NETWORK PROGRAMMING

CHAPTER 6 : SOCKETS FOR CLIENTS

CHANDAN GUPTA BHAGAT

https://me.chandanbhagat.com.np

CONTENT

- Introduction to Sockets
- Using Sockets : Investigating Protocols with telnet, Reading from Server with Sockets, Writing to Servers with Sockets
- Constructing and Connecting Sockets : Basic Constructors, Picking a Local Interface to Connect From, Constructing without connecting, Server Addresses and Proxy Servers
- Getting Information about a Socket: Closed or Connected, toString()
- Setting Socket Options: TCP_NODELAY, SO_LINGER, SO_TIMEOUT, SO_RCVBUF and SO_SNDBUF, SO_KEEPALIVE, OOBINLINE, SO_REUSEADDR and IP_TOS Class of Service
- Sockets in GUI Applications: Whois and a Network Client Library

Introduction to Sockets

Introduction to Sockets

- Data is transmitted across the Internet in packets of finite size called *datagrams*. Each datagram contains a *header* and a *payload*.
- The header contains the address and port to which the packet is going, the address and port from which the packet came, a checksum to detect data corruption, and various other housekeeping information used to ensure reliable transmission. The payload contains the data itself.
- Datagrams have a finite length, it's often necessary to split the data across multiple packets and reassemble it at the destination. It's also possible that one or more packets may be lost or corrupted in transit and need to be retransmitted or that packets arrive out of order and need to be reordered.
- Keeping track of this, splitting the data into packets, generating headers, parsing the headers of incoming packets, keeping track of what packets have and haven't been received, and so on is a lot of work and requires a lot of intricate code.
- Sockets allow the programmer to treat a network connection as just another stream onto which bytes can be written and from which bytes can be read.
- Sockets shield the programmer from low-level details of the network, such as error detection, packet sizes, packet splitting, packet retransmission, network addresses, and more.

Using Sockets

Using Sockets

- A socket is a connection between two hosts. It can perform seven basic operations:
  - Connect to a remote machine
  - Send data
  - Receive data
  - Close a connection
  - Bind to a port
  - Listen for incoming data
  - Accept connections from remote machines on the bound port
- Java's Socket class, which is used by both clients and servers, has methods that correspond to the first four of these operations. The last three operations are needed only by servers, which wait for clients to connect to them. They are implemented by the ServerSocket class.

Using Sockets

- Java programs normally use client sockets in the following fashion
  - The program creates a new socket with a constructor.
  - The socket attempts to connect to the remote host.
- Once the connection is established, the local and remote hosts get input and output streams from the socket and use those streams to send data to each other. This connection is *full-duplex*. Both hosts can send and receive data simultaneously. What the data means depends on the protocol; different commands are sent to an FTP server than to an HTTP server. There will normally be some agreed-upon handshaking followed by the transmission of data from one to the other.
- When the transmission of data is complete, one or both sides close the connection. Some protocols, such as HTTP 1.0, require the connection to be closed after each request is serviced. Others, such as FTP and HTTP 1.1, allow multiple requests to be processed in a single connection

Investigating Protocols with Telnet

- The sockets themselves are simple enough; however, the protocols to communicate with different servers make life complex.
- To get a feel for how a protocol operates, you can use Telnet to connect to a server, type different commands to it, and watch its responses. By default, Telnet attempts to connect to port 23. To connect to servers on different ports, specify the port you want to connect to like this:

  $ **telnet localhost 25**

- This requests a connection to port 25, the SMTP port, on the local machine; SMTP is the protocol used to transfer email between servers or between a mail client and a server. If you know the commands to interact with an SMTP server, you can send email without going through a mail program.
- This trick can be used to forge email. For example, some years ago, the summer students at the National Solar Observatory in Sunspot, New Mexico, made it appear that the party one of the scientists was throwing after the annual volleyball match between the staff and the students was in fact a victory party for the students

Investigating Protocols with Telnet

- The interaction with the SMTP server went something like this; input the user types is shown in bold (the names have been changed to protect the gullible):

```
flare% telnet localhost 25
Trying 127.0.0.1 ...
Connected to localhost.sunspot.noao.edu.
Escape character is '^]'.
220 flare.sunspot.noao.edu Sendmail 4.1/SMI-4.1
ready at
Fri, 5 Jul 93 13:13:01 MDT
HELO sunspot.noao.edu
250 flare.sunspot.noao.edu Hello localhost
[127.0.0.1], pleased to meet you
MAIL FROM: bart
250 bart... Sender ok
RCPT TO: local@sunspot.noao.edu
250 local@sunspot.noao.edu... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
In a pitiful attempt to reingratiate myself with
the students after their inevitable defeat of the
staff on the volleyball court at 4:00 P.M., July
24, I will be throwing a victory party for the
students at my house that evening at 7:00.
Everyone is invited.

Beer and Ben-Gay will be provided so the staff
may drown their sorrows and assuage their aching
muscles after their public humiliation.

Sincerely,
Bart
.
250 Mail accepted
QUIT
221 flare.sunspot.noao.edu delivering mail
Connection closed by foreign host.
```

Investigating Protocols with Telnet

- In the 20 years since this happened, most SMTP servers have added a little more security than shown here. They tend to require usernames and passwords, and only accept connections from clients in the local networks and other trusted mail servers.
- It’s still the case that you can use Telnet to simulate a client, see how the client and the server interact, and thus learn what your Java program needs to do. Although this session doesn’t demonstrate all the features of the SMTP protocol, it’s sufficient to enable you to deduce how a simple email client talks to a server.

Reading from Servers with Sockets

- Let’s begin with a simple example. You’re going to connect to the daytime server at the National Institute for Standards and Technology (NIST) and ask it for the current time. This protocol is defined in RFC 867. Reading that, you see that the daytime server listens on port 13, and that the server sends the time in a human-readable format and closes the connection. You can test the daytime server with Telnet like this:

```
$ telnet time.nist.gov 13
Trying 129.6.15.28...
Connected to time.nist.gov.
Escape character is '^]'.
56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST) *
Connection closed by foreign host.
```

- The line “56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST)” is sent by the daytime server. When you read the Socket’s InputStream, this is what you will get. The other lines are produced either by the Unix shell or by the Telnet program.

Reading from Servers with Sockets

- RFC 867 does not specify any particular format for the output other than that it be human readable. In this case, you can see this connection was made on March 24, 2013, at 1:37: 50 P.M., Greenwich Meantime. More specifically, the format is defined as *JJJJJ YY-MM-DD HH:MM:SS TT L H msADV UTC(NIST) OTM* where:
  - *JJJJJ* is the "Modified Julian Date" (i.e., it is the number of whole days since midnight on November 17, 1858).
  - *YY-MM-DD* is the last two digits of the year, the month, and the current day of month.
  - *HH:MM:SS* is the time in hours, minutes, and seconds in Coordinated Universal Time (UTC, essentially Greenwich Mean Time).
  - *TT* indicates whether the United States is currently observing on Standard Time or Daylight Savings Time: 00 means standard time; 50 means daylight savings time. Other values count down the number of days until the switchover.
  - *L* is a one-digit code that indicates whether a leap second will be added or subtracted at midnight on the last day of the current month: 0 for no leap second, 1 to add a leap second, and 2 to subtract a leap second.
  - *H* represents the health of the server: 0 means healthy, 1 means up to 5 seconds off, 2 means more than 5 seconds off, 3 means an unknown amount of inaccuracy, and 4 is maintenance mode.
  - *msADV* is a number of milliseconds that NIST adds to the time it sends to roughly compensate for network delays. In the preceding code, you can see that it added 888.8 milliseconds to this result, because that's how long it estimates it's going to take for the response to return.
  - The string *UTC(NIST)* is a constant, and the *OTM* is almost a constant (an asterisk unless something really weird has happened).

Writing to Servers with Sockets

- Writing to a server is not noticeably harder than reading from one. You simply ask the socket for an output stream as well as an input stream. Although it's possible to send data over the socket using the output stream at the same time you're reading data over the input stream, most protocols are designed so that the client is either reading or writing over a socket, not both at the same time. In the most common pattern, the client sends a request. Then the server responds. The client may send another request, and the server responds again. This continues until one side or the other is done, and closes the connection.
- One simple bidirectional TCP protocol is *dict*, defined in RFC 2229. In this protocol, the client opens a socket to port 2628 on the dict server and sends commands such as "DEFINE eng-lat gold". This tells the server to send a definition of the word *gold* using its English-to-Latin dictionary. (Different servers have different dictionaries installed.) After the first definition is received, the client can ask for another. When it's done it sends the command "quit".

Writing to Servers with Sockets

- We can explore dict with telnet
- $ **telnet dict.org 2628**

  Trying 216.18.20.172...

  Connected to dict.org.

  Escape character is '^]'.

  220 pan.alephnull.com dictd 1.12.0/rf on Linux 3.0.0-14-server

  <auth.mime>

  <499772.29595.1364340382@pan.alephnull.com>

  DEFINE eng-lat gold

  150 1 definitions retrieved

  151 "gold" eng-lat "English-Latin Freedict dictionary"

  gold [gould]

  aurarius; aureus; chryseus

  aurum; chrysos

  .

  250 ok [d/m/c = 1/0/10; 0.000r 0.000u 0.000s]

  DEFINE eng-lat computer

  552 no match [d/m/c = 0/0/9; 0.000r 0.000u 0.000s]

  quit

  221 bye [d/m/c = 0/0/0; 42.000r 0.000u 0.000s]

Writing to Servers with Sockets

- We can explore dict.org with telnet
- $ **telnet dict.org 2628**

  Trying 216.18.20.172...

  Connected to dict.org.

  Escape character is '^]'.

  220 pan.alephnull.com dictd 1.12.0/rf on Linux 3.0.0-14-server

  <auth.mime>

  <499772.29595.1364340382@pan.alephnull.com>

  DEFINE eng-lat gold

  150 1 definitions retrieved

  151 "gold" eng-lat "English-Latin Freedict dictionary"

  gold [gould]

  aurarius; aureus; chryseus

  aurum; chrysos

  .

  250 ok [d/m/c = 1/0/10; 0.000r 0.000u 0.000s]

  DEFINE eng-lat computer

  552 no match [d/m/c = 0/0/9; 0.000r 0.000u 0.000s]

  quit

  221 bye [d/m/c = 0/0/0; 42.000r 0.000u 0.000s]

Writing to Servers with Sockets

- You can see that control response lines begin with a three-digit code. The actual definition is plain text, terminated with a period on a line by itself. If the dictionary doesn't contain the word you asked for, it returns 552 no match. Of course, you could also find this out, and a lot more, by reading the RFC.
- It's not hard to implement this protocol in Java. First, open a socket to a dict server _dict.org__ is a good one—on port 2628:

```
Socket socket = new Socket("dict.org", 2628);
```

- Once again you'll want to set a timeout in case the server hangs while you're connected to it:

```
socket.setSoTimeout(15000);
```

- In the dict protocol, the client speaks first, so ask for the output stream using getOutputStream():

```
OutputStream out = socket.getOutputStream();
```

Writing to Servers with Sockets

- The getOutputStream() method returns a raw OutputStream for writing data from your application to the other end of the socket. You usually chain this stream to a more convenient class like DataOutputStream or OutputStreamWriter before using it. For performance reasons, it's a good idea to buffer it as well. Because the dict protocol is text based, more specifically UTF-8 based, it's convenient to wrap this in a Writer:

```
Writer writer = new OutputStreamWriter(out, "UTF-8");
```

- Now write the command over the socket:

```
writer.write("DEFINE eng-lat gold\r\n");
```

- Finally, flush the output so you'll be sure the command is sent over the network:

```
writer.flush()
```

Writing to Servers with Sockets

- The server should now respond with a definition. You can read that using the socket's input stream:
  - ```
    InputStream in = socket.getInputStream();
    BufferedReader reader = new BufferedReader(
    new InputStreamReader(in, "UTF-8"));
    for (String line = reader.readLine();
    !line.equals(".");
    line = reader.readLine()) {
    System.out.println(line);
    }
    ```
- When you see a period on a line by itself, you know the definition is complete. You can then send the quit over the output stream:

```
writer.write("quit\r\n");
writer.flush();
```
```

Half-closed sockets

- The close() method shuts down both input and output from the socket. On occasion, you may want to shut down only half of the connection, either input or output. The shutdownInput() and shutdownOutput() methods close only half the connection:

```
public void shutdownInput() throws IOException
public void shutdownOutput() throws IOException
```

- Neither actually closes the socket. Instead, they adjust the stream connected to the socket so that it thinks it's at the end of the stream. Further reads from the input stream after shutting down input return –1. Further writes to the socket after shutting down output throw an IOException.

Half-closed sockets

- Many protocols, such as finger, whois, and HTTP, begin with the client sending a request to the server, then reading the response. It would be possible to shut down the output after the client has sent the request. For example, this code fragment sends a request to an HTTP server and then shuts down the output, because it won't need to write anything else over this socket:

```
try (Socket connection = new Socket("www.oreilly.com", 80)) {
    Writer out = new OutputStreamWriter(
     connection.getOutputStream(), "8859_1");
     out.write("GET / HTTP 1.0\r\n\r\n");
     out.flush();
     connection.shutdownOutput();
     // read the response...
} catch (IOException ex) {
    ex.printStackTrace();
}
```

- Notice that even though you shut down half or even both halves of a connection, you still need to close the socket when you're through with it. The shutdown methods simply affect the socket's streams. They don't release the resources associated with the socket, such as the port it occupies.

Constructing and Connecting Sockets

Basic Constructors

- Each Socket constructor specifies the host and the port to connect to. Hosts may be specified as an InetAddress or a String. Remote ports are specified as int values from 1 to 65535:

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

- These constructors connect the socket (i.e., before the constructor returns, an active network connection is established to the remote host). If the connection can't be opened for some reason, the constructor throws an IOException or an UnknownHostException. For example:

```
try {
Socket toOReilly = new Socket("www.oreilly.com", 80);
// send and receive data...
} catch (UnknownHostException ex) {
System.err.println(ex);
} catch (IOException ex) {
System.err.println(ex);
}
```

Basic Constructors

- Three constructors create unconnected sockets. These provide more control over exactly how the underlying socket behaves, for instance by choosing a different proxy server or an encryption scheme:

```
public Socket()
public Socket(Proxy proxy)
protected Socket(SocketImpl impl)
```

Picking a Local Interface to Connect From

- Two constructors specify both the host and port to connect *to* and the interface and port to connect *from*:

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

- This socket connects *to* the host and port specified in the first two arguments. It connects *from* the local network interface and port specified by the last two arguments.
- The network interface may be either physical (e.g., an Ethernet card) or virtual (a multihomed host with more than one IP address). If 0 is passed for the localPort argument, Java chooses a random available port between 1024 and 65535.

Picking a Local Interface to Connect From

- Selecting a particular network interface from which to send data is uncommon, but a need does come up occasionally. One situation where you might want to explicitly choose the local address would be on a router/firewall that uses dual Ethernet ports. Incoming connections would be accepted on one interface, processed, and forwarded to the local network from the other interface. Suppose you were writing a program to periodically dump error logs to a printer or send them over an internal mail server. You'd want to make sure you used the inward-facing network interface instead of the outward-facing network interface. For example:

```
try {
InetAddress inward = InetAddress.getByName("router");
Socket socket = new Socket("mail", 25, inward, 0);
// work with the sockets...
} catch (IOException ex) {
System.err.println(ex);
}
```

Picking a Local Interface to Connect From

- This constructor can throw an IOException or an UnknownHostException for the same reasons as the previous constructors. In addition, it throws an IOException (probably a BindException, although again that's just a subclass of IOException and not specifically declared in the throws clause of this method) if the socket is unable to bind to the requested local network interface. For instance, a program running on *a.example.com* can't connect from *b.example.org*

Constructing Without Connecting

- All the constructors we've talked about so far both create the socket object and open a network connection to a remote host. Sometimes you want to split those operations. If you give no arguments to the Socket constructor, it has nowhere to connect to:

```
public Socket()
```

- You can connect later by passing a SocketAddress to one of the connect() methods.

```
try {
Socket socket = new Socket();
// fill in socket options
SocketAddress address = new InetSocketAddress("time.nist.gov", 13);
socket.connect(address);
// work with the sockets...
} catch (IOException ex) {
System.err.println(ex);
}
```

Constructing Without Connecting

- You can pass an int as the second argument to specify the number of milliseconds to wait before the connection times out:
- `public void connect(SocketAddress endpoint, int timeout) throws IOException`

```
Socket socket = new Socket();
SocketAddress address = new InetSocketAddress(SERVER, PORT);
try {
   socket.connect(address);
   // work with the socket...
} catch (IOException ex) {
   System.err.println(ex);
} finally {
   try {
     socket.close();
   } catch (IOException ex) {
     // ignore
   }
}
```

Socket Addresses

- The SocketAddress class represents a connection endpoint. It is an empty abstract class with no methods aside from a default constructor. At least theoretically, the SocketAddress class can be used for both TCP and non-TCP sockets. In practice, only TCP/IP sockets are currently supported and the socket addresses you actually use are all instances of InetSocketAddress.
- The primary purpose of the SocketAddress class is to provide a convenient store for transient socket connection information such as the IP address and port that can be reused to create new sockets, even after the original socket is disconnected and garbage collected. To this end, the Socket class offers two methods that return SocketAddress objects (getRemoteSocketAddress() returns the address of the system being connected to and getLocalSocketAddress() returns the address from which the connection is made)

Socket Addresses

```
public SocketAddress getRemoteSocketAddress()
public SocketAddress getLocalSocketAddress()
```

- Both of these methods return null if the socket is not yet connected. For example, first you might connect to Yahoo! then store its address:

```
Socket socket = new Socket("www.yahoo.com", 80);
SocketAddress yahoo = socket.getRemoteSocketAddress();
socket.close();
```

- Later, you could reconnect to Yahoo! using this address:

```
Socket socket2 = new Socket();
socket2.connect(yahoo);
```

Socket Addresses

- The InetSocketAddress class (which is the only subclass of SocketAddress in the JDK, and the only subclass I've ever encountered) is usually created with a host and a port (for clients) or just a port (for servers):

```
public InetSocketAddress(InetAddress address, int port)
public InetSocketAddress(String host, int port)
public InetSocketAddress(int port)
```

- You can also use the static factory method InetSocketAddress.createUnresolved() to skip looking up the host in DNS:

```
public static InetSocketAddress createUnresolved(String host, int port)
```

- InetSocketAddress has a few getter methods you can use to inspect the object:

```
Public final InetAddress getAddress()
public final int getPort()
public final String getHostName()
```

Proxy Servers

- The last constructor creates an unconnected socket that connects through a specified proxy server:

```
public Socket(Proxy proxy)
```

- Normally, the proxy server a socket uses is controlled by the socksProxyHost and socksProxyPort system properties, and these properties apply to all sockets in the system. However, a socket created by this constructor will use the specified proxy server instead. Most notably, you can pass Proxy.NO_PROXY for the argument to bypass all proxy servers completely and connect directly to the remote host. Of course, if a firewall prevents direct connections, there's nothing Java can do about it; and the connection will fail.

```
SocketAddress proxyAddress = new InetSocketAddress("myproxy.example.com", 1080);
Proxy proxy = new Proxy(Proxy.Type.SOCKS, proxyAddress)
Socket s = new Socket(proxy);
SocketAddress remote = new InetSocketAddress("login.ibiblio.org", 25);
s.connect(remote);
```

- SOCKS is the only low-level proxy type Java understands. There's also a high-level Proxy.Type.HTTP that works in the application layer rather than the transport layer and a Proxy.Type.DIRECT that represents proxyless connections.

Getting Information About Sockets

Getting Information About Sockets

- Socket objects have several properties that are accessible through getter methods:
  - Remote address
  - Remote port
  - Local address
  - Local port
- Here are the getter methods for accessing these properties:

```
public InetAddress getInetAddress()
public int getPort()
public InetAddress getLocalAddress()
public int getLocalPort()
```

- There are no setter methods. These properties are set as soon as the socket connects, and are fixed from there on. The getInetAddress() and getPort() methods tell you the remote host and port the Socket is connected to; or, if the connection is now closed, which host and port the Socket was connected to when it was connected. The getLocalAddress() and getLocalPort() methods tell you the network interface and port the Socket is connected from.

Getting Information About Sockets

- Unlike the remote port, which (for a client socket) is usually a “well-known port” that has been preassigned by a standards committee, the local port is usually chosen by the system at runtime from the available unused ports. This way, many different clients on a system can access the same service at the same time. The local port is embedded in outbound IP packets along with the local host’s IP address, so the server can send data back to the right port on the client.

```
try {
Socket theSocket = new Socket(host, 80);
   System.out.println("Connected to " + theSocket.getInetAddress() + " on port " + theSocket.getPort() + " from port “  +

   theSocket.getLocalPort() + " of “ + theSocket.getLocalAddress());
} catch (UnknownHostException ex) {
   System.err.println("I can't find " + host);
} catch (SocketException ex) {
   System.err.println("Could not connect to " + host);
} catch (IOException ex) {
   System.err.println(ex);
}
```

Closed or Connected?

- The isClosed() method returns true if the socket is closed, false if it isn't. If you're uncertain about a socket's state, you can check it with this method rather than risking an IOException. For example:

```
if (socket.isClosed()) {
// do something...
}  else {
// do something else...
}
```

- However, this is not a perfect test. If the socket has never been connected in the first place, isClosed() returns false, even though the socket isn't exactly open. The Socket class also has an isConnected() method. The name is a little misleading. It does not tell you if the socket is currently connected to a remote host (like if it is unclosed). Instead, it tells you whether the socket has ever been connected to a remote host. If the socket was able to connect to the remote host at all, this method returns true, even after that socket has been closed. To tell if a socket is currently open, you need to check that isConnected() returns true and isClosed() returns false. For example:

```
boolean connected = socket.isConnected() && ! socket.isClosed();
```

Closed or Connected?

- Finally, the isBound() method tells you whether the socket successfully bound to the outgoing port on the local system. Whereas isConnected() refers to the remote end of the socket, isBound() refers to the local end.

toString()

- The Socket class overrides only one of the standard methods from java.lang.Object: toString(). The toString() method produces a string that looks like this:

  Socket[addr=www.oreilly.com/198.112.208.11,port=80,localport=50055]

- This is useful primarily for debugging. Don’t rely on this format; it may change in the future. All parts of this string are accessible directly through other methods (specifically getInetAddress(), getPort(), and getLocalPort()).

Setting Socket Options

Setting Socket Options

- Socket options specify how the native sockets on which the Java Socket class relies send and receive data. Java supports nine options for client-side sockets:
  - TCP_NODELAY
  - SO_BINDADDR
  - SO_TIMEOUT
  - SO_LINGER
  - SO_SNDBUF
  - SO_RCVBUF
  - SO_KEEPALIVE
  - OOBINLINE
  - IP_TOS

TCP_NODELAY

```
public void setTcpNoDelay(boolean on) throws SocketException
public boolean getTcpNoDelay() throws SocketException
```

- Setting TCP_NODELAY to true ensures that packets are sent as quickly as possible regardless of their size. Normally, small (one-byte) packets are combined into larger packets before being sent.
- Before sending another packet, the local host waits to receive acknowledgment of the previous packet from the remote system. This is known as *Nagle's algorithm*. The problem with Nagle's algorithm is that if the remote system doesn't send acknowledgments back to the local system fast enough, applications that depend on the steady transfer of small parcels of information may slow down. This issue is especially problematic for GUI programs such as games or network computer applications where the server needs to track client-side mouse movement in real time.
- On a really slow network, even simple typing can be too slow because of the constant buffering. Setting TCP_NODELAY to true defeats this buffering scheme, so that all packetsare sent as soon as they're ready. setTcpNoDelay(true) turns off buffering for the socket. setTcpNoDelay(false) turns it back on. getTcpNoDelay() returns true if buffering is off and false if buffering is on.

TCP_NODELAY

- For example, the following fragment turns off buffering (that is, it turns on TCP_NODELAY) for the socket s if it isn’t already off:

```
if (!s.getTcpNoDelay()) s.setTcpNoDelay(true);
```

- These two methods are each declared to throw a SocketException, which will happen if the underlying socket implementation doesn’t support the TCP_ NODELAY option.

SO_LINGER

```
public void setSoLinger(boolean on, int seconds) throws SocketException
public int getSoLinger() throws SocketException
```

- The SO_LINGER option specifies what to do with datagrams that have not yet been sent when a socket is closed. By default, the close() method returns immediately; but the system still tries to send any remaining data.
- If the linger time is set to zero, any unsent packets are thrown away when the socket is closed. If SO_LINGER is turned on and the linger time is any positive value, the close() method blocks while waiting the specified number of seconds for the data to be sent and the acknowledgments to be received. When that number of seconds has passed, the socket is closed and any remaining data is not sent, acknowledgment or no.
- The setSoLinger() method can also throw an IllegalArgumentException if you try to set the linger time to a negative value. However, the getSoLinger() method may return –1 to indicate that this option is disabled, and as much time as is needed is taken to deliver the remaining data;

SO_LINGER

- For example, to set the linger timeout for the Socket s to four minutes, if it's not already set to some other value:

```
if (s.getTcpSoLinger() == -1) s.setSoLinger(true, 240);
```

- The maximum linger time is 65,535 seconds, and may be smaller on some platforms. Times larger than that will be reduced to the maximum linger time. Frankly, 65,535 seconds (more than 18 hours) is much longer than you actually want to wait. Generally, the platform default value is more appropriate.

SO_TIMEOUT

```
public void setSoTimeout(int milliseconds) throws SocketException
public int getSoTimeout() throws SocketException
```

- Normally when you try to read data from a socket, the read() call blocks as long as necessary to get enough bytes. By setting SO_TIMEOUT, you ensure that the call will not block for more than a fixed number of milliseconds. When the timeout expires, an InterruptedIOException is thrown, and you should be prepared to catch it.
- However, the socket is still connected. Although this read() call failed, you can try to read from the socket again. The next call may succeed. Timeouts are given in milliseconds. Zero is interpreted as an infinite timeout; it is the default value.
- For example, to set the timeout value of the Socket object s to 3 minutes if it isn't already set, specify 180,000 milliseconds:

```
if (s.getSoTimeout() == 0) s.setSoTimeout(180000);
```

- These two methods each throw a SocketException if the underlying socket implementation does not support the SO_TIMEOUT option. The setSoTimeout() method also throws an IllegalArgumentException if the specified timeout value is negative.

SO_RCVBUF and SO_SNDBUF

- TCP uses buffers to improve network performance. Larger buffers tend to improve performance for reasonably fast (say, 10Mbps and up) connections whereas slower, dialup connections do better with smaller buffers. Generally, transfers of large, continuous blocks of data, which are common in file transfer protocols such as FTP and HTTP, benefit from large buffers, whereas the smaller transfers of interactive sessions, such as Telnet and many games, do not. Maximum achievable bandwidth equals buffer size divided by latency. For example, on Windows XP suppose the latency between two hosts is half a second (500 ms). Then the bandwidth is 17520 bytes / 0.5 seconds = 35040 bytes / second = 273.75 kilobits /second. That's the *maximum* speed of any socket, regardless of how fast the network is. That's plenty fast for a dial-up connection, and not bad for ISDN, but not really adequate for a DSL line or FIOS.
- You can increase speed by decreasing latency. On the other hand, you do control the buffer size. For example, if you increase the buffer size from 17,520 bytes to 128 kilobytes, the maximum bandwidth increases to 2 megabits per second. Double the buffer size again to 256 kilobytes, and the maximum bandwidth doubles to 4 megabits per second.
- Network itself has limits on maximum bandwidth. Set the buffer too high and your program will try to send and receive data faster than the network can handle, leading to congestion, dropped packets, and slower performance. Thus, when you want maximum bandwidth, you need to match the buffer size to the latency of the connection so it's a little less than the bandwidth of the network.

SO_RCVBUF and SO_SNDBUF

- The SO_RCVBUF option controls the suggested send buffer size used for network input. The SO_SNDBUF option controls the suggested send buffer size used for network output:

```
public void setReceiveBufferSize(int size) throws SocketException, IllegalArgumentException
public int getReceiveBufferSize() throws SocketException
public void setSendBufferSize(int size) throws SocketException, IllegalArgumentException
public int getSendBufferSize() throws SocketException
```

- Although it looks like you should be able to set the send and receive buffers independently, the buffer is usually set to the smaller of these two. For instance, if you set the send buffer to 64K and the receive buffer to 128K, you'll have 64K as both the send and receive buffer size. Java will report that the receive buffer is 128K, but the underlying TCP stack will really be using 64K.

- The setReceiveBufferSize()/setSendBufferSize methods suggest a number of bytes to use for buffering output on this socket. If you attempt to set a larger value, Java will just pin it to the maximum possible buffer size. On Linux, it's not unheard of for the underlying implementation to double the requested size. For example, if you ask for a 64K buffer, you may get a 128K buffer instead.

SO_RCVBUF and SO_SNDBUF

- These methods throw an IllegalArgumentException if the argument is less than or equal to zero. Although they're also declared to throw SocketException, they probably won't in practice, because a SocketException is thrown for the same reason as IllegalArgumentException and the check for the IllegalArgumentException is made first.
- In general, if you find your application is not able to fully utilize the available bandwidth (e.g., you have a 25 Mbps Internet connection, but your data is transferring at a piddling 1.5 Mbps) try increasing the buffer sizes. By contrast, if you're dropping packets and experiencing congestion, try decreasing the buffer size. However, most of the time, unless you're really taxing the network in one direction or the other, the defaults are fine. In particular, modern operating systems use TCP window scaling (not controllable from Java) to dynamically adjust buffer sizes to fit the network. As with almost any performance tuning advice, the rule of thumb is not to do it until you've measured a problem. And even then you may well get more speed by increasing the maximum allowed buffer size at the operating system level than by adjusting the buffer sizes of individual sockets.

SO_KEEPALIVE

- If SO_KEEPALIVE is turned on, the client occasionally sends a data packet over an idle connection (most commonly once every two hours), just to make sure the server hasn't crashed. If the server fails to respond to this packet, the client keeps trying for a little more than 11 minutes until it receives a response. If it doesn't receive a response within 12 minutes, the client closes the socket. Without SO_KEEPALIVE, an inactive client could live more or less forever without noticing that the server had crashed. These methods turn SO_KEEPALIVE on and off and determine its current state:

```
public void setKeepAlive(boolean on) throws SocketException
public boolean getKeepAlive() throws SocketException
```

- The default for SO_KEEPALIVE is false. This code fragment turns SO_KEEPALIVE off, if it's turned on:

```
if (s.getKeepAlive()) s.setKeepAlive(false);
```

OOBINLINE

- TCP includes a feature that sends a single byte of “urgent” data out of band. This data is sent immediately. Furthermore, the receiver is notified when the urgent data is received and may elect to process the urgent data before it processes any other data that has already been received. Java supports both sending and receiving such urgent data. The sending method is named, obviously enough, sendUrgentData():

```
public void sendUrgentData(int data) throws IOException
```

- This method sends the lowest-order byte of its argument almost immediately. If necessary, any currently cached data is flushed first. How the receiving end responds to urgent data is a little confused, and varies from one platform and API to the next. Some systems receive the urgent data separately from the regular data. However, the more common, more modern approach is to place the urgent data in the regular received data queue in its proper order, tell the application that urgent data is available, and let it hunt through the queue to find it.
- By default, Java ignores urgent data received from a socket. However, if you want to receive urgent data inline with regular data, you need to set the OOBINLINE option to true using these methods:

OOBINLINE

```
public void setOOBInline(boolean on) throws SocketException
public boolean getOOBInline() throws SocketException
```

- The default for OOBINLINE is false. This code fragment turns OOBINLINE on, if it's turned off:

```
if (!s.getOOBInline()) s.setOOBInline(true);
```

- Once OOBINLINE is turned on, any urgent data that arrives will be placed on the socket's input stream to be read in the usual way. Java does not distinguish it from nonurgent data. That makes it less than ideally useful, but if you have a particular byte (e.g., a Ctrl-C) that has special meaning to your program and never shows up in the regular data stream, then this would enable you to send it more quickly.

IP_TOS Class of Service

- Different types of Internet service have different performance needs. For instance, video chat needs relatively high bandwidth and low latency for good performance, whereas email can be passed over low-bandwidth connections and even held up for several hours without major harm. VOIP needs less bandwidth than video but minimum jitter. It might be wise to price the different classes of service differentially so that people won't ask for the highest class of service automatically. After all, if sending an overnight letter cost the same as sending a package via media mail, we'd all just use FedEx overnight, which would quickly become congested and overwhelmed. The Internet is no different. The class of service is stored in an eight-bit field called IP_TOS in the IP header. Java lets you inspect and set the value a socket places in this field using these two methods:

```
public int getTrafficClass() throws SocketException
public void setTrafficClass(int trafficClass) throws SocketException
```

- The traffic class is given as an int between 0 and 255. Because this value is copied to an eight-bit field in the TCP header, only the low order byte of this int is used; and values outside this range cause IllegalArgumentExceptions.

Sockets in GUI Applications

Sockets in GUI Applications

- The HotJava web browser was the first large-scale Java GUI network client. HotJava has been discontinued, but there are still numerous network-aware client applications written in Java, including the Eclipse IDE and the Frostwire BitTorrent client. It is completely possible to write commercial-quality client applications in Java; and it is especially possible to write network-aware applications, both clients and servers. This section demonstrates a network client, whois, to illustrate this point; and to discuss the special considerations that arise when integrating networking code with Swing applications.
- The example stops short of what could be done, but only in the user interface. All the necessary networking code is present. Indeed, once again you find out that network code is easy; it’s user interfaces that are hard.

Whois

- Whois is a simple directory service protocol defined in RFC 954; it was originally designed to keep track of administrators responsible for Internet hosts and domains. A whois client connects to one of several central servers and requests directory information for a person or persons; it can usually give you a phone number, an email address, and a snail mail address (not necessarily current ones, though). With the explosive growth of the Internet, flaws have become apparent in the whois protocol, most notably its centralized nature. A more complex replacement called whois++ is documented in RFCs 1913 and 1914 but has not been widely implemented.whois directory service protocol)
- Let's begin with a simple client to connect to a whois server. The basic structure of the whois protocol is:
    1. The client opens a TCP socket to port 43 on the server.
    2. The client sends a search string terminated by a carriage return/linefeed pair (\r\n). The search string can be a name, a list of names, or a special command, as discussed shortly. You can also search for domain names, like *www.oreilly.com* or *netscape.com*, which give you information about a network.
    3. The server sends an unspecified amount of human-readable information in response to the command and closes the connection.
    4. The client displays this information to the user.

Whois

- $ **telnet whois.internic.net 43**

  Trying 199.7.50.74...

  Connected to whois.internic.net.

  Escape character is '^]'.

  Harold

  Whois Server Version 2.0

  Domain names in the .com and .net domains can now be registered

  with many different competing registrars. Go to

  http://www.internic.net

  for detailed information.

  HAROLD.LUCKYLAND.ORG

  HAROLD.FRUGAL.COM

  HAROLD.NET

  HAROLD.COM

  To single out one record, look it up with "xxx", where xxx is one of the

  of the records displayed above. If the records are the same, look them up

  with "=xxx" to receive a full display for each record.

  >>> Last update of whois database: Sat, 30 Mar 2013 15:15:05 UTC <<<

  ...

  Connection closed by foreign host.

Whois

% **telnet whois.nic.fr 43**

telnet whois.nic.fr 43

Trying 192.134.4.18...

Connected to winter.nic.fr.

Escape character is '^]'.

Harold

Tous droits reserves par copyright.

Voir http://www.nic.fr/outils/dbcopyright.html

Rights restricted by copyright.

See http://www.nic.fr/outils/dbcopyright.html

person: Harold Potier

address: ARESTE

address: 154 Avenue Du Brezet

address: 63000 Clermont-Ferrand

address: France

phone: +33 4 73 42 67 67

fax-no: +33 4 73 42 67 67

nic-hdl: HP4305-FRNIC

mnt-by: OLEANE-NOC

changed: hostmaster@oleane.net 20000510

changed: migration-dbm@nic.fr 20001015

source: FRNIC

person: Harold Israel

address: LE PARADIS LATIN

address: 28 rue du Cardinal Lemoine

address: Paris, France 75005 FR

phone: +33 1 43252828

fax-no: +33 1 43296363

e-mail: info@cie.fr

nic-hdl: HI68-FRNIC

notify: info@cie.fr

changed: registrar@ns.il 19991011

changed: migration-dbm@nic.fr 20001015

source: FRNIC

Whois prefixes

Prefix	Meaning
Domain	Find only domain records.
Gateway	Find only gateway records.
Group	Find only group records.
Host	Find only host records.
Network	Find only network records.
Organization	Find only organization records.
Person	Find only person records.
ASN	Find only autonomous system number records.
Handle or !	Search only for matching handles.
Mailbox or @	Search only for matching email addresses.
Name or :	Search only for matching names.
Expand or *	Search only for group records and show all individuals in that group.
Full or =	Show complete record for each match.
Partial or suffix	Match records that start with the given string.
Summary or $	Show just the summary, even if there’s only one match.

A Network Client Library

- It’s best to think of network protocols like whois in terms of the bits and bytes that move across the network, whether as packets, datagrams, or streams. No network protocol neatly fits into a GUI (with the arguable exception of the Remote Framebuffer Protocol used by VNC and X11). It’s usually best to encapsulate the network code into a separate library that the GUI code can invoke as needed.

THANK YOU

YOU CAN FIND THIS SLIDES/NOTES IN

https://chandanbhagat.com.np/docs/network-programming/

AND CODES IN

https://github.com/chandan-g-bhagat/network-programming