COMP90015 Distributed Systems

Interprocess Communication

Aaron Harwood

School of Computing and Information Systems

2022 Semester II

1. Abstract IPC
   - Data Exchange
   - Mechanisms of IPC
   - Communication Protocols and Interactions
   - Data Representation
2. Socket Paradigm
   - Network Communication
   - Network Address
   - Stream
   - Datagram

Application/User-level Data Exchange

IPC is the exchange of data between processes.

- Processes arrange data into data structures, often leading to a quite complex and significantly large aggregate structure representing an application's state.
- Data must be transferred from persistent storage, or files, into memory in order for the process to work on it, and results must be transferred back to the persistent storage if they are to be available outside of the process lifetime.
- Ostensibly, different processes can exchange data by reading/writing to the same files, and this is indeed a simple and ubiquitous approach:
  - file locking can be used for concurrency control,
  - if a distributed file system is available (discussed in later lectures) then the same file can be accessible from processes on different machines,
  - vastly different applications can exchange data in this way, e.g. a Word Processor can exchange data with an Email Client,
  - OS user interfaces often provide user-level mechanisms for exchange of data between processes, e.g. cut-and-paste between applications.

However these high-level, somewhat ad-hoc IPC approaches can be grossly inefficient, sometimes ill-defined and downright dangerous to make use of, compared to functionality specifically provided by the OS to undertake IPC.

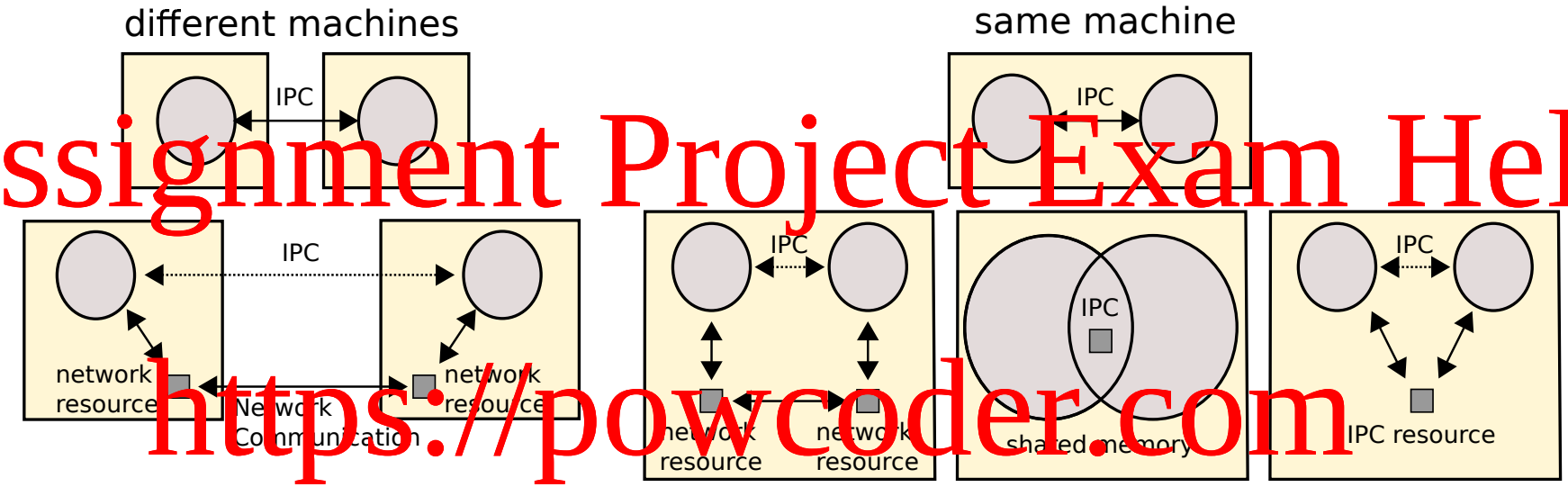
Data Format

Whether high-level data is exchanged in ad-hoc ways or using specific IPC mechanisms, the data format and more specifically the data structures and representations used by the application process, as well as the application or user requirements that drive the need for data exchange in the first place, can significantly influence the choice of IPC and the design of the distributed system.

- file transfer – must be reliable and high throughput
- data entry, databases – high iops and throughput
- audio and video streaming – high throughput and high quality

- instant chat, voice and video conferencing – interactive and high quality
- collaborative document editing – complex interactions
- remote user interface and control – responsive, light weight
- online games – low latency, responsive

The availabilty of IPC mechanisms is determined by the process locations



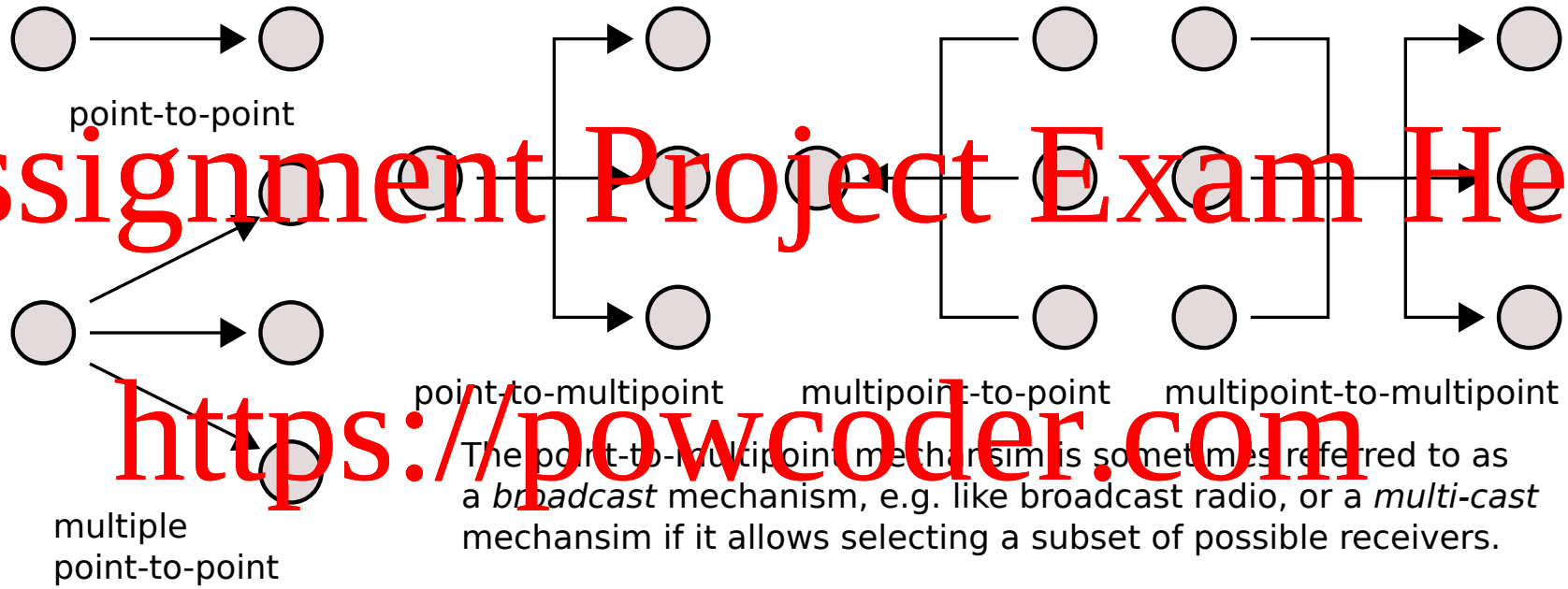
- Unlike threads within a process that share the same address space and can refer to the same data, the address space of processes are initially exclusive and for processes to refer to the same data they must communicate the data to each other somehow.
  - Network-based IPC can take place between processes on different machines and also between processes on the same machine – processes need not know if they are local or remote to each other.
  - Shared Memory IPC can only take place between processes on the same machine – it is the fastest form of IPC, much like thread interaction within a process.
  - Other forms of IPC resources like Pipes and Memory Mapped IO are also only available between processes on the same machine – they are faster than network-based IPC.
  - Shared Memory, Pipes and Memory Mapped IO are usually quite OS specific: we can study them later as an advanced topic.

Open Systems Interconnection (OSI) model

With respect to Network IPC, we consider distributed systems that are based upon functionality associated with the Transport Layer of the OSI model. The Transport Layer provides mechanisms for host-to-host or point-to-point interprocess communication over a network. At the Session Layer we make use of these mechanisms to implement long-running communication protocols that support the requirements of the Application Layer. In between in the Presentation Layer which provides data representations of Application Layer complex data structures in a form suitable for the Session Layer protocols.

- **Application Layer**: Fundamental distributed applications like Web Servers and Email, supporting services like the Domain Name Service (DNS) and the Network Time Protocol (NTP), and middleware like message queues and publish/subscribe systems.
- **Presentation Layer**: Data representations of complex data structures.
- **Session Layer**: Long-running communication protocols to support application requirements.
- **Transport Layer**: Host-to-Host communication services for applications; primarily TCP and UDP.
- Network Layer: Packet forwarding and routing through the network.
- Data Link Layer: Packet transmission from one device to another.
- Physical Layer: Data transmission over a communication channel.

Multi-party Communication



- Usually we consider IPC to be between 2 processes/parties: *point-to-point*.
- Communication between 3 or more parties is usually considered as multiple, independent point-to-point communications.
- However multi-party communication mechanisms are sometimes available that can be more efficient than using multiple point-to-point IPCs:
  - UDP provides *point-to-multipoint* communication, where a process can send a single packet, replicated by the network as required, and delivered to multiple processes. Multiple point-to-point would require the sending process to send the same packet multiple times.
  - Multipoint-to-point and multipoint-to-multipoint are less common. In this case the IPC mechanism must provide some kind of aggregation of communication that is more efficient that using multiple point-to-point or point-to-multipoint IPCs.

Discussion questions

Question (1): What kind of application functionality do you think would be suitable for multipoint-to-point and/or multipoint-to-multipoint IPC mechanisms? For the functionality that you discussed, without such multipoint IPC mechanisms, how many individual messages would need to be sent using multiple point-to-point communications to achieve the same functionality? What is lacking from the multiple point-to-point solution?

Question (2): Wireless broadcast for point-to-multipoint IPC uses something like a wireless basestation or a radio transmitter to transmit a signal that is received (practically) simultaneously by all receivers. Such a wireless broadcast mechanism cannot provide multi-cast in this situation. Why not? In what situations could it be provided if any?

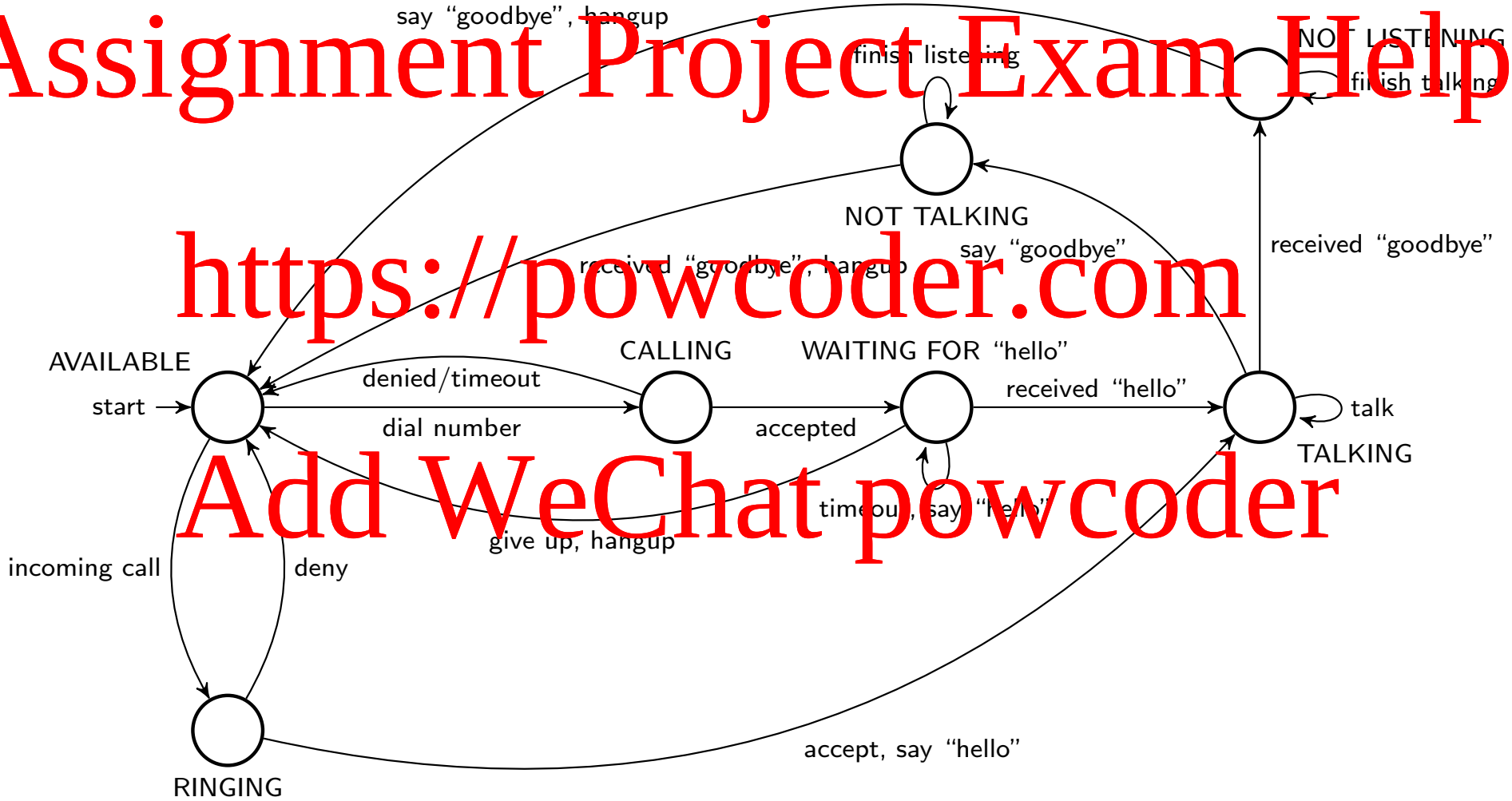
Question (3): Shared memory can be established between more than 2 processes, if those processes are on the same machine, i.e. 3 processes can address the same physical memory location. Is this point-to-multipoint or multipoint-to-point or multipoint-to-multipoint or something else? Justify your answer.

Communication Protocols

A *communication protocol* defines a deterministic, potentially unbounded sequence of interactions between a number of communicating parties. The purpose of the protocol is to define how the parties can systematically interact in order to effectively communicate given some existing communication mechanisms; e.g. a communication network.

- A communication protocol differs from a distributed algorithm in that a protocol may not terminate, whereas an algorithm is only valid if it terminates.
- Communication protocols require at least as much computational power as a Finite State Machine (FSM):
  - A party engaged in communication is in one of several states defined by the protocol, or is transitioning from one state to another.
  - State transitions are triggered by events in the environment or by the actions of the communicating parties, e.g. data is received, data is sent, or other events occur such as timeouts.
  - Communication protocols apply to *all* layers of a distributed system, from the lowest layer (Physical Layer) to the highest layer (Application or User Layer).

Telephone Communication Protocol



Discussion questions

Consider the Telephone Communication Protocol.

Question (4): Explain what happens if an incoming call arises when the person is TALKING.

Question (5): Currently the TALKING state allows both people to talk at the same time. Expand the protocol TALKING state, by replacing it with a number of other states, to ensure that only one person can talk at a time.

Question (6): What *communication errors* are catered for by the protocol? What common telephone communication errors are not handled by the protocol? Expand the protocol, by including new states to handle one of the communication errors that you discussed.

Question (7): Explain why there are states where the person is either NOT TALKING or NOT LISTENING.

Interaction Diagrams

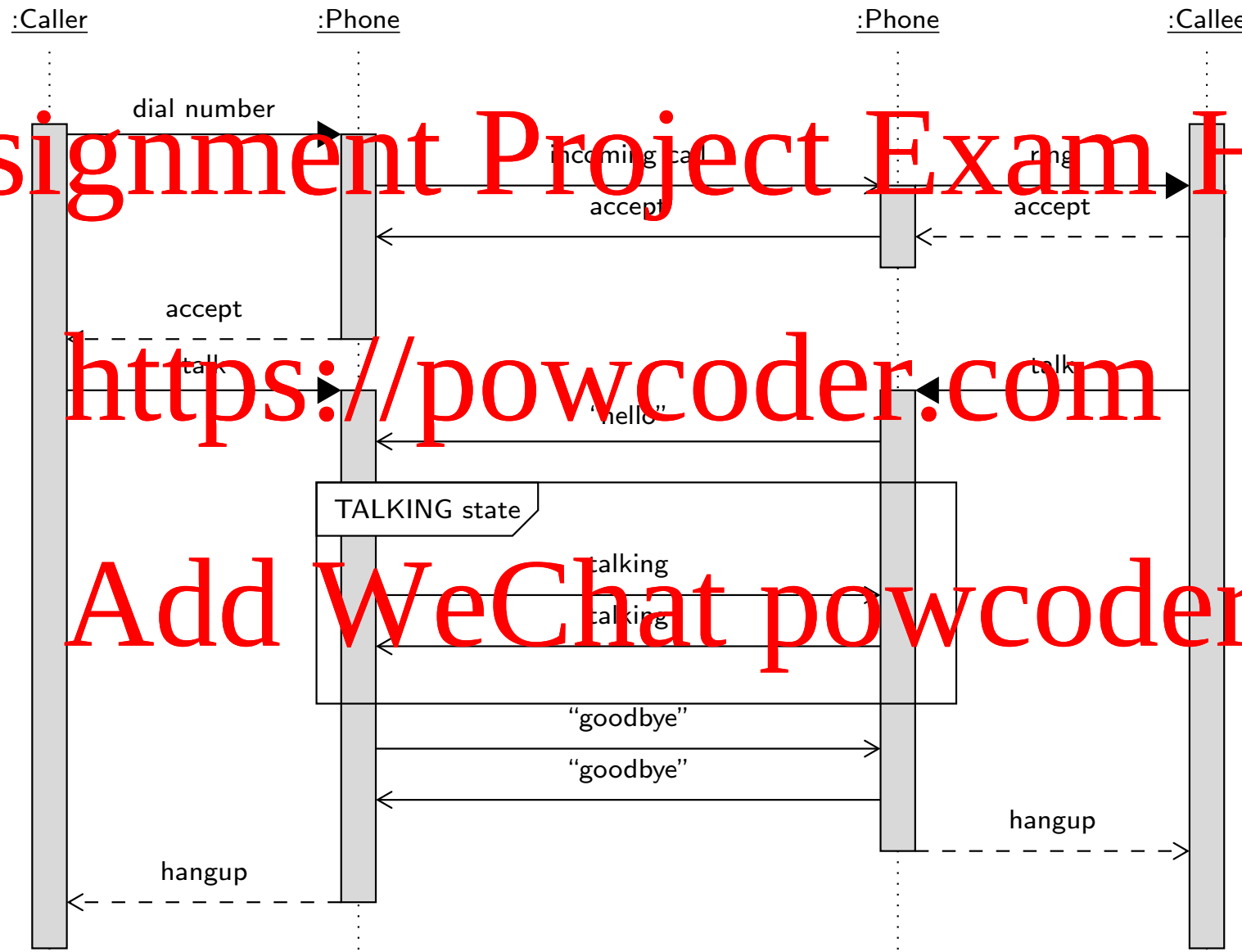
https://www.omg.org/spec/UML

It is sometimes helpful to represent sequences of interactions between communicating parties, arising from a communication protocol, using a sequence or *interaction diagram*, sometimes called an event diagram.

- The interaction diagram shows component interactions arranged in a time sequence.
- Parallel vertical lines on the interaction diagram show the lifetime of threads and other components in the system, and horizontal arrows show method calls and messages.
  - A solid arrow head represents a *synchronous* operation, whereby the caller must wait until the operation completes (returns) before it can continue.
  - A Hollow arrow head represents an *asynchronous* operation, whereby the caller can continue without having to wait for the operation's outcome.
  - Activation boxes on the timelines show activity being undertaken.

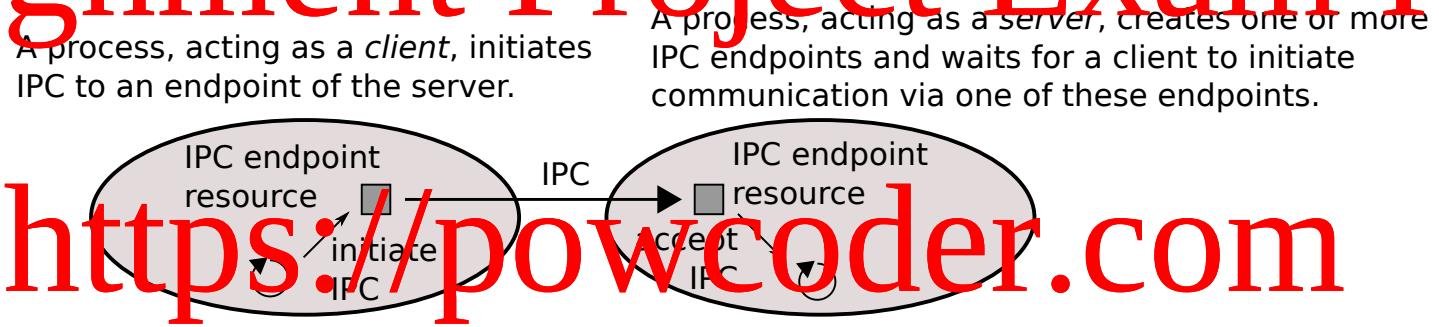
In this subject we make use of UML when its convenient and useful to do so, however we are not concerned with a rigorous treatment of UML techniques – that would be of interest in Software Engineering.

An interaction arising from the Telephone Communication Protocol



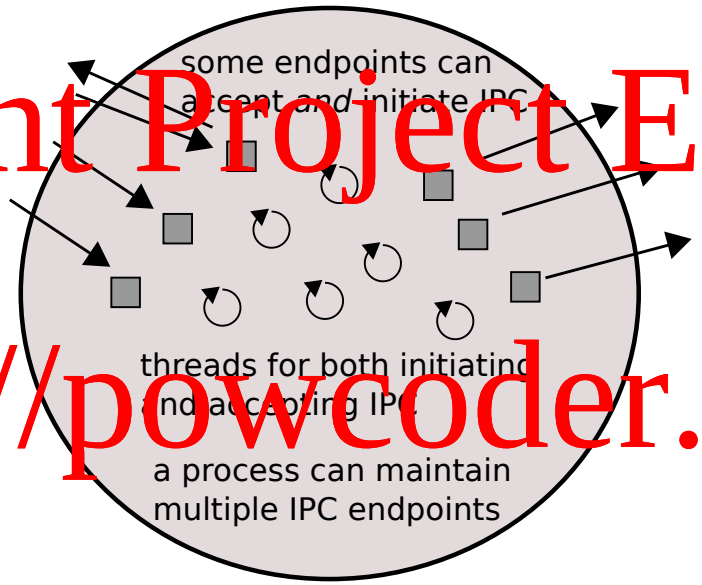
Caller/Callee ⇒ Client/Server roles

Two communicating parties take on roles depending on who initiates the communication



- IPC endpoints are resources provided by the OS.
- IPC requires a process to act as a *server*, which waits for another process, acting as a *client*, to initiate IPC.
- The server's endpoint must be known *a priori* to the client.
- A process that acts only as a client does not allow other processes to initiate IPC to it and we typically call it a *client process* or *client*.
- A process that acts only as a server never initiates IPC to other processes and we typically call it a *server process* or *server*.

A process can be both a client and server



A process may act as both a client and a server, allowing processes to initiate IPC to it, and also initiating IPC to other processes.

- A server in a multi-server achitecture may receive connections from clients but also make/receive connections to/from other servers.
- A *peer*, in a file sharing system may act as a client and make connections to the file index servers, but may also act as a server and allow other peers to connect to it.

Session Layer

The protocols that allow the communicating parties to initiate communication, provide the rules that they must follow to communicate data, and the rules for terminating the communication, are expressed in the **Session Layer**:

- A *session* is a long-running communication, that may involve the exchange of large and varied amounts of information.
- Sessions start with a negotiation or handshake that establishes the rules of the session, e.g. to support different protocols, data formats and algorithms used throughout the session.
- Sessions typically allow an unbounded amount of communication to take place, or as allowed by the negotiated protocols. The session protocol must specify how and when the different kinds of data will be transmitted.

- Sessions have a well defined termination, rather than the communicating parties simply ceasing to communicate.

Presentation Layer

Whatever the high level format of data to be communicated, IPC will ultimately lead to that data being represented as an *array of bytes*, either as a fixed (known) size array, e.g. when the source of the data is a file or data structure in memory, or as an unbounded array, e.g. when the source of the data is an audio device, or a keyboard that generates data. Perhaps the only exception to this is shared-memory IPC where the data may be communicated *in situ*, without the need to organize it into an array. In any case we know that all data is represented in the machine as bytes of information.

The **Presentation Layer** is concerned with APIs to support the translation of high level data formats to byte arrays, in an *external data representation* that is agreed upon by the communicating parties, and *vice versa*, which is sometimes called *marshalling* and *unmarshalling* respectively.

Data Encodings

While the data is in its most basic form an array of bytes, how information is encoded into those bytes is called the data *encoding*, and there are a number of popular choices:

- *Text encodings* are used to represent textual information:
  - **ASCII**: The American Standard Code for Information Interchange is one of the first global standards that uses 7 bits to encode numerals, the English alphabet (both upper and lower case), punctuation and control characters such as line feed, carriage return, backspace, etc. Various "Extended ASCII" encodings arose that use the 8th bit to encode a range of symbols useful for representing graphical user interfaces and extended alphabets, punctuation, etc.
  - **Unicode**: The modern standard for representing text written in all of the world's languages. The Unicode standards inlude Unicode Transformation Formats (UTF): UTF-8, UTF-16, and UTF-32, and several other encodings. UTF-8 has become the standard encoding on the World Wide Web and on many OSes. The first 128 code points of UTF-8 represent ASCII encodings for backwards compatibility, where each ASCII code point is also a UTF-8 code point. Some UTF-8 code ponts use more than 1 byte.
- *Binary encodings* are used to represent non-textual information:
  - Primitive data types: integer, float, boolean
  - Executable data: machine code
  - Image, audio and video: JPEG, MP3, MP4
  - Compressed data: GZIP
  - Encrypted data: RSA, AES

ASCII CONTROL CODE CHART

b7 b6 b5 / BITS b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 0 0 NUL	16 10 20 DLE	32 20 40 SP	48 30 60 0	64 40 100 @	80 50 120 P	96 60 140 ‘	112 70 160 p
0 0 0 1	1 1 1 SOH	17 11 21 DC1	33 21 41 !	49 31 61 1	65 41 101 A	81 51 121 Q	97 61 141 a	113 71 161 q
0 0 1 0	2 2 2 STX	18 12 22 DC2	34 22 42 ”	50 32 62 2	66 42 102 B	82 52 122 R	98 62 142 b	114 72 162 r
0 0 1 1	3 3 3 ETX	19 13 23 DC3	35 23 43 #	51 33 63 3	67 43 103 C	83 53 123 S	99 63 143 c	115 73 163 s
0 1 0 0	4 4 4 EOT	20 14 24 DC4	36 24 44 $	52 34 64 4	68 44 104 D	84 54 124 T	100 64 144 d	116 74 164 t
0 1 0 1	5 5 5 ENQ	21 15 25 NAK	37 25 45 %	53 35 65 5	69 45 105 E	85 55 125 U	101 65 145 e	117 75 165 u
0 1 1 0	6 6 6 ACK	22 16 26 SYN	38 26 46 &	54 36 66 6	70 46 106 F	86 56 126 V	102 66 146 f	118 76 166 v
0 1 1 1	7 7 7 BEL	23 17 27 ETB	39 27 47 ’	55 37 67 7	71 47 107 G	87 57 127 W	103 67 147 g	119 77 167 w
1 0 0 0	8 8 10 BS	24 18 30 CAN	40 28 50 (	56 38 70 8	72 48 110 H	88 58 130 X	104 68 150 h	120 78 170 x
1 0 0 1	9 9 11 HT	25 19 31 EM	41 29 51)	57 39 71 9	73 49 111 I	89 59 131 Y	105 69 151 i	121 79 171 y
1 0 1 0	10 A 12 LF	26 1A 32 SUB	42 2A 52 *	58 3A 72 :	74 4A 112 J	90 5A 132 Z	106 6A 152 j	122 7A 172 z
1 0 1 1	11 B 13 VT	27 1B 33 ESC	43 2B 53 +	59 3B 73 ;	75 4B 113 K	91 5B 133 [	107 6B 153 k	123 7B 173 {
1 1 0 0	12 C 14 FF	28 1C 34 FS	44 2C 54 ,	60 3C 74 <	76 4C 114 L	92 5C 134 \	108 6C 154 l	124 7C 174 |
1 1 0 1	13 D 15 CR	29 1D 35 GS	45 2D 55 –	61 3D 75 =	77 4D 115 M	93 5D 135]	109 6D 155 m	125 7D 175 }
1 1 1 0	14 E 16 SO	30 1E 36 RS	46 2E 56 .	62 3E 76 >	78 4E 116 N	94 5E 136 ^	110 6E 156 n	126 7E 176 ~
1 1 1 1	15 F 17 SI	31 1F 37 US	47 2F 57 /	63 3F 77 ?	79 4F 117 O	95 5F 137 _	111 6F 157 o	127 7F 177 DEL

LEGEND: dec CHAR hex oct

Victor Eijkhout
Dept. of Comp. Sci.
University of Tennessee
Knoxville TN 37996, USA

Data Formats

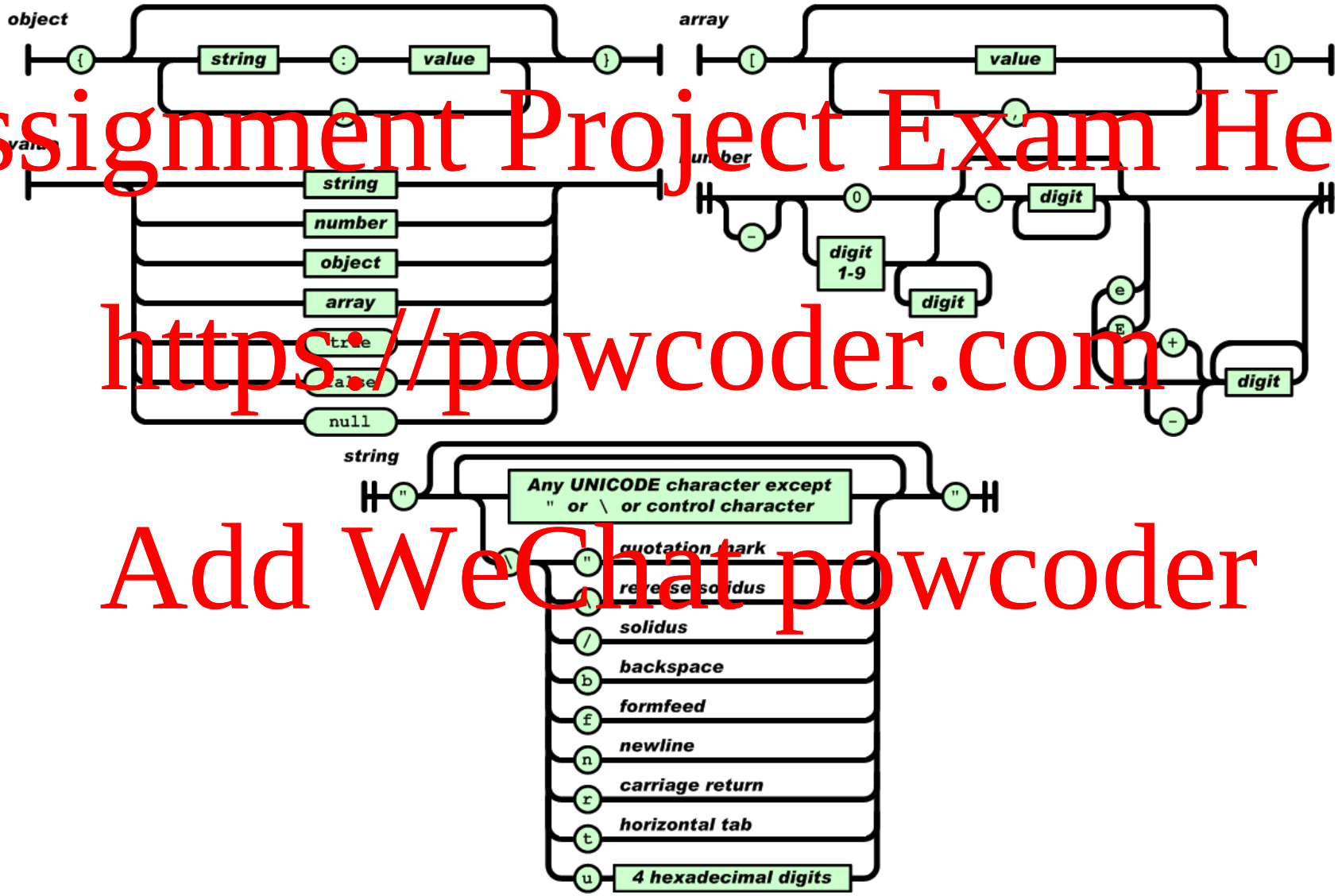
A data format is a syntax that describes how potentially arbitrary complex data and high-level semantics is assembled from encoded text and binary data:

- Complex data structures using text encodings: XML, JSON
- Text documents using text encodings: ASCII and UTF-8 files
- Formatted text documents using text encodings: HTML, Markdown
- Formatted text documents using binary encodings: Rich Text Format, DOC files
- Proprietary object representations: Java Serialization/Externalization

Usually, data formats based on text encodings cannot directly include binary encoded data, e.g. including a binary encoded image in an ASCII file is not valid. However there are formats that allow mixing of different encodings:

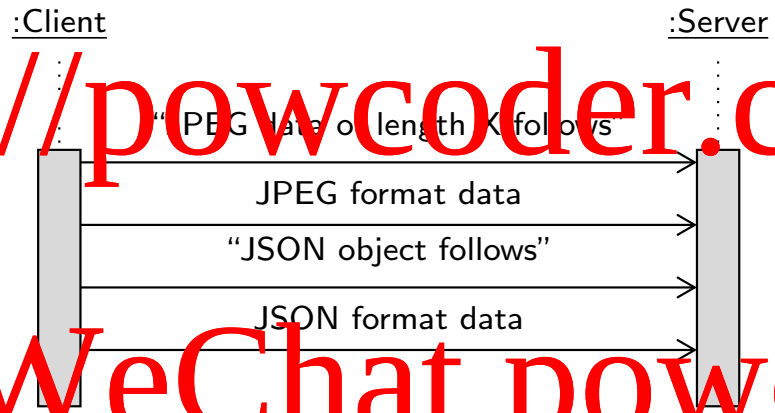
- Base64 encoding: encode binary data as text data.
- MIME: Multipurpose Internet Mail Extensions, uses Base64 encoding to allow email (which uses text encoded data) to effectively include images and other binary data.

Example: JSON Syntax Diagrams



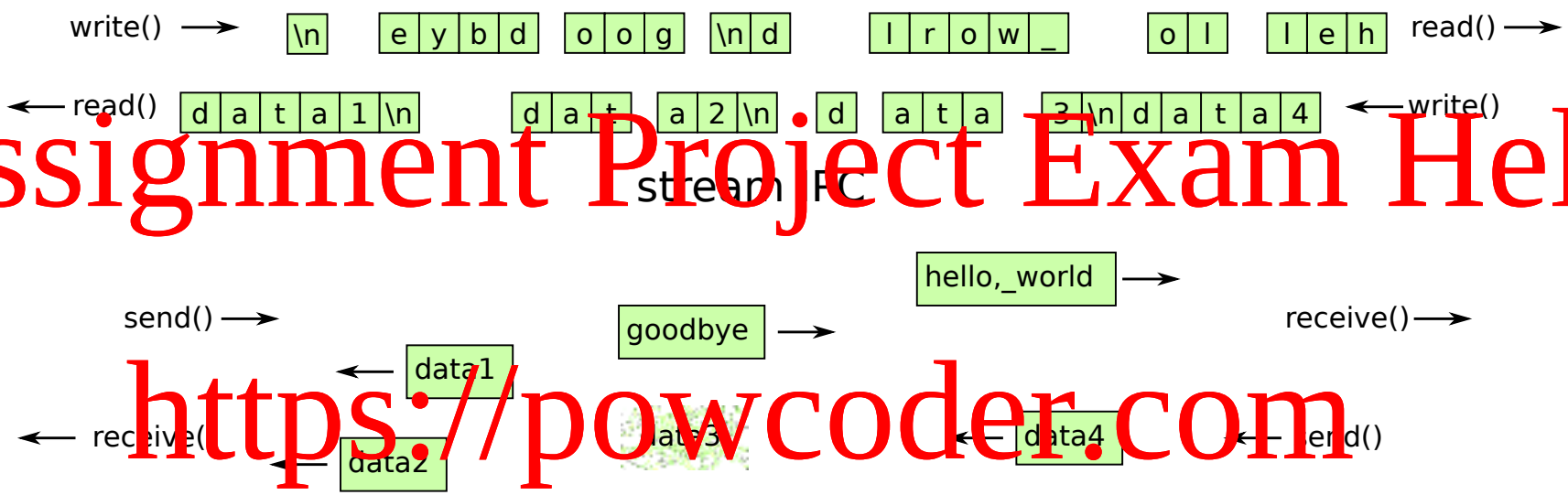
Data Protocol

The **Session Layer** protocols need to support communication of the desired data formats, so that e.g. the server knows what to expect and how to unmarshall or interpret the data that it receives.



Sometimes the encodings and data format are either implicit or possibly the protocol itself follows a data format such that there is little or no distinction between the protocol and the format.

Streams and Datagram IPC



- A stream is an *unbounded sequence* of data elements, with bytes as the most primitive elements:
  - The sending process *writes* elements to the stream and the receiving process *reads* elements from the stream.
  - Data elements are read in the same order that they are written.
  - No data corruption is allowed.
- A datagram is a *bounded or fixed size* data array:
  - The sending process *sends* the datagram to the receiver which *receives* it.
  - The order that datagrams are sent is not necessarily the order that they are received.
  - Datagram loss is assumed; not all datagrams sent are received.

POSIX Socket API

- The *Socket* paradigm is the dominant API for IPC over the Internet, originating with the 4.2BSD Unix OS, released in 1983.
- The Socket API by default uses the TCP/UDP/IP protocol stack managed by the OS, but may support other protocol stacks (implementations) as well.
  - TCP provides reliable byte stream IPC with congestion control.
  - UDP provides "best-effort" datagram IPC.
- The POSIX Socket API is fairly consistently implemented across many OSes.
- On Microsoft Windows OSes, the WinAPI provides WinSock that is very similar to the POSIX standard.
- In this subject we use the Java VM and the Java Socket API for socket programming, which allows our programs to run on all common hardware/OS platforms with relatively few platform-specific code variations required.

Port Numbers

- Protocols such as TCP/UDP use a set of 2^{16} *ports*, numbered 0 to 65535, called *port numbers*, to identify individual processes or services on a machine.
- The OS manages access to port numbers: processes can request to be associated with them.
- All TCP/UDP based IPC always specifies an IP address and a port number as the destination.

From https://en.wikipedia.org/wiki/Port_(computer_networking)

Port	Usage
20	File Transfer Protocol (FTP) Data Transfer
21	File TRansfer Protocol (FTP) Command Control
22	Secure Shell (SSH) Secure Login
23	Telnet remote login service, unencrypted text messages
25	Simple Mail Transfer Protocol (SMTP) e-mail delivery
53	Domain Name System (DNS) service
67, 68	Dynamic Host Configuration Protocol (DHCP)
80	Hypertext Transfer Protocol (HTTP)
110	Post Office Protocol (POP3)
119	Network News Transfer Protocol (NNTP)
123	Network Time Protocol (NTP)
143	Internet Message Access Protocol (IMAP)
161	Simple Network Management Protocol (SNMP)
194	Internet Relay Chat (IRC)
443	HTTP Secure (HTTPS) HTTP over TLS/SSL

IP Address

- Each machine on the Internet or on a private network using the Internet Protocols has one or more network interfaces (typically Ethernet) and one or more assigned IP addresses that each associate to a given network interface on the machine. An interface may associate with multiple IP addresses.
- IPv4 is 4 bytes, while IPv6 is 16 bytes. The support for IPv6 across ISPs is growing but yet complete. In this subject we will assume IPv4 in our discussions.
- An interface with a public IP address can be reached by any machine connected to the Internet.
- An interface with a private IP address can be reached by any machine on the same private network.
- Some addresses have special meaning:
  - `127.0.0.1` is a *loopback* address, any packet sent this address is not sent via the network but rather is looped back to the machine itself, i.e. so that processes on the machine can use network communication to other processes on the same machine, without requiring an actual network interface.
  - `localhost` usually resolves to `127.0.0.1`
  - Some address ranges are reserved for private networks, including `10.0.0.0/8` and `192.0.0.0/24` and `192.168.0.0/16`

InetAddress class

The InetAddress class encapsulates an IP address and provides a range of helper methods.

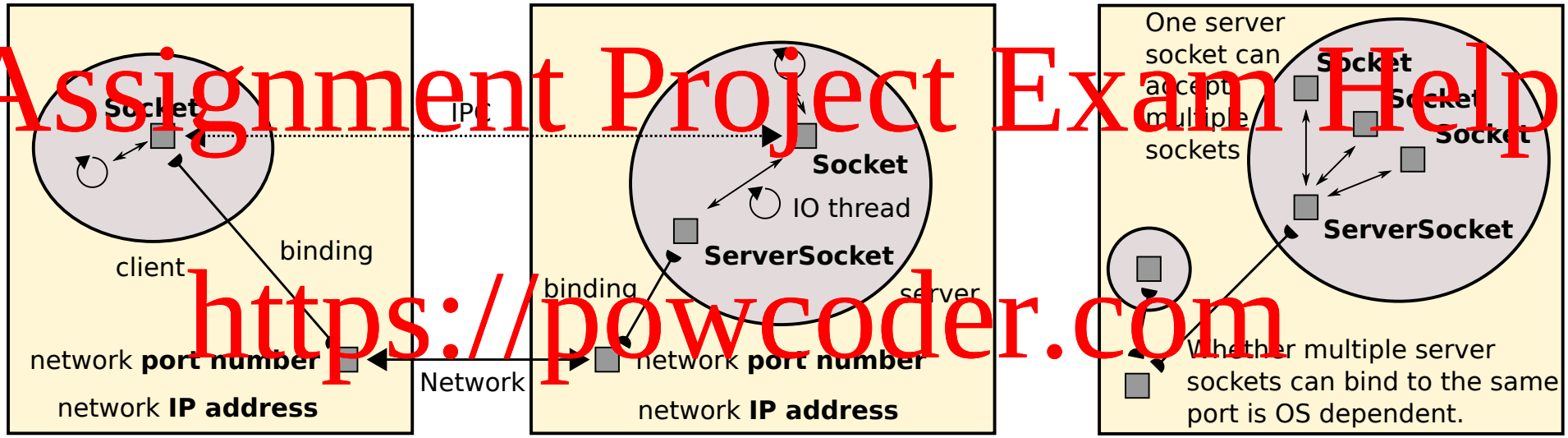
InetAddress API: selected methods	
`static InetAddress`	`getByName(String host) throws UnknownHostException` Determine the IP address of a host, given the host's name. The host name can either be a machine name, such as "java.sun.com", or a textual representation of its IP address. If a literal IP address is supplied, only the validity of the address format is checked.
`static InetAddress`	`getLocalHost() throws UnknownHostException` Return the address of the local host. This is achieved by retrieving the name of the host from the system, then resolving that name into an InetAddress.
`boolean`	`isReachable(int timeout) throws IOException` Test whether the address is reachable. Best effort is made by the implementation to try to reach the host, but firewalls and server configuration may block requests resulting in a unreachable status while some specific ports may be accessible. The timeout value, in milliseconds, indicates the maximum amount of time the try should take. If the operation times out before getting an answer, the host is deemed unreachable.
`String`	`getHostName()` Get the host name for this IP address. If this InetAddress was created with a host name, this host name will be remembered and returned; otherwise, a reverse name lookup will be performed and the result will be returned based on the system configured name lookup service.
`String`	`getCanonicalHostName()` Get the *fully qualified domain name* for this IP address. Best effort method, meaning we may not be able to return the FQDN depending on the underlying system configuration.
`String`	`getHostAddress()` Return the IP address string in textual presentation.

Discussion questions

Review your understanding of computer networks to answer this question.

Question (8): A process running on a machine with a private IP address can initiate communication with a machine on the public Internet if it knows the public IP address of the machine, and the private network is connected to the public Internet via one or more routers. But how can the machine on the public Internet initiate communication with a machine on the private network? Note that the private IP address is only unique within the private network that the machine resides, and so many machines in many different private networks may have the same private IP address.

Socket and ServerSocket

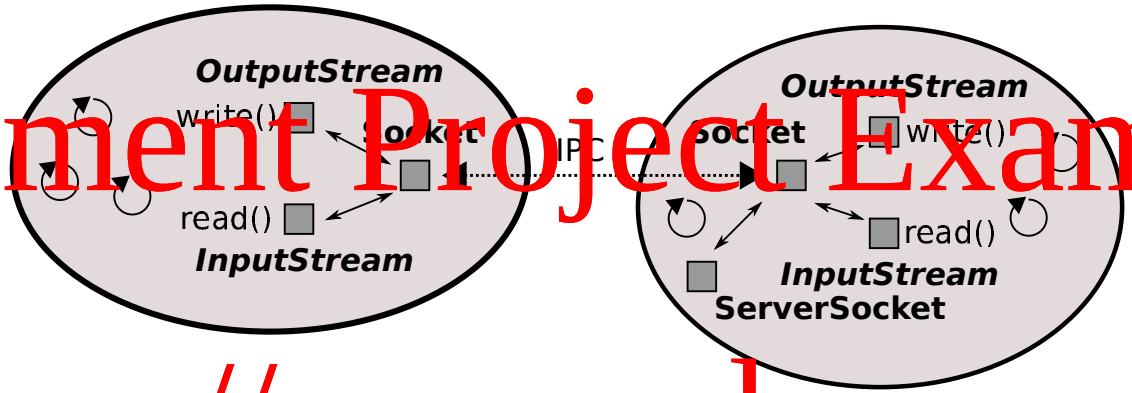


- The `Socket` and `ServerSocket` classes use TCP by default.
- The server process creates a `ServerSocket` and *binds* it to a network *port number* and network *IP address* on the machine.
- The client process creates a `Socket` and *connects* to a server, knowing the server's port number and IP address. The client's socket is also bound to a port number and IP address, which allows the server to respond.
- The `ServerSocket` creates a `Socket` for each incomming connection.
- The two processes communicate using a stream via their respective `Socket` objects.
- When communication is finished, the associated `Socket` objects are destroyed.

ServerSocket API: selected constructors and methods	
	`ServerSocket(int port) throws IOException` Create a server socket bound to the specified port. A port number of 0 means that the port number is automatically allocated. The maximum queue length for incoming connection indications (a request to connect) is set to 50. If a connection indication arrives when the queue is full, the connection is refused.
`Socket`	`accept() throws IOException` Listen for a connection to this socket and accept it. *Blocks* until a connection is made.
`void`	`close() throws IOException` Close this server socket. Any thread currently blocked in `accept()` will throw a `SocketException`.
`InetAddress`	`getInetAddress()` Get the *local* address of this server socket.
`int`	`getLocalPort()` Get the *local* port number on which this socket is listening.

Socket API: selected constructors and methods	
	`Socket(String host, int port) throws UnknownHostException, IOException` Create a socket and initiate a connection to the named host and port. If the specified host is `null` it is the equivalent of specifying the address as `InetAddress.getByName(null)`.
`void`	`close() throws IOException` Close the socket. Any thread currently blocked in an I/O operation upon this socket will throw a `SocketException`. Once a socket has been closed, it is not available for further networking use (i.e. can't be reconnected or rebound). A new socket needs to be created. Closing this socket will also close the socket's `InputStream` and `OutputStream`.
`OutputStream`	`getOutputStream() throws IOException` Return the socket's *output stream*. Closing the returned `OutputStream` will close the associated socket.
`InputStream`	`getInputStream() throws IOException` Return the socket's *input stream*. Closing the returned `InputStream` will close the associated socket.
`InetAddress`	`getInetAddress()` Return the *remote* address to which the socket is connected.
`InetAddress`	`getLocalAddress()` Return the *local* address to which the socket is bound.
`int`	`getPort()` Return the *remote* port number to which this socket is connected.
`int`	`getLocalPort()` Return the *local* port number to which this socket is bound.

Abstract InputStream and OutputStream classes



The `Socket` object provides an `InputStream` for receiving data and `OutputStream` for writing data, with byte streams as the most primitive concrete class:

- `BufferedInputStream` and `BufferedOutputStream` for simple *byte streams*,
- `DataInputStream` and `DataOutputStream` for *primitive Java data type streams*,
- `ObjectInputStream` and `ObjectOutputStream` for *Java object streams*.

For example we can create a `DataInputStream` from any `InputStream` object:

```
DataInputStream dataInputStream = new DataInputStream(socket.getInputStream());
```

TCP Client Code Snippet

```
1   import java.net.*;
2   import java.io.*;
3   public class TCPClient {
4       public static void main (String args[]) {
5           Socket socket = null;
6           try{
7               int serverPort = 7899; // the known server port
8               socket = new Socket(args[1], serverPort);  // connect to the server
9               System.out.println("Connected to: "+socket.getInetAddress()+":"+socket.getPort());
10              BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
11              PrintWriter out = new PrintWriter(socket.getOutputStream(),true); // autoflush true
12              System.out.println("Sending: "+args[0]);
13              out.println(args[0]);
14              /* The following read will block until a line of data has been received.
15               * If the server fails to send a line of data back, then this line will
16               * block until an exception occurs, aka hanging. */
17              String data = in.readLine(); // read a line of data from the stream, omits the newline
18              System.out.println("Received: "+data);
19          } catch (UnknownHostException e) {
20              System.out.println("The provided host could not be resolved: "+args[1]);
21          } catch (IOException e){
22              e.printStackTrace(); // most of the IO operations above throw this exception
23          } finally {
24              if(socket!=null) try {
25                  socket.close(); // will also close IO streams
26              } catch (IOException e){
27                  System.out.println("close: "+e.getMessage());
28              }
29          }
30      }
31  }
```

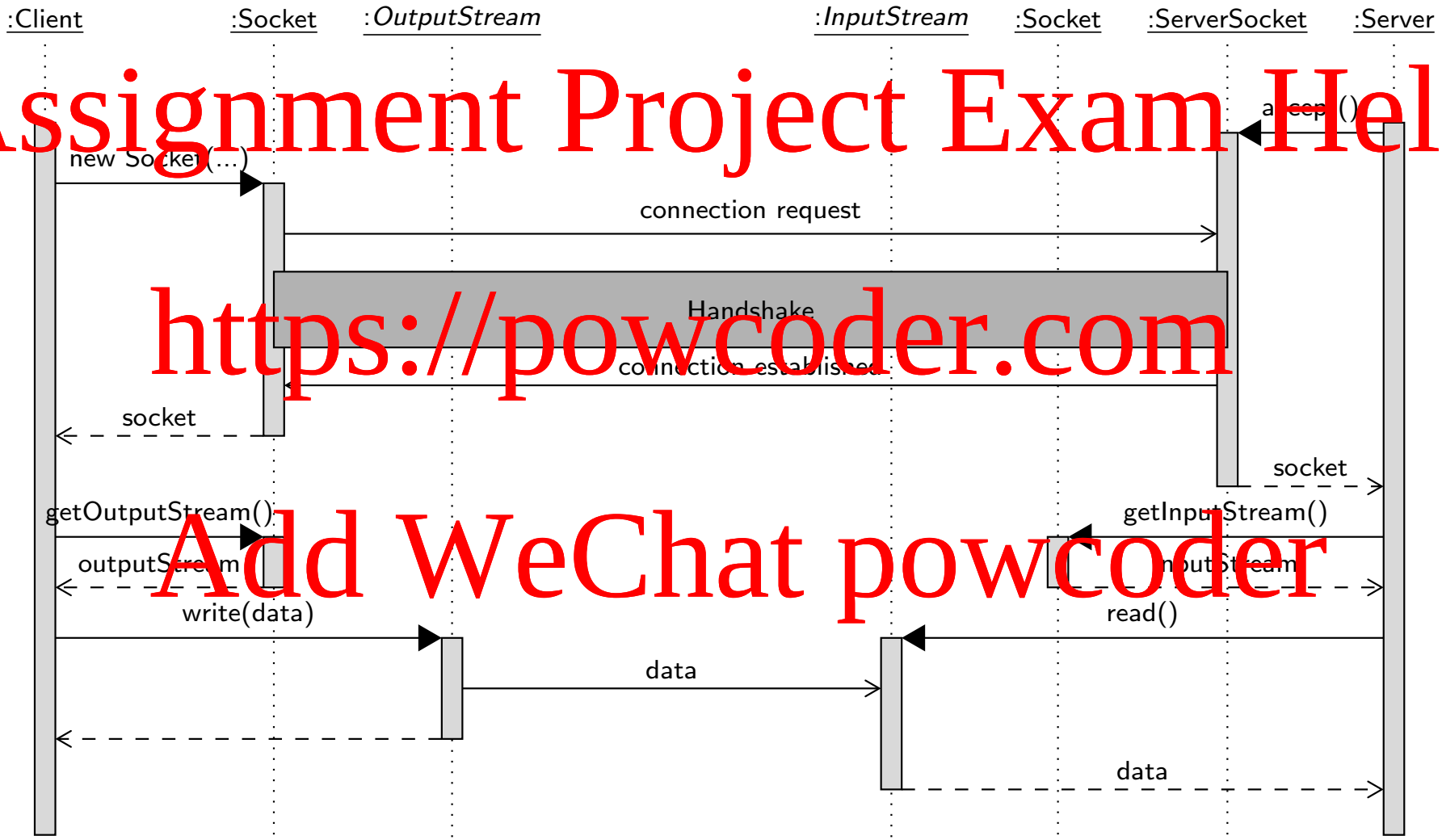
TCP Server Code Snippet

```
1  import java.net.*;
2  import java.io.*;
3
4  public class TCPServer {
5      public static void main (String args[]) {
6          try{
7              int serverPort = 7899; // the server port
8              ServerSocket serverSocket = new ServerSocket(serverPort);
9              int i = 0;
10             while(true) { // will loop until an exception occurs
11                 System.out.println("Server listening on " +
12                     serverSocket.getInetAddress()+":"+serverSocket.getLocalPort());
13                 /* The following line blocks until a connection is received, or
14                  * an exception occurs. */
15                 Socket clientSocket = serverSocket.accept();
16                 i++;
17                 System.out.println("Received connection: " + i +" from " +
18                     clientSocket.getInetAddress()+":"+clientSocket.getPort())
19                 /* Hand the connection over to another (threaded) object. */
20                 Connection c = new Connection(clientSocket);
21             }
22         } catch(IOException e) {
23             e.printStackTrace()
24         }
25     }
26 }
```

Connection Code Snippet

```
1   class Connection extends Thread {
2       BufferedReader in;
3       PrintWriter out;
4       Socket clientSocket;
5       public Connection (Socket clientSocket) {
6           try {
7               this.clientSocket = clientSocket;
8               in = new BufferedReader( new InputStreamReader(clientSocket.getInputStream()));
9               out = new PrintWriter( clientSocket.getOutputStream(),true);
10              this.start();
11          } catch(IOException e) {
12              System.out.println("Connection:"+e.getMessage());
13          }
14      }
15      public void run(){
16          try { // an echo server
17              System.out.println("server reading data");
18              /* The following blocks until the client writes a line of data.
19               * If the client fails to write a line then this line blocks until
20               an exception is thrown. */
21              String data = in.readLine(); // read a line of data from the stream
22              System.out.println("server writing: "+data);
23              out.println(data);
24          } catch(IOException e) {
25              e.printStackTrace();
26          } finally {
27              try {
28                  clientSocket.close();
29              } catch (IOException e){/*close failed*/}
30          }
31      }
32  }
```

Simplified client/server socket interaction

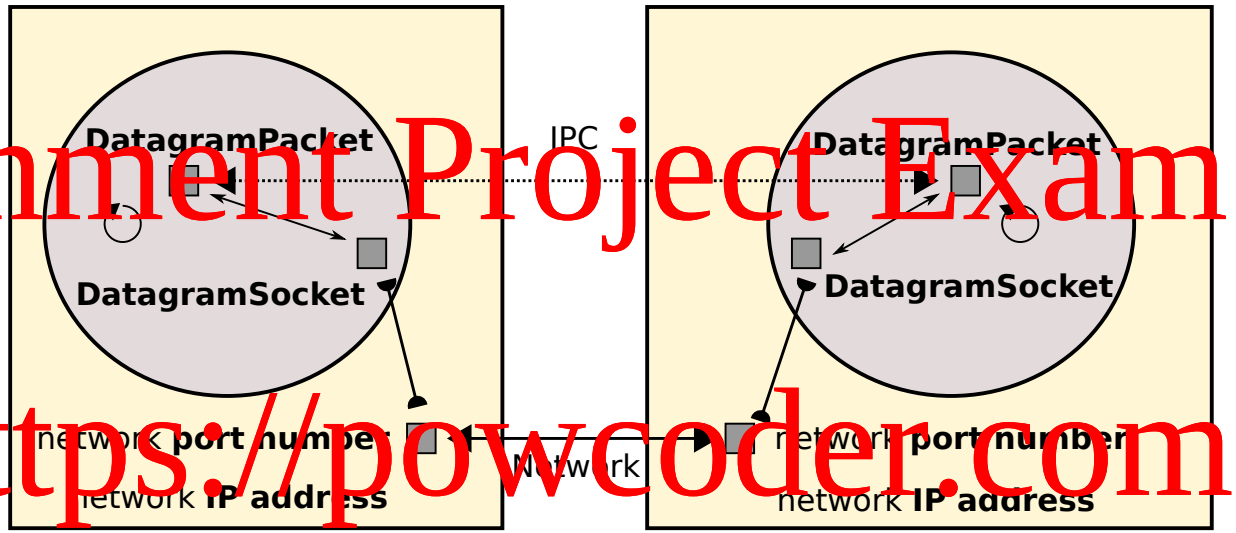


Discussion question

Question (9): The previous diagram is quite simplified. Draw a client/server socket interaction diagram that shows all of the objects involved, method calls, and more detailed data flow, with the example code as a reference. Indicate on your diagram which methods calls on the client and server *must* be in a specific time order for the depicted sequence of operations to succeed, and which method calls can be arbitrary in time order.

Question (10): Draw a FSM that describes the (quite simple) communication protocol employed by the TCP client and server.

DatagramSocket and DatagramPacket



- The `DatagramSocket` and `DatagramPacket` use UDP communication by default.
- Both processes create a `DatagramSocket` that binds to a port number and IP address.
- The client process initiates IPC by creating a `DatagramPacket` object and sending its content via its `DatagramSocket` to the server process.
- The server process receives data in a `DatagramPacket` object from its `DatagramSocket`.
- The server process can respond by sending the content of a `DatagramPacket`, using its `DatagramSocket`, back to the client process.

DatagramSocket API: selected constructors and methods	
	`DatagramSocket() throws SocketException` Constructs a datagram socket and binds it to any available port on the local host machine. The socket will be bound to the wildcard address, an IP address chosen by the kernel.
	`DatagramSocket(int port) throws SocketException` Constructs a datagram socket and binds it to the specified port on the local host machine. The socket will be bound to the wildcard address, an IP address chosen by the kernel.
`void`	`close()` Closes this datagram socket. Any thread currently blocked in `receive(java.net.DatagramPacket)` upon this socket will throw a `SocketException`.
`void`	`receive(DatagramPacket p) throws IOException` Receives a datagram packet from this socket. When this method returns, the `DatagramPacket`'s buffer is filled with the data received. The datagram packet also contains the sender's IP address, and the port number on the sender's machine. This method *blocks* until a datagram is received. The length field of the datagram packet object contains the length of the received message. If the message is longer than the packet's length, the message is truncated.
`void`	`send(DatagramPacket p) throws IOException` Sends a datagram packet from this socket. The `DatagramPacket` includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.
`InetAddress`	`getLocalAddress()` Gets the local address to which the socket is bound.
`int`	`getLocalPort()` Returns the port number on the local host to which this socket is bound.

DatagramPacket API: selected constructors and methods	
	`DatagramPacket(byte[] buf, int length)` Constructs a `DatagramPacket` for receiving packets of length `length`. The length argument must be less than or equal to `buf.length`.
	`DatagramPacket(byte[] buf, int offset, int length)` Constructs a datagram packet for sending packets of length `length`, specifying an offset into the buffer. The length argument must be less than or equal to `buf.length`.
	`DatagramPacket(byte[] buf, int length, InetAddress address, int port)` Constructs a datagram packet for sending packets of length `length` to the specified port number on the specified host. The length argument must be less than or equal to `buf.length`.
	`DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)` Constructs a datagram packet for sending packets of length `length`, specifying an offset into the buffer, to the specified port number on the specified host. The length argument must be less than or equal to `buf.length`.
`InetAddress`	`getAddress()` Returns the IP address of the machine to which this datagram is being sent or from which the datagram was received.
`int`	`getPort()` Returns the port number on the remote host to which this datagram is being sent or from which the datagram was received.
`byte[]`	`getData()` Returns the data buffer. The data received or the data to be sent starts from the offset in the buffer, and runs for length long.
`int`	`getOffset()` Returns the offset of the data to be sent or the offset of the data received.
`int`	`getLength()` Returns the length of the data to be sent or the length of the data received.
`void`	`setData(byte[] buf, int offset, int length)` Set the data buffer for this packet. This sets the data, length and offset of the packet.
`void`	`setAddress(InetAddress addr)` Sets the IP address of the machine to which this datagram is being sent.
`void`	`setPort(int port)` Sets the port number on the remote host to which this datagram is being sent.

UDP Client Code Snippet

```
1   import java.net.*;
2   import java.io.*;
3   public class UDPClient {
4       public static void main(String args[]){
5       // args[0] is message args[1] is server's hostname
6           DatagramSocket aSocket = null;
7           try {
8               aSocket = new DatagramSocket(); // we don't care which port it binds to
9               byte[] m = args[0].getBytes(); // get the message as a byte array
10              InetAddress serverAddress = InetAddress.getByName(args[1]); // resolve the server's name
11              int serverPort = 6789; // the server's known port number
12              DatagramPacket request =
13                  new DatagramPacket(m, args[0].length(), serverAddress, serverPort);
14              System.out.println("Sending data: "+args[0]);
15              aSocket.send(request);
16              byte[] buffer = new byte[1000]; // magic number :-S
17              DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18              System.out.println("Client waiting to receive a response");
19              /* The following line blocks until the server sends a response.
20                 If the server fails to send a response then this line blocks
21                  * until an exception occurs. */
22              aSocket.receive(reply);
23              System.out.println("Reply: " + new String(reply.getData()));
24          } catch (SocketException e){
25              System.out.println("Socket: " + e.getMessage());
26          } catch (IOException e){
27              System.out.println("IO: " + e.getMessage());
28          } finally {if(aSocket != null) aSocket.close();}
29      }
30  }
```

UDP Server Code Snippet

```
1   import java.net.*;
2   import java.io.*;
3   public class UDPServer {
4       public static void main(String args[]){
5           DatagramSocket aSocket = null;
6           try {
7               aSocket = new DatagramSocket(6789); // create socket at agreed port
8               byte[] buffer = new byte[1000]; // magic number :-S
9               while(true){
10                  DatagramPacket request = new DatagramPacket(buffer, buffer.length);
11                  System.out.println("Server waiting to receive data");
12                  aSocket.receive(request);
13                  System.out.println("Received Data: " + new  String(request.getData()));
14                  DatagramPacket reply = new DatagramPacket(request.getData(),
15                      request.getLength(), request.getAddress(), request.getPort());
16                  aSocket.send(reply);
17              }
18          } catch (SocketException e){
19              System.out.println("Socket: " + e.getMessage());
20          } catch (IOException e) {
21              System.out.println("IO: " + e.getMessage());
22          } finally {if(aSocket != null) aSocket.close();}
23      }
24  }
```

Simplified client/server datagram interaction