Jan Graba

An Introduction to Network Programming with Java

Java 7 Compatible

Third Edition

Jan Graba, *An Introduction to Network Programming with Java*, 3rd ed. 2013, Java 7 Compatible, DOI: 10.1007/978-1-4471-5254
© Springer-Verlag London 2013

Jan Graba

An Introduction to Network Programming with Java

Java 7 Compatible

Jan Graba
Department of Computing, Sheffield Hallam University, Sheffield, South Yorkshire, UK

ISBN 978-1-4471-5253-8 e-ISBN 978-1-4471-5254-5

Springer London Heidelberg New York Dordrecht

Library of Congress Control Number: 2013946037

© Springer-Verlag London 2013

Additional material to this book can be downloaded from http://extras.springer.com. 1st edition: © Addison-Wesley 2003 1st edition: © Addison-Wesley 2003

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission o information storage and retrieval, electronic adaptation, computer software, or by similar or dissimil methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of th work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyrigh Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility fo any errors or omissions that may be made. The publisher makes no warranty, express or implied, wit respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface to Third Edition

It is now 7 years since I wrote the second edition of *An Introduction to Network Programming with Java* and so, when approached to produce a third edition, I felt that it was an appropriate time to agre to do so (possibly rather later than it should have been). One of the very first things that I did after being approached was examined the literature to find out what texts had been produced in the Java/network programming area in the interim period, and so what the current state of the competitio was. (Though I had had a strong interest in this area for a considerable number of years, I had also been involved in other areas of software development, of course, and hadn't had cause to examine the literature in this area for some time.) To my great surprise, I found that virtually nothing of any consequence had been produced in this area during those years! Of course, this was a very welcome surprise and provided further impetus to go ahead with the project.

The changes in this third edition are not as profound as those in the second edition, largely becaus Java 5 brought in major language changes (both network and non-network) that needed to be reflecte in the second edition, whereas neither Java 6 nor Java 7 has had such an impact, particularly in the area of network programming. One major change that did occur during this time, and is worth mentioning, was Sun's takeover by Oracle in April of 2009, but this has had no significant effect on the way in which Java has been developed.

Since the changes that have been necessary since the second edition are somewhat more small-scale than those that were desirable after the first edition, I think that it would be useful to give a chapter-by-chapter indication of what has been changed, what has been introduced that wasn't there before and, in some cases, what has been removed completely (the last of these hopefully resulting in a more 'streamlined' product). Consequently, the great bulk of the remainder of this preface will comprise a chapter-by-chapter breakdown of those changes.

Chapter 1

- Updating of browsers and browser versions used (along with associated updating of screenshots)
- Updating of the comparison between TCP and UDP.

Chapter 2

- Removal of Sect. 2.4 ('Downloading Web Pages'), felt by me to be of little use to most people.
- Some very minor changes to lines of code.

Chapter 3

- Extra text devoted to the differing strategies for determining which thread from a group of competing threads is to be given use of the processor at any given time.

Chapter 4

- Addition of *ArrayList* s and associated relegation of *Vector* s (with consequent modification of example program).
- Comparison of *Vector* s and *ArrayList* s.

Chapter 5

- Removal of step 2 in 5.3 (compiling with *rmic*), which has actually been unnecessary since Java 5.
- *Vector* references replaced with *ArrayList* ones in bank example of 5.4.

Chapter 6

- Some very minor URL changes.

Chapter 7

- Statement of redundancy of the loading of the database driver (for all JDBC4-compatible drivers), with consequent removal from examples. (JDBC4 was part of Java 6, which was introduced in December of 2006.)
- Addition of material on Apache Derby/Java DB, which came in with Java 6. This material introduced in a new Sect. 7.6 , with consequent re-numbering of the old 7.6 and all later sections in this chapter.
- Information about Jakarta's retirement on 21/12/11 (and consequent direct control of Jakarta's sub-projects by Apache).
- Changes to the steps required in the new Sect. 7.12 (previously 7.11) for using the *DataSource* interface and creating a DAO (this method having changed somewhat since 2006), with consequent changes to the code of the example.
- Modification of the steps required for downloading and extracting DBCP files.

Chapter 8

- Updating of the Servlet API installation instructions.
- Removal of references to Tomcat's ROOT folder (now no longer in existence).
- Introduction of servlet annotation lines (introduced in Java 6).

Chapter 9

- Replacement of some HTML code with HTML-5 compatible CSS.
- Some very minor changes to lines of code.

Chapter 10

- Removal of Sect. 10.1, due to the Bean Builder now being defunct and no replacement for this software having appeared.
- Removal of the requirement that beans implement the *Serializable* interface, since this is (now?) unnecessary, with associated removal of the clause *implements Serializable* from the examples.
- Introduction of CSS into examples, to make examples HTML-5 compatible.

(Old) Chapter 11

- **Removal** of this entire chapter (with consequent re-numbering of later chapters). This has been done partly because EJPs are no longer of such importance since the emergence of frameworks such as Hibernate and Spring and partly because I felt that the complexity of EJBs probably didn't warrant their inclusion in this text.

New 'Chapter 11' (Previously 'Chapter 12')

- Very minor changes of wording.

New 'Chapter 12' (Previously 'Chapter 13')

- Updating of browsers used.

In keeping with the society-wide move towards Internet storage, there is now no CD accompanying this text. Model solutions for end-of-chapter exercises are accessible by lecturers and other authorised individuals through access/application form via http://springer.com/978-1-4471-5253-8 . Also included at this URL is a Word document called *Java Environment Installation* that provides downloading and installation instructions for Java 7 and all associated software required to complete the end-of-chapter exercises. (The instructions will not refer to the latest update of Java 7, s please download whatever is the latest update.)

At a second URL (http://extras.springer.com) are the items listed below, which can be found by searching for the book's ISBN (978-1-4471-5253-8).

- Chapter examples
- Supplied code
- GIF files
- JPEG files
- Sound files
- Videos

All that remains now is for me to wish you luck and satisfaction in your programming endeavour Good luck!

Jan Grab

27 March 201

Contents

1 Basic Concepts, Protocols and Terminology

1.1 Clients, Servers and Peers

1.2 Ports and Sockets

1.3 The Internet and IP Addresses

1.4 Internet Services, URLs and DNS

1.5 TCP

1.6 UDP

2 Starting Network Programming in Java

2.1 The InetAddress Class

2.2 Using Sockets

2.2.1 TCP Sockets

2.2.2 Datagram (UDP) Sockets

2.3 Network Programming with GUIs

3 Multithreading and Multiplexing

3.1 Thread Basics

3.2 Using Threads in Java

3.2.1 Extending the Thread Class

3.2.2 Explicitly Implementing the Runnable Interface

3.3 Multithreaded Servers

3.4 Locks and Deadlock

3.5 Synchronising Threads

3.6 Non-blocking Servers

3.6.1 Overview

3.6.2 Implementation

3.6.3 Further Details

---

4 File Handling

4.1 Serial Access Files

4.2 File Methods

4.3 Redirection

4.4 Command Line Parameters

4.5 Random Access Files

4.6 Serialisation [U.S. Spelling Serialization]

4.7 File I/O with GUIs

4.8 ArrayLists

4.9 ArrayLists and Serialisation

4.10 Vectors Versus ArrayLists

5 Remote Method Invocation (RMI)

5.1 The Basic RMI Process

5.2 Implementation Details

5.3 Compilation and Execution

5.4 Using RMI Meaningfully

5.5 RMI Security

6 CORBA

6.1 Background and Basics

6.2 The Structure of a Java IDL Specification

6.3 The Java IDL Process

6.4 Using Factory Objects

6.5 Object Persistence

6.6 RMI-IIOP

7 Java Database Connectivity (JDBC)

7.1 The Vendor Variation Problem

7.2 SQL and Versions of JDBC

7.3 Creating an ODBC Data Source

7.4 Simple Database Access

7.5 Modifying the Database Contents

7.6 Java DB/Apache Derby

7.7 Transactions

7.8 Meta Data

7.9 Using a GUI to Access a Database

7.10 Scrollable ResultSets

7.11 Modifying Databases via Java Methods

7.12 Using the DataSource Interface

7.12.1 Overview and Support Software

7.12.2 Defining a JNDI Resource Reference

7.12.3 Mapping the Resource Reference onto a Real Resource

7.12.4 Obtaining the Data Source Connection

7.12.5 Data Access Objects

8 Servlets

8.1 Servlet Basics

8.2 Setting Up the Servlet API

8.3 Creating a Web Application

8.4 The Servlet URL and the Invoking Web Page

8.5 Servlet Structure

8.6 Testing a Servlet

8.7 Passing Data

8.8 Sessions

8.9 Cookies

8.10 Accessing a Database via a Servlet

9 JavaServer Pages (JSPs)

9.1 The Rationale Behind JSPs

9.2 Compilation and Execution

9.3 JSP Tags

9.3.1 Directives

9.3.2 Declarations

9.3.3 Expressions

9.3.4 Scriptlets

9.3.5 Comments

9.3.6 Actions

9.4 Implicit JSP Objects

9.5 Collaborating with Servlets

9.6 JSPs in Action

9.7 Error Pages

9.8 Using JSPs to Access Remote Databases

10 JavaBeans

10.1 Creating a JavaBean

10.2 Exposing a Bean’s Properties

10.3 Making Beans Respond to Events

10.4 Using JavaBeans Within an Application

10.5 Bound Properties

10.6 Using JavaBeans in JSPs

10.6.1 The Basic Procedure

10.6.2 Calling a Bean’s Methods Directly

10.6.3 Using HTML Tags to Manipulate a Bean’s Properties

11 Multimedia

11.1 Transferring and Displaying Images Easily

11.2 Transferring Media Files

11.3 Playing Sound Files

11.4 The Java Media Framework

12 Applets

12.1 Applets and JApplets

12.2 Applet Basics and the Development Process

12.3 The Internal Operation of Applets

12.4 Using Images in Applets

12.4.1 Using Class Image

12.4.2 Using Class ImageIcon

12.5 Scaling Images

12.6 Using Sound in Applets

Appendix: Structured Query Language (SQL)

A.1 DDL Statements

A.1.1 Creating a Table

A.1.2 Deleting a Table

A.1.3 Adding Attributes

A.1.4 Removing Attributes

A.2 DML Statements

A.2.1 SELECT

A.2.2 INSERT

A.2.3 DELETE

A.2.4 UPDATE

Index

Jan Graba, *An Introduction to Network Programming with Java*, 3rd ed. 2013, Java 7 Compatible, DOI: 10.1007/978-1-4471-5254
5_1, © Springer-Verlag London 2013

1. Basic Concepts, Protocols and Terminology

Jan Graba[1]

(1) Department of Computing, Sheffield Hallam University, Sheffield, South Yorkshire, UK

Abstract
The fundamental purpose of this opening chapter is to introduce the underpinning network principles and associated terminology with which the reader will need to be familiar in order to make sense of the later chapters of this book. The material covered here is entirely generic (as far as any programming language is concerned) and it is not until the next chapter that we shall begin to consid how Java may be used in network programming. If the meaning of any term covered here is not clear when that term is later encountered in context, the reader should refer back to this chapter to refresh his/her memory.

Learning Objectives
After reading this chapter, you should:

- have a high level appreciation of the basic means by which messages are sent and received on modern networks;
- be familiar with the most important protocols used on networks;
- understand the addressing mechanism used on the Internet;
- understand the basic principles of client/server programming.

The fundamental purpose of this opening chapter is to introduce the underpinning network principles and associated terminology with which the reader will need to be familiar in order to make sense of the later chapters of this book. The material covered here is entirely generic (as far as any programming language is concerned) and it is not until the next chapter that we shall begin to consid how Java may be used in network programming. If the meaning of any term covered here is not clear when that term is later encountered in context, the reader should refer back to this chapter to refresh his/her memory.

It would be very easy to make this chapter considerably larger than it currently is, simply by including a great deal of dry, technical material that would be unlikely to be of any practical use to th intended readers of this book. However, this chapter is intentionally brief, the author having avoided the inclusion of material that is not of relevance to the use of Java for network programming. The reader who already has a sound grasp of network concepts may safely skip this chapter entirely.

1.1 Clients, Servers and Peers

The most common categories of network software nowadays are *clients* and *servers*. These two

categories have a symbiotic relationship and the term *client/server programming* has become very widely used in recent years. It is important to distinguish firstly between a server and the machine upon which the server is running (called the *host* machine), since I.T. workers often refer loosely to the host machine as ‘the server’. Though this common usage has no detrimental practical effects for the majority of I.T. tasks, those I.T. personnel who are unaware of the distinction and subsequently undertake network programming are likely to be caused a significant amount of conceptual confusion until this distinction is made known to them.

A server, as the name implies, provides a service of some kind. This service is provided for client that connect to the server’s host machine specifically for the purpose of accessing the service. Thus, is the clients that initiate a dialogue with the server. (These clients, of course, are also programs and are **not** human clients!) Common services provided by such servers include the ‘serving up’ of Web pages (by Web servers) and the downloading of files from servers’ host machines via the File Transf Protocol (FTP servers). For the former service, the corresponding client programs would be Web browsers (such as Firefox, Chrome or Internet Explorer). Though a client and its corresponding serve will normally run on different machines in a real-world application, it is perfectly possible for such programs to run on the *same* machine. Indeed, it is often very convenient (as will be seen in subsequent chapters) for server and client(s) to be run on the same machine, since this provides a ver convenient ‘sandbox’ within which such applications may be tested before being released (or, more likely, before final testing on separate machines). This avoids the need for multiple machines and multiple testing personnel.

In some applications, such as messaging services, it is possible for programs on users’ machines communicate directly with each other in what is called *peer-to-peer* (or *P2P*) mode. However, for many applications, this is either not possible or prohibitively costly in terms of the number of simultaneous connections required. For example, the World Wide Web simply does not allow clients to communicate directly with each other. However, some applications use a server as an intermediary in order to provide ‘simulated’peer-to-peer facilities. Alternatively, both ends of the dialogue may ac as both client and server. Peer-to-peer systems are beyond the intended scope of this text, though, an no further mention will be made of them.

1.2 Ports and Sockets

These entities lie at the heart of network communications. For anybody not already familiar with the use of these terms in a network programming context, the two words very probably conjure up image of hardware components. However, although they are closely associated with the hardware communication links between computers within a network, *ports* and *sockets* are not themselves hardware elements, but abstract concepts that allow the programmer to make use of those communication links.

A port is a *logical* connection to a computer (as opposed to a physical connection) and is identifie by a number in the range 1–65535. This number has no correspondence with the number of physical connections to the computer, of which there may be only one (even though the number of ports used on that machine may be much greater than this). Ports are implemented upon all computers attached to a network, but it is only those machines that have server programs running on them for which the network programmer will refer explicitly to port numbers. Each port may be dedicated to a particular server/service (though the number of available ports will normally greatly exceed the number that is actually used). Port numbers in the range 1–1023 are normally set aside for the use of specified standard services, often referred to as ‘well-known’ services. For example, port 80 is normally used b Web servers. Some of the more common well-known services are listed in Sect. 1.4. Application

programs wishing to use ports for non-standard services should avoid using port numbers 1–1023. (A range of 1024–65535 should be more than enough for even the most prolific of network programmers!).

For each port supplying a service, there is a server program waiting for any requests. All such programs run together in parallel on the host machine. When a client attempts to make connection with a particular server program, it supplies the port number of the associated service. The host machine examines the port number and passes the client’s transmission to the appropriate server program for processing.

In most applications, of course, there are likely to be multiple clients wanting the same service at the same time. A common example of this requirement is that of multiple browsers (quite possibly thousands of them) wanting Web pages from the same server. The server, of course, needs some way of distinguishing between clients and keeping their dialogues separate from each other. This is achieved via the use of *sockets*. As stated earlier, a socket is an abstract concept and **not** an element c computer hardware. It is used to indicate one of the two end-points of a communication link between two processes. When a client wishes to make connection to a server, it will create a socket at its end c the communication link. Upon receiving the client’s initial request (on a particular port number), the server will create a new socket at its end that will be dedicated to communication with that particular client. Just as one hardware link to a server may be associated with many ports, so too may one port b associated with many sockets. More will be said about sockets in Chap. 2.

1.3 The Internet and IP Addresses

An internet (lower-case ‘i’) is a collection of computer networks that allows any computer on any of the associated networks to communicate with any other computer located on any of the other associated networks (or on the same network, of course). The protocol used for such communication called the Internet Protocol (IP). *The* Internet (upper-case ‘I’) is the world’s largest IP-based network Each computer on the Internet has a unique IP address, the current version of which is still, for most people, IPv4 (Internet Protocol version 4), though this is likely to change at some point during the next few years. This represents machine addresses in what is called **quad notation**. This is made up c four eight-bit numbers (i.e., numbers in the decimal range 0–255), separated by dots. For example, 131.122.3.219 would be one such address. Due to a growing shortage of IPv4 addresses, IPv4 is due t be replaced with IPv6, the draft standard for which was published on the 10th of August, 1998. IPv6 uses 128-bit addresses, which provide massively more addresses. Many common Internet application already work with IPv6 and it is expected that IPv6 will gradually replace IPv4, with the two coexisting for a number of years during a transition period.

Recent years have witnessed an explosion in the growth and use of the Internet. As a result, there has arisen a need for a programming language with features designed specifically for network programming. Java provides these features and does so in a platform-independent manner, which is vital for a heterogeneous network such as the Internet. Java is sometimes referred to as ‘the language of the Internet’ and it is the use of Java in this context that has had a major influence on the popularisation of the language. For many programmers, the need to program for the Internet is one of the main reasons, if not *the* reason, for learning to program in Java.

1.4 Internet Services, URLs and DNS

Whatever the service provided by a server, there must be some established *protocol* governing the communication that takes place between server and client. Each end of the dialogue must know what

may/must be sent to the other, the format in which it should be sent, the sequence in which it must be sent ~~(if sequence matters) and, for ‘open-ended’ dialogues, how the dialogue is to be terminated~~. For the standard services, such protocols are made available in public documents, usually by either the Internet Engineering Task Force (IETF) or the World Wide Web Consortium (W3C). Some of the more common services and their associated ports are shown in Table 1.1. For a more esoteric or ‘bespoke’ service, the application writer must establish a protocol and convey it to the intended users of that service.

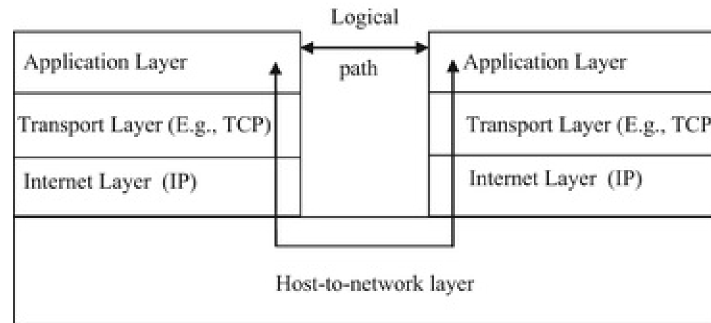

Fig. 1.1 The 4-layer network model

Table 1.1 Some well-known network services

Protocol name	Port number	Nature of service
Echo	7	The server simply echoes the data sent to it. This is useful for testing purposes
Daytime	13	Provides the ASCII representation of the current date and time on the server
FTP-data	20	Transferring files. (FTP uses two ports.)
FTP	21	Sending FTP commands like PUT and GET
Telnet	23	Remote login and command line interaction
SMTP	25	E-mail. (Simple Mail Transfer Protocol.)
HTTP	80	HyperText Transfer Protocol (the World Wide Web protocol)
NNTP	119	Usenet. (Network News Transfer Protocol.)

A URL (Uniform Resource Locator) is a unique identifier for any resource located on the Interne It has the following structure (in which BNF notation is used):

```
<protocol>://<hostname>[:<port>][/<pathname>][/<filename>[#<section>]]
```

For example:

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

For a well-known protocol, the port number may be omitted and the default port number will be assumed. Thus, since the example above specifies the HTTP protocol (the protocol of the Web) and does not specify on which port of the host machine the service is available, it will be assumed that th service is running on port 80 (the default port for Web servers). If the file name is omitted, then the server sends a default file from the directory specified in the path name. (This default file will commonly be called *index.html* or *default.html*.) The ‘section’ part of the URL (not often specified) indicates a named ‘anchor’ in an HTML document. For example, the HTML anchor in the tag

```
<A HREF="#summary">Summary of Report</A>
```

would be referred to as `summary` by the section component of the URL.

Since human beings are generally much better at remembering meaningful strings of characters than they are at remembering long strings of numbers, the Domain Name System was developed. A *domain name*, also known as a *host name*, is the user-friendly equivalent of an IP address. In the previous example of a URL, the domain name was www.oracle.com. The individual parts of a domai name don’t correspond to the individual parts of an IP address. In fact, domain names don’t always

have four parts (as IPv4 addresses must have).

~~Normally, human beings will use domain names in preference to IP addresses, but they can~~ just a well use the corresponding IP addresses (if they know what they are!). The *Domain Name System* provides a mapping between IP addresses and domain names and is held in a distributed database. Th IP address system and the DNS are governed by ICANN (the Internet Corporation for Assigned Name and Numbers), which is a non-profitmaking organisation. When a URL is submitted to a browser, the DNS automatically converts the domain name part into its numeric IP equivalent.

1.5 TCP

In common with all modern computer networks, the Internet is a **packet-switched** network, which means that messages between computers on the Internet are broken up into blocks of information called **packets**, with each packet being handled separately and possibly travelling by a completely different route from that of other such packets from the same message. IP is concerned with the **routing** of these packets through an internet. Introduced by the American military during the Cold War, it was designed from the outset to be robust. In the event of a military strike against one of the network routers, the rest of the network **had** to continue to function as normal, with messages that would have gone through the damaged router being re-routed. IP is responsible for this re-routing. It attaches the IP address of the intended recipient to each packet and then tries to determine the most efficient route available to get to the ultimate destination (taking damaged routers into account).

However, since packets could still arrive out of sequence, be corrupted or even not arrive at all (without indication to either sender or intended recipient that anything had gone wrong), it was decided to place another protocol layer on top of IP. This further layer was provided by TCP (Transmission Control Protocol), which allowed each end of a connection to acknowledge receipt of IP packets and/or request retransmission of lost or corrupted packets. In addition, TCP allows the packets to be rearranged into their correct sequence at the receiving end. IP and TCP are the two commonest protocols used on the Internet and are almost invariably coupled together as TCP/IP. TCI is the higher level protocol that uses the lower level IP.

For Internet applications, a four-layer model is often used, which is represented diagrammaticall in Fig. 1.1 below. The transport layer will often comprise the TCP protocol, but may be UDP (described in the next section), while the internet layer will always be IP. Each layer of the model represents a different level of abstraction, with higher levels representing higher abstraction. Thus, although applications may appear to be communicating directly with each other, they are actually communicating directly only with their transport layers. The transport and internet layers, in their turn, communicate directly only with the layers immediately above and below them, while the host-t network layer communicates directly only with the IP layer at each end of the connection. When a message is sent by the application layer at one end of the connection, it passes through each of the lower layers. As it does so, each layer adds further protocol data specific to the particular protocol at that level. For the TCP layer, this process involves breaking up the data packets into TCP segments and adding sequence numbers and checksums; for the IP layer, it involves placing the TCP segments into IP packets called **datagrams** and adding the routing details. The host-to-network layer then converts the digital data into an analogue form suitable for transmission over the carrier wire, sends the data and converts it back into digital form at the receiving end.

At the receiving end, the message travels up through the layers until it reaches the receiving application layer. As it does so, each layer converts the message into a form suitable for receipt by th next layer (effectively reversing the corresponding process carried out at the sending end) and carrie out checks appropriate to its own protocol. If recalculation of checksums reveals that some of the dat

has been corrupted or checking of sequence numbers shows that some data has not been received, the the transport layer requests re-transmission of the corrupt/missing data. Otherwise, the transport laye acknowledges receipt of the packets. All of this is completely transparent to the application layer. Once all the data has been received, converted and correctly sequenced, it is presented to the recipien application layer as though that layer had been in direct communication with the sending application layer. The latter may then send a response in exactly the same manner (and so on). In fact, since TCP provides full duplex transmission, the two ends of the connection may be sending data simultaneousl

The above description has deliberately hidden many of the low-level details of implementation, particularly the tasks carried out by the host-to-network layer. In addition, of course, the initial transmission may have passed through several routers and their associated layers before arriving at it ultimate destination. However, this high-level view covers the basic stages that are involved and is quite sufficient for our purposes.

Another network model that is often referred to is the seven-layer Open Systems Interconnection (OSI) model. However, this model is an unnecessarily complex one for our purposes and is better suited to non-TCP/IP networks anyway.

1.6 UDP

Most Internet applications use TCP as their transport mechanism. In contrast to TCP, User Datagram Protocol (UDP) is an unreliable protocol, since:

(i) it doesn’t guarantee that each packet of data will arrive;

(ii) it doesn’t guarantee that packets will be in the right order.

UDP doesn’t re-send a packet if it fails to arrive or there is some other error and it doesn’t re-assemble packets into the correct sequence at the receiving end. However, the TCP overhead of providing facilities such as confirmation of receipt and re-transmission of lost or corrupted packets used to mean that UDP was significantly *faster* than TCP. For many applications (e.g., file transfer), this didn’t really matter greatly. As far as these applications were concerned, it was much more important that the data arrived intact and in the correct sequence, both of which were guaranteed by TCP. For some applications, however, the relatively slow throughput speed offered by TCP was simply not feasible. Such applications included the *streaming* of audio and video files (i.e., the playi of those files while they were being downloaded). Such applications didn’t use TCP, because of its large overhead. Instead, they used UDP, since their major objective was to keep playing the sound/video without interruption and losing a few bytes of data was much better than waiting for re-transmission of the missing data.

Nowadays, network transmission speeds are considerably greater than they were only a few years ago, meaning that UDP is now a feasible transport mechanism for applications in which it would not once have been considered. In addition to this, it is much easier for TCP packets to get through firewalls than it is for UDP packets to do so, since Web administrators tend to allow TCP packets from remote port 80s to pass through unchallenged. For these reasons, the choice of whether to use TCP or UDP for speed-critical applications is not nearly as clear cut as it used to be.

Jan Graba, *An Introduction to Network Programming with Java*, 3rd ed. 2013, Java 7 Compatible, DOI: 10.1007/978-1-4471-5254-5_2, © Springer-Verlag London 2013

2. Starting Network Programming in Java

Jan Graba[1]

(1) Department of Computing, Sheffield Hallam University, Sheffield, South Yorkshire, UK

Abstract
Having covered fundamental network protocols and techniques in a generic fashion in Chap. 1, it is now time to consider how those protocols may be used and the techniques implemented in Java. Core package *java.net* contains a number of very useful classes that allow programmers to carry out network programming very easily. Package *java x .net*, introduced in J2SE 1.4, contains factory classes for creating sockets in an implementation-independent fashion. Using classes from these packages (primarily from the former), the network programmer can communicate with any server on the Internet or implement his/her own Internet server.

Learning Objectives
After reading this chapter, you should:

- know how to determine the host machine's IP address via a Java program;
- know how to use TCP sockets in both client programs and server programs;
- know how to use UDP sockets in both client programs and server programs;
- appreciate the convenience of Java's stream classes and the consistency of the interface afforded by them;
- appreciate the ease with which GUIs can be added to network programs;
- know how to check whether ports on a specified machine are running services.

Having covered fundamental network protocols and techniques in a generic fashion in Chap. 1, it is now time to consider how those protocols may be used and the techniques implemented in Java. Core package *java.net* contains a number of very useful classes that allow programmers to carry out network programming very easily. Package *java x .net*, introduced in J2SE 1.4, contains factory classes for creating sockets in an implementation-independent fashion. Using classes from these packages (primarily from the former), the network programmer can communicate with any server on the Internet or implement his/her own Internet server.

2.1 The *InetAddress* Class

One of the classes within package *java.net* is called *InetAddress*, which handles Internet addresses both as host names and as IP addresses. Static method *getByName* of this class uses DNS (Domain Name System) to return the Internet address of a specified host name as an *InetAddress* object. In order to display the IP address from this object, we can simply use method *println* (which will cause

the object's *toString* method to be executed). Since method *getByName* throws the checked exception *UnknownHostException* if the host name is not recognised, we must either throw this exception or (preferably) handle it with a `catch` clause. The following example illustrates this.

Example

```
import java.net.*;
import java.util.*;
public class IPFinder
{
        public static void main(String[] args)
        {
                String host;
                Scanner input = new Scanner(System.in);
                InetAddress address;
                System.out.print("\n\nEnter host name: ");
                host = input.next();
                try
                {
                        address = InetAddress.getByName(host);
                        System.out.println("IP address: "
                                                             + address.toString());
                }
                catch (UnknownHostException uhEx)
                {
                             System.out.println("Could not find " + host);
                }
        }
}
```

The output from a test run of this program is shown in Fig. 2.1.

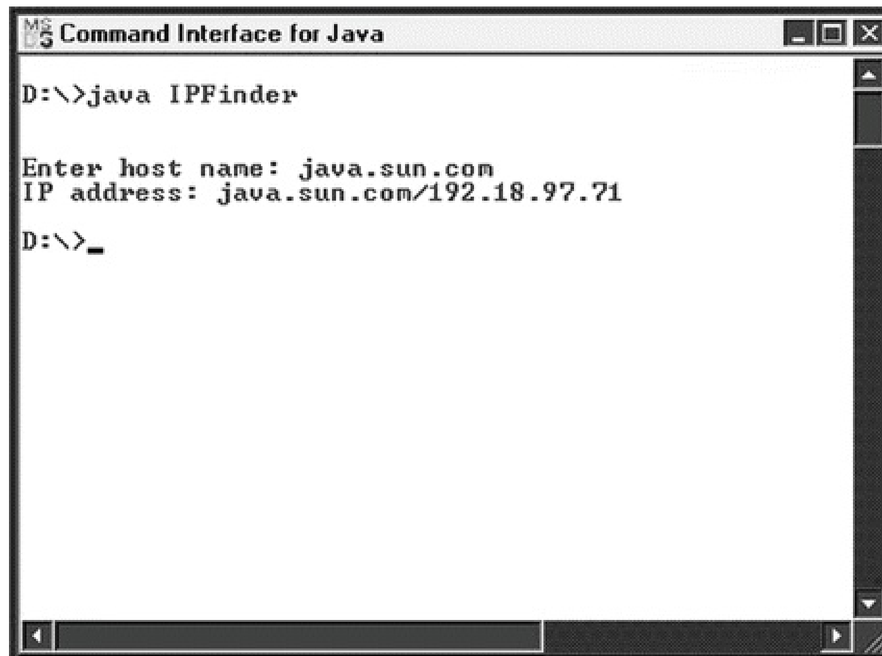

Fig. 2.1 Using method *getByName* to retrieve IP address of a specified host

It is sometimes useful for Java programs to be able to retrieve the IP address of the current machine. The example below shows how to do this.

Example

```
import java.net.*;
public class MyLocalIPAddress
{
    public static void main(String[] args)
    {
        try
        {
                InetAddress address =
                              InetAddress.getLocalHost();
                System.out.println(address);
           }
           catch (UnknownHostException uhEx)
            {
                System.out.println(
                           "Could not find local address!");
            }
    }
 }
```

Output from this program when run on the author's office machine is shown in Fig. 2.2.

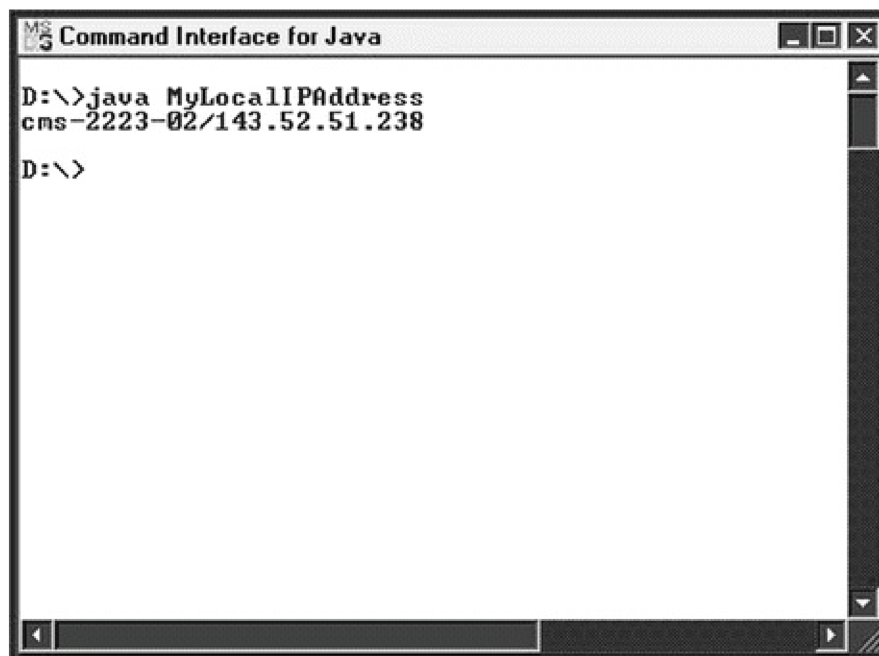

Fig. 2.2 Retrieving the current machine's IP address

2.2 Using Sockets

As described in Chap. 1, different processes (programs) can communicate with each other across networks by means of sockets. Java implements both **TCP/IP** sockets and **datagram** sockets (UDP sockets). Very often, the two communicating processes will have a *client/server* relationship. The steps required to create client/server programs via each of these methods are very similar and are outlined in the following two sub-sections.

2.2.1 TCP Sockets

A communication link created via TCP/IP sockets is a **connection-orientated** link. This means that the ~~connection between server and client remains open throughout the duration of the dialogue~~ between the two and is only broken (under normal circumstances) when one end of the dialogue formally terminates the exchanges (via an agreed protocol). Since there are two separate types of process involved (client and server), we shall examine them separately, taking the server first. Settin up a server process requires five steps…

1. Create a ServerSocket object .

The *ServerSocket* constructor requires a port number (1024–65535, for non-reserved ones) as an argument. For example:

```
ServerSocket serverSocket = new ServerSocket(1234);
```

In this example, the server will await ('listen for') a connection from a client on port 1234.

2. Put the server into a waiting state.

The server waits indefinitely ('blocks') for a client to connect. It does this by calling method *accept* of class *ServerSocket*, which returns a *Socket* object when a connection is made. For example:

```
Socket link = serverSocket.accept();
```

3. Set up input and output streams .

Methods *getInputStream* and *getOutputStream* of class *Socket* are used to get references to stream associated with the socket returned in step 2. These streams will be used for communication with the client that has just made connection. For a non-GUI application, we can wrap a *Scanner* object aroun the *InputStream* object returned by method *getInputStream*, in order to obtain string-orientated input (just as we would do with input from the standard input stream, *System.in*). For example:

```
Scanner input = new Scanner(link.getInputStream());
```

Similarly, we can wrap a *PrintWriter* object around the *OutputStream* object returned by method *getOutputStream*. Supplying the *PrintWriter* constructor with a second argument of `true` will cause the output buffer to be flushed for every call of *println* (which is usually desirable). For example:

```
PrintWriter output =
        new PrintWriter(link.getOutputStream(),true);
```

4. Send and receive data.

Having set up our *Scanner* and *PrintWriter* objects, sending and receiving data is very straightforward. We simply use method *nextLine* for receiving data and method *println* for sending data, just as we might do for console I/O. For example:

```
output.println("Awaiting data…");
String input = input.nextLine();
```

5. Close the connection (after completion of the dialogue).

This is achieved via method *close* of class *Socket*. For example:

```
link.close();
```

The following example program is used to illustrate the use of these steps.

<u>Example</u>

In this simple example, the server will accept messages from the client and will keep count of those messages, echoing back each (numbered) message. The main protocol for this service is that client and server must alternate between sending and receiving (with the client initiating the process with its opening message, of course). The only details that remain to be determined are the means of indicating when the dialogue is to cease and what final data (if any) should be sent by the server. For this simple example, the string "***CLOSE***" will be sent by the client when it wishes to close down the connection. When the server receives this message, it will confirm the number of preceding messages received and then close its connection to this client. The client, of course, must wait for the final message from the server before closing the connection at its own end.

Since an *IOException* may be generated by any of the socket operations, one or more `try` blocks must be used. Rather than have one large `try` block (with no variation in the error message produced and, consequently, no indication of precisely what operation caused the problem), it is probably good practice to have the opening of the port and the dialogue with the client in separate `try` blocks. It is also good practice to place the closing of the socket in a `finally` clause, so that, whether an exception occurs or not, the socket will be closed (unless, of course, the exception is generated when actually closing the socket, but there is nothing we can do about that). Since the `finally` clause will need to know about the *Socket* object, we shall have to declare this object within a scope that covers both the `try` block handling the dialogue and the `finally` block. Thus, step 2 shown above will be broken up into separate declaration and assignment. In our example program, this will also mean that the *Socket* object will have to be explicitly initialised to `null` (as it will not be a global variable).

Since a server offering a public service would keep running indefinitely, the call to method *handleClient* in our example has been placed inside an 'infinite' loop, thus:

```
do
{
     handleClient();
}while (true);
```

In the code that follows (and in later examples), port 1234 has been chosen for the service, but it could just as well have been any integer in the range 1024–65535. Note that the lines of code corresponding to each of the above steps have been clearly marked with emboldened comments.

```
//Server that echoes back client's messages.
//At end of dialogue, sends message indicating
//number of messages received. Uses TCP.
import java.io.*;
import java.net.*;
import java.util.*;
public class TCPEchoServer
{
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;
    public static void main(String[] args)
    {
        System.out.println("Opening port…\n");
        try
        {
            serverSocket = new ServerSocket(PORT); //Step 1.
        }
        catch(IOException ioEx)
        {
            System.out.println(
                                     "Unable to attach to port!");
            System.exit(1);
        }
        do
        {
            handleClient();
        }while (true);
    }
    private static void handleClient()
```

```
    {
        Socket link = null;                             //Step 2.
        try
        {
            link = serverSocket.accept();               //Step 2.
            Scanner input =
                new Scanner(link.getInputStream()); //Step 3.
            PrintWriter output =
                    new PrintWriter(
                        link.getOutputStream(),true); //Step 3.
            int numMessages = 0;
             String message = input.nextLine();    //Step 4.
            while (!message.equals("***CLOSE***"))
            {
                System.out.println("Message received.");
                numMessages++;
                output.println("Message " + numMessages
                                      + ": " + message); //Step 4.
                message = input.nextLine();
            }
            output.println(numMessages
                              + " messages received."); //Step 4.
          }
          catch(IOException ioEx)
          {
              ioEx.printStackTrace();
          }
          finally
          {
             try
             {
                 System.out.println(
                              "\n* Closing connection… *");
                 link.close();                          //Step 5.
             }
             catch(IOException ioEx)
             {
                  System.out.println(
                                  "Unable to disconnect!");
                System.exit(1);
             }
          }
    }
  }
```

Setting up the corresponding client involves four steps…

1. Establish a connection to the server.

We create a *Socket* object, supplying its constructor with the following two arguments:

sample content of An Introduction to Network Programming with Java: Java 7 Compatible

- download online Women of the Vine: Inside the World of Women Who Make, Taste, and Enjoy Wine pdf, azw (kindle), epub
- Think: Why You Should Question Everything for free
- download online The Ultimate Book of Saturday Science: The Very Best Backyard Science Experiments You Can Do Yourself
- download online The Last Bookaneer: A Novel online

- http://bestarthritiscare.com/library/Women-of-the-Vine--Inside-the-World-of-Women-Who-Make--Taste--and-Enjoy-Wine.pdf
- http://test.markblaustein.com/library/Magic-for-Beginners.pdf
- http://www.1973vision.com/?library/Steam-Train--Dream-Train.pdf
- http://test.markblaustein.com/library/Gender-Outlaws--The-Next-Generation.pdf