Chapter 4

Let's Talk

If you made a list of programs that you use frequently, chances are that the list would include several programs that depend on network access such as your web browser, email program, or IM/chat program. Given the importance of such programs, we introduce the basic techniques used to write programs that use the network in this chapter.

The features of the Java language and libraries that we will employ to write network programs are actually quite simple. If all we needed to do was present these details, then this would be a very short chapter. In addition, however, we have to give you some background on the fundamentals of communications between computers on the Internet. Basically, teaching you the Java mechanisms required to send a message through the network will be of little use to you if you don't know what messages to send. Therefore, we will begin by exploring the nature of the "conversations" that take place between computers on the Internet. Then, we will study the Java primitives used to send and receive messages and use these primitives to construct examples of programs that use the network.

4.1 Protocols

Communications between humans depend on a vast collection of rules and conventions we share with one another. Most obviously, there are the rules of the languages we use. Beyond the grammar and vocabulary, however, there are shared expectations about the forms conversations will take. Introductory phrases like "Good morning" and "Can I ask you a question?" don't really add much useful information to a discussion, but if a speaker doesn't include such phrases, chances are that the flow of conversation will suffer. The aphorism "It's not what you say, but how you say it" applies to very mundane aspects of our speech and writing.

We are so familiar with such conventions that we usually take them for granted. One context where they are more explicitly recognized is when we talk on the telephone. A web search for the phrase "Telephone Etiquette" yields an extensive collection of documents providing advice on how to conduct a proper telephone conversation. For example, the University of California at Fullerton has a telephone etiquette web site for their office staff that includes little "algorithms" for handling phone calls such as:

- Answer the phone by saying: "[Department name], how may I help you?"
- If the caller asks to speak to the dean (for example), ask "May I tell him/her who is calling?"
  - Ask the caller "What is this in regard to?" (if appropriate)

- Press Xfer and the extension.
- Wait for the dean to answer.
- Announce the name of the caller.
- Wait for a response as to whether the call will be taken.
  - If the called party wishes to take the call, press the Xfer button again.
  - If the calling party does not wish to take the call, press the RLSE button and then the button where the caller is. SAY: "_______ is out of the office, may I take a message or would you like his/her voicemail?"

The site also provides somewhat amusing advice including the importance of saying "He has stepped out of the office. Would you like to leave a message on his voicemail?" rather than saying "He is in the men's room."[1]

One interesting aspect of the advice provided by such sites it that it is clear that no single script is appropriate for all telephone calls. While the instructions above correspond to the conversational pattern you would expect if you called the administrative offices at a school, you would surprise many callers if you followed these instructions when answering the phone at home or in your dorm room. We have different patterns we follow for different forms of conversation. A complete guide to telephone etiquette would have to cover how to handle calls while at home, in the office, and even when answering the phone for friends while visiting their home. It would also have to discuss placing a call to a friend, placing a call to make dinner reservations, placing a call to order a pizza, and many other scenarios. Each situation would involve different patterns and expectations.

Communications between computers depend on similar rules and conventions. In fact, if anything, such rules and conventions are even more critical to computer communications than to human communications.

All data sent from one computer to another will ultimately be just a sequence of binary symbols. For any information to be transferred, the designers of the software that sends the messages and the software that receives the messages must agree on a common "language" that associates meanings with such sequences of 0s and 1s. In addition, communications software must have conversational patterns to follow. When two computers communicate, they do so because they are following algorithms that specify what messages to send and when to send them. As usual, the computers do not really understand the purpose of the algorithms they are following. In particular, the computers don't really understand the purposes of the messages they send and receive.

If a person says something unexpected (or at an unexpected point) in a conversation, the listener may be surprised, but the conversation can usually continue because the individuals involved understand what they are doing well enough to adjust. Since a computer does not really understand the conversation in which it is participating, it cannot adjust to the unexpected. As a result, it is critical that communications between two computers follow carefully designed conversational plans. These conversational plans determine the form of the algorithms implemented by programs that involve network communications.

A collection of rules that specify the manner in which two computers should communicate is called a *protocol*. Like humans, computers have different conversational rules to follow depending

[1]At the time of writing, this web page could be found at:
http://www.fullerton.edu/it/services/Telecomm/FAQ/etiquetteguide.asp
If it is still available, it is definitely worth visiting just for its clear statement debunking the "Three Myths about Students/Callers" — 1. Students try to make things difficult; 2. Students like to complain; and 3. Students expect the impossible.

on the type of communications involved. For example, the collection of rules a computer on the Internet follows when it wants to ask an email server to transmit a message is called the Simple Mail Transfer Protocol or SMTP. On the other hand, when asking an email server to allow you to read mail you have received, your computer probably follows either a set of rules known as the Post Office Protocol (POP) or the Internet Mail Access Protocol (IMAP). The protocol a computer must follow when fetching a web page is called the Hypertext Transfer Protocol or HTTP. There are even protocols whose names don't end with the word "protocol"! The rules a computer must follow when talking to AOL's IM service are named "Open System for CommunicAtion in Realtime" (or OSCAR for short).

4.1.1 The Client/Server Paradigm

In many conversations, people participate as equals or peers. As two friends approach one another on the street, either person might start a conversation by saying "Hi." Either person may ask a question or make a comment about the weather, what the other person is wearing, or any other subject that comes to mind. Throughout the ensuing conversation either party may change the subject when it seems appropriate, and, eventually, whoever feels like it first will say "Goodbye" and continue on their way.

There are, however, many examples of situations where the participants in a conversation have very distinct roles that play a major part in determining which party says what and when they say it. Consider for example, the "conversation" that occurs when you make a call to place an order using a 1-800 number (assuming, for the sake of discussion, that you decided not to just order through the web). When you place such a call, you expect to hear a pleasant voice say something like "Thank you for calling 1-800-Flowers. My name is Brian. How can I help you today?"

First, note that in such a conversation you consider it your role to start the conversation by placing the call. You would probably react very differently if, instead, your phone rang and someone said "I'm calling from 1-800-Flowers. My name is Brian. How can I help you today?"

There are many other points in such a conversation where it is clear that each party has a distinct role and that these roles are not interchangeable. Near the end of the call, you expect the salesperson to ask for your credit card information and you willingly provide it. Someday, try asking the salesperson to give you his or her credit card information during one of these calls.

Many computer protocols, including all the protocols identified in the preceding section, assume that the computers involved will be assigned roles somewhat similar to those we can identify in a 1-800 call. One computer plays the role of the customer or *client*. The other plays the role of salesperson or *server*. The computer playing the client role starts the conversation just as a customer is expected to place a 1-800 call. The client computer then typically provides information indicating what it wants the server computer to do and justifying the request (e.g. by providing a password). In response, the server performs the task the client requested. This may include activities like delivering an email message or providing information from a data base. In many such computer protocols, the interchange of information takes place through a series of exchanges. This style of interaction is so common in computer protocols, that it is given a name of its own — *the Client/Server Paradigm.*

We have seen that a computer's behavior is determined by the instructions that make up the program it is executing. From this we can conclude that the program being executed will determine whether a computer acts as a client or a server in a particular exchange of network messages. Thus, when you launch an IM chat program like AIM, Adium, or iChat on your computer, the machine suddenly becomes an IM client. Similarly, when you launch a web browser like FireFox, Safari, or Internet Explorer, it becomes a web or HTTP client.

Servers work the same way. There is no fundamental difference between your computer and the AOL chat server or Yahoo's web server. What makes a machine become a server is the fact that someone installs and runs an appropriate program on the machine. Such programs are not as well known as client programs, but they definitely exist. For example, Apache and Microsoft's Internet Information System (IIS) are two examples of web server programs. Sendmail, Eudora WorldMail, and Microsoft Exchange Server are examples of electronic mail server programs.

You are probably accustomed to running several programs on your computer at the same time. For example, you might find yourself switching back and forth between a word processor, a chat program, your web browser, and a program for editing pictures from your digital camera. If several of these programs are network-based, your computer may be playing the role of client for several different protocols at the same time. Similarly, it is possible to run several server applications on a computer at the same time. Therefore, some computers are simultaneously functioning as web servers and mail servers. In fact, a single computer can be both a server and a client at the same time. For example, if you share music from your computer using iTunes, your computer is acting as a server. While it is doing this, however, you can also have it act as a client for a different protocol by using your email program or web browser. The point is that while the types of messages a computer can send as part of a single conversation are strictly constrained by the role it is playing under a given protocol, a computer can be configured to play different roles in different conversations quite flexibly.

4.1.2 Mail Servers and Clients

We can make things more concrete by examining the use of servers and clients in the protocols that are used to handle electronic mail. The Internet's email system is based on the existence of servers that hold messages for users between the point when a message is first sent and when the person to whom the message is addressed reads the message. These mail servers are expected to be continuously available to respond to client requests.

You may use a mail server provided by the same organization through which you obtain Internet access or by an independent organization. For example, while on a college campus, you probably depend on the college's computer services department for network access. They most likely also operate a mail server that you can use. Many home users obtain network access through a telephone or cable television provider. These companies typically provide mail servers that their subscribers can use. On the other hand, there are many companies that are in the business of providing mail service that do not also provide Internet access to their users. Yahoo and Hotmail are two well-known examples. Whether you access the Internet through a school's network or through a commercial provider you can decide to use a distinct provider like Yahoo or Hotmail for your email server. You tell your email program what mail server it should use as part of its configuration process.

When you send a message, your mail program delivers the message to your mail server by sending the server a series of messages that follow the rules of the SMTP protocol. If the message is addressed to a person who uses the same mail server as you, then the server simply holds the message until the intended recipient requests to see his or her mail. Figure 4.1 depicts such a mail transfer. In the figure, the computers that look like laptops represent user computers. For this scenario, there are just two users named Alice and Bob. The monitor and keyboard between the

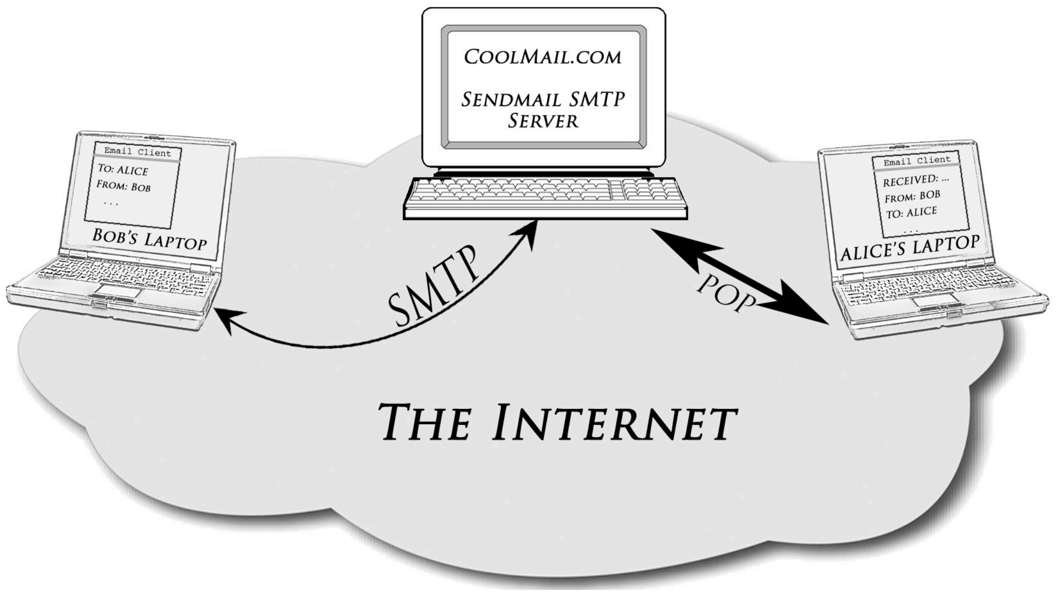


Figure 4.1: Email exchange between two users through a common mail server

laptops represents a computer that acts as a mail server for both Alice and Bob. This computer has the name `coolmail.com`.

Both users are running mail client programs on their computers. Bob is using his email client to send an email to Alice. In this case, Bob would enter something like `alice301@coolmail.com` as the "To" address for the message. To deliver this message, Bob's client will exchange a series of messages through the Internet with his server as depicted by the curved, double-tipped arrow in the figure. These messages will conform to the rules of SMTP. After this exchange is complete, the text of the message will be stored on the `coolmail.com` server. At some later point, Alice will ask her email client to check for new mail (or it might be configured to do this automatically every few minutes). When requesting to see mail, the recipient's mail program communicates with the server using a protocol that is different from SMTP. The two most widely used protocols for retrieving mail from a server are POP and IMAP. In the figure, Alice's email client exchanges a series of messages that conform to the POP protocol with the server to check for new mail. This exchange is shown as a straight double-tipped arrow in the figure. After it is complete, Bob's message will appear in the list of unread messages displayed by Alice's email program.

If a message is sent to someone who uses a different mail server, then a bit more work is required. This scenario is depicted in Figure 4.2. The email client program still passes the message to the sender's mail server using SMTP. The sender's mail server, however, does not just store the message until the recipient asks for it. Instead, the sender's mail server passes the message on to the recipient's mail server as soon as it can. The communication between the sender's mail server and the recipient's mail server is also conducted by following the rules of SMTP.

In the figure, Bob is shown sending an email to a user named Carol. Carol uses her school's email server which has the name `myuniversity.edu`. Therefore, the figure shows that two conversations based on SMTP are required to deliver the message. First, Bob's email program delivers the message to Bob's server, `coolmail.com`. Rather than simply holding the message, `coolmail.com`

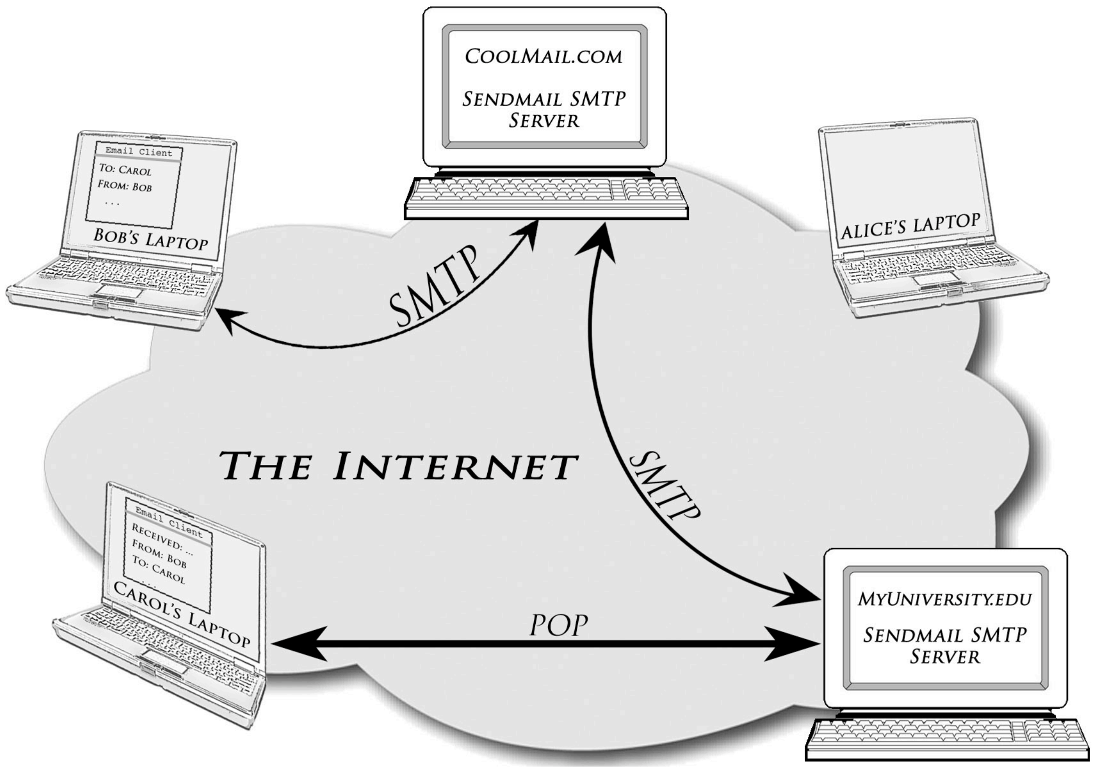


Figure 4.2: Email exchange between two users with distinct mail servers

will quickly attempt to forward the message to Carol's server, `myuniversity.edu`, by exchanging a similar sequence of SMTP messages with that server. Ultimately, Carol will retrieve the message and any other mail she may have received by exchanging POP (or IMAP) messages with her mail server.

4.1.3 The Transmission Control Protocol

Suppose for a moment that you did set out to write a complete guide to telephone etiquette. This would involve writing guidelines for many different types of phone calls. The guidelines for calling home on Mother's Day would clearly be different from those for calling to make dinner reservations or handling a call from a telephone solicitor. Making any call, however, requires certain common steps. You have to pick up the receiver (or push the right button on a cell phone). You have to wait for a dial tone. You have to know how to dial the number (including whether to include an area code or not). You have to know what to do if the phone just keeps ringing and ringing. You would not want to discuss these details separately in your guidelines for each type of phone call a person might make. You would instead either just assume people knew how to make calls or write a set of instructions describing the basics of making a phone call that a person could refer to as necessary while following the instructions for a particular type of phone call.

Similarly, while network protocols for sending emails, fetching web pages and chatting through AOL's IM are all different, they all involve common elements. In the Internet, many of these details are collected in a protocol called the Transmission Control Protocol or TCP.[2]

Each of the protocols mentioned in the preceding sections is designed to support a particular application of the Internet. HTTP is designed to support communications between web browsers and web servers, and SMTP, POP, and IMAP are all designed to support email. As a result, such protocols are known as *application* protocols. TCP, on the other hand, is not designed to support any particular application. Instead it addresses fundamental communication issues that arise in many distinct applications. In recognition of this difference, TCP is described as a *transport* protocol. Within the Internet, TCP is the most commonly used transport protocol.

There are two particular aspects of TCP that will become relevant as we explore how to write Java programs that use the network. The first is that TCP, like the telephone system, is based on the notion of conversations. To appreciate this assumption, you have to recognize the fact that there are forms of communications that can't really be called conversations. Contrast the way you talk to someone on the telephone with the way you communicate by email. When you are talking to someone on the phone you don't try to say everything you wanted to say at once. Instead, you provide a little information, wait to hear how the person to whom you are talking responds, and then repeat that process until you are done. With email, however, you typically put everything you want to say in one message. Normally, the first thing you say when you make a phone call is "Hi," possibly also taking the time to say who you are. On the other hand, it would be very odd to send someone an email that simply said "Hi. This is Tom."

The expectation that a person you are addressing will respond is integral to our notion of a conversation. If you sent someone an email in the morning suggesting that they meet you for lunch and they showed up for lunch without sending you a response, you would not be terribly surprised. On the other hand, if you called someone to suggest lunch in the morning and they hung up without agreeing or at least saying "See you then," you would assume that you were accidentally disconnected and try to call them again. Of course, you can only be "disconnected" if you thought you were "connected." This notion of a "connection" is the concrete way that both the phone system and TCP make the concept of a conversation a part of the communication process. When you want to send an email, you just send it. When you want to talk to someone on the phone, you have to first "call" their number to get your phone "connected" to theirs. After you have sent an email to someone, there is no way that your computer can tell whether or not you will send another email to the same person in the near future. On the other hand, when a phone conversation is finished, you let the phone company know by hanging up.

TCP depends on a similar notion of connections. When one program wants to talk to another it begins by making a connection to the other program. Just as we have written programs that "make" `JButton`s and `JTextField`s by executing constructions, you will learn how to write a construction that makes a new network connection, essentially "calling" the other computer. Once the connection is established, the computers at either end can send each other messages through the connection. After the programs have exchanged all desired messages, they can "hang up" by invoking a method to `close` the connection.

The other aspect of TCP that you will need to understand involves how a program identifies the other computer to which it wants to be connected.

In order to call someone on the phone, you need to know the other person's phone number.

[2] You may have encountered the abbreviation TCP/IP while configuring your computer's network preferences or elsewhere. This abbreviation is used to refer to the suite of communication protocols used on the Internet. The fact that TCP is included in this abbreviation reflects that fact that it is a very important component of the collection of Internet protocols.

More basically, however, you have to know how phone numbers work. That is, you have to know things like when to include an area code and how to decide when to start by dialing 1. This may seem trivial to you, but that is only because you use the system every day. Dealing with an unfamiliar phone system quickly makes it obvious that some basic knowledge of the structure of phone numbers is essential. For example, we searched the web yellow pages for some Australian airlines and found the following entries:

> **Jetstar** — For All Day, Every Day, Low Fares
> ph: 13 1538
>
> **Royal Brunei Airlines** — Value Fares To Asia, Middle East & Europe.
> ph: (02) 8267 5300
>
> **Freedom Air** — "Really, Really Small Fares"
> ph: 1800 122 000

Based on these entries, how many digits long do you think an Australian phone number is? Do they have area codes? How long are their area codes?

You are almost certainly familiar with some aspects of the mechanisms TCP uses to identify the endpoint of a connection. Email addresses and web page addresses usually contain names like hotmail.com or www.google.com. These names are called *domain names.* They can be used within a program to identify the remote machine with which the program wants to establish a connection. Each domain name consists of a series of short identifiers separated from one another by periods.

You may also have seen machines identified using sequences of number separated by periods. For example, the sequence 64.233.161.147 is (at least at the moment this paragraph is being written) another way of identifying the machine called www.google.com. Such a sequence of numbers is called an *Internet Address* or *IP address* for short. I can access the Google web site by entering either the web address `http://www.google.com` or `http://64.233.161.147`.

IP addresses are TCP's equivalent of telephone numbers. To create a TCP connection to another machine, a computer needs to know the IP address of the other machine. Fortunately, the Internet provides a service called the *Domain Name System* that acts like a telephone directory for the Internet. Given a domain name, a computer can use the domain name system to determine the IP address associated with the domain name. The software that implements TCP performs domain name lookups automatically. As a result, a programmer can either provide an IP address or a domain name to create a connection.

Identifying the machine that you want to talk to, however, is not enough. As we explained earlier, a single machine may be running a program that makes it act as a web server at the same time that it is running another program that makes it act as a mail server. As a result, it isn't enough to send a message to a particular machine. You instead have to send each message to a particular program on a particular machine. An IP address alone only identifies a machine. Therefore, TCP requires a bit more addressing information for the messages it delivers.

The extra information TCP uses to identify the particular program a message should be delivered to is called a *port number.* Continuing with our analogy with the telephone system, a good way to think of a port number is that it is much like a telephone extension number. Many large organizations have a single telephone number. Within the organization, departments and/or individuals are assigned extension numbers. When you call the organization's telephone number, an operator or an automatic menu system asks you to provide the extension for the person you are trying to contact and then connects you with the requested extension. Similarly, within a computer running several programs that use the network, each program is assigned a port number. When you want to create a connection to a particular program on a machine, you use the machine's IP address together with the port number assigned to the program as the complete address.

By convention, particular port numbers are associated with programs providing common network services. For example, port 80 is associated with web servers and port 25 is associated with servers designed to accept email for delivery using the SMTP protocol. The complete address used to connect to the Yahoo web server would therefore be (www.yahoo.com, 80) or (64.233.161.147, 80) while the address for the mail server at Hotmail is (mail.hotmail.com, 25) or (65.54.245.40, 25).

4.1.4 Talking SMTP

As a last bit of background before we discuss the mechanisms used within a Java program to send messages through the network, we will explore the rules of one important Internet application protocol, the Simple Mail Transfer Protocol (SMTP). We will later use examples from this protocol to illustrate the use of Java network programming mechanisms.

Internet protocols are described in documents know as RFCs (Requests for Comment). The complete details of the SMTP protocol are provided in one of these documents. RFCs are identified by number. SMTP is described by RFC 2822. If you wish to learn more about SMTP than we provide below, an online copy of this document can be found at

`http://www.ietf.org/rfc/rfc2821.txt`

One rule of the SMTP protocol is that it is the client program's responsibility to start the process of communication. The client does this by establishing a connection to the SMTP server. The protocol does not actually require that this be a TCP connection, but TCP is the mechanism most commonly used for SMTP connections. The protocol also states that as soon as the connection is established, the server should send a message to the client identifying itself. For example, if a program makes a connection to port 25 on the server mail.adelphia.com, it will quickly be sent a message like:

> 220 mta9.adelphia.net ESMTP server (InterMail vM.6.01.05.02 201-2131-123-102-20050715) ready Wed, 28 Jun 2006 11:24:37 -0400

On the other hand, if it connects to port 25 on mail.hotmail.com, the program will receive a message like:

> 220 bay0-mc1-f10.bay0.hotmail.com Sending unsolicited commercial or bulk email to Microsoft's computer network is prohibited. Other restrictions are found at http://privacy.msn.com/Anti-spam/. Violations will result in use of equipment located in California and other states. Wed, 28 Jun 2006 22:42:26 -0700

Obviously, although the SMTP protocol governs the form of messages exchanged between a client and server, it still leaves a great deal of flexibility. While mail.adelphia.com appears to introduce itself using computerese, Microsoft's Hotmail server clearly prefers legalese. It is a bit hard to see how two messages that are so different could be conforming to the same guidelines. In fact, these two messages reveal quite a few interesting aspects of the SMTP protocol.

First, both of these messages are composed entirely of text. That is, while the actual messages that travel between the two computers are just sequences of 0s and 1s, when SMTP is being used

Reply Code	Interpretation
220	Service ready
221	Service closing transmission channel
250	Requested mail action completed
354	Start mail input
450	Requested mail action not taken: mailbox unavailable
452	Requested action not taken: insufficient system storage
500	Syntax error, command unrecognized
501	Syntax error in parameters or arguments

Figure 4.3: A sampling of SMTP server reply codes

these sequences are all interpreted as text encoded using a standard encoding system called ASCII (American Standard Code for Information Interchange). The designers of SMTP could have instead used special binary codes to reduce the number of bits required in many cases. We can speculate that the use of text for all SMTP messages was motivated by the effort required to correct mistakes in early implementations of the protocol. If someone wrote a program that was intended to act as an SMTP client or server, and the program did not work as expected, it would likely be necessary to examine the messages exchanged as part of the process of isolating and correcting the mistake. Messages encoded in ASCII could be examined using standard tools for text processing. If SMTP had used a specialized encoding scheme, specialized tools would have been required.

In addition to using text, the designers of SMTP gave those implementing server programs considerable leeway to provide information that might be useful to a human reader beyond the information required by the protocol for use by the client program. The introductory messages received from the two servers above each start with two required fields: the code "220" and the official domain name of the server to which the client has connected.[3] The server is free to provide any additional information it feels might be useful in the remainder of the message.

The additional information included by the server `mail.adelphia.com` seems designed to help someone struggling to diagnose a problem with mail software. It provides a description of the mail server program being run on `mail.adelphia.com`, `InterMail`, including what appears to be a precise version number. It also includes the time and date at which the message was sent. The message sent by the Hotmail server, on the other hand, suggests that Microsoft might be less concerned with helping someone debug a mailer than with establishing a legal basis for enforcing their use policies. This must be because their software does not contain any bugs.

The code "220" at the beginning of these two server messages is actually the most important piece of information from the point of view of the client software. Each message an SMTP server sends starts with such a three digit code. A list of the codes used and their interpretations is included in the specification of the protocol. A portion of this list is reproduced in Figure 4.3. As indicated in the figure, the code 220 simply indicates that the server is ready. In many cases, the code is the only information required in the message.

All messages the server sends to the client during an SMTP exchange are sent in reply to actions performed by the client. The introductory "220" message is sent in response to the client creating a

[3]A machine on the Internet can be associated with several "nicknames" in addition to its official name. Thus, in both the examples shown, the name that follows the "220" code is somewhat different from the name of the host to which we said we were connecting.

connection. All other server messages are sent in response to messages sent by the client. Messages sent by the client are called commands. While each server reply starts with a three digit numeric code, each command sent by the client must begin with a four character command name. The use of a fixed length for all command names simplifies writing client software, but it leads to some odd spellings. For example, the first "command" the client is expected to sent is a "Hello" message in which the client identifies itself in much the same way the server is identified within the "220" reply message. The command name used in the message is therefore "HELO".[4] The command code should be followed by the name of the client machine. Therefore, a typical `HELO` command might look like:

```
HELO tcl216-44.cs.williams.edu
```

The server must send a reply to every client command. If the server is able to process a client command successfully, it sends a reply starting with the code 250. In the case of the `HELO` command, most servers define success rather loosely. Typical servers will respond to any HELO command with a 250 reply, even if the machine name provided is clearly incorrect. For example, sending the command

```
HELO there
```

to the SMTP server at `mail.yale.edu` elicits a reply of the form

```
250 po09.its.yale.edu Hello tom.cs.williams.edu [137.165.8.83], pleased to meet you
```

The server clearly knows that the domain name included in the command is incorrect since it includes the correct domain name for my machine in its reply. Also note that this particular server program takes advantage of the fact that most of the text in its reply is ignored by the client software by including a cute little "pleased to meet you" in the message.

The actual transfer of a mail message is accomplished through a sequence of three client commands named `MAIL`, `RCPT`, and `DATA`.

The `MAIL` command is used to specify the return address of the individual sending the message. The command name is followed by the word "FROM" and the sender's email address. Thus, if a mail client was trying to deliver the message from Bob to Carol discussed in section 4.1.2, it might start by sending the command

```
MAIL FROM:<bob@coolmail.com>
```

The server will usually respond to this command with a 250 reply such as

```
250 <bob@coolmail.com>...  Sender ok
```

Next, the client specifies to whom the message should be delivered by sending one or more `RCPT` (i.e., recipient) commands. In each such command, the command name is followed by the word "TO" and the email address of a recipient. In the case of Bob's email program trying to deliver a message to Carol, this command might look like

```
RCPT TO:<carol@myuniversity.edu>
```

[4]To support extensions to the original protocol, the current version of SMTP allows clients to substitute an "EHLO" command for the "HELO" command. Use of this peculiar spelling informs the server that the client would like information about which SMTP protocol extensions the server supports.

and would elicit a reply such as

```
250 <carol@myuniversity.edu>...  Recipient ok
```

from the server.

Finally, the client initiates the transmission of the actual contents of the mail message by sending the command

```
DATA
```

to the server. The server will probably respond to this message with a reply of the form

```
354 Enter mail, end with "." on a line by itself
```

The server uses the 354 reply code rather than 250 to indicate that it has accepted but not completed the requested `DATA` command. It needs to receive the contents of the mail message before it can complete the command. After the 354 reply code, it kindly tells us how to send the message (even though any mail client program that received this reply should already know!). After sending a DATA command, a mail client sends the actual contents of the mail message to the server line by line. When it is done, it informs the server by sending a single period on a line by itself. Once it sees the period, the server will try to deliver the message and send a 250 reply code if it can.

There are several additional commands and many more reply codes than we have discussed. For our purposes, however, we only need to use one additional command. When the client has finished sending messages to the server it sends the server a command of the form

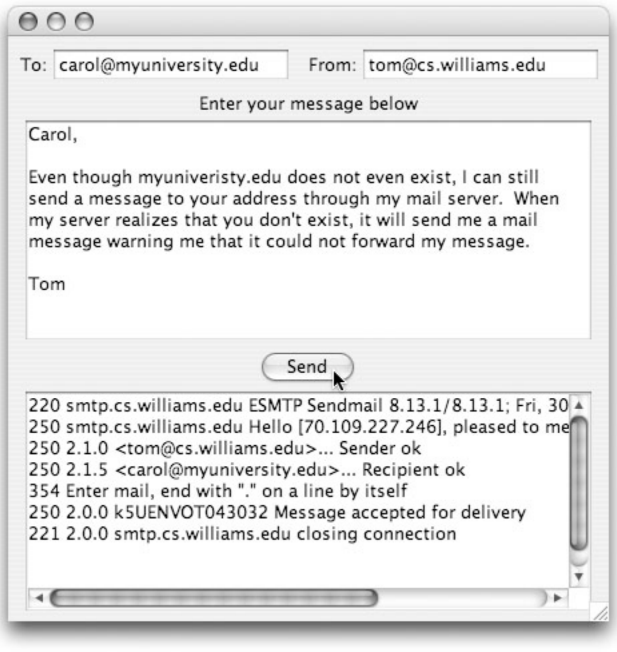
```
QUIT
```

The server will then send a 250 reply message and disconnect from the client. That is, while it is the client's job to create the connection to the server, it is the server that gets to "hang up" first when they are done.

4.2 Using `NetConnections`

We noted that all command and reply messages sent using SMTP are encoded as text using the ASCII code. Many other important Internet protocols share this property, including POP, IMAP and HTTP. There are, however, examples of protocols that depend on more specialized encodings of information in binary including the OSCAR instant messaging protocol and the DNS protocol used to translate domain names into IP addresses.

The Squint library contains several classes designed for implementing programs that involve network communications. Among these are two classes named `TCPConnection` and `NetConnection`. The `TCPConnection` class provides general purpose features that make it flexible enough to work with specialized encoding schemes. It provides the functionality needed to implement clients and servers based on protocols like OSCAR and DNS. `NetConnection`, on the other hand, provides primitives for network communication that are specialized to make it easy to implement programs that use text-based protocols. To simplify our introduction to network programming in Java, we will restrict our attention to such text-based protocols in this chapter. We will introduce the use of the `NetConnection` class and, as an example, present the implementation of a very simple mail client.



carol@myuniversity.edu

tom@cs.williams.edu

Carol,

Even though myuniveristy.edu does not even exist, I can still send a message to your address through my mail server. When my server realizes that you don't exist, it will send me a mail message warning me that it could not forward my message.

Tom

```
220 smtp.cs.williams.edu ESMTP Sendmail 8.13.1/8.13.1; Fri, 30
250 smtp.cs.williams.edu Hello [70.109.227.246], pleased to me
250 2.1.0 <tom@cs.williams.edu>... Sender ok
250 2.1.5 <carol@myuniversity.edu>... Recipient ok
354 Enter mail, end with "." on a line by itself
250 2.0.0 k5UENVOT043032 Message accepted for delivery
221 2.0.0 smtp.cs.williams.edu closing connection
```

Figure 4.4: GUI interface for a simple SMTP client

4.2.1 An Interface for an SMTP Client

The GUI interface for the program we will implement is shown in Figure 4.4. The program is really only half of a mail client. It can be used to send mail, but not to read incoming mail messages. As a result, it only sends messages based on the SMTP protocol, while a complete mail program would also have to use POP or IMAP.

The program provides two text fields and a text area where the user can enter the destination address for the email message, the user's own address, and the body of the message to be sent. Once the correct information has been placed in these areas, the user can tell the program to send the message by pressing the “Send” button. Unlike most mail client programs, this program provides no mechanism that allows the user to specify the name of the SMTP server to contact when a message is to be delivered. That information is included in the actual text of the program. This is not a desirable feature, but it will make the example a bit simpler.

The text area at the bottom of the program window is used to display the reply messages received from the SMTP server. This information would not be displayed by a typical mail client, but it is displayed by our program to make it clear how the process depends on the underlying protocol. The snapshot shown in Figure 4.4 shows how the window would look shortly after the user pressed the “Send” button. At this point, we can see the complete list of replies from the server displayed in the area at the bottom of the window.

The instance variable declarations and the constructor for such a program are shown in Figures 4.5 and 4.6. None of this code actually involves using the network. Instead, it only creates the GUI components described above and adds them to the program's window. The only real clues that this program will involve network access are the declarations of the variables `SMTP_SERVER` and `SMTP_PORT`. These variables are associated with the name of the server the program will con-

```
import squint.*;
import javax.swing.*;

//  A simple client program that can be used to send email messages
// through an SMTP server.
public class SMTPClient extends GUIManager {
   // Change these values to adjust the size of the program's window
   private final int WINDOW_WIDTH  = 450, WINDOW_HEIGHT = 460;

   // Amount of space to allow in text fields and text areas
   private final int ADDR_WIDTH = 14;

   // How many characters wide program's text areas should be
   private final int AREA_WIDTH = 35;

   // How many lines tall the message area should be
   private final int MESSAGE_LINES = 10;

   // How many lines tall the log area should be
   private final int LOG_LINES = 10;

   // Address of SMTP server to use
   private final String SMTP_SERVER = "smtp.cs.williams.edu";

   // Standard port number for connection to an SMTP server
   private final int SMTP_PORT = 25;

   // Fields for to and from addresses
   private JTextField to;
   private JTextField from;

   // Area in which user can type message body
   private JTextArea message;

   // Area in which responses from server will be logged
   private JTextArea log;
```

Figure 4.5: Instance variable declarations for an SMTP client

```
   // Place fields and text areas on screen to enable user to
   // enter mail.  Provide a "Send" button and a text area in which
   // server replies can be displayed.
   public SMTPClient() {
      // Create window to hold all the components
      this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

      // Create fields for to and from addresses
      JPanel curPane = new JPanel();
      curPane.add( new JLabel( "To:" ) );
      to = new JTextField( ADDR_WIDTH );
      curPane.add( to );
      contentPane.add( curPane );

      curPane = new JPanel();
      curPane.add( new JLabel( "From:" ) );
      from = new JTextField( ADDR_WIDTH );
      curPane.add( from );
      contentPane.add( curPane );

      // Create the message entry area
      contentPane.add ( new JLabel( "Enter your message below") );
      message = new JTextArea( MESSAGE_LINES, AREA_WIDTH );
      contentPane.add( new JScrollPane( message ) );

      // Add the Send button
      contentPane.add( new JButton( "Send" ) );

      // Create the server response log
      log = new JTextArea( LOG_LINES, AREA_WIDTH );
      contentPane.add( new JScrollPane( log ) );
      log.setEditable( false );
}
```

Figure 4.6: Constructing the interface for an SMTP client

tact, `smtp.cs.williams.edu`, and the standard port through which an SMTP server can be contacted, port 25. In this program, all the code that involves accessing the network will fall in the `buttonClicked` method that is discussed in the following sections.

4.2.2 Making Connections

To communicate with another computer using TCP, a program must establish a connection with a program on the other computer. This can be accomplished by constructing a `NetConnection`.

We have seen many examples of constructions in the preceding chapters. We know that by including an expression of the form

```
new JTextArea( ... )
```

in a program we can cause the computer to create a text area that we will eventually be able to see on the screen.

We can similarly cause a computer to create a network connection by executing an expression of the form

```
new NetConnection( ... )
```

The difference is that constructing a `NetConnection` doesn't produce anything we can ever see on the screen. From the program's point of view, however, it does not really matter whether an object can be seen or not. Our programs never see text areas either. Once a text area is constructed, however, statements within the program can apply methods like `getText` and `setText` to the text area to display information for the program's user or to access information provided by the user. Similarly, once a `NetConnection` is constructed, we can apply methods to the `NetConnection` to send information to the computer at the other end of the connection or to access information sent by the program at the other end of the connection. Of course, to apply methods to a `NetConnection` we must associate a name with the object constructed. Therefore, we will typically include such constructions in assignment statements or in a declaration initializer. For example, if we decided to use the name `connection` to refer to a `NetConnection` we might use the declaration

```
NetConnection connection = new NetConnection( ... );
```

to create the connection and associate it with its name.

When we construct GUI components, we provide parameters in the constructions that specify details of the object to be constructed like the width of a text field or the label to appear on a button. When we construct a `NetConnection` we need to provide two critical pieces of information as parameter values: the name or address of the machine with which we wish to communicate and the port number associated with the program to which we are connecting. The first value will be provided as a `String` specifying either a domain name or an IP address. The second can either be a `String` or an `int` value. Therefore, we could construct a connection to the Yahoo web server using the construction

```
new NetConnection( "www.yahoo.com", 80 )
```

Alternately, we can use a `String` value to describe the port to use as in

```
new NetConnection( "www.yahoo.com", "80" )
```

or, assuming that 209.73.186.238 is the IP address for the machine named www.yahoo.com, we could type

```
new NetConnection( "209.73.186.238", 80 )
```

In the SMTP client program we described in the preceding section, we will want to construct a `NetConnection` to an SMTP server as soon as the user clicks the "Send" button. The name of the server and the SMTP port number (25) are associated with the names `SMTP_SERVER` and `SMTP_PORT` in the program's instance variable declarations as shown in Figure 4.5. Therefore, within the `buttonClicked` method we can create the desired `NetConnection` by including a declaration of the form:

```
NetConnection connection = new NetConnection( SMTP_SERVER, SMTP_PORT );
```

4.2.3 The Ins and Outs of `NetConnections`

Once a connection is established, it is easy to use the `NetConnection` to send and receive lines of text through the network. There is a method named `nextLine` that is used to access each line received through the connection and a method named `println` (pronounced "print line") that is used to send a line of data through the connection.[5] The surprise, however, is that these methods are not applied to the connection itself. They are instead applied to subcomponents of the connection.

In the diagrams shown in Figures 4.1 and 4.2, we depicted TCP connections as double-tipped arrows because information flows back and forth between server and client. It would, however, more accurately reflect the structure of Squint `NetConnections` to instead use pairs of single-tipped arrows to represent each TCP connection as shown in Figure 4.7. Each of these arrows represents a one-way flow of data through the network. Java provides a variety of classes for handling such one-way flows of data called *streams*. A `NetConnection` is basically a pair of such streams. Within a given `NetConnection` the stream that carries data from the program that constructed the `NetConnection` to the remote machine is named `out` (since it carries data out of the program), and the stream through which data is received is named `in`. The `println` method mentioned above is applied on the `out` stream while the `nextLine` method is applied to the `in` stream.

For example, we know that immediately after a program establishes a connection to an SMTP server by evaluating a construction like:

```
NetConnection connection = new NetConnection( SMTP_SERVER, SMTP_PORT );
```

the server will send a line to indicate it is ready that will look something like:

```
220 smtp.cs.williams.edu ESMTP Sendmail 8.13.1/8.13.1; Fri, 30 Jun 2006 10:23:31 -0400 (EDT)
```

The program can access whatever line the server sends by evaluating an expression of the form

```
connection.in.nextLine()
```

[5]The name `println` is a historical artifact lingering from a time when a program that was sending data anywhere was probably sending it to an attached printer rather than to another computer. The name `sendLine` might be more appropriate, but the designers of the Java libraries opted to be faithful to the more traditional name.

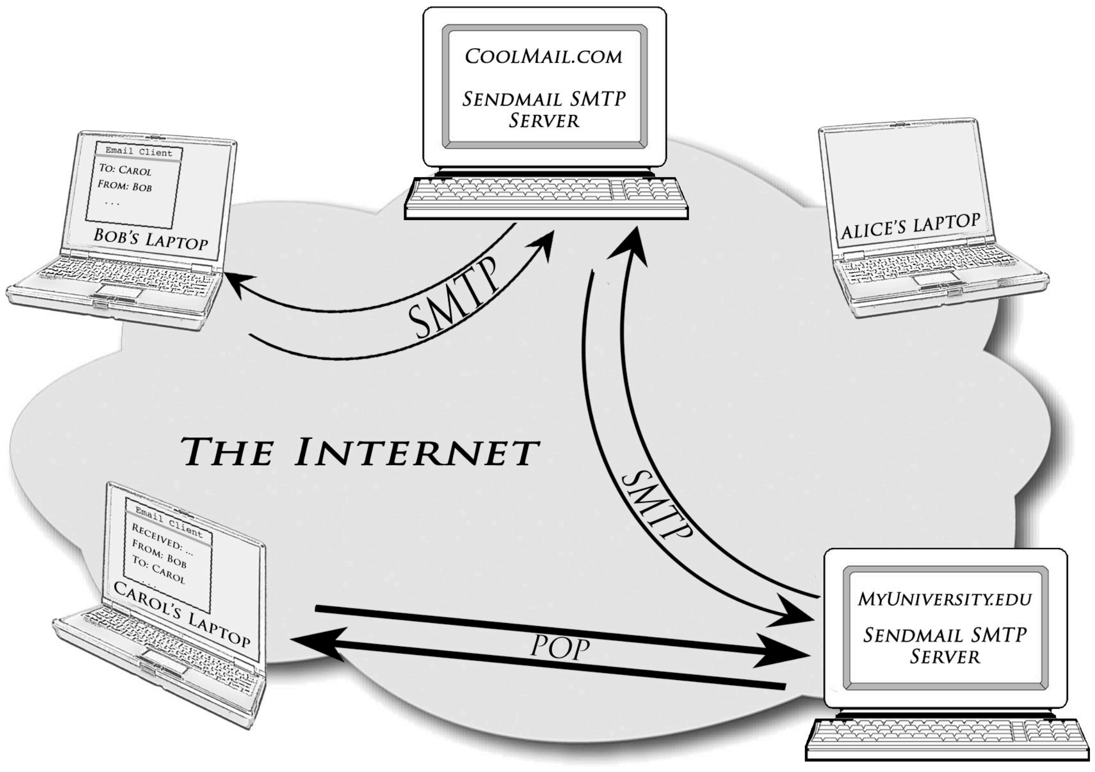


Figure 4.7: Email exchange between two users showing two-way traffic through connections

The sample program we wish to write is supposed to display all lines received form the server, including this introductory line, in the text area named `log`. Therefore, for that program we should include the invocation of `nextLine` within an invocation that will add the text received to that displayed in the log. Such an invocation would be

```
log.append( connection.in.nextLine() + "\n");
```

Similarly, we can send lines by applying the `println` method to the stream identified as `connection.out`. Consider, for example, how we would send the "QUIT" command that must be sent at the end of the interaction with the server. This is one of the simplest SMTP commands. The client program is not expected to send anything after the word "QUIT". As a result, we can send this command by including the single line

```
connection.out.println( "QUIT" );
```

in our `buttonClicked` method. Very similar code can be used to send a `DATA` command.

Sending the other SMTP commands to the server is not much more complicated. For example, the "MAIL" command sent to initiate the transmission of a message is supposed to include the return address that the person using our program entered in the text field named `from`. We can specify the argument to an invocation of `println` by using an expression to describe how to construct the desired value out of simpler parts. Thus, we can use a statement of the form

```
connection.out.println( "MAIL FROM:<" + from.getText() + ">" );
```

to send a `MAIL` command to the server. If the user types the text

```
jrl@cs.williams.edu
```

in the `from` text field, this command will send the line

```
MAIL FROM:<jrl@cs.williams.edu>
```

Probably the most interesting use of `println` that occurs in this program is in the commands used to send the body of the email message to the server. This task is interesting because, unless the user has very little to say, this will require sending several lines to the server rather than just one. The surprise is that a single invocation of `println` can send many lines.

If we execute the commands

```
connection.out.println( message.getText() );
connection.out.println( "." );
```

our program will send all the lines of text that the user has entered in the text area named `message` to the server followed by the period on a line by itself. (Recall that this final period is required by the rules of SMTP to indicate the end of the message body.) Obviously, the name `println` is a bit misleading! This method is not limited to sending a single line. It can send many lines.

When we discussed text areas we saw that you could force text to appear on separate lines by including the special symbol `\n` in a string. This is because `\n` corresponds to a special ASCII code that represents the end of a line to a text area. `println` is named `println` instead of just `print` because it inserts a similar code after the end of whatever data it sends. This ensures that the receiver will identify the end of the data as the end of a line. There is, in fact, another method named `print` that just sends the data specified without adding any codes to indicate the end of a line. If, however, we replaced the code above with the statements

```
connection.out.print( message.getText() );
connection.out.println( "." );
```

our program would malfunction in certain situations. If the user did not press the return key after the last line entered in the text area, there would be no end of line code between the end of the user's message and the period sent by the second statement. To the server, the period would appear at the end of the last line the user typed rather than on a line by itself. As a result, the server would not realize the message was complete.

Many of the Internet's text-based protocols require that commands be sent on separate lines, so it will be critical that you use `println` rather than `print`.

According to the SMTP protocol specification, our client program should include the name of the machine on which it is running in the `HELO` command it sends to the server. To make it easy to obtain this information, there is a `getHostName` method associated with the `GUIManager` class. Within a class that extends `GUIManager`, an expression of the form

```
this.getHostName()
```

produces a string containing the name of the machine running the program. This makes it possible to use the statement

```
connection.out.println( "HELO " + this.getHostName() );
```

to send the `HELO` command.

```
// Send a message when the button is clicked
public void buttonClicked(  ) {
    // Establish a NetConnection and say hello to the server
    NetConnection connection = new NetConnection( SMTP_SERVER, SMTP_PORT );
    log.append( connection.in.nextLine() + "\n" );
    connection.out.println( "HELO " + this.getHostName() );
    log.append( connection.in.nextLine() + "\n" );

    // Send the TO and FROM addresses
    connection.out.println( "MAIL FROM: <" + from.getText() +">" );
    log.append( connection.in.nextLine() + "\n" );
    connection.out.println( "RCPT TO: <" + to.getText() +">" );
    log.append( connection.in.nextLine() + "\n" );

    // Send the message body
    connection.out.println( "DATA" );
    log.append( connection.in.nextLine() + "\n" );
    connection.out.println( message.getText() );
    connection.out.println( "." );
    log.append( connection.in.nextLine() + "\n" );

    // Terminate the SMTP session
    connection.out.println( "QUIT" );
    log.append( connection.in.nextLine() + "\n" );

    connection.close();
}
```

Figure 4.8: The `buttonClicked` method of a simple SMTP client

4.2.4 A Closing Note

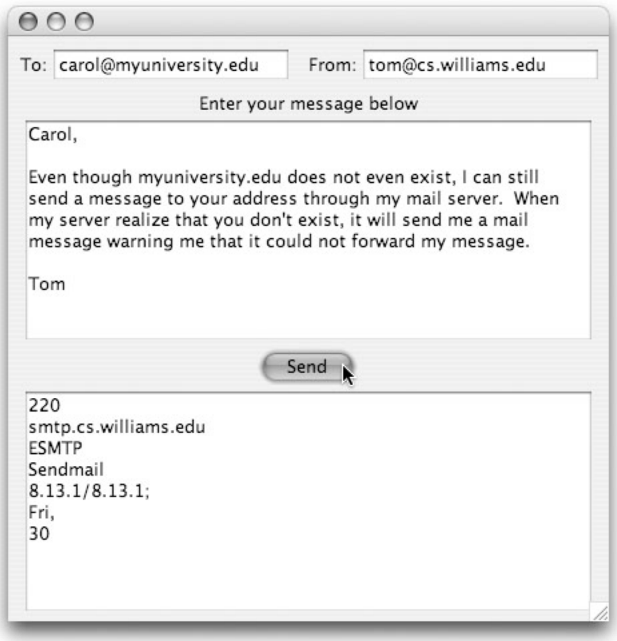
Just as saying "Goodbye" is not the same as hanging up the phone, sending a `QUIT` command to an SMTP server is not the same as ending a TCP connection. As a result, after the `QUIT` command is sent and the server's reply is received, our program's `buttonClicked` method still has to invoke a method to end the TCP connection. This is done by applying the `close` method to the connection itself (rather than to its `in` or `out` streams) using a statement like

```
connection.close();
```

With this detail, we can now present the complete code for our SMTP client's `buttonClicked` method. This code can be found in Figure 4.8.

4.2.5 What's Next

Just as the `out` stream associated with a `NetConnection` provides both a method named `println` and a method named `print`, there is a method named `next` that can be applied to the `in` stream



```
Carol,

Even though myuniversity.edu does not even exist, I can still
send a message to your address through my mail server.  When
my server realize that you don't exist, it will send me a mail
message warning me that it could not forward my message.

Tom
```

```
220
smtp.cs.williams.edu
ESMTP
Sendmail
8.13.1/8.13.1;
Fri,
30
```

Figure 4.9: SMTP client displaying next items rather than next lines

in addition to the `nextLine` method. While the `nextLine` method produces the next line of text received through the connection, the `next` method is defined to return the next *token* received. A token is just a sequence of symbols separated from other symbols by blanks, tabs, or by the end of a line. Informally, a token is just a word. For example, if we had used the expression

```
connection.in.next()
```

in place of all seven occurrences of the expression

```
connection.in.nextLine()
```

in the `buttonClicked` method shown in Figure 4.8, then the output displayed in the log of messages received from the server would appear as shown in Figure 4.9. If you look back at the output produced by the original version of the program, as shown in Figure 4.4, you will see that the first line sent from the server:

```
220 smtp.cs.williams.edu ESMTP Sendmail 8.13.1/8.13.1; Fri, 30 ...
```

has been broken up into individual "words". Each time the `next` method is invoked it returns the next word sent from the server.

In fact, there are many other methods provided by the `in` stream that make it possible to interpret the tokens received in specific ways. For example, if immediately after creating the `NetConnection` we evaluate the expression

```
connection.nextInt()
```

```
// Send a message when the button is clicked
public void buttonClicked(  ) {
    // Establish a NetConnection and introduce yourself
    NetConnection connection = new NetConnection( SMTP_SERVER, SMTP_PORT );
    connection.out.println( "HELO " + this.getHostName() );

    // Send the to and from addresses
    connection.out.println( "MAIL FROM: <" + from.getText() +">" );
    connection.out.println( "RCPT TO: <" + to.getText() +">" );

    // Send the message body
    connection.out.println( "DATA" );
    connection.out.println( message.getText() );
    connection.out.println( "." );

    // Terminate the SMTP session
    connection.out.println( "QUIT" );

    // Display the responses from the server
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );

    connection.close();
}
```

Figure 4.10: The `buttonClicked` method of a simple SMTP client

the value returned will still be 220, but it will be returned as an `int` rather than a `String`. This would make it possible for the program to perform arithmetic operations on the value.

One interesting thing to note about the way in which the `next` method behaves, is that all of the words displayed in Figure 4.9 come from the first line sent by the server. Even though each invocation of `next` comes immediately after a line that sends a request to the server, the system does not assume that it should return a token from the server's response to that request. Instead it simply treats all the data sent by the server as a long sequence of words and each invocation of `next` returns the next word from this sequence.

We emphasize this because the `nextLine` method behaves in a similar way. It treats the data received from the server as a sequence of lines. Each invocation of `nextLine` produces the first line from the sequence that has not previously been accessed through an invocation of `nextLine`.

For example, suppose that we revise the code of the `buttonClicked` method as shown in Figure 4.10. Here, rather than using `nextLine` to access each response sent by the server immediately after our program sends a request, we group all the invocations of nextLine at the end of the method. Somewhat surprisingly, this version of the program will produce the same results as the version shown in Figure 4.8. Even though the first invocation of nextLine appears immediately after the statement that sends the QUIT command to the server, the system will return the first line sent by the server rather than the server's response to the QUIT command as the result of this invocation. Each of the other six consecutive invocations of nextLine shown in this code will return the next member of the sequence of lines sent by the server.

While we can move all of the invocations of nextLine to the end of this method without changing the result the method produces, it is worth noting that moving invocations of nextLine earlier in the method will cause problems. Suppose, for example, that we interchange the third and fourth lines of the original buttonClicked method so that the first four lines of the method are:

```
NetConnection connection = new NetConnection( SMTP_SERVER, SMTP_PORT );
log.append( connection.in.nextLine() + "\n" );
log.append( connection.in.nextLine() + "\n" );
connection.out.println( "HELO " + this.getHostName() );
```

The server will send one line to the client as soon as the connection is established, but it won't send a second line until after it receives the HELO command. The computer, however doesn't understand this. Therefore, with the revised code, when the third line is executed, and the client program attempts to access the line sent, the computer will realize that no line has arrived yet and decide that it should wait for one to arrive. If allowed to do so, it would wait forever! No second line will arrive until it sends the HELO command, but it believes it must wait until the second line arrives before it sends any HELO command. The program will freeze up and the user will have to resort to whatever mechanism the operating system or IDE provides to "force quit" the program.

4.2.6 Network Event Notification

Under SMTP and many other protocols, the server only sends packets as immediate responses to requests received from the client. When writing a client that uses such a protocol, it is safe to simply perform nextLine invocations after sending requests. There are other protocols, however, where the server may send a message to the client at any time, independent of whether the client has recently sent a request. The IM protocol is a good example of this. The server will send the client a message whenever any of the user's buddies send a message to the user. While many such messages arrive in response to a message the user sent, it is also common to receive unsolicited messages saying "Hi" (or something more important).

Consider, then, how one could write an IM client. Recall that when a program invokes nextLine, it waits patiently (i.e., it does not execute any other instructions) until a message from the server is available. Therefore, we can't handle unexpected messages from a server by just constantly doing a nextLine. If that is what our IM client did, it would not do anything else.

The solution is to treat the arrival of messages from the server like button clicks and other events involving GUI components. We have seen that for many GUI components, we can write special methods like buttonClicked, textEntered, and menuItemSelected that contain the code that should be executed when an appropriate event occurs. A similar mechanism is available for use with network connections. We can place code in a method named dataAvailable if we want that code executed when messages are received through a network connection. The header for such a method looks like

```
public void dataAvailable( NetConnection whichConnection ) {
```

The connection through which data becomes available will be associated with the formal parameter name supplied. The code placed in a `dataAvailable` method should include an invocation of `nextLine`, `next`, or one of several other similar methods.

Unlike the event-handling methods for GUI components, the system does not automatically execute the code in a `dataAvailable` method if we define one. In addition to defining the method, we have to explicitly tell the `NetConnection` to notify our program when interesting events occur. We do this by executing an invocation of the form

```
someNetConnection.addMessageListener( this );
```

(Recall that the word `this` is Java's way of letting us talk about our own program).

If a program uses `addMessageListener` to request notification when interesting events happen to a `NetConnection` there is another special event-handling method that can be defined. The header for this method looks like:

```
public void connectionClosed( NetConnection whichConnection ) {
```

The code in this method will be executed when the server terminates the connection.

As a simple example of the use of such event-handling methods in a network application, we present another replacement for the `buttonClicked` method of our SMTP client in Figure 4.11. This time, we have not simply replaced one version of `buttonClicked` with another version. Instead, we have added definitions for two additional methods, `dataAvailable` and `connectionClosed`.

We have changed the code in three ways:

1. We have added the line

   ```
   connection.addMessageListener( this );
   ```

   after the line that constructs the new connection. This informs the `NetConnection` that we want the `dataAvailable` and `connectionClosed` methods to be invoked when new data arrives or when the server terminates the connection.

2. We have removed all seven lines of the form

   ```
   log.append( connection.in.nextLine() + "\n" );
   ```

   from the `buttonClicked` method. Instead, we have placed a single line of the form

   ```
   log.append( incomingConnection.in.nextLine() + "\n" );
   ```

   in the body of the `dataAvailable` method. This line will be executed exactly seven times because the `dataAvailable` method will be invoked each time a new line becomes available.

3. We have removed the line

   ```
   connection.close();
   ```

```
// Send a message when the button is clicked
public void buttonClicked(  ) {
    // Establish a NetConnection and introduce yourself
    NetConnection connection = new NetConnection( SMTP_SERVER, SMTP_PORT );
    connection.addMessageListener( this );
    connection.out.println( "HELO " + this.getHostName() );

    // Send the to and from addresses
    connection.out.println( "MAIL FROM: <" + from.getText() +">" );
    connection.out.println( "RCPT TO: <" + to.getText() +">" );

    // Send the message body
    connection.out.println( "DATA" );
    connection.out.println( message.getText() );
    connection.out.println( "." );

    // Terminate the SMTP session
    connection.out.println( "QUIT" );
}

// Display any messages from server in the log area
public void dataAvailable( NetConnection incomingConnection ) {
   log.append( incomingConnection.in.nextLine() + "\n" );
}

// Close the connection right after the server disconnects
public void connectionClosed( NetConnection closingConnection ) {
    closingConnection.close();
}
```

Figure 4.11: Using `NetConnection` event handling in an SMTP client

from the `buttonClicked` method and placed a similar line in a new `connectionClosed` method. If we had left the invocation of `close` in `buttonClicked`, the program would not always display all of the lines sent by the server. The connection would be closed as soon as the client sent the `QUIT` command. This would definitely mean that it was closed before the server's response to this command. Once the `NetConnection` has been closed, the program will ignore any messages from the server. The `dataAvailable` method will not be invoked when such messages arrive. By placing the `close` in the `connectionClosed` method, we don't close the connection until we know that we have received everything the server will send because we know that the server has closed its end of the connection.

Note that we use the parameter names `incommingConnection` and `closedConnection` to refer to the `NetConnection` within the two network event handling methods. Alternately, we could have changed the program so that `connection` was declared as an instance variable rather than as a local variable within `buttonClicked` and then used this variable in all three methods.

4.2.7 Summary of `NetConnection` constructions and methods

A new `NetConnection` can be created by evaluating a construction of the form

```
new NetConnection( host-name, port-number )
```

where *host-name* is an expression that describes a string that is either the domain name or the IP address of the machine on which the server you would like to contact is running, and *port-number* is an expression that evaluates to the number of the port used to contact the server program. The port number can be described using either an `int` or a `String`. For example, the construction

```
new NetConnection( "www.google.com", 80 )
```

would create a connection to the web server port on the machine `www.google.com`.

There are four basic methods used to send and receive data through a `NetConnection`. These methods are associated with data streams named `in` and `out` that are associated with the connection.

`someConnection.in.nextLine()`	Each invocation of `nextLine` returns a `String` containing one line of text received from the server. As long as the server has not closed the connection, your program will wait until a line of text is received from the server. If the remote server has terminated the connection, your program will terminate with an error.
`someConnection.in.next()`	Each invocation of `next` returns a `String` containing one token/word of text received from the server.
`someConnection.out.println( someString );`	An invocation of `println` causes the program to send the contents of its argument to the server followed by an end of line indicator.

```
someConnection.print( someString );
```

An invocation of `print` causes the program to send the contents of its argument to the server.

In addition, the `NetConnection` class provide two methods to control the connection itself.

```
someConnection.close();
```

The `close` method should be invoked to terminate the connection to the server once no more messages will be sent and all expected messages have been received.

```
someConnection.addMessageListener( someGUIManager );
```

The `addMessageListener` method should be used to inform the `NetConnection` that the program has defined methods named `dataAvailable` and `connectionClosed` and that these methods should be invoked when new messages are received through the connection or when the connection is closed.

4.3 Summary

In this chapter we have introduced two closely related topics: the techniques used to write Java programs that send and receive network messages and the nature of the conventions computers must follow to communicate effectively.

We presented the general notion of a *communications protocol*, a specification describing rules computers must follow while communicating. We explained that the Internet relies on many protocols that are each specialized to accomplish a particular application. We also explained that many of these application protocols depend on a protocol named TCP that specifies aspects of communications that are common to many application protocols. We described the addresses used to identify computers and programs when using TCP and introduced the notion of a connection.

We also explored a new library class named `NetConnection` that can be used to write programs based on TCP. We explained that although a `NetConnection` is designed for 2-way communications, it is actually composed of two objects called *streams* that support 1-way communications. We showed how to send messages to a remote computer using one of these streams and how to receive messages sent to our program using the other.

Finally, to clarify the connection between our abstract introduction to protocols and our concrete introduction to `NetConnections`, we showed how `NetConnections` could be used to implement a program that followed one of the Internet's oldest, but still most important protocols, the mail delivery protocol SMTP.