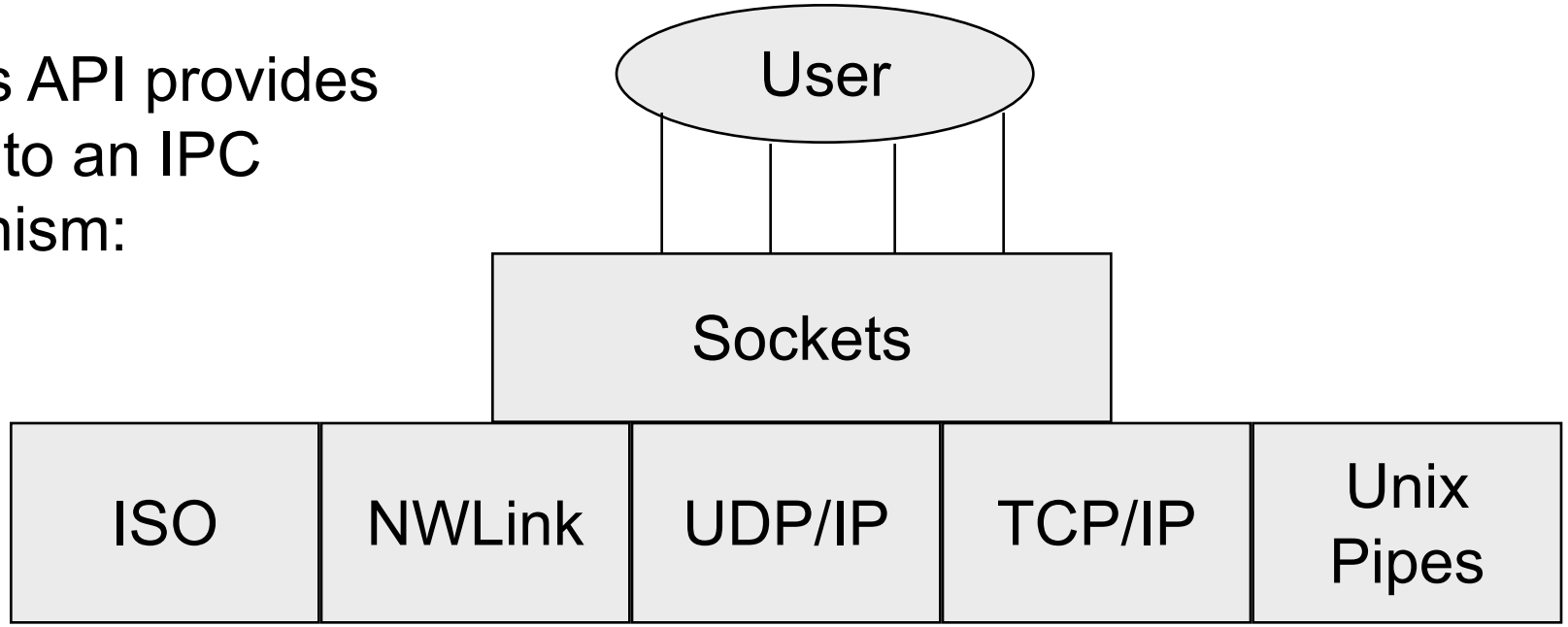
Sockets API

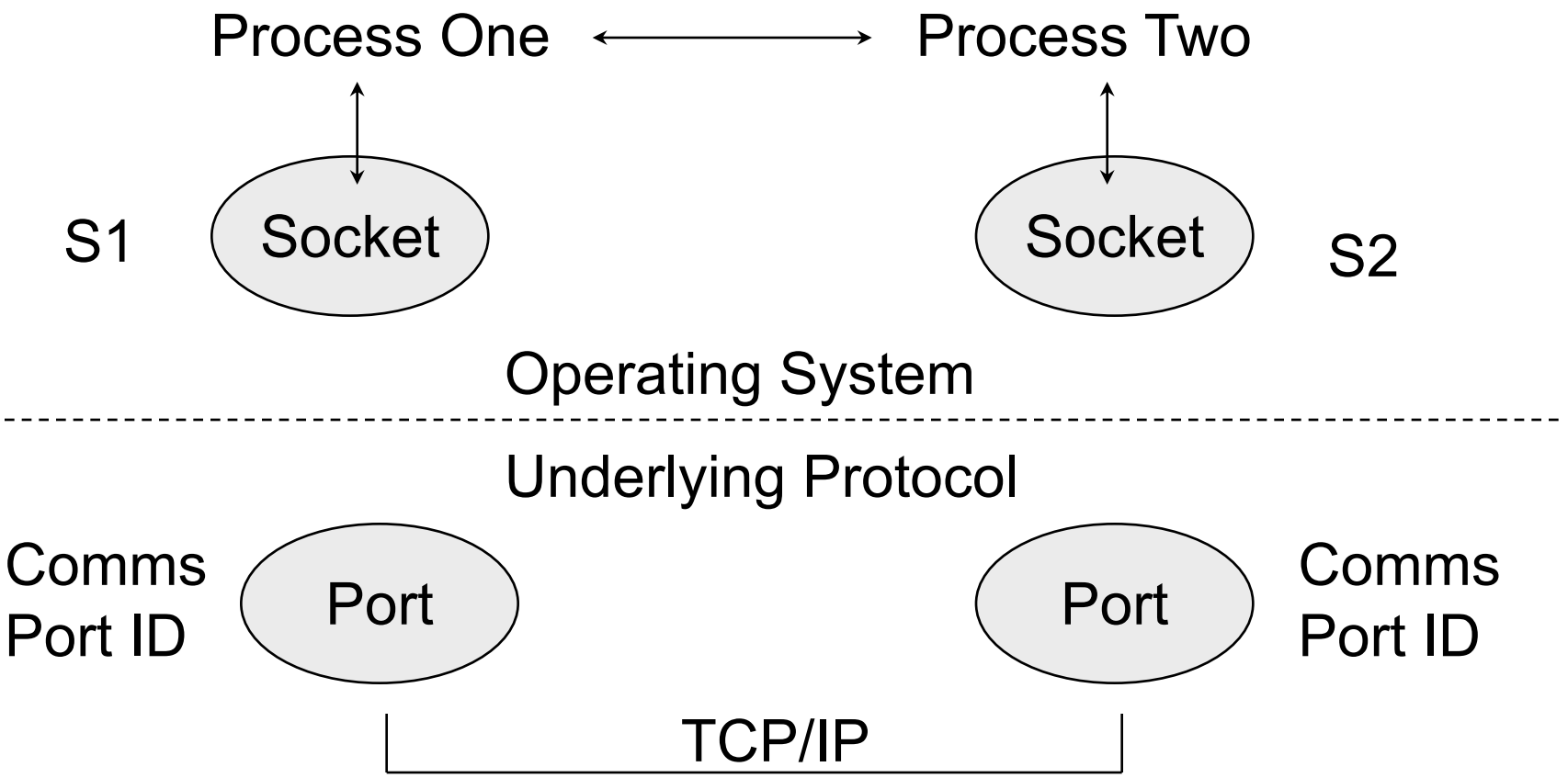
- Sockets Abstraction
  - Sockets were developed as a BSD Unix system abstraction to allow users to write programs that utilise underlying comms protocols.
  - Unrelated processes can communicate with each other (across the network).
  - *Endpoint* of communication to which a *name* may be bound.
  - Allow developer to write programs that are *largely* independent of lower protocols.

Sockets API

Sockets API provides access to an IPC mechanism:



Sockets API



Sockets API

- Two Sockets are required for end-to-end communication.
- Biased towards Client / Server Model.
- Sockets must be:
  - **Created** at both ends (specifying protocol type).
  - **Bind**ed to a Local Address (Named).
  - Optionally **Connected** to a (remote) socket.
- A socket identifier is returned:
  - This is similar to a file ID and they share many operations.
  - Passed to forked child processes.

Sockets API

- Types of sockets
  - In BSD Unix, and the systems derived from it, socket functions are part of the OS itself.
  - As usage increased, other vendors decided to add a Sockets API to their systems.
  - Often this was in the form of a *sockets library* that provides the Sockets API layered above an underlying set of native *system calls*.
  - In practice, Socket libraries are seldom perfect ! Minor differences sometimes occur e.g. in the way errors are handled.

JAVA Sockets

- Networking package is **java.net**
  - » Socket-based communications
    - – Applications view networking as streams of data
    - – Connection-based protocol
    - – Uses TCP (Transmission Control Protocol
  - » Packet-based communications
    - – Individual packets transmitted
    - – Connectionless service
    - – Uses UDP (User Datagram Protocol)

JAVA Sockets

- Sockets in Java
  - A socket is one end-point of a two-way communication link between two programs running on the network.
  - Socket classes are used to represent the connection between a client program and a server program.
  - The java.net package provides two classes - *Socket* and *ServerSocket*:
    - These implement the client side of the connection and the server side of the connection, respectively.

JAVA Sockets

```
// Network server that echoes text messages
// back to the client…

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

class server {
    public static void main(String a[]) throws
    IOException {
        int timeoutsecs = 600;
        int port = 4444;
        Socket sock;
```

JAVA Sockets

```
ServerSocket servsock = new
        ServerSocket(port, timeoutsecs);

while (true) {
    // wait for the next client connection
    sock=servsock.accept();

   // Get I/O streams from the socket
   PrintStream out = new PrintStream(
        sock.getOutputStream() );
   DataInputStream in  = new
      DataInputStream(sock.getInputStream());
```

JAVA Sockets

```
            // get the text string from the client
            String text = in.readLine();

            // Echo it back to the client again
            out.println(text);
            out.flush();   // This is optional
            // Close this connection, (not the overall server socket)
             sock.close();
          }  // Loop and accept the next client
        }
    }
```

JAVA Sockets

```
// This is the Client ...

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

class client {

    public static void main(String a[]) throws
    IOException {
        Socket sock;
        DataInputStream dis;
        PrintStream dat;
```

JAVA Sockets

```
// Open our connection to dcham, at port 4444
// If you try this on your system, insert your system
// in place of "dcham" - "dcham.nuigalway.ie" is my
// system name.
sock = new Socket("dcham",4444);

// Get I/O streams from the socket
dis = new DataInputStream( sock.getInputStream() );
dat = new PrintStream( sock.getOutputStream() );
```

JAVA Sockets

```
            dat.println(”Hello World!”);
            dat.flush();

            String fromServer = dis.readLine();
            System.out.println("Got this from server:" +
                    fromServer);

              sock.close();
        }
    }
```

Connectionless Client/Server

» Connectionless transmission with datagrams
  – No connection maintained with other computer
  – Break message into equal sized pieces and send as packets
  – Message arrive in order, out of order or not at all
  – Receiver puts messages in order and reads them

```
1    // Fig. 17.6: Server.java
2    // Set up a Server that will receive packets from a
3    // client and send packets to a client.
4    // Java core packages
6    import java.io.*;
7    import java.net.*;
8    import java.awt.*;
9    import java.awt.event.*;
10
11   // Java extension packages
12   import javax
5.swing.*;
13
14   public class Server extends JFrame {
15      private JTextArea displayArea;
16      private DatagramPacket sendPacket, receivePacket;
17      private DatagramSocket socket;
18
19      // set up GUI and DatagramSocket
20      public Server()
21      {
22         super( "Server" );
23
24         displayArea = new JTextArea();
25         getContentPane().add( new JScrollPane( displayArea ),
26            BorderLayout.CENTER );
27         setSize( 400, 300 );
28         setVisible( true );
29
30         // create DatagramSocket for sending and receiving packets
31         try {
32            socket = new DatagramSocket( 5000 );
33         }
34
```

Constructor creates GUI

Create **DatagramSocket** at port 5000

```
35          // process problems creating DatagramSocket
36          catch( SocketException socketException ) {
37             socketException.printStackTrace();
38             System.exit( 1 );
39          }
40
41       }  // end Server constructor
42
43       // wait for packets to arrive, then display data and echo
44       // packet to client
45       public void waitForPackets()
46       {
47          // loop forever
48          while ( true ) {
49
50             // receive packet, display contents, echo to client
51             try {
52
53                // set up packet
54                byte data[] = new byte[ 100 ];
55                receivePacket =
56                   new DatagramPacket( data, data.length );
57
58                // wait for packet
59                socket.receive( receivePacket );
60
61                // process packet
62                displayPacket();
63
64                // echo information from packet back to client
65                sendPacketToClient();
66             }
67
```

Method **waitForPackets** uses an infinite loop to wait for packets to arrive

Create a **DatagramPacket** to store received information

Method **receive** blocks until a packet is received

```
68            // process problems manipulating packet
69            catch( IOException ioException ) {
70               displayArea.append( ioException.toString() + "\n" );
71               ioException.printStackTrace();
72            }
73
74         }  // end while
75
76      }  // end method waitForPackets
77
78      // display packet contents
79      private void displayPacket()
80      {
81         displayArea.append( "\nPacket received:" +
82            "\nFrom host: " + receivePacket.getAddress() +
83            "\nHost port: " + receivePacket.getPort() +
84            "\nLength: " + receivePacket.getLength() +
85            "\nContaining:\n\t" +
86            new String( receivePacket.getData(), 0,
87               receivePacket.getLength() ) );
88      }
89
90      // echo packet to client
91      private void sendPacketToClient() throws IOException
92      {
93         displayArea.append( "\n\nEcho data to client..." );
94
95         // create packet to send
96         sendPacket = new DatagramPacket( receivePacket.getData(),
97            receivePacket.getLength(), receivePacket.getAddress(),
98            receivePacket.getPort() );
99
100        // send packet
101        socket.send( sendPacket );
```

Method **displayPacket** appends packet's contents to **displayArea**

Method **getAddress** returns name of computer that sent packet

Method **getPort** returns the port the packet came through

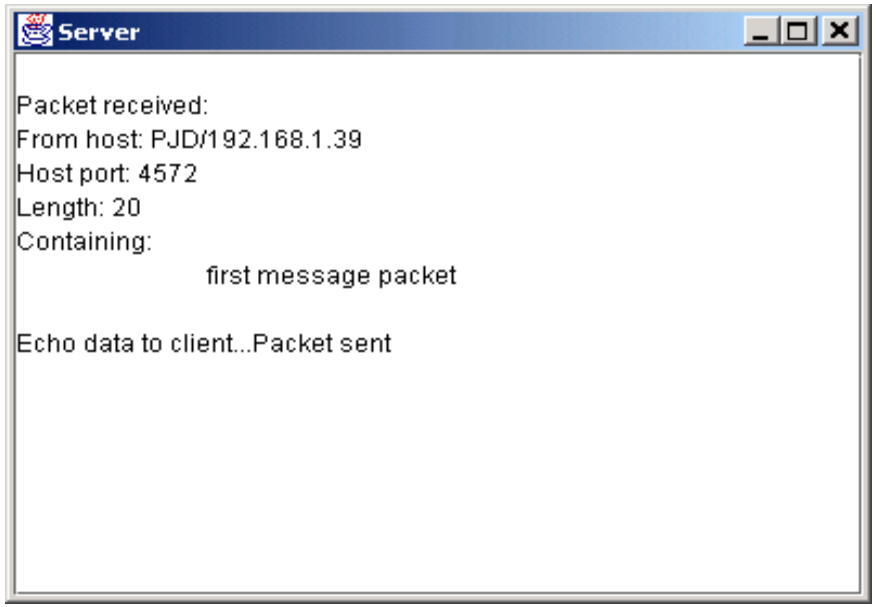
Method **getLength** returns the length of the message sent

Method **getData** returns a byte array containing the sent data

Method **send** sends the pack over the network

```
102
103        displayArea.append( "Packet sent\n" );
104        displayArea.setCaretPosition(
105           displayArea.getText().length() );
106     }
107
108     // execute application
109     public static void main( String args[] )
110     {
111        Server application = new Server();
112
113        application.setDefaultCloseOperation(
114           JFrame.EXIT_ON_CLOSE );
115
116        application.waitForPackets();
117     }
118
119  }  // end class Server
```

Method **main** creates a new server and waits for packets



```
1    // Fig. 17.7: Client.java
2    // Set up a Client that will send packets to a
3    // server and receive packets from a server.
4
5    // Java core packages
6    import java.io.*;
7    import java.net.*;
8    import java.awt.*;
9    import java.awt.event.*;
10
11   // Java extension packages
12   import javax.swing.*;
13
14   public class Client extends JFrame {
15      private JTextField enterField;
16      private JTextArea displayArea;
17      private DatagramPacket sendPacket, receivePacket;
18      private DatagramSocket socket;
19
20      // set up GUI and DatagramSocket
21      public Client()
22      {
23         super( "Client" );
24
25         Container container = getContentPane();
26
27         enterField = new JTextField( "Type message here" );
28
```

Constructor sets up GUI and **`DatagramSocket`** object

```
29          enterField.addActionListener(
30
31             new ActionListener() {
32
33                // create and send a packet
34                public void actionPerformed( ActionEvent event )
35                {
36                   // create and send packet
37                   try {
38                      displayArea.append(
39                         "\nSending packet containing: " +
40                         event.getActionCommand() + "\n" );
41
42                      // get message from textfield and convert to
43                      // array of bytes
44                      String message = event.getActionCommand();
45                      byte data[] = message.getBytes();
46
47                      // create sendPacket
48                      sendPacket = new DatagramPacket(
49                         data, data.length,
50                         InetAddress.getLocalHost(), 5000 );
51
52                      // send packet
53                      socket.send( sendPacket );
54
55                      displayArea.append( "Packet sent\n" );
56                      displayArea.setCaretPosition(
57                         displayArea.getText().length() );
58                   }
59
```

Method **actionPerformed** converts a **String** to a **byte** array to be sent as a datagram

Convert the **String** to a **byte** array

Create the **DatagramPacket** to send

Send the packet with method **send**

```
60                   // process problems creating or sending packet
61                   catch ( IOException ioException ) {
62                      displayArea.append(
63                         ioException.toString() + "\n" );
64                      ioException.printStackTrace();
65                   }
66
67                }  // end actionPerformed
68
69             }  // end anonymous inner class
70
71          ); // end call to addActionListener
72
73          container.add( enterField, BorderLayout.NORTH );
74
75          displayArea = new JTextArea();
76          container.add( new JScrollPane( displayArea ),
77             BorderLayout.CENTER );
78
79          setSize( 400, 300 );
80          setVisible( true );
81
82          // create DatagramSocket for sending and receiving packets
83          try {
84             socket = new DatagramSocket();
85          }
86
87          // catch problems creating DatagramSocket
88          catch( SocketException socketException ) {
89             socketException.printStackTrace();
90             System.exit( 1 );
91          }
92
93       }  // end Client constructor
94
```

Create **`DatagramSocket`** for sending and receiving packets

```
95      // wait for packets to arrive from Server,
96      // then display packet contents
97      public void waitForPackets()
98      {
99         // loop forever
100        while ( true ) {
101
102           // receive packet and display contents
103           try {
104
105              // set up packet
106              byte data[] = new byte[ 100 ];
107              receivePacket =
108                 new DatagramPacket( data, data.length );
109
110              // wait for packet
111              socket.receive( receivePacket );
112
113              // display packet contents
114              displayPacket();
115           }
116
117           // process problems receiving or displaying packet
118           catch( IOException exception ) {
119              displayArea.append( exception.toString() + "\n" );
120              exception.printStackTrace();
121           }
122
123        }  // end while
124
125     }  // end method waitForPackets
126
```

Method **`waitForPackets`** uses an infinite loop to wait for packets from server

Block until packet arrives

Display contents of packet

```
127     // display contents of receivePacket
128     private void displayPacket()
129     {
130        displayArea.append( "\nPacket received:" +
131           "\nFrom host: " + receivePacket.getAddress() +
132           "\nHost port: " + receivePacket.getPort() +
133           "\nLength: " + receivePacket.getLength() +
134           "\nContaining:\n\t" +
135           new String( receivePacket.getData(), 0,
136              receivePacket.getLength() ) );
137
138        displayArea.setCaretPosition(
139           displayArea.getText().length() );
140     }
141
142     // execute application
143     public static void main( String args[] )
144     {
145        Client application = new Client();
146
147        application.setDefaultCloseOperation(
148           JFrame.EXIT_ON_CLOSE );
149
150        application.waitForPackets();
151     }
152
153  }  // end class Client
```

Method **displayPacket** displays packet contents in **JTextArea**

Program Output