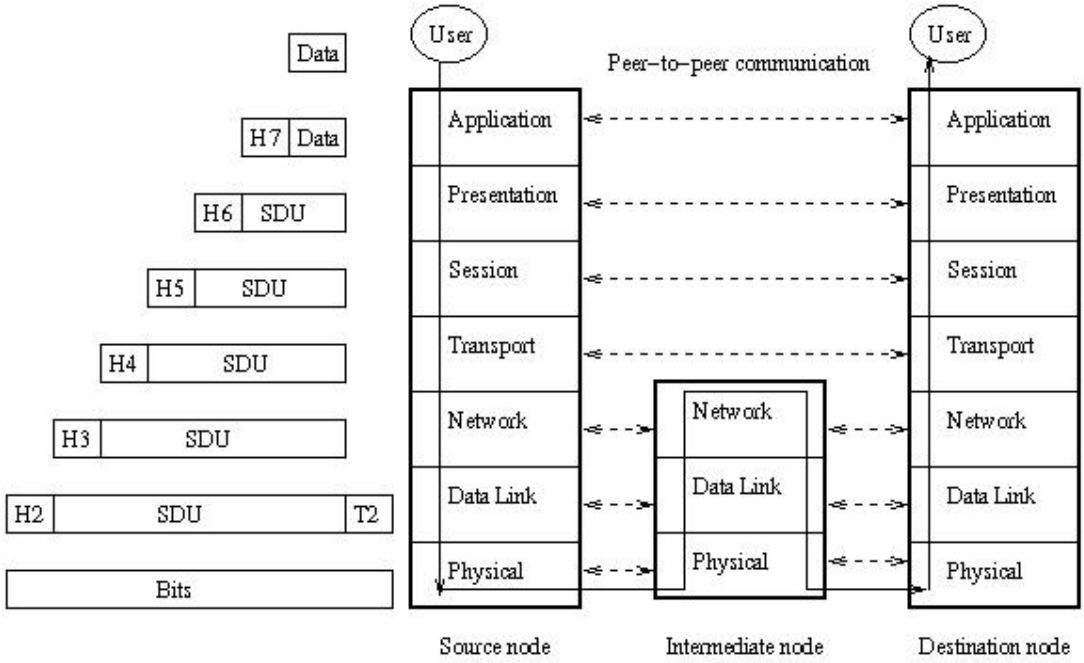

Figure 2.1 OSI process and peer-to-peer communication

2.2.2. Internet Architecture

Internet is the largest data network in the world. It is an interconnection of several packet-switched networks and has a layered architecture. Figure 2.2 shows the comparison of Internet and OSI architectures.

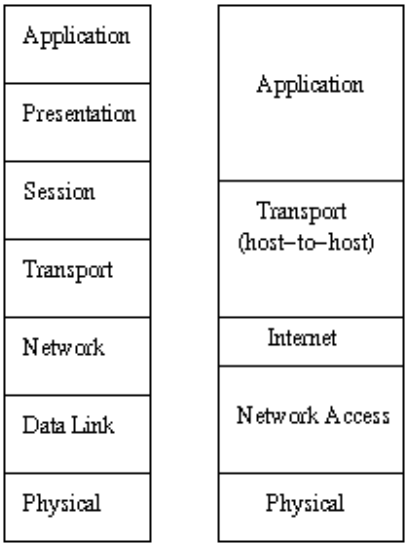

Figure 2.2 Comparison of Internet and OSI architectures

Internet layers:

- Network access layer: It relies on the data link and physical layer protocols of the appropriate network and no specific protocols are defined.
- Internet layer: The Internet Protocol (IP) defined for this layer is a simple connectionless datagram protocol. It offers no error recovery and any error packets are simply discarded.
- Transport layer: Two protocols are defined: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is a connection-oriented protocol that permits the reliable transfer of data between the source and destination end users. UDP is a connectionless protocol that offers neither error recovery and nor flow control.
- User process layer (application): It describes the applications and technologies that are used to provide end-user services.

2.3. BSD Internet Domain Sockets

The ARPANET sponsored by the Advanced Research Projects Agency (ARPA) and developed during late 1960s and early 1970s is a milestone for computer networks. In the early 1980s, a new family of protocols was specified as the standard for the ARPANET. Although the accurate name for this family of protocols is the "DARPA Internet protocol suite," it is commonly referred as the TCP/IP protocol suite, or just TCP/IP.

The *Internet domain sockets* on BSD UNIX use the TCP/IP protocol suite as the communication protocols among processes generally located on different computers across a network. We introduce the TCP/IP in this section.

2.3.1. Overview

Communications between computers connected by computer networks use well-defined protocols. A protocol is a set of rules and conventions agreed by all of the communication participants. As we have mentioned, in the OSI reference model, the communication protocols are modelled in seven layers. Layered models are easier to understand and make the implementation more manageable. A *protocol suite* is defined as a collection of protocols from more than one layer that forms a basis of a useful network. This collection is also called a *protocol family*. The TCP/IP protocol suite is an example.

There are many protocols defined in the TCP/IP protocol suite. We are going to describe three of them: the Transport Control Protocol (TCP), the User Datagram Protocol (UDP) and the Internet Protocol (IP). If using OSI reference model the TCP and UDP protocols are Transport layer protocols, while the IP protocol is a Network layer protocol. Figure 2.3 illustrates the relationship of these protocols and their positions in the OSI reference model.

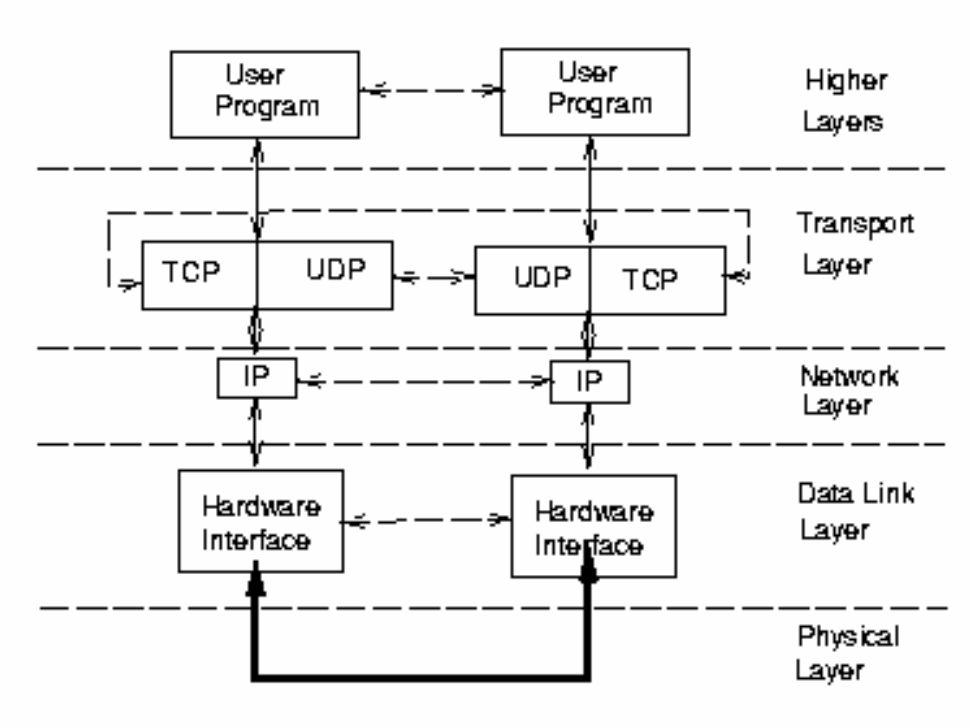

Figure 2.3. The Layered TCP/IP protocol suite

The TCP protocol is a connection-oriented protocol that provides a reliable, full-duplex, byte stream for interprocess communications. The UDP protocol, on the other hand, is a connectionless protocol that provides an unreliable datagram service: there is no guarantee that UDP datagrams ever reach their intended destination. Acknowledgement must be used in order to provide reliable services when using the UDP protocol.

2.3.2. Network Layer: IP

The IP protocol is connectionless and unreliable. It is based on *internet datagrams*. The protocol takes a datagram from the transport layer (for example, from the TCP protocol). A datagram is up to 64k bytes long and it may be part of a longer message. Each datagram is transmitted over the network independently so the communication ls connectionless. During transmission, a datagram may be lost or may be further fragmented into smaller units (called *IP packets*) as it goes through the protocol layers. When all the IP packets of a datagram finally arrive the destination computer, they are reassembled to form the datagram and then transferred to the transport layer of the destination site. If any of the IP packets of a datagram are lost or corrupted, the entire datagram is discarded by the destination site so the IP protocol is therefore unreliable because it cannot guarantee the delivery of a datagram.

The IP datagram consists of a header and a text part. The header includes information such as the type of service, the length of the header, the length of the text part, the address of the source computer, the address of the destination computer, and other information.

It is the IP layer that handles the routing through networks. The Internet address is used to identify networks and computers and is used in an IP datagram header to denote the source and destination computer addresses. An Internet address has 32 bits and encodes both a network ID number and a host ID number. Every host on a TCP/IP internet must have a unique Internet address. The network ID numbers are assigned by some kind of authority, e.g., the Network Information Center (NIC) located at SRI International. While the host ID numbers are assigned locally.

The common notation of an Internet address is to use 4 bytes, as shown in the following:

```
field_1.field_2.field_3.field_4
```

Where $0 <= field_i <= 255_{10}$ (FF_{16}), $1 <= i <= 4$.

Depending on the network's class (described below), the network number can be $field_1$, or $field_1.field_2$ or $field_1.field_2.field_3$. That means the host number can be $field_2.field_3.field_4$, $field_3.field_4$ or $field_4$.

Networks are classified into 3 classes, as listed in the next Table.

Class	**Binary number of $field_1$**	**Network ID (decimal)**
A	000 000 – 0111 111	0 - 126
B	1000 000 – 1011 1111	128 – 191.254
C	1100 0000 – 1101 1111	192 – 223.254.254

A brief description of these classes follows:

Class A: networks are the largest networks with more than 65,536 hosts. A class A network's network ID number is $field_1$.

Class B: networks are mid-size networks with host IDs ranging from 256 to 65,536. A class B network's network ID number is $field_1.field_2$.

Class C: networks are the smallest networks with up to 256 hosts A class C network's network ID number is $field_1.field_2.field_3$.

Figure 2.4 illustrates the Internet address formats of these three network classes.

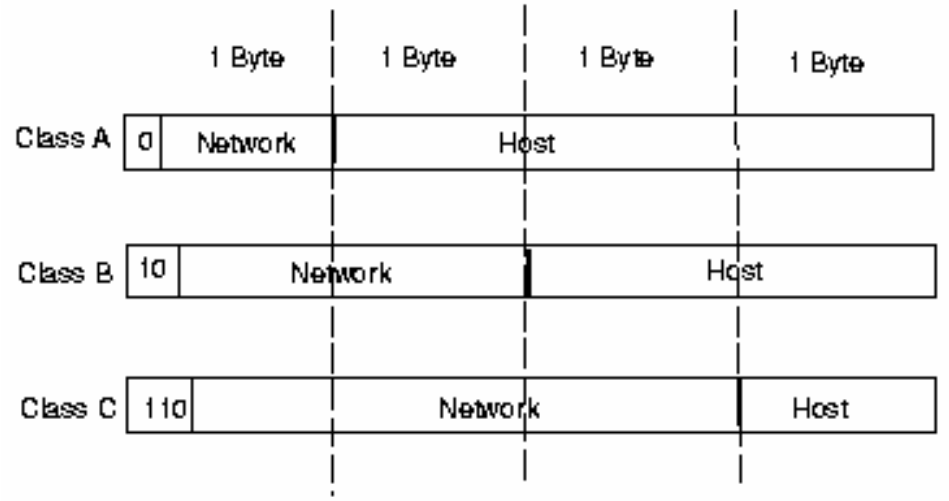

Figure 2.4. Internet network classes

For example, if we have an Internet address 98.15.12.63, we can tell that it is a class A network because $field_1$ is within the range of 0 - 126. Its network ID number is 98 and host ID number is 15.12.63. If we have an Internet address 130.194.1.106, we can tell that it is a class B network because the $field_1.field_2$ is within the range of 128 - 191.254. Its network ID number is 130.194 and host ID number is 1.106.

Some IP addresses have significant meanings. For example, the address of 127.0.0.1 is the address of the local machine. It is used for allowing IP communications to the local machine so that sockets and other systems may run even the machine is isolated from the network.

It is evident that an Internet address can only be assigned to one host. But a host can have several Internet addresses. This is because that in some situations, we want a host to be connected to several networks.

Although an Internet address clearly specifies the address of a host, few persons want to use Internet addresses directly: they are too hard to remember. Domain Name System (DNS) is used to name host addresses in more human-oriented way and to find the Internet addresses corresponding to machine names.

The DNS is a hierarchical naming system: its name space is partitioned into sub-domains, which can themselves be further divided. The DNS is also a distributed system: the name space is delegated to local sites that are responsible for maintaining their part of the database. Programs called *name servers* manage the database.

The DNS name space can be represented as a tree, with the nodes in the tree representing *domain names*. A *fully qualified domain name* is identified by the components (nodes) of the path from the domain name to the root. A component is an arbitrary string of up to 63 octets in length; the length of a fully qualified domain name is limited to 256 octets. By convention, a domain name is written as a dot-separated sequence of components, listed right to left, starting with the component closet to the root. The root is omitted from the name. Thus, *wan_res.cm.deakin.edu.au* is a fully qualified domain names. It is certainly easier to be remembered than the corresponding Internet addresses 139.130.118.102.

DNS name space is divided into *zones of authority*, and name servers have complete control of the names within their zones (domains). For easier management of domains, a large domain can be split into smaller sub-domains, and name servers can delegate authority to other name servers for sub-domains. For example, if *edu.au* represents the domain of all educational institutions in Australia, then *deakin.edu.au* and *anu.edu.au* are its two sub-domains. Queries for DNS information within sub-domain *deakin.edu.au* are first dealt with by the name server of this sub-domain. If this name server cannot answer a query, the query is then directed to the name server of *edu.au* domain. At last, the name server of the root can answer the query.

2.3.3. Transport Layer: TCP and UDP

As we have shown in Figure 2.3, user processes interact with the TCP/IP protocol suite by sending and receiving either TCP data or UDP data. To emphasise that the IP protocol is used, we sometimes refer them as the TCP/IP or UDP/IP protocols.

TCP provides a connection-oriented, reliable, full-duplex, byte-stream service, similar to a virtual circuit, to an application program. UDP, on the other hand, provides a connectionless, unreliable datagram service to an application program.

As we mentioned in the previous section, the Internet address is used to identify networks and computers. In order to let many processes use the TCP or UDP simultaneously (these processes may reside on any computers of a network), both protocols use 16-bit integer *port numbers* for identifying data associated with each user process. The association of port numbers and user processes last as long as the communication, so the following 5-tuple uniquely identifies a communication:

- the protocol (TCP or UDP),
- the local computer's Internet address,
- the local port number,
- the foreign computer's Internet address,
- the foreign port number.

For example, if we have a communication using TCP protocol. The server is on a host with domain name of *wan_res.cm.deakin.edu.au* (Internet address 139.130.118.102), using port number 5100. The client is on a host with domain name of *sky3.cm.deakin.edu.au* (Internet address 139.130.118.5), using port number 5101. The 5-tuple which uniquely defines the communication is:

{tcp, 139.130.118.102, 5100, 139.130.118.5, 5101}

Because the host name is easier to understand and there are some system calls to convert between a host name and its Internet address, the above 5-tuple can then be written as:

{tcp, wan_res.cm.deakin.edu.au, 5100, sky3.cm.deakin.edu.au, 5101}

Because *wan_res.cm.deakin.edu.au* and *sky3.cm.deakin.edu.au* are within the same sub-domain, we can even write the 5-tuple as:

{tcp, wan_res, 5100, sky3, 5101}

There are some restrictions in using port numbers. In TCP and UDP, port numbers in the range 1 through 255 are reserved. All well-known ports (some commonly used utilities use these ports) are in this range. For example, the File Transfer Protocol (FTP) server uses the well-known port number 21 (decimal). Some operating systems also reserve additional ports for privileged usages. For example, 4.3BSD reserves ports 1-1023 for superuser processes. Only port numbers of 1024 or greater can be assigned by user processes.

A TCP protocol entity accepts arbitrarily long messages from user processes, breaks them into datagrams of up to 64k bytes, and sends them to the IP layer. Before the real communication happens, a connection must be set up between the sender and the recipient. After the communication, the connection must be disconnected.

As the IP layer does not guarantee the proper delivery of a datagram, it is the responsibility of the transport layer to ensure that a datagram arrives at the destination properly using time-out and retransmission techniques. Also as datagrams are transmitted independently, the datagrams of a message may arrive at the destination out of order and it is also the TCP protocol’s responsibility to reassemble them into the message in the proper sequence.

Each datagram submitted by the TCP to IP layer contains a TCP header and a data part. The whole TCP datagram is viewed by the IP as data only and an IP header is added to form an IP datagram. The TCP header contains the source port number, the destination port number, the sequence number, and other information.

The TCP protocol has a well-defined service interface. There are primitives used to actively and passively initiate connection, to send and receive data, to gracefully and abruptly terminate connections, and to ask for te status of a connection.

A UDP protocol entity also accepts arbitrarily long messages from user processes, breaks them into datagrams of up to 64k bytes, and sends them to the IP layer. Unlike the TCP protocol, no connection is involved and to guarantee of delivery or sequencing. In effect, UDP is simply a user interface to IP. A header is also added into the datagram by UDP, which contains the source port number and the destination port number.

2.3.4. The Internet

The current “Internet" can be defined as "the collection of all computers that can communicate, using the Internet protocol suite, with the computers and networks registered with the *Internet Network Information Center* (*InterNIC*)." This definition includes all computers to which you can directly send Internet Protocol packets (or indirectly, through a firewall).

Internet Protocol enables communication between computers on the Internet by routing data from a source computer to a destination computer. However, computer-to-computer communication only solves half of the network communication problem. In order for an application program, such as a mail program, to communicate with another application, such as a mail server, there needs to be a way to send data to specific programs within a computer.

Ports, or addresses within a computer, are used to enable communication between programs. An application server, such as a Web server or an FTP server, listens on a particular port for service requests, performs whatever service is requested of it, and returns information to the port used by the application program requesting the service.

Popular Internet application protocols are associated with *well-known ports*. The server programs that implement these protocols listen on these ports for service requests. The well-known ports for some common Internet application protocols are shown in Table 2.1.

Port	**Protocol**
21	File transfer protocol (ftp)
23	Telnet protocol (telnet)
25	Simple Mail Transfer Protocol (SMTP)
80	Hypertext Transfer Protocol (HTTP)

Table 2.1. Common Internet application protocols and their ports

2.4. The Next Generation Internet Protocol: IPv6

2.4.1. Why IPv6?

The primary motivation for change from IPv4 to IPv6 is the limited address space. The 32-bit IP address can only include over a million networks in the Internet. At the current growth rate, each of the possible network prefixes will soon be assigned and no further growth will be possible.

The second motivation for change comes from requirements of new applications, especially applications that require real-time delivery of audio and video data. The current IP has limited capabilities for routing real-time data.

Challenges faced by IPv4 can be summarized:

- Address spaces
    - growth of the Internet. Maximum: 4 billion
    - when will addresses run out? Estimates: 2005
    - single IP addresses for devices
- Mobile Internet
    - Internet services from everywhere
    - Removing location dependency
- Security
    - End-to-end encryption

Data Integrity, authentication Requirements for the new protocol can be summarized as follows:

- Support billions of hosts
- Reduce size of routing tables
- Simplify protocol, process packets faster
- Provide better security (authentication & privacy)
- Better QoS (particularly for real-time data)
- Aid multicasting, anycasting
- Make it possible for a host to roam without changing its address

2.4.2. IPv6 Features

IPv6 retains many design features of IPv4. It's connectionless. IPv6 has the following new features:

- Address size: Instead of 32, each IPv6 address is 128 bits. The address space is large enough for many years of growth of Internet.
- Header format: The IPv6 header has a completely format compared to IPv4 headers.
- Extension header: Unlike IPv4, which uses a single header format for all datagrams, IPv6 encodes information into separate headers. A datagram of IPv6 contains a base header followed by 0 or more extension headers, followed by data.
- Support for audio and video. IPv6 includes a mechanism that allows a sender and receiver o establish a high-quality path through the underlying network and to associate datagrams with that path.
- Extensible protocol. Unlike IPv4, IPv6 does not specify all possible protocol features. Instead, the designers have provided a scheme that allows a sender to add additional informaion to a datagram. The extension makes IPv6 more flexible than IPv4.

2.4.3. Discussions on IPv6

Why does IPv6 use separate extension headers?

- Economy: Partitioning the datagram functionality into separate headers is economical because it saves space. Having separate headers in IPv6 makes it possible to define a large set of features without requiring each datagram header to have at least one field for each feature.
- Extensibility: When adding a new feature, existing IPv6 protocol headers can remain unchanged. A new *next header* type can be defined as well as a new header format.
- The chief advantage of placing new functionality in a new header lies in the ability to experiment with a new feature before changing all computers in the Internet.

IPv6 characteristics

- Practically unlimited address space
- Simplification of packet header

- Optional header fields (better support for options)
- Authentication and Privacy (more Security)
- More attention to type of service

Plug & Play - Better Configuration optionsAdditional features of IPv6:

- Network management: auto configuration
    - Plug-and-Play.
    - Automate network address renumbering
    - DHCP support is mandated: Every host can download their network configurations from a server at startup time
    - Address changes are automated
        - Stateless ; Routers advertise prefixes that identify the subnet(s) associated with a link ; Hosts generate an “interface token” that uniquely identifies an interface on a subnet ; An address is formed by combining the two.
        - Stateful ; Clients obtain address and / or configuration from a DHCP server ; DHCP server maintains the database and has a tight control over address assignments.
- Mobile IPv6:
    - IPv6 Mobility is based on core features of IPv6
    - The base IPv6 was designed to support Mobility
    - Mobility is not an “Add-on” features
        - All IPv6 Networks are IPv6-Mobile Ready
        - All IPv6 nodes are IPv6-Mobile Ready
        - All IPv6 LANs / Subnets are IPv6 Mobile Ready
    - IPv6 Neighbor Discovery and Address: Autoconfiguration allow hosts to operate in any location without any special supportNo single point of failure (Home Agent)
    - More Scalable : Better Performance
        - Less traffic through Home Link
        - Less redirection / re-routing (Traffic Optimisation)IPv6 Security:
- Security features are standardized and mandated. All implementations must offer them
    - No Change to applications
- Authentication (Packet signing)
- Encryption (Data Confidentiality)

- End-to-End security Model
    - Protects DHCP
    - Protects DNS
    - Protects IPv6 Mobility
    - Protects End-to-End traffic over IPv4 networks

2.5. The Client-Server Model

2.5.1. The Basic Client-Server Model

Figure 2.5 shows a basic client-server model. In this case, the client and server processes execute on two different computers. They communicate at the *virtual* (logical) level by exchanging requests and responses. In order to achieve this virtual communication physical messages between these two processes are sent. This implies that operating systems of computers and a communication system of a distributed computing system are actively involved in the service provision.

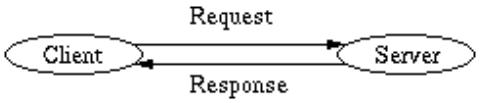

Figure 2.5. The basic client-server model

A more detailed client-server model has three components:

- *Service*: A service is a software entity that runs on one or more machines. It provides an abstraction of a set of well-defined operations.
- *Server*: A server is an instance of a particular service running on a single machine.
- *Client*: A client is a software entity that exploits services provided by servers. A client can but does not have to interface directly with a human user.

Figure 2.6 illustrates the involvement of a communication facility and the operating system in a client-server communication.

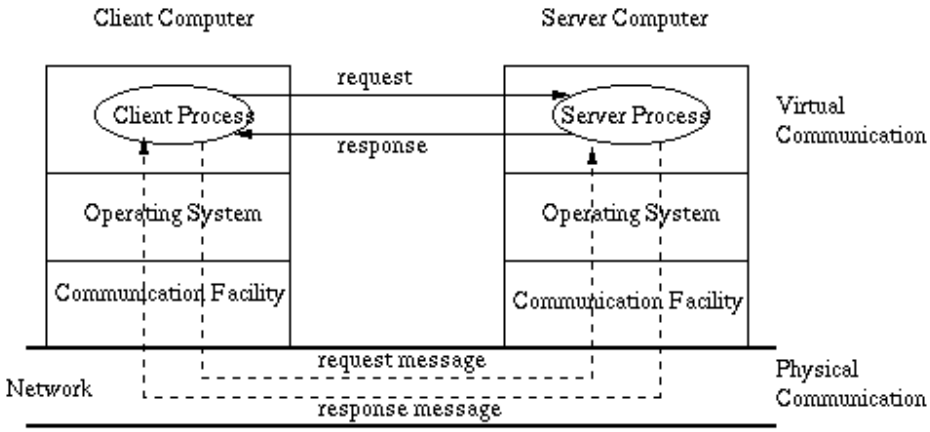

Figure 2.6. The physical implementation of the client-server model

The client-server model has the following advantages:

- Simplicity
- Modularity
- Extensibility
- Flexibility

The three major problems of the client-server model are:

- A server failure may affect the service
- A server is a potential bottleneck
- The cost/reliability trade-off

2.5.2. Client-Server Cooperation and the Extensions to the Client-Server Model

Clients and servers can cooperate in various forms.

- Single server. Figure 2.7 illustrates a printer server providing services to *n* clients.

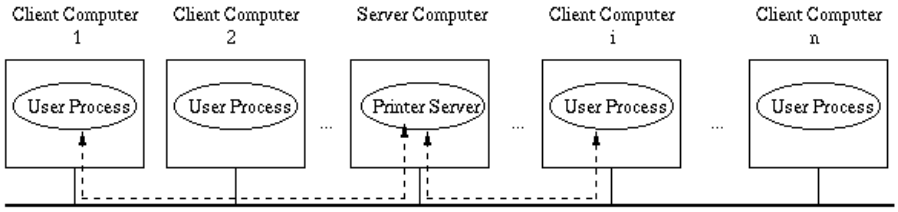

Figure 2.7. A single service example

- Chain-based servers. Figure 2.8 shows an example of a chain of cooperating servers.

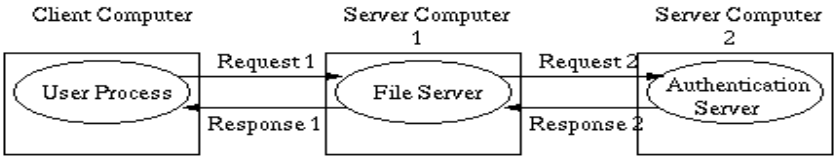

Figure 2.8. An example of a chain of cooperating servers

- Multiple servers
  - parallel execution of programs on a cluster of workstations
  - distributed databases
  - cooperative workgroups
  - service multiplication to increase performance, reliability and availability
- Group servers. Figure 2.9 shows a number of structures of group servers.

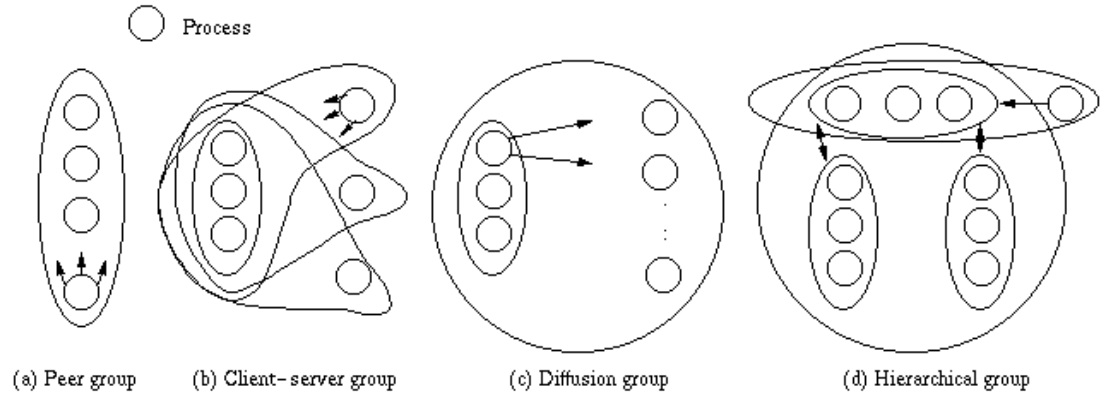

Figure 2.9. Group structures

Two issues have to be considered in group communication:

- Group membership discovery: the current status and the membership of the group.
- Group operations: *create, destroy, join* and *leave*.

A client and server can cooperate either directly or indirectly. In the former case there is no additional entity that participates in exchanging requests and responses between a client and a server. Indirect cooperation in the client-server model requires two additional entities, called *agents*, to request a service and to be provided with the requested service. Figure 2.10 shows such an extension.

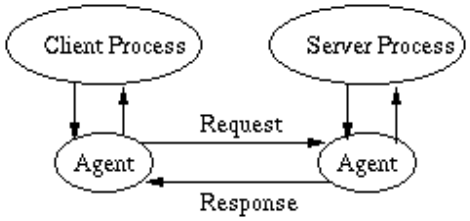

Figure 2.10. Indirect client-server cooperation

The role of these agents can vary from a simple communication module which hides communication network details to an entity which is involved in mediating between clients and servers, resolving heterogeneity issues, and managing resources and cooperating servers.

The three-tier client-server architecture has the following components:

- User interface and presentation processing. These components are responsible for accepting inputs and presenting the results. We say these components belong to the client tier;
- Computational function processing. These components are responsible for providing transparent, reliable, secure, and efficient distributed computing. They are also responsible for performing necessary processing to solve a particular application problem. We say these components belong to the application tier.
- Data access processing. These components are responsible for accessing data stored on external storage devices (such as disk drives). We say these components belong to the back-end tier.

Figure 2.11 shows some example of three-tier configuration.

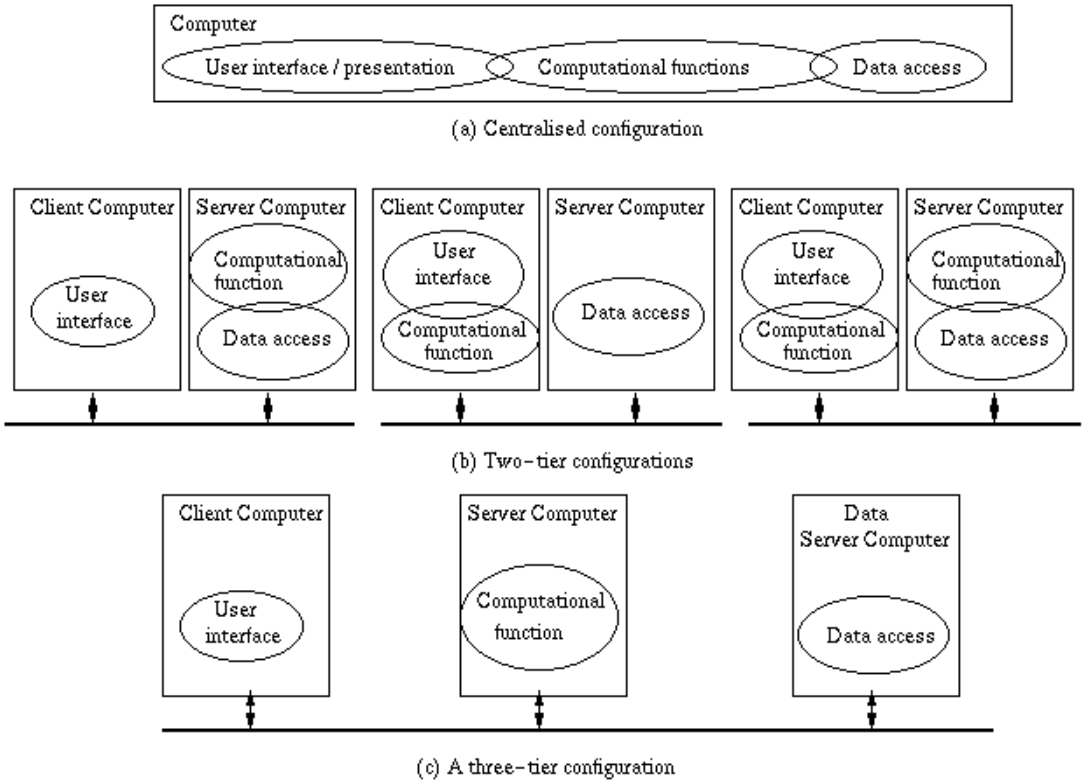

Figure 2.11. Examples of three-tier configurations

Advantages of the three-tier model:

- Better transparency.
- Better scalability.
- Better concurrency, flexibility, reusability, load balancing, and reliability.

2.5.3. Service Discovery

To invoke a desired service a client must know both whether there is a server which is able to provide this service and its characteristics, and its name and location. This is the issue of *service discovery*. In the case of a simple distributed computing system, where there are only a few servers, there is no need to identify an existence of a desired server. Information about all available servers is available a priori. This implies that service discovery is restricted to locating the server which provides the desired service. On the other hand, in a large distributed computing system which is a federation of a set of distributed computing systems, with many service providers who offer and withdraw these services dynamically, there is a need to learn both whether a proper service (e.g., a very fast colour printer of high quality) is available at a given time, and if so its name and location.

Service discovery is achieved through the following modes:

- Computer address (number) is hardwired into client code, as shown in Figure 2.12.

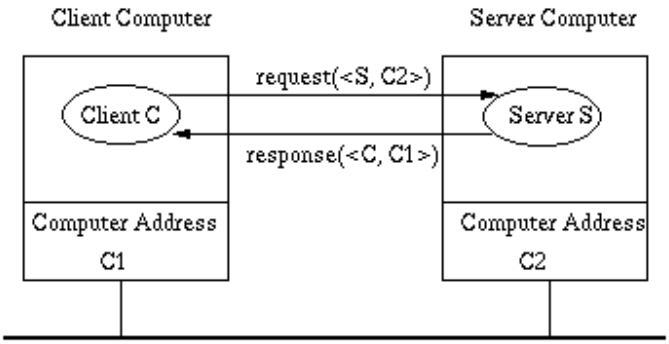

Figure 2.12. Service discovery: hardwired computer addresses

- Broadcast is used to locate servers, as illustrated in Figure 2.13.

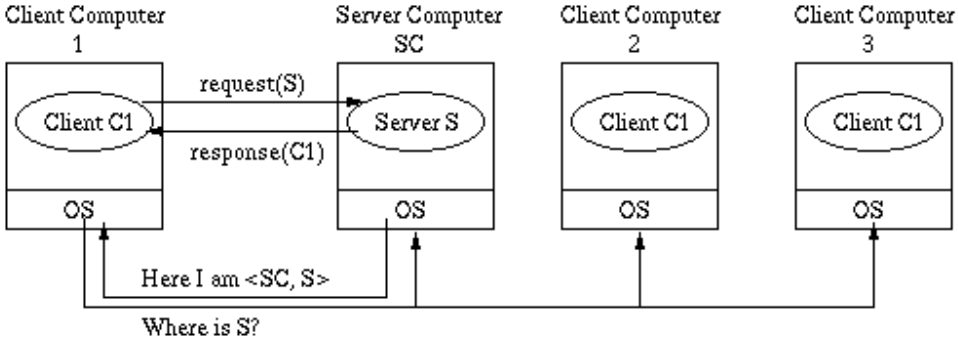

Figure 2.13. Service discovery: broadcast approach

- Server location lookup performed via a name server, as illustrated in Figure 2.14.

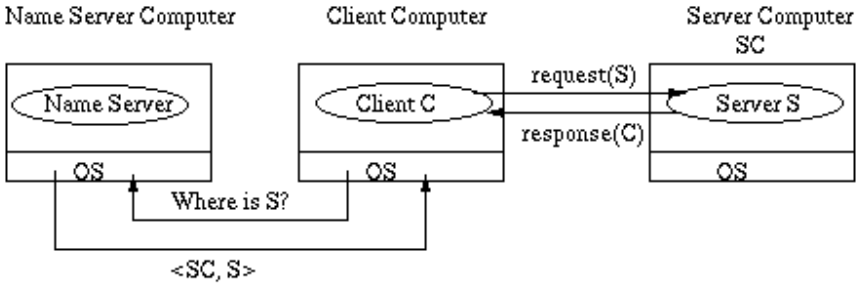

Figure 2.14. Service discovery: name server and server location lookup

- Broker-Based Location Lookup, as shown in Figure 2.15 (a) and (b).

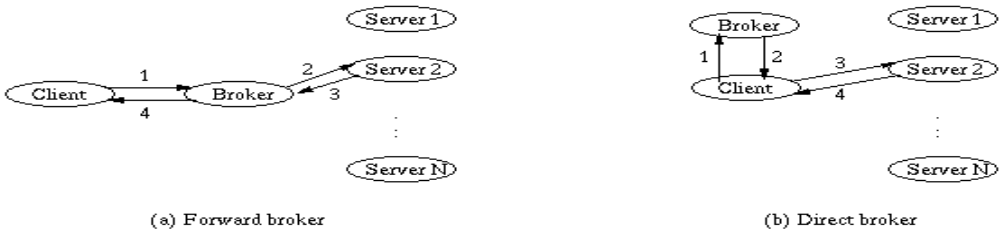

Figure 2.15. Service discovery: broker-based location lookup

Another important issue in client-server computing is the interoperability. Interoperability means the ability of two or more software components to cooperate despite differences in language, interface, and execution platform. There are two aspects of client-server interoperability. a unit of interoperation, and interoperation mechanisms. The basic unit of interoperation is a procedure. However, larger-granularity units of interoperation may be required by software components. Furthermore, preservation of temporal and functional properties may also be required.

There are two major mechanisms for interoperation:

- interface standardisation: the objective of this mechanism is to map client and server interfaces to a common representation.
- interface bridging: the objective of this mechanism is to provide a two-way mapping between a client and a server.

2.6. Communication Mechanisms between Clients and Servers

2.6.1. Factors Influence the Performance of a Communication Facility.

The client-server model is common to use in network computing systems, this implies that communication between the clients and servers cooperating by exchanging requests and responses must be fast. Furthermore, the speed of communication between remote client and server processes should not be highly different from the speed between local processes. Network computing systems based on clusters of workstations and the Internet do not share physical memory. Thus, requests and responses are sent in the form of messages.

There is a set of factors that influence the performance of a communication facility. Firstly, the speed of a communication network ranging from slow 10 Mbps to very fast Gbps. Secondly, communication protocols which span the connection-oriented protocols such as OSI and TCP which generate considerable overhead to specialised fast protocols. Thirdly, the communication paradigm, i.e, the communication model supporting cooperation between clients and servers and a communication facility support provided to deal with the cooperation.

The two issues in the communication paradigm:

- two communication patterns: one-to-one and one-to-many (group) communication

- two interprocess communication techniques: message-passing and remote procedure call (RPC).

Message passing between remote and local processes is visible to the programmer. The flow of information is unidirectional from the client to the server. However, in advanced message passing, such as structured message passing or rendezvous, information flow is bidirectional, i.e., a return message is provided in response to the initial request. Furthermore, message passing is a completely untyped technique.

The RPC technique is based on the fundamental linguistic concept known as the procedure call. The very general term remote procedure call means a type-checked mechanism that permits a language-level call on one computer to be automatically turned into a corresponding language-level call on another computer. Message passing is invisible to the programmer of RPC. RPC requires a transport protocol to support the transmission of its arguments and results. It is important to note that the term remote procedure call is sometimes used to describe just structured message passing. Remote procedure call primitives provide bidirectional flow of information.

2.6.2. Basic Form of the Message Passing

Message-oriented communication is a form of communication in which the user is explicitly aware of the message used in communication and the mechanisms used to deliver and receive messages. The basic message passing primitives are:

- *send(dest, src, buffer)*, the execution of this primitive sends the message stored in *buffer* to a server process named *dest*. The message contains a name of a client process named *src* to be used by the server to send a response back.
- *receive(client, buffer)*, the execution of this primitive causes the receiving server process to be blocked until a message arrives. The server process specifies by providing the *client* name of a process(es) from whom a message is desired, and provides a *buffer* to store an incoming message.

Figure 2.16 shows the time diagram of the execution of these two primitives.

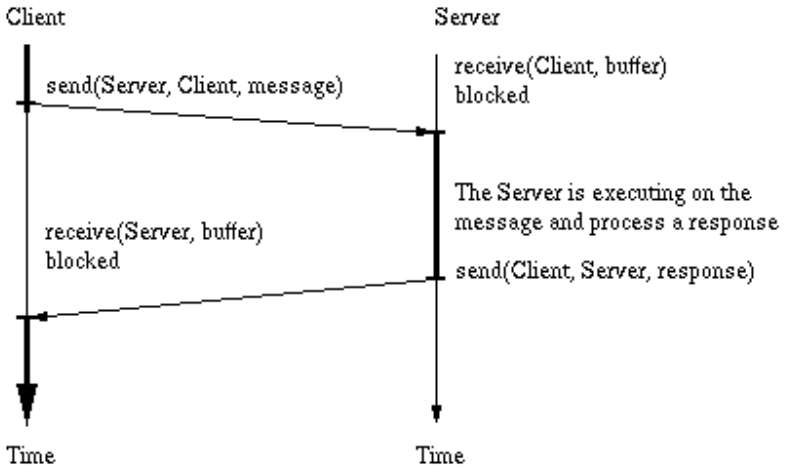

Figure 2.16. Time diagram of the execution of the message-passing primitives

2.6.3. Semantics of Message Passing

There are several different semantics of the message passing primitives:

- direct or indirect communication - ports;
- blocking versus nonblocking primitives;
- buffered versus unbuffered primitives;
- reliable versus unreliable primitives; and
- structured forms of message passing based primitives.

Direct or indirect communication

- direct communication: both the sender and the receiver have to name one another in order to communicate.
- port and indirect communication: A port is a protected kernel object into which messages may be placed by processes and from which messages can be removed, i.e., the messages are sent to and received from ports.

Blocking versus nonblocking primitives

Figure 2.17 shows the blocking and nonblocking *send* primitives.

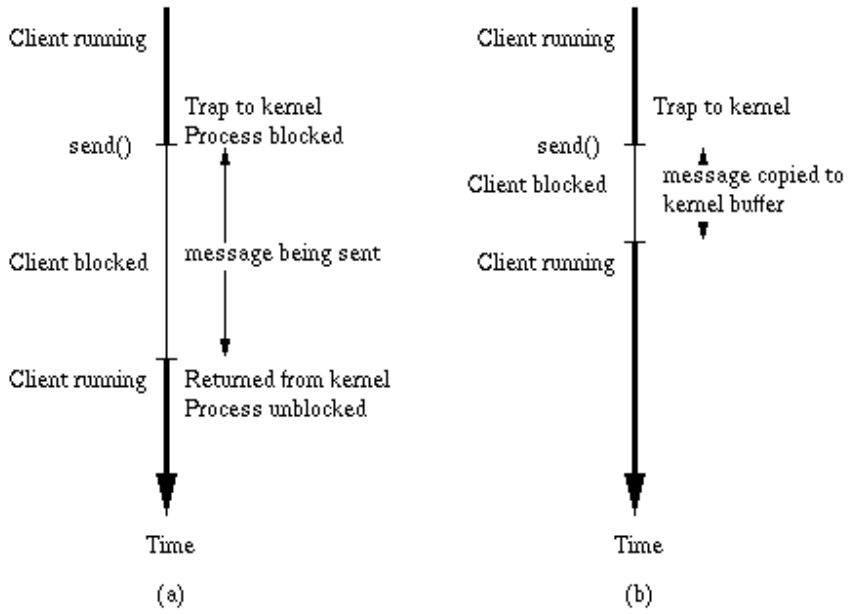

Figure 2.17. Send primitives: (a) blocking; (b) unblocking

Features of primitives

- With nonblocking primitives:
    - send returns control to the user program as soon as the message has been queued for subsequent transmission or a copy made (these alternatives are

determined by the method of cooperation between the network interface and the processor);

- when a message has been transmitted (or copied to a safe place for subsequent transmission), the program is interrupted to inform it that the buffer may be reused;
- the corresponding receive primitive signals a willingness to receive a message and provides a buffer into which the massage may be placed; and
- when a message arrives, the program is informed by interrupt.

- With blocking primitives:
  - for unreliable blocking, send does not return control to the user until the message has been sent;
  - for reliable blocking send does not return control to the user until the message has been sent and an acknowledgment received; and
  - receive does not return control until a message has been placed in the buffer.

Buffered versus unbuffered primitives

- Buffered message passing systems are more complex than unbuffered message passing based systems, since they require creation, destruction, and management of the buffers.
- Buffered message passing systems generate protection problems, and cause catastrophic event problems, when a process owning a port dies or is killed.
- Unbuffered message passing systems require synchronisation (*rendezvous*) for a message transfer to take place.

Figure 2.18 illustrates these two types of primitives.

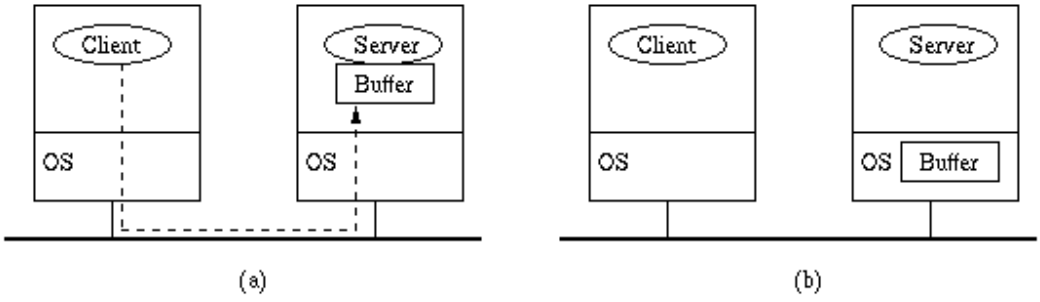

Figure 2.18. (a) Unbuffered: messages are discarded before the server issues the receive primitive; (b) buffered: messages are buffered in an OS area for a limited time.

Reliable versus unreliable primitives

- A reliable *send* primitive handles lost messages using internal retransmissions, and acknowledgments on the basis of timeouts.

- Reliable and unreliable receive differ in that the former automatically sends an acknowledgment confirming message reception, whereas the latter does not.

Reliable and unreliable primitives are contrasted in Figure 2.19.

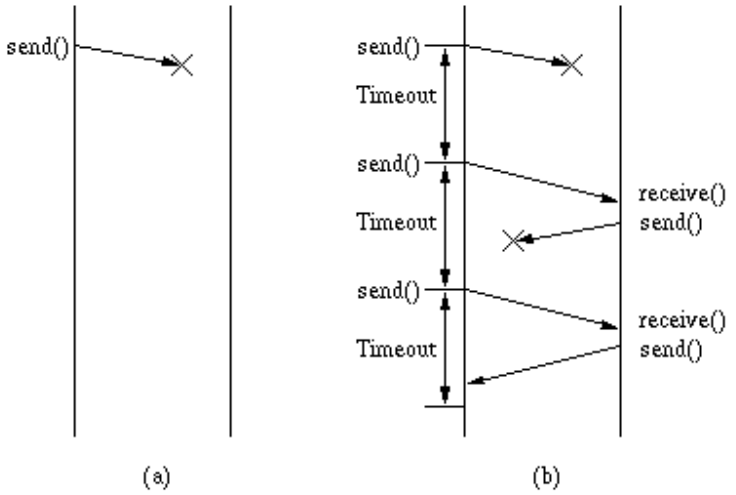

Figure 2.19. (a) Unreliable; (b) reliable}

From the message delivery point of view, message passing may have the following types of delivery semantics.

- at-least-once
- exactly-once

Figure 2.20shows these semantics.

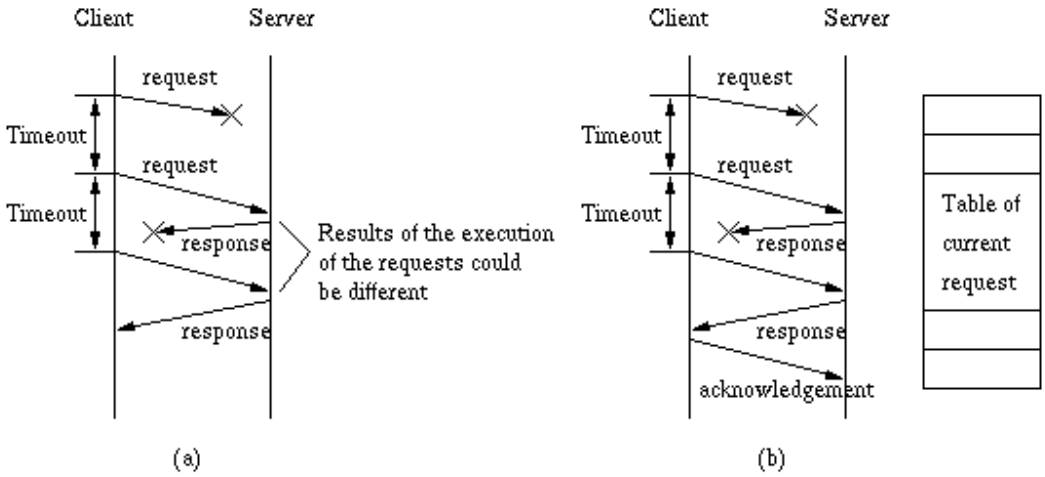

Figure 2.20. Message passing delivery semantics. (a) at-least-once; (b) exactly-once

2.6.4. Remote Procedure Calls (RPC)

RPC is based on the conventional procedure call model and the communication is transparent. The executing of a remote procedure call is

call service_name (value_args, result_args)}

Figure 2.21 illustrates an RPC example of a *read* call.

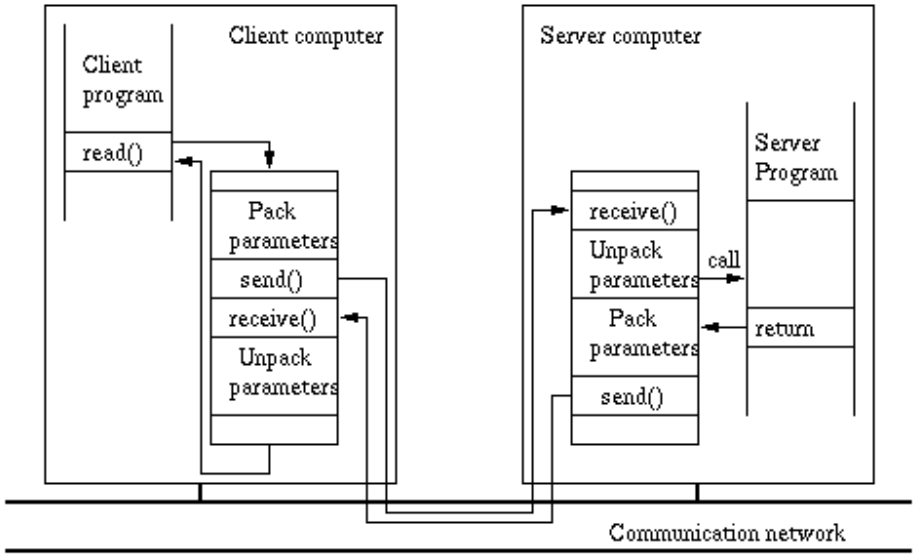

Figure 2.21. An RPC example: a *read* call

Properties of remote procedure calls.

- A transparent remote procedure call implementation must maintain the same semantics as that used for local procedure calls.
- The level of static type checking (by the compiler) applied to local procedure calls applies equally to remote procedure calls.
- All basic data types should be allowed as parameters to a remote procedure call.
- The programming language that supports RPC should provide concurrency control and exception handling.
- A programming language which uses RPC must have some means of compiling, binding, and loading distributed programs onto the network.
- RPC should provide a recovery mechanism to deal with orphans when a remote procedure call fails.

In order to provide transparency, RPC uses the approach of stubs. The interactions between the client, the client-stub, the RPC communication package (RPC Routine), the server-stub, the server involved in a remote procedure call is as Figure 2.22.

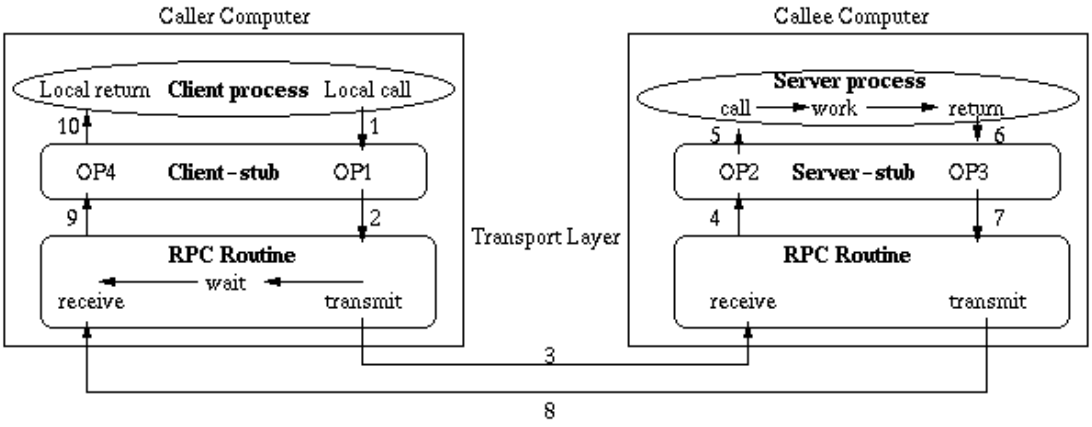

Figure 2.22. The components of the system, and their interaction for a simple RPC. OP1: marshal parameters, generate RPC id, set timer to reply; OP2: unmarshal parameters, note RPC id; OP3: marshal results, set timer for ACK of reply; OP4: unmarshal results, send ACK.

RPC Routine is responsible for retransmissions, acknowledgments, datagram routing, and protection. It can be implemented through message passing.

Parameters and results in RPCs:

- passing parameters by value
- passing parameters by reference (pointers)

Marshalling is a process performed both when sending the call (request) as well as when sending the result, in which three actions can be distinguished:

- Taking the parameters to be passed to the remote procedure and results of executing the procedure;
- Assembling these two into a form suitable for transmission among computers involved in the remote procedure call; and
- Disassembling them on arrival

Client server binding: to bind an RPC stub to the right server and remote procedure.

- Naming and addressing, e.g., through a naming server.
- Binding time
    - Compile time
    - Link time
    - Call time

Error recovery issues:

- Four events that may cause an unsuccessful return to a client's request
    - the call (request) message is lost;
    - the result (response) message is lost;
    - the server computer crashes and is restarted; and
    - the client computer crashes and is restarted.
- Three different semantics of RPC and their mechanisms can be identified to deal with problems generated by these four events

- maybe call semantics
- at-least-once call semantics
- exactly-once call semantics.

3. Java Input and Output Streams

3.1. Study Points

- Understand the principles and the hierarchy of Java streams.
- Understand the usages of various Java streams.
- Be able to construct simple stream I/O programs.
- Be able to use streams to construct simple networking programs.

Reference: (1). [JNP]: Chapter 4. (2). [Java2H]: Chapter 9. (3). [Java2U] Chapter 17. (4) [HSH]: Chapters 5-13.

3.2. Types of Streams

3.2.1. What are Streams?

Input and Output (I/O) is organised differently in Java than it is in most other languages. I/O is a core part of any networking programs since it is responsible for moving data from one system to another.

I/O in Java is built on streams. A stream is a high-level abstraction representing a Java connection to a communication channel, a file, or a memory buffer, and is the basis of most Java network communications. Basically it is a sequence of data of undetermined length. A Java stream is composed of discrete bytes. The bytes may represent chars or other kinds of data.

Almost all the classes that work directly with streams are part of the java.io package. There are two types of streams: Input streams (*java.io.InputStream*) read data and Output streams (*java.io.OutputStream*) write data. These are abstract base classes for many different subclasses with more specialized abilities. Generally speaking, streams may originate from the following situations:

- Console: *System.out* is an *OutputStream*; specifically it's a *PrintStream*. There's a corresponding *System.in* which is an *InputStream* used to read data from the console.
- Files: Data for streams can also come from files. We can use the *File* class and the *FileInputStream* and *FileOutputStream* classes to read and write data from files.
- Networking: Network connections commonly provide streams. When you connect to a web or ftp or some other kind of server, you read the data it sends from an *InputStream* connected from that server and write data onto an *OutputStream* connected to that server.
- Programs: Java programs themselves produce streams. *ByteArrayInputStreams*, *ByteArrayOutputStreams*, *StringBufferInputStreams*, *PipedInputStreams*, and *PipedOutputStreams* all use the stream metaphor to move data from one part of a Java program to another.

3.2.2 The java.io Class Hierarchy

Figure 3.1 identifies the *java.io* class hierarchy. As described in the previous section, the *InputStream*, *OutputStream*, *Reader*, and *Writer* classes are the major components of this hierarchy. Other high-level classes include the File, *FileDescriptor*, *RandomAccessFile*, *ObjectStreamClass*, and *StreamTokenizer* classes.

```
InputStream
        FilterInputStream
                BufferedInputStream
                DataInputStream
                LineNumberInputStream
                PushbackInputStream
        ByteArrayInputStream
        FileInputStream
        ObjectInputStream
                ObjectInputStream GetField(nested)
        PipedInputStream
        SequenceInputStream
        StringBufferInputStream
OutputStream
        FilterOutputStream
                BufferedOutputStream
                DataOutputStream
                PrintStream
        ByteArrayOutputStream
        FileOutputStream
        ObjectOutputStream
                ObjectOutputStream PutField(nested)
        PipedOutputStream
Reader
        BufferedReader
                LineNumberReader
        CharArrayReader
        FilterReader
                PushbackReader
        InputStreamReader
                FileReader
        PipedReader
        StringReader
Writer
        BufferedWriter
        CharArrayWriter
        FilterWriter
        OutputStreamWriter
                FileWriter
        PipedWriter
        PrintWriter
        StringWriter
File
RandomAccessFile
FileDescriptor
FilePermission
ObjectStreamClass
ObjectStreamField
SerializablePermission
StreamTokenizer
```

Figure 3.1. The classes of the java.io hierarchy

The *InputStream* and *OutputStream* classes have complementary subclasses. For example, both have subclasses for performing I/O via memory buffers, files, and pipes. The *InputStream* subclasses perform the input and the *OutputStream* classes perform the output.

Filters are objects that read from one stream and write to another, usually altering the data in some way as they pass it from one stream to another. *Filters* can be used to buffer data, read and write objects, keep track of line numbers, and perform other operations on the data they move. Filters can be combined, with one filter using the output of another as its input. You can create custom filters by combining existing filters.

The *Reader* class is similar to the *InputStream* class in that it is the root of an input class hierarchy. *Reader* supports 16-bit Unicode character input, while *InputStream* supports 8-bit byte input.

The *Writer* class is the output analog of the *Reader* class. It supports 16-bit Unicode character output.

The *File* class is used to access the files and directories of the local file system. The *FileDescriptor* class is an encapsulation of the information used by the host system to track files that are being accessed. The *RandomAccessFile* class provides the capabilities needed to directly access data contained in a file. The *ObjectStreamClass* class is used to describe classes whose objects can be written (serialized) to a stream. The *StreamTokenizer* class is used to create parsers that operate on stream data.

3.3. The InputStream and OutputStream Classes

3.3.1. The InputStream

The *InputStream* class has seven direct subclasses. The *ByteArrayInputStream* class is used to convert an array into an input stream. The *StreamBufferInputStream* class uses a *StreamBuffer* as an input stream. The *FileInputStream* class allows files to be used as input streams. The *ObjectInputStream* class is used to read primitive types and objects that have been previously written to a stream. The *PipedInputStream* class allows a pipe to be constructed between two threads and supports input through the pipe. The *SequenceInputStream* class allows two or more streams to be concatenated into a single stream. The *FilterInputStream* class is an abstract class from which other input-filtering classes are constructed.

`java.io.InputStream` is an abstract class that contains the following basic methods for reading raw bytes of data from a stream.

```
public abstract int read() throws IOException
 public int read(byte[] data) throws IOException
 public int read(byte[] data, int offset, int length) throws IOException
 public long skip(long n) throws IOException
 public int available() throws IOException
 public void close() throws IOException
 public synchronized void mark(int readlimit)
 public synchronized void reset() throws IOException
 public boolean markSupported()
```

Notice that almost all these methods can throw an `IOException`. This is true of pretty much anything to do with input and output.

3.3.2. The OutputStream

The *OutputStream* class hierarchy consists of five direct subclasses. The *ByteArrayOutputStream, FileOutputStream, ObjectOutputStream,* and *PipedOutputStream* classes are the output complements to the *ByteArrayInputStream, FileInputStream, ObjectInputStream,* and *PipedInputStream* classes. The *FilterOutputStream* class provides subclasses that complement the *FilterInputStream* classes.

The *BufferedOutputStream* class is the output analog to the *BufferedInputStream* class. It buffers output so that output bytes can be written to devices in larger groups. The *DataOutputStream* class implements the *DataOutput* interace. This interface complements the *DataInput* interface. It provides methods that write objects and primitive data types to streams so that they can be read by the *DataInput* interface methods. The *PrintStream* class provides the familiar `print()` and `println()` methods used in most of the sample programs that you've developed so far in this book. It provides a number of overloaded methods that simplify data output.

The `java.io.OutputStream` class sends raw bytes of data to a target such as the console or a network server. Like *InputStream*, *OutputStream* is an abstract class.

```
public abstract void write(int b) throws IOException
 public void write(byte[] data) throws IOException
 public void write(byte[] data, int offset, int length) throws IOException
 public void flush() throws IOException
 public void close() throws IOException
```

The `write()` methods send raw bytes of data to whomever is listening to this stream.

Sometimes output streams are buffered by the operating system for performance. In other words, rather than writing each byte as it's written the bytes are accumulated in a buffer ranging from several bytes to several thousand bytes. Then, when the buffer fills up, all the data is written at once. The `flush()` method forces the data to be written whether or not the buffer is full.

3.3.3. Examples

Here's a simple program (`Echo.java`) that echoes back what the user types at the command line using the basic `read()` method of the `InputStream` class:

```
import java.io.*;

public class Echo {
  public static void main(String[] args) {
    echo(System.in);
  }

  public static void echo(InputStream in) {
    try {
      while (true) {
        // Notice that although a byte is read, an int
        // with value between 0 and 255 is returned.
```

```
        // Then this is converted to an ISO Latin-1 char
        // in the same range before being printed.
        int i = in.read();
        // -1 is returned to indicate the end of stream
        if (i == -1) break;
        // without the cast a numeric string like "65"
        // would be printed instead of the character "A"
        char c = (char) i;
        System.out.print(c);
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    System.out.println();
  }
}
```

The following program (`EfficientEcho.java`) is a more efficient version of `Echo` that uses `available()` to test how many bytes are ready to be read, creates an array of exactly that size, reads the bytes into the array, then converts the array to a `String` and prints the `String`:

```
import java.io.*;
public class EfficientEcho {

  public static void main(String[] args) {
    echo(System.in);
  }
  public static void echo(InputStream in) {
    try {
      while (true) {
        int n = in.available();
        if (n > 0) {
          byte[] b = new byte[n];
          int result = in.read(b);
          if (result == -1) break;
          String s = new String(b);
          System.out.print(s);
        } // end if
      } // end while
    } // end try
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

The following *ByteArrayIOApp*.java program creates a *ByteArrayOutputStream* object, *outStream*, and an array, s, that contains the text "This is a test." to be written to the stream. Each s character is written, one at a time, to *outStream*. The contents of `outstream` are then printed, along with the number of bytes written.

```
import java.lang.System;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
public class ByteArrayIOApp {
```

```
 public static void main(String args[]) throws IOException {
  // Create ByteArrayOutputStream object
  ByteArrayOutputStream outStream = new ByteArrayOutputStream();
String s = "This is a test.";
  // Write output to stream
  for(int i=0;i<s.length();++i)
   outStream.write(s.charAt(i));
  System.out.println("outstream: "+outStream);
  System.out.println("size: "+outStream.size());
  ByteArrayInputStream inStream;
  inStream = new ByteArrayInputStream(outStream.toByteArray());
  // Determine how many input bytes are available
  int inBytes = inStream.available();
  System.out.println("inStream has "+inBytes+" available bytes");
  byte inBuf[] = new byte[inBytes];
  // Read input into a byte array
  int bytesRead = inStream.read(inBuf,0,inBytes);
  System.out.println(bytesRead+" bytes were read");
  System.out.println("They are: "+new String(inBuf));
 }
}
```

A *ByteArrayInputStream* object, *inStream*, is created by invoking the `toByteArray()` method of *outStream* to create a byte array that is used as an argument to the *ByteArrayInputStream* constructor. The `available()` method is used to determine the number of available input bytes stored in the buffer. This number is stored as *inBytes* and is used to allocate a byte array to store the data that is read. The `read()` method is invoked for *inStream* to read *inBytes* worth of data. The actual number of bytes read is stored in *bytesRead*. This number is displayed, followed on the next line by the bytes that were read from *inStream*, as follows:

```
outstream: This is a test.
size: 15
inStream has 15 available bytes
15 bytes were read
They are: This is a test.
```

3.4. File I/O Streams

3.4.1. Basics of File I/O

Java supports stream-based file input and output through the File, *FileDescriptor*, *FileInputStream*, and *FileOutputStream* classes. It supports direct or random access I/O using the *File*, *FileDescriptor*, and *RandomAccessFile* classes. Random access I/O is covered later in this chapter. The *FileReader* and *FileWriter* classes support Unicode-based file I/O.

The File class provides access to file and directory objects and supports a number of operations on files and directories. The *FileDescriptor* class encapsulates the information used by the host system to track files that are being accessed. The *FileInputStream* and *FileOutputStream* classes provide the capability to read and write to file streams.

The `java.io.FileInputStream` class represents an *InputStream* that reads bytes from a file. It has the following public methods:

```
public FileInputStream(String name) throws FileNotFoundException
 public FileInputStream(File file) throws FileNotFoundException
 public FileInputStream(FileDescriptor fdObj)
 public native int read() throws IOException
 public int read(byte[] data) throws IOException
 public int read(byte[] data, int offset, int length) throws IOException
 public native long skip(long n) throws IOException
 public native int available() throws IOException
 public native void close() throws IOException
 public final FileDescriptor getFD() throws IOException
```

Except for the constructors and `getFD()`, these methods merely override the methods of the same name in `java.io.InputStream`. You use them exactly like you use those methods, only you'll end up reading data from a file.

The `java.io.FileOutputStream` class represents an *OutputStream* that writes bytes to a file. It has the following public methods:

```
public FileOutputStream(String name) throws IOException
 public FileOutputStream(String name, boolean append) throws IOException
 public FileOutputStream(File file) throws IOException
 public FileOutputStream(FileDescriptor fdObj)
 public native void write(int b) throws IOException
 public void write(byte[] data) throws IOException
 public void write(byte[] data, int offset, int length) throws IOException
 public native void close() throws IOException
 public final FileDescriptor getFD() throws IOException
```

Except for the constructors and `getFD()`, these methods merely override the methods of the same name in `java.io.OutputStream`. You use them exactly like you use those methods, only the output is written into a file.

3.4.2. Examples

The following example is a program to copy a file, and also defines a static copy() method that other programs can use to copy files.

```
import java.io.*;

public class FileCopy {
  /** The main() method of the standalone program.  Calls copy(). */
  public static void main(String[] args) {
    if (args.length != 2)    // Check arguments
      System.err.println("Usage: java FileCopy <source file> <destination>");
    else {
      // Call copy() to do the copy, and display any error messages it throws.
      try { copy(args[0], args[1]); }
      catch (IOException e) { System.err.println(e.getMessage()); }
    }
  }

  /**
   * The static method that actually performs the file copy.
   * Before copying the file, however, it performs a lot of tests to make
   * sure everything is as it should be.
```

```
   */
  public static void copy(String from_name, String to_name) throws IOException{
    File from_file = new File(from_name);  // Get File objects from Strings
    File to_file = new File(to_name);

    // First make sure the source file exists, is a file, and is readable.
    if (!from_file.exists())
      abort("FileCopy: no such source file: " + from_name);
    if (!from_file.isFile())
      abort("FileCopy: can't copy directory: " + from_name);
    if (!from_file.canRead())
      abort("FileCopy: source file is unreadable: " + from_name);

    // If the destination is a directory, use the source file name
    // as the destination file name
    if (to_file.isDirectory())
      to_file = new File(to_file, from_file.getName());

    // If the destination exists, make sure it is a writeable file
    // and ask before overwriting it.  If the destination doesn't
    // exist, make sure the directory exists and is writeable.
    if (to_file.exists()) {
      if (!to_file.canWrite())
        abort("FileCopy: destination file is unwriteable: " + to_name);
      // Ask whether to overwrite it
      System.out.print("Overwrite existing file " + to_name + "? (Y/N): ");
      System.out.flush();
      // Get the user's response.
      BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
      String response = in.readLine();
      // Check the response.  If not a Yes, abort the copy.
      if (!response.equals("Y") && !response.equals("y"))
        abort("FileCopy: existing file was not overwritten.");
    }
    else {
      // if file doesn't exist, check if directory exists and is writeable.
      // If getParent() returns null, then the directory is the current dir.
      // so look up the user.dir system property to find out what that is.
      String parent = to_file.getParent();  // Get the destination directory
      if (parent == null) parent = System.getProperty("user.dir"); // or CWD
      File dir = new File(parent);          // Convert it to a file.
      if (!dir.exists())
        abort("FileCopy: destination directory doesn't exist: " + parent);
      if (dir.isFile())
        abort("FileCopy: destination is not a directory: " + parent);
      if (!dir.canWrite())
        abort("FileCopy: destination directory is unwriteable: " + parent);
    }

    // If we've gotten this far, then everything is okay.
    // So we copy the file, a buffer of bytes at a time.
    FileInputStream from = null;  // Stream to read from source
    FileOutputStream to = null;   // Stream to write to destination
    try {
      from = new FileInputStream(from_file);  // Create input stream
      to = new FileOutputStream(to_file);     // Create output stream
      byte[] buffer = new byte[4096];         // A buffer to hold file contents
      int bytes_read;                         // How many bytes in buffer
      // Read a chunk of bytes into the buffer, then write them out,
      // looping until we reach the end of the file (when read() returns -1).
      // Note the combination of assignment and comparison in this while
      // loop.  This is a common I/O programming idiom.
      while((bytes_read = from.read(buffer)) != -1) // Read bytes until EOF
```

```
        to.write(buffer, 0, bytes_read);            //   write bytes
    }
    // Always close the streams, even if exceptions were thrown
    finally {
      if (from != null) try { from.close(); } catch (IOException e) { ; }
      if (to != null) try { to.close(); } catch (IOException e) { ; }
    }
  }

  /** A convenience method to throw an exception */
  private static void abort(String msg) throws IOException {
    throw new IOException(msg);
  }
}
```

The following program illustrates the use of the *FileInputStream, FileOutputStream*, and *File* classes. It writes a string to an output file and then reads the file to verify that the output was written correctly. The file used for the I/O is then deleted.

```
import java.lang.System;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.File;
import java.io.IOException;
public class FileIOApp {
 public static void main(String args[]) throws IOException {
  // Create output file test.txt
  FileOutputStream outStream = new FileOutputStream("test.txt");
  String s = "This is a test.";
  for(int i=0;i<s.length();++i)
outStream.write(s.charAt(i));
  outStream.close();
  // Open test.txt for input
  FileInputStream inStream = new FileInputStream("test.txt");
  int inBytes = inStream.available();
  System.out.println("inStream has "+inBytes+" available bytes");
  byte inBuf[] = new byte[inBytes];
  int bytesRead = inStream.read(inBuf,0,inBytes);
  System.out.println(bytesRead+" bytes were read");
  System.out.println("They are: "+new String(inBuf));
  inStream.close();
  File f = new File("test.txt");
  f.delete();
 }
}
```

The *FileOutputStream* constructor creates an output stream on the file `test.txt`. The file is automatically created in the current working directory. It then writes the string "This is a test." to the output file stream. Note the similarity between this program and the previous one. The power of streams is that the same methods can be used no matter what type of stream is being used.

The output stream is closed to make sure that all the data is written to the file. The file is then reopened as an input file by creating an object of class *FileInputStream*. The same methods used in the *ByteArrayIOApp* program are used to determine the number of

available bytes in the file and read these bytes into a byte array. The number of bytes read is displayed along with the characters corresponding to those bytes.

The input stream is closed and then a File object is created to provide access to the file. The File object is used to delete the file using the `delete()` method. The program's output follows:

```
inStream has 15 available bytes
15 bytes were read
They are: This is a test.
```

3.5. The SequenceInputStream, Buffered Stream and Data Stream

3.5.1. The SequenceInputStream

The *SequenceInputStream* class is used to combine two or more input streams into a single input stream. The input streams are concatenated, which allows the individual streams to be treated as a single, logical stream. The *SequenceInputStream* class does not introduce any new access methods. Its power is derived from the two constructors that it provides. One constructor takes two *InputStream* objects as arguments. The other takes an Enumeration of *InputStream* objects. It provides methods for dealing with a sequence of related objects.

The following program of `SequenceIOApp.java` reads the two Java source files, `ByteArrayIOApp.java` and `FileIOApp.java`, as a single file courtesy of the *SequenceInputStream* class.

```
import java.lang.System;
import java.io.FileInputStream;
import java.io.SequenceInputStream;
import java.io.IOException;
public class SequenceIOApp {
 public static void main(String args[]) throws IOException {
  SequenceInputStream inStream;
  FileInputStream f1 = new FileInputStream("ByteArrayIOApp.java");
  FileInputStream f2 = new FileInputStream("FileIOApp.java");
  // Concatentate two files into a single input stream
  inStream = new SequenceInputStream(f1,f2);
  boolean eof = false;
  int byteCount = 0;
  while (!eof) {
   int c = inStream.read();
   if(c == -1) eof = true;
   else{
    System.out.print((char) c);
    ++byteCount;
   }
  }
  System.out.println(byteCount+" bytes were read");
  inStream.close();
  f1.close();
  f2.close();
 }
```

```
}
```

The program creates two objects of class *FileInputStream* for the files `ByteArrayIOApp.java` and `FileIOApp.java`. The *SequenceInputClass* constructor is used to construct a single input stream from the two *FileInputStream* objects. The program then uses a while loop to read all bytes in the combined file and display them to the console window. The loop stops when the end of the combined file is encountered. This is signaled when the `read()` method returns -1. The streams are closed after the combined files have been read.

The *SequenceIOApp* program displays the combined contents of the two source files, followed by a line identifying the number of bytes that were read.

3.5.2. Buffered I/O

Buffered input and output is used to temporarily cache data that is read from or written to a stream. This allows programs to read and write small amounts of data without adversely affecting system performance. When buffered input is performed, a large number of bytes are read at a single time and stored in an input buffer. When a program reads from the input stream, the input bytes are read from the input buffer. Several reads may be performed before the buffer needs to be refilled. Input buffering is used to speed up overall stream input processing.

Output buffering is performed in a similar manner to input buffering. When a program writes to a stream, the output data is stored in an output buffer until the buffer becomes full or the output stream is flushed. Only then is the buffered output actually forwarded to the output stream's destination.

Java implements buffered I/O as filters. The filters maintain and operate the buffer that sits between the program and the source or destination of a buffered stream.

The `java.io.BufferedInputStream` and `java.io.BufferedOutputStream` classes buffer reads and writes by first storing the in a buffer (an internal array of bytes). Then the program reads bytes from the stream without calling the underlying native method until the buffer is empty. The data is read from or written into the buffer in blocks; subsequent accesses go straight to the buffer.

The only real difference to the programmer between a regular stream and a buffered stream are the constructors:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int size)
```

The `size` argument is the number of bytes in the buffer. If a size isn't specified, a 512 byte buffer is used.

3.5.3. Data Streams

The `java.io.DataInputStream` and `java.io.DataOutputStream` classes read and write primitive Java data types and Strings in a machine-independent way. Generally you use a `DataInputStream` to read data written by a `DataOutputStream`. This format uses IEEE 754 for floating point data, big-endian format for integer data, and a modified UTF-8 for Unicode data.

`DataOutputStream` declares these methods:

```
public DataOutputStream(OutputStream out)
 public synchronized void write(int b) throws IOException
 public synchronized void write(byte[] data, int offset, int length)
  throws IOException
 public final void writeBoolean(boolean b) throws IOException
 public final void writeByte(int b) throws IOException
 public final void writeShort(int s) throws IOException
 public final void writeChar(int c) throws IOException
 public final void writeInt(int i) throws IOException
 public final void writeFloat(float f) throws IOException
 public final void writeDouble(double d) throws IOException
 public final void writeBytes(String s) throws IOException
 public final void writeChars(String s) throws IOException
 public final void writeUTF(String s) throws IOException
 public final int size()
 public void flush() throws IOException
```

3.5.2. Examples

The following `BufferedIOApp.java` program builds on the *SequenceIOApp* example that was previously presented. It performs buffering on the *SequenceInputStream* object used to combine the input from two separate files. It also performs buffering on program output so that characters do not need to be displayed to the console window a single character at a time.

```
import java.lang.System;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.SequenceInputStream;
import java.io.IOException;
public class BufferedIOApp {
 public static void main(String args[]) throws IOException {
  SequenceInputStream f3;
  FileInputStream f1 = new FileInputStream("ByteArrayIOApp.java");
  FileInputStream f2 = new FileInputStream("FileIOApp.java");
  f3 = new SequenceInputStream(f1,f2);
  // Create the buffered input and output streams
  BufferedInputStream inStream = new BufferedInputStream(f3);
  BufferedOutputStream outStream = new BufferedOutputStream(System.out);
  inStream.skip(500);
  boolean eof = false;
  int byteCount = 0;
  while (!eof) {
   int c = inStream.read();
   if(c == -1) eof = true;
   else{
```

```
    outStream.write((char) c);
    ++byteCount;
   }
  }
  String bytesRead = String.valueOf(byteCount);
  bytesRead+=" bytes were read\n";
  outStream.write(bytesRead.getBytes(),0,bytesRead.length());
  inStream.close();
  outStream.close();
  f1.close();
  f2.close();
 }
}
```

The program begins by creating two objects of *FileInputStream* and combining them into a single input stream using the *SequenceInputStream* constructor. It then uses this stream to create an object of *BufferedInputStream* using the default buffer size.

A *BufferedOutputStream* object is created using the System.out output stream and a default buffer size. The `skip()` method is used to skip over 500 bytes of the input stream. This is done for two reasons: to illustrate the use of the `skip()` method and to cut down on the size of the program output. The rest of the input is read and printed, as in the previous example.

The program output is similar to that of the preceding example.

The following program of `DataIOApp.java` shows how *DataInputStream* and *DataOutputStream* can be used to easily read and write a variety of values using streams.

```
import java.lang.System;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.File;
import java.io.IOException;
public class DataIOApp {
 public static void main(String args[]) throws IOException {
  File file = new File("test.txt");
  FileOutputStream outFile = new FileOutputStream(file);
  DataOutputStream outStream = new DataOutputStream(outFile);
  // Write various data types to the output stream
  outStream.writeBoolean(true);
  outStream.writeInt(123456);
  outStream.writeChar('j');
  outStream.writeDouble(1234.56);
  System.out.println(outStream.size()+" bytes were written");
  outStream.close();
  outFile.close();
  FileInputStream inFile = new FileInputStream(file);
  DataInputStream inStream = new DataInputStream(inFile);
  System.out.println(inStream.readBoolean());
  System.out.println(inStream.readInt());
  System.out.println(inStream.readChar());
  System.out.println(inStream.readDouble());
  inStream.close();
  inFile.close();
  file.delete();
 }
}
```

The program creates an object of class *File* that is used to access the `test.txt` file. This object is used to create an instance of class *FileOutputStream* that is assigned to the outFile variable. An object of class *DataOutputStream* is then constructed as a filter for the *FileOutputStream* object.

The `writeBoolean(), writeChar(), writeInt(),` and `writeDouble()` methods of *DataOutputStream* are used to write examples of primitive data types to the filtered output stream. The number of bytes written to the output stream is determined by the `size()` method and displayed to the console window. The output streams are then closed.

The *File* object, created at the beginning of the program, is then used to create an object of class *FileInputStream*. The output stream is then filtered by creating an object of *DataInputStream.*

The primitive data types that were written to the output file in the beginning of the program are now read from the filtered input stream and displayed to the console window.

The program's output shows that the data values were successfully written and read using the data I/O filters:

```
15 bytes were written
true
123456
j
1234.56
```

3.6. Readers and Writers

3.6.1. Basics

The `java.io.Reader` and `java.io.Writer` classes are abstract superclasses for classes that read and write character based data. The subclasses are notable for handling the conversion between different character sets.

Input and output streams are fundamentally byte based. However readers and writers are based on characters, which can have varying widths depending on the character set being used. For example, ASCII and ISO Latin-1 use one byte characters. Unicode uses two byte characters. UTF-8 uses characters of varying width between one and three bytes. Readers and writers know how to handle all these character sets and many more seamlessly.

The methods of the `java.io.Reader` class are deliberately similar to the methods of the `java.io.InputStream` class. However rather than working with bytes, they work with chars. All the `read()` methods block until some input is available, an I/O error occurs, or the end of the stream is reached.

The methods of the `java.io.Writer` class are deliberately similar to the methods of the `java.io.OutputStream` class. However rather than working with bytes, they work with

chars. Like OutputStreams, Writers may be buffered. To force the write to take place, call `flush()`.

The `java.io.InputStreamReader` class serves as a bridge between byte streams and character streams: It reads bytes from the input stream and translates them into characters according to a specified character encoding.

The `java.io.FileWriter` class writes text files using the platform's default character encoding and the buffer size. If you need to change these values, construct an `OutputStreamReader` on a `FileOutputStream` instead.

The `java.io.BufferedReader` class is a subclass of `java.io.Reader` that you chain to another `Reader` class to buffer characters. This allows more efficient reading of characters and lines.

The `BufferedReader` is also notable for its `readLine()` method that allows you to read text a line at a time.

Each time you read from an unbuffered `Reader`, there's a matching read from the underlying input stream. Therefore it's a good idea to wrap a `BufferedReader` around each `Reader` whose `read()` operations are expensive, such as a `FileReader`. For example,

```
BufferedReader br = new BufferedReader(new FileReader("37.html"));
```

3.6.2. An Example

The following example reads a text file, line by line, and prints it to `System.out`:

```
// Implement the Unix cat utility in Java
import java.io.*;
class cat  {
  public static void main (String args[]) {
    String thisLine;
    //Loop across the arguments
   for (int i=0; i < args.length; i++) {
      //Open the file for reading
     try {
       BufferedReader br = new BufferedReader(new FileReader(args[i]));
       while ((thisLine = br.readLine()) != null) { // while loop begins here
         System.out.println(thisLine);
       } // end while
     } // end try
     catch (IOException e) {
       System.err.println("Error: " + e);
     }
  } // end for
} // end main
}
```

The `java.io.BufferedWriter` class is a subclass of `java.io.Writer` that you chain to another `Writer` class to buffer characters. This allows more efficient writing of text.

Each time you write to an unbuffered `Writer`, there's a matching write to the underlying output stream. Therefore it's a good idea to wrap a `BufferedWriter` around each `Writer`

whose `write()` operations are expensive and that does not require immediate response, such as a `FileWriter`. For example,

```
BufferedWriter bw = new BufferedWriter(new FileWriter("37.html"));
```

3.7. Piped I/O and Character Array and String I/O

3.7.1. Piped I/O

Piped I/O provides the capability for threads to communicate via streams. A thread sends data to another thread by creating an object of *PipedOutputStream* that it connects to an object of *PipedInputStream*. The output data written by one thread is read by another thread using the *PipedInputStream* object.

The process of connecting piped input and output threads is symmetric. An object of class *PipedInputThread* can also be connected to an existing object of class *PipedOutputThread*.

Java automatically performs synchronization with respect to piped input and output streams. The thread that reads from an input pipe does not have to worry about any conflicts with tasks that are being written to the corresponding output stream thread.

Both *PipedInputStream* and *PipedOutputStream* override the standard I/O methods of *InputStream* and *OutputStream*. The only new method provided by these classes is the `connect()` method. Both classes provide the capability to connect a piped stream when it is constructed by passing the argument of the piped stream to which it is to be connected as an argument to the constructor.

3.7.2. An Example of Piped I/O

The `PipedIOApp.java` program creates two threads of execution, named Producer and Consumer, that communicate using connected objects of classes *PipedOutputStream* and *PipedInputStream*. Producer sends the message "This is a test." to Consumer one character at a time, and Consumer reads the message in the same manner. Producer displays its name and any characters that it writes to the console window. Consumer reads the message and displays its name and the characters it reads to the console window.

```
import java.lang.Thread;
import java.lang.System;
import java.lang.InterruptedException;
import java.lang.Runnable;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import java.io.IOException;
class PipedIOApp {
 public static void main(String args[]) {
  Thread thread1 = new Thread(new PipeOutput("Producer"));
  Thread thread2 = new Thread(new PipeInput("Consumer"));
  thread1.start();
  thread2.start();
```

```
  boolean thread1IsAlive = true;
  boolean thread2IsAlive = true;
  do {
   if(thread1IsAlive && !thread1.isAlive()){
    thread1IsAlive = false;
    System.out.println("Thread 1 is dead.");
   }
   if(thread2IsAlive && !thread2.isAlive()){
    thread2IsAlive = false;
    System.out.println("Thread 2 is dead.");
   }
  }while(thread1IsAlive || thread2IsAlive);
 }
}
class PipeIO {
 static PipedOutputStream outputPipe = new PipedOutputStream();
 static PipedInputStream inputPipe = new PipedInputStream();
 static {
  try {
   // Connect input and output pipes
   outputPipe.connect(inputPipe);
  }catch (IOException ex) {
   System.out.println("IOException in static initializer");
  }
 }
 String name;
 public PipeIO(String id) {
  name = id;
 }
}
class PipeOutput extends PipeIO implements Runnable {
 public PipeOutput(String id) {
  super(id);
 }
 public void run() {
  String s = "This is a test.";
  try {
   for(int i=0;i<s.length();++i){
    outputPipe.write(s.charAt(i));
    System.out.println(name+" wrote "+s.charAt(i));
   }
   outputPipe.write(`!');
  } catch(IOException ex) {
   System.out.println("IOException in PipeOutput");
  }
 }
}
class PipeInput extends PipeIO implements Runnable {
 public PipeInput(String id) {
  super(id);
 }
 public void run() {
  boolean eof = false;
  try {
   while (!eof) {
    int inChar = inputPipe.read();
    if(inChar != -1) {
     char ch = (char) inChar;
     if(ch=='!'){
      eof=true;
      break;
     }else System.out.println(name+" read "+ch);
    }
```

```
   }
  } catch(IOException ex) {
   System.out.println("IOException in PipeOutput");
  }
 }
}
```

The `main()` method creates the two Producer and Consumer threads as objects of classes *PipeOutput* and *PipeInput*. These classes are subclasses of *PipeIO* that implement the *Runnable* interface. The `main()` method starts both threads and then loops, checking for their death.

The *PipeIO* class is the superclass of the *PipeOutput* and *PipeInput* classes. It contains the static variables, *outputPipe* and *inputPipe*, that are used for interthread communication. These variables are assigned objects of classes *PipedOutputStream* and *PipeInputStream*. The static initializer is used to connect *outputPipe* with *inputPipe* using the `connect()` method. The *PipeIO* constructor provides the capability to maintain the name of its instances. This is used by the *PipeInput* and *PipeOutput* classes to store thread names.

The *PipeOutput* class extends *PipeIO* and implements the *Runnable* interface, making it eligible to be executed as a separate thread. The required `run()` method performs all thread processing. It loops to write the test message one character at a time to the *outputPipe*. It also displays its name and the characters that it writes to the console window. The ! character is used to signal the end of the message transmission. Notice that *IOException* is handled within the thread rather than being identified in the throws clause of the run() method. In order for `run()` to properly implement the *Runnable* interface, it cannot throw any exceptions.

The *PipeInput* class also extends *PipeIO* and implements the *Runnable* interface. It simply loops and reads a character at a time from *inputPipe*, displaying its name and the characters that it reads to the console window. It also handles *IOException* in order to avoid having to identify the exception in its throws clause.

The output of *PipeIOApp* shows the time sequencing of the thread input and output taking place using the connected pipe I/O streams. The output generated by running the program on your computer will probably differ because of differences in your computer's execution speed and I/O performance.

3.7.3. Character Array and String I/O

The *CharArrayReader* and *CharArrayWriter* classes are similar to the *ByteArrayInputStream* and *ByteArrayOutputStream* classes in that they support I/O from memory buffers. The difference between these classes is that *CharArrayReader* and *CharArrayWriter* support 16-bit character I/O, and *ByteArrayInputStream* and *ByteArrayOutputStream* support 8-bit byte array I/O.

The *StringReader* class provides the capability to read character input from a string. Like *CharArrayReader*, it does not add any additional methods to those provided by *Reader*. The *StringWriter* class is used to write character output to a *StringBuffer* object. It adds

the `getBuffer()` and `toString()` methods. The `getBuffer()` method returns the *StringBuffer* object corresponding to the output buffer. The `toString()` method returns a *String* copy of the output buffer.

The following `CharArrayIOApp.java` program is based on the `ByteArrayIOApp.java` program (see Section 3.3.3) introduced earlier. It writes the string "This is a test." one character at a time to a *CharArrayWriter* object. It then converts the output buffer to a *CharArrayReader* object. Each character of the input buffer is read and appended to a *StringBuffer* object. The *StringBuffer* object is then converted to a *String* object. The number of characters read and the *String* object are then displayed.

```
import java.lang.System;
import java.io.CharArrayReader;
import java.io.CharArrayWriter;
import java.io.IOException;
public class CharArrayIOApp {
 public static void main(String args[]) throws IOException {
  CharArrayWriter outStream = new CharArrayWriter();
  String s = "This is a test.";
  for(int i=0;i<s.length();++i)
   outStream.write(s.charAt(i));
  System.out.println("outstream: "+outStream);
  System.out.println("size: "+outStream.size());
  CharArrayReader inStream;
  inStream = new CharArrayReader(outStream.toCharArray());
  int ch=0;
  StringBuffer sb = new StringBuffer("");
  while((ch = inStream.read()) != -1)
   sb.append((char) ch);
  s = sb.toString();
  System.out.println(s.length()+" characters were read");
  System.out.println("They are: "+s);
 }
}
```

The program output follows:

```
outstream: This is a test.
size: 15
15 characters were read
They are: This is a test.
```

The following `StringIOApp.java` program is similar to `CharArrayIOApp.java` program. It writes output to a *StringBuffer* instead of a character array. It produces the same output as `CharArrayIOApp.java`.

```
import java.lang.System;
import java.io.StringReader;
import java.io.StringWriter;
import java.io.IOException;
public class StringIOApp {
 public static void main(String args[]) throws IOException {
  StringWriter outStream = new StringWriter();
```

```
  String s = "This is a test.";
  for(int i=0;i<s.length();++i)
   outStream.write(s.charAt(i));
  System.out.println("outstream: "+outStream);
  System.out.println("size: "+outStream.toString().length());
  StringReader inStream;
  inStream = new StringReader(outStream.toString());
  int ch=0;
  StringBuffer sb = new StringBuffer("");
  while((ch = inStream.read()) != -1)
   sb.append((char) ch);
  s = sb.toString();
  System.out.println(s.length()+" characters were read");
  System.out.println("They are: "+s);
 }
}
```

4. Connection-Oriented Communication in Java

4.1. Study Points

- Understand the difference of connection-oriented and connectionless communication.
- Understand the basic concepts of connection-oriented communication using Java.
- Be familiar with the Socket and ServerSocket classes.
- Be able to write Java program to complete simple TCP communication tasks.

Reference: (1). [JNP]: Chapters 10 and 11. (2). [Java2H] Chapter 10. (3). [Java2U]: Chapter 30.

4.2. Introduction

4.2.1. Connection-Oriented Versus Connectionless Communication

Transport protocols are used to deliver information from one port to another and thereby enable communication between application programs. They use either a connection-oriented or connectionless method of communication. TCP is a connection-oriented protocol, and UDP is a connectionless transport protocol.

The TCP connection-oriented protocol establishes a communication link between a source port/IP address and a destination port/IP address. The ports are bound together via this link until the connection is terminated and the link is broken. An example of a connection-oriented protocol is a telephone conversation. A telephone connection is established, communication takes place, and then the connection is terminated.

The reliability of the communication between the source and destination programs is ensured through error-detection and error-correction mechanisms that are implemented within TCP. TCP implements the connection as a stream of bytes from source to destination. This feature allows the use of the stream I/O classes provided by java.io.

The UDP connectionless protocol differs from the TCP connection-oriented protocol in that it does not establish a link for the duration of the connection. An example of a connectionless protocol is postal mail. To mail something, you just write down a destination address (and an optional return address) on the envelope of the item you're sending and drop it into a mailbox. When using UDP, an application program writes the destination port and IP address on a datagram and then sends the datagram to its destination. UDP is less reliable than TCP because there are no delivery-assurance or error-detection-and-correction mechanisms built into the protocol.

Application protocols such as FTP, SMTP, and HTTP use TCP to provide reliable, stream-based communication between client and server programs. Other protocols, such as the Time Protocol, use UDP because speed of delivery is more important than end-to-end reliability.

4.2.2. The java.net Package

The java.net package provides several classes that support socket-based client/server communication.

The InetAddress class encapsulates Internet IP addresses and supports conversion between dotted decimal addresses and host names.

The Socket, ServerSocket, DatagramSocket, and MulticastSocket classes implement client and server sockets for connection-oriented and connectionless communication. The DatagramPacket class is used to construct UDP datagram packets. The SocketImpl and DatagramSocketImpl classes and the SocketImplFactory interface provide hooks for implementing custom sockets.

The URL, URLConnection, HttpURLConnection, and URLEncoder classes implement high-level browser-server Web connections. The ContentHandler and URLStreamHandler classes are abstract classes that have provided the basis for the implementation of Web content and stream handlers. They are supported by the ContentHandlerFactory and URLStreamHandlerFactory interfaces.

The FileNameMap interface is used to map filenames to MIME types.

The classes in the java.net package can be listed as follows:

The Classes

- `ContentHandler`
- `DatagramPacket`
- `DatagramSocket`
- `DatagramSocketImpl`
- `HttpURLConnection`
- `InetAddress`
- `MulticastSocket`
- `ServerSocket`
- `Socket`
- `SocketImpl`
- `URL`
- `URLConnection`
- `URLEncoder`
- `URLStreamHandler`

The Interfaces

- `ContentHandlerFactory`
- `FileNameMap`
- `SocketImplFactory`
- `URLStreamHandlerFactory`

Exceptions

- `BindException`
- `ConnectException`
- `MalformedURLException`
- `NoRouteToHostException`
- `ProtocolException`
- `SocketException`

- `UnknownHostException`
- `UnknownServiceException`

In this session we mainly discuss the Socket and ServerSocket classes.

4.2.3. The Socket Class

The Socket class implements client connection-based sockets. These sockets are used to develop applications that utilize services provided by connection-oriented server applications.

The access methods of the Socket class are used to access the I/O streams and connection parameters associated with a connected socket. The getInetAddress() and getPort() methods get the IP address of the destination host and the destination host port number to which the socket is connected. The getLocalPort() method returns the source host local port number associated with the socket. The getLocalAddress() method returns the local IP address associated with the socket. The getInputStream() and getOutputStream() methods are used to access the input and output streams associated with a socket. The close() method is used to close a socket.

The getSoLinger() and setSoLinger() methods are used to get and set a socket's SO_LINGER option, which identifies how long a socket is to remain open after a close() method has been invoked and data remains to be sent over the socket.

The getSoTimeout() and setSoTimeout() methods are used to get and set a socket's SO_TIMEOUT option, which is used to identify how long a read operation on the socket is to be blocked before it times out and the blocking ends.

The getTcpNoDelay() and setTcpNoDelay() methods are used to get and set a socket's TCP_NODELAY option, which is used to specify whether Nagle's algorithm should be used to buffer data that is sent over a socket connection. When TCP_NODELAY is true, Nagle's algorithm is disabled.

The setSocketImplFactory() class method is used to switch from the default Java socket implementation to a custom socket implementation. The toString() method returns a string representation of the socket.

The following PortTalkApp.java program is used to talk to a particular port on a given host on a line-by-line basis. It provides the options of sending a line to the specified port, receiving a line from the other host, or terminating the connection.

```
import java.lang.System;
import java.net.Socket;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.io.*;
public class PortTalkApp {
 public static void main(String args[]){
  PortTalk portTalk = new PortTalk(args);
  portTalk.displayDestinationParameters();
  portTalk.displayLocalParameters();
  portTalk.chat();
  portTalk.shutdown();
 }
}
```

```
class PortTalk {
 Socket connection;
 DataOutputStream outStream;
 BufferedReader inStream;
 public PortTalk(String args[]){
  if(args.length!=2) error("Usage: java PortTalkApp host port");
  String destination = args[0];
  int port = 0;
  try {
   port = Integer.valueOf(args[1]).intValue();
  }catch (NumberFormatException ex){
   error("Invalid port number");
  }
  try{
   connection = new Socket(destination,port);
  }catch (UnknownHostException ex){
   error("Unknown host");
  }
  catch (IOException ex){
   error("IO error creating socket");
  }
  try{
   inStream = new BufferedReader(
    new InputStreamReader(connection.getInputStream()));
   outStream = new DataOutputStream(connection.getOutputStream());
  }catch (IOException ex){
   error("IO error getting streams");
  }
  System.out.println("Connected to "+destination+" at port "+port+".");
 }
 public void displayDestinationParameters(){
  InetAddress destAddress = connection.getInetAddress();
  String name = destAddress.getHostName();
  byte ipAddress[] = destAddress.getAddress();
  int port = connection.getPort();
  displayParameters("Destination ",name,ipAddress,port);
 }
 public void displayLocalParameters(){
  InetAddress localAddress = null;
  try{
   localAddress = InetAddress.getLocalHost();
  }catch (UnknownHostException ex){
   error("Error getting local host information");
  }
  String name = localAddress.getHostName();
  byte ipAddress[] = localAddress.getAddress();
  int port = connection.getLocalPort();
  displayParameters("Local ",name,ipAddress,port);
 }
 public void displayParameters(String s,String name,
    byte ipAddress[],int port){
  System.out.println(s+"host is "+name+".");
  System.out.print(s+"IP address is ");
  for(int i=0;i<ipAddress.length;++i)
   System.out.print((ipAddress[i]+256)%256+".");
  System.out.println();
  System.out.println(s+"port number is "+port+".");
 }
 public void chat(){
  BufferedReader keyboardInput = new BufferedReader(
   new InputStreamReader(System.in));
  boolean finished = false;
  do {
```

```
   try{
    System.out.print("Send, receive, or quit (S/R/Q): ");
    System.out.flush();
    String line = keyboardInput.readLine();
    if(line.length()>0){
     line=line.toUpperCase();
     switch (line.charAt(0)){
     case `S':
      String sendLine = keyboardInput.readLine();
      outStream.writeBytes(sendLine);
      outStream.write(13);
      outStream.write(10);
      outStream.flush();
      break;
     case `R':
      int inByte;
      System.out.print("***");
      while ((inByte = inStream.read()) != `\n')
      System.out.write(inByte);
      System.out.println();
      break;
     case `Q':
      finished=true;
      break;
     default:
      break;
     }
    }
   }catch (IOException ex){
    error("Error reading from keyboard or socket");
   }
  } while(!finished);
 }
 public void shutdown(){
  try{
   connection.close();
  }catch (IOException ex){
   error("IO error closing socket");
  }
 }
 public void error(String s){
  System.out.println(s);
  System.exit(1);
 }
}
```

To see how PortTalkApp works, run it using the following command line:

```
>java PortTalkApp smaug.cm.deakin.edu.au 7
Connected to smaug.cm.deakin.edu.au at port 7.
Destination host is smaug.cm.deakin.edu.au.
Destination IP address is 128.184.80.150.
Destination port number is 7.
Local host is wanlei.
Local IP address is 139.132.118.113.
Local port number is 1548.
Send, receive, or quit (S/R/Q):
```

PortTalkApp connects to my server at port 7. This is the port number for the echo server application. It is used to test Internet communication between hosts. It identifies my host's name, IP address, and destination port number. In this example, I am connecting

from my laptop on my local area network. Its name is wanlei and it has the 139.132.118.113 IP address. When you run the program, your host name and IP address will be displayed. The local port number that I am connecting from is port 1548.

PortTalkApp asks you whether you want to send a line, receive a line, or quit the program. Whether you elect to send or receive is important. If you decide to receive a line and the host is not sending any data, your program will block while it waits to receive information from a socket-based stream.

You can also use PortTalkApp to talk to other ports. For example, you can use it to talk to port 25 of hosts that support the Simple Mail Transport Protocol in order to send email to someone who is served by that host.

4.2.4. The ServerSocket Class

The ServerSocket class implements a TCP server socket. It provides three constructors that specify the port to which the server socket is to listen for incoming connection requests, an optional maximum connection request queue length, and an optional Internet address. The Internet address argument allows *multihomed* hosts (that is, hosts with more than one Internet address) to limit connections to a specific interface.

The accept() method is used to cause the server socket to listen and wait until an incoming connection is established. It returns an object of class Socket once a connection is made. This Socket object is then used to carry out a service for a single client. The getInetAddress() method returns the address of the host to which the socket is connected. The getLocalPort() method returns the port on which the server socket listens for an incoming connection. The toString()method returns the socket's address and port number as a string in preparation for printing.

The getSoTimeout() and setSoTimeout() methods set the socket's SO_TIMEOUT parameter. The close() method closes the server socket.

The static setSocketFactory() method is used to change the default ServerSocket implementation to a custom implementation. The implAccept() method is used by subclasses of ServerSocket to override the accept() method.

The following ReverServerApp.java program is a simple server that listens on port 1234 for incoming connections from client programs. When ReverServerApp connects to a client it reads one line of text at a time from the client, reverses the characters in the text line, and sends them back to the client.

```
import java.lang.System;
import java.net.ServerSocket;
import java.net.Socket;
import java.io.*;
public class ReverServerApp {
 public static void main(String args[]){
  try{
   ServerSocket server = new ServerSocket(1234);
   int localPort = server.getLocalPort();
   System.out.println("Reverse Server is listening on port "+
     localPort+".");
   Socket client = server.accept();
   String destName = client.getInetAddress().getHostName();
```

```
   int destPort = client.getPort();
   System.out.println("Accepted connection to "+destName+" on port "+
    destPort+".");
   BufferedReader inStream = new BufferedReader(
    new InputStreamReader(client.getInputStream()));
   DataOutputStream outStream =
     new DataOutputStream(client.getOutputStream());
   boolean finished = false;
   do {
    String inLine = inStream.readLine();
    System.out.println("Received: "+inLine);
    if(inLine.equalsIgnoreCase("quit")) finished=true;
    String outLine=new ReverseString(inLine.trim()).getString();
    for(int i=0;i<outLine.length();++i)
     outStream.write((byte)outLine.charAt(i));
    outStream.write(13);
    outStream.write(10);
    outStream.flush();
    System.out.println("Sent: "+outLine);
   } while(!finished);
   inStream.close();
   outStream.close();
   client.close();
   server.close();
  }catch (IOException ex){
   System.out.println("IOException occurred.");
  }
 }
}
class ReverseString {
 String s;
 public ReverseString(String in){
  int len = in.length();
  char outChars[] = new char[len];
  for(int i=0;i<len;++i)
   outChars[len-1-i]=in.charAt(i);
  s = String.valueOf(outChars);
 }
 public String getString(){
  return s;
 }
}
```

To see how ReverServerApp works, you need to run it in a separate window and then use PortTalkApp to feed it lines of text. First, run ReverServerApp using the following command line:

```
>java ReverServerApp
Reverse Server is listening on port 1234.
```

ReverServerApp notifies you that it is up and running. In a separate window, run PortTalkApp as follows, supplying your host name instead of wanlei.cm.deakin.edu.au:

```
java PortTalkApp wanlei.cm.deakin.edu.au 1234
```

You can use localhost or 127.0.0.1 as an IP address if you do not have a host name or cannot determine your IP address.

4.3. A Simple Connection-Oriented Communication Example

4.3.1. Essential Components of TCP Communication

Our first example of Java client-server communication shows some basic steps in establishing a TCP communication connection. The essential components of any communication are:

- The underlying communication protocol. In this instance, the TCP.
- The application's communication protocol
- The client program.
- The server program.

In a TCP communication, the following steps are needed:

- Create the server socket and listen to client connection request.
- Create the client socket and issue a connection request to the server.
- The server accepts the connection. The communication channel is then established and communications between the client and the server can be carried out using the application's communication protocol.

The application's protocol is very simple in this case:

- The client sends a "Hello, Server" string to the server.
- The server replies a string "You have connected to the Very Simple Server."
- Both client and server exit.

4.3.2. Implementing a TCP Client Program

The following steps are carried out when implementing a TCP client program:

- Create a socket for communicating with the server on a specific port.
- Create an *InputStream*, in our case, a *BufferedReader*, to receive responses from the server.
- Create an *OutputStream*, in our case, a *PrintWriter*, to send messages to the server.
- Write to the *OutputStream*.
- Read from the *InputStream*.
- Close the *InputStream*, the *OutputStream*, and the socket before the client exits.

The client program, named C.java, is as follows.

```
import java.io.*;
import java.net.*;
public class C {
  public static final int DEFAULT_PORT = 6789;
```

```
  public static void usage() {
    System.out.println("Usage: java C [<port>]");
    System.exit(0);
  }
  public static void main(String[] args) {
    int port = DEFAULT_PORT;
    Socket s = null;
    // parse the port specification
    if ((args.length != 0) && (args.length != 1)) usage();
    if (args.length == 0) port = DEFAULT_PORT;
    else {
      try {
        port = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e) {
        usage();
      }
    }
    try {
      BufferedReader reader;
      PrintWriter writer;
      // create a socket to communicate to the specified host and port
      s = new Socket("localhost", port);
      // create streams for reading and writing
       reader = new BufferedReader(new InputStreamReader(s.getInputStream()));
       writer = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
      // tell the user that we've connected
      System.out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());
      String line;
       // write a line to the server
      writer.println("Hello, Server");
       writer.flush();
      // read the response (a line) from the server
      line = reader.readLine();
      // write the line to console
      System.out.println("Server says: " + line);
       reader.close();
       writer.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

The program assumes that the server runs on the “local host” (i.e., with an IP address of 127.0.0.1) and uses a default port number of 6789. The statement

```
System.out.println("Connected to " + s.getInetAddress() + ":" + s.getPort());
```

displays the server host IP address and the port number that the client has connected to.

4.3.3. Implementing a TCP Server Program

When implementing a TCP server, the following steps are carried out:

- Create a server socket to listen and accept client connection requests.
- Create an *InputStream*, in our case, a *BufferedReader*, to read messages from the client.
- Create an *OutputStream*, in our case, a *PrintWriter*, to send replies to the client.
- Read from the *InputStream*.
- Write to the *OutputStream*.
- Close the *InputStream*, the *OutputStream*, and the socket before the server exits.

The server program, named S.java, is as follows.

```
import java.net.*;
import java.io.*;
public class S {
  public final static int DEFAULT_PORT = 6789;
  public static void main (String args[]) throws IOException {
    Socket client;
    if (args.length != 1)
      client = accept (DEFAULT_PORT);
    else
      client = accept (Integer.parseInt (args[0]));
    try {
      PrintWriter writer;
      BufferedReader reader;
       reader = new BufferedReader(new
       InputStreamReader(client.getInputStream()));
       writer = new PrintWriter(new
       OutputStreamWriter(client.getOutputStream()));
       // read a line
       String line = reader.readLine();
       System.out.println("Client says: " + line);
       // write a line
      writer.println ("You have connected to the Very Simple Server.");
       writer.flush();
       reader.close();
       writer.close();
    } finally { // closing down the connection
      System.out.println ("Closing");
      client.close ();
    }
  }
  static Socket accept (int port) throws IOException {
    System.out.println ("Starting on port " + port);
    ServerSocket server = new ServerSocket (port);
    System.out.println ("Waiting");
    Socket client = server.accept ();
    System.out.println ("Accepted from " + client.getInetAddress ());
    server.close ();
    return client;
  }
}
```

The server uses a default port of 6789 for communication. When a connection request is accepted, the server uses the following statement to display the IP address of the client computer:

```
System.out.println ("Accepted from " + client.getInetAddress ());
```

4.4. Variations on the Simple Communication Example

The simple communication example is of no practical use at all. A number of issues need to be addressed in order to improve the simple example into practical use:

- The exchange of multiple messages between the client and the server.
- The ability to run the server and the client programs on any Internet host.
- The ability for the server to deal with multiple client connections simultaneously.

4.4.1. Exchange of Multiple Messages

The first issue is to define the application's communication protocol to allow multiple exchanges of messages. Here is an example:

- The server:
  - After the establishing of the connection, the server sends an initial message to the client.
  - The server waits for the client's messages.
  - When the message arrives, the serve responses an "OK" to the clients. The message is displayed. If the incoming message is "Server Exit", then the server exits. Otherwise, return to the waiting step.
- The client:
  - After a successful connection, the client displays the initial response from the server.
  - The client reads a line from the keyboard, and sends it to the server. Then the client reads the response from the server and displays it.
  - The input string from the keyboard is checked. The client exits if the keyboard input string is "Server Exit" or the server is disconnected. Otherwise, return to the previous step.

The server program, named S1.java, is as follows:

```
import java.net.*;
import java.io.*;
public class S1 {
  public final static int DEFAULT_PORT = 6789;
  public static void main (String args[]) throws IOException {
    Socket client;
    if (args.length != 1)
      client = accept (DEFAULT_PORT);
    else
      client = accept (Integer.parseInt (args[0]));
```

```
    try {
      PrintWriter writer;
      BufferedReader reader;
       reader = new BufferedReader(new
InputStreamReader(client.getInputStream()));
       writer = new PrintWriter(new
OutputStreamWriter(client.getOutputStream()));
      writer.println ("You are now connected to the Simple Echo Server.");
       writer.flush();
      for (;;) {
         // read a line
         String line = reader.readLine();
        // and send back ACK
         writer.println("OK");
         writer.flush();
         System.out.println("Client says: " + line);
         if (line.equals("Server Exit")) {
           break;
        }
      }
       reader.close();
       writer.close();
    } finally {
      System.out.println ("Closing");
      client.close ();
    }
  }
  static Socket accept (int port) throws IOException {
    System.out.println ("Starting on port " + port);
    ServerSocket server = new ServerSocket (port);
    System.out.println ("Waiting");
    Socket client = server.accept ();
    System.out.println ("Accepted from " + client.getInetAddress ());
    server.close ();
    return client;
  }
}
```

The client program, named C1.java, is as follows:

```
import java.io.*;
import java.net.*;
public class C1 {
  public static final int DEFAULT_PORT = 6789;
  public static void usage() {
    System.out.println("Usage: java C1 [<port>]");
    System.exit(0);
  }
  public static void main(String[] args) {
    int port = DEFAULT_PORT;
    Socket s = null;
    int end = 0;
    // parse the port specification
    if ((args.length != 0) && (args.length != 1)) usage();
    if (args.length == 0) port = DEFAULT_PORT;
    else {
      try {
        port = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e) {
         usage();
      }
```

```
    }
    try {
      PrintWriter writer;
      BufferedReader reader;
      BufferedReader kbd;
      // create a socket to communicate to the specified host and port
      //InetAddress myhost = getLocalHost();
      s = new Socket("localhost", port);
      // create streams for reading and writing
       reader = new BufferedReader(new InputStreamReader(s.getInputStream()));
       OutputStream sout = s.getOutputStream();
       writer = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
      // create a stream for reading from keyboard
      kbd = new BufferedReader(new InputStreamReader(System.in));
      // tell the user that we've connected
      System.out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());
      String line;
      // read the first response (a line) from the server
      line = reader.readLine();
      // write the line to console
      System.out.println(line);
      while (true) {
        // print a prompt
        System.out.print("> ");
        System.out.flush();
        // read a line from console, check for EOF
        line = kbd.readLine();
        if (line.equals("Server Exit")) end = 1;
        // send it to the server
        writer.println(line);
        writer.flush();
        // read a line from the server
        line = reader.readLine();
        // check if connection is closed, i.e., EOF
        if (line == null) {
          System.out.println("Connection closed by server.");
          break;
        }
        if (end == 1) {
           break;
         }
        // write the line to console
        System.out.println("Server says: " + line);
      }
       reader.close();
       writer.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

4.4.2. Executing the Programs on Internet Hosts

The first thing to execute the client-server programs on Internet hosts is to know the IP addresses or/and the host names of the computers. The following program, named *InetExample.java*, displays the details of a host:

```
import java.net.*;
import java.io.*;
public class InetExample {
  public static void main (String args[]) {
    printLocalAddress ();
    Reader kbd = new FileReader (FileDescriptor.in);
    BufferedReader bufferedKbd = new BufferedReader (kbd);
    try {
      String name;
      do {
        System.out.print ("Enter a hostname or IP address: ");
        System.out.flush ();
        name = bufferedKbd.readLine ();
        if (name != null)
          printRemoteAddress (name);
      } while (name != null);
      System.out.println ("exit");
    } catch (IOException ex) {
      System.out.println ("Input error:");
      ex.printStackTrace ();
    }
  }
  static void printLocalAddress () {
    try {
      InetAddress myself = InetAddress.getLocalHost ();
      System.out.println ("My name : " + myself.getHostName ());
      System.out.println ("My IP : " + myself.getHostAddress ());
      System.out.println ("My class : " + ipClass (myself.getAddress ()));
    } catch (UnknownHostException ex) {
      System.out.println ("Failed to find myself:");
      ex.printStackTrace ();
    }
  }
  static char ipClass (byte[] ip) {
    int highByte = 0xff & ip[0];
    return (highByte < 128) ? 'A' : (highByte < 192) ? 'B' :
      (highByte < 224) ? 'C' : (highByte < 240) ? 'D' : 'E';
  }
  static void printRemoteAddress (String name) {
    try {
      System.out.println ("Looking up " + name + "...");
      InetAddress machine = InetAddress.getByName (name);
      System.out.println ("Host name : " + machine.getHostName ());
      System.out.println ("Host IP : " + machine.getHostAddress ());
      System.out.println ("Host class : " +
                          ipClass (machine.getAddress ()));
    } catch (UnknownHostException ex) {
      System.out.println ("Failed to lookup " + name);
    }
  }
}
```

The *main()* method first calls the *PrintLocalAddress()* methods to display the local host name and IP address. Then it sits in a loop that reads host names from the keyboard and

uses the *PrintRemoteAddress()* method to display the *InetAddress* information about the host.

To allow our client-server program to run on any host, we only need to change the client program: the server program can be the same. Here is the new client program, named as C2.java:

```
import java.io.*;
import java.net.*;
public class C2 {
  public static final int DEFAULT_PORT = 6789;
  public static void usage() {
    System.out.println("Usage: java C2 <serverhost>");
    System.exit(0);
  }
  public static void main(String[] args) {
    int port = DEFAULT_PORT;
    String address = "";
    Socket s = null;
    int end = 0;
    // parse the port specification
    if ((args.length != 0) && (args.length != 1)) usage();
    if (args.length == 0) {
      port = DEFAULT_PORT;
       address = "localhost";
    } else {
        address = args[0];
    }
    try {
      PrintWriter writer;
      BufferedReader reader;
      BufferedReader kbd;
      // create a socket to communicate to the specified host and port
      s = new Socket(address, port);
      // create streams for reading and writing
       reader = new BufferedReader(new InputStreamReader(s.getInputStream()));
       OutputStream sout = s.getOutputStream();
       writer = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
      // create a stream for reading from keyboard
      kbd = new BufferedReader(new InputStreamReader(System.in));
      // tell the user that we've connected
      System.out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());
      String line;
      // read the first response (a line) from the server
      line = reader.readLine();
      // write the line to console
      System.out.println(line);
      while (true) {
        // print a prompt
        System.out.print("> ");
        System.out.flush();
        // read a line from console, check for EOF
        line = kbd.readLine();
        if (line.equals("Server Exit")) end = 1;
        // send it to the server
        writer.println(line);
        writer.flush();
        // read a line from the server
        line = reader.readLine();
        // check if connection is closed, i.e., EOF
        if (line == null) {
```

```
          System.out.println("Connection closed by server.");
          break;
        }
        if (end == 1) {
           break;
         }
        // write the line to console
        System.out.println("Server says: " + line);
      }
       reader.close();
       writer.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

In this version, we use a variable

```
String address = "";
```

To store the IP address entered from the keyboard. The socket is then created using the following statement:

```
s = new Socket(address, port);
```

4.4.3. Supporting Multiple Clients

To support multiple clients, only the server program needs to be changed; the client program remains the same. When the server is initialized, we obtain the server socket and waiting for client connection requests. When a client connection request is accepted, we use a thread to deal with the accepted incoming client connection. The server then goes back to wait for new connection requests. Clients can issue two commands in this time, one is a “Client Exit” command, telling the server that the current client is willing to disconnect. The other is the “Server Exit” command, in which the the whole program exits.

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

public class S3 extends Thread {
  public final static int DEFAULT_PORT = 6789;
  private Socket client = null;

  public S3(Socket inSock) {
    super("echoServer");
    client = inSock;
  }

  public void run() {
    Socket cSock = client;
```

```
    PrintWriter writer;
    BufferedReader reader;
    try {
       String line;
       System.out.println ("Accepted from " + cSock.getInetAddress());
       reader = new BufferedReader(new
InputStreamReader(cSock.getInputStream()));
       writer = new PrintWriter(new
OutputStreamWriter(cSock.getOutputStream()));
      writer.println ("You are now connected to the Simple Echo Server.");
       writer.flush();
      for (;;) {
         // read a line
         line = reader.readLine();
        // and send back ACK
         writer.println("OK");
         writer.flush();
         System.out.println("Client says: " + line);
         if (line.equals("Server Exit") || line.equals("Client Exit")) break;
      }
      System.out.println ("Closing the client " + cSock.getInetAddress());
      reader.close();
       writer.close();
      cSock.close ();
       if (line.equals("Server Exit")) {
        System.out.println ("Closing the server");
        // server.close ();
         System.exit(0);
       }
    } catch (IOException e1) {
      System.err.println("Exception: " + e1.getMessage());
       System.exit(1);
    }
  }

  public static void main (String args[]) {
    ServerSocket server = null;
    try {
      server = new ServerSocket(DEFAULT_PORT);
      System.out.println ("Starting on port " + DEFAULT_PORT);
    } catch (IOException e) {
      System.err.println("Exception: could't make server socket.");
       System.exit(1);
    }
    while (true) {
      Socket incomingSocket = null;
      // wait fot a connection request
      System.out.println("Waiting...");
      try {
        incomingSocket = server.accept();
        // call a thread to deal with a connection
        S3 es = new S3(incomingSocket);
        es.start();
      } catch (IOException e) {
        System.err.println("Exception: could't make server socket.");
         System.exit(1);
      }
    }
  }
}
```

5. Developing Clients

5.1. Study Points

- Understand basics of client-side networking.
- Be able to complete simple client programs using various Internet protocols.
- Be able to use the URL class provided by Java.
- Understand the principles of the URLConnection class and be able to use its basic functions.
- Understand basics of protocol and content handlers.

References: (1). [JNP] Chapters 10, 15, 16, 17. (2). [Java2U]: Chapters 31, 33. (3). [Java2H]: Chapters 14, 15, 19.

5.2. The Client and its Sockets

5.2.1. Types of Clients

Of the client/server applications that are found on the Internet, only a small group is typically used. These include email, the Web, FTP, Usenet newsgroups, and Telnet. Typical Internet client programs include email programs, Web browsers, FTP programs, news reader programs, and Telnet clients.

- *Email programs* provide an easy-to-use interface by which mail can be created, sent, retrieved, displayed, and managed. Popular Windows-based clients include Eudora and Outlook. UNIX systems provide a number of popular email clients including Pine, Elm, and mh.
- *Web browsers* provide a window on the World Wide Web and support the display of Web pages, including Java programs. The Netscape Navigator and Microsoft Internet Explorer browsers are the most popular browsers on the Web and are Java-capable. They are supported on UNIX, Windows, Macintosh, and other systems.
- *FTP programs* provide a convenient way to retrieve files from public Internet file servers and from private file directories. Although a number of user-friendly FTP client programs are available, the simple text-based FTP client is still the most popular and most widely supported. WS_FTP is a popular GUI-based FTP client for Windows platforms.
- *News reader* programs simplify the process of working with messages that are posted to Usenet newsgroups. A number of netnews client programs are available for Windows, Macintosh, UNIX, and other operating system platforms.
- *Telnet clients* are used to remotely log in to other systems. These systems are usually UNIX or other operating systems that are powerful enough to provide the underlying capabilities needed to implement multiuser support. Windows and Macintosh

systems, because of their inherent limitations, do not support Telnet server applications.

Some client programs, such as Netscape Communicator, consist of an integrated suite of popular programs. For example, Netscape Communicator includes a Web browser, a mail client, and a news reader, among other clients.

Of course, in specific applications you can create your own clients and servers.

Client programs perform a service for their users by connecting with their server counterparts, forwarding service requests based on user inputs, and providing the service results back to the user.

In most cases, the client must initiate the connection. Typically, the server listens on a well-known port for a client connection. The client initiates the connection, which is accepted by the server. The client sends a service request to the server, based on user inputs. The server receives the service request, performs the service, and returns the results of the service to the client. The client receives the service results and displays them to the user.

5.2.2. The TCP socket for Clients

The TCP socket represents a reliable connection for the transmission of data between two hosts. It isolates the Java program from the details of packet encodings, lost and retransmitted packets, and packets that arrive out of order.

There are four fundamental operations a client socket performs. These are:

1. Connect to a remote machine
2. Send data
3. Receive data
4. Close the connection

A socket may not be connected to more than one host at a time.

The `java.net.Socket` class allows you to perform all four fundamental socket operations. You can connect to remote machines; you can send data; you can receive data; you can close the connection.

Connection is accomplished through the constructors. Each `Socket` object is associated with exactly one remote host. To connect to a different host, you must create a new `Socket` object.

```
public Socket(String host, int port)
 throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddress,
 int localPort) throws IOException
public Socket(InetAddress address, int port, InetAddress localAddress,
  int localPort)  throws IOException
```

Sending and receiving data is accomplished with output and input streams. There are methods to get an input stream for a socket and an output stream for the socket.

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Java places no restrictions on reading and writing to sockets. One thread can read from a socket while another thread writes to the socket at the same time. (How does this differ from one thread reading a file while another thread writes to the file?)

5.2.3. Reading from and Writing to a Socket for Echo

The echo protocol simply echoes back anything its sent. The following echo client reads data from an input stream, then passes it out to an output stream connected to a socket, connected to a network echo server. A second thread reads the input coming back from the server. The `main()` method reads some file names from the command line and passes them into the output stream.

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

public class Echo {
  InetAddress server;
  int port = 7;
  InputStream theInput;
  public static void main(String[] args) {
    if (args.length == 0) {
      System.err.println("Usage: java Echo file1 file2...");
      System.exit(1);
    }
    Vector v = new Vector();
    for (int i = 0; i < args.length; i++) {
      try {
        FileInputStream fis = new FileInputStream(args[i]);
        v.addElement(fis);
      }
      catch (IOException e) {
      }
    }
    InputStream in = new SequenceInputStream(v.elements());
    try {
      Echo d = new Echo("smeagol.cm.Deakin.edu.au", in);
      d.start();
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }

  public Echo() throws UnknownHostException {
    this (InetAddress.getLocalHost(), System.in);
  }

  public Echo(String name) throws UnknownHostException {
    this(InetAddress.getByName(name), System.in);
  }
```

```
  public Echo(String name, InputStream is) throws UnknownHostException {
    this(InetAddress.getByName(name), is);
  }

  public Echo(InetAddress server) {
    this(server, System.in);
  }

  public Echo(InetAddress server, InputStream is) {
    this.server = server;
    theInput = is;
  }

  public void start() {
    try {
      Socket s = new Socket(server, port);
      CopyThread toServer = new CopyThread("toServer",
       theInput, s.getOutputStream());
      CopyThread fromServer = new CopyThread("fromServer",
       s.getInputStream(), System.out);
      toServer.start();
      fromServer.start();
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}

class CopyThread extends Thread {
  InputStream in;
  OutputStream out;

  public CopyThread(String name, InputStream in, OutputStream out) {
    super(name);
    this.in = in;
    this.out = out;
  }

  public void run() {
    byte[] b = new byte[128];
    try {
      while (true) {
        int n = in.available();
        if (n == 0) Thread.yield();
        else System.err.println(n);
        if (n > b.length) n = b.length;
        int m = in.read(b, 0, n);
        if (m == -1) {
          System.out.println(getName() + " done!");
          break;
        }
        out.write(b, 0, n);
      }
    }
    catch (IOException e) {
    }
  }
}
```

5.3. Dealing with HTTP Servers

5.3.1. The Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems and has been in use by the World-Wide Web global information initiative since 1990. It is designed to support communication between clients and a hypermedia information server. The commonly used version is HTTP 1.0, described in RFC 1945, and is the seventh (and last) release of the HTTP/1.0 specification. The most recent version is the HTTP/1.1, first described in RFC 2068. A recent revision of HTTP/1.1 is described in the document called "*RFC 2616: Hypertext Transfer Protocol - HTTP/1.1*", published June 1999.

HPPT basically defines the internal structure of supported requests and responses. Thus, from the clients' point of view, the capabilities of the server are completely captured by the description of message exchanges (requests and responses) the protocol views.

The Uniform Resource Locator (URL) is used in HTTP to identify the data to be transmitted. It is made up of four parts: *protocol*, *host*, *path*, and *file*.

- The *protocol* part identifies the service protocol to be used to retrieve the resource.
- The *host* part must be a complete Internet address or official name of the host machine on which the data resides. It may also contain a port number on which the HTTP server listens.
- The *path* part must specify the complete path through the server's directory hierarchy.
- The *file* part must specify the file name containing the required data.

Given a URL, a client can therefore determine the protocol to use and the target server of the desired document/resource. The appropriate communication can then be established, with the client providing the complete pathname of the resource and the server responding by transmitting back the contents of the resource.

The main characteristics of HTTP are listed below:

- In one connection, a client can request only a certain number of documents from the server. This maximum number can be specified in the web server as well as the client.
- The connection can be closed by either party.
- HTTP is a stateless protocol. Every transaction between the client and the server is assumed to be independent of other transactions.
- It is a message-based protocol that follows the object-oriented model.
- The MIME (Multipurpose Internet Mail Extensions) format of data expression allows client-server systems to be built independently of the data being transferred.

An HTTP request has the following format (NL represents a newline):

```
Method URL_AbsolutePath ProtocolVersion NL [Header][NL Data]
```

The *Method* can be GET, HEADER, and POST. The GET method is the simplest and fastest one for retrieving information identified by the URL_AbsolutePath. Note that if URL_AbsolutePath refers to an executable program then the GET method expects the result of that program from the server.

The HEAD method is identical to the GET method, except that the server only returns the header information. The actual document is not transferred. This method is useful if the client wants to test the validity and availability of the information source.

The POST method allows a client to supply supplemental data with the HTTP request to the server. This data is generally used by the requested document, which is normally an executable program to work out the details of the accessing that data after it is transferred to the server. The CGI is one prominent standard supported by most web servers for making the data available to the executable script or program. The client program can utilize the POST method to annotate existing resources residing at the server, provide keywords to search databases, post messages to newsgroups or mailing lists, and even supply data for remotely adding or updating database records.

An HTTP request can contain optional header fields to supply additional information about the request or about the client itself.

The data optionally supplied with an HTTP request can be any MIME-conforming message. The protocol version identifies the HTTP version being used by the server (for example, HTTP/1.0 or HTTP/1.1).

An HTTP response from the server has the following format:

```
ProtocolVersion StatusCode Reason NL [Header][NL Data]
```

The status code in a response is a three-digit integer describing the result of the request. The reason string is a more detailed description of the returned status code. The first digit of the status code characterizes the type of status returned as follows:

- Informational (type 1xx)
- Successful (type 2xx)
- Redirection (type 3xx)
- Client error (type 4xx)
- Server error (type 5xx)

For detailed description of the HTTP protocol, refer to the specifications at:

http://www.w3.org/pub/WWW/Protocols/

5.3.2. Getting Web Pages from a Web Server

The first client-side example using the HTTP protocol is to download a Web page from an HTTP server then displays the page to the screen. This example is illustrated in section 14.4 of the text. Basically, the program asks the user to type in a URL, then uses a *socket* to connect to the Web server and try to download the specified web page from the server. The content of the page is displayed on the screen if the download is successful. After

that, the program goes back to the loop to get another URL from the keyboard. In order to discuss some features of the program, we list the program below:

```
import java.net.*;
import java.io.*;
public class GrabPage {
  public GrabPage (String textURL) throws IOException {
    dissect (textURL);
  }
  protected String host, file;
  protected int port;
  protected void dissect (String textURL) throws MalformedURLException {
    URL url = new URL (textURL);
    host = url.getHost ();
    port = url.getPort ();
    if (port == -1)
      port = 80;
    file = url.getFile ();
  }
  public void grab () throws IOException {
    connect ();
    try {
      fetch ();
    } finally {
      disconnect ();
    }
  }
  protected Writer writer;
  protected BufferedReader reader;
  protected void connect () throws IOException {
    Socket socket = new Socket (host, port);
    OutputStream out = socket.getOutputStream ();
    writer = new OutputStreamWriter (out, "latin1");
    InputStream in = socket.getInputStream ();
    Reader reader = new InputStreamReader (in, "latin1");
    this.reader = new BufferedReader (reader);
  }
  protected void fetch () throws IOException {
    writer.write ("GET " + file + " HTTP/1.0\r\n\n");
    writer.flush ();
    PrintWriter console = new PrintWriter (System.out);
    String input;
    while ((input = reader.readLine ()) != null)
      console.println (input);
    console.flush ();
  }
  protected void disconnect () throws IOException {
    reader.close ();
  }
  public static void main (String[] args) throws IOException {
    Reader kbd = new FileReader (FileDescriptor.in);
    BufferedReader bufferedKbd = new BufferedReader (kbd);
    while (true) {
      String textURL;
      System.out.print ("Enter a URL: ");
      System.out.flush ();
      if ((textURL = bufferedKbd.readLine ()) == null)
        break;
      try {
        GrabPage grabPage = new GrabPage (textURL);
        grabPage.grab ();
      } catch (IOException ex) {
```

```
        ex.printStackTrace ();
        continue;
      }
      System.out.println ("- OK -");
    }
    System.out.println ("exit");
  }
}
```

A few notes about the program. First, the program uses the URL class (to be discussed later in this session) to dissect the URL string typed in by the user into various parts. Then the program connects to the HTTP server, fetches the web page according to the given URL using the "GET" command of the HTTP protocol, and disconnect from the server. The display of the content of the web page is done in the *fetch*() method of the program, i.e., when a line is read from the server, it is displayed immediately on the screen.

The second example in the text is a class that performs an HTTP POST operation. Students are required to understand the example and use it in a client program to access the web server.

5.4. Dealing with Servers of other Internet Protocols

There have been many Internet protocols used in today's Internet applications. Table 15.1 lists some commonly used ones and the RFC document numbers that specify these protocols.

Protocol	**Name**	**RFC**
IP	Internet Protocol	791
UDP	User Datagram Protocol	768
TCP	Transmission Control Protocol	793
TELNET	Telnet Protocol	854, 855
FTP	File Transfer Protocol	959
SMTP	Simple Mail Transfer Protocol	821
DOMAIN	Domain Name System	1034, 1035
ECHO	Echo Protocol	862
TIME	Time Server Protocol	868
TFTP	Trivial File Transfer Protocol	1350
PPP	Point-to-Point Protocol	1661
FINGER	Finger User Information Protocol	1288

Table 15.1. Some Internet protocol specification documents

5.4.1. A Finger Client

The finger protocol is specified in RFC 1288. When using a finger server, the client should make a TCP connection to port 79 of the remote machine and transfer a finger query in 8-bit ASCII; the host will respond with an ASCII result, detailing the user(s) logged on the machine. The following finger program *Finger.java* asks the user to input a host to which to connect and an optional username. The returned information about the logged on users will then be displayed if the connection is successful.

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

public class Finger {
  public static final int DEFAULT_PORT = 79;

  protected boolean verbose;
  protected int port;
  protected String host, query;

  public Finger (String request, boolean verbose) throws IOException {
    this.verbose = verbose;
    int at = request.lastIndexOf ('@');
    if (at == -1) {
      query = request;
      host = InetAddress.getLocalHost ().getHostName ();
      port = DEFAULT_PORT;
    } else {
      query = request.substring (0, at);
      int colon = request.indexOf (':', at + 1);
      if (colon == -1) {
        host = request.substring (at + 1);
        port = DEFAULT_PORT;
      } else {
        host = request.substring (at + 1, colon);
        port = Integer.parseInt (request.substring (colon + 1));
      }
      if (host.equals (""))
        host = InetAddress.getLocalHost ().getHostName ();
    }
  }

  public Finger (String query, String host, int port, boolean verbose) throws
IOException {
    this.query = query;
    this.host = host.equals ("") ?
      InetAddress.getLocalHost ().getHostName () : host;
    this.port = (port == -1) ? DEFAULT_PORT : port;
    this.verbose = verbose;
  }

  public Reader finger () throws IOException {
    Socket socket = new Socket (host, port);
    OutputStream out = socket.getOutputStream ();
    OutputStreamWriter writer = new OutputStreamWriter (out, "latin1");
    if (verbose)
      writer.write ("/W");
    if (verbose && (query.length () > 0))
      writer.write (" ");
    writer.write (query);
```

```
    writer.write ("\r\n");
    writer.flush ();
    return new InputStreamReader (socket.getInputStream (), "latin1");
  }

  public static void display (Reader reader, Writer writer) throws IOException
{
    PrintWriter printWriter = new PrintWriter (writer);
    BufferedReader bufferedReader = new BufferedReader (reader);
    String line;
    while ((line = bufferedReader.readLine ()) != null)
      printWriter.println (line);
    reader.close ();
  }

  public static void main (String[] args) throws IOException {
    if (((args.length == 2) && !args[0].equals ("-l")) || (args.length > 2))
      throw new IllegalArgumentException
        ("Syntax: Finger [-l] [<username>][{@<hostname>}[:<port>]]");

    boolean verbose = (args.length > 0) && args[0].equals ("-l");
    String query = (args.length > (verbose ? 1 : 0)) ?
      args[args.length - 1] : "";

    Finger finger = new Finger (query, verbose);
    Reader result = finger.finger ();
    Writer console = new FileWriter (FileDescriptor.out);
    display (result, console);
    console.flush ();
  }
}
```

The command syntax is:

```
Java Finger [-l] [<username>][@[<hostname>{@<hostname>}][:<port>]]
```

The “-l” flag indicates that a verbose query should be made. If no port number is specified, the default port number of 79 is used. For example, the command:

```
>java Finger wanlei@smaug.cm.deakin.edu.au
```

returns the following information:

```
Login       Name               TTY         Idle    When    Where
wanlei   Wanlei Zhou           pts/34        10 Thu 13:31  smeagol.cm.deakin.ed
```

while the command:

```
>java Finger –l wanlei@smaug.cm.deakin.edu.au
```

returns the following information:

```
Login name: wanlei                      In real life: Wanlei Zhou
Directory: /home/wanlei                 Shell: /bin/csh
On since Oct 18 13:31:52 on pts/34 from smeagol.cm.deakin.edu.au
10 minutes Idle Time
```

```
No unread mail
No Plan.
```

Note that some hosts may not have a finger server running on them or the firewall may forbid the finger connection from outside of the intranet.

5.4.2. A DNS Client

A more complex client program is the use of the DNS (domain naming system) server described in section 15.3 of the [HSH] book. The DNS is a globally distributed database storing the mapping between the Internet address and the host name of Internet hosts. It also stores other information such as the IP class of the host and so on. The description of the DNS protocol is specified in RFC 1035. Most organizations have their own DNS server(s) that list(s) the mapping between the hosts and their IP address within the organization. For example, the DNS server of School of Computing and Mathematics is named as *akma.cm.deakin.edu.au*, with an IP address of 128.184.80.2.

The DNS client developed in section 15.3 of the [HSH] book allows you to query a name server for information about a given host name or domain name. Since the DNS protocol is more complex than the finger protocol, a number of helper classes are developed in the text to complete the task. The implementation consists of the following programs:

DNS.java: This class contains various constants that are used by the DNS-related classes.

DNSQuery.java: This class represents a DNS query. It includes details of the host name being queried and the type of query, and provides methods to allow this query to be transmitted and a response to be received.

DNSRR.java: This class represents a DNS resource record, which is the encapsulation of a piece of DNS information.

DNSInputStream.java: This class is an InputStream that provides helper methods to decode the typical data that are returned in a DNS response.

NSLookup.java: This is a command-line nslookup client that uses these DNS classes to perform DNS resolution.

Students are encouraged to understand the working of the program and try to test its execution.

5.5 The URL Class

5.5.1. The Basics of the URL Class

A *URL*, short for "Uniform Resource Locator", is a way to unambiguously identify the location of a resource on the Internet. The simplest way for a Java program to locate and retrieve data from the network is to use the URL class provided by Java in the *java.net.URL*. This class is an abstraction of a URL like *http://www.cm.deakin.edu.au/*.

The `java.net.URL` class represents a URL. There are constructors to create new URLs and methods to parse the different parts of a URL. However the heart of the class are the

methods that allow you to get an InputStream from a URL so you can read data from a server.

The `URL` class is closely tied to protocol and content handlers. The objective is to separate the data being downloaded from the the protocol used to download it. The protocol handler is responsible for communicating with the server, that is moving bytes from the server to the client. It handles any necessary negotiation with the server and any headers. Its job is to return only the actual bytes of the data or file requested. The content handler takes those bytes and translates them into some kind of Java object such as an `InputStream` or `ImageProducer`.

When you construct a `URL` object, Java looks for a protocol handler that understands the protocol part of the URL such as "http" or "mailto". If no such handler is found, the constructor throws a `MalformedURLException`. The protocols include ftp, http, file, gopher, mailto, etc.

You can create URL instances using its constructors. The simplest one is the constructor that takes an absolute URL in string form as its single argument:

public URL(String url) throws MalformedURLException

The following example (named `ProtocolTester.java`) uses this constructor to create a number of URL instances to determine which protocol a virtual machine does and does not support. It attempts to construct a URL object for each of the 14 protocols (8 standard ones, 3 custom protocols for various Java APIs, and 4 undocumented protocols used internally by HotJava). If the constructor succeeds, you know that the protocol is supported. Otherwise, a *MalformedURLException* is thrown, and you know that the protocol is not supported.

```
/* Which protocols does a virtual machine support? */
import java.net.*;
public class ProtocolTester {
  public static void main(String[] args) {
    // hypertext transfer protocol
    testProtocol("http://www.adc.org");
    // secure http
    testProtocol("https://www.amazon.com/exec/obidos/order2/");
    // file transfer protocol
    testProtocol("ftp://metalab.unc.edu/pub/languages/java/javafaq/");
    // Simple Mail Transfer Protocol
    testProtocol("mailto:elharo@metalab.unc.edu");
    // telnet
    testProtocol("telnet://dibner.poly.edu/");
    // local file access
    testProtocol("file:///etc/passwd");
    // gopher
    testProtocol("gopher://gopher.anc.org.za/");
    // Lightweight Directory Access Protocol
    testProtocol(
"ldap://ldap.itd.umich.edu/o=University%20of%20Michigan,c=US?postalAddress");
    // Jar
    testProtocol(
"jar:http://metalab.unc.edu/java/books/javaio/ioexamples/javaio.jar!/com/macfaq
/io/StreamCopier.class");
    // NFS, Network File System
    testProtocol("nfs://utopia.poly.edu/usr/tmp/");
```

```
    // a custom protocol for JDBC
    testProtocol("jdbc:mysql://luna.metalab.unc.edu:3306/NEWS");
    // rmi, a custom protocol for remote method invocation
    testProtocol("rmi://metalab.unc.edu/RenderEngine");
    // custom protocols for HotJava
    testProtocol("doc:/UsersGuide/release.html");
    testProtocol("netdoc:/UsersGuide/release.html");
    testProtocol("systemresource://www.adc.org/+/index.html");
    testProtocol("verbatim:http://www.adc.org/");
  }
  private static void testProtocol(String url) {
    try {
      URL u = new URL(url);
      System.out.println(u.getProtocol() + " is supported");
    }
    catch (MalformedURLException e) {
      String protocol = url.substring(0, url.indexOf(':'));
      System.out.println(protocol + " is not supported");
    }
  }
}
```

The results of this program depend on which virtual machine runs it. Students are required to understand this program and test run it on their own machines.

5.5.2. Constructing a URL from its Component Parts

The second constructor builds a URL from three strings specifying the protocol, the hostname, and the file:

public URL(String protocol, String hostname, String file) throws MalformedURLException

This constructor sets the port to –1 if default port for the protocol is used. Otherwise, a parameter “int port” is used after the hostname to specify the port explicitly.

The following example (named `ProtocolTesterApplet.java`) is an applet that uses the above constructor to test the protocols supported by a browser’s virtual machine.

```
import java.net.*;
import java.applet.*;
import java.awt.*;
public class ProtocolTesterApplet extends Applet {
  TextArea results = new TextArea();
  public void init() {
    this.setLayout(new BorderLayout());
    this.add("Center", results);
  }
  public void start() {
    String host = "www.peacefire.org";
    String file = "/bypass/SurfWatch/";
    String[] schemes = {"http",   "https",   "ftp",  "mailto",
                        "telnet", "file",    "ldap", "gopher",
                        "jdbc",   "rmi",     "jndi", "jar",
                        "doc",    "netdoc",  "nfs",  "verbatim",
                        "finger", "daytime", "systemresource"};
    for (int i = 0; i < schemes.length; i++) {
      try {
        URL u = new URL(schemes[i], host, file);
        results.append(schemes[i] + " is supported\r\n");
      }
```

```
      catch (MalformedURLException e) {
        results.append(schemes[i] + " is not supported\r\n");
      }
    }

  }
}
```

You also need the following HTML file (named `ProtocolTester.html`) to test the above program:

```
<HTML>
<title>Which schemes does this browser support?</title>
<body bgcolor=#ffffff text=#000000>
<H1>Which schemes does this browser support?</H1>
<APPLET width=400 height=300 code=ProtocolTesterApplet>
</APPLET>
</BODY>
</HTML>
```

5.5.3. Other URL Constructors and Methods

The third URL constructor creates a URL object using a relative URL and a base URL:

public URL(URL base, String relative) throws MalformedURLException

For example, the following code segment creates a new URL u2 from the base URL of u1 by removing the filename of u1 and appending the new filename specified in the constructor:

```
      try {
        URL u1 = new URL("http://www3.cm.deakin.edu.au/~wanlei/#hmteach");
        URL u2 = new URL(u1, "apweb99.htm"
      }
      catch (MalformedURLException e) {
        System.err.println(e);
      }
```

The following program, named as *RelativeURLTestt.java*, uses the *getDocumentBase*() (or *getCodeBase*()) method of the *java.applet.Applet* class to get the document base (or code base) relative to the applet, and then uses the above constructor to create a new URL relative to the document base.

```
import java.net.*;
import java.applet.*;
import java.awt.*;
public class RelativeURLTest extends Applet {
  public void init () {
    try {
      URL base = this.getDocumentBase();
      URL relative = new URL(base, "mailinglists.html");
      this.setLayout(new GridLayout(2,1));
      this.add(new Label(base.toString()));
      this.add(new Label(relative.toString()));
    }
    catch (MalformedURLException e) {
      this.add(new Label("This shouldn't happen!"));
    }
```

```
  }
}
```

The following HTML file (named *relativeURL.html*) is used to run the applet:

```
<HTML>
<title>Relative URL Test</title>
<body bgcolor=#ffffff text=#000000>
<APPLET width=250 height=100 code=RelativeURLTest>
</APPLET>
</BODY>
</HTML>
```

Java 2 adds two additional constructors that allow you to specify the protocol handler for the URL.

URLs can be split into five parts:

- The scheme, as known as protocol
- The authority
- The path
- The ref, also known as the section or named anchor
- The query string

Five public methods are provided for the read-only access to these parts: *getFile(), getHost(), getPort(), getProtocol(),* and *getRef()*. Java 1.3 adds four more methods: *getQuery(), getPath() getUserInfo(), and getAuthority()*. The text book has a thorough description of most of them. Here is an example:

```
try {
    URL u = new URL("http://www.deakin.edu.au/~wanlei/index.html#hre");
    System.out.println("The protocol is " + u.getProtocol());
    System.out.println("The host is " + u.getHost());
    System.out.println("The port is " + u.getPort());
    System.out.println("The file is " + u.getFile());
    System.out.println("The anchor is " + u.getRef());
  }
  catch (MalformedURLException e) {
}
```

If a port is not explicitly specified in the URL, it's set to -1. This does not mean that the connection is attempted on port -1 (which doesn't exist) but rather that the default port is to be used.

If the ref doesn't exist, it's just `null`, so watch out for `NullPointerExceptions`. Better yet, test to see that it's non-null before using it.

Finally if the file is left off completely, e.g. http://java.sun.com, then it's set to "/".

5.5.4. Retrieve Data from a URL

The URL class has three (four in Java 1.3) methods to retrieve data from a URL:

public final InputStream openStream() throws IOException

public final Object getContent() throws IOException

public final URLConnection openConnection() throws IOException

public final Object getContent(Class[] classes) throws IOException // Java 1.3 only

We give examples for the first two methods in this section. The *URLConnection* class will be discussed in the next section. The method for Java 1.3 is not discussed in this unit. The following program, named as `SourceViewer.java`, uses the *openStream*() method to download data from a URL reference. It reads a URL from the command line, opens an *InputStream* from the URL, chains the resulting *InputStream* to an *InputStreamReader* using the default encoding, and then uses *InputStreamReader*'s *read*() method to read successive characters from the file, each of which is printed on the screen.

```
import java.net.*;
import java.io.*;
public class SourceViewer {
  public static void main (String args[]) {
    if  (args.length > 0) {
      try {
        //Open the URL for reading
        URL u = new URL(args[0]);
        InputStream in = u.openStream();
        // buffer the input to increase performance
        in = new BufferedInputStream(in);
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(in);
        int c;
        while ((c = r.read()) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException e) {
        System.err.println(args[0] + " is not a parseable URL");
      }
      catch (IOException e) {
        System.err.println(e);
      }
    } //  end if
  } // end main
}  // end SourceViewer
```

The following program uses the *getContent()* method to download data referenced by a URL. This method retrieves the data referenced by a URL and tries to make it into some type of object. If the data is a text object, such as an ASCII or HTML file, the object returned is usually some sort of *InputStream*. If the data is an image, such as a GIF or a JPEG file, then the method usually returns a *java.awt.ImageProducer*. This method works by looking at the Content-type field in the MIME header of data it gets from the server. The program is listed below:

```
import java.net.*;
import java.io.*;
public class ContentGetter {
  public static void main (String[] args) {
    if  (args.length > 0) {
      //Open the URL for reading
      try {
```

```
        URL u = new URL(args[0]);
        try {
          Object o = u.getContent();
          System.out.println("I got a " + o.getClass().getName());
        } // end try
        catch (IOException e) {
          System.err.println(e);
        }
      } // end try
      catch (MalformedURLException e) {
        System.err.println(args[0] + " is not a parseable URL");
      }
    } //  end if
  } // end main
}  // end ContentGetter
```

Students are required to run the program by supplying different types of URL at the command line and verify the types returned by the program.

5.6. The URLConnection Class

5.6.1. Basic Principles

The *URLConnection* is an abstract class that represents an active connection to a resource specified by a URL. It has two purposes. The first is to provide more control over the interaction with a server than the URL class. Using a *URLConnection* class, you can inspect the MIME headers sent by an HTTP server and respond accordingly. You can adjust the MIME header fields used by the client request. You can use a *URLConnection* to download binary files. Finally, you can use a *URLConnection* to send data back to a web server with POST or PUT and other HTTP request methods. The second function of a *URLConnection* class is to be a part of Java's protocol handler mechanism, which also includes the *URLStreamHandler* class. This section discusses the first function.

A program that uses the *URLConnection* class directly has the following steps, although the program does not have to perform all the steps.

- Construct a URL object.
- Invoke the URL object's *openConnection*() method to retrieve a *URLConnection* object for that URL.
- Configure the *URLConnection*
- Reader the header fields.
- Get an input stream and read data.
- Get an output stream and write data.
- Close the connection.

5.6.2. A Simple URLConnection Example

The following program reads data from a server. It only uses four steps:

- Construct a URL object.
- Invoke the URL object's *openConnection*() method to retrieve a *URLConnection* object for that URL.
- Invoke the *URLConnection*'s *getInputStream*() method..
- Read from the input stream using the usual stream API.

```
import java.net.*;
import java.io.*;
public class SourceViewer2 {
  public static void main (String[] args) {
    if  (args.length > 0) {
      try {
        //Open the URLConnection for reading
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection();
        InputStream raw = uc.getInputStream();
        InputStream buffer = new BufferedInputStream(raw);
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(buffer);
        int c;
        while ((c = r.read()) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException e) {
        System.err.println(args[0] + " is not a parseable URL");
      }
      catch (IOException e) {
        System.err.println(e);
      }
    } //  end if
  } // end main
}  // end SourceViewer2
```

This program is almost the same as the one that uses the *URL* class. Actually the difference between *URL* and *URLConnection* classes are not apparent with just simple input streams as in the example. The biggest different between the two classes are:

- *URLConnection* provides access to the MIME header associated with an HTTP 1.0 response.
- *URLConnection* class lets you configure the request parameters.
- *URLConnection* class lets you write data to the server as well as read data from the server.

5.6.3. Dealing with the MIME Header

The following program downloads a binary file from the server and saves it on the local disk. It creates a *URLConnection* object and test the type of the file. The file is downloaded only if it is a binary file (*ContentType* is text). If the *ContentType* is missing or the *ContentLength*==-1, the program throws an *IOException.*

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

```
public class BinarySaver {
  public static void main (String args[]) {
    for (int i = 0; i < args.length; i++) {
      try {
        URL root = new URL(args[i]);
        saveBinaryFile(root);
      }
      catch (MalformedURLException e) {
        System.err.println(args[i] + " is not URL I understand.");
      }
      catch (IOException e) {
        System.err.println(e);
      }
    } // end for
  } // end main
  public static void saveBinaryFile(URL u) throws IOException {
    URLConnection uc = u.openConnection();
    String contentType = uc.getContentType();
    int contentLength = uc.getContentLength();
    if (contentType.startsWith("text/") || contentLength == -1 ) {
      throw new IOException("This is not a binary file.");
    }
    InputStream raw = uc.getInputStream();
    InputStream in  = new BufferedInputStream(raw);
    byte[] data = new byte[1024];
    int bytesRead = 0;
    int offset = 0;
    while (offset < contentLength) {
       bytesRead = in.read(data, offset, data.length-offset);
       if (bytesRead == -1) break;
       offset += bytesRead;
    }
    in.close();
    if (offset != contentLength) {
      throw new IOException("Only read " + offset
       + " bytes; Expected " + contentLength + " bytes");
    }
    String filename = u.getFile();
    filename = filename.substring(filename.lastIndexOf('/') + 1);
    FileOutputStream fout = new FileOutputStream(filename);
    fout.write(data);
    fout.flush();
    fout.close();
  }
} // end BinarySaver
```

A number of methods of the *URLConnection* class can be used to read the properties of the MIME header. Following is an example to print the content type, content length, content encoding, date of last modification, expiration date, and current date:

```
import java.net.*;
import java.io.*;
import java.util.*;
public class MIMEHeadersViewer {
  public static void main(String args[]) {
    for (int i=0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection();
        System.out.println("Content-type: " + uc.getContentType());
        System.out.println("Content-encoding: "
```

```
         + uc.getContentEncoding());
        System.out.println("Date: " + new Date(uc.getDate()));
        System.out.println("Last modified: "
         + new Date(uc.getLastModified()));
        System.out.println("Expiration date: "
         + new Date(uc.getExpiration()));
        System.out.println("Content-length: " + uc.getContentLength());
      }  // end try
      catch (MalformedURLException e) {
        System.err.println(args[i] + " is not a URL I understand");
      }
      catch (IOException e) {
        System.err.println(e);
      }
      System.out.println();
    }  // end for
  }  // end main
}  // end MIMEHeadersViewer
```

Students are required to run this program and view examine the returned parameters of various files.

A number of methods are also provided by the *URLConnection* class to retrieve arbitary MIME header fields. The following is an example that displays all header fields of a URL:

```
import java.net.*;
import java.io.*;
public class AllMIMEHeaders {
  public static void main(String args[]) {
    for (int i=0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection();
        for (int j = 0; ; j++) {
          String header = uc.getHeaderField(j);
          if (header == null) break;
          System.out.println(uc.getHeaderFieldKey(j) + ": " + header);
        }  // end for
      }  // end try
      catch (MalformedURLException e) {
        System.err.println(args[i] + " is not a URL I understand.");
      }
      catch (IOException e) {
        System.err.println(e);
      }
      System.out.println();
    }  // end for
  }  // end main
}  // end AllMIMEHeaders
```

5.6.4. The URLConnection Configuration

The *URLConnection* class has seven protected instance fields that define exactly how the client will make the request to the server. They are:

```
  protected URL     url;
  protected boolean doInput = true;
  protected boolean doOutput = false;
```

```
protected boolean allowUserInteraction = defaultAllowUserInteraction;
protected boolean useCaches = defaultUseCashes;
protected long    ifModifiedSince = 0;
protected boolean connected = false;
```

These fields are accessed and modified by using the obvious named setter and getter methods:

```
public URL        getURL();
public void       setDoInput(boolean doInput);
public boolean    getDoInput();
public void       setDoOutput(boolean doOutput);
public boolean    getDoOutput();
public void       setAllowUserInteraction(boolean allowUserInteraction);
public boolean    getAllowUserInteraction();
public void       setUseCaches(boolean useCaches);
public boolean    getUseCaches();
public void       setIfModifiedSince(long ifModifiedSince);
public boolean    getDoInput();
```

You can modify these fields only before the *URLConnection* is connected.

5.7. Handlers for Contents and Protocols

5.7.1. What are Content and Protocol Handlers

Java provides two mechanisms to make Web-based clients (such as browsers) extensible: handling protocols and handling contents. Handling a protocol means taking care of the interaction between a client and a server; generating requests in the correct format, interpreting the headers that come back with the data, acknowledging that the data has been received, etc. Handling the content means converting the raw data into a format Java understands.

Content handlers are implemented as subclasses of the ContentHandler class. A content handler is only required to implement a single method, the getContent()method, which overrides the method provided by the ContentHandler class. This method takes an URLConnection object as a parameter and returns an object of a specific MIME type.

The purpose of a content handler is to extract an object of a given MIME type from an URLConnection object's input stream. Content handlers are not directly instantiated or accessed. The getContent() methods of the URL and URLConnection classes cause content handlers to be created and invoked to perform their processing.

A content handler is associated with a specific MIME type through the use of the ContentHandlerFactory interface. A class that implements the ContentHandlerFactory interface must implement the createContentHandler() method. This method returns a ContentHandler object to be used for a specific MIME type. A ContentHandlerFactory object is installed using the static setContentHandlerFactory() method of the URLConnection class.

Protocol handlers are implemented as subclasses of the URLStreamHandler class. The URLStreamHandler class defines four access methods that can be overridden by its subclasses, but only the openConnection() method is required to be overridden.

The openConnection() method takes an URL with its assigned protocol as a parameter and returns an object of class URLConnection. The URLConnection object can then be used to create input and output streams and to access the resource addressed by the URL.

The parseURL() and setURL() methods are used to implement custom URL syntax parsing. The toExternalForm() method is used to convert an URL of the protocol type to a String object.

The purpose of a protocol handler is to implement a custom protocol needed to access Web objects identified by URLs that require the custom protocol. Protocol handlers, like content handlers, are not directly instantiated or accessed. The methods of the URLConnection object that is returned by a protocol handler are invoked to access the resource referenced by the protocol.

A protocol is identified beginning with the first character of the URL and continuing to the first colon (:) contained in the URL. For example, the protocol of the URL http://www.jaworski.com is http, and the protocol of the URL fortune:// jaworski.com is fortune.

A protocol handler is associated with a specific protocol through the use of the URLStreamHandlerFactory interface. A class that implements the URLStreamHandlerFactory interface must implement the createURLStreamHandler() method. This method returns an URLStreamHandler object to be used for a specific protocol. An URLStreamHandlerFactory object is installed using the static setURLStreamHandlerFactory() method of the URL class.

5.7.2. Developing Content and Protocol Handlers

The first step in implementing a content handler is to define the class of the object to be extracted by the content handler. The content handler is then defined as a subclass of the ContentHandler class. The getContent() method of the content handler performs the extraction of objects of a specific MIME type from the input stream associated with an URLConnection object.

A content handler is associated with a specific MIME type through the use of a ContentHandlerFactory object. The createContentHandler() method of the ContentHandlerFactory interface is used to return a content handler for a specific MIME type.

Finally, the setContentHandlerFactory() method of the URLConnection class is used to set a ContentHandlerFactory as the default ContentHandlerFactory to be used with all MIME types.

The first step in implementing a protocol handler is to define it as a subclass of the URLStreamHandler class. The openConnection() method of the protocol handler creates an URLConnection object that can be used to access an URL designating the specified protocol.

A protocol handler is associated with a specific protocol type through the use of an URLStreamHandlerFactory object. The createURLStreamHandler() method of the

URLStreamHandlerFactory interface is used to return a protocol handler for a specific protocol type.

The setURLStreamHandlerFactory() method of the URL class is used to set an URLStreamHandlerFactory as the default URLStreamHandlerFactory to be used with all protocol types.

When an URL object is created, the system determines the right protocol. The MalformedURL exception signals that an unparseable URL has been used in the URL constructor. The first time a protocol name is encountered when constructing a URL, the appropriate stream protocol handler is automatically loaded.

If everything is correct, a call to the getContent() method returns the content of the URL. As it's not possible to know the data type refered to by the URL (text, zip file, tar file,..), this method returns an Object. The programers must use a cast to convert it to the right object. For example:

```
URL u=new URL("zip://www.inside-java.com/test.zip");
zipFile z=(zipFile) u.getContent();
```

6. Developing Servers

6.1. Study Points

- Understand the principles of server-side networking.
- Be able to use the *ServerSocket* class.
- Understand the working of a number of example servers presented in the text and in this study guide.

Reference: (1). [JNP] Chapters 11. (2). [HSH] Chapters 16, 17. (3). [Java2U]: Chapter 32.

6.2. The *ServerSocket* Class

6.2.1 Basics of the ServerSocket Class

Servers are like receptionists who sit by the phone and wait for incoming calls. They do not know who will call and when, only that when the phone rings, they have to pick up the phone and answer to whoever is there. Java provides a *ServerSocket* class to allow programmers to write servers behaved like a receptionist.

Technically speaking, a Java *ServerSocket* runs on a server and listens on a particular port of the server machine for incoming TCP connections. When a client *Socket* on a remote host attempts to connect to the port, the server wakes up, negotiates the connection between the server and the client, and opens a regular *Socket* between the two hosts for the regular communication between the client and the server.

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

No more than one server socket can listen to a particular port at one time. Therefore, since a server may need to handle many connections at once, server programs tend to be heavily multi-threaded. Generally the server socket listening on the port will only accept the connections. It then passes off the actual processing of connections to a separate thread.

The *ServerSocket* contains everything you need to write a server in Java. There are three public *ServerSocket* constructors:

```
public ServerSocket(int port) throws IOException, BindException
public ServerSocket(int port, int queueLength) throws IOException,
  BindException
public ServerSocket(int port, int queueLength, InetAddress bindAddress)
  throws IOException
```

The operating system stores incoming connection requests addressed to a particular port in an FIFO queue. The default length of the queue is normally 50, though this can vary from operating system to operating system. Incoming connections are refused if the queue is already full. Managing incoming queues is done by the operating system. You can change the default queue length using the above constructors (up to the maximum length set by the operating system). On most systems the default queue length is between 5 and 50.

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

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

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

On Unix systems (but not Windows or the Mac) your program must be running as root to bind to a port between 1 and 1023.

0 is a special port number. It tells Java to pick an available port. You can then find out what port it's picked with the getLocalPort() method. This is useful if the client and the server have already established a separate channel of communication over which the chosen port number can be communicated.

The following program determines which ports of a local host are currently occupied by trying to create server sockets on all of them, and seeing where that operation fails.

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

public class LocalPortScanner {
  public static void main(String[] args) {
    // first test to see whether or not we can bind to ports
    // below 1024
    boolean rootaccess = false;
    for (int port = 1; port < 1024; port += 50) {
      try {
        ServerSocket ss = new ServerSocket(port);
        // if successful
        rootaccess = true;
        ss.close();
        break;
      }
      catch (IOException e) {
      }
    }
    int startport = 1;
```

```
    if (!rootaccess) startport = 1024;
    int stopport = 65535;
    for (int port = startport; port <= stopport; port++) {
      try {
        ServerSocket ss = new ServerSocket(port);
        ss.close();
      }
      catch (IOException e) {
        System.out.println("Port " + port + " is occupied.");
      }
    }
  }
}
```

6.2.2. An Example

The following example implements a simple daytime server, as per RFC 867. Since this server sends a single line of text in response to each connection, it processes each connection immediately. More complex servers should spawn a thread to handle each request. In this case, the overhead of spawning a thread would be greater than the time used to process the request.

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DaytimeServer {
  public final static int DEFAULT_PORT = 13;
  public static void main(String[] args) {
   int port = DEFAULT_PORT;
   if (args.length > 0) {
     try {
        port = Integer.parseInt(args[0]);
        if (port < 0 || port >= 65536) {
          System.out.println("Port must between 0 and 65535");
          return;
        }
     }
     catch (NumberFormatException e) {
       // use default port
     }
   }
   try {
     ServerSocket server = new ServerSocket(port);

     Socket connection = null;
     while (true) {
       try {
         connection = server.accept();
         OutputStreamWriter out
          = new OutputStreamWriter(connection.getOutputStream());
         Date now = new Date();
         out.write(now.toString() +"\r\n");
         out.flush();
         connection.close();
       }
       catch (IOException e) {}
       finally {
         try {
           if (connection != null) connection.close();
```

```
          }
          catch (IOException e) {}
        }
      }  // end while
    }  // end try
    catch (IOException e) {
      System.err.println(e);
    } // end catch
  } // end main
} // end DaytimeServer
```

Note that if you are running UNIX you cannot use port 13 if you are not the root. You can then change the port number to, say 1313. The following is the client program to use this Daytime server (the port number also needs to be changed to the same as the server's port number if necessary).

```
import java.net.*;
import java.io.*;
public class DaytimeClient {
  public static void main(String[] args) {
    String hostname;
    if (args.length > 0) {
      hostname = args[0];
    }
    else {
      hostname = "sky.cm.deakin.edu.au";
    }
    try {
      Socket theSocket = new Socket(hostname, 13);
      InputStream timeStream = theSocket.getInputStream();
      StringBuffer time = new StringBuffer();
      int c;
      while ((c = timeStream.read()) != -1) time.append((char) c);
      String timeString = time.toString().trim();
      System.out.println("It is " + timeString + " at " + hostname);
    }  // end try
    catch (UnknownHostException e) {
      System.err.println(e);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }  // end main
} // end DaytimeClient
```

6.2.3. Use Telnet to Testing Servers

You can also use the *telnet* program to test a server. By default, *telnet* connects to port 23. To connect to a different server and port, you specify the host name and the port number in the command line:

```
telnet localhost 13
```

This example requests a connection to the local host on port 13. If the Daytime server is executing on the local host, the response from the server will be displayed on the *telnet* window. You can replace `localhost` to the host's name or IP address of any server.

6.3. Issues in Building Servers

6.3.1. Reading Data

There are no getInputStream() or getOutputStream() methods for ServerSocket. Instead you use accept() to return a Socket object, and then call its getInputStream() or getOutputStream() methods. For example,

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

Notice in this example, I closed the Socket s, not the ServerSocket ss. ss is still bound to port 2345. You get a new socket for each connection but it's easy to reuse the server socket.

6.3.2. Writing Data

The following simple program repeatedly answers client requests by sending back the client's address and port. Then it closes the connection.

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

public class HelloServer {
  public final static int defaultPort = 2345;
  public static void main(String[] args) {
    int port = defaultPort;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
    }
    if (port <= 0 || port >= 65536) port = defaultPort;
    try {
      ServerSocket ss = new ServerSocket(port);
      while (true) {
        try {
          Socket s = ss.accept();
          PrintWriter pw = new PrintWriter(s.getOutputStream());
          pw.println("Hello " + s.getInetAddress() + " on port "
           + s.getPort());
          pw.println("This is " + s.getLocalAddress() + " on port "
           + s.getLocalPort());
          pw.flush();
          s.close();
        }
```

```
        catch (IOException e) {
        }
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

You can run this program on a host and then try to use Telnet to connect the host via port 2345 several times. You may then note how the port from which the connection comes changes each time.

6.3.3. Interacting with a Client

Commonly a server needs to both read a client request and write a response. The following program reads whatever the client sends and then sends it back to the client. In short this is an echo server.

```
import java.net.*;
import java.io.*;
public class EchoServer {

  public final static int defaultPort = 2346;
  public static void main(String[] args) {
    int port = defaultPort;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
    }
    if (port <= 0 || port >= 65536) port = defaultPort;
    try {
      ServerSocket ss = new ServerSocket(port);
      while (true) {
        try {
          Socket s = ss.accept();
          OutputStream os = s.getOutputStream();
          InputStream is = s.getInputStream();
          while (true) {
            int n = is.read();
            if (n == -1) break;
            os.write(n);
            os.flush();
          }
        }
        catch (IOException e) {
        }
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

6.3.4. Parallel Processing

The operation of most servers involves listening for connections, accepting connections, processing requests received over the connections, and terminating connections after all requests have been processed. The handling of multiple connections is generally performed using multiple threads. As such, a general framework for multithreaded servers can be shown as the following GenericServer.java program.

```
import java.net.*;
import java.io.*;
import java.util.*;
public class GenericServer {
    // Replace 1234 with the well-known port used by the server.
    int serverPort = 1234;
    public static void main(String args[]){
        // Create a server object and run it
        GenericServer server = new GenericServer();
        server.run();
    }
    public GenericServer() {
        super();
    }
    public void run() {
        try {
            // Create a server socket on the specified port
            ServerSocket server = new ServerSocket(serverPort);
            do {
                // Loop to accept incoming connections
                Socket client = server.accept();
                // Create a new thread to handle each connection
                (new ServerThread(client)).start();
            } while(true);
        } catch(IOException ex) {
            System.exit(0);
        }
    }
}
class ServerThread extends Thread {
    Socket client;
    // Store a reference to the socket to which the client is connected
    public ServerThread(Socket client) {
        this.client = client;
    }
    // Thread's entry point
    public void run() {
        try {
            // Create streams for communicating with client
            ServiceOutputStream outStream = new ServiceOutputStream(
                new BufferedOutputStream(client.getOutputStream()));
            ServiceInputStream inStream =
             new ServiceInputStream(client.getInputStream());
            // Read client's request from input stream
            ServiceRequest request = inStream.getRequest();
            // Process client's request and send output back to client
            while (processRequest(outStream)) {};
        }catch(IOException ex) {
            System.exit(0);
        }
        try {
            client.close();
```

```
        }catch(IOException ex) {
            System.exit(0);
        }
    }
    // Stub for request processing
    public boolean processRequest(ServiceOutputStream outStream) {
        return false;
    }
}
// Input stream filter
class ServiceInputStream extends FilterInputStream {
    public ServiceInputStream(InputStream in) {
        super(in);
    }
    // Method for reading client requests from input stream
    public ServiceRequest getRequest() throws IOException {
        ServiceRequest request = new ServiceRequest();
        return request;
    }
}
// Output stream filter
class ServiceOutputStream extends FilterOutputStream {
    public ServiceOutputStream(OutputStream out) {
        super(out);
    }
}
// Class to implement client requests
class ServiceRequest {
}
```

The GenericServer class is the main class of the program. It defines serverPort as the number of the port that the server is to listen on. You would change 1234 to the well-known port of the service that the server is to implement. The main() method creates an instance of GenericServer and invokes the instance's run() method. The run() method creates a new ServerSocket and assigns the socket to the server variable. It then executes an infinite loop where it listens for an incoming connection and creates a Socket instance to service the connection. The Socket instance is assigned to the client variable. A new ServerThread object is created to process client requests and the start() method of this object is invoked to get the thread up and running.

The ServerThread class extends the Thread class. It declares the client field variable to keep track of the client socket. Its run() method creates objects of class ServiceInputStream and ServiceOutputStream to communicate with the client. These streams are buffered to enhance I/O performance. A while statement is used to repeatedly invoke the processRequest() method to process client requests. If processRequest() returns a value of false, the service is completed, the while loop ends, and the client socket is closed. The processRequest() method is a stub for implementing service-specific request processing.

The ServiceInputStream, ServiceOutputStream, and ServiceRequest classes are placeholders for implementing client I/O and request processing.

6.4. Some Useful Servers

6.4.1. Testing Clients

As we mentioned in section 6.2.3, you can use *telnet* to test a server. However, there is no such an equivalent utility for testing clients. The following example is used for such a purpose. This program uses two threads to handle input and output simultaneously: one to handle input from a client and the other to send output from the server.

```
import java.net.*;
import java.io.*;
import com.macfaq.io.SafeBufferedReader;
public class ClientTester {
  public static void main(String[] args) {
    int port;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
      port = 0;
    }
    try {
      ServerSocket server = new ServerSocket(port, 1);
      System.out.println("Listening for connections on port "
       + server.getLocalPort());
      while (true) {
        Socket connection = server.accept();
        try {
          System.out.println("Connection established with "
           + connection);
          Thread input = new InputThread(connection.getInputStream());
          input.start();
          Thread output
           = new OutputThread(connection.getOutputStream());
          output.start();
          // wait for output and input to finish
          try {
            input.join();
            output.join();
          }
          catch (InterruptedException e) {
          }
        }
        catch (IOException e) {
          System.err.println(e);
        }
        finally {
          try {
            if (connection != null) connection.close();
          }
          catch (IOException e) {}
        }
      }
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
class InputThread extends Thread {
```

```
  InputStream in;
   public InputThread(InputStream in) {
     this.in = in;
   }
   public void run()  {
     try {
       while (true) {
         int i = in.read();
         if (i == -1) break;
         System.out.write(i);
       }
     }
     catch (SocketException e) {
       // output thread closed the socket
     }
     catch (IOException e) {
       System.err.println(e);
     }
     try {
       in.close();
     }
     catch (IOException e) {
     }
  }
}
class OutputThread extends Thread {
  Writer out;
  public OutputThread(OutputStream out) {
    this.out = new OutputStreamWriter(out);
  }
  public void run() {
    String line;
    BufferedReader in
     = new SafeBufferedReader(new InputStreamReader(System.in));
    try {
      while (true) {
        line = in.readLine();
        if (line.equals(".")) break;
        out.write(line +"\r\n");
        out.flush();
       }
    }
    catch (IOException e) {
    }
    try {
      out.close();
    }
    catch (IOException e) {
    }
   }
}
```

The program uses a special reader, the *SafeBufferedReader*, defined in the program *SafeBufferedReader.java*, to read lines from an *InputStreamReader*. The program is listed as follows. You need to compile it using the *"-d"* option of the *javac* compiler to save the compiled package into the right class path.

```
package com.macfaq.io;
import java.io.*;
public class SafeBufferedReader extends BufferedReader {
  public SafeBufferedReader(Reader in) {
```

```
    this(in, 1024);
  }
  public SafeBufferedReader(Reader in, int bufferSize) {
    super(in, bufferSize);
  }
  private boolean lookingForLineFeed = false;
  public String readLine() throws IOException {
    StringBuffer sb = new StringBuffer("");
    while (true) {
      int c = this.read();
      if (c == -1) { // end of stream
        return null;
      }
      else if (c == '\n') {
        if (lookingForLineFeed) {
          lookingForLineFeed = false;
          continue;
        }
        else {
          return sb.toString();
        }
      }
      else if (c == '\r') {
        lookingForLineFeed = true;
        return sb.toString();
      }
      else {
        lookingForLineFeed = false;
        sb.append((char) c);
      }
    }
  }
}
```

You may test the program by running the *ClientTester* server on your local host and listens to a port. For example, the following command runs the server on port 80, which is the default port of a web server:

```
Java ClientTester 80
```

You then can use a web browser to connect to the server. The server will then output the testing result for the web browser. Commands can be sent from the server to the client via the server window. The “.” command will end the connection.

6.4.2. Building a Web Server

The textbook describes a number of steps and examples in building a web server. It has a brief introduction on the HTTP protocol and the requests and responses of the protocol. Then a complete discussion on a web server is presented in the text (pages 369-387. Students are encouraged to understand the principles presented in this chapter and test run the examples given in the chapter.

7. Connectionless Communication in Java

7.1. Study Points

- Understand the difference of datagram and TCP communication mechanisms.
- Understand the *DatagramSocket* and the *DatagramPacket* clases.
- Be able to use datagrams to complete simple Java communication programs.
- Understand the application programs presented in the textbook and this study guide.

Reference: (1). [JNP]: Chapter 13. (2). [HSH] Chapters 20, 21.

7.2. Connectionless Communication Basics

For many applications the convenience of the TCP sockets outweighs the overhead required. However, for certain applications it is much more efficient to utilize connectionless communications via datagrams: small, fixed-length messages sent between computers of a network.

7.2.1. Why Datagrams

A TCP connection carries a number of overhead. First, one needs to go through several steps to open a connection. This takes certain time. Once a connection is open, sending and receiving data each involving several steps. The final step is to tear down the connection after the communication. If one is to send large amount of data that must be reliably delivered, then the TCP protocol is suitable. However, if one only needs to send a short, simple message quickly, then all these steps may not be worthwhile.

The difference between datagram and TCP connections is like the difference between pagers and telephones. With a telephone, you make a connection to a specific telephone number, if the person on the destination answers the phone, you two are able to talk for certain period of time, exchanging arbitrary amount of information, and then you close the connction. With a pager, you typically send a message via a radio tower one-way to a tiny radio receiver. Because of broadcast difficulties and delays you cannot be certain if or when the paged party receives the message. The only way to tell is if you request and receive some kind of acknowledgement. You may retry several times if you get no response, then give up.

On an IP network such a the Internet, the UDP (Unreliable Datagram Protocol) is used to transmit fixed-length datagrams. This is the protocol that Java taps with the *DatagramSocket* class. Protocols that use UDP include NFS, FSP, and TFTP.

7.2.2. Overview of Datagrams

Datagrams have the following advantages:

- Speed. UDP involves low overhead since there is no need to set up the connection, to maintain the order and correctness of the message delivery, and to tear down the connection after the communication.

- Message-oriented instead of stream-oriented. If the message to be sent is small and simple, it may be easier to simply send the chunk of bytes instead of going through the steps of converting it to and from streams.

Two *java.net* classes define the heart of datagram-based messaging in Java, the *DatagramSocket* and the *DatagramPacket*.

The *DatagramSocket* is the interface through which *DatagramPacket* are transmitted. A *DatagramPacket* is simply an IP-specific wrapper for a block of data.

The *DatagramSocket* class provides a good interface to the UDP protocol. This class is responsible for sending and receiving *DatagramPacket* via the UDP protocol. The most commonly used *DatagramSocket* methods are listed below:

- *DatagramSocket*(). Constructor comes in two formats: one is used to specify the local port used and the other the system picks an ephemeral local port for you.
- *receive*(). Receive a *DatagramPacket* from any remote server.
- *send*(). Send a *DatagramPacket* to the remote server specified in the *DatagramPacket*.
- *close*(). Tear down local communication resources. After this method has been called, release this object.
- *getLocalPost*(). Returns the local port this *DatagramSocket* is using.

Note that there are two flavors of *DatagramSocket*: those create to send *DatagramPackets*, and those created to receive *DatagramPackets*. A "send" *DatagramSocket* uses an ephemeral local port assigned by the native UDP implementation. A "receive" *DatagramSockect* requires a specific local port number.

A *DatagramPacket* represents the datagram transmitted via a *DatagramSocket*. The most frequently used methods of *DatagramPacket* are:

- *DatagramPacket*(). Constructor comes in two formats: a "send" packet and a "receive" packet. For the *send* packet, you need to specify a remote *InetAddress* and a port to which the packet should be sent, as well as a data buffer and length to be sent. For the *receive* packet, you need to provide an empty buffer into which data should be stored, and the maximum number of bytes to be stored.
- *getAddress*(). This method allows one to either obtain the *InetAddress* of the host that sent the *DatagramPacket*, or to obtain the *InetAddress* of the host to which this packet is addressed.
- *getData*(). This method allows one to access the raw binary data wrapped in the *DatagramPacket*.
- *getLength*(). This method allows one to determine the length of data wrapped in the *DatagramPacket* without getting a reference to the data block itself.
- *getPort*(). This method returns either the port of the server to which this packet will be sent, else it returns the port of the server that sent this packet, depending on whether the packet was built to be sent or built to receive data.

It is also possible to exchange data via datagrams using the *Socket* class. To do so, you must use one of the *Socket* constructors that includes the Boolean *useStream* parameter, as in,

```
Socket(InetAddress address, int port, Boolean useStream)
```

And set *useStream* to *false*. This tells *Socket* to use the faster UDP. The advantage to use this interface is that it provides a stream interface to datagram. Also, there is no need to instantiate and maintain a separate *DatagramPacket* to hold the data.

But there are significant disadvantages as well. First, there is no way to detect if a particular datagram sent does not arrive to the destination. Your stream interface can lie to you. Second, you still have to go through the hassle of setting up the connection.

UDP ports are separate from TCP ports. Each computer has 65,536 UDP ports as well as its 65,536 TCP ports. You can have a ServerSocket bound to TCP port 20 at the same time as a DatagramSocket is bound to UDP port 20. Most of the time it should be obvious from context whether or not I'm talking about TCP ports or UDP ports.

7.2.3. A Local Port Scanner

The LocalPortScanner developed in Section 6.2.1 only found TCP ports. The following program detects UDP ports in use. As with TCP ports, you must be root on Unix systems to bind to ports below 1024.

```
import java.net.*;
import java.io.IOException;
public class UDPPortScanner {
  public static void main(String[] args) {
    // first test to see whether or not we can bind to ports
    // below 1024
    boolean rootaccess = false;
    for (int port = 1; port < 1024; port += 50) {
      try {
        ServerSocket ss = new ServerSocket(port);
        // if successful
        rootaccess = true;
        ss.close();
        break;
      }
      catch (IOException e) {
      }
    }
    int startport = 1;
    if (!rootaccess) startport = 1024;
    int stopport = 65535;
    for (int port = startport; port <= stopport; port++) {
      try {
        DatagramSocket ds = new DatagramSocket(port);
        ds.close();
      }
      catch (IOException e) {
        System.out.println("UDP Port " + port + " is occupied.");
      }
    }
  }
}
```

Since UDP is connectionless it is not possible to write a remote UDP port scanner. The only way you know whether or not a UDP server is listening on a remote port is if it sends something back to you.

7.2.4. Sending and Receiving UDP Datagrams

To send data to a particular server, you first must convert the data into byte array. Next you pass this byte array, the length of the data in the array (most of the time this will be the length of the array) and the InetAddress and port to which you wish to send it into the DatagramPacket() constructor. For example,

```
try {
  InetAddress turin = new InetAddess("turin.cm.deakin.edu.au");
  int chargen = 19;
  String s = "My second UDP Packet";
  byte[] b = s.getBytes();
  DatagramPacket dp = new DatagramPacket(b, b.length, turin, chargen);
}
catch (UnknownHostException e) {
  System.err.println(e);
}
```

Next you create a DatagramSocket object and pass the packet to its send() method: For example,

```
try {
  DatagramSocket sender = new DatagramSocket();
  sender.send(dp);
}
catch (IOException e) {
  System.err.println(e);
}
```

To receive data sent to you, you construct a DatagramSocket object on the port on which you want to listen. Then you pass an empty DatagramPacket object to the DatagramSocket's receive() method.

public synchronized void receive(DatagramPacket dp) throws IOException

The calling thread blocks until the a datagram is received. Then dp is filled with the data from that datagram. You can then use getPort() and and getAddress() to tell where the packet came from, getData() to retrieve the data, and getLength() to see how many bytes were in the data. If the received packet was too long for the buffer, then it's truncated to the length of the buffer. For example,

```
try {
  byte buffer = new byte[65536]; // maximum size of an IP packet
  DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2134);
  ds.receive(dp);
  byte[] data = dp.getData();
```

```
  String s = new String(data, 0, data.getLength());
  System.out.println("Port " + dp.getPort() + " on " + dp.getAddress()
   + " sent this message:");
  System.out.println(s);
}
catch (IOException e) {
  System.err.println(e);
}
```

7.3. Simple Examples of Connectionless Communication

7.3.1. The Server

The steps for setting up a datagram server is as follows:

- Create a *DatagramPacket* for receiving the data, indicating the buffer to hold the data and the maximum length of the buffer.
- Create a *DatagramSocket* to listen to on.
- Receive a packet from the client.

Here is a simple server example. The program is named as *DatagramReceive.java*.

```
import java.net.*;

public class DatagramReceive {
  static final int PORT = 7890;
  public static void main( String args[] ) throws Exception {
    String theReceiveString;
    byte[] theReceiveBuffer = new byte[ 2048 ];

    // Make a packet to receive into...
    DatagramPacket theReceivePacket =
      new DatagramPacket( theReceiveBuffer, theReceiveBuffer.length );

    // Make a socket to listen on...
    DatagramSocket theReceiveSocket = new DatagramSocket( PORT );

    // Receive a packet...
    theReceiveSocket.receive( theReceivePacket );

    // Convert the packet to a string...
    theReceiveString =
      new String( theReceiveBuffer, 0, theReceivePacket.getLength() );

    // Print out the string...
    System.out.println( theReceiveString );

    //  Close the socket...
    theReceiveSocket.close();
    }
}
```

7.3.2. The Client

The steps of setting up a datagram client is as follows:

- Find the destination's IP address.
- Create a *DatagramPacket* based on the destination address and the data to be sent.
- Create a *DatagramSocket* for sending the packet.
- Send the *DatagramPacket* over the *DatagramSocket*.

Here is the program named DatagramSend.java:

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

public class DatagramSend {
  static final int PORT = 7890;
  public static void main( String args[] ) throws Exception {
    String theStringToSend = "I'm a datagram and I'm O.K.";
    byte[] theByteArray = new byte[ theStringToSend.length() ];
    theByteArray = theStringToSend.getBytes();

    // Get the IP address of our destination...
    InetAddress theIPAddress = null;
    try {
      theIPAddress = InetAddress.getByName( "localhost" );
    } catch (UnknownHostException e) {
      System.out.println("Host not found: " + e);
      System.exit(1);
    }

    // Build the packet...
    DatagramPacket thePacket = new DatagramPacket( theByteArray,
      theStringToSend.length(),
      theIPAddress,
      PORT );

    // Now send the packet
    DatagramSocket theSocket = null;
    try {
      theSocket = new DatagramSocket();
    } catch (SocketException e) {
        System.out.println("Underlying network software has failed: " + e);
        System.exit(1);
    }
    try {
      theSocket.send( thePacket );
    } catch (IOException e) {
      System.out.println("IO Exception: " + e);
    }
    theSocket.close();
    }
}
```

7.3.3. A Time Server Application

The TimeServerApp.java and GetTimeApp.java programs illustrate further the use of client/server computing using datagrams. TimeServerApp listens on a UDP socket on port 2345 for incoming datagrams. When a datagram is received, it displays the data contained in the datagram to the console window and returns a datagram with the current

date and time to the sending client program. It terminates its operation when it receives a datagram with the text quit as its data.

The GetTimeApp.java program sends five datagrams with the text time in each datagram to local port 2345. After sending each datagram, it waits for a return datagram from TimeServerApp. It displays the datagrams that it sends and receives to the console window. It then sends a quit datagram to TimeServerApp and terminates its operation.

The TimeServerApp.java program is shown below:

```
import java.lang.System;
import java.net.DatagramSocket;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.io.IOException;
import java.util.Date;
public class TimeServerApp {
 public static void main(String args[]){
  try{
   DatagramSocket socket = new DatagramSocket(2345);
   String localAddress = InetAddress.getLocalHost().getHostName().trim();
   int localPort = socket.getLocalPort();
   System.out.print(localAddress+": ");
   System.out.println("Time Server is listening on port "+localPort+".");
   int bufferLength = 256;
   byte outBuffer[];
   byte inBuffer[] = new byte[bufferLength];
   DatagramPacket outDatagram;
   DatagramPacket inDatagram =
     new DatagramPacket(inBuffer,inBuffer.length);
   boolean finished = false;
   do {
    socket.receive(inDatagram);
    InetAddress destAddress = inDatagram.getAddress();
    String destHost = destAddress.getHostName().trim();
    int destPort = inDatagram.getPort();
    System.out.println("\nReceived a datagram from "+destHost+" at port "+
     destPort+".");
    String data = new String(inDatagram.getData()).trim();
    System.out.println("It contained the data: "+data);
    if(data.equalsIgnoreCase("quit")) finished=true;
    String time = new Date().toString();
    outBuffer=time.getBytes();
    outDatagram =
      new DatagramPacket(outBuffer,outBuffer.length,destAddress,
    destPort);
    socket.send(outDatagram);
    System.out.println("Sent "+time+" to "+destHost+" at port "+
      destPort+".");
  } while(!finished);
  }catch (IOException ex){
   System.out.println("IOException occurred.");
  }
 }
}
```

The code for GetTimeApp.java is listed below:

```
import java.lang.System;
import java.net.DatagramSocket;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.io.IOException;
public class GetTimeApp {
 public static void main(String args[]){
  try{
   DatagramSocket socket = new DatagramSocket();
   InetAddress localAddress = InetAddress.getLocalHost();
   String localHost = localAddress.getHostName();
   int bufferLength = 256;
   byte outBuffer[];
   byte inBuffer[] = new byte[bufferLength];
   DatagramPacket outDatagram;
   DatagramPacket inDatagram =
     new DatagramPacket(inBuffer,inBuffer.length);
    for(int i=0;i<5;++i){
    outBuffer = new byte[bufferLength];
    outBuffer = "time".getBytes();
    outDatagram = new DatagramPacket(outBuffer,outBuffer.length,
     localAddress,2345);
    socket.send(outDatagram);
    System.out.println("\nSent time request to "+localHost+
      " at port 2345.");
    socket.receive(inDatagram);
    InetAddress destAddress = inDatagram.getAddress();
    String destHost = destAddress.getHostName().trim();
    int destPort = inDatagram.getPort();
    System.out.println("Received a datagram from "+destHost+" at port "+
     destPort+".");
    String data = new String(inDatagram.getData());
    data=data.trim();
    System.out.println("It contained the following data: "+data);
   }
   outBuffer = new byte[bufferLength];
   outBuffer = "quit".getBytes();
   outDatagram = new DatagramPacket(outBuffer,outBuffer.length,
    localAddress,2345);
   socket.send(outDatagram);
  }catch (IOException ex){
   System.out.println("IOException occurred.");
  }
 }
}
```

TimeServerApp and GetTimeApp should be run in separate windows. These two simple programs illustrate the basic mechanics of datagram-based client/ server applications. A UDP client sends a datagram to a UDP server at the server's port address. The UDP server listens on its port for a datagram, processes the datagram, and sends back information to the UDP client.

7.4. Some Datagram Applications

This section is to study some example datagram applications. The first example uses an alarm to trigger a resend of packets if no response is received to a transmission. The assumption is that the server should send back a reply within a certain period of time. If the response is not received within the time, then the packet is lost. The second example

is a Ping client, allowing a user to determine whether a remote host is alive, and so to determine the round-trip time of packets sent to the host.

7.4.1. A Reliable UDP Packet Delivery Example

The server is a simple echo server, with some code for simulating the lost of packets. In reality, a UPD packet may be discarded if the network is congested. This might be common on the Internet, but is very rare in a LAN.

The server program is named as *UDPEchoServer.java*. It creates a UDP socket, then waits for packets, echoing back what it receives. We simulate 10 percent packet loss to demonstrate the function of the program. Although UDP may drop packets naturally due to network congestions, this is not commonly observable on a small LAN.

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

public class UDPEchoServer {
  protected int port;

  public UDPEchoServer (int port) {
    this.port = port;
  }

  public void execute () throws IOException {
    DatagramSocket socket = new DatagramSocket (port);
    while (true) {
      DatagramPacket packet = receive (socket);
      if (Math.random () < .9) {
        sendEcho (socket, packet);
      } else {
        System.out.println ("Dropped!");
      }
    }
  }

  protected DatagramPacket receive (DatagramSocket socket) throws IOException {
    byte buffer[] = new byte[65508];
    DatagramPacket packet = new DatagramPacket (buffer, buffer.length);
    socket.receive (packet);
    return packet;
  }

  protected void sendEcho (DatagramSocket socket, DatagramPacket packet) throws
IOException {
    DatagramPacket response = new DatagramPacket
      (packet.getData (), packet.getLength (),
       packet.getAddress (), packet.getPort ());
    socket.send (response);
  }

  public static void main (String[] args) throws IOException {
    if (args.length != 1)
      throw new IllegalArgumentException ("Syntax: UDPEchoServer <port>");
    UDPEchoServer server = new UDPEchoServer (Integer.parseInt (args[0]));
    server.execute ();
  }
}
```

The client program is named as SureDelivery.java. This program attempts to protect against packet loss by resending a request if no response is received within a certain time-out (10 seconds in this example).

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

public class SureDelivery implements Alarmable {
  protected DatagramSocket socket;
  protected DatagramPacket packet;
  protected Alarm alarm;

  public SureDelivery (String message, String host, int port)
    throws IOException {
    socket = new DatagramSocket ();
    buildPacket (message, host, port);
    try {
      sendPacket ();
      receivePacket ();
    } finally {
      alarm.stop ();
      socket.close ();
    }
  }

  protected void buildPacket (String message, String host, int port) throws
IOException {
    ByteArrayOutputStream byteArrayOut = new ByteArrayOutputStream ();
    DataOutputStream dataOut = new DataOutputStream (byteArrayOut);
    dataOut.writeUTF (message);
    byte[] data = byteArrayOut.toByteArray ();
    packet = new DatagramPacket (data, data.length, InetAddress.getByName
(host), port);
  }

  protected void sendPacket () throws IOException {
    socket.send (packet);
    System.out.println ("Sent packet.");
    alarm = new Alarm (10000, this);
    alarm.start ();
  }

  protected boolean received;

  protected void receivePacket () throws IOException {
    byte buffer[] = new byte[65508];
    DatagramPacket packet = new DatagramPacket (buffer, buffer.length);
    socket.receive (packet);
    received = true;
    ByteArrayInputStream byteArrayIn =
      new ByteArrayInputStream (packet.getData (), 0, packet.getLength ());
    DataInputStream dataIn = new DataInputStream (byteArrayIn);
    String result = dataIn.readUTF ();
    System.out.println ("Received " + result + ".");
  }

  public synchronized void alarmCall (Object object) {
    try {
      System.out.println ("Alarm!");
```

```
      if (!received)
        sendPacket ();
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
  }

  public static void main (String[] args) throws InterruptedException,
IOException {
    if (args.length != 3)
      throw new IllegalArgumentException
        ("Syntax: SureDelivery <host> <port> <message>");
    while (true) {
      new SureDelivery (args[2], args[0], Integer.parseInt (args[1]));
      System.out.println ("Pause...");
      Thread.sleep (2000);
    }
  }
}
```

The time-out is performed by the *Alarm* class of the following *Alarm.java* program. This class allows us to schedule a callback to occur after a specified delay.

```
public class Alarm implements Runnable {
  public Alarm (int time, Alarmable target) {
    this (time, target, null);
  }

  protected Alarmable target;
  protected Object arg;
  protected int time;
  public Alarm (int time, Alarmable target, Object arg) {
    this.time = time;
    this.target = target;
    this.arg = arg;
  }

  protected Thread alarm;
  public synchronized void start () {
    if (alarm == null) {
      alarm = new Thread (this);
      alarm.start ();
    }
  }

  public synchronized void stop () {
    if (alarm != null) {
      alarm.interrupt ();
      alarm = null;
    }
  }

  public void run () {
    try {
      Thread.sleep (time);
      synchronized (this) {
        if (Thread.interrupted ())
          return;
        alarm = null;
      }
```

```
      target.alarmCall (arg);
    } catch (InterruptedException ignored) {
    }
  }
}
```

An object that wishes to receive alarm calls must implement an interface called *Alarmable*, defined in the following *Alarmable.java* program. An Alarmable object is thus any object that might receive an alarm call to signal an event. In this program, we use this mechanism to resend lost packets.

```
public interface Alarmable {
  public void alarmCall (Object arg);
}
```

To test this program, you need to run the server program first and specify a port for accepting client calls:

```
>java UDPEchoServer 2345
```

Then the client program can be executed. The host, port, and message are to be specified in the command line. For example, if the server runs on a computer called "turin.cm.deakin.edu.au", then the execution of the client could be:

```
>java SureDelivery turin.cm.deakin.edu.au 2345 "This is a test message"
```

7.4.2. A Ping Client

The ping protocol (RFC 862) is very commonly used in getting the liveness information of a remote node. For each host, if it is alive, it has a simple echo server running on port 7 to simply echo back any packets that are received. The application then sends a packet to port 7 of the remote machine in question. If it responses, then it is alive. Note that not all machines support this service, so sometimes ping will not response even the remote node is alive. Also, some firewalls may block these echo packets before they arrive the destination. However, if you know a server is running on a port (above 1024), you may test the Ping client by specifying the port number in the command line. The Ping command has the following format:

```
Ping [-c <count>] [-I <wait>] [-s <packetsize>] [-f] <hostname>[:<port>]
```

The following *Ping.java* program implements the *ping* command.

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

```
public class Ping {
  static final int DEFAULT_PORT = 7;
  static final int UDP_HEADER = 20 + 8;
  static final int BACKSPACE = 8;

  public static void main (String[] args) throws IOException {
    parseArgs (args);
    init ();
    for (int i = 0; (i < count) || (count == 0); ++ i) {
      long past = System.currentTimeMillis ();
      ping (i, past);
      try {
        pong (i, past);
      } catch (InterruptedIOException ignored) {
      }
    }
    socket.close ();
    printStats ();
  }

  static String host = null;
  static int port, count = 32, delay = 1000, size = 64;
  static boolean flood = false;

  static void parseArgs (String args[]) {
    for (int i = 0; i < args.length; ++ i) {
      if (args[i].startsWith ("-")) {
        if (args[i].equals ("-c") && (i < args.length - 1))
          count = Integer.parseInt (args[++ i]);
        else if (args[i].equals ("-i") && (i < args.length - 1))
          delay = Math.max (10, Integer.parseInt (args[++ i]));
        else if (args[i].equals ("-s") && (i < args.length - 1))
          size = Integer.parseInt (args[++ i]);
        else if (args[i].equals ("-f"))
          flood = true;
        else
          syntaxError ();
      } else {
        if (host != null)
          syntaxError ();
        int colon = args[i].indexOf (":");
        host = (colon > -1) ? args[i].substring (0, colon) : args[i];
        port = ((colon > -1) && (colon < args[i].length () - 1)) ?
          Integer.parseInt (args[i].substring (colon + 1)) : DEFAULT_PORT;
      }
    }
    if (host == null)
      syntaxError ();
  }

  static void syntaxError () {
    throw new IllegalArgumentException
      ("Ping [-c count] [-i wait] [-s packetsize] [-f] <hostname>[:<port>]");
  }

  static DatagramSocket socket;
  static byte[] outBuffer, inBuffer;
  static DatagramPacket outPacket, inPacket;

  static void init () throws IOException {
    socket = new DatagramSocket ();
    outBuffer = new byte[Math.max (12, size - UDP_HEADER)];
```

```
    outPacket = new DatagramPacket (outBuffer, outBuffer.length,
                                    InetAddress.getByName (host), port);
    inBuffer = new byte[outBuffer.length];
    inPacket = new DatagramPacket (inBuffer, inBuffer.length);
  }

  static int sent = 0;

  static void ping (int seq, long past) throws IOException {
    writeInt (seq, outBuffer, 0);
    writeLong (past, outBuffer, 4);
    socket.send (outPacket);
    ++ sent;
    if (flood) {
      System.out.write ('.');
      System.out.flush ();
    }
  }

  static final void writeInt (int datum, byte[] dst, int offset) {
    dst[offset] = (byte) (datum >> 24);
    dst[offset + 1] = (byte) (datum >> 16);
    dst[offset + 2] = (byte) (datum >> 8);
    dst[offset + 3] = (byte) datum;
  }

  static final void writeLong (long datum, byte[] dst, int offset) {
    writeInt ((int) (datum >> 32), dst, offset);
    writeInt ((int) datum, dst, offset + 4);
  }

  static int received = 0;

  static void pong (int seq, long past) throws IOException {
    long present = System.currentTimeMillis ();
    int tmpRTT = (maxRTT == 0) ? 500 : (int) maxRTT * 2;
    int wait = Math.max (delay, (seq == count - 1) ? tmpRTT : 0);
    do {
      socket.setSoTimeout (Math.max (1, wait - (int) (present - past)));
      socket.receive (inPacket);
      ++ received;
      present = System.currentTimeMillis ();
      processPong (present);
    } while ((present - past < wait) && !flood);
  }

  static long minRTT = 100000, maxRTT = 0, totRTT = 0;

  static void processPong (long present) {
    int seq = readInt (inBuffer, 0);
    long when = readLong (inBuffer, 4);
    long rtt = present - when;
    if (!flood) {
      System.out.println
        ((inPacket.getLength () + UDP_HEADER) +
         " bytes from " + inPacket.getAddress ().getHostName () +
         ": seq no " + seq + " time=" + rtt + " ms");
    } else {
      System.out.write (BACKSPACE);
      System.out.flush ();
    }
    if (rtt < minRTT) minRTT = rtt;
    if (rtt > maxRTT) maxRTT = rtt;
```

```
    totRTT += rtt;
  }

  static final int readInt (byte[] src, int offset) {
    return (src[offset] << 24) | ((src[offset + 1] & 0xff) << 16) |
      ((src[offset + 2] & 0xff) << 8) | (src[offset + 3] & 0xff);
  }

  static final long readLong (byte[] src, int offset) {
    return ((long) readInt (src, offset) << 32) |
      ((long) readInt (src, offset + 4) & 0xffffffffL);
  }

  static void printStats () {
    System.out.println
      (sent + " packets transmitted, " + received + " packets received, " +
       (100 * (sent - received) / sent) + "% packet loss");
    if (received > 0)
      System.out.println ("round-trip min/avg/max = " + minRTT + '/' +
                          ((float) totRTT / received) + '/' + maxRTT + " ms");
  }
}
```

To test the program, we use the host of smeagol.cm.deakin.edu.au to run the UDPEchoServer server using port 2345. The test would result in:

```
>java Ping –c 6 smeagol.cm.deakin.edu.au:2345
64 bytes from smeagol.cm.deakin.edu.au: seq no 0 time=60 ms
64 bytes from smeagol.cm.deakin.edu.au: seq no 2 time=0 ms
64 bytes from smeagol.cm.deakin.edu.au: seq no 3 time=0 ms
64 bytes from smeagol.cm.deakin.edu.au: seq no 5 time=0 ms
6 packets transmitted, 4 packets received, 33% packet loss
round-trip min/avg/max = 0/15.0/60 ms
```

8. Parallel Processing in Java

8.1. Study Points

- Understand the principles of parallel processing.
- Understand the basic concepts of Java threads.
- Be able to create parallel client-server Internet applications using Java threads.

Reference: (1). [JNP]: Chapter 5. (2). [HSH] Chapter 4. (3). [Java2H]: chapter 7.

8.2. Parallel Processing

8.2.1. Concurrency vs. Parallelism.

Let us differentiate concurrency from parallelism first. Concurrent multithreading systems give the appearance of several tasks executing at once, but these tasks are actually split up into chunks that share the processor with chunks from other tasks. The following figure (Figure 8.1) illustrates the issues. In parallel systems, two tasks are actually performed simultaneously. Parallelism requires a multiple-CPU system.

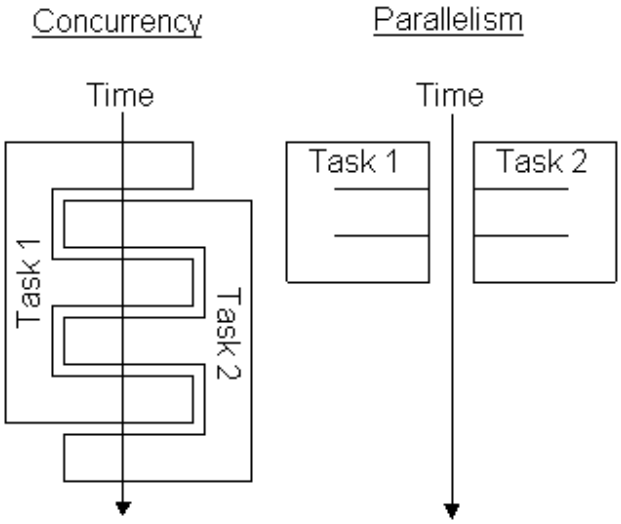

Figure 8.1. Concurrency vs. parallelism

Unless you're spending a lot of time blocked, waiting for I/O operations to complete, a program that uses multiple concurrent threads will often run slower than an equivalent single-threaded program, although it will often be better organized than the equivalent single-thread version. A program that uses multiple threads running in parallel on multiple processors will run much faster.

8.2.2. Thread Cooperation

There are typically two threading models supported by operating systems: cooperative and preemptive.

The cooperative multithreading model: In a cooperative system, a thread retains control of its processor until it decides to give it up (which might be never). The various threads have to cooperate with each other or all but one of the threads will be "starved" (meaning, never given a chance to run). Scheduling in most cooperative systems is done strictly by priority level. When the current thread gives up control, the highest-priority waiting thread gets control. (An exception to this rule is Windows 3.x, which uses a cooperative model but doesn't have much of a scheduler. The window that has the focus gets control.)

The main advantage of cooperative multithreading is that it's very fast and has a very low overhead. For example, a context swap -- a transfer of control from one thread to another -- can be performed entirely by a user-mode subroutine library without entering the OS kernel. (In NT, which is something of a worst-case, entering the kernel wastes 600 machine cycles. A user-mode context swap in a cooperative system does little more than a C setjump/longjump call would do.) You can have thousands of threads in your applications significantly impacting performance. Since you don't lose control involuntarily in cooperative systems, you don't have to worry about synchronization either. That is, you never have to worry about an atomic operation being interrupted. The main disadvantage of the cooperative model is that it's very difficult to program cooperative systems. Lengthy operations have to be manually divided into smaller chunks, which often must interact in complex ways.

The preemptive multithreading model: The alternative to a cooperative model is a preemptive one, where some sort of timer is used by the operating system itself to cause a context swap. The interval between timer ticks is called a time slice. Preemptive systems are less efficient than cooperative ones because the thread management must be done by the operating-system kernel, but they're easier to program (with the exception of synchronization issues) and tend to be more reliable since starvation is less of a problem. The most important advantage to preemptive systems is parallelism. Since cooperative threads are scheduled by a user-level subroutine library, not by the OS, the best you can get with a cooperative model is concurrency. To get parallelism, the OS must do the scheduling. Of course, four threads running in parallel will run much faster than the same four threads running concurrently.

Threaded environments like Java allow a thread to put locks on shared resources so that while one thread is using data no other thread can touch that data. This is done with *synchronization*. Synchronization should be used sparingly since the purpose of threading is defeated if the entire system gets stopped waiting for a lock to be released. The proper choice of objects and methods to synchronize is one of the more difficult things to learn about threaded programming.

8.2.3. Race Conditions

A race condition occurs when two threads try to access the same object at the same time, and the behavior of the code changes depending on who wins. Figure 8.2 shows a single (unsynchronized) object accessed simultaneously by multiple threads. A thread can be preempted in fred() after modifying one field but before modifying the other. If another thread comes along at that point and calls any of the methods shown, the object will be left in an unstable state, because the initial thread will eventually wake up and modify the other field.

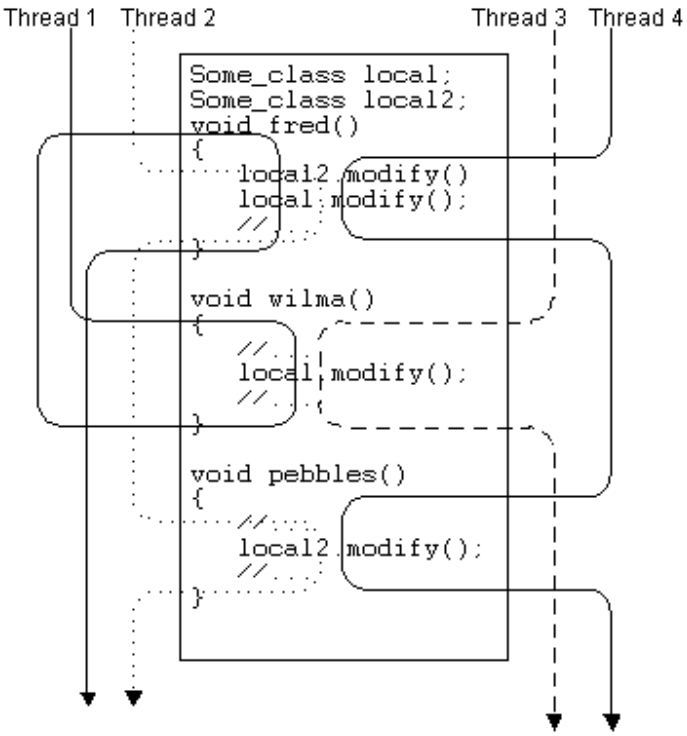

Figure 8.2. A race condition example

Usually, you think of objects sending messages to other objects. In multithreaded environments, you must think about message handlers running on threads. Think: this thread causes this object to send a message to that object. A race condition can occur when two threads cause messages to be sent to the same object at the same time.

8.2.4. Deadlocks

Deadlock is a scenario where a thread is blocked forever, waiting for something to happen that can't. The most common deadlock scenario occurs when two threads are both waiting for each other to do something. The following code snippet makes what's going on painfully obvious:

```
class Flintstone
{
  int field_1; private Object lock_1 = new int[1];
  int field_2; private Object lock_2 = new int[1];

  public void fred( int value )
  { synchronized( lock_1 )
    { synchronized( lock_2 )
      {
        field_1 = 0;
        field_2 = 0;
      }
    }
  }

  public void barney( int value )
```

```
  { synchronized( lock_2 )
    { synchronized( lock_1 )
      {
        field_1 = 0;
        field_2 = 0;
      }
    }
  }
}
```

Now, imagine a scenario whereby one thread (call it A) calls fred(), passes through the synchronization of lock_1, and is then preempted, allowing another thread (call it B) to execute. B calls barney(), acquires lock_2, and tries to acquire lock_1, but can't because A has it. B is now blocked, waiting for lock_1 to become available, so A wakes up and tries to acquire lock_2 but can't because B has it. A and B are now deadlocked. Neither one can ever execute. (Note that lock_1 and lock_2 have to be one-element arrays rather than simple ints, because only objects have monitors in Java; the argument to synchronized must be an object. An array is a first-class object in Java; a primitive-type such as int is not. Consequently, you can synchronize on it. Moreover, a one-element array is efficient to bring into existence compared to a more elaborate object (like an Integer) since it's both small and does not require a constructor call.

A and B are a simplified example, but the multiple-lock situation comes up frequently.

8.3. Multithreading Basics

8.3.1 Basic Concepts

All Java programs other than simple console-based applications are multithreaded, whether you like it or not. A thread is a piece of code that can act independently of any other code within an application. The piece of code is usually started by another piece of code, but once it is started, it runs independently of the original caller, and controls its own destiny, unless there is some outside interference. This is extremely useful in network computing.

- Threads are a great boon to communications code because they combine the convenience of synchronous programming with the efficiency of asynchronous programming.
- In a sophisticated communication application, you might have one thread managing communications, another thread handling screen drawing, another collecting user input, and yet another thread dealing with file I/O.
- Threads make it easier to create multiple simultaneous communications sessions; you simply create a new thread each time you create a new session manager.
- Java threads can potentially execute on separate processors, potentially resulting in much faster execution.
- You can solve certain problems faster using concurrent threads.

Although threads can run independently, they also share resources when running. Two classic problems in parallel processing, the race conditions and deadlocks, should also be considered when doing multithreading.

To use a thread to run an object, the class needs to be declared as a thread, by either extending *java.lang.Thread* or by implementing *java.lang.Runnable*. Within both of these there is a *run*() method that is to be called by the Java VM when the thread is started.

Application code overwrites the *run*() method. Once this method exits, the thread exits. Note that the *run*() method is called by the VM, not from the user code. A *start*() method is used to start a thread's execution from user code, which signals the VM to start a separate thread to run an instance of the class. Once the *start*() method exits, the *run*() method is called by the VM and the thread starts to run. A *stop*() method is also provided to stop the execution of a thread (the *stop*() method has been deprecated in Java 2. If you want to stop the execution of a thread, you can use the *interrupt*() method to notify the thread).

8.3.2. Simple Threading Examples

Our first example uses the *java.lang.Thread*. This is an abstract class that can be extended and provides the core requirements for threading. Here are the steps to use this mechanism:

- Write a new class that extends *Thread* overriding the *run*() method.
- Create an instance of that class in your Java code.
- Call the start method on that thread.

The example program, *ThreadTest1.java* is shown below:

```
//ThreadTest1.java
public class ThreadTest1 extends Thread
{
  private int thread_number;
  public ThreadTest1(int threadNumber)
  {
    thread_number = threadNumber;
  }
  public void run()
  {
    System.out.println("running thread " + thread_number);
    while(true)
    {
       try
       {
         Thread.sleep(1000); // do the real work here
         System.out.println("thread " + thread_number + " is running");
       }
       catch(InterruptedException e)
       {
         // do nothing
       }
    }
  }
  public static void main(String[] args)
  {
```

```
    // create two instances of the class
    ThreadTest1 test1 = new ThreadTest1(1);
    ThreadTest1 test2 = new ThreadTest1(2);

    // now run the threads
    test1.start();
    test2.start();
  }
}
```

Note that you need to use Ctrl-C to kill the program. For a thread, the `start()` method prepares a thread to be run; the `run()` method actually performs the work of the thread; and the `stop()` method halts the thread. The thread *dies* when the the `run()` method terminates or when the thread's `stop()` method is invoked.

You never call `run()` explicitly. It is called automatically by the runtime as necessary once you've called `start()`. There are also methods to supend and resume threads, to put threads to sleep and wake them up, to yield control to other threads, and many more.

The second example uses the *Runnable* interface. Runnable requires you implement the *run*() method. Here are the steps to use this mechanism:

- Write a new class that implements *Runnable*, providing the *run*() method.
- Create an instance of the class in the Java code.
- Create an instance of the *Thread*, passing it the class that you have just created as part of the constructor.
- Call the *start*() method on that thread.

Here is the example program, named *ThreadTest2.java*:

```
//ThreadTest2.java
public class ThreadTest2 implements Runnable
{
  private int thread_number = -1;
  public ThreadTest2(int threadNumber)
  {
    thread_number = threadNumber;
  }
  public void run()
  {
    System.out.println("running thread " + thread_number);
    while(true)
    {
       try
       {
         Thread.sleep(1000); // do the real thing here
          System.out.println("thread " + thread_number + " is running");
       }
       catch(InterruptedException e)
       {
         // do nothing
       }
    }
  }
  public static synchronized void check()
  {
```

```
    System.out.println("Checked");
  }
  public static void main(String[] args)
  {
    // create two instances of the class
    ThreadTest2 test1 = new ThreadTest2(1);
    ThreadTest2 test2 = new ThreadTest2(2);

    Thread thread1 = new Thread(test1);
    Thread thread2 = new Thread(test2);

    // now run the threads
    thread1.start();
    thread2.start();
  }
}
```

The difference of the two mechanisms is, when you extend *Thread*, you are creating the equivalent of a one-shot thread. That is, once you have called *start*() and the thread completes it execution, you cannot restart the thread from the beginning. On the other hand, classes which implement the *Runnable* interface can be restarted as many times as you need just by calling *start*() as required.

8.4. Multithreaded Servers

8.4.1. Adding Threading to Servers

The HelloServer of Section 6.3.2 and the EchoServer of Section 6.3.3 could only handle one client at a time. That wasn't so much of a problem for HelloServer because it had only a very brief interaction with each client. However the EchoServer might hang on to a connection indefinitely. In this case, it's better to make your server multi-threaded. There should be a loop which continually accepts new connections. However, rather than handling the connection directly the Socket should be passed to a Thread object that handles the connection.

The following example is a threaded echo program.

```
import java.net.*;
import java.io.*;
public class ThreadedEchoServer extends Thread {
  public final static int defaultPort = 2347;
  Socket theConnection;
  public static void main(String[] args) {
    int port = defaultPort;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
    }
    if (port <= 0 || port >= 65536) port = defaultPort;
    try {
      ServerSocket ss = new ServerSocket(port);
      while (true) {
        try {
          Socket s = ss.accept();
          ThreadedEchoServer tes = new ThreadedEchoServer(s);
```

```
          tes.start();
        }
        catch (IOException e) {
        }
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
  public ThreadedEchoServer(Socket s) {
    theConnection = s;
  }
  public void run() {
    try {
      OutputStream os = theConnection.getOutputStream();
      InputStream is = theConnection.getInputStream();
      while (true) {
        int n = is.read();
        if (n == -1) break;
        os.write(n);
        os.flush();
      }
    }
    catch (IOException e) {
    }
  }
}
```

Note that explicit yields are not required because all the different threads will tend to block on calls to read() and accept().

8.4.2. Adding a Thread Pool to a Server

Multi-threading is a good thing but it's still not a perfect solution. For example, let's take a look at the accept loop of the ThreadedEchoServer:

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

Every time you pass through this loop, a new thread gets created. Every time a connection is finished the thread is disposed of. Spawning a new thread for each connection takes a non-trivial amount of time, especially on a heavily loaded server. It would be better not to spawn so many threads.

An alternative approach is to create a pool of threads when the server launches, store incoming connections in a queue, and have the threads in the pool progressively remove connections from the queue and process them. This is particularly simple since the operating system does in fact store the incoming connections in a queue. The main

change you need to make to implement this is to call accept() in the run() method rather than in the main() method. The program below demonstrates this approach.

```
import java.net.*;
import java.io.*;
public class PoolEchoServer extends Thread {
  public final static int defaultPort = 2347;
  ServerSocket theServer;
  static int numberOfThreads = 10;
  public static void main(String[] args) {
    int port = defaultPort;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
    }
    if (port <= 0 || port >= 65536) port = defaultPort;
    try {
      ServerSocket ss = new ServerSocket(port);
      for (int i = 0; i < numberOfThreads; i++) {
        PoolEchoServer pes = new PoolEchoServer(ss);
        pes.start();
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
  public PoolEchoServer(ServerSocket ss) {
    theServer = ss;
  }
  public void run() {
    while (true) {
      try {
        Socket s = theServer.accept();
        OutputStream out = s.getOutputStream();
        InputStream in = s.getInputStream();
        while (true) {
          int n = in.read();
          if (n == -1) break;
          out.write(n);
          out.flush();
        } // end while
      } // end try
      catch (IOException e) {
      }
    } // end while
  } // end run
}
```

In the program above the number of threads is set to ten. This can be adjusted for performance reasons. How would you go about testing the performance of this program relative to the one that spawns a new thread for each connection? How would you determine the optimum number of threads to spawn?

8.5. An Interesting Parallel Client-Server Application

The example is a chat program consisting of three programs: *ChatClient.java*, *ChatServer.java* and *ChatHandler.java*.

8.5.1. The Chat Client

The *Chatclient.java* implements the chat client. It takes the server's machine name and the port number as the input from the command line. Then it makes a connection to the server and opens a window with two regions for displaying the chat contents and entering the chat input line. When the user enters a line in the input region and types the Return, the line is transmitted to the server and the server echoes everything it received from a client to all clients. The client displays everything received from the server in its output region.

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

public class ChatClient implements Runnable, WindowListener, ActionListener {
  protected String host;
  protected int port;
  protected Frame frame;
  protected TextArea output;
  protected TextField input;

  public ChatClient (String host, int port) {
    this.host = host;
    this.port = port;
    frame = new Frame ("ChatClient [" + host + ':' + port + "]");
    frame.addWindowListener (this);
    output = new TextArea ();
    output.setEditable (false);
    input = new TextField ();
    input.addActionListener (this);
    frame.add ("Center", output);
    frame.add ("South", input);
    frame.pack ();
  }

  protected DataInputStream dataIn;
  protected DataOutputStream dataOut;
  protected Thread listener;

  public synchronized void start () throws IOException {
    if (listener == null) {
      Socket socket = new Socket (host, port);
      try {
        dataIn = new DataInputStream
          (new BufferedInputStream (socket.getInputStream ()));
        dataOut = new DataOutputStream
          (new BufferedOutputStream (socket.getOutputStream ()));
      } catch (IOException ex) {
        socket.close ();
        throw ex;
      }
      listener = new Thread (this);
      listener.start ();
```

```
      frame.setVisible (true);
    }
  }

  public synchronized void stop () throws IOException {
    frame.setVisible (false);
    if (listener != null) {
      listener.interrupt ();
      listener = null;
      dataOut.close ();
    }
  }

  public void run () {
    try {
      while (!Thread.interrupted ()) {
        String line = dataIn.readUTF ();
        output.append (line + "\n");
      }
    } catch (IOException ex) {
      handleIOException (ex);
    }
  }

  protected synchronized void handleIOException (IOException ex) {
    if (listener != null) {
      output.append (ex + "\n");
      input.setVisible (false);
      frame.validate ();
      if (listener != Thread.currentThread ())
        listener.interrupt ();
      listener = null;
      try {
        dataOut.close ();
      } catch (IOException ignored) {
      }
    }
  }

  public void windowOpened (WindowEvent event) {
    input.requestFocus ();
  }

  public void windowClosing (WindowEvent event) {
    try {
      stop ();
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
  }

  public void windowClosed (WindowEvent event) {}
  public void windowIconified (WindowEvent event) {}
  public void windowDeiconified (WindowEvent event) {}
  public void windowActivated (WindowEvent event) {}
  public void windowDeactivated (WindowEvent event) {}

  public void actionPerformed (ActionEvent event) {
    try {
      input.selectAll ();
      dataOut.writeUTF (event.getActionCommand ());
      dataOut.flush ();
    } catch (IOException ex) {
```

```
      handleIOException (ex);
    }
  }

  public static void main (String[] args) throws IOException {
    if ((args.length != 1) || (args[0].indexOf (':') < 0))
      throw new IllegalArgumentException ("Syntax: ChatClient <host>:<port>");
    int idx = args[0].indexOf (':');
    String host = args[0].substring (0, idx);
    int port = Integer.parseInt (args[0].substring (idx + 1));
    ChatClient client = new ChatClient (host, port);
    client.start ();
  }
}
```

8.5.2. The Chat Server

The chat server is implemented by two Java programs. The *ChatServer.java* contains the main class *ChatServer* that accepts connections from clients and assigns them to new connection handler objects.

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

public class ChatServer {
  public static void main (String args[]) throws IOException {
    if (args.length != 1)
      throw new IllegalArgumentException ("Syntax: ChatServer <port>");
    int port = Integer.parseInt (args[0]);
    ServerSocket server = new ServerSocket (port);
    while (true) {
      Socket client = server.accept ();
      System.out.println ("Accepted from " + client.getInetAddress ());
      ChatHandler handler = new ChatHandler (client);
      handler.start ();
    }
  }
}
```

The other program, *ChatHandler.java*, implements the *Chathandler* class to listen for messages from a client and broadcast them to all connected clients. A list of current handlers is maintained within the *ChatHandler* class. The *broadcast*() method of the class uses this list to transmit a message to all connected clients.

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

public class ChatHandler implements Runnable {
  protected Socket socket;

  public ChatHandler (Socket socket) {
    this.socket = socket;
  }

  protected DataInputStream dataIn;
```

```
  protected DataOutputStream dataOut;
  protected Thread listener;

  public synchronized void start () {
    if (listener == null) {
      try {
        dataIn = new DataInputStream
          (new BufferedInputStream (socket.getInputStream ()));
        dataOut = new DataOutputStream
          (new BufferedOutputStream (socket.getOutputStream ()));
        listener = new Thread (this);
        listener.start ();
      } catch (IOException ignored) {
      }
    }
  }

  public synchronized void stop () {
    if (listener != null) {
      try {
        if (listener != Thread.currentThread ())
          listener.interrupt ();
        listener = null;
        dataOut.close ();
      } catch (IOException ignored) {
      }
    }
  }

  protected static Vector handlers = new Vector ();

  public void run () {
    try {
      handlers.addElement (this);
      while (!Thread.interrupted ()) {
        String message = dataIn.readUTF ();
        broadcast (message);
      }
    } catch (EOFException ignored) {
    } catch (IOException ex) {
      if (listener == Thread.currentThread ())
        ex.printStackTrace ();
    } finally {
      handlers.removeElement (this);
    }
    stop ();
  }

  protected void broadcast (String message) {
    synchronized (handlers) {
      Enumeration enum = handlers.elements ();
      while (enum.hasMoreElements ()) {
        ChatHandler handler = (ChatHandler) enum.nextElement ();
        try {
          handler.dataOut.writeUTF (message);
          handler.dataOut.flush ();
        } catch (IOException ex) {
          handler.stop ();
        }
      }
    }
  }
}
```

Students are required to compile the three programs and test run the server on one machine, the client instances on at least two machines. If possible, a number of students can form a group to test run the chat program (one runs the server and the client is run on multiple sites).

9. Distributed Database Applications using Java

9.1. Study Points

- Understand the principles and architectures of Web-based databases.
- Understand the basic principles of JDBC.
- Be able to use JDBC to develop simple database applications.
- Be able to combine the JDBC and Java networking facilities to build simple Web-based database applications.

Reference: (1). [FAR] Chapter 7. (2). [Java2H] Chapter 11. (3). [Java2U]. Chapters 43, 44, 45, 46. (4). Mingjun Lan, Shui Yu, and Wanlei Zhou, "*Current Technologies and Development of Web-Based Databases: A Survey*", Technical Report, Deakin University, TR C01/12, 2001.

9.2. Web-Based Database

9.2.1. Why Web-Based Database?

Java network programming opens the possibility of building Web-based distributed databases. A Web-based distributed database is a key component of many Internet-related applications, such as applications in electronic commerce, information retrieval, and multimedia.

The current wisdom on databases is that information stored in databases is owned by the database management systems (DBMS) that manage the databases. The DBMS is a closed system in the sense that all operations on the data managed by the DBMS will be stored back to the database. A further development on distributed databases and heterogeneous databases allows information to be stored in different databases using various formats and to be shared among participating databases. However, a distributed heterogeneous database system is still a closed system that is managed by a distributed database management system (DDBMS).

The Web-based database approach represents a deviation from this traditional mode of thinking. It allows data to be represented in objects (consisting of data and methods that manipulate the data) and the access of these objects are open to anyone with the correct access rights. Information stored in a Web-based database is independent of any particular software (such as the DBMSs in the traditional database approach). Access to the Web-based database can be easily integrated into any user interface, such as a conventional WWW browser or a particular application program. Web-based databases have a great potential in electronic commerce, information retrieval and multimedia applications.

Web-based databases possess of a number of advantages:

- Maintenance and Updating. It separates content (database) from presentation (an HTML page). It means that the owner of the site is able to update the content of their site without constantly having to go through their webmaster or designer. Creating an

Web template once and merging it with new content (database) is a more reliable way than publishing information with a consistent layout.

- Reusability and modularity. By designing additional templates, one can easily reuse content on another Web site or modify it to fit a new design. For users, databases make site searches more accurate: they can be limited to certain fields, returning better-quality hits than full-text searches.
- The ability to distribute data update. With the right interface, even a novice user can go into the database to update information; the Web publishing system can then send out the changes immediately.
- Security. Databases help ensure that contents are accessed by authorized users.

9.2.2. The Two-tier Architecture of Web-based Databases

There are different WBDB frameworks according to various technologies and requirements. Generally speaking, the WBDB can be considered as a single huge database as well as multiple data sources. There are a lot of technologies that can be used for WBDB. Languages for web applications and web servers are Java, PHP, Perl, HTML, DHTML, XML, SQL and so forth. Access technologies contain CGI, JavaScript, Servlet, JDBC, and ODBC. Common enterprise databases include Oracle, Sysbase, Informix, DB2,mSQL, mySQL, SQL-Server, and Butler-SQL. We generally classify WBDB architectures into the following types Two-tier Architecture of WBDB, Three-tier Architecture of WBDB, and Hybrid Architectures of WBDB.

The minimal spatial configuration of a WBDB is the two-tier architecture. The basic framework is shown in Figure 9.1.

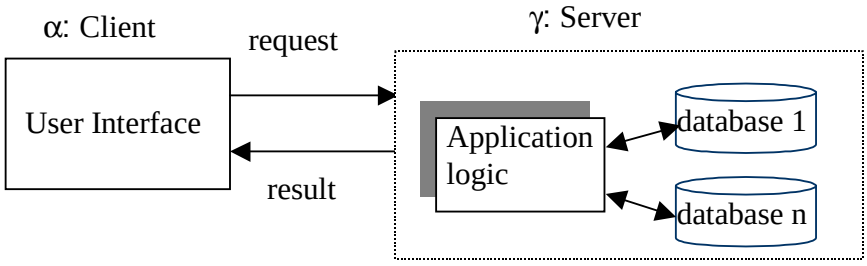

Figure 9.1. Two-tier architecture of Web-based databases

9.2.3. The Three-tier Architecture of Web-based Databases

The three-tier architecture is a popular model, which contains generally the Client (we called it α), Application Server (we called it β), and Data Server (we called it γ), see Figure 9.2. A full-fledged WBDB requires these three essential components although they can represent various types of technologies. In the following, we discuss some current three-tier architectures of WBDB.

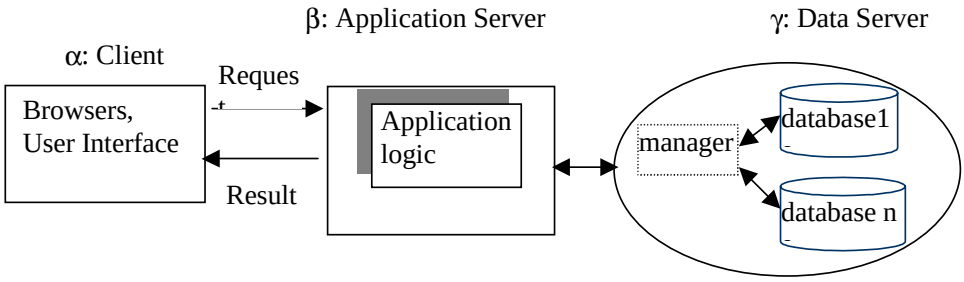

Figure 9.2. Three-tier architecture of Web-based databases

In the three-tier model of a database gateway, the three components are Client API Library, Server API Library, and Glue. The α component is the Client API Library, which consists of client-side APIs. They determine the format and meaning of the requests that the client applications may issue. Glue is the β component, which owns translation and mapping mechanisms. It transforms the client API to the DBMS (Database Management System) server's API, and vice versa for the data returned to the Clients. Server API Library on the database server-side is the γ component. It manages the database service available to the clients. The services change in terms of the authentication from the DBMS.

The TP (transaction-processing) monitor model is also a kind of three-tier architecture. In this context, Client Application (α component) consists of the user-interface functions, such as screen logic, screen handling, input handling, and some validation functions. Application Servers (β components) provides all of the details of application services. Resource managers (γ component) can provide all of lower-level services, such as communication between the database and the application services, see the dashed diagram in Figure 2.

The extended client/server model is a typical three-tier architecture. In such a model, the client Web browser (α components) sends the request to the Web server (β components). The Web server transfers the request to a database server (γ components). After the database server processes the query, the results are retrieved to the client Web browser by the reverse pathway. In the transition, the web server can handle the results from the database.

In the Multi-distributed databases (MDBS) scenario, the Web server (β) requests the MDBS (γ) to retrieve the required data. The γ server does this by issuing a global-level SQL query to the MDBS. The MDBS then decomposes the whole query and generates the local queries according to various features of engaging database servers. Then these local queries can be issued to corresponding database servers that may be managed by the DBMS servers. But these DBMS servers can be accessed through all sorts of database access technologies. The MDBS integrates the local results it receives from all the database servers and finally presents a global result to the web server. In this case, the MDBS handles all the operations including data locating, interrelating, and integrating. The web server just sends the requests from clients, which is different from the typical client/server model.

All the technologies can be used in the three-tier architecture according to different user requirements. The three-tier or even n-tier models are essential models to structure a WBDB.

9.2.4. The Hybrid Architecture of Web-based Databases

There are several ways combining various technologies into Web or database to enhance the performances of WBDB. A general architecture is to apply agent-based computing concepts in building WBDBs, see Figure 9.3. Strictedly speaking, however, it also is a three-tier architecture

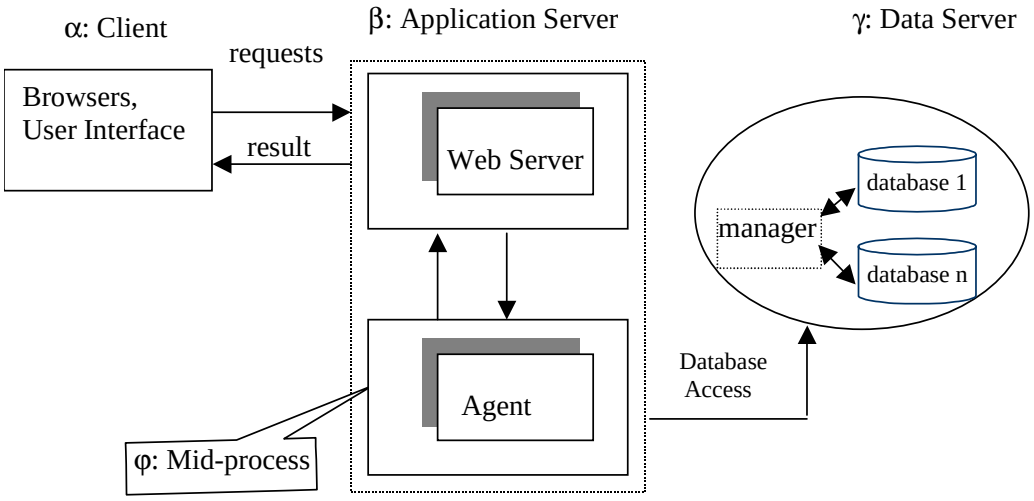

Figure 9.3. Hybrid (agent-based) architecture of Web-based databases

In an agent-based scenario, the clients (α) send either data or data and programs over the Web server that activates the agent (φ). The agent then processes the request data using its own programs or using the received programs. After the completion of the preliminary processing, the agent will send the data/program/medium result to the application server (β) for further processing. Then the web server communicates with the database and the database server (γ) finishes the manipulation to the databases and transfers the results to the Web server. The web server will return the results back to the client directly or via the agent.

9.3. An Overview of Java Database Connectivity (JDBC)

9.3.1. What is JDBC

The JDBC package is a set of Java classes that can be used by applications to make database calls. It specifies the interfaces between Java and databases (based on SQL2). JavaSoft (the group that defined JDBC) only specifies the interfaces. All implementation of JDBC drivers is done by third part companies with special expertise. A change of a driver will not change the program. The major advantages of using JDBC are the cross-platform independence and the possibility of delivering database functionality using Java applets through the Internet.

JDBC's classes are contained in the "*java.sql*" package. It includes the following major classes:

- *DriverManger*: The `DriverManager` object is used to facilitate the use of multiple database drivers in a single application. Each JDBC driver can be used to connect to a different data source.
- *Connection*: After a JDBC driver has been registered with the `DriverManager`, a data source, user ID, password, or other pertinent information can be specified to create a connection to a database. this `Connection` object is used in later calls to specify against which database the call should be placed. JDBC supports having multiple `Connection` objects open at any given time.
- *Statement*: The `Statement` object mimics the SQL statement that the application wants to apply against a database.
- After a call is made by a `Statement` object, the results of the query are put into a `ResultSet` object. This object can then be traversed to retrieve multiple rows as well as multiple columns.
- *ResultSetMetaData*: The `ResultSetMetaData` object can be used to inquery about the contents of a `ResultSet` object.
- *DatabaseMetaData*: The *DatabaseMetaData* object can be used to query the support options for a given database.
- *SQLException*: This exception is used to capture most problems that are returned from database systems. In addition, the JDBC offers a `SQLWarning` class that returns information that is not as severe as the `SQLException` class.

9.3.2. Using JDBC

A JDBC program initially invokes the *DriverManager* class's *getConnection*() method to establish a connection to the database. Once the connection is established, the program calls either the *createStatement*(), *prepareStatement*(), or *prepareCall*() method of the Connection object and prepares for executing the SQL statements. SQL statements can be executed by invoking the Statement object, or via the *PreparedStatement* object or the *CallableStatement* object.

Next, the program either calls the *executeQuery*(), *executeUpdate*(), or the *execute*() method of the *Statement*, *PreparedStatement*, or *CallableStatement* object. The *executeQuery*() method is used when only one *ResultSet* is needed, and the *execute*() method is used when more than one *ResultSet* is returned. The *executeUpdate*() method is used if no *ResultSet* is needed and the SQL statement contains an `UPDATE`, `INSERT`, or `DELETE`. The *next*() method of the *ResultSet* object can be used to process multiple rows of data.

9.4. Developing JDBC Applications: Using the Access Database

9.4.1. Prepare the Database

The first step of developing a web-based database using JDBC is to prepare the database for JDBC connection. We use an Access database as an example. First, a blank database,

named *dbtest.mdb*, is created. Second, the following steps are used to prepare the database for JDBC access:

- From the *Start* menu select the *Settings*
- Click the Control Panel, then the ODBC Data Source (32bit)
- Click *System DSN*, then *Add*
- Select Microsoft Database Driver (*.mdb)
- Type in data source name and use "*Select*" to find the database "*dbtest.mdb*"; Use "*Options*" to set the username and password.

9.4.2. Create the Database Tables

Assume that the database contains three tables: `CUSTOMER`, `PRODUCT`, and `TRANSACTIONS`.

The following Java program "*CreateCustomer.java*" creates the customer table with four columns (`C_NAME`, `C_ID`, `C_ADDR`, and `C_PHONE`):

```
import java.sql.*;
public class CreateCustomer {
	public static void main(String args[]) {
		String url = "jdbc:odbc:dbtest";
		Connection con;
		String createString;
		createString = "create table CUSTOMER " +
			"(C_NAME varchar(30), " +
			"C_ID int, " +
			"C_ADDR varchar(50), " +
			"C_PHONE varchar(12) )";
		Statement stmt;
		try {
		Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
		} catch(Exception e) {
			System.err.print("ClassNotFoundException: ");
			System.err.println(e.getMessage());
		}
		try {
			con = DriverManager.getConnection(url, "admin", "admin");
			stmt = con.createStatement();

			stmt.executeUpdate(createString);
			stmt.close();
			con.close();
		} catch(SQLException ex) {
			System.err.println("SQLException: " + ex.getMessage());
		}
	}
}
```

Now, you can compile the program and run it. After that, check the database to see if the table is created.

The creation of the `PRODUCT` table (with five columns of `P_NAME`, `P_DESC`, `P_CODE`, `P_UNIT`, and `P_STOCK`) is similar. The Java program "*CreateProduct.java*" is listed below:

```
import java.sql.*;
public class CreateProduct {
	public static void main(String args[]) {
		String url = "jdbc:odbc:dbtest";
		Connection con;
		String createString;
		createString = "create table PRODUCT " +
			"(P_NAME varchar(20), " +
			"P_DESC varchar(40), " +
			"P_CODE varchar(8), " +
			"P_UNIT_PRICE float, " +
			"P_STOCK int )";
		Statement stmt;
		try {
		Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
		} catch(Exception e) {
			System.err.print("ClassNotFoundException: ");
			System.err.println(e.getMessage());
		}
		try {
			con = DriverManager.getConnection(url, "admin", "admin");
			stmt = con.createStatement();

			stmt.executeUpdate(createString);
			stmt.close();
			con.close();
		} catch(SQLException ex) {
			System.err.println("SQLException: " + ex.getMessage());
		}
	}
}
```

You can also compile the program, run it, and check the database to see if the table is created.

The creation of the `TRANSACTION` table (with column of `T_ID, C_ID, P_CODE, T_NUM, T_TOTAL_PRICE,` and `T_DAATE`) is also similar. The following Java program "*CreateTransaction.java*" completes such a task:

```
import java.sql.*;
public class CreateTransaction {
	public static void main(String args[]) {
		String url = "jdbc:odbc:dbtest";
		Connection con;
		String createString;
		createString = "create table TRANSACTION " +
			"(T_ID int, " +
			"C_ID int, " +
			"P_CODE varchar(8), " +
			"T_NUM int, " +
			"T_TOTAL_PRICE float, " +
			"T_DATE date )";
		Statement stmt;
		try {
		Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
		} catch(Exception e) {
			System.err.print("ClassNotFoundException: ");
			System.err.println(e.getMessage());
		}
		try {
```

```
                    con = DriverManager.getConnection(url, "admin", "admin");
                    stmt = con.createStatement();

                    stmt.executeUpdate(createString);
                    stmt.close();
                    con.close();
             } catch(SQLException ex) {
                    System.err.println("SQLException: " + ex.getMessage());
             }
      }
}
```

You should also compile the program, run it, and check the database to see if the table is created.

9.4.3. Populate the Tables

You can populate the three tables using the following programs, named *InsertCustomer.java*, *InsertProduct.java*, and *InsertTransaction.java*, respectively:

```
// "InsertCustomer.java"
import java.sql.*;
public class InsertCustomer {
       public static void main(String args[]) {
              String url = "jdbc:odbc:dbtest";
              Connection con;
              Statement stmt;
              String query = "select * from CUSTOMER";
              try {
                     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

              } catch(java.lang.ClassNotFoundException e) {
                     System.err.print("ClassNotFoundException: ");
                     System.err.println(e.getMessage());
              }
              try {
                     con = DriverManager.getConnection(url, "admin", "admin");
                     stmt = con.createStatement();

                     stmt.executeUpdate("insert into CUSTOMER " +
                       "values('John Smith', 100, '123 King St.', '03-9123 4567')");
                     stmt.executeUpdate("insert into CUSTOMER " +
                       "values('Alex Lee', 101, '234 Queen St.', '03-9234 5678')");
                     stmt.executeUpdate("insert into CUSTOMER " +
                       "values('Anne Wong', 102, '345 Yarra Ave.', '03-9345 6789')");
                     stmt.executeUpdate("insert into CUSTOMER " +
                       "values('Tanya Foo', 103, '456 Irving Rd.', '03-9456 7890')");

                     ResultSet rs = stmt.executeQuery(query);
                     System.out.println("C_NAME C_ID C_ADDR C_PHONE");
                     while (rs.next()) {
                            String s = rs.getString("C_NAME");
                            int i = rs.getInt("C_ID");
                            String s1 = rs.getString("C_ADDR");
                            String s2 = rs.getString("C_PHONE");
                            System.out.println(s + "  " + i +
                                   "  " + s1 + "  " + s2);
                     }
                     stmt.close();
                     con.close();
              } catch(SQLException ex) {
                     System.err.println("SQLException: " + ex.getMessage());
              }
       }
}
```

// “InsertProduct.java”:

```
import java.sql.*;
public class InsertProduct {
	public static void main(String args[]) {
		String url = "jdbc:odbc:dbtest";
		Connection con;
		Statement stmt;
		String query = "select * from PRODUCT";
		try {
			Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

		} catch(java.lang.ClassNotFoundException e) {
			System.err.print("ClassNotFoundException: ");
			System.err.println(e.getMessage());
		}
		try {
			con = DriverManager.getConnection(url, "admin", "admin");
			stmt = con.createStatement();

			stmt.executeUpdate("insert into PRODUCT " +
			  "values('TV', 'Philip, 68cm, flat screen', 'T0010', 1200.00,
10)");
			stmt.executeUpdate("insert into PRODUCT " +
			  "values('VCR', 'Sony, Mid-Drive', 'V100', 500.00, 15)");
			stmt.executeUpdate("insert into PRODUCT " +
			  "values('TV', 'Tohisba, 34cm, remote control', 'T0012', 300.00,
20)");
			stmt.executeUpdate("insert into PRODUCT " +
			  "values('PC', 'Dell, 256M RAM, 10GHD, 17\" monitor', 'P0012',
2400.00, 12)");

			ResultSet rs = stmt.executeQuery(query);
			System.out.println("P_NAME  P_DESC  P_CODE  P_UNIT_PRICE
P_STOCK");
			while (rs.next()) {
				String s = rs.getString("P_NAME");
				String s1 = rs.getString("P_DESC");
				String s2 = rs.getString("P_CODE");
				float f = rs.getFloat("P_UNIT_PRICE");
				int i = rs.getInt("P_STOCK");
				System.out.println(s + "  " + s1 + "  " + s2 +
					"  " + f + "  " + i);
			}
			stmt.close();
			con.close();
		} catch(SQLException ex) {
			System.err.println("SQLException: " + ex.getMessage());
		}
	}
}
```

// “InsertTransaction.java”:

```
import java.sql.*;
public class InsertTransaction {
	public static void main(String args[]) {
		String url = "jdbc:odbc:dbtest";
		Connection con;
		Statement stmt;
		String query = "select * from TRANSACTION";
		try {
			Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

		} catch(java.lang.ClassNotFoundException e) {
			System.err.print("ClassNotFoundException: ");
			System.err.println(e.getMessage());
		}
		try {
			con = DriverManager.getConnection(url, "admin", "admin");
			stmt = con.createStatement();
```

```
                        stmt.executeUpdate("insert into TRANSACTION " +
                          "values(500, 100, 'T0010', 1, 1200.00, #1/8/2000#)");
                        stmt.executeUpdate("insert into TRANSACTION " +
                          "values(501, 101, 'V100', 2, 1000.00, #2/20/2000#)");

                        ResultSet rs = stmt.executeQuery(query);
                        System.out.println("T_ID  C_ID  P_CODE  T_NUM  T_TOTAL_PRICE
T_DATE");
                        while (rs.next()) {
                               int i = rs.getInt("T_ID");
                               int i1 = rs.getInt("C_ID");
                               String s = rs.getString("P_CODE");
                               int i2 = rs.getInt("T_NUM");
                               float f = rs.getFloat("T_TOTAL_PRICE");
                               Date d = rs.getDate("T_DATE");
                               System.out.println(i + "  " + i1 + "  " + s +
                                      "  " + i2 + "  " + f + "  " + d);
                        }
                        stmt.close();
                        con.close();
                } catch(SQLException ex) {
                        System.err.println("SQLException: " + ex.getMessage());
                }
        }
}
```

You can compile the programs, run them, and check the database to see if the tables are populated.

9.4.4. Print Columns of Tables

The following Java program "*PrintColumns.java*" prints all contents of the PRODUCT table:

```
import java.sql.*;
class PrintColumns  {
       public static void main(String args[]) {
              String url = "jdbc:odbc:dbtest";
              Connection con;
              String query = "select * from CUSTOMER";
              Statement stmt;
              try {
                     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
              } catch(java.lang.ClassNotFoundException e) {
                     System.err.print("ClassNotFoundException: ");
                     System.err.println(e.getMessage());
              }
              try {
                     con = DriverManager.getConnection(url, "admin", "admin");
                     stmt = con.createStatement();

                     ResultSet rs = stmt.executeQuery(query);
                     ResultSetMetaData rsmd = rs.getMetaData();
                     PrintColumnTypes.printColTypes(rsmd);
                     System.out.println("");
                     int numberOfColumns = rsmd.getColumnCount();
                     for (int i = 1; i <= numberOfColumns; i++) {
                            if (i > 1) System.out.print(",  ");
                            String columnName = rsmd.getColumnName(i);
                            System.out.print(columnName);
                     }
                     System.out.println("");
                     while (rs.next()) {
```

```
                        for (int i = 1; i <= numberOfColumns; i++) {
                              if (i > 1) System.out.print(",  ");
                              String columnValue = rs.getString(i);
                              System.out.print(columnValue);
                        }
                        System.out.println("");
                  }
                  stmt.close();
                  con.close();
            } catch(SQLException ex) {
                  System.err.print("SQLException: ");
                  System.err.println(ex.getMessage());
            }
      }
}
```

The program uses a Java class called PrintColumnTypes, to identify the types used in Database and JDBC. The program (called PrintColumnTypes.java) is shown below:

```
import java.sql.*;
public class PrintColumnTypes  {
      public static void printColTypes(ResultSetMetaData rsmd)
            throws SQLException {

            int columns = rsmd.getColumnCount();
            for (int i = 1; i <= columns; i++) {
                  int jdbcType = rsmd.getColumnType(i);
                  String name = rsmd.getColumnTypeName(i);
                  System.out.print("Column " + i + " is JDBC type " +
jdbcType);
                  System.out.println(", which the DBMS calls " + name);
            }
      }
}
```

You should compile the program, run it, and check if the table is printed properly.

To print the columns of the PRODUCT and the TRANSACTION tables, only one line of the above program needs to be changed:

```
            String query = "select * from CUSTOMER";
```

Just change the “CUSTOMER” into PRODUCT or TRANSACTION will print the contents of these tables, respetively.

9.4.5. Execute a Select Statement (one table)

The following Java program “*SelectStatement.java*” executes the following SQL statement:

```
            select P_DESC, P_STOCK
            from PRODUCT
            where P_NAME like 'TV';
```

You can execute other SQL statements by simply changing the correspond statement in the program.

```
import java.sql.*;
public class SelectStatement  {
       public static void main(String args[]) {
              String url = "jdbc:odbc:dbtest";
              Connection con;
              String query = "select P_DESC, P_STOCK " +
                 "from PRODUCT " +
                 "where P_NAME like 'TV'";
              Statement stmt;
              try {
                     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
              } catch(java.lang.ClassNotFoundException e) {
                     System.err.print("ClassNotFoundException: ");
                     System.err.println(e.getMessage());
              }
              try {
                     con = DriverManager.getConnection(url, "admin", "admin");
                     stmt = con.createStatement();

                     ResultSet rs = stmt.executeQuery(query);
                     ResultSetMetaData rsmd = rs.getMetaData();
                     int numberOfColumns = rsmd.getColumnCount();
                     int rowCount = 1;
                     while (rs.next()) {
                            System.out.println("Row " + rowCount + ":  ");
                            for (int i = 1; i <= numberOfColumns; i++) {
                                   System.out.print("   Column " + i + ":  ");
                                   System.out.println(rs.getString(i));
                            }
                            System.out.println("");
                            rowCount++;
                     }
                     stmt.close();
                     con.close();

              } catch(SQLException ex) {
                     System.err.print("SQLException: ");
                     System.err.println(ex.getMessage());
              }
       }
}
```

You should compile the program, run it, and check the database to see if the result is selected properly.

9.5. Developing JDBC Applications: Using the Oracle Database

9.5.1. Using the Oracle Database from Windows OS.

Use the following steps to prepare the use of the Oracle database:

- Install the Oracle JDBC driver by unzipping the `oracle.zip` file (can be found in the unit web page) and store it into the correct directory (e.g. your working directory) or your class directory.
- Use the right URL and the database. The URL for our School's Oracle database is:

  ```
  final String url = "jdbc:oracle:thin:@turin.cm.deakin.edu.au:1521:cmodb";
  ```

- Use the right driver class:

```
final String driverClass="oracle.jdbc.driver.OracleDriver";
```

- Use the right username/password (please consult your tutor or lecturer for the username and password).

9.5.2. Create, Populate and Update the CUSTOMER Table

The following program creates the CUSTOMER table. To use it for your own program, you need to change the username/password from "scott/tiger" into your own. You also need to change the URL if you want to use another Oracle database.

```
import java.sql.*;
public class CreateCustomer1 {
  public static void main(String args[]) {
       final String url = "jdbc:oracle:thin:@turin.cm.deakin.edu.au:1521:cmodb";
       final String driverClass="oracle.jdbc.driver.OracleDriver";
       Connection con;
       String createString;
       createString = "create table CUSTOMER " +
              "(C_NAME varchar(20), " +
              "C_ID int, " +
              "C_ADDR varchar(20), " +
              "C_PHONE varchar(12) )";
       Statement stmt;
       try {
              Class.forName(driverClass).newInstance();
       } catch(Exception e) {
              System.err.print("ClassNotFoundException: ");
              System.err.println(e.getMessage());
       }
       try {
              con = DriverManager.getConnection(url, "scott", "tiger");
              stmt = con.createStatement();
              stmt.executeUpdate(createString);
              stmt.close();
              con.close();
       } catch(SQLException ex) {
              System.err.println("SQLException: " + ex.getMessage());
       }
  }
}
```

The following program populates the CUSTOMER table:

```
import java.sql.*;
public class InsertCustomer1 {
       public static void main(String args[]) {
              final String url =
"jdbc:oracle:thin:@turin.cm.deakin.edu.au:1521:cmodb";
              final String driverClass="oracle.jdbc.driver.OracleDriver";
              Connection con;
              Statement stmt;
              String query = "select * from CUSTOMER";
              try {
                     Class.forName(driverClass).newInstance();
              } catch(Exception e) {
                     System.err.print("ClassNotFoundException: ");
                     System.err.println(e.getMessage());
              }
              try {
                     con = DriverManager.getConnection(url, "scott", "tiger");
```

```
                stmt = con.createStatement();
                stmt.executeUpdate("insert into CUSTOMER " +
            "values('John Smith', 100, '123 King St.', '03-9123 4567')");
                stmt.executeUpdate("insert into CUSTOMER " +
            "values('Alex Lee', 101, '234 Queen St.', '03-9234 5678')");
                stmt.executeUpdate("insert into CUSTOMER " +
            "values('Anne Wong', 102, '345 Yarra Ave.', '03-9345 6789')");
                stmt.executeUpdate("insert into CUSTOMER " +
            "values('Tanya Foo', 103, '456 Irving Rd.', '03-9456 7890')");
                ResultSet rs = stmt.executeQuery(query);
                System.out.println("C_NAME C_ID C_ADDR C_PHONE");
                while (rs.next()) {
                        String s = rs.getString("C_NAME");
                        int i = rs.getInt("C_ID");
                        String s1 = rs.getString("C_ADDR");
                        String s2 = rs.getString("C_PHONE");
                        System.out.println(s + "  " + i +
                                "  " + s1 + "  " + s2);
                }
                stmt.close();
                con.close();
            } catch(SQLException ex) {
                System.err.println("SQLException: " + ex.getMessage());
            }
        }
}
```

The following program updates the CUSTOMER table:

```
import java.sql.*;

public class UpdateCustomer {
  public static void main(String args[]) {
       final String url = "jdbc:oracle:thin:@turin.cm.deakin.edu.au:1521:cmodb";
       final String driverClass="oracle.jdbc.driver.OracleDriver";
       Connection con;
       String createString;
       createString = "update CUSTOMER set C_ADDR = '99 Fred St.' where
C_ID=100";
       Statement stmt;
       try {
             Class.forName(driverClass).newInstance();
       } catch(Exception e) {
             System.err.print("ClassNotFoundException: ");
             System.err.println(e.getMessage());
       }
       try {
             con = DriverManager.getConnection(url,"scott","tiger");
             stmt = con.createStatement();
             stmt.executeUpdate(createString);
             stmt.close();
             con.close();
       } catch(SQLException ex) {
             System.err.println("SQLException: " + ex.getMessage());
       }
  }
}
```

Note that the above programs are very similar to the program in Section 9.4. The only real difference is the URL.

9.6. A JDBC Application Example

This example has a database that stores data, a server that manages the access of the database, and a client that interfaces with users. The client uses a Java applet to access the server and the server uses JDBC to access the database.

9.6.1. Prepare the Access Database and the HTML File

We use the Access database, "*dbtest.mdb*", created in the previous section. It includes three tables: CUSTOMER, PRODUCT, and TRANSACTION.

The first step is to prepare the following HTML file, named *Applet.html*, to use the applet:

```
<HTML>
<title>Database Operations</title>

<applet code="ClientApplet.class"
       width=600 height=350>
</applet>
</HTML>
```

9.6.2. Prepare the Java Applet Programs

Create the main applet program, "*ClientApplet.java*". This program implements the user interface. You should change the IP address in the program to a proper IP address.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import java.io.*;

public class ClientApplet extends Applet {
  private static final String host= "192.168.0.1";
  TextArea ta;
  ClientComm cc;
  ClientCommExit cce;
  ClientCommSQL ccs;
  TextField sqlcommand;

  public void init () {
    Panel p1 = new Panel(new BorderLayout(10, 10));
    Button p1b1 = new Button ("Resulsts Returned");
    p1.add (p1b1, BorderLayout.NORTH);
    ta = new TextArea ();
    ta.setEditable(false);
    p1.add (ta, BorderLayout.CENTER);
    sqlcommand = new TextField ("", 50);
    p1.add (sqlcommand, BorderLayout.SOUTH);
    add (p1);

    Panel p2 = new Panel (new FlowLayout());
    Button p2bCus = new Button ("All Customers");
    p2.add (p2bCus);
    p2bCus.addActionListener (new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        ByteArrayOutputStream bao = new ByteArrayOutputStream();
```

```
        cc = new ClientComm(host, 0, 1, bao);
        ta.setText (bao.toString()+"Returned by the <All Customer> request");
      }
    } );

    Button p2bPro = new Button ("All Products");
    p2.add (p2bPro);
    p2bPro.addActionListener (new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        ByteArrayOutputStream bao = new ByteArrayOutputStream();
        cc = new ClientComm(host, 0, 2, bao);
        ta.setText (bao.toString()+"Returned by the <All Product> request");
      }
    } );

    Button p2bTra = new Button ("All Transactions");
    p2.add (p2bTra);
    p2bTra.addActionListener (new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        ByteArrayOutputStream bao = new ByteArrayOutputStream();
        cc = new ClientComm(host, 0, 3, bao);
        ta.setText (bao.toString()+"Returned by the <All Transaction>
request");
      }
    } );

    Button p2bSQL = new Button ("SQL Command");
    p2.add (p2bSQL);
    p2bSQL.addActionListener (new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        ByteArrayOutputStream bao = new ByteArrayOutputStream();
        ccs = new ClientCommSQL(host, 0, 4, sqlcommand.getText(), bao);
        ta.setText (bao.toString()+"Returned by the <SQL Command> request");
      }
    } );

    Button p2bExit = new Button ("ShutDown Server");
    p2.add (p2bExit);
    p2bExit.addActionListener (new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        ByteArrayOutputStream bao = new ByteArrayOutputStream();
        cce = new ClientCommExit(host, 0, bao);
        ta.setText (bao.toString()+"Returned by the <ShutDown Server>
request");
        //System.exit(0);
      }
    } );

    add (p2);
  }
}
```

Create the Java program that implements the "*ClientComm*" class used in the applet: "*ClientComm.java*". This program deals with the major communication work between the applet and the server.

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

public class ClientComm {
  public static final int DEFAULT_PORT = 6789;
```

```
  private String host = "";
  private int port = 0;
  private OutputStream os = null;
  boolean DEBUG = true;

  public ClientComm (String h, int p, int choice, OutputStream o) {
    host = h;
    port = ((p == 0) ? DEFAULT_PORT : p);
    os = o;
    Socket s = null;
    PrintWriter out = new PrintWriter (os, true);
    if (DEBUG) {
      System.out.println("Applet about to create a socket on "
          + host + " at port " + port);
    }

    try {
      // create a socket to communicate to the specified host and port
      s = new Socket(host, port);
      // create streams for reading and writing
      BufferedReader sin = new BufferedReader(new
InputStreamReader(s.getInputStream()));
      PrintStream sout = new PrintStream(s.getOutputStream(), true);
      if (DEBUG) {
        System.out.println("Applet has created sin and sout ");
      }

      // tell the user that we've connected
      out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());

      if (DEBUG) {
        System.out.println("Applet has connected to "+ s.getInetAddress() +
            ":" + s.getPort());
      }

      String line;
      // read the first response (a line) from the server
      line = sin.readLine();
      if (DEBUG) {
        System.out.println("Applet has read a line: " + line);
      }

      // write the line to the user
      out.println(line);
      out.flush();
      // send the command choice to the server
      if (choice <=3) {
        sout.println(choice);
      } else {
        sout.println("Wrong command");
      }
      if (DEBUG) {
        System.out.println("Applet has sent sout the choice: "+ choice);
      }

      // read a line from the server
      line = sin.readLine();
      if (DEBUG) {
        System.out.println("Applet has read a line: " + line);
      }

      out.println(line);
```

```
      // check if connection is closed, i.e., EOF
      if (line == null) {
        out.println("Connection closed by server.");
      }
       while (true) {
        line=sin.readLine();
        if (DEBUG) {
          System.out.println("Applet has read a line: " + line);
        }

         out.println(line);
         if (line.equals("EndOfRecord")) break;
       }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

Create the Java program that implements the "*ClientCommExit*" class used in the applet: "*ClientCommExit.java*". This program deals with the special applet command of "Server Exit".

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

public class ClientCommExit {
  public static final int DEFAULT_PORT = 6789;
  private String host = "";
  private int port = 0;
  private OutputStream os = null;
  boolean DEBUG = true;

  public ClientCommExit (String h, int p, OutputStream o) {
    host = h;
    port = ((p == 0) ? DEFAULT_PORT : p);
    os = o;
    Socket s = null;
    PrintWriter out = new PrintWriter (os, true);
    if (DEBUG) {
      System.out.println("Applet about to create a socket on "
                         + host + " at port " + port);
    }

    try {
      // create a socket to communicate to the specified host and port
      s = new Socket(host, port);
      // create streams for reading and writing
      BufferedReader sin = new BufferedReader(new
InputStreamReader(s.getInputStream()));
      PrintStream sout = new PrintStream(s.getOutputStream(), true);
      if (DEBUG) {
        System.out.println("Applet has created sin and sout ");
```

```
      }

      // tell the user that we've connected
      out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());

      if (DEBUG) {
        System.out.println("Applet has connected to "+ s.getInetAddress() +
           ":" + s.getPort());
      }

      String line;
      // read the first response (a line) from the server
      line = sin.readLine();
      if (DEBUG) {
        System.out.println("Applet has read a line: " + line);
      }

      // write the line to the user
      out.println(line);
      out.flush();
      // send the command choice to the server
      sout.println("Server Exit");
      if (DEBUG) {
        System.out.println("Applet has sent sout the command: Server Exit");
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

Create the Java program that implements the "*ClientCommSQL*" class used in the applet: "*ClientCommSQL.java*". This program deals with the special applet commands for SQL statements.

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

public class ClientCommSQL {
  public static final int DEFAULT_PORT = 6789;
  private String host = "";
  private int port = 0;
  private OutputStream os = null;
  boolean DEBUG = true;

  public ClientCommSQL (String h, int p, int choice, String cmd, OutputStream
o) {
    host = h;
    port = ((p == 0) ? DEFAULT_PORT : p);
    os = o;
    Socket s = null;
    PrintWriter out = new PrintWriter (os, true);
```

```
    if (DEBUG) {
      System.out.println("Applet about to create a socket on "
          + host + " at port " + port);
    }

    try {
      // create a socket to communicate to the specified host and port
      s = new Socket(host, port);
      // create streams for reading and writing
      BufferedReader sin = new BufferedReader(new
InputStreamReader(s.getInputStream()));
      PrintStream sout = new PrintStream(s.getOutputStream(), true);
      if (DEBUG) {
        System.out.println("Applet has created sin and sout ");
      }

      // tell the user that we've connected
      out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());

      if (DEBUG) {
        System.out.println("Applet has connected to "+ s.getInetAddress() +
            ":" + s.getPort());
      }

      String line;
      // read the first response (a line) from the server
      line = sin.readLine();
      if (DEBUG) {
        System.out.println("Applet has read a line: " + line);
      }

      // write the line to the user
      out.println(line);
      out.flush();
      // send the command choice to the server
      if (choice ==4) {
        sout.println(choice);
        sout.println(cmd);
      } else {
        sout.println("Wrong command");
      }
      if (DEBUG) {
        System.out.println("Applet has sent sout the choice/SQL: "+ choice+
"/"+cmd);
      }

      // read a line from the server
      line = sin.readLine();
      if (DEBUG) {
        System.out.println("Applet has read a line: " + line);
      }

      out.println(line);
      // check if connection is closed, i.e., EOF
      if (line == null) {
        out.println("Connection closed by server.");
      }
       while (true) {
        line=sin.readLine();
        if (DEBUG) {
          System.out.println("Applet has read a line: " + line);
        }
```

```
         out.println(line);
         if (line.equals("EndOfRecord")) break;
       }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

9.6.3. Prepare the Main Server Program

Create the main server program, "*SDB.java*". This program accepts applet connections and user commands and then dispatches the commands to individual processing programs accordingly.

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

public class SDB {

  public static void main (String args[]) throws IOException {
    Socket client;
    int port = 0;
    int end = 0;
    BufferedReader in;
    PrintStream out;

    if (args.length != 1)
      port = 6789;
    else
      port = Integer.parseInt(args[0]);

    try {
      while (end == 0) {
        client = accept (port);
        in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
        out = new PrintStream(client.getOutputStream());
        out.println ("You are now connected to the Simple Database Server.");
         // read a line
         String line = in.readLine();
        // and send back ACK
         // out.println("OK");
         System.out.println("Received: " + line);
        if (line.equals("1")) DispCus.DispCus(out);
        else if (line.equals("2")) DispPro.DispPro(out);
        else if (line.equals("3")) DispTra.DispTra(out);
        else if (line.equals("4")) {
           out.println("OK");
           line = in.readLine();
          System.out.println ("Received: " + line);
```

```
          ExeSQL.ExeSQL(out, line);
        }
         if (line.equals("Server Exit")) {
          end = 1;
        }
        client.close();
      }
    } finally {
      System.out.println ("Closing");
    }
  }

  static Socket accept (int port) throws IOException {
    System.out.println ("Starting on port " + port);
    ServerSocket server = new ServerSocket (port);

    System.out.println ("Waiting");
    Socket client = server.accept ();
    System.out.println ("Accepted from " + client.getInetAddress ());

    server.close ();
    return client;
  }
}
```

9.6.4. Prepare the Database Access Programs

Create the Java program, "*DispCus.java*" to display the customer table.

```
import java.net.*;
import java.io.*;
import java.sql.*;
public class DispCus  {
       public static void DispCus(PrintStream out) {
              String url = "jdbc:odbc:dbtest";
              Connection con;
              String query = "select * from Customer ";
              Statement stmt;
              try {
                     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
              } catch(java.lang.ClassNotFoundException e) {
                     System.err.print("ClassNotFoundException: ");
                     System.err.println(e.getMessage());
              }
              try {
                     con = DriverManager.getConnection(url, "admin", "admin");
                     stmt = con.createStatement();

                     ResultSet rs = stmt.executeQuery(query);
                     ResultSetMetaData rsmd = rs.getMetaData();
                     int numberOfColumns = rsmd.getColumnCount();
                     int rowCount = 1;
                     while (rs.next()) {
                            out.println("Row " + rowCount + ":  ");
                            for (int i = 1; i <= numberOfColumns; i++) {
                                   out.print("   Column " + i + ":  ");
                                   out.println(rs.getString(i));
                            }
                            out.println("");
                            rowCount++;
                     }
```

```
                    out.println("EndOfRecord");
                    stmt.close();
                    con.close();

             } catch(SQLException ex) {
                    System.err.print("SQLException: ");
                    System.err.println(ex.getMessage());
             }
      }
}
```

Create the Java program, "*DispPro.java*" to display the product table.

```
import java.net.*;
import java.io.*;
import java.sql.*;
public class DispPro  {
      public static void DispPro(PrintStream out) {
             String url = "jdbc:odbc:dbtest";
             Connection con;
             String query = "select * from Product ";
             Statement stmt;
             try {
                    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
             } catch(java.lang.ClassNotFoundException e) {
                    System.err.print("ClassNotFoundException: ");
                    System.err.println(e.getMessage());
             }
             try {
                    con = DriverManager.getConnection(url, "admin", "admin");
                    stmt = con.createStatement();

                    ResultSet rs = stmt.executeQuery(query);
                    ResultSetMetaData rsmd = rs.getMetaData();
                    int numberOfColumns = rsmd.getColumnCount();
                    int rowCount = 1;
                    while (rs.next()) {
                           out.println("Row " + rowCount + ":  ");
                           for (int i = 1; i <= numberOfColumns; i++) {
                                  out.print("   Column " + i + ":  ");
                                  out.println(rs.getString(i));
                           }
                           out.println("");
                           rowCount++;
                    }
                    out.println("EndOfRecord");
                    stmt.close();
                    con.close();

             } catch(SQLException ex) {
                    System.err.print("SQLException: ");
                    System.err.println(ex.getMessage());
             }
      }
}
```

Create the Java program, "*DispTra.java*" to display the transaction table.

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

```
public class DispTra  {
	public static void DispTra(PrintStream out) {
		String url = "jdbc:odbc:dbtest";
		Connection con;
		String query = "select * from Transaction ";
		Statement stmt;
		try {
			Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
		} catch(java.lang.ClassNotFoundException e) {
			System.err.print("ClassNotFoundException: ");
			System.err.println(e.getMessage());
		}
		try {
			con = DriverManager.getConnection(url, "admin", "admin");
			stmt = con.createStatement();

			ResultSet rs = stmt.executeQuery(query);
			ResultSetMetaData rsmd = rs.getMetaData();
			int numberOfColumns = rsmd.getColumnCount();
			int rowCount = 1;
			while (rs.next()) {
				out.println("Row " + rowCount + ":  ");
				for (int i = 1; i <= numberOfColumns; i++) {
					out.print("   Column " + i + ":  ");
					out.println(rs.getString(i));
				}
				out.println("");
				rowCount++;
			}
			out.println("EndOfRecord");
			stmt.close();
			con.close();

		} catch(SQLException ex) {
			System.err.print("SQLException: ");
			System.err.println(ex.getMessage());
		}
	}
}
```

Create the Java program, "*ExeSQL.java*" to execute an SQL statement.

```
import java.net.*;
import java.io.*;
import java.sql.*;
public class ExeSQL  {
	public static void ExeSQL(PrintStream out, String sqlstr) {
		String url = "jdbc:odbc:dbtest";
		Connection con;
		Statement stmt;
		try {
			Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
		} catch(java.lang.ClassNotFoundException e) {
			System.err.print("ClassNotFoundException: ");
			System.err.println(e.getMessage());
		}
		try {
			con = DriverManager.getConnection(url, "admin", "admin");
			stmt = con.createStatement();

			ResultSet rs = stmt.executeQuery(sqlstr);
			ResultSetMetaData rsmd = rs.getMetaData();
```

```
                    int numberOfColumns = rsmd.getColumnCount();
                    int rowCount = 1;
                    while (rs.next()) {
                           out.println("Row " + rowCount + ":  ");
                           for (int i = 1; i <= numberOfColumns; i++) {
                                  out.print("   Column " + i + ":  ");
                                  out.println(rs.getString(i));
                           }
                           out.println("");
                           rowCount++;
                    }
                    out.println("EndOfRecord");
                    stmt.close();
                    con.close();

             } catch(SQLException ex) {
                    System.err.print("SQLException: ");
                    System.err.println(ex.getMessage());
                    out.println("No result.");
                    out.println("EndOfRecord");

             }
       }
}
```

9.6.5. Compile and Test the Programs

The following steps are used to compile and execute the example:

- Compile all the Java programs.
- Execute the server program *SDB* class first.
- Execute the applet via the *applet.html* using the *appletviewer* browser.

Note the server's host IP address is hard-coded into the *ClientApplet.java* program. It can be changed to any host address that the server is running. Of course, the applet program has to be re-compiled. This address can be easily entered as a parameter of the program.

Students are required to understand the working principles of the program and test run the example. Then, students are required to change the server program to deal with multiple client requests simultaneously.

10. Developing Distributed Applications using Java RMI and CORBA

10.1. Study Points

- Understand the basic concepts of RMI.
- Be able to build client-server applications using RMI.
- Understand the basic concepts in CORBA.

References: (1). [JNP] Chapter 18. (2). [HSH]: Chapters 23, 24, 25; (3). [FAR] Chapter 3. (4). [Java2U] Chapters 38, 39, 40. (5). [Java2H]: Chapters 12, 13.

10.2. Web-Based Client-Server Computing

We can categorise Web-based client-server computing systems into four types: the *proxy computing* model, the *code shipping* model, the *remote computing* model and the *agent-based computing* model.

10.2.1. The Proxy Computing Model

The proxy computing (PC) model is typically used in Web-based scientific computing. According to this model the client sends data and program to the server over the Web and requests the server to perform the computing. The server receives the request, performs the computing using the program and data supplied by the client and returns the result back to the client. Typically, the server is a powerful high-performance computer or it has some special system programs (such as special mathematical and engineering libraries) that are necessary for the computing. The client is mainly used for interfacing with the user. Figure 10.1 depicts this model.

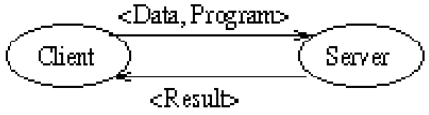

Figure 10.1. The proxy computing model

10.2.2. The Code Shipping Model

The code shipping (CS) model is a popular Web-based client-server computing model. A typical example is the downloading and then execution of Java applets on Web browsers, such as Netscape Communicator and Internet Explorer. According to this model, the client makes a request to the server, the server then ships the program (e.g., the Java applets) over the Web to the client and the client executes the program (possibly) using some local data. The server acts as the repository of programs and clients perform the computation and interface with the user. Figure 10.2 illustrates this model.

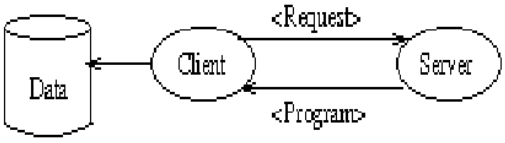

Figure 10.2. The code shipping model

10.2.3. The Remote Computing Model

The remote computing (RC) model is typically used in Web-based scientific computing and database applications. According to this model, the client sends data over the Web to the server and the server performs the computing using programs residing in the server. After the completion of the computation, the server sends the result back to the client. Typically the server is a high-performance computing server equipped with the necessary computing programs and/or databases. The client is responsible for interfacing with the user. The NetSolve system uses this model. Figure 10.3 depicts this model.

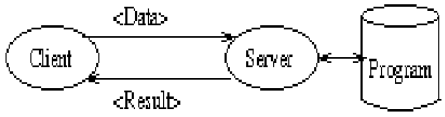

Figure 10.3. The remote computing model

10.2.4. The Agent-Based Computing Model

The agent-based computing (AC) model is a three-tier model. According to this model, the client sends either data or data and programs over the Web to the agent. The agent then processes the data using its own programs or using the received programs. After the completion of the processing, the agent will either send the result back to the client if the result is complete, or send the data/program/midium result to the server for further processing. In the latter case, the server will perform the job and return the result back to the client directly (or via the agent). Nowadays, more and more Web-based applications have shifted to the AC model. Figure 10.4 shows this model.

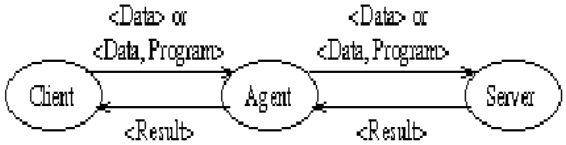

Figure 10.4. The agent-based computing model

10.3. RMI Overview

10.3.1. RMI Architecture

Java Remote Method Invocation (RMI) is a simple, yet powerful, Java-based framework for distributed object design. A remote invocation is a form of the RPC, where procedures can be invoked from remote machines. Java RMI extends the RPC further to

the distributed objects' world. RMI permits executing methods of objects residing in remote machines, with results returned to the calling environment.

RMI is a higher level abstraction than servlets and servers. We typically develop an application level protocol to communicate between Java clients and servers, but with RMI we do not need to do this. RMI is as simple as invoking a method of an object. RMI takes care of communication details for us.

Figure 10.5 shows the RMI architecture in which a Java client invokes a remote Java server object.

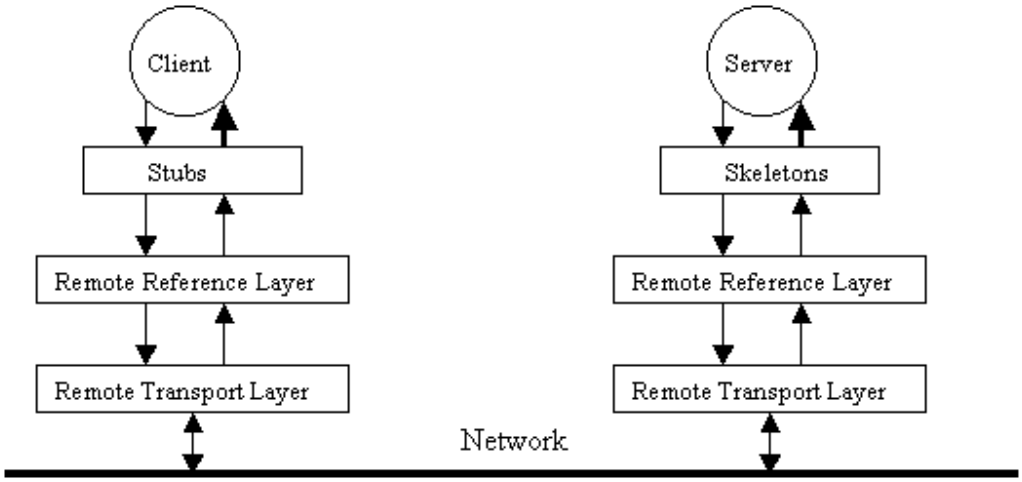

Figure 10.5 The RMI architecture

A client RMI call invokes the client-side stub (the proxy of the remote method that resides on the client's machine). The stub uses *Object Serialization* to marshal the arguments, i.e., render argument object values into a stream of bytes that can be transmitted over a network. The stub then passes control to the Java Virtual Machine's RMI layer. The skeleton on the server side dispatches the call to the actual remote object after unmarshaling the arguments into variables in memory. The stub and skeleton programs are generated by the *rmic* compiler.

The Remote Reference layer permits various protocols for remote invocation, such as unicast point-to-point (the one currently has been implemented).

Before a remote object can be accessed, it has to be registered into the naming server. The RMI framework provides a simple naming service. Remote objects can register to the naming server using the *java.rmi.Naming* class using a URL-like naming scheme.

10.3.2. Implementing Distributed Programs using RMI

The key interfaces and classes in RMI are:

- *Remote*. An interface in *java.rmi* package. It defines all remote interfaces.
- *RemoteObject*. It is a class. RMI server functions are provided by the *RemoteObject* and its subclasses.

- *RemoteServer*. It is a subclass of *RemoteObject*.
- *UnicastRemoteObject*. It is a subclass of *RemoteObject* in the *java.rmi.server* package.
- *RemoteException*. A class in *java.rmi* package, used for RMI to throw exceptions at runtime.

RMI implementation involves the following steps:

- Defining the remote interface: This is the interface through which remote clients will access the server and is done by extending the *Remote* interface and defining methods that can be invoked remotely.
- Implementing the remote interface. Remote method calls will ultimately be made upon their implementations. The interface is normally implemented via the extending of the *unicastRemoteObject* class. The *unicastRemoteObject* class defines a remote object which is valid when the server is running. This object hides the implementation of the interface from the public interface and can contain some methods that are not visible through the interface. Any remote object passed as an argument to RMI must also be defined as an interface.
- Create stubs and skeletons using *rmic* compiler.
- Compile the remote interface and implementation file using *javac* compiler.
- Create a client program, either a pure Java application or an applet and the HTML page to invoke the server services.

10.4. Simple RMI Examples

10.4.1. A Date Service

In this example we create a simple date server to provide the date and time information to the clients. The first step is to define the server interface, named *DateServer*, that lists all methods a client can call. In this example, only one method is defined. Here is the Java program *DateServer.java*:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface DateServer extends Remote {
  public Date getDate () throws RemoteException;
}
```

Note that the *getDate()* method must throw a *RemoteException* to allow the program to detect problems occurred in remote invocation.

The second step is to implement the remote object interface, through the program *DateServerImpl.java*:

```
import java.rmi.*;
import java.rmi.server.*;
```

```
import java.util.Date;

public class DateServerImpl extends UnicastRemoteObject implements DateServer {
  public DateServerImpl () throws RemoteException {
  }

  public Date getDate () {
    return new Date ();
  }

  public static void main (String[] args) throws Exception {
    DateServerImpl dateServer = new DateServerImpl ();
    System.out.println("Registering to Name Server");
    Naming.bind ("Date Server", dateServer);
    System.out.println("Registered!");
  }
}
```

All remote object implementation must extend *RemoteObject* or one of its subclasses (such as the *UnicastRemoteObject*, provided by the JDK for implementing TCP-based client-server programs). The *getDate* method simply returns the date information of the server host. The *main* method creates a new *DateServerImpl* named *dateServer* and registers it to the RMI naming registry using the name of "Date Server". If the name is already registered, then an *AlreadyBoundException* will be raised. To overcome this, we could use the *rebind* method instead of the *bind* method.

The third step is to generate the stub and the skeleton programs:

```
rmic DateServerImpl
```

The two classes (*DateServerImpl_Stub.class* and *DateServerImpl_Skel.class*) will be generated after the compilation.

The fourth step is to create the client program to access the services provided by the server. The client is named *DateClient.java*:

```
import java.rmi.Naming;
import java.util.Date;

public class DateClient {
  public static void main (String[] args) throws Exception {
    if (args.length != 1)
      throw new IllegalArgumentException ("Syntax: DateClient <hostname>");
    DateServer dateServer = (DateServer) Naming.lookup
      ("rmi://" + args[0] + "/Date Server");
    Date when = dateServer.getDate ();
    System.out.println (when);
  }
}
```

The client program uses the *lookup* method of *java.rmi.Naming* to get the information of the "Date Server" from the registry. The *lookup* method has two parameters, one provides the location of the registry and the other provides the name of the server.

The last step of using this simple RMI program is to run it via the following executions:

- Start the registry: *rmiregistry*
- Start the server: java DateServerImpl

- Start the client: *java DateClient localhost*

10.4.2. A Demo Service

The second simple example uses the object-oriented approach to demonstrate the building of RMI distributed objects. Remote objects are referenced via interfaces. In order to implement a remote object, you must first create an interface for that object. This interface must be public and must extend the Remote interface. Define the remote methods that you want to invoke within this interface. These methods must throw RemoteException.

The following MyServer.java program defines two methods: getDataNum() and getData(). The getDataNum() method returns an integer indicating the total number of data strings that are available on the server. The getData() method returns the *n*th data string.

```
import java.rmi.*;
public interface MyServer extends Remote {
 int getDataNum() throws RemoteException;
 String getData(int n) throws RemoteException;
}
```

Compile this program using

```
javac MyServer.java.
```

After creating the remote interface, you must create a class that implements the remote interface. This class typically extends the UnicastRemoteObject class. However, it could also extend other subclasses of the RemoteServer class.

The implementation class should have a constructor that creates and initializes the remote object. It should also implement all of the methods defined in the remote interface. It should have a main() method so that it can be executed as a remote class. The main() method should use the setSecurityManager() method of the System class to set an object to be used as the remote object's security manager. It should register a name by which it can be remotely referenced with the remote registry

The following MyServerImpl.java program provides the implementation class for the MyServer interface. You should change the hostName value to the name of the host where the remote object is to be located.

The data array contains five strings that are retrieved by the client object via the getDataNum() and getData() methods. The getDataNum() method returns the length of data, and the getData() method returns the *n*th element of the data array.

The main() method sets the security manager to an object of the RMISecurityManager class. It creates an instance of the MyServerImpl class and invokes the rebind() method of Naming to register the new object with remote registry. It registers the object with the name MyServer and then informs you that it has successfully completed the registration process

```
import java.rmi.*;
import java.rmi.server.*;
```

```
public class MyServerImpl extends UnicastRemoteObject
 implements MyServer {
 static String hostName="localhost";
 static String data[] = {"Remote","Method","Invocation","Is","Great!"};
 public MyServerImpl() throws RemoteException {
  super();
 }
 public int getDataNum() throws RemoteException {
  return data.length;
 }
 public String getData(int n) throws RemoteException {
  return data[n%data.length];
 }
 public static void main(String args[]){
  try {
   MyServerImpl instance = new MyServerImpl();
   Naming.rebind("//"+hostName+"/MyServer", instance);
   System.out.println("I'm registered!");
  } catch (Exception ex) {
   System.out.println(ex);
  }
 }
}
```

Compile this program using the RMI compiler:

```
rmic MyServerImpl
```

The rmic compiler creates the MyServerImpl_Stub.class and MyServerImpl_Skel.class files in the current directory.

You'll need the MyServer.class interface file to compile your client software, and you'll need MyServer.class and MyServerImpl_Stub.class to run your client. Before going any further, you should copy these files to an appropriate location on your client host.

Now you must start your remote registry server on your server host (use `start rmiregistry` in Windows or `rmiregistry&` in Unix). This program listens on the default port 1099 for incoming requests to access named objects. The named objects must register themselves with the remote registry program in order to be made available to requesters. When you execute the MyServerImpl program (use `java MyServerImpl`), it creates an object of the MyServerImpl class and registers the object with the remote registry.

Now we have the remote server up and running, the following program MyClient.java remotely invokes the methods of the MyServer object and displays the results it returns. You must change the hostName variable to the name of the remote server host where the remote object is registered.

MyClient consists of a single main() method that invokes the lookup() method of the Naming class to retrieve a reference to the object named MyServer on the specified host. It casts this object to the MyServer interface. It then invokes the getDataNum() method of the remote object to retrieve the number of available data items, and the getData() method to retrieve each specific data item. The retrieved data items are displayed in the console window.

```
import java.rmi.*;
public class MyClient {
```

```
 static String hostName="localhost";
 public static void main(String args[]) {
  try {
   MyServer server = (MyServer) Naming.lookup("//"+hostName+"/MyServer");
   int n = server.getDataNum();
   for(int i=0;i<n;++i) {
    System.out.println(server.getData(i));
   }
  } catch (Exception ex) {
   System.out.println(ex);
  }
 }
}
```

Compile this program using the RMI compiler:

```
javac MyClient.java.
```

Now you can run your client to access the service provided by the server.

10.5. Interfaces and Classes from RMI-Related Packages

The main packages for the RMI framework are: *java.rmi, java.rmi.server, java.rmi.registry, java.rmi.dgc*, and *java.rmi.activation* (Java 1.2).

- java.rmi--Provides the Remote interface, a class for accessing remote names, the MarshalledObject class, and a security manager for RMI
- java.rmi.registry--Provides classes and interfaces that are used by the remote registry
- java.rmi.server--Provides the classes and interfaces used to implement remote objects, stubs, and skeletons, and to support RMI communication. This package implements the bulk of the RMI API
- java.rmi.activation--Supports persistent object references and remote object activation
- java.rmi.dgc--Provides classes and interfaces that are used by the RMI distributed garbage collector

10.5.1. The java.rmi Package

The java.rmi package declares the Remote interface, the MarshalledObject, Naming and RMISecurityManager classes, and a number of exceptions that are used with remote method invocation.

The Remote interface must be implemented by all remote objects. This interface has no methods. It is used for identification purposes.

The MarshalledObject class was added in JDK 1.2. It is used to maintain a serialized byte stream of an object. Its get() method is used to retrieve a deserialized version of the object.

The Naming class provides static methods for accessing remote objects via RMI URLs. The bind() and rebind() methods bind a remote object name to a specific RMI URL. The unbind() method removes the binding between an object name and an RMI URL. The lookup() method returns the remote object specified by an RMI URL. The list() method

returns the list of URLs that are currently known to the RMI registry. The syntax for RMI URLs is as follows:

```
rmi://host:port/remoteObjectName
```

The *host* and TCP *port* are optional. If the *host* is omitted, the local host is assumed. The default TCP *port* is 1099.

The RMISecurityManager class defines the default security policy used for remote object stubs. It only applies to applications. Applets use the AppletSecurityManager class even if they perform RMI. You can extend RMISecurityManager and override its methods to implement your own RMI security policies. Use the setSecurityManager() method of the System class to set an RMISecurityManager object as the current security manager to be used for RMI stubs.

The java.rmi package defines a number of exceptions. The RemoteException class is the parent of all exceptions that are generated during RMI. It must be thrown by all methods of a remote object that can be accessed remotely.

10.5.2. The java.rmi.registry Package

The java.rmi.registry package provides the Registry and RegistryHandler interfaces and the LocateRegistry class. These interfaces and classes are used to register and access remote objects by name. Remote objects are registered when they are identified to a host's registry process. The registry process is created when the rmiregistry program is executed.

The Registry interface defines the bind(), rebind(), unbind(), list(), and lookup() methods that are used by the Naming class to associate object names and RMI URLs. The registry interface also defines the REGISTRY_PORT constant that identifies the default TCP port used by the registry service.

The RegistryHandler interface provides methods for accessing objects that implement the Registry interface. The registryStub() method returns the local stub of a remote object that implements the Registry interface. The registryImpl() method constructs a Registry object and exports it via a specified TCP port.

The LocateRegistry class provides the static getRegistry() method for retrieving Registry objects on the local host or a remote host. It also provides the createRegistry() method to construct a Registry object and export it via a specified TCP port.

10.5.3. The java.rmi.server Package

The java.rmi.server package implements several interfaces and classes that support both client and server aspects of RMI.

The RemoteObject class implements the Remote interface and provides a remote implementation of the Object class. All classes that implement remote objects, both client and server, extend RemoteObject.

The RemoteServer class extends RemoteObject and is a common class that is subclassed by specific types of remote object implementations. It provides the static setLog() and

getLog() methods for setting and retrieving an output stream used to log information about RMI accesses. It also provides the getClientHost() method that is used to retrieve the host name of the client performing the remote method invocation.

The UnicastRemoteObject class extends RemoteServer and provides the default remote object implementation. Classes that implement remote objects usually subclass UnicastRemoteObject. Objects of the UnicastRemoteObject class are accessed via TCP connections on port 1099, exist only for the duration of the process that creates them, and rely on a stream-based protocol for client/server communication.

The RemoteStub class extends RemoteObject and provides an abstract implementation of client side stubs. A *client stub* is a local representation of a remote object that implements all remote methods of the remote object. The static setRef() method is used to associate a client stub with its corresponding remote object.

The RemoteCall interface provides methods that are used by stubs and skeletons to implement remote method invocations.

The RemoteRef interface is used by RemoteStub objects to reference remote objects. It provides methods for comparing and invoking remote objects and for working with objects that implement the RemoteCall interface.

The ServerRef interface extends the RemoteRef interface and is implemented by remote objects to gain access to their associated RemoteStub objects.

The Skeleton interface is implemented by remote skeletons. It provides methods that are used by the skeleton to access the methods being requested of the remote object, and for working with method arguments and return values.

The Unreferenced interface is implemented by a remote object to enable it to determine when it is no longer referenced by a client.

The RMIClassLoader class supports the loading of remote classes. The location of a remote class is specified by either an URL or the java.rmi.server.codebase system property. The static loadClass() method loads a remote class, and the static getSecurityContext() returns the security context in which the class loader operates. The LoaderHandler interface defines methods that are used by RMIClassClassLoader to load classes.

The Operation class is used to store a reference to a method. The getOperation() method returns the name of the method. The toString() method returns a String representation of the method's signature.

The ObjID class is used to create objects that serve as unique identifiers for objects that are exported as remote by a particular host. It provides methods for reading the object ID from and writing it to a stream. The UID class is an abstract class for creating unique object identifiers.

The LogStream class extends the PrintStream class to support the logging of errors that occur during RMI processing.

The RMISocketFactory class is used to specify a socket implementation for transporting information between clients and servers involved in RMI. This class provides three

alternative approaches to establishing RMI connections that can be used with firewalls. The static setSocketFactory() method can be used to specify a custom socket implementation. The RMIClientSocketFactory and RMIServerSocketFactory interfaces provide support for both client and server sockets. The RMIFailureHandler interface defines methods that handle the failure of a server socket creation. The RMIFailureHandler interface provides the failure() method for handling exceptions that occur in the underlying RMI socket implementation.

10.5.4. The java.rmi.activation Package

The java.rmi.activation package is a new RMI package that was added to JDK 1.2. It provides the capabilities to activate remote objects as needed and to use persistent object references.

The Activatable class defines the basic methods implemented by activatable, persistent objects. It contains two constructors. One constructor is used to create and register (with the activation system) objects that can be accessed via specific TCP ports. The other constructor is used to activate an object based upon an ActivationID object and persistent data that has been stored for that object. The export() object methods are used to make an object available for use via a specific TCP port. The getID() method returns an object's ActivationID (used to uniquely identify the object). The register() and unregister() methods register (and unregister) an object with the runtime system. The inactive() method is used to tell the activation system that an object is inactive, or, if active, that it should be deactivated.

Objects of the ActivationID class are used to uniquely identify activatable objects and contain information about how objects are to be activated. An object's ActivationID is created when the object is registered with the activation system. The activate() method is used to activate the object referenced by an ActivationID object. The equals() and hashCode() methods are used to compare two ActivationID objects. Two ActivationID objects are equal if they reference the same object.

The ActivationDesc class encapsulates the information necessary to activate an object. It provides five methods that can be used to retrieve this information. The getClassName() method returns the described object's class name. The getCodeSource() method returns a CodeSource object that identifies the described object's location and other source information. The getData() method returns a MarshalledObject object that contains serialized information used to initialize the described object. The getGroupID() method returns the described object's ActivationGroupID object. The getRestartMode() method returns the restart mode associated with the activation descriptor.

The ActivationGroup class is used to group activatable objects so that they execute in the same JVM. ActivationGroup objects are used to create instances of the activatable objects within their group. The activeObject() method is used to inform an ActivationGroup that an activatable object has been activated. The createGroup() method is used to specify the current ActivationGroup object for the current JVM instance. The currentGroupID() method returns the ActivationGroupID object of the current ActivationGroup object. The getSystem() method returns the current ActivationSystem object. The inactiveObject() method is invoked when an object in the group is deactivated

(becomes inactive). This method deactivates the object if the object is still active. The inactiveGroup() method is used to report an inactive group to the group's ActivationMonitor object. The newInstance() method creates a new instance of an activatable object. The setSystem() method sets the ActivationSystem object for the current JVM.

The ActivationGroupID class uniquely identifies an ActivationGroup object and contains information about the object's activation system. The getSystem() method returns the ActivationSystem object that is used to activate the referenced ActivationGroup object. The equals() and hashCode() methods are used to compare ActivationGroupID objects in terms of their referenced ActivationGroupID objects.

The ActivationGroupDesc class encapsulates the information necessary to create an ActivationGroup object. The getClassName() method returns the described ActivationGroup object's class name. The getCodeSource() method returns the described ActivationGroup object's CodeSource object. The getData() method returns a MarshalledObject object that contains serialized data about the described ActivationGroup object. The CommandEnvironment inner class provides support for implementation-specific options.

The ActivationSystem interface is implemented by objects that register activatable objects and activatable object groups. The SYSTEM_PORT constant identifies the TCP port used by the activation system. The registerGroup(), registerObject(), unregisterGroup(), and unregisterObject() methods are used to register and unregister Activatable and ActivationGroup objects. The activeGroup() method is used to inform the activation system about an active ActivationGroup object.

The Activator interface is implemented by objects that activate objects that are registered with an ActivationSystem (object). The activate() method activates an object based upon its associated ActivationID object.

The ActivationInstantiator interface provides methods for classes that create instances of activatable objects. The newInstance() method creates new object instances based on their associated ActivationID and ActivationDesc objects.

The ActivationMonitor provides methods for maintaining information about active and inactive objects. The activeObject(), inactiveObjet(), and inactiveGroup() methods are used to collect this information.

10.5.5. The java.rmi.dgc Package

The java.rmi.dgc package contains classes and interfaces that are used by the distributed garbage collector. The DGC interface is implemented by the server side of the distributed garbage collector. It defines two methods: dirty() and clean(). The dirty() method indicates that a remote object is being referenced by a client. The clean() method is used to indicate that a remote reference has been completed.

The Lease class creates objects that are used to keep track of object references. The VMID class is used to create an ID that uniquely identifies a Java virtual machine on a particular host.

10.6. An Interesting RMI Application

The example is an implementation of the previous chat program (Section 8.5) using RMI. This program consists of three programs: *RMIChatClient.java, RMIChatServer.java* and *RMIChatServerImpl.java*.

10.6.1. The Chat Server and Its Implementation

The chat server program (*RMIChatServer.java*) is simple. It just defines the API through which the client can call the server:

```
import java.rmi.*;

public interface RMIChatServer extends Remote {
  public static final String REGISTRY_NAME = "Chat Server";
  public abstract String[] getMessages (int index) throws RemoteException;
  public abstract void addMessage (String message) throws RemoteException;
}
```

The implementation of the chat server is done by the program *RMIChatServerImpl.java*. This program provides the implementation of the remote API, instantiates the server, and register the server with the naming registry.

```
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class RMIChatServerImpl extends UnicastRemoteObject implements
RMIChatServer {
  protected Vector messages;

  public RMIChatServerImpl () throws RemoteException {
    messages = new Vector ();
  }

  public String[] getMessages (int index) {
    int size = messages.size ();
    String[] update = new String[size - index];
    for (int i = 0; i < size - index; ++ i)
      update[i] = (String) messages.elementAt (index + i);
    return update;
  }

  public void addMessage (String message) {
    messages.addElement (message);
  }

  public static void main (String[] args) throws RemoteException {
    RMIChatServerImpl chatServer = new RMIChatServerImpl ();
    Registry registry = LocateRegistry.getRegistry ();
    registry.rebind (REGISTRY_NAME, chatServer);
  }
}
```

10.6.2. The Chat Client

The chat client program (*RMIChatClient.java*) opens a typical chat `Frame`, locates the chat server, and proceeds to remotely access the server. A thread that regularly queries the server for new messages via the *getMessages()* method is used to receive update from the server.

```
import java.awt.*;
import java.rmi.*;
import java.awt.event.*;
import java.rmi.registry.*;

public class RMIChatClient implements Runnable, ActionListener {
  protected static final int UPDATE_DELAY = 10000;

  protected String host;
  protected Frame frame;
  protected TextField input;
  protected TextArea output;

  public RMIChatClient (String host) {
    this.host = host;

    frame = new Frame ("RMIChatClient [" + host + "]");
    frame.add (output = new TextArea (), "Center");
    output.setEditable (false);
    frame.add (input = new TextField (), "South");
    input.addActionListener (this);
    frame.addWindowListener (new WindowAdapter () {
      public void windowOpened (WindowEvent ev) {
        input.requestFocus ();
      }

      public void windowClosing (WindowEvent ev) {
        stop ();
      }
    });
    frame.pack ();
  }

  protected RMIChatServer server;
  protected Thread updater;

  public synchronized void start () throws RemoteException, NotBoundException {
    if (updater == null) {
      Registry registry = LocateRegistry.getRegistry (host);
      server = (RMIChatServer) registry.lookup (RMIChatServer.REGISTRY_NAME);
      updater = new Thread (this);
      updater.start ();
      frame.setVisible (true);
    }
  }

  public synchronized void stop () {
    if (updater != null) {
      updater.interrupt ();
      updater = null;
      server = null;
    }
    frame.setVisible (false);
```

```
  }

  public void run () {
    try {
      int index = 0;
      while (!Thread.interrupted ()) {
        String[] messages = server.getMessages (index);
        int n = messages.length;
        for (int i = 0; i < n; ++ i)
          output.append (messages[i] + "\n");
        index += n;
        Thread.sleep (UPDATE_DELAY);
      }
    } catch (InterruptedException ignored) {
    } catch (RemoteException ex) {
      input.setVisible (false);
      frame.validate ();
      ex.printStackTrace ();
    }
  }

  public void actionPerformed (ActionEvent ev) {
    try {
      RMIChatServer server = this.server;
      if (server != null)
        server.addMessage (ev.getActionCommand ());
      input.setText ("");
    } catch (RemoteException ex) {
      Thread tmp = updater;
      updater = null;
      if (tmp != null)
        tmp.interrupt ();
      input.setVisible (false);
      frame.validate ();
      ex.printStackTrace ();
    }
  }

  public static void main (String[] args) throws RemoteException,
NotBoundException {
    if (args.length != 1)
      throw new IllegalArgumentException ("Syntax: RMIChatClient <host>");
    RMIChatClient chatClient = new RMIChatClient (args[0]);
    chatClient.start ();
  }
}
```

10.7 CORBA

10.7.1. What is CORBA?

The following points list a brief answer to the question: “what is the Common Object Request Broker Architecture (CORBA)?”

- CORBA is a specification of an architecture and interface that allows an application to make request of objects (servers) in a transparent, independent manner, regardless of platform, operating system or locale considerations.
- CORBA was developed by the Object Management Group (OMG).

- The CORBA programming paradigm combines distributed client-server programming and object-oriented programming methodologies.
- CORBA is the specification of the Object Request Broker (ORB) component of the Object Management Architecture (OMA).
- The ORB is the message bus that facilitates object communications across distributed heterogeneous computing environment.
- The OMA provides fundamental models on which CORBA and other standard interfaces are based.

The Object management architecture (OMA) consists of the following key features:

- The Core Object Model: defines the concepts that allow distributed application development to be facilitated by an Object Request Broker (ORB).
    - objects
    - operations
    - non-object types
    - interfaces and substitutability
- The Reference Architecture: provides standardised interfaces for supporting application development.
    - The ORB
    - Object Services
    - Domain Interfaces
    - Common Facilities
    - Application Interfaces

Figure 10.2 illustrates the OMA Reference Architecture.

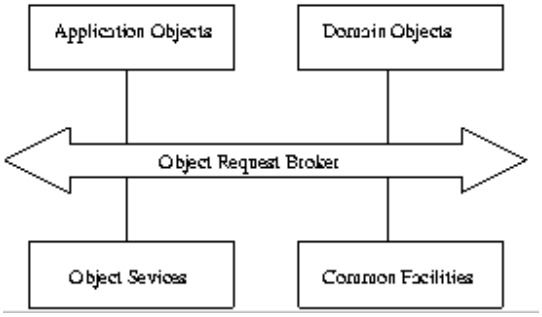

Figure 10.6 OMA Reference Architecture

10.7.2. The CORBA Architecture

CORBA supports the client-server programming:

- Client: makes requests to other components in the distributed application.
- Server: provides an implementation of a component that a client uses. A server can also acts as clients to other servers.
- Interface definition: describes the functionality of a CORBA object. Clients of CORBA objects rely only on the interface.
- CORBA servers: programs that provide the implementation of one or more CORBA objects.

Figure 10.3 illustrates the client-server interaction in CORBA

.

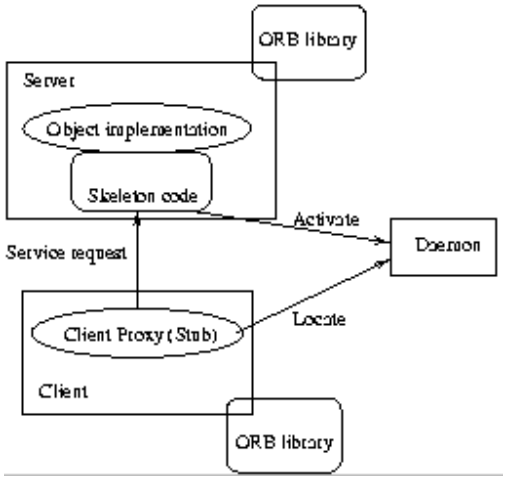

Figure 10.7 CORBA client and server

CORBA provides location transparency and programming language transparency via the use of OMG's Interface Definition Language (IDL).

CORBA Architecture can be viewed as following:

- IDL compiler generated code
  - Stub code: linked into a CORBA client
  - Skeleton code: linked into a CORBA object implementation
- An ORB agent/daemon process
- Library code
- Interfaces among ORB components. See Figure 10.4.

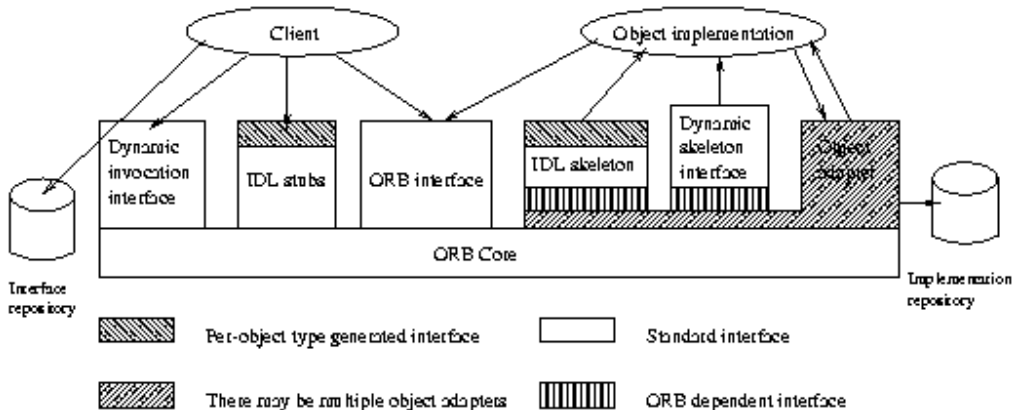

Figure 10.8 CORBA architecture

Steps to invoke operations on objects:

- The client request a service from the object implementation. The ORB transports the request which invokes the method using object adapters and the IDL skeleton.
- Objects are accessed through object references.

- Object references contain enough information to allow the ORB to locate the machine, the server, and the object to which the reference refers.
- Static invocation interface: the the IDL definition of an object reference is known at the compile time.
- Dynamic reference interface: the the IDL definition of an object reference is not known at the compile time. A request to the Interface Repository (IR) is used to discover the object's operations and parameters.
- Operation invocation semantics: at-most-once.

Interface definition language (IDL):

- IDL is designed to specify the functionality of objects.
- Programming language code is generated from the IDL definition to perform the tedious, error prone, and repetitive tasks of establishing network connections, marshaling, locating object implementations, and invoking the right code to perform an operation.
- Data types.
  - basic types
  - structure
  - discriminated union
  - array
  - sequence
  - exception
- Attributes and operations: actions declared in IDL that can be requested of a CORBA object.
- Inheritance: to extend the functionality of an existing interface. CORBA inheritance is independent of implementation inheritance, as shown in Figure 10.5.

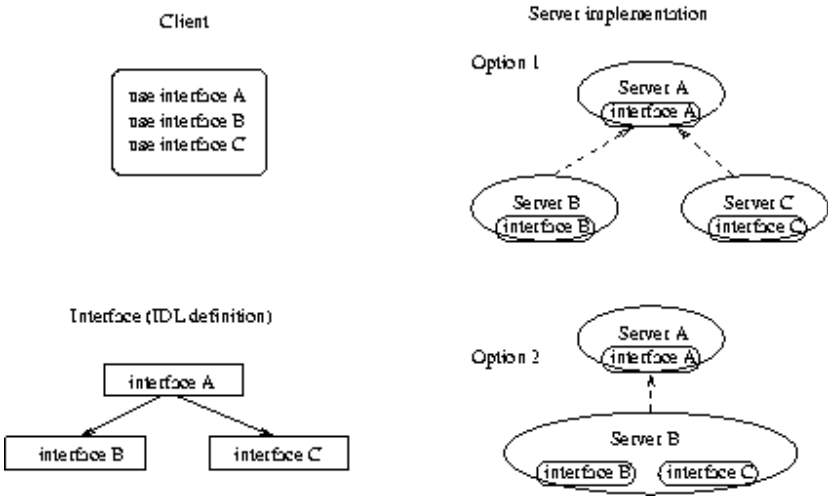

Figure 10.9. Interface inheritance and implementation inheritance

CORBA uses stubs and skeletons in much the same way as Java RMI. A *stub* is a local proxy for a remote object. It presents the same interface as the server object, but runs on the same computer as the client. A *skeleton* is a remote interface to the server object's

implementation. It runs on the same computer as the server object and provides an interface between the server object's implementation and other objects.

The stub and skeleton are connected via an ORB. The ORB forwards method invocations from the stub to the skeleton and uses a special object called an Object Adapter (OA), which runs on the same computers as the server object. The OA activates the server object, if required, and helps to manage its operation. You can think of the ORB as analogous to the remote reference and transport layers of Java RMI, and the OA as being like the remote registry. The OA is sometimes referred to as a *Basic Object Adapter (BOA)*.

10.7.3. The Interface Definition Language

Because one of CORBA's main goals is to provide a distributed object-oriented framework in which objects created in a variety of programming languages can interact, it needs a way to bridge the gap between multiple programming language interfaces and the ORB. IDL is the key to achieving this goal.

IDL provides a language-neutral way of describing the interfaces of objects. It describes an interface in the same manner that Java interfaces do. It defines the methods contained in an interface, their arguments, and their return values, but does not specify how the interfaces are implemented.

A server object's interface is specified in IDL, and the IDL specification is compiled to produce the stub and skeleton to be used for that object. For example, you could specify the interface of a server in IDL and then compile the IDL to create the C++ source code for the server's skeleton. You could also compile the same IDL to create a Java stub.

IDL compilers are available for C, C++, Smalltalk, Ada, and (of course) Java. These compilers translate IDL into stubs and skeletons in the source code of these languages. You then use a language-specific compiler to compile the stubs and skeletons to binary code or byte code.

In order to develop an IDL compiler for a particular language, a language mapping must be developed that shows how the datatypes and method invocation semantics of IDL map to the language. The Java language mapping has been completed and is available at http://splash.javasoft.com/products/jdk/idl/docs/idl-java.html. An IDL-to-Java compiler, aptly named idltojava, is also available from the Java Developer Connection at http://developer.javasoft.com. You should download idltojava to perform the example in this chapter. Copy the idltojava program to a directory that is in your execution path.

10.7.4. An Example of CORBA for Java

One commonly used CORBA for Java is the Orbix Web developed by IONA Technologies Ltd. The general syntax of CORBA IDL is as follows:

```
module <identifier> {
  <type declarations>;
  <constant declarations>;
  <exception declarations>;
  interface <identifier> [: <inheritance>] {
```

```
    <type declarations>;
    <constant declarations>;
    <attribute declarations>;
    <exception declarations>;
    [<operation_type>] <identifier> (<parameters>)
    [raises exception] [<context>};
    ...
    [<operation_type>] <identifier> (<parameters>)
    [raises exception] [<context>};
  }
  interface <identifier> [: inheritance>] {
    ...
  }
}
```

Below shows an example of a IDL definition for a grid application:

```
// IDL;
// in file grid.idl

interface Grid {

  readonly attribute short height;
  readonly attribute short width;

    void set(in short n, in short m, in long value);
    void get(in short n, in short m);
```

The interface provides two attributes, height and width which define the size of a grid. Since they are labeled *readonly*, they cannot be directly modified by a client. There are also two operations: the *set*() operation allows an element of grid to be set, and the *get*() operation returns an element. Parameters here are labeled as *in*, which means they are passed from the client to the server. Other labels can be *out* or *inout*.

The following command compiles the idl file:

```
idl grid.idl
```

After the compilation, the following files are generated and stored in a local directory *java_output*:

- *_GridRef.java*: A Java interface; the methods of this interface define the Java client view of the IDL interface.
- *Grid.java*: A Java class which implements the methods defined in interface *_GridRef*. This class provides functionality which allows client method invocations to be forwarded to a server.
- *_GridHolder.java:* A Java class which defines a Holder type for class *Grid*. This is required for passing *Grid* objects as *inout* or *out* parameters to and from IDL operations.
- *_GridOperations.java*: A Java interface which maps the attributes and operations of the IDL definition to Java methods. These methods must be implemented by a class in the server.

- *boaimpl_Grid.java*: An abstract Java class which allows server-side developers to implement the Grid interface using one of two techniques available in OrbixWeb; this technique is called the BOAImpl approach to interface implementation.
- *tieGrid.java*: A Java class which allows server-side developers to implement the Grid interface using one of two techniques available in OrbixWeb; this technique is called the TIE approach to interface implementation.
- *dispatcher_Grid.java*: A Java class used internally by OrbixWeb to dispatch incoming server requests to implementation objects. Application developers do not require an understanding of this class.

After the implementation of the *_GridOperations.java* program and a client program, the program can be compiled. Then the server should be registered to the registry by using the *putit* command. The client can now access the server from any machine since it knows the machine name, the server name and object names.

11. Java Servlets and Java Beans

11.1. Study Points

- Understand the basic concepts of Java servlets.
- Understand the basic concepts of Java Beans.

Reference: (1). http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html. (2). [HSH] Chapters 26 and 27. (3). [Java2U] Chapters 24, 25, 26, 27, 28, and 29.

11.2. Introduction of Servlets

11.2.1. What is a Servlet?

Servlets can be thought of as server-side applets that perform server-side tasks, sending responses to web clients. Servlets built on top of HTTP protocol are faceless objects without a visible user interface. Servlets were first made available with Sun's Jeeves web server. Servlets can be called from web page when a dynamic document is requested or can be invoked directly via a URL. The servelet API can be integrated with API components of JDBC, RMI, and IDL, making it possible to create sophisticated applications using pure Java services.

Servlets can be started when the server starts, or activated on demand when a client request comes in. Servlet can either be loaded across the network or from a local disk. After servlets are activated, they handle requests until the server dies or its *destroy*() method has been called.

The main difference of servlets and CGI scripts is that a CGI script requires a new process for every client request. This is a slow, expensive operation on many operating systems. Servlets, on contrast, keep running as a separate process and do not require overhead of starting and stoping additional processes for each request. If a site is performing many short-running CGI scripts, the cost of script invocation can be high. If a relatively small number of CGI hits is received and each performs a compute bound function in compiled code, CGI might outperform interpreted Java code.

CGI scripts are also platform dependent, and their code often needs to be re-written or recompiled for different platforms. However, their behavior is specified by the universally accepted HTTP protocol. Servlets, on the other hand, can be ported to any web server supporting Java and the servlet extension. However, web servers with servlet capabilities are not yet popular.

The Java server-side API used in Jeeves has three major components: Base server framework (package *sun.server*), HTTP server framework (package *sun.server.http*) and servlet application programming interface.

The base server architecture provides a framework for servers maintaining multiple threads for client connections. An acceptor thread accepts connections on a continuous basis. When a connection is made, it is sent to a handler thread.

The HTTP framework includes classes for handling standard HTTP functionality such as form processing (*FormServlet*), CGI requests (*CGIServlet*), server side includes

(*SSIncludeServlet*), file handling (*FileServlet*), and server-side image map requests (*ImagemapServlet*). The servlet application interface API (package *java.servlet*) provides classes for servlet customization.

11.2.2. Servlets Security and Applications

Four different types of servlets with different levels of trust exist: Jeeves servlets, local servlets, signed servlets, and unsigned servlets.

Jeeves servlets are part of the base server framework and are automatically granted full access rights. Local servlets are loaded from the local disks and are therefore trusted and granted full access rights. Signed servlets have either limited or full access rights according to the rules set by the Jeeves administrator. Finally, constraints are placed on unsigned servlets, which run in the server sandbox environment, which has limited access to server functionality. Servlets can be coded to require that the user enter an ID and password.

Servlets can be used for applications ranging from server administration to complex interactions between servlets and browser-side applets. Servlets can provide persistence and state data, making it possible to create interesting applications for transactions, in place of existing approaches like “cookies” and hidden fields.

11.2.3. Servlet Life Cycle

The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps:

1. If an instance of the servlet does not exist, the container:
    a. Loads the servlet class
    b. Instantiates an instance of the servlet class
    c. Initializes the servlet instance by calling the `init` method.
2. Invokes the `service` method, passing a request and response object.

If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's `destroy` method.

You can monitor and react to events in a servlet's life cycle by defining listener objects whose methods get invoked when life cycle events occur. To use these listener objects you must define and specify the listener class.

Any number of exceptions can occur when a servlet is executed. The web container will generate a default page containing the message `A Servlet Exception Has Occurred` when an exception occurs, but you can also specify that the container should return a specific error page for a given exception.

11.3. Developing Servlets

11.3.1. Downloading and Installing Servlets

To use servlets, you should download the J2SE 1.3, which is available from the Sun Microsystem's Web site: `http://java.sun.com/j2se/1.3/` and the java servlet classes, available from `http://java.sun.com/products/servlet/download.html`. You also need a web server capable of executing servlets. One of such a web server is the tomcat web server written in java and is available from the link `http://jakarta.apache.org/site/binindex.html`. You simply select the archive that suits your operating system to download.

To install the java servlet extension classes, you need to uncompress the downloaded archive and copy the *servlet.jar* to your library extensions folder. The extensions library folder is usually found under your JDK's jre folder. For example on a Linux installation the folder is found at */usr/java/jdk1.3/jre/lib/ext/*. Simply copy the files to this directory is all that is required.

To install tomcat, you need to uncompress the archive and copy it to a folder, for example "`c:\tomcat`" on windows and "`/var/tomcat`" for Unix/Linux. Set the environment variable `TOMCAT_HOME` to point to your tomcat folder. Under windows this would be

```
set TOMCAT_HOME=c:\tomcat
```

and under Unix/Linux would be

```
export TOMCAT_HOME=/var/tomcat
```

(or an equivalent shell command). Next, you need to set the `JAVA_HOME` environment variable in the same manner to point to your JDK's folder. With the environment variables correctly set you can start and stop the server using the startup and shutdown scripts under the bin directory. (*.sh for Unix, *.bat for Windows)

The installation can be checked by opening a web browser and opening the page `http://localhost:8080/` where you should get the default tomcat page and some examples.

11.3.2. Basic Techniques for Using Servlets

Section 26.3 of [HSH] provides basic information about what is needed to begin developing and testing servlets and section 26.14, section 25.22, and section 26.23 provide a number of simple servlet examples. Students are required to understand the basic steps in developing servlets applications and the working principles of the examples.

11.3.3. More Examples in using Servlets

Chapter 27 of [HSH] provides a number of advanced examples of servlets in action. It includes techniques for maintaining persistence data in servlets; a servlet example for counting the number of get accesses made to the servlet over its lifetime; techniques of using servlets to implement collaborative applications over the Internet; and an

application of distributed list implemented using servlets. Students are required to understand the working principle of these examples.

11.4. Java Beans

Java Beans (as well as ActiveX and OpenDoc) supports the notion of an application component model. A component model enables several different kinds of programmer parts to work together. By developing reusable components, you can preserve the effort you placed into software development by packaging them in modules that you can publish to others.

A “Bean” is a component made up of several other objects. By putting them all in one place, with a well-defined interface to the group, you can give them out to others to reuse. Beans brings object-oriented programming and Java into next big wave of computing: components.

11.4.1. Introduction of the Component Model

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Every component must have a well-defined standard interface or a set of standard interfaces and open Application Programming Interfaces (API) so it can be successfully used in a plug and play environment.

Components encapsulate semantically meaningful application or technical services and offer standard interfaces. In contrast to more conventional uses of components, e-commerce components require more attention to the issues of security, reliability, standards, and interaction. The leading component-based software development methodoligies are OMG's CORBA, Sun's JavaBeans and Microsoft's DCOM (Distributed Component Object Model) and ActiveX.

A framework for building component-based software should contain at least the following four elements:

- Objects: Objects are the minimum building blocks of components viewed from the component-based programming point of view. Each object has a unique identity, a set of data structures that can be used to hold the state of the object, and a set of operations that can be used to manipulate the data stored in the object’s data structure. An object cannot exist by its own as a useful program.
- Components: Components are independent software entities possess some program logic, ranging from small graphical user interface widgets such as buttons to complex components such as stock ticker display, to full-size applications such as word processors and spreadsheets.
- Containers: Containers are software entities used to assemble components. They provide a context in which components interact and may be arranged. Containers can be nested within other containers.

- Glues: Glues are scripting languages used to initiate and direct interactions between components and objects. They are used to describe the relationships among objects and components and to assemble objects and components into larger components or even applications within a container.

Figure 11.1. shows the use of components and containers.

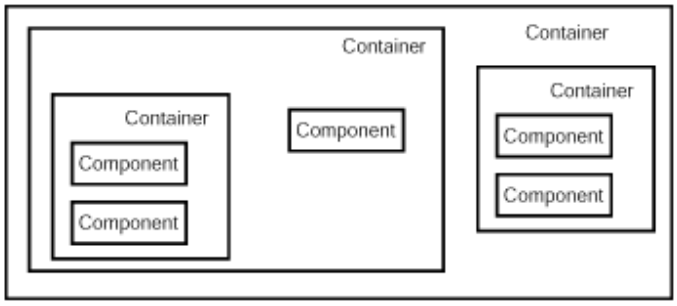

Figure 11.1. Components and containers.

It is essential to understand the following issues in building a framework for component-based programming:

- How to define and describe objects, components, and containers?
- What are the relationships among these elements?
- How to prove that a particular glue of certain objects and components is correct?

The framework for building component-based systems should provide at least the following essential services: component publishing and discovery service, event handling service, and an assembly line.

- Component publishing and discovery service: When a component is created or is placed in a container, it needs to identify itself and the interfaces it supports. The new component registers or publishes its existence and interfaces with the framework. As a consequence, other components learn through the discovery facility of its existence and how to interact with it. A number of questions must be answered to create such a service. For example, how to express and classify the information about components? How to store, update, and retrieve information about components? Will the service be centralised or distributed?
- Event handling service: Objects and components communicate with each other through messages. An object or a component raises or broadcasts a message or an event, and the framework is responsible for delivering the message to the appropriate objects or components. Messages may be generated by the system itself, for example, by a click of the mouse; or they may be generated by other objects, such as when a database record is changed. A number of research issues are associated to this service. For example, how to express events? How to effectively and efficiently raise and capture events in a Web-based environment? How to effectively handle events?
- An assembly line: Components can be assembled during run-time, or during development period. In both cases, an assembly line is required to perform the job. The assembly line enables objects and components to expose their properties and

behaviours to development tools or other components. The assembly line can provide mechanisms such as inspectors, editors, and debuggers for assembling components into applications. Many research issues are also associated to the development of the assembly line. For example, what scripting language(s) should be used during the assembly process? How do we know an assembly is correct? If there exist different assembly approaches, how can we select the one that will result in the most efficient component or application?

11.4.2. Overview of Beans Component Model

Every “Bean” should provide each of five different services designed to promote interaction between one another:

- Interface publishing
- Event handling
- Persistence
- Layout
- Builder support.

In order to enable one Bean to make another Bean do something, the Beans must have a published and pre-defined set of routines. When several Beans join together, they form a Java Beans application.

The component Beans of a Java Beans application must publish their interfaces to the container Bean application so that any Bean within the application can acquire a reference to a component. Other components may invoke the Bean and use it as it were intended.

Beans must be able to pass events to one another. A Beans application may have several applets. When something happens to one applet, the other applets may want to know. Beans components can be made to talk to one another and trigger events in each other. The powerful components model on top of which Beans was developed promotes the idea of object separation.

Persistence moves applications from a session-based paradigm in which objects are started, exist for a little while, and the disappear, to a lifecycle-based paradigm in which objects are started and exist for a little while. This time, however, instead of the object disappearing, it is saved, restored, and allowed to exist again afterwards. Java Beans supports persistence primarily through object serialization.

The Beans framework provides a set of routines to effectively lay out the various parts so that they don’t step on one another. The layout mechanism also allows the sharing of resources. The Beans layout mechanism allows you to position your Beans in rectanglar areas. The programmer is left to decide whether the regines overlap or maintain a discrete layout space.

The builder support in Beans provides a way that other builder applications can obtain a catalog of methods used by your Bean application, as well as proper means to access each individual Bean.

11.4.3. Downloading and Installing the BDK

The JavaBeans Development Kit (BDK) is freely available from the JavaBeans home page, located at http://java.sun.com/products/javabeans/software/bdk_download.html.

The BDK provides several examples of JavaBeans, a tutorial, and supporting documentation. But most important, it provides a tool, referred to as the BeanBox, that can be used to display, customize, and test the beans that you'll develop. The BeanBox also serves as a primitive visual development tool. You'll use the BeanBox to see the important aspects of visual component-based software development as it applies to JavaBeans. Download and install the BDK before continuing on to the next section. Once you've installed the BDK, restart your system to make sure that all installation changes take effect.

In the following section we assume that the BDK is installed in directory of `c:\bdk1.1\.`

11.5. Using Java Beans

11.5.1. Using the BeanBox

The BeanBox of the BDK is an example of a simple visual development tool for JavaBeans. It is located in the c:\bdk\beanbox directory. Change to this directory and start the BeanBox as follows:

```
c:\bdk1.1\beanbox>run
```

The BeanBox application loads and displays four windows labeled *ToolBox*, *BeanBox*, and *PropertySheet*, and *MethodBox*.

You should be impressed by how easy it was to develop an interesting (or at least entertaining) application using the BeanBox and JavaBeans. In fact, you didn't have to write a single line of code to create the application. That's the power of component-based software development. Given a good stock of beans, you can quickly and easily assemble a large variety of useful applications.

Figure 11.2 shows the four initial windows of DBK.

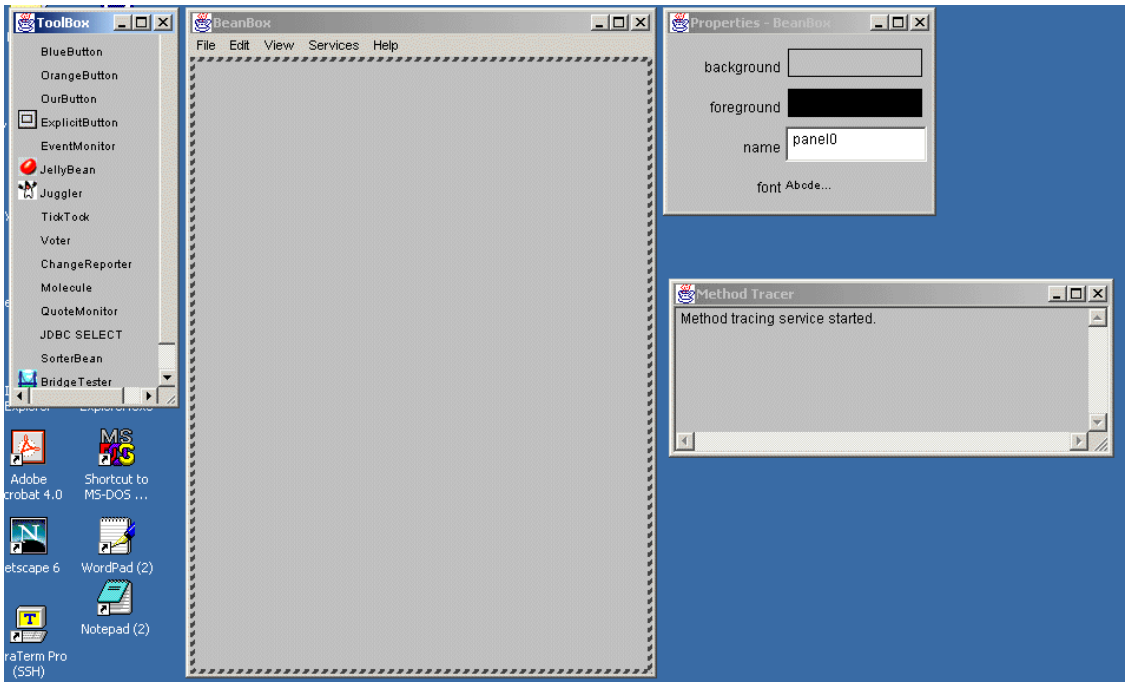

Figure 11.2. The initial windows of DBK

The ToolBox window contains a list of available Java beans. These beans are components that can be used to build more complex beans, Java applications, or applets. Visual software development tools, such as the BeanBox, allow beans to be visually organized by placing them at the location where you want them to be displayed. For example, by clicking the Juggler bean in the ToolBox window and then clicking in the BeanBox; the Juggler bean will be placed in the BeanBox, as shown in Figure 11.3.

When the Juggler bean is placed in the BeanBox, other windows will display corresponding information about the Juggler bean. For example, the PropertySheet is updated to display the properties of the Juggler bean. You can customize the Juggler bean by changing its properties. For example, changing the animationRate property to 500, the Juggler will juggle slower than before.

Now let us add a Start and a Stop buttons to the BeanBox to control the animation. First, we select an OurButton bean in the ToolBox and then place it in the BeanBox. A button labeled Press is displayed. Use the button's property sheet to change its label to Start. Use the same method to create a second button labeled Stop.

Second, we connect the Start button's actionPerformed() event handler to the startJuggling() method of the Juggler bean and the Stop button's actionPerformed() event handler to the stopJuggling() method of the Juggler bean. To do this, we click the Start button and then select Edit | Events | Action | ActionPerformed from the BeanBox menu bar. A red line is now shown emanating from the Start button. This line represents a logical connection from the Start button's actionPerformed() event handler. Click the Juggler bean to close the connection. When you do, the EventTargetDialog box, shown in Figure 11.4, is displayed. This dialog box lists the interface methods of the Juggler bean. Select startJuggling. By doing so, you connect the clicking of the Start button to the startJuggling() method via the actionPerformed() event handler of the Start button bean.

The EventTargetDialog box notifies you that it is compiling an adapter class. The BeanBox creates a special class, referred to as an adapter class, to connect the clicking of the button with the startJuggling() method of the Juggler. It must compile this class and add it to the running BeanBox to support this connection. You can use the same method to connect the Stop button to the stopJuggling() method of the Juggler bean.

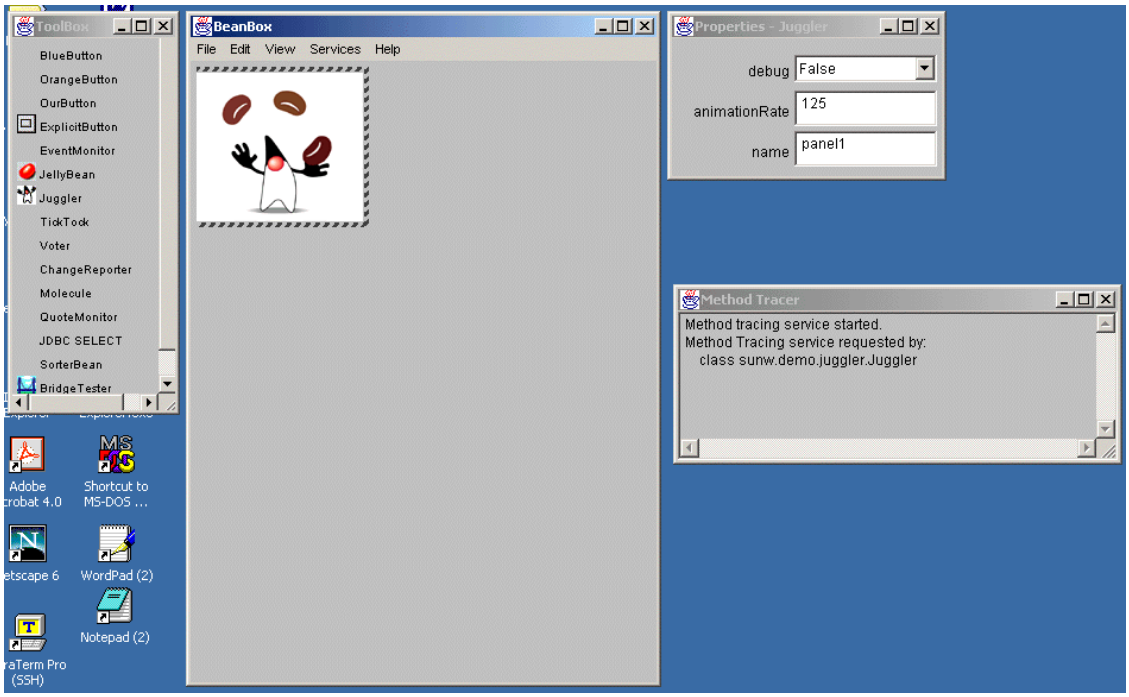

Figure 11.3. Adding the Juggler bean into the BeanBox

Figure 11.4 shows the process and result of the example event handling. Now you can have some fun with the Juggler. Click the Stop button to stop the juggling and click the Start button to get it going again.

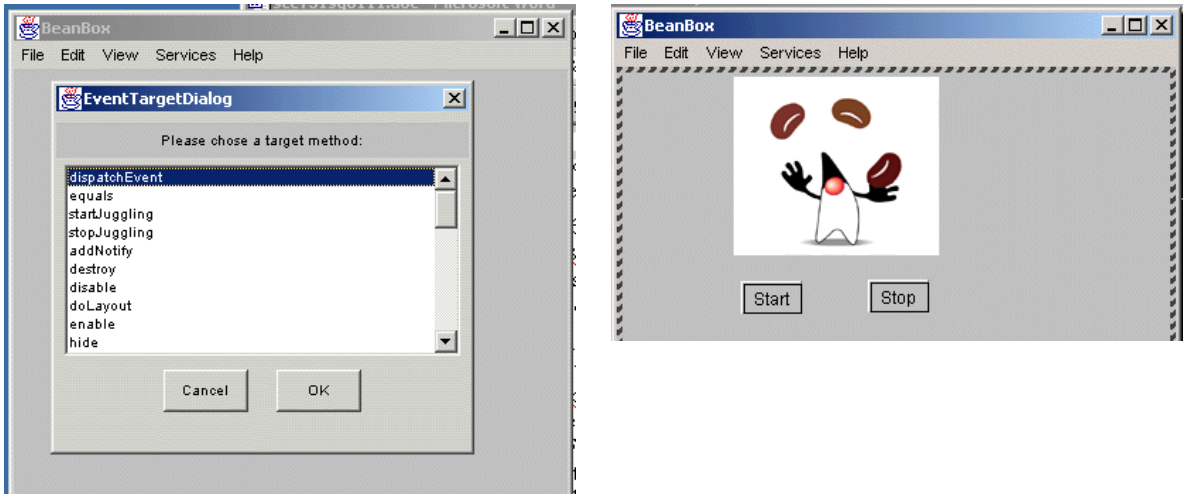

Figure 11.4. Event handling

Beans, being primarily GUI components, generate and respond to events. Visual development tools provide the capability to link events generated by one bean with event-

handling methods implemented by other beans. For example, a button component may generate an event as the result of the user clicking on that button. A visual development tool would enable you to connect the handling of this event to the interface methods of other beans. The bean generating the event is referred to as the *event source*. The bean listening for (and handling) the event is referred to as the *event listener*.

11.5.2. Beans by JavaSoft and IBM

Since the debut of JavaBeans, thousands of beans have been developed. Hundreds of these beans are available as off-the-shelf Java components. the JavaBeans Directory located at http://www.javasoft.com/beans/directory/ have links to many of these beans.

One of the most powerful bean sets on the market is the HotJava HTML Component from JavaSoft. This product consists of several beans that can be used to add Web-browsing support to window applications. It parses and renders HTML files that are loaded from the Web and includes the following features:

- HTML 3.2 support
- HTTP 1.1 compatibility
- Frames and tables support
- The ability to use cookies
- Multimedia support
- JAR file support
- Implementation of the FTP, Gopher, SMTP, and SOCKS protocols

A trial version of the HotJava HTML Component can be found from the link: http://java.sun.com/products/hotjava/bean/index.html.

IBM is one of the most committed developers of beans. It has created dozens of handy beans and makes them available through its fine Java software development tools. In particular, the WebRunner Toolkit includes a number of beans that are great for building applets and window applications. Some of the beans available through the WebRunner Toolkit include the following:

- A collection of Network Beans that support the FTP, NNTP, SMTP, and POP3 protocols
- User interface beans including a MultiColumnListbox bean, a ProgressBar bean, and a Charting bean
- A DatePicker bean that provides full calendar support
- A MaskedTextField bean for controlling use input
- A set of Gauge beans that were designed by IBM's Human-Computer Interaction Strategy and Design Lab

The link of http://www.software.ibm.com/ad/webrunner/WRBeans.html has the rial versions of these beans.

11.5.3. Assembly of Beans

The goal of bean-based software development is to use beans to quickly and easily assemble applets and applications. To accomplish this, you need a suitable collection of beans and an approach to integrating them into your programs. The InfoBus, developed

by Lotus Development Corporation and JavaSoft, provides a mechanism for bean integration. InfoBus supports a standard interface for communication between beans and allows information to be exchanged between beans in a structured way.

Normally, all beans that are loaded from the same classloader are visible to each other. Beans can find each other by searching the container-component hierarchy or their bean context. They can then use reflection and design patterns to determine which services are provided by other beans. However, this approach is often cumbersome and prone to error. The software engineers at Lotus Development Corporation and JavaSoft recognized that a standard approach to data exchange between beans was needed and collaborated to simplify inter-bean communication. The InfoBus is the result of this effort.

The InfoBus is analogous to a PC system bus. Data producers and consumers connect to an InfoBus in the same way that PC cards connect to a PC's system bus. Data producers use the bus to send data items to data consumers. The InfoBus is asynchronous and symmetric. This means that the producer and consumer do not have to synchronize to exchange data, and any member of the bus can send data to any other member of the bus.

The InfoBus operates as follows:

- Beans, components, and other objects join the InfoBus by implementing the InfoBusMember interface, obtaining an InfoBus instance, and using an appropriate method to join the instance.
- Data producers implement the InfoBusDataProducer interface, and data consumers implement the InfoBusDataConsumer interface. These interfaces define methods for handling events required for data exchange.
- Data producers signal that named data items are available on an InfoBus object by invoking the object's fireItemAvailable() method.
- Data consumers get named data items from an InfoBus object by invoking the requestDataItem() method of the InfoBusItemAvailableEvent event received via the InfoBusDataConsumer interface.

This list summarizes the typical usage of the InfoBus. However, the InfoBus is flexible and provides additional usage options, which you'll learn about in the next section. The advantage of InfoBus is that it eliminates the need for inference and discovery on the part of beans. Instead, it provides a standard, structured mechanism for named data items to be exchanged.

12. Network Security

12.1. Study Points

- Understand the basic concepts of a secure network.
- Understand the principles of a private key encryption and a public key encryption.
- Understand the concepts of digital signatures, packet filters, and firewalls.
- Understand the Java security model.
- Be familiar with the secure sockets.

References: [JNP] Chapters 3, 12. [FAR] Chapter 5. [Java2U] Chapter 8.

12.2. Secure Networks

12.2.1. What is a Secure Network?

A work on any types of networking is not complete without a discussion on network security. Networks cannot be simply classified as secure or not secure since the term of "secure" is not absolute: each group of users may define the level of security differently. For example, some organizations may regard the stored data as valuable and require that only authenticated users gaining access to these data. Some organizations allow outside users to browse their data, but prevent the data to be altered by outside users. Some organizations may regard the communication as the most important issue in network security and require that the messages be kept private and that the senders and recipients be authenticated. Yet many organizations need some combinations of the above requirements. Therefore, the first step for an organization to building a secure network is to define its security policy. The security policy specifies clearly and unambiguously the items that are to be protected.

A number of issues need to be considered in defining a security policy. They include the assessment of the values of information within an organization and the assessment of the costs and benefits of various security policies. Generally speaking, the following three aspects of security can be considered:

- Data integrity: it refers to the correctness of data and the protection from changes.
- Data availability: it refers to the protection against disruption of services.
- Data confidentiality and privacy: they refer to the protection against snooping or wiretapping.

12.2.2. Integrity Mechanisms and Access Control

The techniques used to ensure the integrity of data against accidental damage are the checksums and cyclic redundancy checks (CRC). To use such techniques, a sender compute a small, integer value as a function of the data in a packet. The receiver re-

computes the function from the data that arrives, and compares the result to the value that the sender computed.

However, the checksums or the CRC cannot absolutely guarantee data integrity. For example, a planned attacker can alter the data and then can create a valid checksum for the altered data.

The password mechanism is used in most computer system to control access to resources. This method works well in a conventional computer system but may be vulnerable in a networked environment. If a user at one location sends a password across a network to a computer at another location, anyone who wiretaps the network can obtain a copy f the password. Wiretapping is easy when packets travel across a LAN because many LAN technologies permit an attached station to capture a copy of all traffic. In such a situation, additional steps must be taken to prevent passwords from being reused.

12.3. Data Encryption

12.3.1. Encryption Principles

Encryption is a method that transforms the information in such a way that it cannot be understood by anyone except the intended recipient who possesses a secret method that makes it possible to decrypt the message. Figure 12.1 depicts the encryption process.

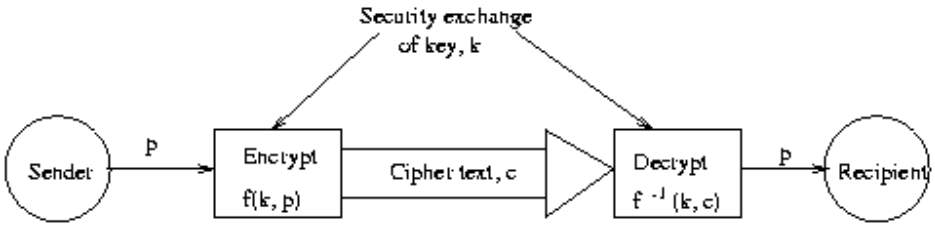

Figure 12.1. Single (private) key encryption

The two most popular private key techniques are DES (Data Encryption Standard) and IDEA (International Data Encryption Algorithm). The problem with the above scheme is the secured exchange of the key *k*: how can we be sure the key is known by both parties at the first place? A key distribution server sometimes is used to supply secret keys to clients. Figure 12.2 illustrates an example of key distribution server. Here user *A* needs to communicate with user *B*. A key is needed for both parties.

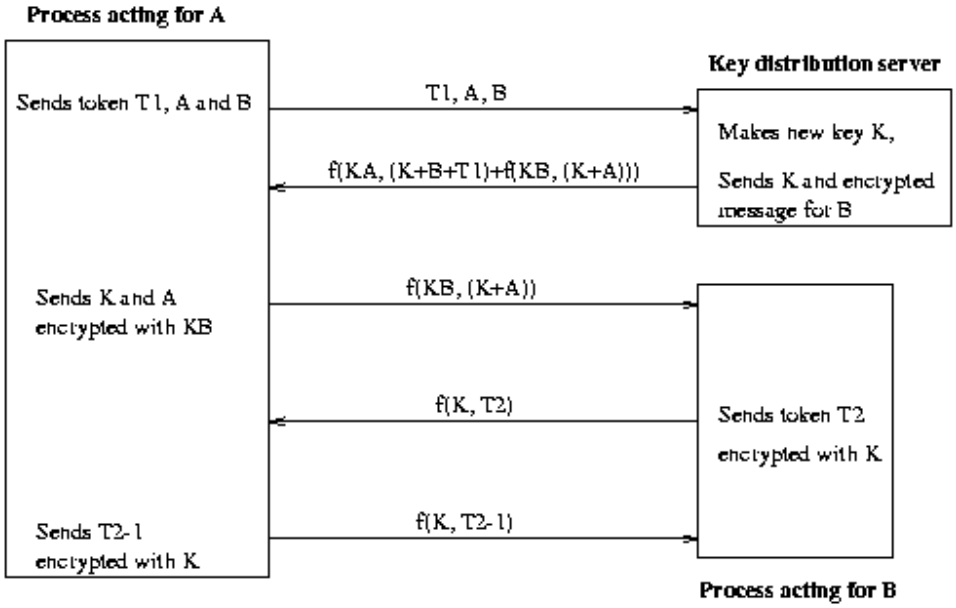

Figure 12.2. Key distribution server

A well known encryption method is the *public key encryption*. It uses two different keys (encryption key K_e and decryption key K_d), K_e is known to the sender and K_d the recipient. K_e can be made known publicly for use by anyone who wants to communicate, while K_d is kept secret. The RSA (after its inventors Rivest, Shamir, and Adleman) technique is one of the most popular public key encryption techniques and is based on the difficulty of factoring large numbers.

Here is an example: *A* (the Receiver) requires some secret information from *B* (the Sender). *A* generates a pair of keys K_e and K_d. K_d is kept secret and K_e is sent to *B*. *B* uses `c = E(Ke, p)` to encrypt the message and sends *c* to *A*. *A* then decrypts the message *c* using `p = D(Kd, c).` Figure 12.3 depicts this process.

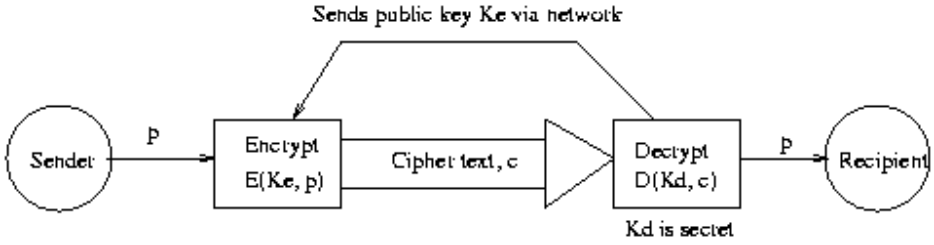

Figure 12.3. Public key encryption

12.3.2. Decryption

Many institutions and individuals read data that is not intend for them. They include:

- Government agencies. Traditionally governments around the world have reserved the rights to tape into any communication they think may be against the national interests.

- Spies who tap into communication for government and industry information.
- Individuals who like to read other people's messages.
- Individuals who hacks into systems and read sensitive information.
- Criminals who intercept information in order to use it for crime, such as intercepting PIN numbers on back accounts.

For example, the US government has proposed to beat encryption by trying to learn everyone's encryption key with the Clipper chip. The US government keeps a record of all the series numbers and encryption keys for each Clipper chip manufactured.

No matter how difficult an encryption is, every code is crackable and the measure of the security of a code is the amount of time it takes persons not addressed in the code to break the code. Normally to break a code, a computer tries all the possible keys until it finds the match. Thus a 1-bit code only has two keys. A 2-bit code would have 4 keys, and so on. For a 64-bit code it has 18,400,000,000,000,000,000 different keys. If one key is tested every 10μs, then it would take $1.84*10^{14}$ seconds (or 5,834,602 years).

However, as the improvement of computer power and techniques in parallel processing, the time used to crack a code may decrease dramatically. For example, if we think 1 million years would be safe for a code and we assume an increase of computer power of a factor of 2 every year, then it would take 500,000 years the next year. The same code would then be cracked in 1 year after 20 years. If we use parallel processing techniques then the code would be cracked much sooner.

12.4. Security Mechanisms on the Internet

12.4.1. Digital Signatures

Digital signatures are widely used on the Internet to authenticate the sender of a message. To sign a message, the sender encrypts the message using a key known only to the sender. The recipient uses the inverse function to decrypt the message. The receiver knows who sent the message because only the sender has the key needed to encrypt the message. A public key technique can be used in such a situation.

To sign a message, a sender encrypts the message using the private key. To verify the signature, the recipient looks up the user's public key and uses it to decrypt the message. Because only the user knows the private key, only the user can encrypt a message that can be decoded with the public key.

Interestingly, two levels of encryption can be used to guarantee that a message is both authentic and private. First, the message is signed by using the sender's private key to encrypt it. Second, the encrypted message is encrypted again using the recipient's public key. Here is the expression:

```
X = encrypt(public-rec, encrypt(private-sen, M))
```

Where *M* denotes a message to be sent, *X* denotes the string results from the two-level encryption, *private-sen* denotes the sender's private key, and *public-rec* denotes the recipient's public key.

At the recipient's side, the decryption process is the reverse of the encryption process. First, the recipient uses the private key to decrypt the message, resulting a digitally signed message. Then, the recipient uses the public key of the sender to decrypt the message again. The process can be expressed as follows:

```
M = decrypt(public-sen, decrypt(private-rec, X))
```

Where X is the encrypted message received by the recipient, M is the original message, private-rec denotes the recipient's private key, and public-sen denotes the sender's public key.

If a meaningful message results from the double decryption, it must be true that the message was confidential and authentic.

12.4.2. Packet Filtering

To prevent each computer on a network from accessing arbitrary computers or services, many sites use a technique known as *packet filtering*. A packet filter is a program that operates in a router. The filter consists of software that can prevent packets from passing through the router on a path from one network to another. A manager must configure the packet filter to specify which packets are permitted and which should be blocked. Figure 12.4 illustrates such a filter.

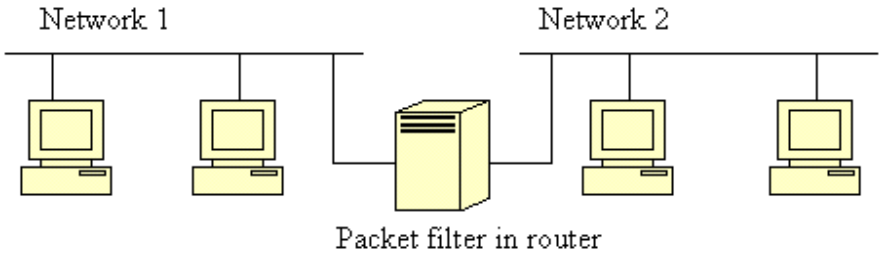

Figure 12.4. Packet filter in a router

A packet filter is configured to examine the packet header of each packet in order to decide which packets are allowed to pass through from one network to another. For example, the source and destination fields of a packet are examined to determine the if the packet is to be blocked or not. The filter can also examine the protocol of each packet or high-level services to block the access of a particular protocol or service For example, a packet filter can be configured to block all WWW access but to allow for email packets.

12.4.3. Internet Firewall

A packet filter can be used as a firewall for an organization to protect its computers from unwanted Internet traffic, as illustrated in Figure 12.5.

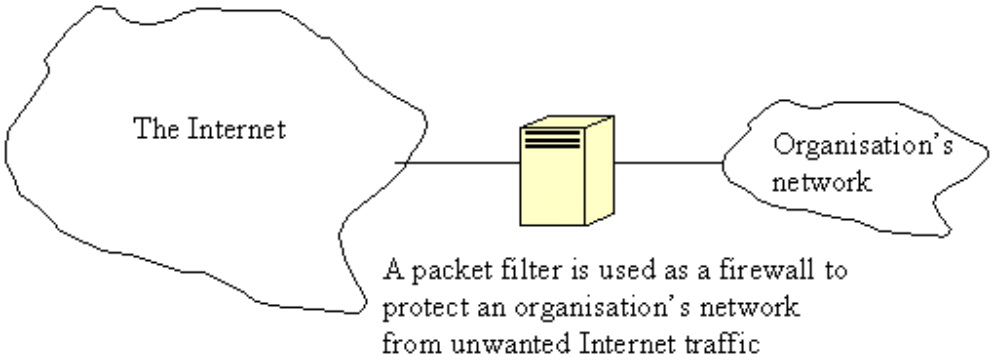

Figure 12.5. Internet firewall

Like a conventional firewall, an Internet firewall is designed to keep problems in the Internet from spreading into an organization's computer network. Without a firewall, an organization has to make all its computers secure to prevent unwanted Internet traffic. With a firewall, however, the organization can save money by only install the firewall and configure it to meet the requirements.

12.5. Java Secure Model and the Security API

12.5.1. The JDK 1.2 Security Architecture and Security Policy Specification

JDK 1.2 introduces a security architecture for implementing the security principle of *least privilege*. This principle states that an application should be given only those privileges that it needs to carry out its function and no more. According to least privilege, trusted applets and applications should be limited in the privileges they are allowed. This architecture is based on the capability to specify a security policy that determines what accesses an applet or application is allowed, based on its source and on the identities of those who have signed the applet on application code.

The security policy feature of JDK 1.2 allows you to specify the following types of policies easily and without programming:

- Grant all applets from http://www.trusted.com/ permission to read files in the C:\tmp directory.
- Grant all applets (from any host) permission to listen on TCP ports greater than 1023.
- Grant all applets signed by Mary and Ted (hypothetical Java programmers) that are from http://www.trusted.com permission to read and write to files in the C:\tmp directory.
- Grant all applications loaded from the C:\trusted directory permission to set security properties

Specifying a custom security policy is easy to do. All you have to do is edit the appropriate policy configuration file. JDK 1.2 provides you with a number of ways to do this:

- You can create or edit the default system policy file located at <*java.home*>\lib\ security\java.policy, where <*java.home*> identifies the location of your JDK 1.2 installation. It is specified by the value of the java.home system property. By default, java.home is C:\jdk1.2. If you edit java.policy, the new policy will apply to all users of your JDK 1.2 installation.
- You can set the value of the policy.java system property to the name of an alternative security policy file.
- You can create or edit the user policy file located at <*user.home*>\.java.policy, where <*user.home*> identifies the current user's home directory. It is specified by the value of the user.home system property.
- You can set the value of the java.security.policy property to a different user security policy file using the -D command-line option. For example, suppose that you want to run the Test class using the test.policy user security policy file. You could use the -D option as follows:

  ```
  java -Djava.security.policy=="test.policy" Test
  ```

- You can also use the -Djava.security.manager option to ensure that the policy is installed. The double equal sign specifies that the policy should be the only policy that is in effect. A single equal sign specifies that the policy is added to the current policy that is in effect.
- You can change the class used to implement the security policy from java. security.PolicyFile to another class by editing the java.security file located at <*java.home*>\lib\security\java.security. Change the line policy.provider=java.security.PolicyFile to policy.provider=OtherClass, where OtherClass is the fully qualified name of the class to be used.

When the Java byte code interpreter is run, it loads in the system policy followed by the user policy. If neither of these policies is available, the original sandbox policy is used.

You can also use the -Djava.security.manager option to ensure that the policy is installed. The double equal sign specifies that the policy should be the only policy that is in effect. A single equal sign specifies that the policy is added to the current policy that is in effect.

12.5.2. The ClassLoader and the SecurityManager

Java provides two levels of security: low-level intrinsic security and resource-level security. Java’s intrinsic security relates to the integrity of the bytecodes that come across the network, and consists of a bytecode verifier and a *ClassLoader*. The verifier attempts to make sure that incoming bytecodes do not perform illegal type conversions, memory accesses, and other similar forbidden activities. The *ClassLoader* partitions the name spaces of classes loaded from across the network and prevent collisions related name resolution problems. The *ClassLoader* also ensures that local classes are loaded first to prevent spoofing of system classes. On top of this, Java provides resource access restriction through a *SecurityManager* class.

Students are required to read through Chapter 2 of the text book to understand the Java Security Model, paying particular attention to the *SecurityManager* and the networked applets.

You can use the following methods from java.lang.SecurityManager to check how much network access you'll have:

```
public void checkConnect(String host, int port)
 public void checkConnect(String host, int port, Object context)
 public void checkListen(int port)
 public void checkAccept(String hostname, int port)
 public void checkMulticast(InetAddress maddr)
 public void checkMulticast(InetAddress maddr, byte ttl)
```

Each of these methods throws a SecurityException (which is a runtime exception so it doesn't need to be declared) if the requested operation is not permitted. For example, to check whether you're allow to open a socket to port 80 of www.deakin.edu.au you would write:

```
try {
  SecurityManager sm = SecurityManager.getSecurityManager();
  if (sm != null) sm.checkConnect("www.deakin.edu.au", 80);
  // open the socket...
}
catch (SecurityException e) {
  System.err.println("Sorry. I'm not allowed to connect to that host.");
}
```

checkConnect() tests whether a socket connection is allowed. checkListen() tests whether binding to a particular port is allowed. checkAccept() tests whether you can accept a connection from a particular remote host and port. checkMulticast() tests whether multicasting is allowed.

12.5.3. Using Java Security API: An Example

The client reads a file, generates a pair of key, signs the content of a file and send an object through the network to a server. This object contains the public key generated by the client, the signature, and the content of the file. The server then receives this object, and verifies that the signature is correct.

This example may be a good starting point to understand for example how a SSL browser works, and how the Java Security API may be used in client-server applications.

There are 4 files: Client.java, Main.java, Server.java, signedData.java.

The client class creates a pair of key (512 bytes) and a signature object. This Signature class is used to provide the functionality of a digital signature algorithm, such as RSA with MD5 or DSA. Here we use the DSA algorithm.

Then, it reads the file supplied on the command line and computes a digital signature. A signedData object is created with the content of the file, the client's public key and the

digital signature. It's then sent to the server using serialization through an ObjectOutputStream.

```
/* Client.java */
import java.net.*;
import java.io.*;
import java.security.*;

class Client {
  public static void main(String [] argv)
  {
    try {
      Socket s=new Socket("localhost",4096);
      System.out.println("Generating keys...");
      ObjectOutputStream oos=new ObjectOutputStream(s.getOutputStream());
      //Generates keys
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
      keyGen.initialize(512);
      KeyPair pair = keyGen.generateKeyPair();
      System.out.println("Keys generated.");

      //Generates signature object
      Signature dsa = Signature.getInstance("SHA/DSA");
      PublicKey pub=pair.getPublic();
      PrivateKey priv=pair.getPrivate();
      dsa.initSign(priv);
      System.out.println("Signature object generated.");

      //Read the input file and computes a signature
      FileInputStream fis = new FileInputStream(argv[0]);
      byte[] b=new byte[fis.available()];
      fis.read(b);
      dsa.update(b);

      //Build the signedData and send it
      signedData Data=new signedData(b,dsa.sign(),pub);
      Data.printKey();
      oos.writeObject(Data);

      //Close streams
      fis.close();
      oos.close();
      s.close();
    } catch (Exception e) {System.out.println(e);};
  }
}
```

The main class simply starts the Server that listens on a port (4096).

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

class Main {
    public static void main(String [] argv)
    {
             try {
                 Server s=new Server();
                 s.startServer();
             } catch (IOException e) {};
    }
```

```
}
```

The Server class waits for an incoming connection using a ServerSocket. When a connection is made, it reads a signedData object from the stream, and computes a digital signature using the client's public key to verify that the file transmitted from the client is valid.

```
/* Server.java */
import java.net.*;
import java.io.*;
import java.security.*;

class Server implements Runnable
{
  ServerSocket server=null;
  Socket data=null;
  Thread t=null;
  KeyPairGenerator keyGen;
  KeyPair pair;

  public synchronized void startServer() throws IOException {
    if (t==null)
    {
      server=new ServerSocket(4096);
      t=new Thread(this);
      t.start();
    }
  }

  public synchronized void stopServer() throws IOException {
    if (t!=null)
    {
      t=null;
      server.close();
    }
  }

  public void run()
  {
    Thread thisThread=Thread.currentThread();
    System.out.println("Ready to accept connections.");
    while (t==thisThread) {
       try {
         Socket data=server.accept();
         process(data);
       } catch( Exception e) {}
    }
  }

  private void process(Socket data) throws IOException
  {

    System.out.println("Accepting connexion...");
    try {
      ObjectInputStream ois=new ObjectInputStream(data.getInputStream());

      //Read the object from the network
      signedData Data=(signedData)ois.readObject();

      //Generate the signature object
      Signature dsa = Signature.getInstance("SHA/DSA");
      dsa.initVerify(Data.pub);
```

```
      dsa.update(Data.b);

      //Verify the signature
      Data.printKey();
      boolean verifies = dsa.verify(Data.sig);
      System.out.println(verifies?"Signature is correct!":"Signature not
valid!");

      ois.close();
      data.close();

    } catch (Exception e) {System.out.println(e);}
  }
}
```

The following signedData class is sent through the network from the client to the server. It contains an array of byte, a digital signature, and a public key.

```
/* signedData.java */
import java.io.*;
import java.security.*;

public class signedData implements Serializable
{
  byte[] b;
  byte[] sig;
  PublicKey pub;

  signedData(byte b[],byte [] sig ,PublicKey pub) {
    this.b=b;
    this.sig=sig;
    this.pub=pub;
  }

  void printKey() {
     System.out.println("Key dump:");
     System.out.println(pub);
  }
}
```

Here is the execution result of the server:

```
>Java Main test.txt
Ready to accept connections.
Accepting connexion...
Key dump:
Sun DSA Public Key
    Parameters:DSA
        p:      fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
                01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
        q:      962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
        g:      678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
                35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

  y:
    cb40d181 09ab15e5 60ae95f3 6b49f109 c1be351b d194ca87 0ce9d153 d553235f
    1084fd3b 81658bf0 2dd737a0 ed6029db 378a9e8c a875c62e 1adedc96 fdd70a28

Signature is correct!
```

Here is the execution result of the client:

```
>java Client test.txt
Generating keys...
Keys generated.
```

```
Signature object generated.
Key dump:
Sun DSA Public Key
    Parameters:
        p:      fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
                01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
        q:      962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
        g:      678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
                35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

  y:
    cb40d181 09ab15e5 60ae95f3 6b49f109 c1be351b d194ca87 0ce9d153 d553235f
    1084fd3b 81658bf0 2dd737a0 ed6029db 378a9e8c a875c62e 1adedc96 fdd70a28
```

12.6. Secure Sockets

One of the consumer fears of buying goods over the Internet is that some hacker will steal the credit card details when the information is transmitted. In reality, the risk of a hacker grabbing credit card information during message transmission is much less than a shop clerk stealing the credit card details from a receipt. All the cases in e-business related to credit card thefts so far have been related to the poorly secured databases and files after the credit card information has been successfully transmitted across the Internet. Nonetheless, to make Internet connection more fundamentally secure, sockets can be encrypted. This allows transactions to be confidential, authenticated, and accurate.

However, encryption is a complex subject. Finding good encryption and authentication algorithms is a challenging work. Writing such software also needs highly skilled experts. Also, such software is still subject to the arms control laws in the USA and other countries. And consequently cannot be exported or imported freely. Therefore such capabilities are not built into the standard *java.net* classes in JDK. Instead, they are provided as a standard extension to the JDK called the Java Secure Socket Extension (JSSE). This is an add-on for the JDK that secures network communications using the Secure Sockets Layer (SSL) Version 3 and Transport Layer Security (TLS) protocols and their associated algorithms. SSL is a security protocol to allow web browsers to talk to web servers using various levels of confidentiality and authentication.

JSSE is the standard extension to Java 1.2. It allows you to create sockets and server sockets that transparently handle the negotiations and encryption necessary for secure communication. The JSSE is divided into four packages:

- *javax.net.ssl:* the abstract classes that define Java's API for secure network communication.
- *javax.net:* the abstract socket factory class used instead of constructors to create secure sockets.
- *javax.security.cert:* a minimal set of classes for handling public key certification that's needed for SSL in Java 1.1 (in Java 1.2 and later, the *java.security.cert* package should be used).
- *com.sun.net.ssl:* the concret classes that implement the encryption algorithms and protocols in Sun's reference implementation of the JSSE.

None of these classes are included in the standard distribution of JDK. Before you can use any of these classes, you have to download the JSSE from http://java.sun.com/products/jsse/.