Chapter 18. Networking Basics

The next four chapters are about networked games.

This chapter runs through networking fundamentals (e.g. what are the client/server and peer-to-peer models?) and explains basic network programming with sockets, URLs, and servlets

Chapter 19 is about online chat, the "hello world" of network programming. We look at *three* chat variants: one using a client/server model, one employing multicasting, and chatting with servlets.

Chapter 20 describes a networked version of the FourByFour application, a turn-based game which is a demo in the Java 3D distribution.

Chapter 21 revisits the Tour3D application of chapter ?? (the robot walking about a checkboard landscape), and adds networking to allow multiple users to share the world. We also discuss some of the advanced issues concerning networked virtual environments (NVEs), of which NetTour3D is a simple example.

The structure of this chapter in more detail:

- descriptions of the core attributes of network communication;
- explanations of IP, UDP, TCP, network addresses, and sockets;
- overviews of the client/server and peer-to-peer models;
- four small client/server applications (sequential, threaded, nonblocking multiplexing, and UDP multiplexing);
- a small peer-to-peer application (UDP multicasting);
- the problem of firewalls, leading to URLs and servlets for HTTP tunneling;
- a brief word about some other kinds of Java networking (RMI, Jini, JavaSpaces, the Java Shared Data Toolkit).

1. The Elements of Network Communication

Network communication is often characterized by five attributes:

1) Topology
2) Bandwidth
3) Latency
4) Reliability
5) Protocol.

1.1. Topology

Topology is the interconnection 'shape' of machines linked over a network.

© Andrew Davison. 2003

Popular shapes are the ring, star, and all-to-all, illustrated in Figure 1.

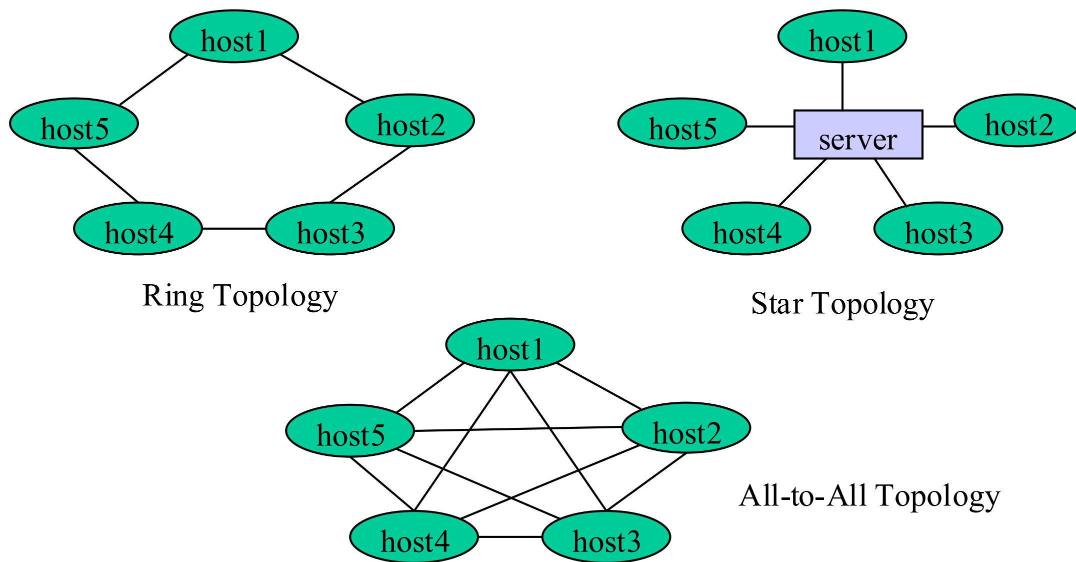

Figure 1. Some Network Topologies.

Choosing the best topology for an application is a complicated matter, depending on many network characteristics, such as bandwidth and latency, and the particular communication patterns inherent in the system.

The star topology, present in client/server applications, is the most popular, while the all-to-all topology appears (in modified forms) in peer-to-peer systems.

1.2. Bandwidth

Bandwidth is the rate at which the network can deliver data from a sender to the destination host. Modems can typically deliver 14,400-56,000 bits/second (14.4 – 56 Kbps), Ethernet can attain 10-100 Mbps (million bits/second), with newer technologies offering 1 Gbps (gigabits/second). Fiber optic cable exhibits speeds of up to 10 Gbps.

Dial-up usage is declining in the US, but 66.2% of users still connect with modems. According to March 2003 figures from Nielsen/NetRatings (http://www.nielsen-netratings.com/), 53.26% use 56 Kbps modems, 9.79% have 28/33.3 Kbps, and 3.17% are stuck with 14.4 Kbps modems.

A study by MetaFacts (http://www.metafacts.com/) showed that the vast majority of the 25,000 people they surveyed still utilize low-speed internet access; only 9% have DSL and 16% use cable modems. However, almost half of the narrowband users were considering an upgrade and, if current rates of uptake continue, broadband will be more widespread than dial-up by 2005.

More details can be found in the article:

> *"Your Speed May Vary"*
> Robyn Greenspan, April 25th, 2003

 © Andrew Davison. 2003

http://cyberatlas.internet.com/markets/
broadband/article/0,,10099_2196961,00.html

These statistics indicate that 56 Kbps is a realistic bandwidth estimate for users, at least for the next year or two.

Bandwidth restricts the number of players in a game: too many participants will swamp the network with data traffic. However, the nature of the extra load depends on topology. For example, a message sent around a ring will have to travel through many links, occupying bandwidth for longer. A message travelling through a star topology only requires two links to go from any sender to any receiver, while a message in an all-to-all topology goes directly to its destination.

A knowledge of the available bandwidth allows us to estimate upper limits for the amount of data that can be transferred per each frame rendered in a game, and to suggest a likely maximum for the number of users.

A 56 Kbps modem, means that 7000 bytes can be transferred per second. We assume that the game updates at 30 frames/sec, and each player sends and receives data from all the other players during each frame update. The total amount of data that can be transferred each frame is $7000/30 \approx 233$ bytes/frame. This is about 117 bytes for output, 117 bytes for input

If there are n players, then each player will send n-1 output messages and receive n-1 input messages at each frame update. As n increases, the 117 bytes limit will quickly be reached.

We can estimate the maximum number of users by starting with a lower bound for the amount of data that must be transferred during each frame update. For instance, if the lower bound is 20 bytes, and there is 117 bytes available for output, then only a maximum of about 6 messages can be sent out (117/20), which means seven players altogether in the game!

These kinds of ballpark figures explain why games programmers try very hard to avoid broadcast models linked to frame rate, and why transmitted data is kept as small as possible. Some techniques for reducing data transmission in multiplayer games are discussed in chapter 21, when a networked version of the Tour3D application is developed.

A network protocol solution is to move from a broadcast model to *multicasting*. Multicasting is a form of subscription-based message distribution: a player sends a single message to a multicast group, where it is automatically distributed to all the other players who are currently subscribed to that group. The saving is in the number of messages sent out: one instead of n-1.

A similar saving can be made in a star topology: a player sends a single message to the server, which then sends copies to the other players. This is a software solution, dependent on the server's implementation, which allows for further server-side optimizations of the communication protocols.

Multicasting and client/server systems are described more fully below.

 © Andrew Davison. 2003

1.3. Latency

Latency is the amount of time required to transfer a bit of data from one place to another (typically from one player's machine to another). In a MMORPG (Massively Multiplayer Online Role-Playing Game), latency exhibits itself as the delays when two players are interacting (e.g. shooting at each other); the goal is to reduce latency by as much as possible.

Most latency can be accounted for by the modems involved in the network: typically each one adds 30-40 ms to the latency. A client/server system may involve four modems between the sending of a message and its reception by another user: the sender's modem, the modem inbound to the server, the modem outbound from the server, and the receiver's modem, resulting in a total latency of perhaps 160 ms.

Another major contributor, especially over the internet, are routers, which may easily add hundreds of milliseconds. This is due to their caching of data before forwarding it, and delays while a router decides where to send a message. Routers may drop data when overloaded, which can introduce penalties in the 400-500 ms range for protocols like TCP which detect lost data and resend it.

Another issue is the speed of light – games would certainly benefit if it was faster! A message sent from the east to west coast of the US must take about 20 ms to get there. In general, about 8 ms of travel time are added for each time zone that a message passes through.

Designers often incorporate tolerances of 250 ms into the communication models used in games. A key observation is that most games do not require complete synchronization between all the players all the time. Synchronization is usually important only for small groups of players (e.g. those in close proximity inside the game world), and even then only at certain moments.

This relaxing of the synchronization requirement opens the door to various approaches based on temporarily 'guessing' information about other players. For instance, the game running on the user's machine estimates other players' current positions and velocities, and corrects them when the correct data arrives over the network.

Another implementation strategy is to decouple general game play from the networking side of the application. A separate thread (or process) waits for incoming data, so the rest of the application can continue unaffected.

Latency is essentially a WAN or internet issue; applications running over a LAN rarely need to consider latency, which may total less than 10 ms.

1.4. Reliability

Increased network reliability may increase latency time. For example, the possibility that a message may be lost while travelling over the network, led to the development of the TCP protocol which deals with data loss by resending. The disadvantage is that the actual arrival time of a message can be increased significantly, so impacting latency.

A desire to measure reliability has led to more complex checking of data based on cyclic redundancy checks (CRCs), which add further overheads.

© Andrew Davison. 2003

An alternative view is to live with a certain degree of unreliability, reducing overheads as a consequence. Many forms of internet-based multimedia, such as streaming audio and video, take this line because losing small amounts of data only means a momentarily loss in sound or picture quality. The rapid transmission of the data is a more valued attribute.

1.5. Protocol

A protocol is a notation for specifying the communication patterns between the components of a networked application.

Different protocols support different capabilities, with the choice of protocol depending on many factors, such as the type of data being transmitted, the number of destinations, and the required reliability.

Most gaming application uses multiple protocols: one for data, another for control, and perhaps others specialised for particular types of data such as audio or video.

Different protocols exist for different levels of communication, which are defined in the ISO OSI model (see Figure 2).

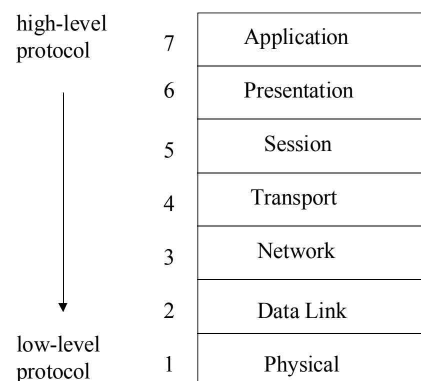

Figure 2. The ISO OSI Model

A *protocol suite* is a collection of related protocols for different layers of the OSI model. The most popular is TCP/IP, originally developed by DARPA in the early 1980's. Its wide popularity stems from it being implemented on everything from PCs to supercomputers, not being vendor specific, and its suitability for all kinds of network, from LANs up to the internet.

 © Andrew Davison. 2003

This isn't a book about data communications, so we'll only consider TCP/IP within a simplified version of the OSI model, limited to just four layers. Communication between two networked systems can then be viewed as in Figure 3.

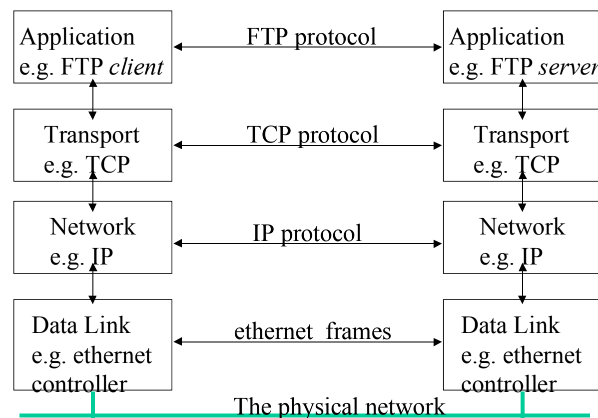

Figure 3. Communication between two Systems.

Figure 3 shows that application-level protocols, which include FTP, HTTP (the Web protocol), and SNMP (the e-mail protocol), are implemented on top of lower-level protocols (TCP or UDP).

At a particular layer, the data appears to cross directly to the equivalent layer on the other system. In fact, data is passed down through the layers, across the physical network, and back up through the layers of the other system. As each layer is descended, the user's data is wrapped inside more header (and footer) information. The headers (and footers) are removed as the data rises through the layers on the other system.

The data link layer corresponds to the OSI physical and data layers. Frames are delivered between two directly connected machines.

The network layer delivers datagrams between any two machines on the internet, which may be separated by intervening gateways/routers. Each datagram is routed independently, which means that two datagrams sent from the same source may travel along different paths, and may arrive in a different order from the way they were sent. Also, since a gateway/router has limited memory, it may discard datagrams if too many arrive at once. A node or link in the network may fail, losing packets. For these reasons, the IP protocol makes no guarantees that a datagram will arrive at its destination.

A datagram has size constraints, which often forces a single piece of user data to be divided into multiple datagrams. The data arriving at the receiver may therefore have missing pieces, and parts in the wrong order.

A datagram is sent to an IP address, which represents a machine's address as a 32-bit integer. Programs usually employ IP addresses in dotted-decimal form (e.g. 172.30.0.5) or as a dotted-name (e.g. fivedots.coe.psu.ac.th). Dotted-name to IP address translation is carried out using a combination of local machine configuration information and network naming services such as the Domain Name System (DNS).

 © Andrew Davison. 2003

The IP protocol is currently in transition from version 4 to version 6 (IPv6), which supports 128-bit addresses, a simpler header, multicasting, authentication, and security elements. Java supports both IPv4 and IPv6 through its InetAddress class.

The transport layer delivers datagrams between transport end-points (machine ports) for any two machines on the Internet. An application's location is specified by the IP address of its host, and the number of the port where it is 'listening'. A port number is a 16-bit integer.

The TCP/IP transport layer protocols are:

- UDP: the User Datagram Protocol
- TCP: the Transmission Control Protocol

UDP is a *connectionless* transport service: a user message is split into datagrams and each datagram is sent along an available path to its destination. There is no (expensive) long-term, dedicated link created between the two systems. UDP inherits the drawbacks of the IP protocol: datagrams may arrive in any order, and there is no guarantee that a datagram will arrive. UDP is often compared to the postal service.

TCP is a *connection-oriented* transport service. From the user's point of view, a long-term, dedicated link is set up between the sender and receiver, and two-way stream-based communication then becomes possible. For this reason, TCP is often compared to the telephone service.

However, the 'dedicated' link is implemented on top of IP, and so packets of information are still being sent, with the chance of reordering and loss. Consequently, TCP employs sophisticated error-checking internally to ensure that its component TCP datagrams arrive in the order they were sent, and that none are lost. This overhead may be too severe for gaming applications which value low latency.

Both UDP and TCP use a socket data structure to represent an end-point in the communication. For UDP, the socket is something like a mailbox, while a TCP socket is more like a telephone.

Java supports both TCP and UDP. A programmer uses the Socket class for creating a sender's TCP socket, and the ServerSocket class for the recipient. Java offers the DatagramSocket for both sides of UDP communication, and a DatagramPacket class for creating UDP packets.

© Andrew Davison. 2003

2. The Client/Server Model

The client/server model is the most common networking architecture employed in distributed applications. A server is a program (or collection of cooperating programs) which provides services and/or manages resources on the behalf of other programs, known as its clients. Figure 4 shows a typical client/server environment.

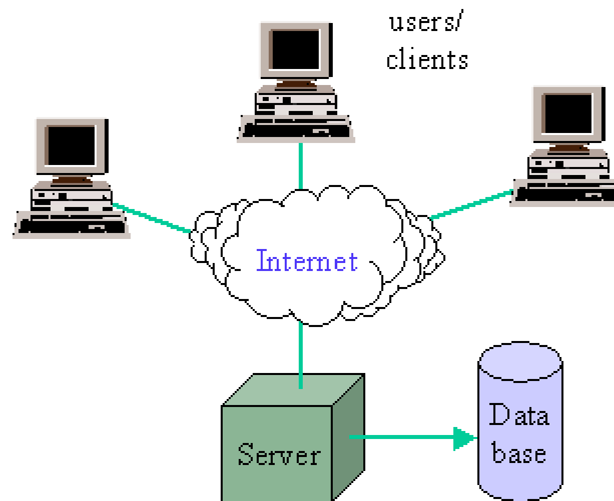

Figure 3. A Simple Client/Server

A key advantage of the client/server approach is the ability for the server to *control* its clients, by localising important processing and data within itself. For example, in online games, decisions about the outcome of a player's actions (e.g. the shooting of a monster) will be made by the server. The alternative is to delegate this to the client's application which may lead to abuse by hackers.

A central location for client data and processing makes it easier to monitor users, administer them, and *charge* them for game playing. These are significant benefits for commercial multiplayer gaming.

The server is an arbiter for conflicting client requests. For instance, it is impossible for two clients to update the same data state at the same time.

Placing state information in one place makes changing it straightforward, and avoids inconsistencies arising when updating multiple copies stored on clients.

Concentrating processing on the server-side permits the client side to run on a less powerful machine, an important consideration for network applications on PDAs or phones.

Concentrating most of the application in the server makes it is easier to maintain and update, compared to trying to upgrade code spread over a large, possibly non-technical, user base.

The main disadvantage of a 'fat' server is the potential for it to become a bottleneck, overloaded by too many clients, increasing latency to unacceptable levels. Excessive numbers of clients may also overload the server's data storage capabilities.

Another significant issue is reliability – if the server fails then everyone will be affected. Almost as bad as failure is the (temporary) non-availability of the server,

 © Andrew Davison. 2003

either because it is currently dealing with too many users, or because of attack by hackers, or because it has been taken offline for servicing or upgrades.

These concerns have led to the widespread use of multiple servers, sometimes specialised for different elements of client processing, or acting as duplicates, ready to stand-in for a server which fails. Different servers may be in different geographical locations, catering only to clients in those areas, so improving latency times.

Many MMORPGs map areas of their virtual world to different physical servers (e.g. Ultima Online does so). When a client moves across a boundary in the world, he/she is switched to a different server. This also acts as a form of load balancing, although its success depends on the different zones having roughly the same levels of popularity.

High-powered servers are not cheap, especially ones with complex database management, transaction processing, security, and reliability capabilities.

A successful application (game) will require an expensive, high-speed internet connection. A predicted trend is that gaming companies will start to offer their own backbone networks, giving them greater control over bandwidth, latency, and the specification of firewall access. This latter point will make it easier for applications to use multiple protocols (e.g UDP for data, TCP for commands) without worrying about clients being unable to create the necessary communication links due to firewall restrictions.

3. The Peer-to-Peer Model

Peer-to-Peer (P2P) encourages ordinary people to share their resources with others; resources include hard disk storage, CPU time, and files (audio, video). This is different from today's Web/internet where business/government/university servers present information, and the rest of us read it. The difference is illustrated by Figure 4.

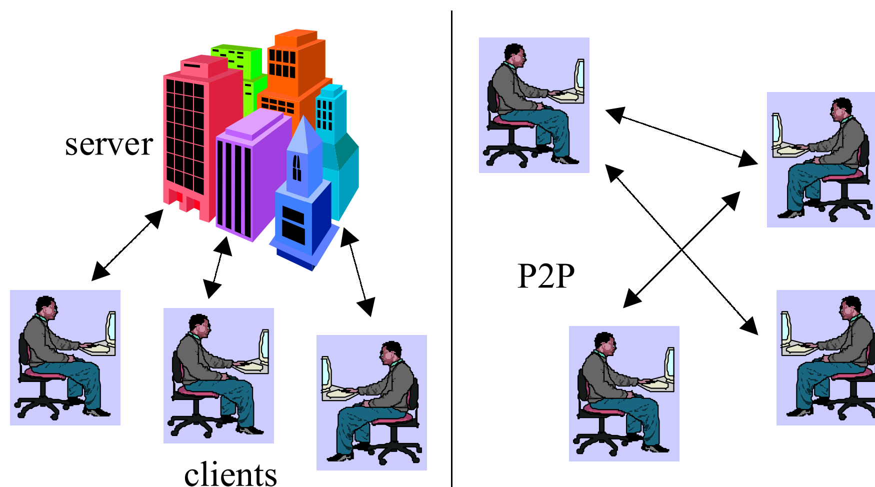

Figure 4. Client/Server Versus P2P.

P2P isn't a new idea: the early internet (at that time, the Arpanet) was designed to be P2P so that US universities and government installations could share computing resources.

© Andrew Davison. 2003

Many of the killer apps of the 1980's, such as telnet and e-mail, were client/server based but their usage patterns were symmetric – most machines ran both clients and servers. Usenet is a P2P application: it employs a decentralized file sharing model for distributing news, but an unofficial backbone has developed over time, based around server inequalities in storage capacity, processing speeds, and network connectivity.

Things started to change with the growth of the Web. It is a client/server model like most of the earlier applications, but differences lie in its political and social components.

Most users only browse the Web, they don't run their own Web servers. This is due to their lack of technical knowledge, the difficulty of setting up a server, and because commercial ISPs do not allow them. Many ISPs only allocate dynamic addresses to clients making the running of servers impossible, and impose firewall restrictions which only permit Web page access. Broadband connections, such as cable modems, offer asymmetric bandwidth which makes the serving of material slow.

The restrictions on users (e.g. firewalls) are quite recent, triggered by the lack of accountability mechanisms in the internet protocol: there are no technological barriers to users sending spam, and attacking machines. The original designers made the fatal assumption that users are responsible! The problem is sometimes called "the Tragedy of the Commons": a commonly owned resource will be overused until it degrades, due to the users putting self-interest first.

The issue of accountability has led to better support for cryptography in IPv6, and experimental technologies such as micropayments and reputation schemes.

With micropayments, a person wishing to use someone else's resource (e.g. one of their files) must compensate that person in some way. This might be in the form of digital money or another valuable resource. Micropayments have the benefit of solving many forms of hacker attack, such as spam and distributed denial of service, since the hacker must pay an excessive amount in order to 'flood' the network with their datagrams.

A reputation scheme typically requires a respected user to verify the reliability of a new user. This idea is well-known in Java which utilizes encrypted signatures and third party verifiers to make trusted applets.

Many P2P systems are concerned with anonymity, so that external agencies cannot know who is involved in a P2P group, where files are stored, and who has published what. These aims make accountability and trust harder to support.

Part of the drive behind these systems is the case of Napster, which was effectively closed down because of its publication of music files that it hadn't authored. Napster could be targeted because it was a hybrid of P2P and client/server: a Napster server stored details about who was logged on, and the published files. Unfortunately, this is a common situation – most current P2P systems require some server capabilities. For instance, games must validate new players, maintain account information, supply current status information for the game, and notify other users when a player joins or leaves.

Pure P2P has the advantage that there is no central point (no server) whose demise would cause the entire system to stop. This makes P2P resistant to some forms of attack, such as distributed denial of service and legal rulings!

 © Andrew Davison. 2003

The main drawback of P2P is paradoxically its lack of a server, which makes it difficult to control and manage the overall application.With no server, how does a new user discover what is available?

The P2P diagram in Figure 4 suggests that participants use broadcasting to communicate, but this approach soon consumes all the available bandwidth, for reasons outlined earlier. Large scale P2P applications use IP multicasting with UDP packets, which currently relies on the MBone, a virtual overlay installed on the internet to facilitate multicasting. A special pseudo-IP address, called a multicast IP address or class D address, is employed, which can be in the range 224.0.0.0 to 239.255.255.255 (though some addresses are reserved).

Multicasting avoids the potential for loops and packet duplication inherent in broadcasting, since it creates a tree-based communication pattern between the multicast group members. Java supports IP multicasting with UDP.

The Java Media Framework (JMF) is an API extension to Java for handling time-based media, specifically streaming audio and video (see http://java.sun.com/products/java-media/jmf/). The underlying protocol is RTP (the Real-Time Transport Protocol), implemented on top of UDP. Many networked multimedia systems use RTP and multicasting.

JXTA provides core functionality so that developers can build P2P services and applications (see http://www.jxta.org/). The core JXTA layer includes protocols and building blocks to enable key mechanisms for P2P networking. These include discovery, transport (e.g. firewall handling and limited security), and the creation of peers and peer groups.

The JXTA specification is not tied to any particular programming language or platform/device. Its communication model is general enough so that it can be implemented on top of TCP/IP, HTTP, Bluetooth, and many other protocols. Currently JXTA utilizes Java; it was initiated at Sun Microsystems by Bill Joy and Mike Clary. A good site for finding out about P2P is O'Reilly's http://www.openp2p.com/.

4. Client/Server Programming in Java

The four examples in this section have a similar structure: the server maintains a high scores list (similar to the list in the arcade game of chapter ??), and the clients read and add scores to the list. The four variants of this idea are:

1. A client and *sequential server*. The server can only process a single client at a time. The TCP/IP protocol is utilized.
2. The same client, same protocol as (1), but the *server is threaded* enabling it to process multiple clients at the same time. Synchronization issues arise because the high score list may be accessed by multiple threads at the same time.
3. The same client, same protocol as (1) and (2), but Java's NIO is used to implement a *multiplexing server* without the need of threads.

 © Andrew Davison. 2003

4. A *client and server using UDP*. The server exhibits multiplexing due to the self-contained nature of datagram communication.

4.1. TCP Client and Sequential Server

The communications network created by the client and server is shown in Figure 5.

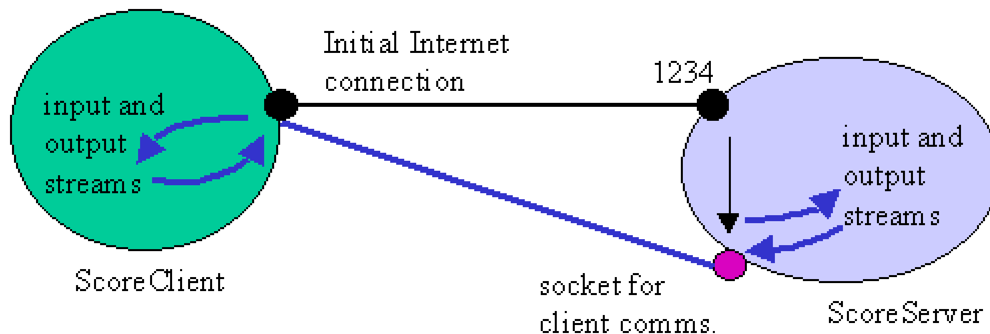

Figure 5. Client and Sequential Server.

The server instantiates a ServerSocket object at port 1234 and waits for a connection from a client by calling accept(). When a connection is established, a new socket is created (some people call this a *rendezvous socket*), which is used for the subsequent communication with the client. Input and output streams are layered on top of the socket, which is possible because of the bi-directional nature of a TCP link. When the client has finished, the rendezvous socket is closed (terminated), and the server waits for another connection.

The client instantiates a Socket object to link to the server at its specified IP address and port. When a connection is obtained, the client layers input and output streams on top of its socket, and commences communication.

A great aid to understanding networked applications, is to understand the protocol employed between clients and the server. In simple examples (like these), this means the message interactions between them.

A client can send the following messages, which all terminate with a newline character. The text after the '//' states how the server responds:

- `get` // the server returns the high score list
- `score name & score &` // the server adds the name/score pair to its list
- `bye` // the server closes the client's connection

A client only receives one kind of message from the server, the high scores list, which is sent in response to a 'get' request. The list is sent as a string terminated with a newline character. The string has the format:

```
HIGH$$ name1 & score1 & .... nameN & scoreN &
```

The server only stores at most ten names and scores, so the string is unlikely to be excessively long.

 © Andrew Davison. 2003

The UML diagrams for the various classes are given in Figure 6. Only the public methods are shown.

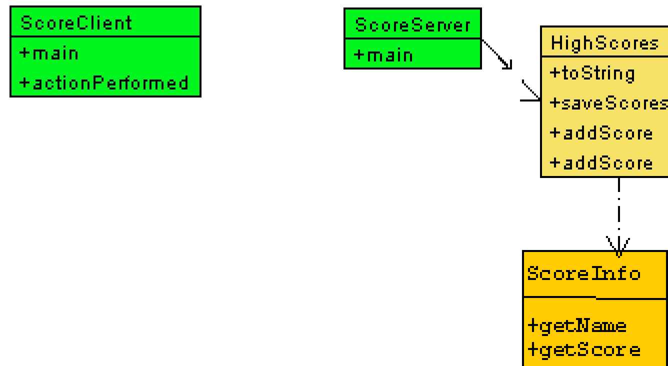

Figure 6. UML Classes for the Client and Sequential Server.

A HighScores object manages the high scores list, with each name/score pair held in a ScoreInfo object.

The ScoreServer Class

The constructor for the ScoreServer creates the ServerSocket object, then enters a loop which waits for a client connection by calling accept(). When accept() returns, the server processes the client, then goes back to waiting for the next connection.

```
public ScoreServer()
{
  hs = new HighScores();
  try {
    ServerSocket serverSock = new ServerSocket(PORT);
    Socket clientSock;
    BufferedReader in;      // i/o for the server
    PrintWriter out;

    while (true) {
      System.out.println("Waiting for a client...");
      clientSock = serverSock.accept();
      System.out.println("Client connection from " +
          clientSock.getInetAddress().getHostAddress() );

      // Get I/O streams from the socket
      in  = new BufferedReader(  new InputStreamReader(
                     clientSock.getInputStream() ) );
      out = new PrintWriter( clientSock.getOutputStream(), true );

      processClient(in, out); // interact with a client

      // Close client connection
      clientSock.close();
      System.out.println("Client connection closed\n");
      hs.saveScores();       // backup scores after client finish
```

 © Andrew Davison. 2003

```
      }
    }
    catch(Exception e)
    {  System.out.println(e);  }
  }  // end of ScoreServer()
```

The server-side socket is created with the ServerSocket class:

```
ServerSocket serverSock = new ServerSocket(PORT);
```

The waiting for a connection is done via a call to accept():

```
Socket clientSock;
     :
clientSock = serverSock.accept();
```

When accept() returns it instantiates the rendezvous socket, clientSock.

clientSock can be used to retrieve client details, such as its IP address and host name.

The input stream is a BufferedReader to allow calls to readLine(). Since client messages end with a newline, this is a convenient way to read messages. The output stream is a PrintWriter, allowing println() to be employed. The stream's creation includes a boolean to switch on autoflushing so there is no delay between printing and actual output to the client.

```
in  = new BufferedReader( new InputStreamReader(
                  clientSock.getInputStream() ) );
out = new PrintWriter( clientSock.getOutputStream(), true );
```

The client is processed by a call to processClient(), which contains the application-specific coding. Almost all the rest of ScoreServer is reusable in different sequential servers.

When processClient() returns, the communication has finished and the client link can be closed:

```
clientSock.close();
```

The call to saveScores() in the HighScores object is a precautionary measure: it saves the high scores list to a file, so data will not be lost if the server crashes.

Most of the code inside the constructor is inside a try-catch block, to handle IO and network exceptions.

processClient() deals with message extraction from the input stream, which is complicated by having to deal with link termination.

The connection may close because of a network fault, which is detected by a null being returned by the read, or be signaled by the client sending a "bye" message. In both cases, the loop in processClient() finishes, passing control back to the ScoreServer constructor.

```
private void processClient(BufferedReader in, PrintWriter out)
{
  String line;
  boolean done = false;
  try {
    while (!done) {
      if((line = in.readLine()) == null)
```

```
          done = true;
        else {
          System.out.println("Client msg: " + line);
          if (line.trim().equals("bye"))
            done = true;
          else
            doRequest(line, out);
        }
      }
    }
    catch(IOException e)
    {  System.out.println(e);  }
  }  // end of processClient()
```

The method uses a try-catch block to deal with possible IO problems.

processClient() does a very common task, and should be portable across various applications. It requires that the termination message ("bye") can be read using readLine().

doRequest() deals with the remaining two kinds of client message: "get" and "score". Most of the work in doRequest() involves the checking of the input message embedded in the request string. The score processing is carried out by the HighScores object.

```
private void doRequest(String line, PrintWriter out)
{
  if (line.trim().toLowerCase().equals("get")) {
    System.out.println("Processing 'get'");
    out.println( hs.toString() );
  }
  else if ((line.length() >= 6) &&     // "score "
      (line.substring(0,5).toLowerCase().equals("score"))) {
    System.out.println("Processing 'score'");
    hs.addScore( line.substring(5) );    // cut the score keyword
  }
  else
    System.out.println("Ignoring input line");
}
```

It is a good idea to include a default else case to deal with unknown messages. In doRequest() the server only reports problems to standard output on the server-side. It may also be advisable to send a message back to the client.

The HighScores Class

The HighScores object maintains an array of ScoreInfo objects, which it initially populates by calling loadScores() to load the "scores.txt" text file from the current directory. saveScores() writes the array's contents back into "scores.txt".

It is preferable to maintain simple data (like these name/score pairs) in text form rather than as a serialized object. This makes the data easy to examine and edit with ordinary text processing tools.

© Andrew Davison. 2003

The ScoreClient Class

The ScoreClient class seems complicated because of its GUI interface. Figure 7 shows a ScoreClient object.

Figure 7. A ScoreClient Object.

The large text area is represented by the jtaMesgs object. There are two text fields for entering a name and score. Pressing enter in the score field will trigger a call to actionPerformed() in the object, as will pressing the "Get Scores" button.

ScoreClient calls makeContact() to instantiate a Socket object for the server at its specified IP address and port. When the connection is made, input and output streams are layered on top of its socket.

```
    :
private static final int PORT = 1234;     // server details
private static final String HOST = "localhost";

private Socket sock;
private BufferedReader in;     // i/o for the client
private PrintWriter out;
    :

private void makeContact()
{
  try {
    sock = new Socket(HOST, PORT);
    in  = new BufferedReader(
          new InputStreamReader( sock.getInputStream() ) );
    out = new PrintWriter( sock.getOutputStream(), true );
  }
  catch(Exception e)
  {  System.out.println(e);  }
}
```

"localhost" is given as the server's host name, since the server is running on the same machine as the client. "localhost" is a *loopback address* and can be employed even when the machine is disconnected from the internet, although the TCP/IP protocol

 © Andrew Davison. 2003

must be set up in the OS. On most systems (including Windows), it is possible to enter the command "ping localhost" to check the functioning of the loopback.

actionPerformed() differentiates between the two kinds of user input:

```
public void actionPerformed(ActionEvent e)
// Either a name/score is to be sent or the "Get Scores"
// button has been pressed
{
  if (e.getSource() == jbGetScores)
    sendGet();
  else if (e.getSource() == jtfScore)
    sendScore();
}
```

sendGet() shows how the client sends a message to the server (in this case a "get" string), and waits for a response (a "HIGH$$..." string), which it displays in the text area.

```
private void sendGet()
{
// Send "get" command, read response and display it
// Response should be "HIGH$$ n1 & s1 & .... nN & sN & "
  try {
    out.println("get");
    String line = in.readLine();
    System.out.println(line);
    if ((line.length() >= 7) &&     // "HIGH$$ "
        (line.substring(0,6).equals("HIGH$$")))
      showHigh( line.substring(6).trim() );
             // remove HIGH$$ keyword and surrounding spaces
    else    // should not happen
      jtaMesgs.append( line + "\n");
  }
  catch(Exception ex)
  { jtaMesgs.append("Problem obtaining high scores\n");
    System.out.println(ex);
  }
}
```


© Andrew Davison. 2003

Figure 8 shows how the high score list looks in the text area window.

Figure 8. High Scores Output in the Client.

sendGet() makes the client wait for the server to reply:

```
out.println("get");
String line = in.readLine();
```

This means that the client will be unable to process further user commands until the server has sent back the high scores information. This is not a good design strategy for more complex client applications. The chat client of chapter 19 shows how threads can be employed to make network interaction separate from the rest of the application.

The client should send a "bye" message before it breaks a connection, and this is achieved by calling closeLink() when the client's close box is clicked:

```
public ScoreClient()
{
   super( "High Score Client" );
      :
   addWindowListener( new WindowAdapter() {
     public void windowClosing(WindowEvent e)
     { closeLink(); }
   });
    :
} // end of ScoreClient();


private void closeLink()
{
  try {
    out.println("bye");     // tell server
    sock.close();
  }
  catch(Exception e)
  {  System.out.println( e );   }

  System.exit( 0 );
}
```

 © Andrew Davison. 2003

An Simple Alternative Client

Since the client is communicating with the server using TCP/IP, it is possible to replace the client with the telnet command:

```
telnet localhost 1234
```

This will initiate a TCP/IP link at the specified host address and port, where the server is listening. The advantage is the possibility of testing the server without writing a client. The disadvantage is that the user must type the messages directly, without the help of a GUI interface. Figure 9 shows a telnet window after the server has responded to a "get" message.

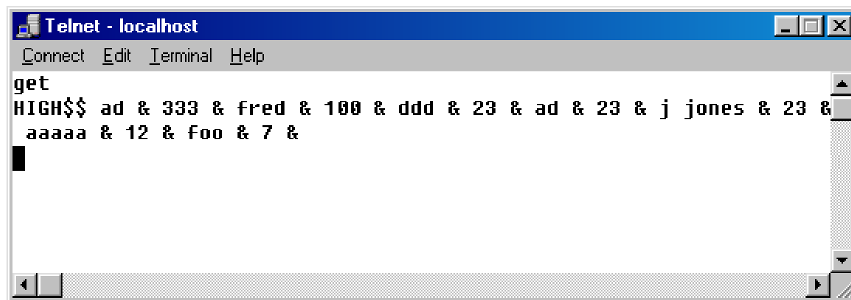

Figure 9. A Telnet Client.

It may be necessary to switch on the 'local echo' feature in telnet's preferences dialog before user typing (e.g. the "get" message) is seen on screen.

4.2. TCP Client and Multithreaded Server

The ScoreServer class is inadequate for real server-side applications because it can only deal with a single client at a time. The ThreadedScoreServer class described in this section solves that problem by creating a thread (a ThreadedScoreHandler object) to process each client who connects. Since a thread interacts with each client, the main server is free to accept multiple connections. Figure 10 shows this in diagram form.

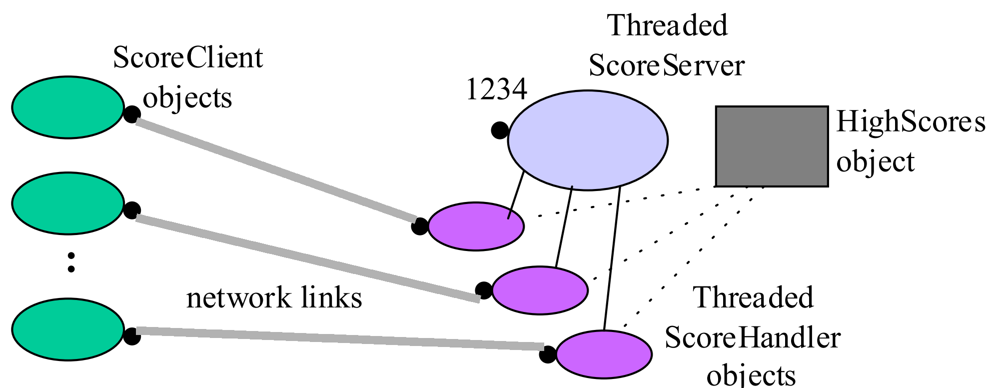

Figure 10. Clients and Multithreaded Server.

 © Andrew Davison. 2003

The ScoreClient class need not be changed: a client is unaware that it is talking to a thread.

The HighScores object is referenced by all the threads (indicated by the dotted lines in Figure 10) so that the scores can be read and changed. The possibility of change means that the data inside HighScores must be protected from concurrent updates by two or more of the threads, and from an update occurring at the same time as the scores are being read. These synchronization problems are quite easily solved, as explained below.

ThreadedScoreServer is simpler than its sequential counterpart, since it no longer processes client requests. It consists only of a constructor which sets up a loop waiting for client contacts, handled by threads:

```
public ThreadedScoreServer()
{
  hs = new HighScores();
  try {
    ServerSocket serverSock = new ServerSocket(PORT);
    Socket clientSock;
    String cliAddr;

    while (true) {
      System.out.println("Waiting for a client...");
      clientSock = serverSock.accept();
      cliAddr = clientSock.getInetAddress().getHostAddress();
      new ThreadedScoreHandler(clientSock, cliAddr, hs).start();
    }
  }
  catch(Exception e)
  {  System.out.println(e);  }
}  // end of ThreadedScoreServer()
```

Each thread gets a reference to the client's rendezvous socket and the HighScores object.

ThreadedScoreHandler contains almost identical code to the sequential ScoreServer class; for example, it has processClient() and doRequest() methods. The main difference is the run() method:

```
public void run()
{
  try {
    // Get I/O streams from the socket
    BufferedReader in  = new BufferedReader(
       new InputStreamReader( clientSock.getInputStream() ) );
    PrintWriter out =
      new PrintWriter( clientSock.getOutputStream(), true );

    processClient(in, out); // interact with a client

    // Close client connection
    clientSock.close();
    System.out.println("Client (" + cliAddr +
```

 © Andrew Davison. 2003

```
                              ") connection closed\n");
    }
    catch(Exception e)
    {  System.out.println(e);  }
  }
```

A comparison with the sequential server shows that run() contains code like that executed in ScoreServer's constructor after a rendezvous socket is created.

The HighScores Class

The ThreadedScoreHandler objects call HighScores' toString() and addScore() methods. toString() returns the current scores list as a string, while addScore() updates the list.

The danger of concurrent access to the scores list is easily avoided since the data is maintained in a single object which is referenced by all the threads. Manipulation only occurs through the toString() and addScore() methods.

A 'lock' can be placed on the object by making the toString() and addScore() methods synchronized:

```
synchronized public String toString()
{ ... }

synchronized public void addScore(String line)
{ ... }
```

The lock means that only a single thread can be executing inside toString() or addScore() at any time. Concurrent access is no longer possible.

This approach is relatively painless because of the decision to wrap the shared data (the scores list) inside a single shared object.

One concern may be the impact on response times by prohibiting concurrent access, but both toString() and addScores() are short, simple methods which quickly return after being called.

One advantage of the threaded handler approach is not illustrated by this example: each handler can store client-specific data locally. For instance, each ThreadedScoreHandler could maintain information about the history of client communication in its session. Since this data is managed by each thread, it relieves the main server of unnecessary complexity.

4.3. TCP Client and Multiplexing Server

Java 2 Standard Edition 1.4. introduced *nonblocking sockets*, which allow networked applications to communicate without blocking the processes/threads involved. This made it possible to implement a server which can multiplex (switch) between different clients without the need for threads.

© Andrew Davison. 2003

At the heart of this approach is a method called select(), which may remind UNIX network programmers of the select() system call. They are closely related, and the coding strategy for a multiplexing server in Java is quite close to one in C on UNIX.

An advantage of the multiplexing server technique is the return to a single server without threads. This may be an important gain on a platform with limited resources.

A related advantage is the absence of synchronization problems with shared data, because there are no threads. The only process is the server.

A disadvantage is that any client-specific state (which had previously been placed in each thread) must be maintained by the server. For instance, if there are multiple clients connected to the server, each with a communications history to be maintained, then the server must hold all their histories.

Nonblocking sockets mean that method calls that might potentially block forever, such as accept() and readLine(), need only be executed when data is known to be present, or can be wrapped in timeouts. This is particularly useful for avoiding some forms of hacker attack, or dealing with users who are too slow!

Figure 11 shows the various objects involved in the multiplexing server example.

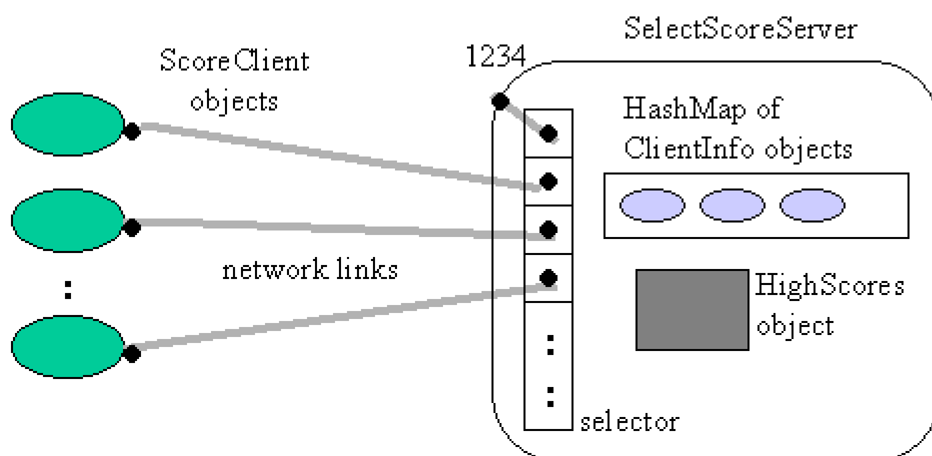

Figure 11. Clients and Multiplexing Server.

Selector is the main new class in the nonblocking additions to Java. A Selector object is able to monitor multiple socket channels, and returns a collection of *keys* (client requests) as required. A socket channel is a new type of socket (from the SocketChannel class).

ClientInfo is our own class, developed for this application. Each ClientInfo object in the hash map contains details about a client, and methods for receiving and sending messages to the client. The principal complexity of ClientInfo is involved in dealing with nonblocking client input.

The main purpose of the server is to listen to several socket channels at once by using a Selector object. Initially, the server's own socket channel is added to the selector, and subsequent connections by new clients, represented by new socket channels, are also added.

© Andrew Davison. 2003

When input is available from a client, it is read immediately. However, the input may only contain part of a message: there is no waiting for a complete message.

The pseudo-code algorithm for the server is given below.

```
create a SocketChannel for the server;
create a Selector;
register the SocketChannel with the Selector (for accepts);

while(true) {
  wait for keys (client requests) in the Selector;
  for each key in the Selector {
    if (key is Acceptable) {
      create a new SocketChannel for the new client;
      register the SocketChannel with the Selector (for reads);
    }
    else if (key is Readable) {
      extract the client SocketChannel from the key;
      read from the SocketChannel;
      store partial message, or process full message;
    }
  }
}
```

The server waits inside a while loop for keys to be generated by the Selector. A key contains information about a pending client request. A Selector object may store four types of key:

- a request by a new client to connect to the server (an isAcceptable key)
- a request by an existing client to deliver some input (an isReadable key)
- a request by an existing client for the server to send it data (an isWriteable key)
- a request by a server accepting a client connection (an isConnectable key)

The first two request types are used in the multiplexing server. The last type of key is typically employed by a nonblocking client to detect when a connection has been successfully made with a server.

The socket channel for the server is created in the SelectScoreServer() constructor:

```
  ServerSocketChannel serverChannel = ServerSocketChannel.open();
  serverChannel.configureBlocking(false);   // use non-blocking mode

  ServerSocket serverSocket = serverChannel.socket();
  serverSocket.bind( new InetSocketAddress(PORT_NUMBER) );
```

The nonblocking nature of the socket is made possible by first creating a ServerSocketChannel object, and then extracting a ServerSocket.

 © Andrew Davison. 2003

The server's socket channel is registered a Selector so it will collect connection requests.

```
Selector selector = Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

Other possible options to register() are OP_READ, OP_WRITE and OP_CONNECT, corresponding to the different types of keys.

The while loop of the pseudocode is translated fairly directly into real code:

```
while (true) {
  selector.select();                  // wait for keys
  Iterator it = selector.selectedKeys().iterator();
  SelectionKey key;
  while (it.hasNext()) {              // look at each key
    key = (SelectionKey) it.next();   // get a key
    it.remove();                      // remove it
    if (key.isAcceptable())       // a new connection?
      newChannel(key, selector);
    else if (key.isReadable())    // data to be read?
      readFromChannel(key);
    else
      System.out.println("Did not process key: " + key);
  }
}
```

newChannel() is called when a new client has requested a connection. The connection is accepted and registered with the selector to make it collect read requests (i.e. data from the client which should be read).

```
private void newChannel(SelectionKey key, Selector selector)
{
  try {
    ServerSocketChannel server =
              (ServerSocketChannel) key.channel();
    SocketChannel channel = server.accept();         // get channel
    channel.configureBlocking (false);          // use non-blocking
    channel.register(selector, SelectionKey.OP_READ);
                           // register it with selector for reading

    clients.put(channel,
              new ClientInfo(channel, this) );   // store info
  }
  catch (IOException e)
  {  System.out.println( e ); }
}
```

The call to accept() is nonblocking: it will raise a NotYetBoundException if there is no pending connection. The connection is represented by a SocketChannel (a nonblocking version of the Socket class), and this is registered with the Selector to collect its read requests.

Since a new client has just connected, a new ClientInfo object is added to the HashMap. The key for the HashMap entry is the client's channel, which is unique.

 © Andrew Davison. 2003

readFromChannel() is called when there is a request by an existing client for the server to read its data. As mentioned above, this may not be a complete message, which introduces some problems. It is necessary to store partial messages from clients until they are completed (a complete message ends with a '\n'). The reading and storage is managed by the ClientInfo object representing the client with the request.

```
private void readFromChannel(SelectionKey key)
// process input that is waiting on a channel
{
  SocketChannel channel = (SocketChannel) key.channel();
  ClientInfo ci = (ClientInfo) clients.get(channel);
  if (ci == null)
    System.out.println("No client info for channel " + channel);
  else {
    String msg = ci.readMessage();
    if (msg != null) {
      System.out.println("Read message: " + msg);
      if (msg.trim().equals("bye")) {
        ci.closeDown();
        clients.remove(channel);  // delete ci from hash map
      }
      else
        doRequest(msg, ci);
    }
  }
} // end of readFromChannel()
```

readFromChannel() extracts the channel reference from the client request, and uses it to lookup the associated ClientInfo object in the hash map. The ClientInfo object deals with the request via a call to readMessage(), which returns the full message or null if the message is still incomplete.

If the message is "bye" then the server requests that the ClientInfo object closes the connection, and the object is discarded. Otherwise, the message is processed using doRequest().

```
private void doRequest(String line, ClientInfo ci)
/*  The input line can be one of:
          "score name & score &"
    or    "get"     */
{
  if (line.trim().toLowerCase().equals("get")) {
    System.out.println("Processing 'get'");
    ci.sendMessage( hs.toString() );
  }
  else if ((line.length() >= 6) &&     // "score "
      (line.substring(0,5).toLowerCase().equals("score"))) {
    System.out.println("Processing 'score'");
    hs.addScore( line.substring(5) );    // cut the score keyword
  }
  else
    System.out.println("Ignoring input line");
}
```

 © Andrew Davison. 2003

The input line can be a "get" or a "score" message. If it is a "get", then the ClientInfo object is asked to send the high scores list to the client. If the message is a new name/score pair, then the HighScores object is notified.

The ClientInfo Class

ClientInfo has three public methods: readMessage(), sendMessage(), and closeDown (). readMessage() reads input from a client's socket channel. sendMessage() sends a message along a channel to the client, and closeDown() closes the channel.

Data is read from a socket channel into a Buffer object holding bytes. Buffer is another new class in the nonblocking additions. A Buffer object is a fixed size container for items belong to a Java base type, such as byte, int, char, double, boolean, and so on. A Buffer object works in a similar way to a file: there is a "current position", and after each read or write operation, the current position indicates the next item in the buffer.

There are two important size notions for a buffer: its *capacity* and its *limit*. The capacity is the maximum number of items the buffer can contain, while the limit is a value between 0 and capacity representing the current size limit for the buffer.

Since data sent through a socket channel is stored in a ByteBuffer object (a buffer of bytes), it is necessary to translate it (decode it) into a String before the data can be tested to see if it is a complete or not. A complete message is a string ending with a '\n'.

The constructor for ClientInfo initialises the byte buffer and the decoder.

```
// globals
private static final int BUFSIZ = 1024;  // max size of a message

private SocketChannel channel;  // the client's channel
private SelectScoreServer ss;   // the top-level server
private ByteBuffer inBuffer;    // for storing input

private Charset charset;     // for decoding bytes --> string
private CharsetDecoder decoder;


public ClientInfo(SocketChannel chan, SelectScoreServer ss)
{
  channel = chan;
  this.ss = ss;
  inBuffer = ByteBuffer.allocateDirect(BUFSIZ);
  inBuffer.clear();

  charset = Charset.forName("ISO-8859-1");
  decoder = charset.newDecoder();

  showClientDetails();
}
```

The buffer is a fixed size, 1024 bytes. The obvious question is whether this is sufficient for message passing. The only long message is the high scores list which is sent from the server back to the client, and 1024 characters (bytes) should be adequate.

 © Andrew Davison. 2003

readMessage() is called when the channel contains data.

```
public String readMessage()
{
  String inputMsg = null;
  try {
    int numBytesRead = channel.read(inBuffer);
    if (numBytesRead == -1) {     // channel has gone
      channel.close();
      ss.removeChannel(channel); // tell SelectScoreServer
    }
    else
      inputMsg = getMessage(inBuffer);
  }
  catch (IOException e)
  {  System.out.println("rm: " + e);
     ss.removeChannel(channel); // tell SelectScoreServer
  }

  return inputMsg;
}  // end of readMessage()
```

A channel read() will not block, returning the number of bytes read (which may be 0). If it returns -1 then something has happened to the input channel; the channel is closed and the ClientInfo object removed by calling removeChannel() in the main server. It is also possible for read() to raise an IOException, which triggers the same removal.

The real work of reading a message is done by getMessage()

```
private String getMessage(ByteBuffer buf)
{
  String msg = null;
  int posn = buf.position();   // current buffer sizes
  int limit = buf.limit();

  buf.position(0);    // set range of bytes for translation
  buf.limit(posn);
  try {   // translate bytes-->string
    CharBuffer cb = decoder.decode(buf);
    msg = cb.toString();
  }
  catch(CharacterCodingException cce)
  { System.out.println( cce );  }

  // System.out.println("Current msg: " + msg);
  buf.limit(limit);    // reset buffer to full range of bytes
  buf.position(posn);

  if (msg.endsWith("\n")) {    // we assume '\n' is the last char
    buf.clear();
    return msg;
  }
  return null;      // since we still only have a partial mesg
}  // end of getMessage()
```


© Andrew Davison. 2003

position() returns the index position of the next empty spot in the buffer: there are bytes stored from position 0 up to posn-1. The current limit for the buffer is also stored, and then changed to be the current position. This means that when decode() is called, only the part of the buffer containing bytes will be considered.

After the translation, the resulting string is checked to see if it ends with a '\n', in which case the buffer is reset (treated as being empty) and the message returned.

There is a potential problem with this approach: the assumption that the last character in the buffer will eventually be '\n'. This depends on what data is present in the channel when read() is called in readMessage(). It might be that the channel contains the final bytes of one message *and* some bytes of the next, as illustrated by Figure 12.

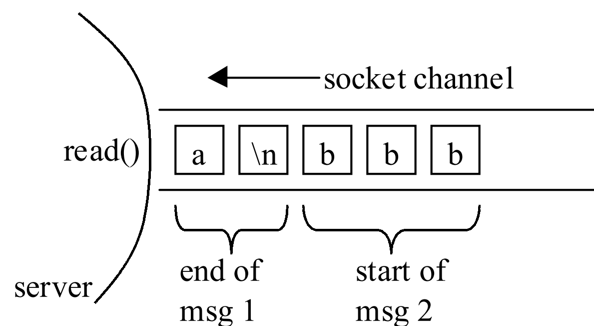

Figure 12. Reading Bytes in a Channel.

The read() call in Figure 11 will add the bytes for "a\nbbb" to the byte buffer, so placing '\n' is in the midst of the buffer rather than at the end. Consequently, a simple endsWith() test of the extracted string is insufficient:

```
if (msg.endsWith("\n")) {. . . }
```

However, in our tests this problem never appeared since read() was called very quickly, adding each incoming byte to the buffer as soon as it arrived – a '\n' was always read before the next byte appeared in the channel.

sendMessage() sends a specified message along the channel back to the client. The two issues here are:

1) the need to translate the string to bytes in a buffer before transmission, and
2) dealing with the case that the message requires several writes before it is all placed onto the channel.

```
public boolean sendMessage(String msg)
{
  String fullMsg = msg + "\r\n";

  ByteBuffer outBuffer = ByteBuffer.allocateDirect(BUFSIZ);
  outBuffer.clear();
  outBuffer.put( fullMsg.getBytes() );
  outBuffer.flip();
```

 © Andrew Davison. 2003

```
    boolean msgSent = false;
    try {
      // send the data, don't assume it goes all at once
      while( outBuffer.hasRemaining() )
        channel.write(outBuffer);
      msgSent = true;
    }
    catch(IOException e)
    { System.out.println(e);
      ss.removeChannel(channel); // tell SelectScoreServer
    }

    return msgSent;
  }  // end of sendMessage()
```

The buffer is filled with the message. The flip() call sets the buffer limit to the current position (i.e. the just after the end of the message), and then sets the current position back to 0.

The while loop uses hasRemaining(), which returns true so long has there are elements remaining between the current position and the buffer limit. Each write() calls advanced the position through the buffer. One write() call should be sufficient to place all the bytes onto the channel unless the buffer is very large or the channel is overloaded.

write() may raise an exception, which causes the channel and the ClientInfo object to be discarded.

The Client

ScoreClient stays as the client-side application. The advantage is that high-level IO can be used instead of byte buffers.

A nonblocking client can be useful for attempting a connection without having to wait for the server to respond. Whether a connection operation is in progress can be checked by calling isConnectionPending().

4.4. UDP Client and Server

All the previous examples use TCP, but the client and server in this section are recoded to utilize UDP communication. The result is another form of multiplexing server, but without the need for nonblocking sockets. The complexity of the code is much less than in the last example.

The downsides of this approach are the usual ones related to UDP: the possibility of packet loss and reordering, although these problems did not occur in our tests which were run on the same machine, and on machines connected by a LAN.

Another disadvantage of using UDP is the need to write a client before the server can be tested; telnet uses TCP/IP so cannot be employed here.

 © Andrew Davison. 2003

Figure 13 illustrates the form of communication between ScoreUDPClient objects and the ScoreUDPServer.

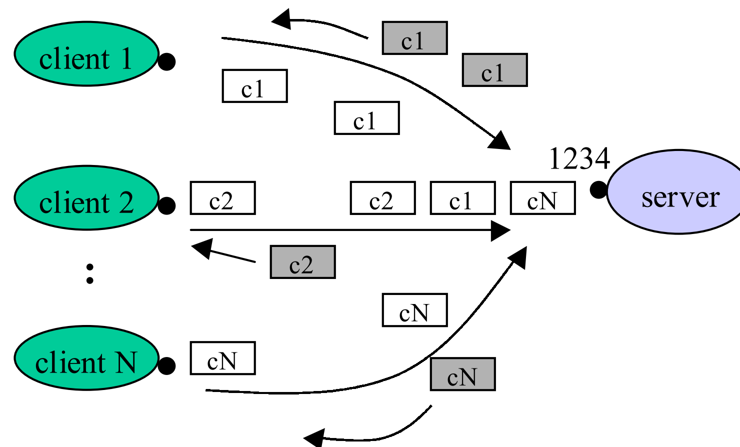

Figure 13. UDP Clients and Server.

Since there is no long-term connection between a client and the server, it is possible for multiple clients to send datagrams at the same time. The server will process them in their order of arrival. A datagram automatically includes the hostname and IP address of its sender, so response messages can be easily sent back.

The server sets up a DatagramSocket listening at port 1234, and then enters a loop which waits for a packet, processes it, and repeats.

```
// globals
private static final int PORT = 1234;
private static final int BUFSIZE = 1024;   // max size of a message

private HighScores hs;
private DatagramSocket serverSock;


public ScoreUDPServer()
{
  try {  // try to create a socket for the server
    serverSock = new DatagramSocket(PORT);
  }
  catch(SocketException se)
  {  System.out.println(se);
     System.exit(1);
  }
  waitForPackets();
}


private void waitForPackets()
{
  DatagramPacket receivePacket;
  byte data[];
  hs = new HighScores();

  try {
```

 © Andrew Davison. 2003

```
      while (true) {
        data = new byte[BUFSIZE];  // set up an empty packet
        receivePacket = new DatagramPacket(data, data.length);
        System.out.println("Waiting for a packet...");
        serverSock.receive( receivePacket );

        processClient(receivePacket);
        hs.saveScores();   // backup scores after each package
      }
    }
    catch(IOException ioe)
    {  System.out.println(ioe);  }
  }  // end of waitForPackets()
```

The data in a packet is extracted into a byte array of a fixed size. Naturally, the size of the array should be sufficient for the kinds of messages being delivered.

processClient() extracts the client's address, IP number and converts the byte array into a string.

```
  InetAddress clientAddr = receivePacket.getAddress();
  int clientPort = receivePacket.getPort();
  String clientMesg = new String( receivePacket.getData(), 0,
                                  receivePacket.getLength() );
```

These are passed to doRequest() which deals with the two possible message types: "get" and "score". There is no "bye" message because there is no long-term connection that needs to be broken. Part of the reason for the simplicity of coding with UDP is the absence of processing related to connection termination (whether intended or due to an error).

A reply is sent by calling sendMessage():

```
  private void sendMessage(InetAddress clientAddr,
                              int clientPort, String mesg)
  // send message to socket at the specified address and port
  {
    byte mesgData[] = mesg.getBytes();   // convert to byte[] form
    try {
      DatagramPacket sendPacket =
          new DatagramPacket( mesgData, mesgData.length,
                                    clientAddr, clientPort);
      serverSock.send( sendPacket );
    }
    catch(IOException ioe)
    {  System.out.println(ioe);  }
  }
```


© Andrew Davison. 2003

The ScoreUDPClient

The client has the same GUI interface as the TCP version (see Figure 14), allowing the user to send commands by clicking on the "Get Scores" button or by entering name/score pairs into the text fields.

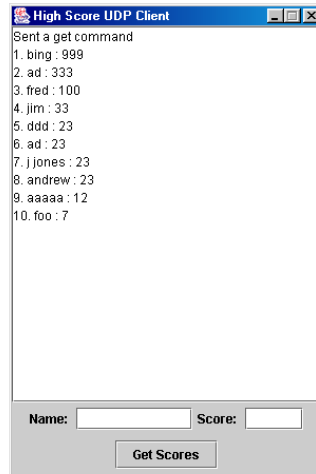

Figure 14. The UDP Client GUI.

The application uses the implicit thread associated with Swing's processing of GUI events to send commands to the server. Processing of the messages returned by the server is handled in the application's main execution thread.

The constructor starts the application thread by setting up the client's datagram socket:

```
// globals
private static final int SERVER_PORT = 1234;     // server details
private static final String SERVER_HOST = "localhost";
private static final int BUFSIZE = 1024;   // max size of a message

private DatagramSocket sock;
private InetAddress serverAddr;
   :

public ScoreUDPClient()
{  super( "High Score UDP Client" );

   initializeGUI();
   try {   // try to create the client's socket
     sock = new DatagramSocket();
   }
   catch( SocketException se ) {
     se.printStackTrace();
     System.exit(1);
   }
   try {  // try to turn the server's name into an internet address
     serverAddr = InetAddress.getByName(SERVER_HOST);
   }
   catch( UnknownHostException uhe) {
     uhe.printStackTrace();
     System.exit(1);
   }
```

 © Andrew Davison. 2003

```
    setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    setSize(300,450);
    setResizable(false);    // fixed size display
    show();

    waitForPackets();
  } // end of ScoreUDPClient();
```

waitForPackets() bears a striking resemblance to the same named method in the server. It contains a loop which waits for an incoming packet (from the server), processes it, and repeats.

```
  private void waitForPackets()
  { DatagramPacket receivePacket;
    byte data[];
    try {
      while (true) {
        // set up an empty packet
        data = new byte[BUFSIZE];
        receivePacket = new DatagramPacket(data, data.length);

        System.out.println("Waiting for a packet...");
        sock.receive( receivePacket );

        processServer(receivePacket);
      }
    }
    catch(IOException ioe)
    {  System.out.println(ioe);  }
  }
```

processServer() extracts the address, port number, and message string from the packet, prints the address and port to standard output, and adds the message into the text area.

The GUI thread is triggered by the system calling actionPerformed():

```
  public void actionPerformed(ActionEvent e)
  {
    if (e.getSource() == jbGetScores) {
      sendMessage(serverAddr, SERVER_PORT, "get");
      jtaMesgs.append("Sent a get command\n");
    }
    else if (e.getSource() == jtfScore)
      sendScore();
  }
```

An important issue with threads is synchronization of shared data. The DatagramSocket is shared but the GUI thread only transmits datagrams, while the application thread only receives, so conflict is avoided.

The JTextArea component, jtaMesgs, *is* shared between the threads, as a place to write messages for the user. However, there is little danger of multiple writes occurring at the same time due to the request/response nature of the communication between the client and server – a message from the server only arrives as a response

to an earlier request by the client. Synchronization would be more important if the server could deliver messages to the client at any time, as in the Chat systems developed in chapter 19.

Another reason for the low risk of undue interaction is that the GUI thread only places short messages into the text area, which are added quickly.

5. Peer-to-Peer Programming in Java

The simplest form of P2P programming in Java employs UDP multicasting – datagram packets with a MulticastSocket object.

A MulticastSocket requires a multicast IP address (a class D IP address) and a port number. Class D IP addresses fall in the range 224.0.0.0 to 239.255.255.255, although certain addresses are reserved.

A peer wishing to 'subscribe' to a multicast group must create a MulticastSocket representing the group, and use joinGroup() to begin receiving communication. A peer leaves a group by calling leaveGroup(), or by terminating.

Currently applets are not allowed to use multicast sockets.

The application described here takes a (welcome) break from accessing/modifying high score lists, which doesn't make for a particularly suitable P2P example. Instead, a MultiTimeServer transmits a packet to a multicast group every second, containing the current time and date. MultiTimeClient objects wait for packets to appear in the group, and print them to standard output. The situation is shown in Figure 15.

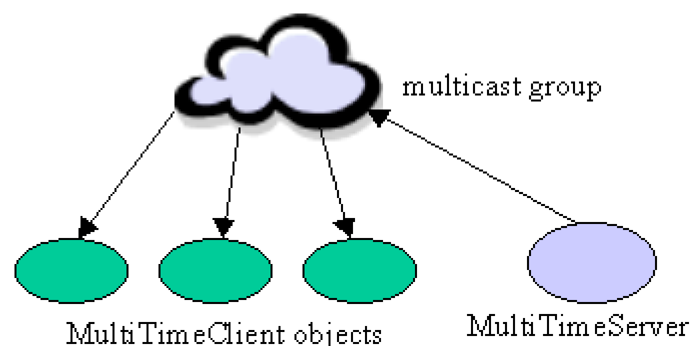

Figure 15. UDP Multicasting Clients and Server.

The use of the words 'client' and 'server' are a little misleading since all the objects involved in the group can potentially send and receive messages. It is our choice to restrict the 'server' to sending, and the 'clients' to reading.

The MultiTimeServer Class

The MultiTimeServer object creates a multicast socket for a group at IP address 228.5.6.7, port 6789, and enters a loop which sends a packet out every second.

 © Andrew Davison. 2003

```
public class MultiTimeServer
{
  private static final String MHOST = "228.5.6.7";
  private static final int PORT = 6789;

  public static void main(String args[]) throws Exception
  {
     InetAddress address = InetAddress.getByName(MHOST);
     MulticastSocket msock = new MulticastSocket(PORT);
     msock.joinGroup(address);

     DatagramPacket packet;
     System.out.print("Ticking");
     while(true){
       Thread.sleep(1000);   // 1 second delay
       System.out.print(".");
       String str = (new Date()).toString();
       packet = new DatagramPacket(str.getBytes(), str.length(),
                                   address, PORT);
       msock.send(packet);
     }
  }

} // end of MultiTimeServer class
```

The server is started like so:

```
$ java MultiTimeServer
Ticking.......
```

The MultiTimeClient Class

The client creates a multicast socket for the same group, and enters an infinite loop waiting for packets to appear.

```
public class MultiTimeClient
{
  private static final String MHOST = "228.5.6.7";
  private static final int PORT = 6789;

  public static void main(String args[]) throws IOException
  {
     InetAddress address = InetAddress.getByName(MHOST);
     MulticastSocket msock = new MulticastSocket(PORT);
     msock.joinGroup(address);

     byte[] buf = new byte[1024];
     DatagramPacket packet = new DatagramPacket(buf, buf.length);
     String dateStr;
     while(true){
       msock.receive(packet);
       dateStr = new String( packet.getData() ).trim();
       System.out.println(packet.getAddress() + " : " + dateStr);
     }
   }
}
```


© Andrew Davison. 2003

A client's execution is shown in Figure 16.

```
D>java MultiTimeClient
/172.30.0.157 : Wed Jun 11 15:55:46 ICT 2003
/172.30.0.157 : Wed Jun 11 15:55:47 ICT 2003
/172.30.0.157 : Wed Jun 11 15:55:48 ICT 2003
/172.30.0.157 : Wed Jun 11 15:55:49 ICT 2003
/172.30.0.157 : Wed Jun 11 15:55:50 ICT 2003
/172.30.0.157 : Wed Jun 11 15:55:51 ICT 2003
/172.30.0.157 : Wed Jun 11 15:55:52 ICT 2003
```

Figure 16. Multicast UDP Client in Action.

Chapter 19 contains a UDP multicasting version of a Chat application.

6. Firewalls

Firewalls are unfortunately a part of today's networking experience, or rather the lack of it. Most companies, government institutions, universities, and so on, utilize firewalls to block access to the wider internet – typically socket creation on non-standard ports, and most standard ones, are prohibited, and Web pages must be retrieved through a proxy which filters (limits) the traffic.

This situation means that the code described so far may not actually work, because it requires the creation of sockets. The DayPing example given below is a simple Java application that attempts to contact a 'time of day' server. The example in Figure 17 uses the server at the National Institute of Standards and Technology in Boulder, Colorado.

```
D>java DayPing time-A.timefreq.bldrdoc.gov
time-A.timefreq.bldrdoc.gov is alive at

52802 03-06-12 01:45:22 50 0 0 714.4 UTC(NIST) *

D>_
```

Figure 17. Time of Day Client

DayPing opens a socket to the host at port 13 where the standard 'time of day' service is always set to be listening. The response is printed out using println() after layering a BufferedReader stream on top of the network link.

```
public class DayPing
{
   public static void main(String args[]) throws IOException
   {  if (args.length != 1) {
         System.out.println("usage:  java DayPing <host> ");
         System.exit(0);
      }

      Socket sock = new Socket(args[0],13);   // host and port
      BufferedReader br = new BufferedReader(
              new InputStreamReader( sock.getInputStream() ) );
```

 © Andrew Davison. 2003

```
      System.out.println( args[0] + " is alive at ");
      String line;
      while ( (line = br.readLine()) != null)
        System.out.println(line);
      sock.close();
   }
} // end of DayPing class
```

Figure 17 is the desired output, but this program will not work on most student machines in the department where I work. For them, the result is:

```
D> java DayPing time-A.timefreq.bldrdoc.gov
Exception in thread "main"
java.net.NoRouteToHostException: Operation timed out: no further
information
        at java.net.PlainSocketImpl.
                  socketConnect (Native Method)
                        :
        at DayPing.main(DayPing.java:34)
```

There is a long delay (2-3 minutes) before the exception is raised, due to the OS waiting for a possible connection. The exception indicates the presence of a firewall preventing the link.

With TCP client applications like DayPing, it is possible to check the server with telnet:

$ telnet time-A.timefreq.bldrdoc.gov 13

There is a similar outcome: a delay of a few minutes, followed by an error message saying that the connection could not be made.

My university has a policy of disallowing socket creation for hosts outside the local domain, and Web pages can only be retrieved by going through a proxy located at cache.psu.ac.th, port 8080.

There are two choices:

1) use a commercial ISP which does allow socket creation, or
2) rewrite the DayPing application to utilize URLs. This is the approach taken here.

6.1. Retrieving a Web Page

The simplest way of obtaining a Web page in Java is with a URL object, which retrieves it as a stream of text in a similar way to streams connected to sockets. The GetWebPage application downloads a Web page using a URL specified on the command line.

```
public class GetWebPage
{
   public static void main(String args[]) throws IOException
   { if (args.length != 1) {
        System.out.println("usage:  java GetWebPage <url> ");
```

 © Andrew Davison. 2003

```
      System.exit(0);
    }

    URL url  = new URL(args[0]);

    BufferedReader br = new BufferedReader(
             new InputStreamReader( url.openStream() ));

    // print first ten lines of contents
    int numLine = 0;
    String line;
    while ( ((line = br.readLine()) != null) && (numLine <= 10) ) {
      System.out.println(line);
      numLine++;
    }
    if (line != null)
      System.out.println(". . .");

    br.close();
  }
} // end of GetWebPage class
```

GetWebPage can (theoretically) access any Web page, including one giving the current time in Thailand, http://www.bsdi.com/xdate?Asia/Bangkok. However, the command line must include three proxy options (proxySet, proxyHost, proxyPort) to tell the JVM to employ the university's proxy server. The result is shown in Figure 18.

```
D>java -DproxySet=true -DproxyHost=cache.psu.ac.th -DproxyPort=8080  GetWebPage
http://www.bsdi.com/xdate?Asia/Bangkok
<HEAD>
<TITLE>Date and Time Gateway</TITLE>
</HEAD>
<BODY>
<H1><IMG ALIGN="right" ALT="" SRC="/icons/misc/datetime.gif"> Date and Time Gate
way</H1>
<h3><i>      <u>Data <font color="#990033">NOT</fo
nt> guaranteed to be accurate!</u></i></h3>
<H2>Current Timezone: Asia/Bangkok</H1>
<H2>Thu Jun 12 09:37:32 ICT 2003</H2>
<H3>Available Timezones</H3>

<H4>Greenwich Mean Time (GMT/UCT/UTC/Universal/Zulu)</H4>
. . .
```

Figure 18. Retrieving a Web Page through a Proxy.

An alternative to command line settings is to specify the proxy details within the program:

```
    Properties props = System.getProperties();
    props.put("proxySet", "true");
    props.put("proxyHost", "cache.psu.ac.th");
    props.put("proxyPort", "8080");
    System.setProperties(props);
```

This should be done before the creation of the URL object.

© Andrew Davison. 2003

An important aspect of this coding style is the processing of the text stream arriving from the Web server. It is often far from trivial to delve through the mix of HTML tags, JavaScript, and others, to find the required piece of information (e.g. that the current time in Bangkok is 9.37am).

Another problem is that text analysis code tends to break after a while, as the format of the Web page is changed/updated by the Web server's administrator.

6.2. Proxy Authorization

Some proxy servers demand a login and password before pages can be downloaded (this is true of one of the high bandwidth links in my department).

Supplying these from within a Java application requires the use of a URLConnection object to permit greater control over the URL link.

```
URL url  = new URL(args[0]);
URLConnection conn = url.openConnection();
```

The login and password strings must be passed to the proxy server as a single string of the form "login:password" translated into Base64 encoding

```
Base64Converter bc = new Base64Converter();
String encoding = bc.encode(login + ":" + password );
```

The Base64Converter class was written by David W. Croft, and available with documentation from:

> *Java Tip 47*, JavaWorld.com, April 6th 2000
> `http://www.javaworld.com/javaworld/javatips/jw-javatip47.html`

There is also an undocumented BASE64Encoder class in the sun.misc package of J2SDK. It is used like so:

```
Base64Encoder bc = new Base64Encoder();
String mesg = login + ":" + password;
String encoding = bc.encode( mesg.getBytes() );
```

The encoded string is sent to the proxy as an authorization request:

```
conn.setRequestProperty("Proxy-Authorization",
                              "Basic " + encoding);
```

GetWebPageP.java contains all of this functionality; it reads the user's password and desired URL from the command line.

```
public class GetWebPageP
{
  private static final String LOGIN = "ad";  // modify this

  public static void main(String args[]) throws IOException
  {  if (args.length != 2) {
      System.out.println("usage: java GetWebPageP <password> <url>");
      System.exit(0);
     }

     // set the properties used for proxy support
```

 © Andrew Davison. 2003

```
      Properties props = System.getProperties();
      props.put("proxySet", "true");
      props.put("proxyHost", "cache.psu.ac.th");
      props.put("proxyPort", "8080");
      System.setProperties(props);

      // create a URL and URLConnection
      URL url  = new URL(args[1]);   // URL string
      URLConnection conn = url.openConnection();

      // encode the "login:password" string
      Base64Converter bc = new Base64Converter();
      String encoding = bc.encode( LOGIN + ":" + args[0] );

      // send the authorization
      conn.setRequestProperty("Proxy-Authorization",
                             "Basic " + encoding);

      BufferedReader br = new BufferedReader (
                new InputStreamReader ( conn.getInputStream() ));

      // print first ten lines of contents
      int numLine = 0;
      String line;
      while ( ((line = br.readLine()) != null) && (numLine <= 10) ) {
        System.out.println(line);
        numLine++;
      }
      if (line != null)
        System.out.println(". . .");

      br.close();
      System.exit(0);
  } // end of main()
} // end of GetWebPageP class
```

6.3. A Web-based Client and Server

These 'time of day' examples fit the familiar client/server model, but in the case when a server already exists. However, most applications require new clients *and* a new server. The question then is how to make the server-side of the program act as a Web server, deliver Web pages, and so satisfy the restrictions of the client-side firewall?

Enter J2EE, the *Java 2 Enterprise Edition*, aimed at the construction of Web-based client/server applications: it supports simplified networking, concurrency, transactions, easy access to databases, and much more (http://java.sun.com/j2ee/).

Aside from Sun's implementation, many companies offer J2EE compatible systems, including Tomcat from the Jakarta Project (http://jakarta.apache.org/tomcat/) and JRun from Macromedia (http://www.macromedia.com/software/jrun).

J2EE is a complex development environment, centered around servlets, Java Server Pages (JSPs), and Enterprise Java Beans (EJBs). Servlets are objects specialised for the serving of Web content, typically Web pages, in response to client requests. JSPs are Web pages which may contain embedded Java. EJBs focus on server-side processing, including the connection of server-side applications to other Java

 © Andrew Davison. 2003

functionality, such as JTA (the Java Transaction API), JMS (the Java Message Service), and JDBC (Java's database connectivity).

Servlets deal with client requests using the HTTP protocol, which thankfully only contains a few commands; the two principal ones are the GET method and POST method. A GET method (request) is usually sent by a Web browser when it asks for a page from a server. A POST method (request) is more typically associated with the submission of details taken from a Web page form.

A servlet which inherits the HttpServlet class will automatically call its doGet() method when a GET request arrives from a client; there is also a doPost() for processing POST requests.

Our Web-based client will communicate with a simple servlet which implements a 'time of day' service. The use of the HTTP protocol to 'bypass' firewall restrictions on client/server communication is called *HTTP Tunneling*.

TimeServlet will be called when the TimeClient application refers to the servlet's URL (i.e. sends a GET request to the Web server managing the servlet). The situation is illustrated by Figure 19.

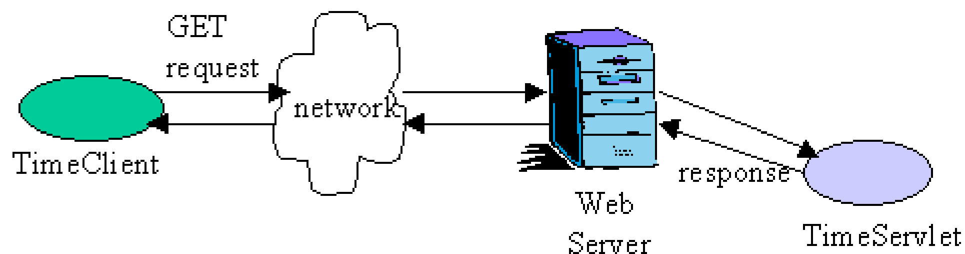

Figure 19. Client and Servlet Configuration.

A servlet can output a stream of HTML which is displayed nicely in the client, if it is a browser. However, TimeServlet will deliver ordinary text since the client does not require extraneous formatting around the result.

An excellent book on servlets and JSPs is:

> *Core Servlets and Java Server Pages*
> by Marty Hall, Sun Microsystems Press, 2001
> `http://www.coreservlets.com/`

The TimeServlet Servlet

The doGet() method in TimeServlet is called automatically when the servlet's URL is sent to the Web server.

```
public class TimeServlet extends HttpServlet
{
   public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
      throws ServletException, IOException
```

 © Andrew Davison. 2003

```
    {
      SimpleDateFormat formatter =
                  new SimpleDateFormat("d M yyyy HH:mm:ss");
      Date today = new Date();
      String todayStr = formatter.format(today);
      System.out.println("Today is: " + todayStr);

      PrintWriter output = response.getWriter();
      output.println( todayStr );  // send date & time
      output.close();
    }
}
```

Various client and request information is made available in the HttpServletRequest object passed to doGet(), but TimeServlet doesn't need it.

The other argument of doGet() is a HttpServletResponse object which permits various forms of output to be delivered to the client. TimeServlet creates an output stream and sends a formatted date and time.

The TimeClient Application

The TimeClient application is a simple variant of the GetWebPage program described earlier, except that the servlet's URL is hardwired into the code.

Since the client and servlet are running on the same machine, there is no need for proxy settings. The Web server is running at port 8080, and stores its servlets in a fixed location referred to by the URL "http://localhost:8080/servlet/".

```
public class TimeClient
{
   public static void main(String args[]) throws IOException
   {
     URL url  = new URL("http://localhost:8080/servlet/TimeServlet");
       BufferedReader br = new BufferedReader(
              new InputStreamReader( url.openStream() ));
     String line;
     while ( (line = br.readLine()) != null)
       System.out.println(line);
     br.close();
   }
}
```

 © Andrew Davison. 2003

Figure 20 shows the output from TimeClient.

```
D>java TimeClient
12 6 2546 13:20:51

D>java TimeClient
12 6 2546 13:20:53
```

Figure 20. TimeClient in Action.

An advantage of (simple) servlets is that they can also be tested from a browser. Figure 21 shows the output when http://localhost:8080/servlet/TimeServlet is typed into the Opera browser.

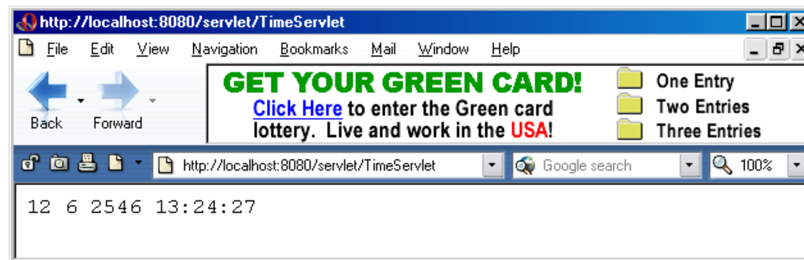

Figure 21. A Browser as Client for TimeServlet.

Servlets enable clients to use the HTTP protocol for communication with the server part of the application (i.e. the clients employ URLs to access the server). This mode of communication is permitted by firewalls, which usually block socket links.

The downside is that HTTP is purely a request/response protocol – the client must initiate the communication to receive a reply. It is not at all easy for the server to send a message to the client without first receiving one from it. This means that a Web-based server cannot easily broadcast (multicast) a message received from one client to all the others; a common requirement of multiplayer games and chat applications.

In chapter 19, we implement a Web-based version of a chat system by having a client periodically query the server to receive messages from the other clients.

6.4. Applets as Clients

The default security associated with applets means that they can only connect back to their home server. This restriction applies to sockets and to the downloading of Web pages. However, the security can be modified with a policy file and by signing the applet. However, most multiplayer games which utilize applets host them on their own servers, and so the default security policy is sufficient.

The real problem with applets is the excessive download time required to move the necessary functionality and resources (e.g. images, sounds) to the client side. Commercial games (e.g. Everquest) distribute CDs containing client applications.

 © Andrew Davison. 2003

7. Other Kinds of Java Networking

The networking support in Java is one of its greatest strengths. This section is a brief tour of some of its capabilities that have not previously been mentioned.

J2SDK 1.4 added support for secure sockets (using the SSL and TLS protocols). Security is a complex topic, but it is still fairly easy to do a common thing such as retrieve a Web page using the HTTPS protocol.

Remote Method Invocation (RMI) allows Java objects on separate machines to communicate via remote method calls. This is considerably higher-level than the transmission of data through sockets. RMI is based on a procedural programming technique, the Remote Procedure Call (RPC), but with some powerful extensions. One is dynamic code loading, which allows a client to download the communication code (called a stub) for accessing a remote method at run time. Code loading from clients can also be carried out by a server.

RMI is the basis of a number of expressive networking models, including Jini and JavaSpaces mentioned below. RMI is also integrated with the communications protocol for CORBA (the Common Object Request Broker Architecture) which permits Java objects to interact with objects coded in other languages.

Jini is a service discovery architecture which allows Jini-enabled clients to find and utilize whatever services are available on a network, dynamically adjusting their connections as new services come on-stream, and others leave. This is of key importance for mobile devices. The starting point for Jini is http://www.jini.org/.

A JavaSpaces service creates a shared space for its clients, where objects of any kind can be added, read, or removed. JavaSpaces makes it much easier to implement coordination models between groups of peers, where the interchange of data and tasks is represented by objects. The virtual space also provides persistence. More information can be found at http://java.sun.com/products/javaspaces/.

Java Shared Data Toolkit (JSDT) (http://java.sun.com/products/java-media/jsdt/) shares the same aim as JavaSpaces – to support collaborative applications. However, its basic abstract is a *session* – a group of objects associated with some common communications pattern. JSDT offers full-duplex multipoint communication among the participants, and multicasting.

© Andrew Davison. 2003