InterProcess Communication (IPC)

Distributed software systems
CdLM Informatica - Università di Bologna

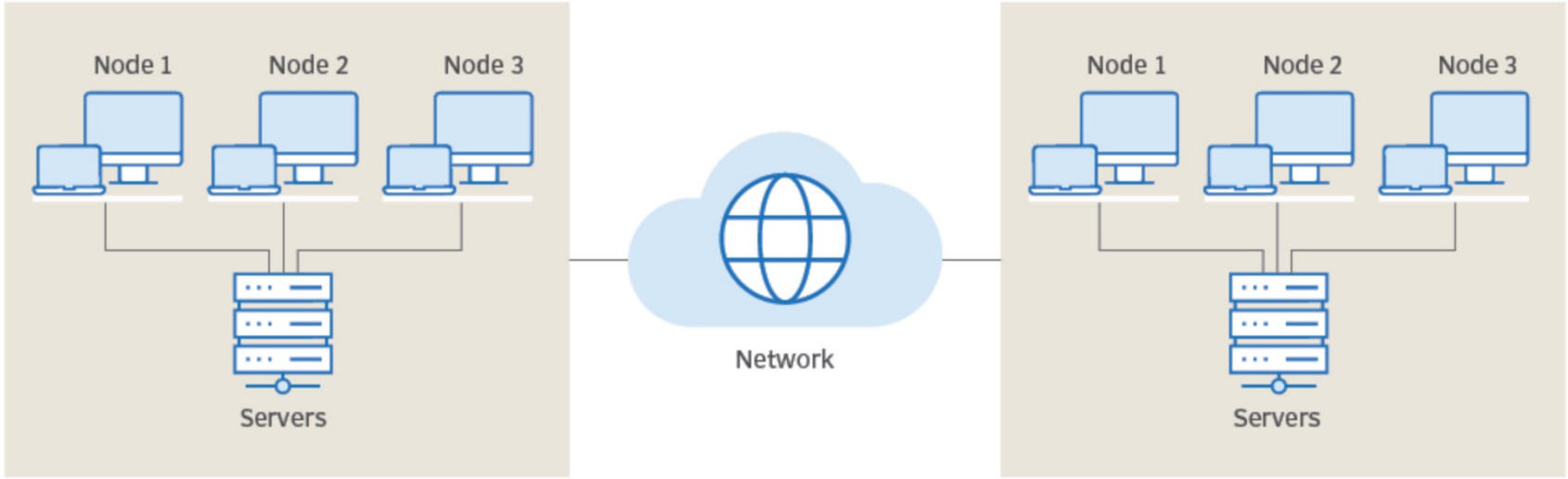
Agenda

The API for programming with the Internet protocols

External data representation and marshalling

Distributed systems are made of software communicating through networks

- Distributed systems need communication infrastructures: local area networks, wide area networks, and the internet.
- The performance, reliability, scalability, mobility, and Quality of Service properties of these infrastructures impact the design and behaviour of distributed systems.



Interprocess Communication

API for Internet protocols

- sockets
- UDP datagrams
- TCP streams

Marshalling

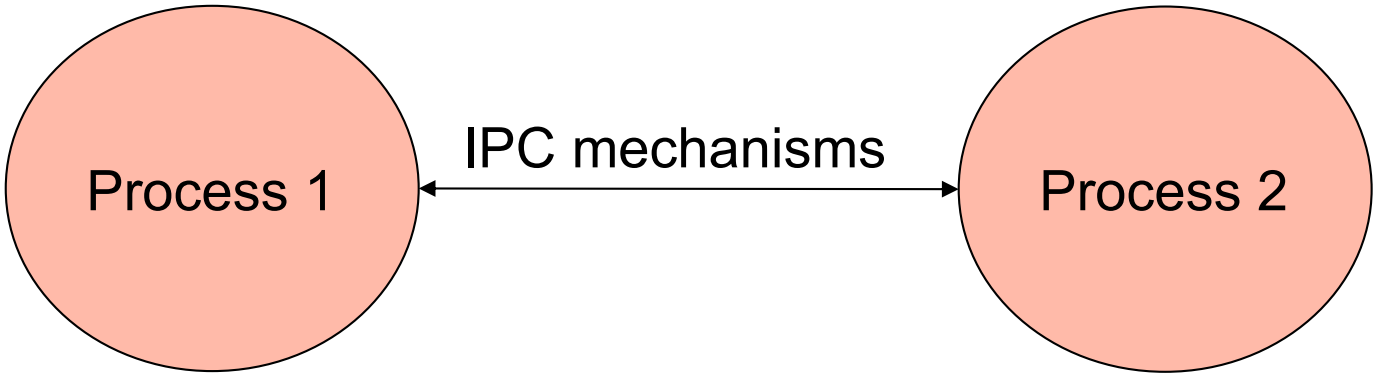
- XML, JSON

InterProcess Communication (IPC)

Two processes which run on the same machine can communicate very fast, via the operating system

Question: how do processes on different machines exchange information?

Answer: with difficulty … ☹



Models of InterProcess Communication (processes on the same machine)

- Send / receive
- Blocking (synchronous) or not blocking (asynchronous)
- Messages on channel: transient or persistent

- Unix domain Sockets – data communication endpoints
  - SOCK_STREAM for a stream-oriented socket (TCP)
  - SOCK_DGRAM for a datagram-oriented socket (UDP)

Models of InterProcess Communication (processes on different machines)

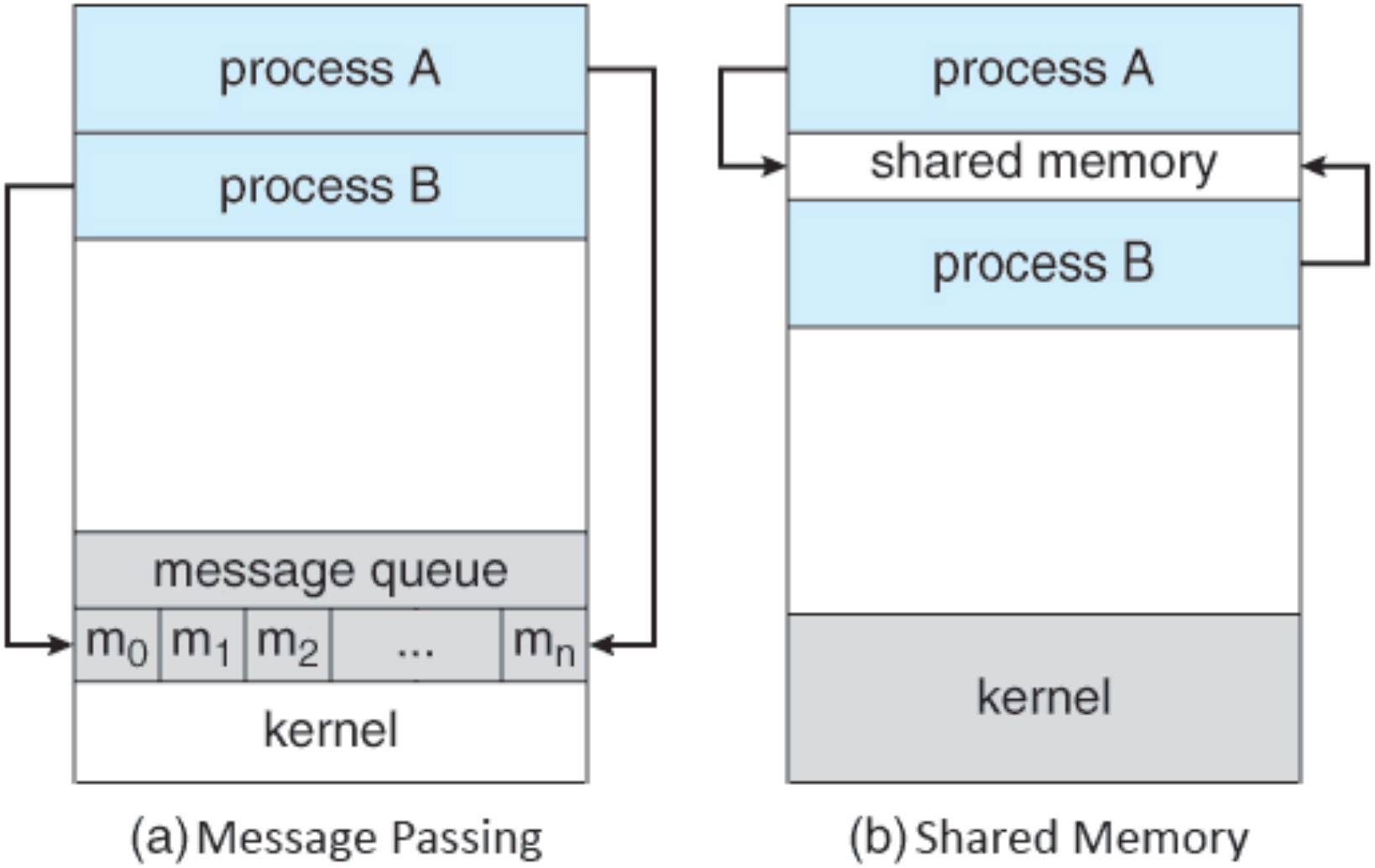
- Computer network facilities are **very primitive**, resulting in distributed software difficult to develop
  - How do you name different resources in different machines?
  - How do you establish several “channels” for the same process?
- Four IPC models are popular:
  - RPC remote procedure calls;
  - RMI remote method invocation;
  - Streams – example Apache Flink;
  - MOM Message Oriented Middleware – example: Kafka

InterProcess Communication (IPC)

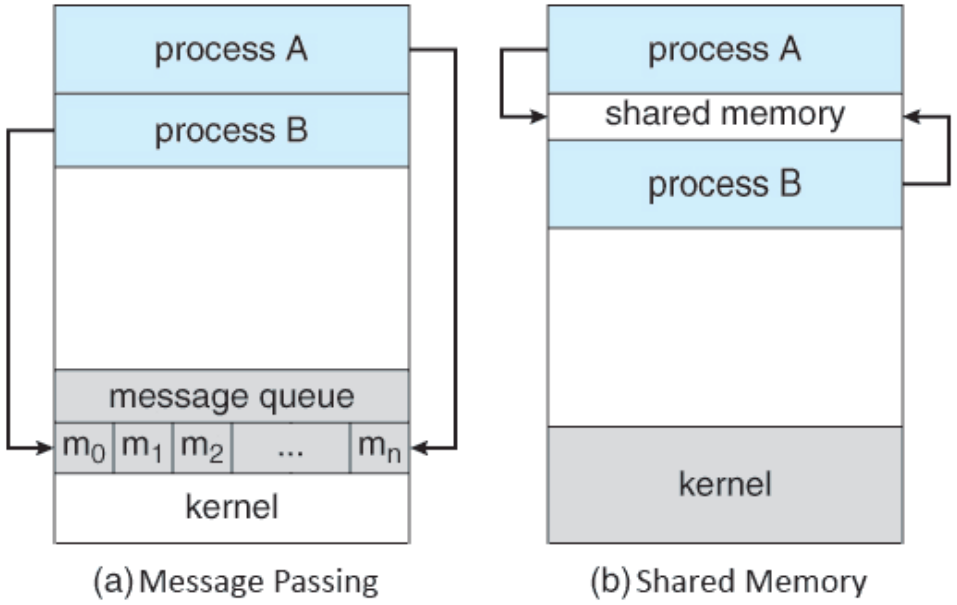
- Programs in execution become *processes*
- Inside a single computer the operating system supports IPC via shared memory with special high level mechanisms
- In networked systems IPC is based on low-level message passing (send message over a channel, receive message from a channel; the channels are called *sockets*)

- IMPORTANT: communication through message passing is harder to program and more error-prone than using primitives based on shared memory

IPC: message passing vs shared memory (single computer)



Example: IPC on Windows



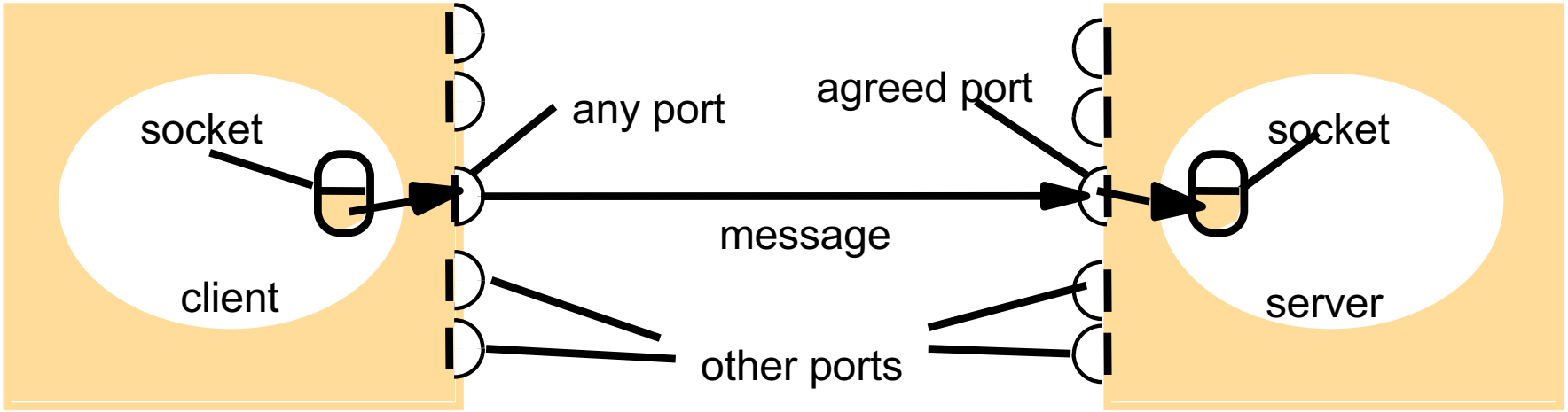
Windows IPC alternatives

- Clipboard
- COM
- Data Copy
- DDE
- File Mapping
- Mailslots
- Pipes
- RPC
- Windows Sockets

Unix sockets

- A *Unix domain socket (*or IPC socket) is a communication endpoint for exchanging data between processes executing on the same host operating system
- Unix domain sockets support transmission of a reliable stream of bytes (SOCK_STREAM, compare to TCP).
- In addition, they support ordered and reliable transmission of datagrams (SOCK_SEQPACKET, compare to SCTP), or unordered and unreliable transmission of datagrams (SOCK_DGRAM, compare to UDP).
- The Unix domain socket facility is a standard component of POSIX operating systems.
- The API for Unix domain sockets is similar to that of an Internet socket, but rather than using an underlying network protocol, all communication occurs entirely within the operating system kernel.
- Unix domain sockets use the file system as their address name space

Sockets and ports



- Messages sent to a particular address and port number can be received only by a process whose socket is associated with that address and port number.
- Processes may use the same socket for sending and receiving messages.
- Each computer operating system has a large number of possible port numbers for use by local processes for receiving messages.
- Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer

Sockets: primitives for exchanging messages

consider the following two programs, also known as a **client** and a **server**, written in Python (some code has been removed for clarity).

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept() # returns new socket and addr.
client
4 while True: # forever
5 data = conn.recv(1024) # receive data from client
6 if not data: break # stop if client stopped
7 conn.send(str(data)+"*") # return sent data plus an "*"
8 conn.close() # close the connection
```

(a) A simple server

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # to server (block until accepted
4 s.send('Hello, world') # send some data
5 data = s.recv(1024) # receive the response
6 print data # print the result
7 s.close() # close the connection
```

(b) A client

In this example, a (sequential) server is created that uses of connection-oriented library available in Python: this allows two communicating parties to reliably send and receive data over a connection. The main functions available in its interface are:

socket(): to create an object representing the connection

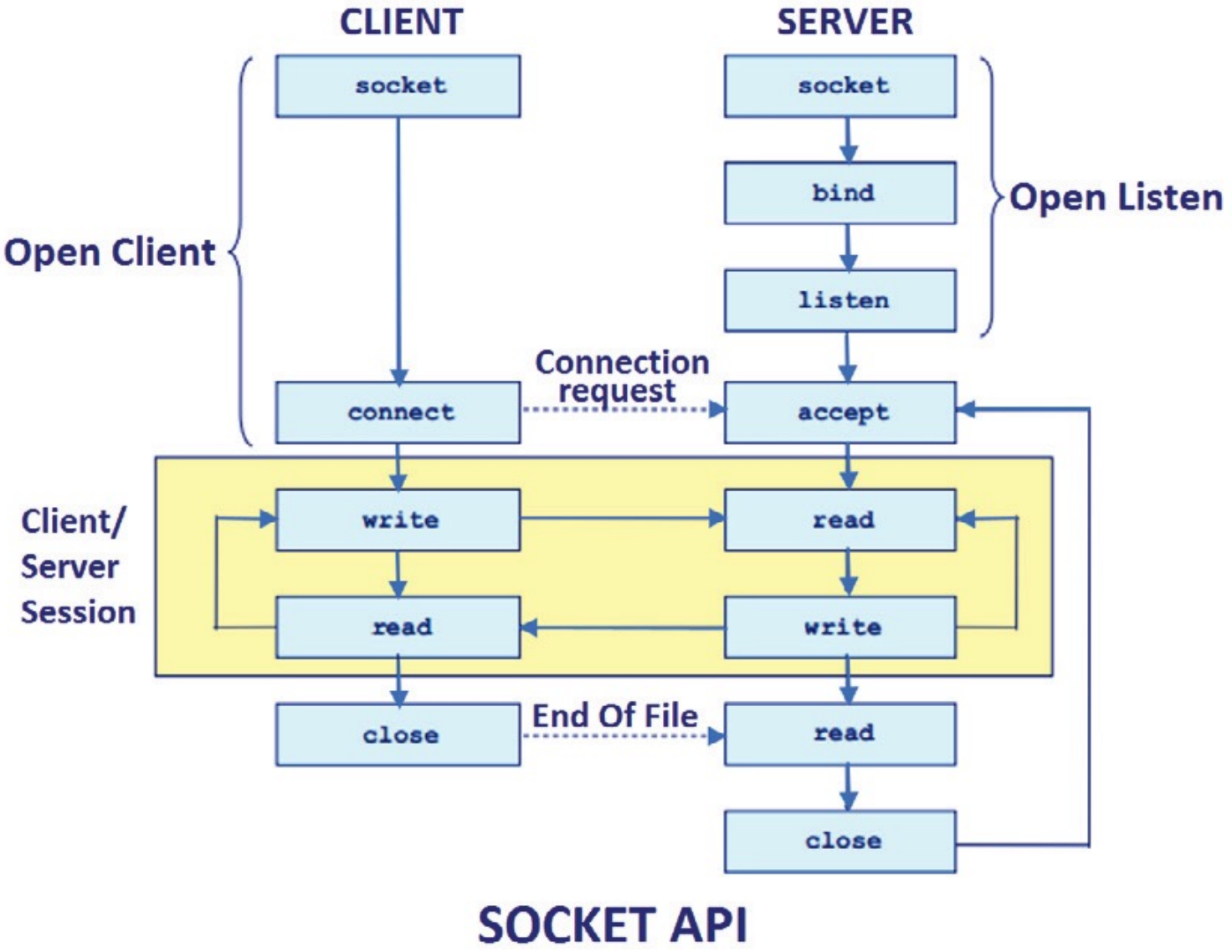
accept(): a blocking call to wait for incoming connection requests; if successful, the call returns a new socket for a separate connection

connect(): to set up a connection to a specified party

close(): to tear down a connection

send(), recv(): to send and receive (blocking receive, by default) data over a connection, respectively

Socket API


CLIENT
SERVER
socket
socket
bind
Open Listen
Open Client
listen
Connection request
connect
accept
Client/ Server Session
write
read
read
write
close
End Of File
read
close
SOCKET API

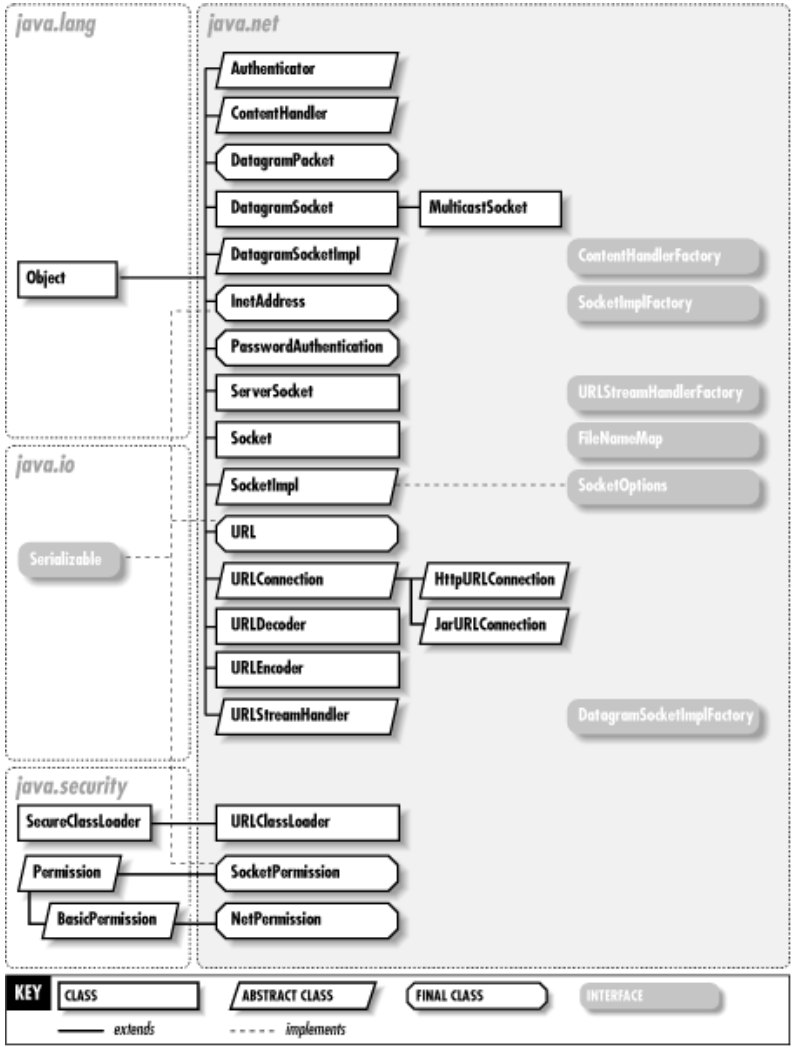
InterProcess Communication over the Internet

- IPC inside a single operating system is based on *sockets*
- IPC over the Internet includes both datagram and stream communication
- Together, these services offer support for the construction of high-level communication services and middleware

Example: in Java the package `java.net` has formats and protocols for the representation of collections of data objects in messages and of references to remote objects

Java.net

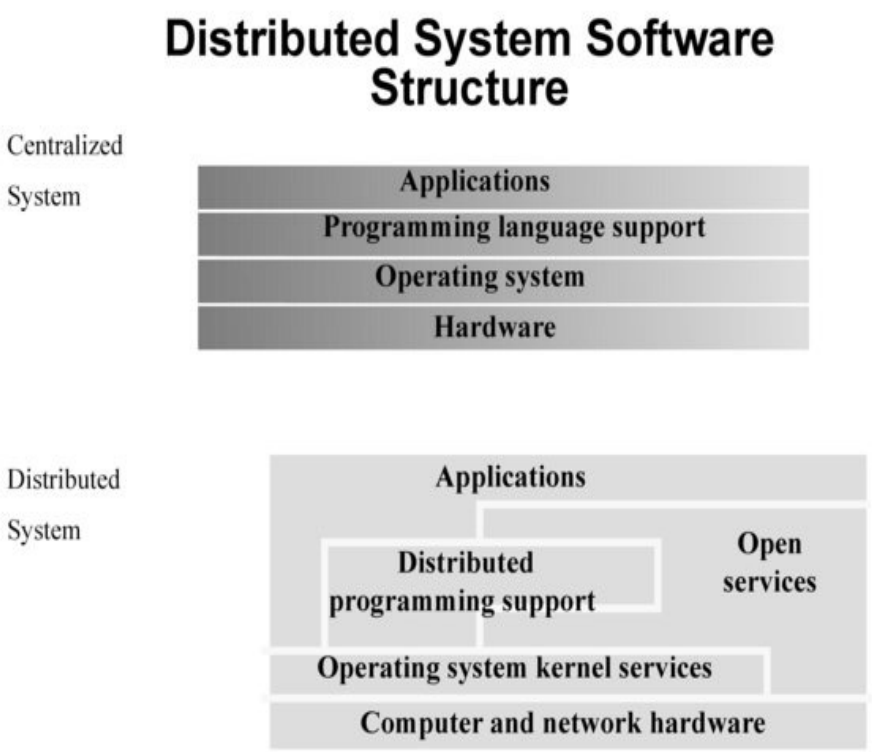
- The UDP/TCP socket APIs for Java are object oriented but are similar to the ones designed originally in the Berkeley BSD 4.x UNIX operating system
- You can consult the online Java documentation for the full specification of the classes discussed, which are in the package java.net.



https://docstore.mik.ua/orelly/java-ent/jnut/ch16_01.htm

https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html

Distributed vs centralized software structure

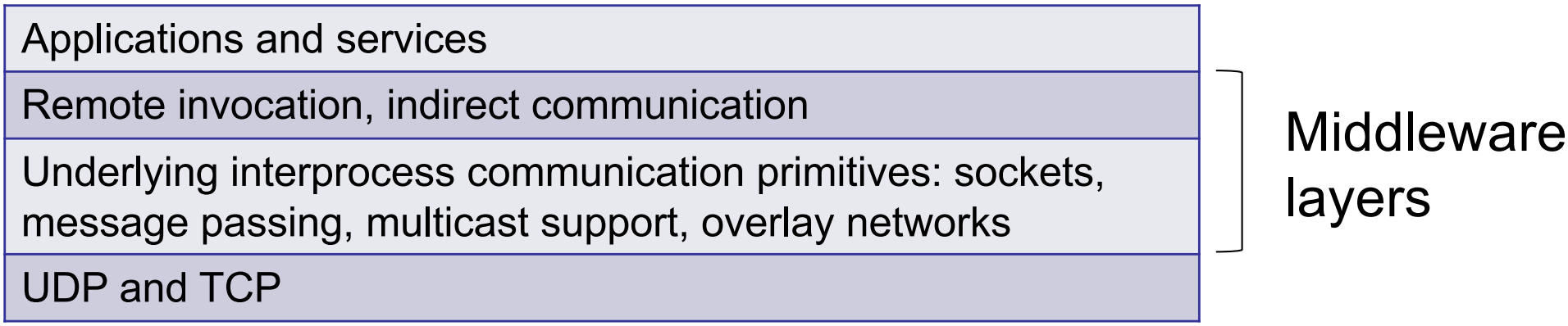


The technology stack of a distributed system includes support for IPC and some additional services

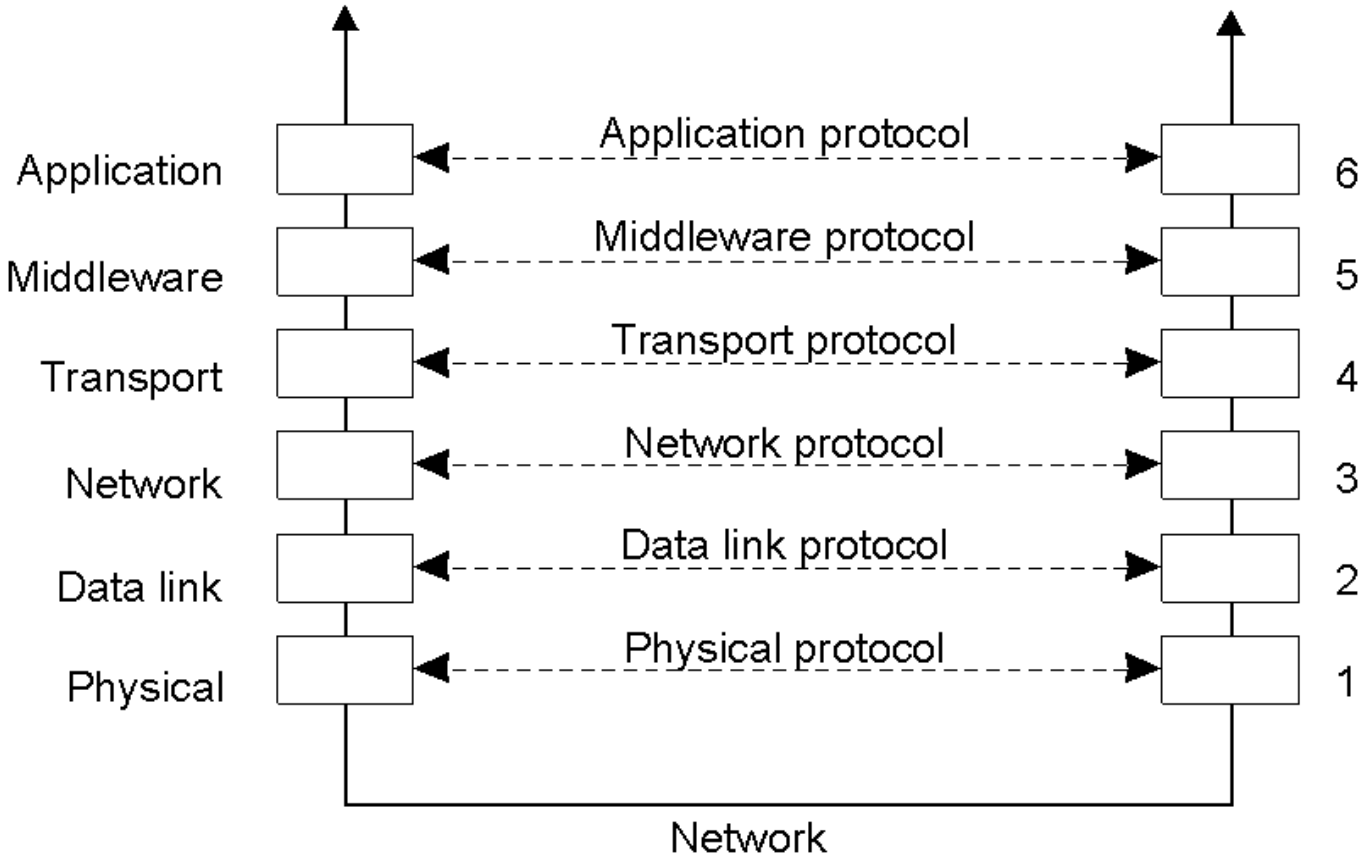
For instance the language support can give the programmer some control over process distribution and coordination

A typical service useful for distributed programming is the *naming* service (eg. DNS)

Middleware layers



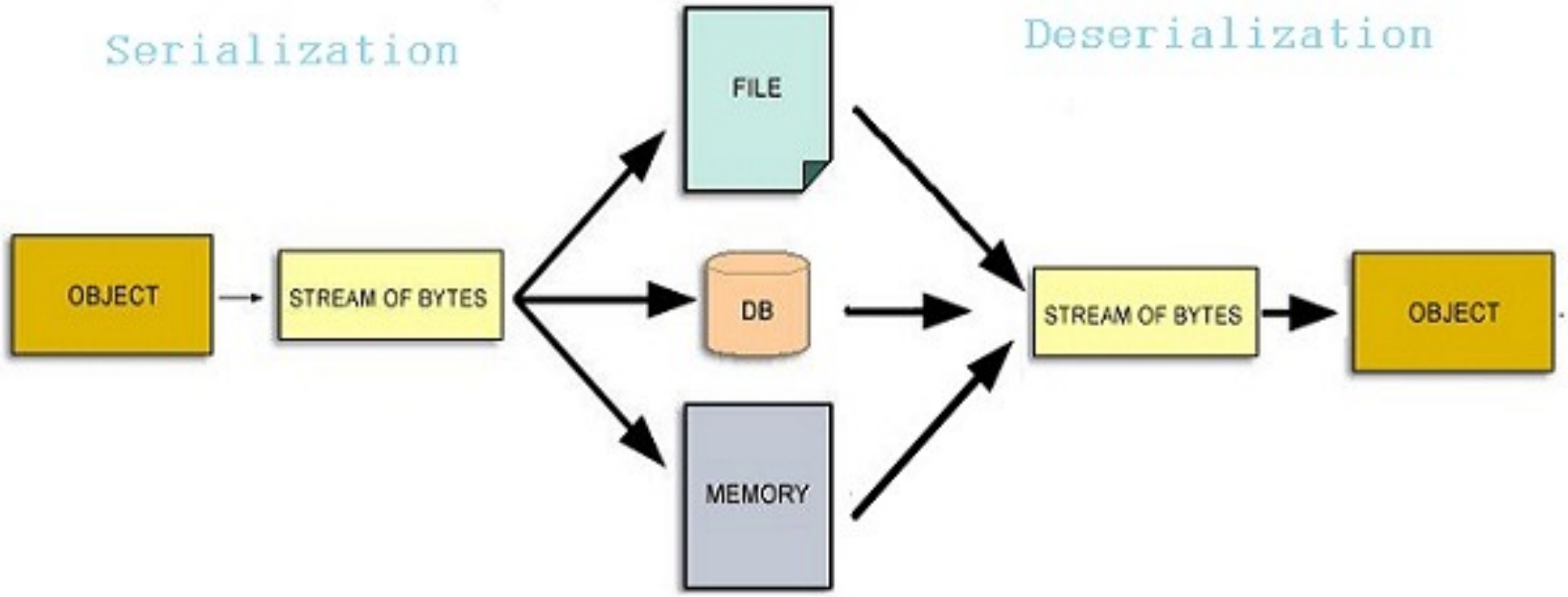
Middleware Protocols



An adapted reference model for networked communication.

Serialization of software data objects

Serialization is is the process of translating a data structure into a format (a sequence of bytes) that can be stored (for example, in a file) or transmitted (for example, over a network) and reconstructed later (possibly in a different computer)



Marshalling and unmarshalling

- Marshalling is the process of converting a software object into a data format (eg. XML), useful to store or transmit the object, and it is used when data must be moved from one program to another.
- Marshalling is similar to *serialization*, and is used to communicate to remote objects with an object, in this case a serialized object.
- Marshalling is used within implementations of different remote procedure call (RPC) mechanisms, where it is necessary to transport data between processes and/or between threads
- The inverse of marshalling is called unmarshalling (similar to deserialization)

```
public class Student {
   private char name[50];
   private int ID;
   public String getName()
                { return this.name; }
   public int getID() { return this.ID; }
   void setName(String name)
                { this.name = name; }
   void setID(int ID) { this.ID = ID; }
}
```

Java class

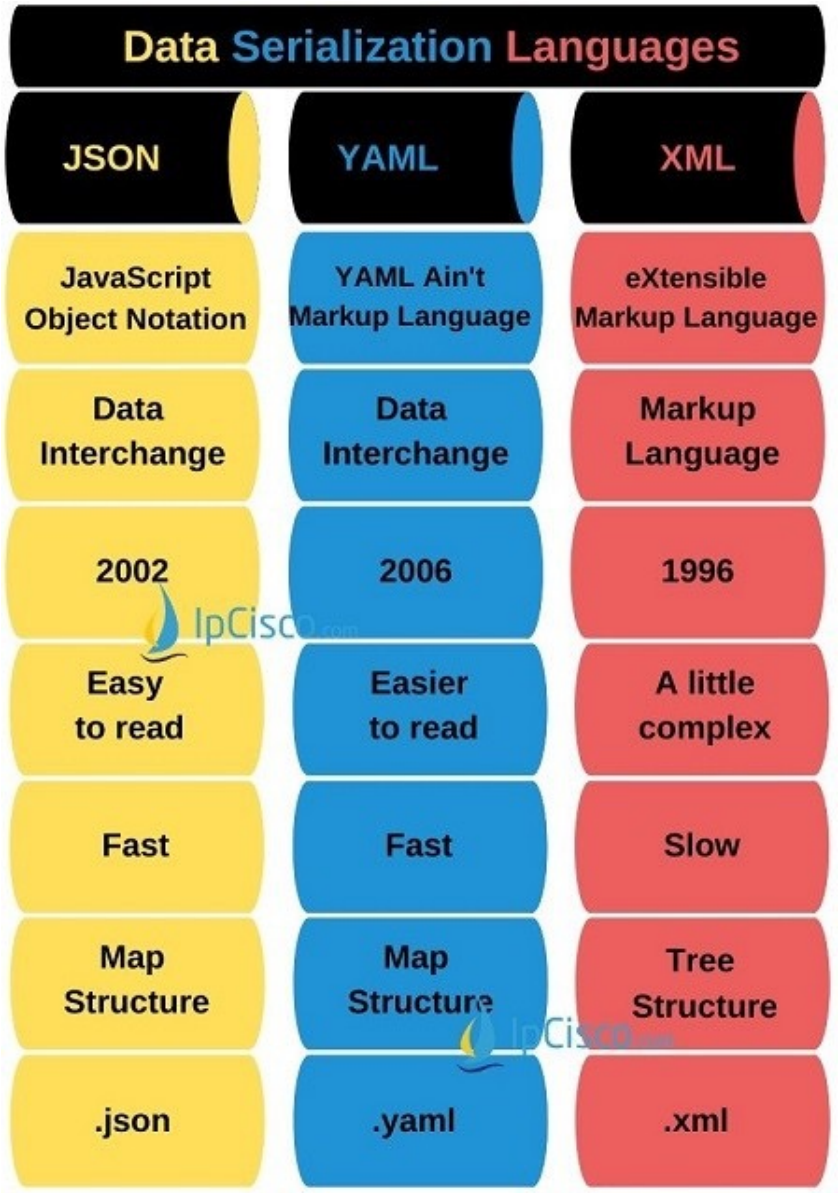
```
<?xml version = "1.0" encoding = "UTF-8"?>
<student id = "134558">
          <name>Paolo</name>
</student>
<student id = "154182">
          <name>Ivan</name>
</student>
```

XML representation of two student objects

```
Student s1 = new Student();
s1.setID(134558);
s1.setName("Paolo");
Student s2 = new Student();
s2.setID(154182);
s2.setName("Ivan");
```

Unmarshalled («executable» source code) representation of the two student objects

Data serialization languages



- Choice of data serialization format for an application depends on factors such as data complexity, need for human readability, speed and storage space constraints.
- XML, JSON, BSON, YAML, MessagePack, and protobuf are some commonly used data serialization formats.

XML definition of the Person structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

Illustration of the use of a namespace in the *Person* structure

XML Namespaces provide a method to avoid element name conflicts.

In fact, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

We use a namespace to avoid conflicts

```
<person pers:id="123456789"
            xmlns:pers = "http://www.example.net/person">
      <pers:name> Smith </pers:name>
      <pers:place> London </pers:place >
      <pers:year> 1984 </pers:year>
</person>
```

An XML schema for the *Person* structure

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions>
        <xsd:element name= "person" type ="personType" />
                <xsd:complexType name="personType">
                        <xsd:sequence>
                                <xsd:element name = "name"  type="xs:string"/>
                                <xsd:element name = "place"  type="xs:string"/>
                                <xsd:element name = "year"  type="xs:positiveInteger"/>
                        </xsd:sequence>
                        <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
                </xsd:complexType>
</xsd:schema>
```

XML is an interchange format, it is meant to be used when you have to give out or accept data from the 'outside'.
However, any XML document should be validated (dealing with invalid XML is terrible)

example of online validator: https://www.xmlvalidation.com

JSON

```
{"firstName": "John",
 "lastName": "Smith",
 "age": 25,
}
```

The JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value).
It is a common data format used for asynchronous browser–server communication

It became popular as a replacement for XML in AJAX-style systems
It was derived from JavaScript, but many programming languages include code to generate and parse JSON-format data

YAML

- YAML is a human-readable data-serialization language.
- It is commonly used for configuration files and in applications where data is being stored or transmitted.
- YAML targets many of the same communications applications as Extensible Markup Language (XML) but has a minimal syntax
- It uses both Python-style indentation to indicate nesting, and a more compact format that uses [...] for lists and {...} for maps thus JSON files are valid YAML 1.2.
- YAML has many additional features lacking in JSON, including comments, extensible data types, relational anchors, strings without quotation marks, and mapping types preserving key order.

XML vs JSON

- JSON is a **data format** whereas XML is a **markup language**
- XML is used to describe structured data and to serialize objects.
- Various XML-based protocols exist to represent the same kind of data structures as JSON for the same kind of data interchange purposes.
- Data can be encoded in XML in several ways. The most expansive form using tag pairs results in a much larger representation than JSON. Thus, JSON is faster to transmit and unmarshal
- XML has the concept of schema: this permits strong typing, user-defined types, predefined tags, and formal structure, allowing for formal validation of an XML stream in a portable way.
- XML supports comments

XML

```
<empinfo>
   <employees>
      <employee>
         <name>James Kirk</name>
         <age>40</age>
      </employee>
      <employee>
         <name>Jean-Luc Picard</name>
         <age>45</age>
      </employee>
      <employee>
         <name>Wesley Crusher</name>
         <age>27</age>
      </employee>
   </employees>
</empinfo>
```

JSON

```
{  "empinfo" :
      {
            "employees" :  [
            {
                  "name" : "James Kirk",
                  "age" : 40,
            },
            {
                  "name" : "Jean-Luc Picard",
                  "age" : 45,
            },
            {
                  "name" : "Wesley Crusher",
                  "age" : 27,
            }
                              ]
      }
}
```

Exercise Producer-Consumer System with JSON Serialization/Deserialization

Scenario: Create a producer-consumer system with two producers and one consumer. The producers will generate data, serialize to JSON, and place in a shared queue. The consumer will retrieve items from the queue, deserialize from JSON, and process them.

Requirements:

1. Create a shared queue (e.g., using Python's queue.Queue module) that can hold a limited number of JSON-serialized items (e.g., dictionaries or objects).
2. Implement two producer functions, P1 and P2, which generate random data, serialize it to JSON, and place it in the shared queue. Each producer should run in its own thread.
3. Implement a consumer function C which retrieves items from the queue, deserializes from JSON, and processes them. The consumer should run in its own thread as well.
4. Ensure proper synchronization between producers and the consumer using Python's threading mechanisms (e.g., threading.Lock).
5. Use a JSON library in Python (e.g., the json module) for serialization and deserialization.
6. Start both producer threads and the consumer thread, and demonstrate the concurrent behavior of the system.
7. Reconfigure to 3 producers and 2 consumers, and 1 shared queue. Verify what happens

Conclusions

- Internet protocols provide two alternative net protocols from which application protocols may be constructed: UDP and TCP
- There is a trade-off between the two protocols:
- UDP provides a simple message-passing facility that suffers from failures (eg some messages are lost) but carries no built-in performance penalties,
- on the other hand, in good conditions TCP guarantees message delivery, but at the expense of additional messages and higher latency and storage costs.

Conclusions

We showed some alternative styles of marshalling.

- when Java serializes data, it includes full information about the types of its contents, allowing the recipient to reconstruct it purely from the content.
- XML and JSON, like Java, include full type information.

A variety of different means are used for generating XML, depending on the context. For example, many programming languages, including Java, provide processors for translating between XML and language-level objects