Interprocess Communication

Agenda

The API for the Internet protocols

External data representation and marshalling

Multicast communication

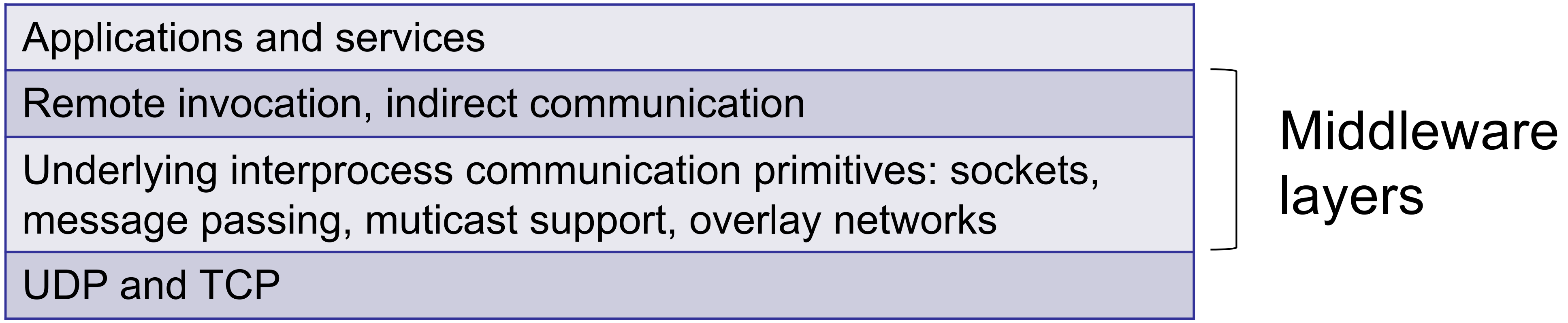
Network virtualization: Overlay networks

Case studies: MPI and Linda

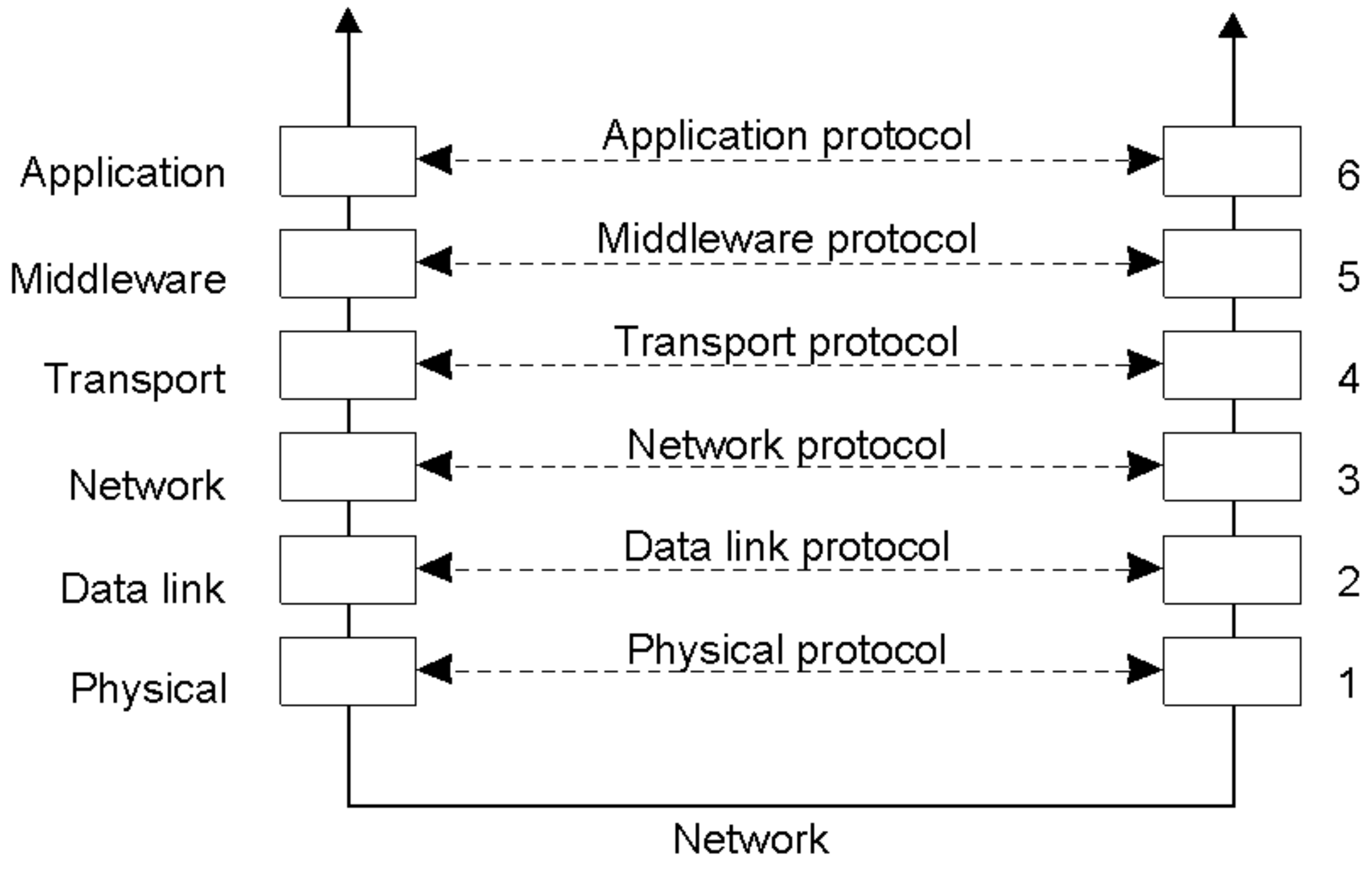
InterProcess Communication over the Internet

- InterProcess Communication inside a single operating system is based on *sockets*
- IPC over the Internet includes both datagram and stream communication. We show their use in Java as defined by the package `java.net`
- We present both formats and protocols for the representation of collections of data objects in messages and of references to remote objects
- Together, these services offer support for the construction of higher-level communication services and middleware platforms.

Middleware layers


Applications and services
Remote invocation, indirect communication
Underlying interprocess communication primitives: sockets, message passing, muticast support, overlay networks
UDP and TCP
Middleware layers

Middleware Protocols

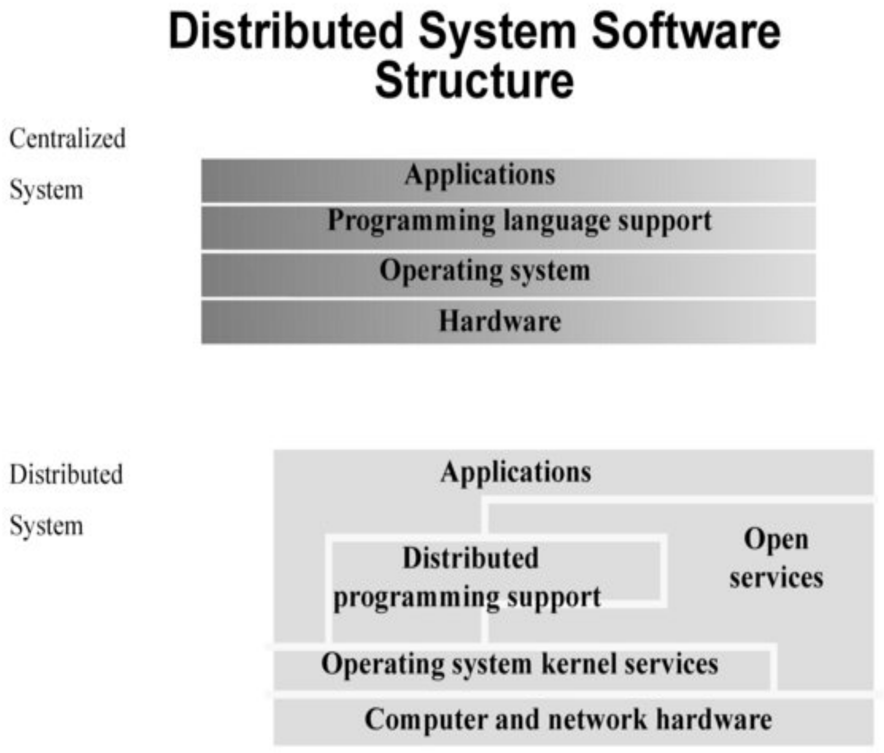


An adapted reference model for networked communication.

Interprocess Communication

- API for Internet protocols
  - sockets
  - UDP datagrams
  - TCP streams
- Marshalling
  - CORBA, Java, XML, JSON
- Multicast
- Overlay networks

Distributed vs centralized software structure


Distributed System Software Structure
Centralized System
Applications
Programming language support
Operating system
Hardware
Distributed System
Applications
Distributed programming support
Open services
Operating system kernel services
Computer and network hardware

The technology stack of a distributed system includes distributed programming support and some services

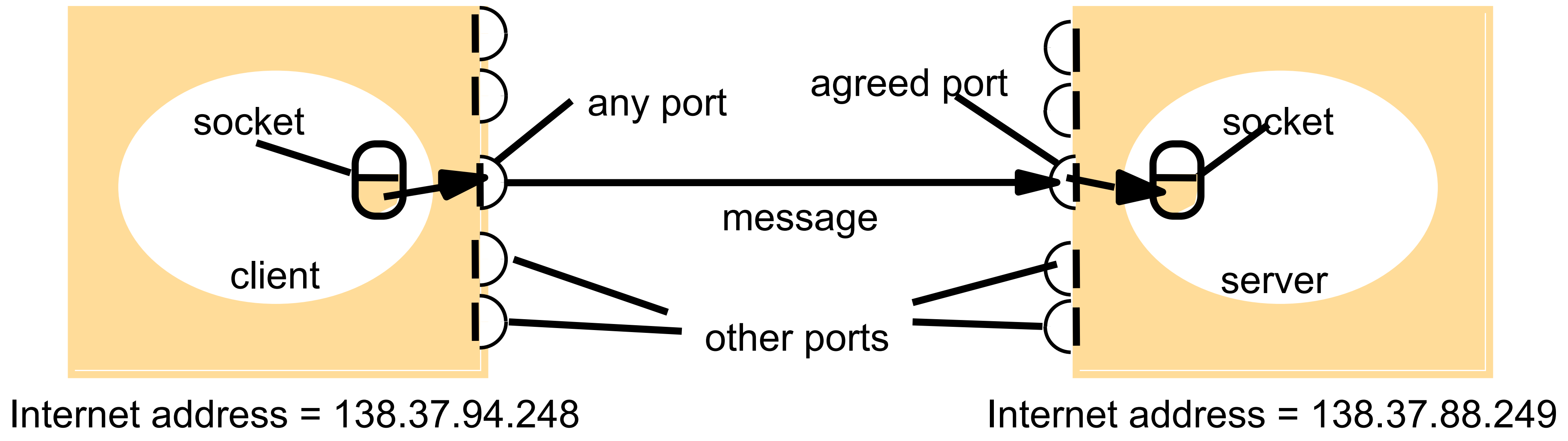
For instance the language support can give the programmer some control over process distribution and coordination

A typical service could be a naming service

POSIX (Unix) sockets

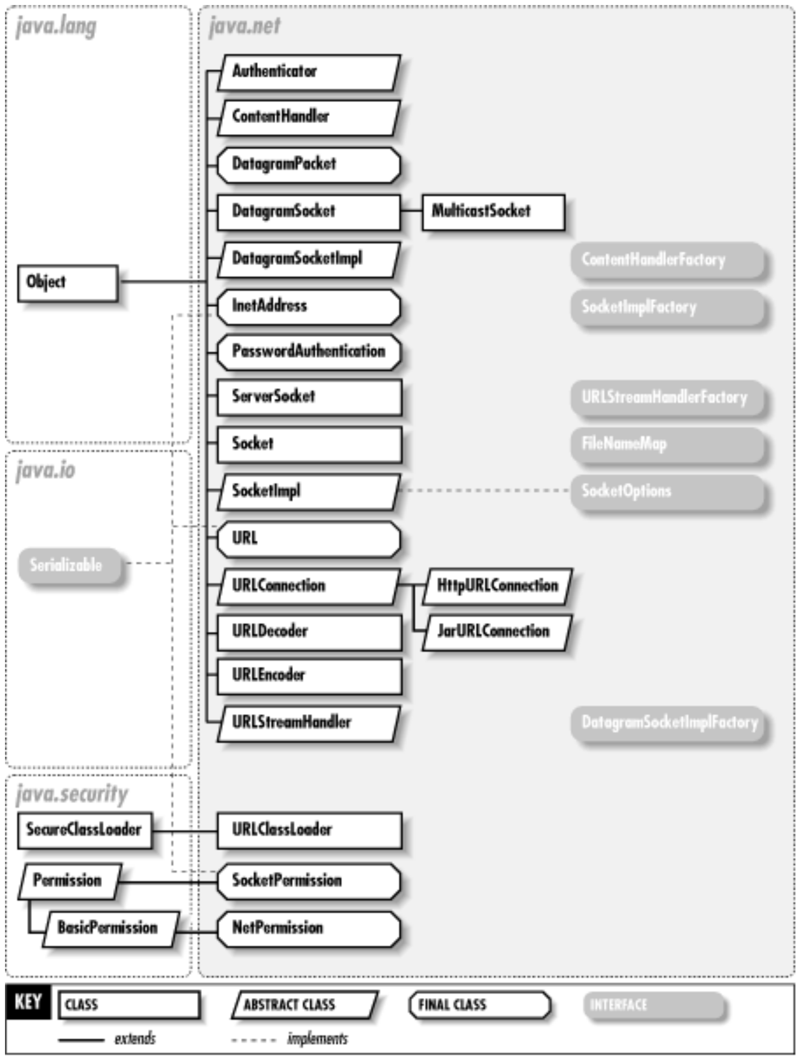
- A Unix domain socket or IPC socket is a communication endpoint for exchanging data between processes executing on the same host operating system
- Unix domain sockets support transmission of a reliable stream of bytes (SOCK_STREAM, compare to TCP).
- In addition, they support ordered and reliable transmission 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


socket
client
any port
agreed port
socket
server
message
other ports
Internet address = 138.37.94.248
Internet address = 138.37.88.249

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
- Readers studying the programming examples should 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

UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
            // args give message contents and server hostname
            DatagramSocket aSocket = null;
              try {
                        aSocket = new DatagramSocket();
                        byte [] m = args[0].getBytes();
                        InetAddress aHost = InetAddress.getByName(args[1]);
                        int serverPort = 6789;
                        DatagramPacket request = new DatagramPacket(m,  m.length(), aHost, serverPort);
                        aSocket.send(request);
                        byte[] buffer = new byte[1000];
                        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                        aSocket.receive(reply);
                        System.out.println("Reply: " + new String(reply.getData()));
              }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
              }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(aSocket != null) aSocket.close();}
  }
}
```

UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
            public static void main(String args[]){
            DatagramSocket aSocket = null;
               try{
                        aSocket = new DatagramSocket(6789);
                        byte[] buffer = new byte[1000];
                        while(true){
                          DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                          aSocket.receive(request);
                          DatagramPacket reply = new DatagramPacket(request.getData(),
                                   request.getLength(), request.getAddress(), request.getPort());
                          aSocket.send(reply);
                        }
               }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
              }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
            }finally {if(aSocket != null) aSocket.close();}
   }
}
```

TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
          try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);            // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
          }catch (UnknownHostException e){
                        System.out.println("Sock:"+e.getMessage());
          }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
          }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException e){System.out.println("close:"+e.getMessage());}}
        }
}
```

TCP server makes a connection for each client and then echoes the client’s request/1

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
               try{
                              int serverPort = 7896;
                              ServerSocket listenSocket = new ServerSocket(serverPort);
                              while(true) {
                                             Socket clientSocket = listenSocket.accept();
                                             Connection c = new Connection(clientSocket);
                              }
               } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}

// this figure continues on the next slide
```

TCP server makes a connection for each client and then echoes the client’s request/2

```
class Connection extends Thread {
            DataInputStream in;
            DataOutputStream out;
            Socket clientSocket;
            public Connection (Socket aClientSocket) {
               try {
                        clientSocket = aClientSocket;
                        in = new DataInputStream( clientSocket.getInputStream());
                        out =new DataOutputStream( clientSocket.getOutputStream());
                        this.start();
                } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
            }
            public void run(){
               try {                                         // an echo server
                        String data = in.readUTF();
                        out.writeUTF(data);
               } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
               } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
               } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
            }
}
```

Marshalling and unmarshalling

- Marshalling is the process of converting an 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>
```

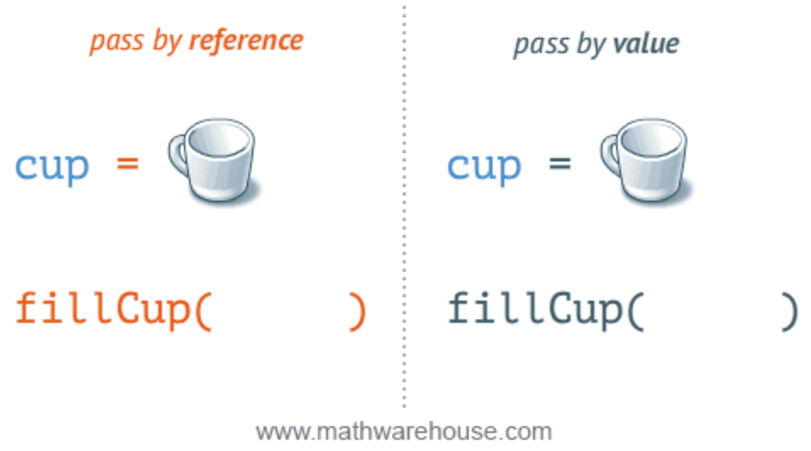
XML representation of student objects

```
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 student objects

Pass by value and pass by reference

- When an object is passed by value, this means that the object itself has been serialized and passed down to the client, where it will be deserialized and executed locally.
- When an object is passed by reference, this means that an identifier is passed down to the client that references the original object back on the server.
- When a message is sent to the referenced object, the message is then sent back to the server, where the message is then executed against the original object.



CORBA Common Data Representation (CDR) for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

CDR is the external data representation defined with CORBA 2.0
CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA.

CORBA CDR message

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1984	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Indication of Java serialized form

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

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

```
<person pers:id="123456789"
                  xmlns:pers = "http://www.cdk5.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, XML should validate (dealing with invalid XML is terrible)

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 very 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

XML vs JSON

- 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.
- XML also 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

Remote object identifier

- A remote object reference is an identifier for a remote object that is valid throughout a distributed system
- Remote object references must be generated in a manner that ensures uniqueness over space and time.
- In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system

Unicity of a remote object reference

- There are several ways to ensure that a remote object reference is unique.
- One way is to construct a remote object reference by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number.
- The local object number is incremented each time an object is created in that process; the port number and time together produce a unique process identifier on that computer
- To allow remote objects to be relocated into a different process on a different computer, the remote object reference should not be used as the address of the remote object

Local vs remote (distributed) objects

- Life cycle: Creation, migration and deletion of distributed objects is different from local objects
- Reference: Remote references to distributed objects are more complex than simple pointers to memory addresses
- Request Latency: A distributed object request is orders of magnitude slower than local method invocation
- Object Activation: Distributed objects may not always be available to serve an object request at any point in time
- Parallelism: Distributed objects may be executed simultaneously.
- Communication : There are different communication primitives available for distributed objects requests
- Failure : Distributed objects have far more points of failure than corresponding local objects.
- Security : Distribution makes objects vulnerable to attack.

Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

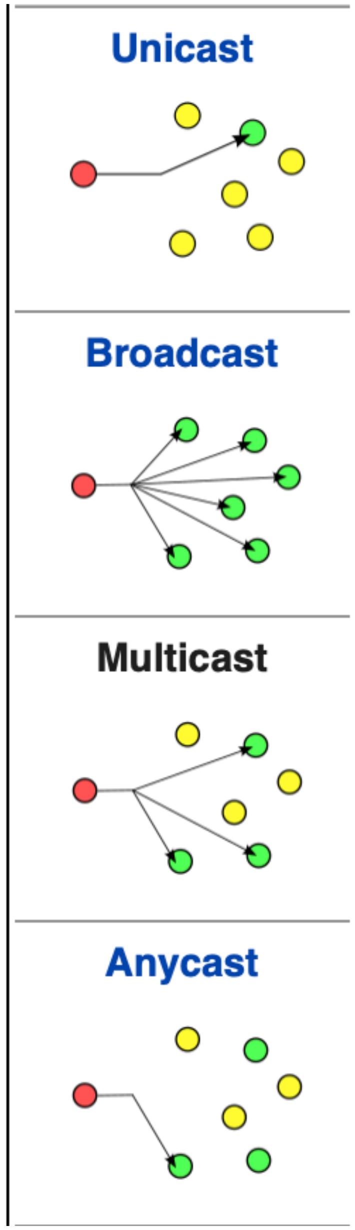
a remote object reference by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number.
The local object number is incremented each time an object is created in that process.
The port number and time together produce a unique process identifier on that computer

Multicast

- Multicast is group communication where data transmission is addressed to a group of destination computers simultaneously.
- Multicast can be one-to-many or many-to-many.
- Multicast should not be confused with physical layer point-to-multipoint communications
- IP multicast is a technique for one-to-many communication over an IP network

We use multicast for

- Fault tolerance based on replicated services
- Discovering services in spontaneous networking
- Improve performance though replicated data
- Propagation of event notifications



Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);


         // this figure continued on the next slide
```

Multicast peer joins a group (continued)

```
            // get messages from others in group
                byte[] buffer = new byte[1000];
                for(int i=0; i< 3; i++) {
                    DatagramPacket messageIn =
                            new DatagramPacket(buffer, buffer.length);
                    s.receive(messageIn);
                    System.out.println("Received:" + new String(messageIn.getData()));
                }
                s.leaveGroup(group);
            }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
            }catch (IOException e){System.out.println("IO: " + e.getMessage());}
        }finally {if(s != null) s.close();}
    }
}
```

Overlay networks

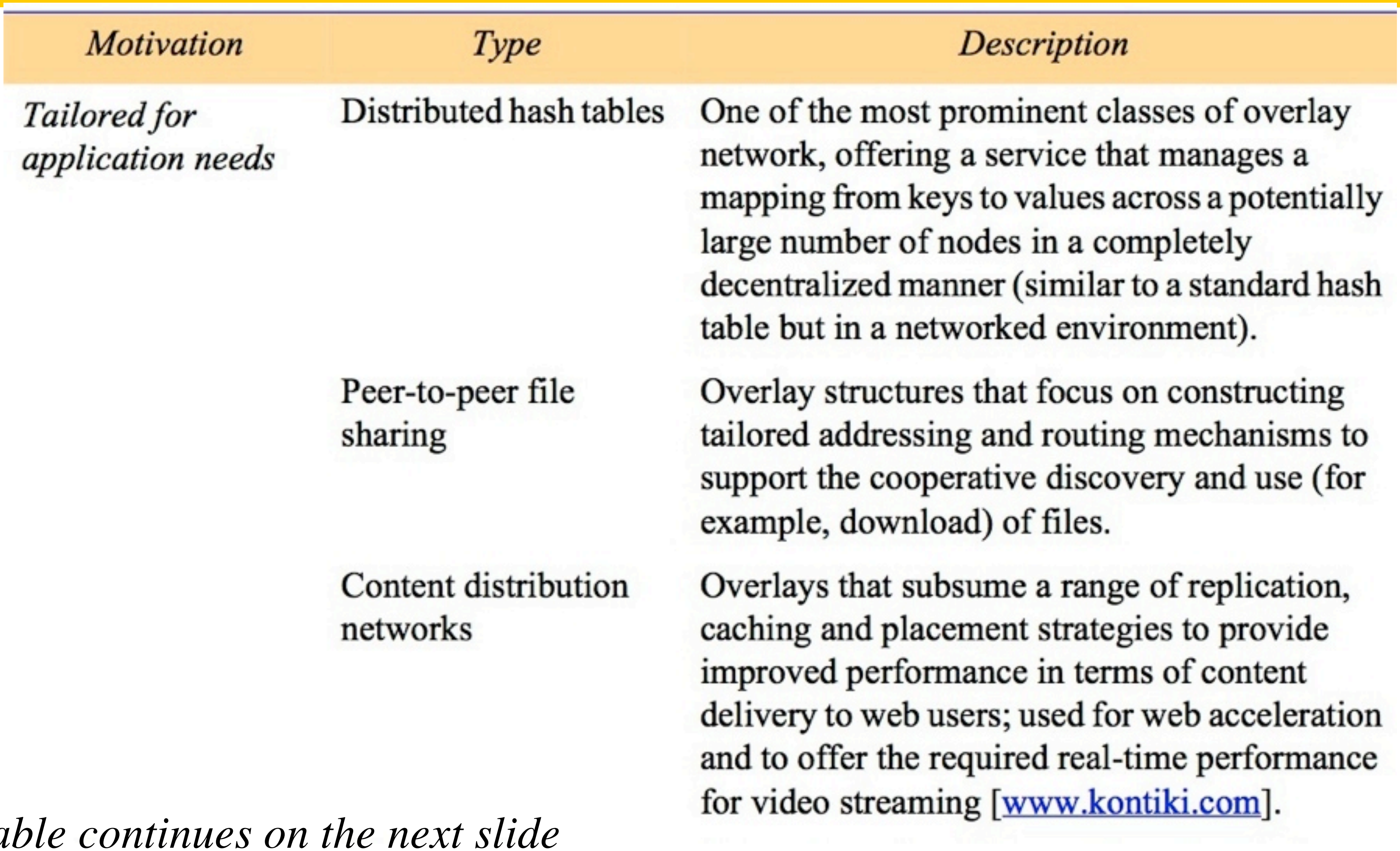
An overlay network is a virtual network consisting of nodes and links on top of an IP network offering

- a service for a class of applications: eg. multimedia content distribution
- efficient operation supporting the main use cases, eg. routing in an «ad hoc» network
- additional features, eg. multicast or secure communication

For example, Akamai Technologies manages an overlay network which provides reliable, efficient content delivery (a kind of multicast)

www.akamai.com

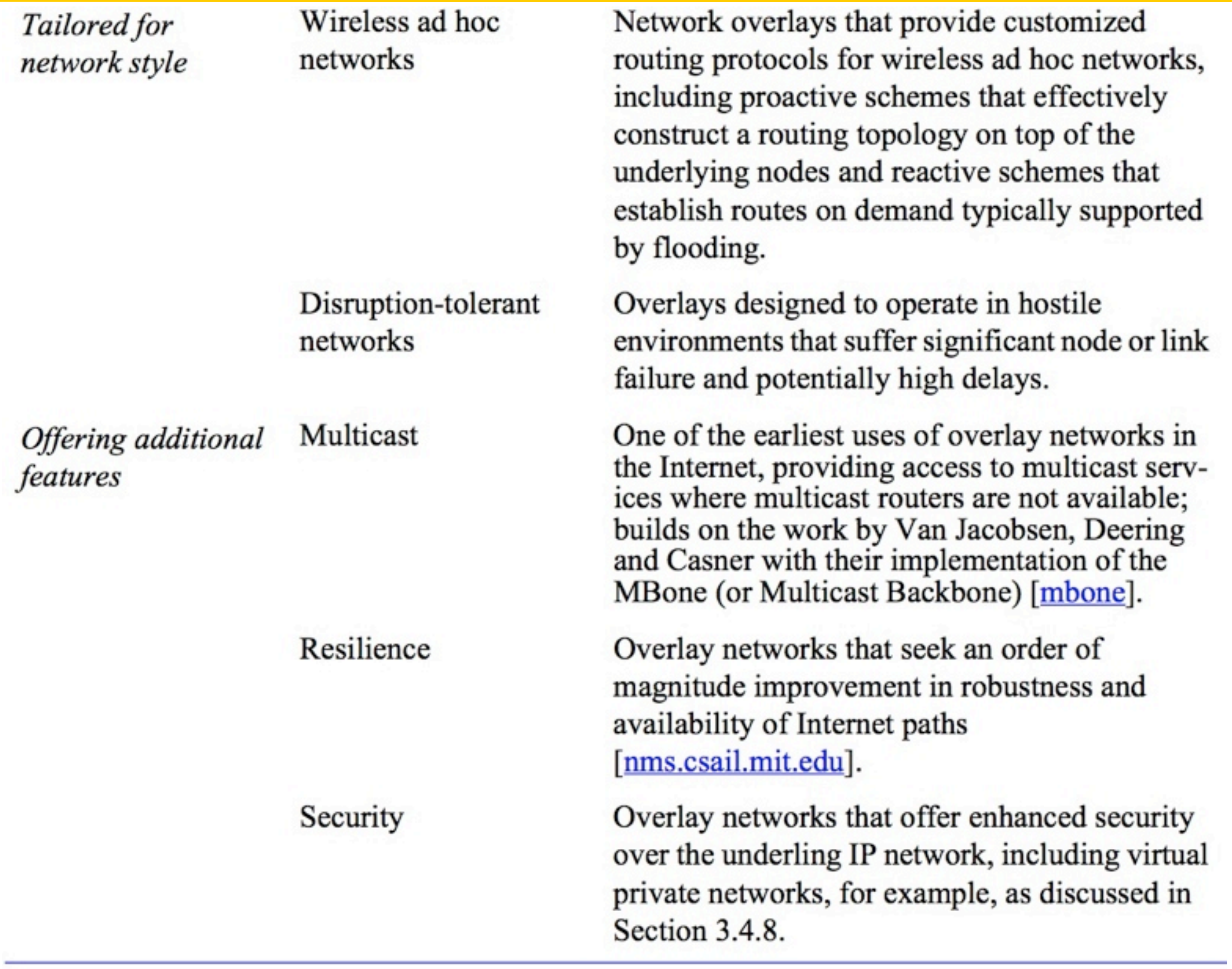
Types of overlay



Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].

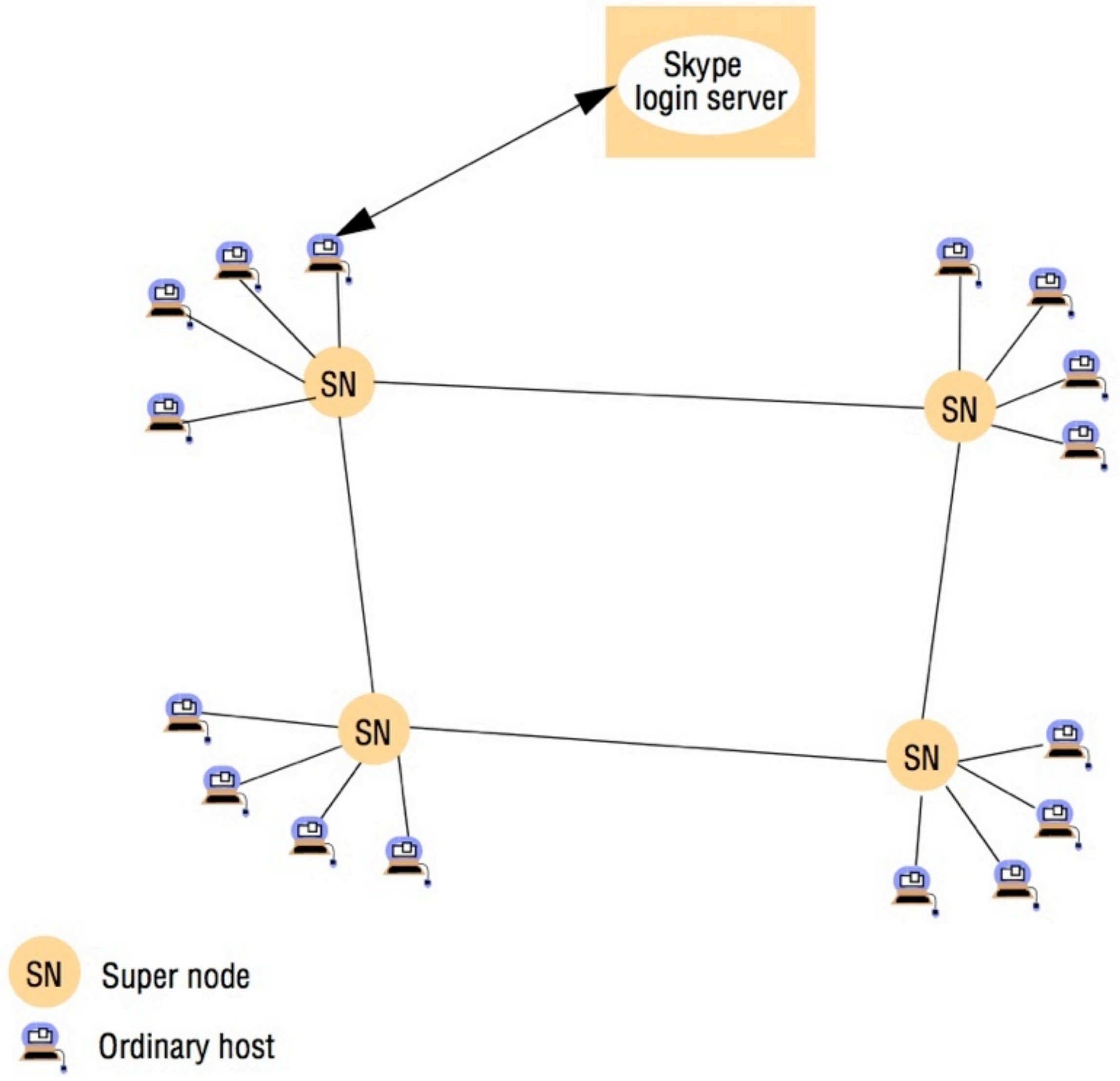
table continues on the next slide

(continued)
Types of overlay



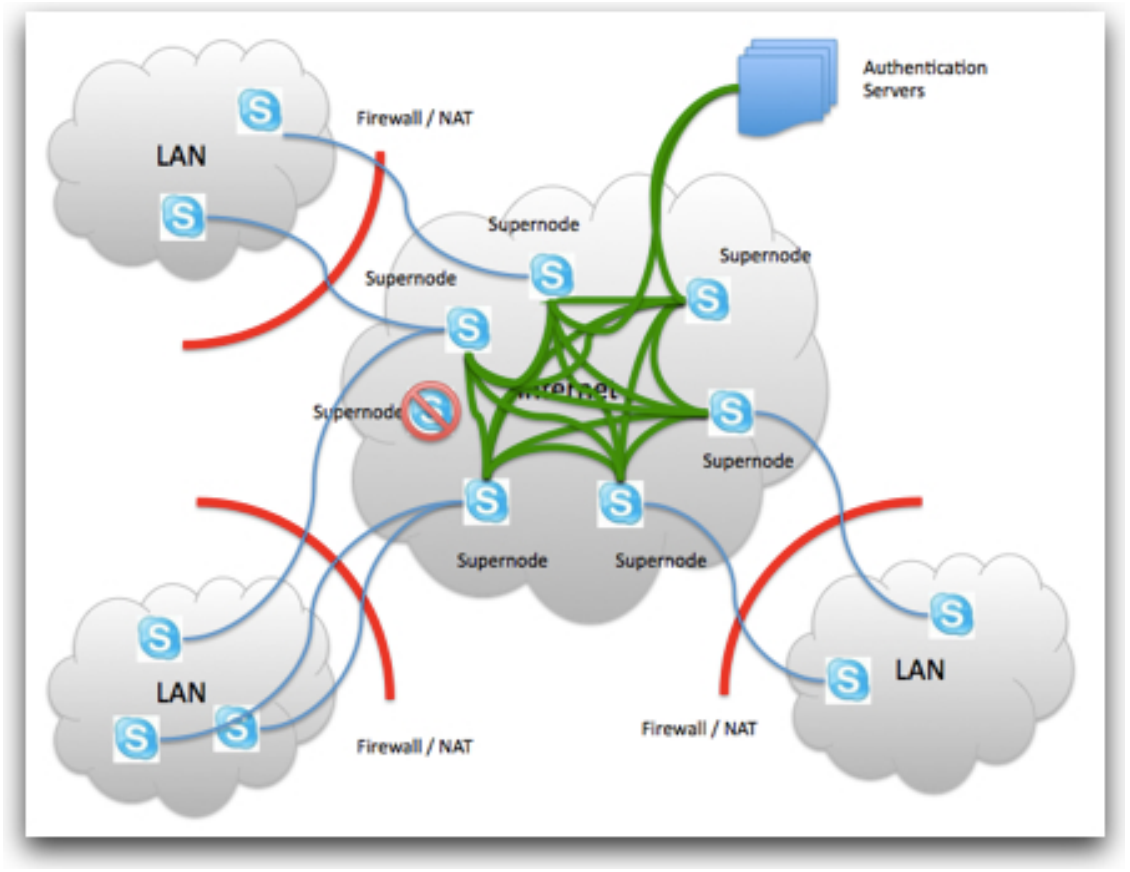
Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

Skype overlay architecture



Skype supernodes connect as peers

If you want to talk to someone, and your Skype app can’t find them immediately ... your computer or phone will first try to find a supernode to figure out how to reach them
The supernodes are connected *to each other* creating Skype's globally distributed directory database
Skype clients need to connect to some authentication servers in order to validate their username and password, and to validate their calling plan, how much money they have left in their account for calls, etc.
The cool part about the "self-healing" aspect of the supernode architecture is that if a supernode goes down, Skype clients will simply *attach to another supernode*



https://www.disruptivetelephony.com/2010/12/understanding-todays-skype-outage-explaining-supernodes.html

Languages for distributed programming

Languages with explicit distribution concepts

Languages extending other languages with distribution concepts

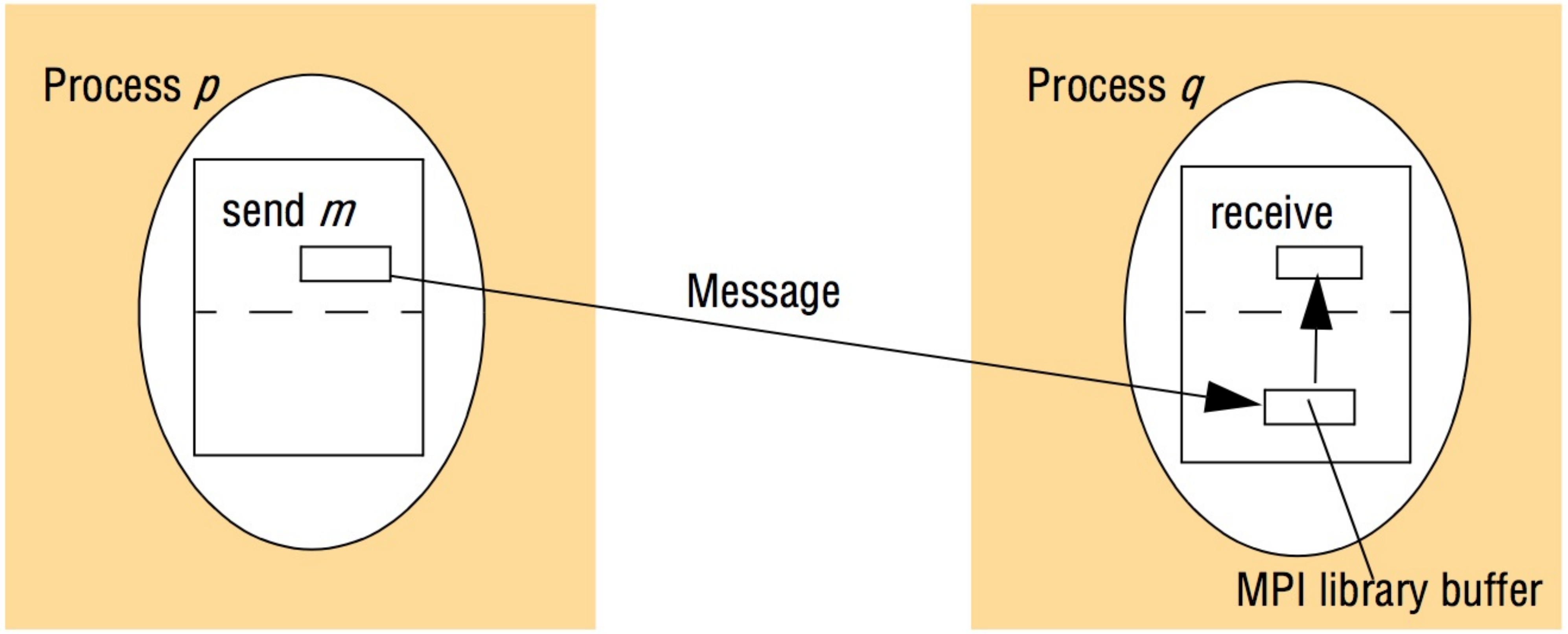
Languages implemented on top of distributed middleware (transparency)

MPI: Message Passing Interface

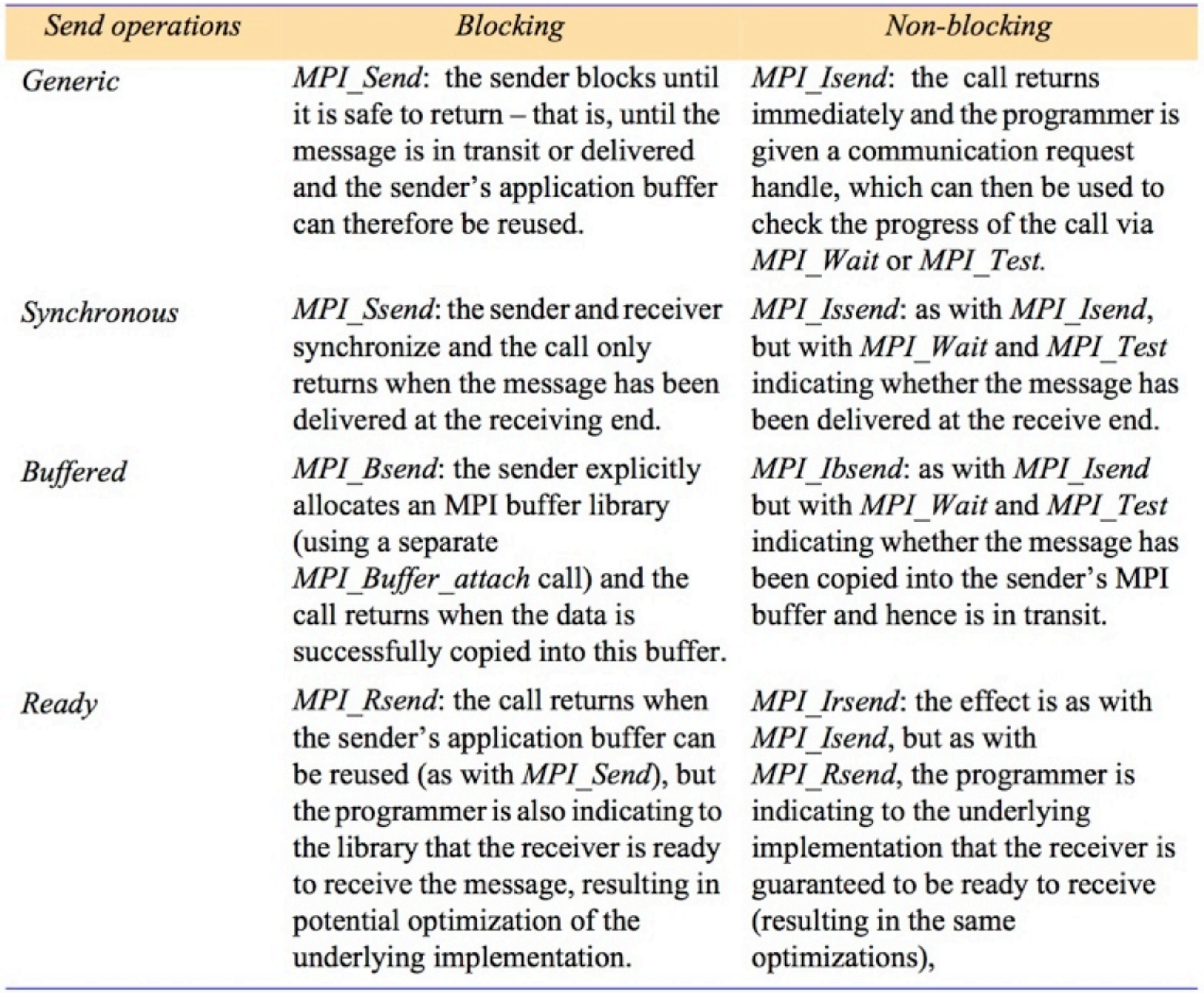
MPI was designed to be flexible, and the result is a comprehensive specification of message passing in all its variants (with over 115 operations).

Applications use the MPI interface via a message-passing library available for a variety of operating systems and programming languages, including C++ and Fortran

An overview of point-to-point communication in MPI



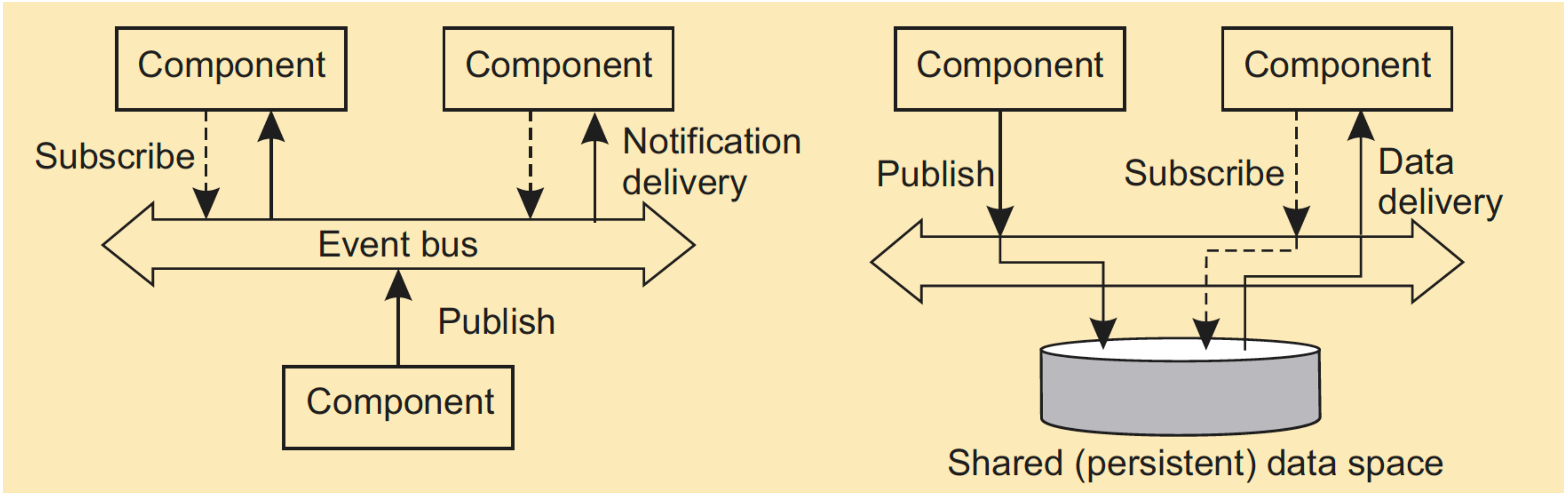
Selected send operations in MPI



Send operations	*Blocking*	*Non-blocking*
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via *MPI_Wait* or *MPI_Test*.
Synchronous	*MPI_Ssend*: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend*: as with *MPI_Isend*, but with *MPI_Wait* and *MPI_Test* indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend*: the sender explicitly allocates an MPI buffer library (using a separate *MPI_Buffer_attach* call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend*: as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend*: the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend*: the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

Temporal and referential coupling

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct send/rec	mailbox
Referentially decoupled	Event-based	Shared data space



Linda: a programming language for distributed systems

- Linda is a programming language developed at Yale Univ. by Gelernter and Carriero
- It offers an abstract virtual shared memory to support parallelism and coordination on heterogeneous networks
- Simple language primitives added to a sequential language to allow minimal code rewriting:
  - `In()` `Out()` move “tuples” from/to TupleSpace
  - `Rd()` performs non-destructive reads in T.S.
  - `Inp()` `Rdp()` non-blocking versions of `In()` `Rd()`
  - `Eval()` spawns processes in Tuplespace

Example: Linda Tuple Space

Three simple operations

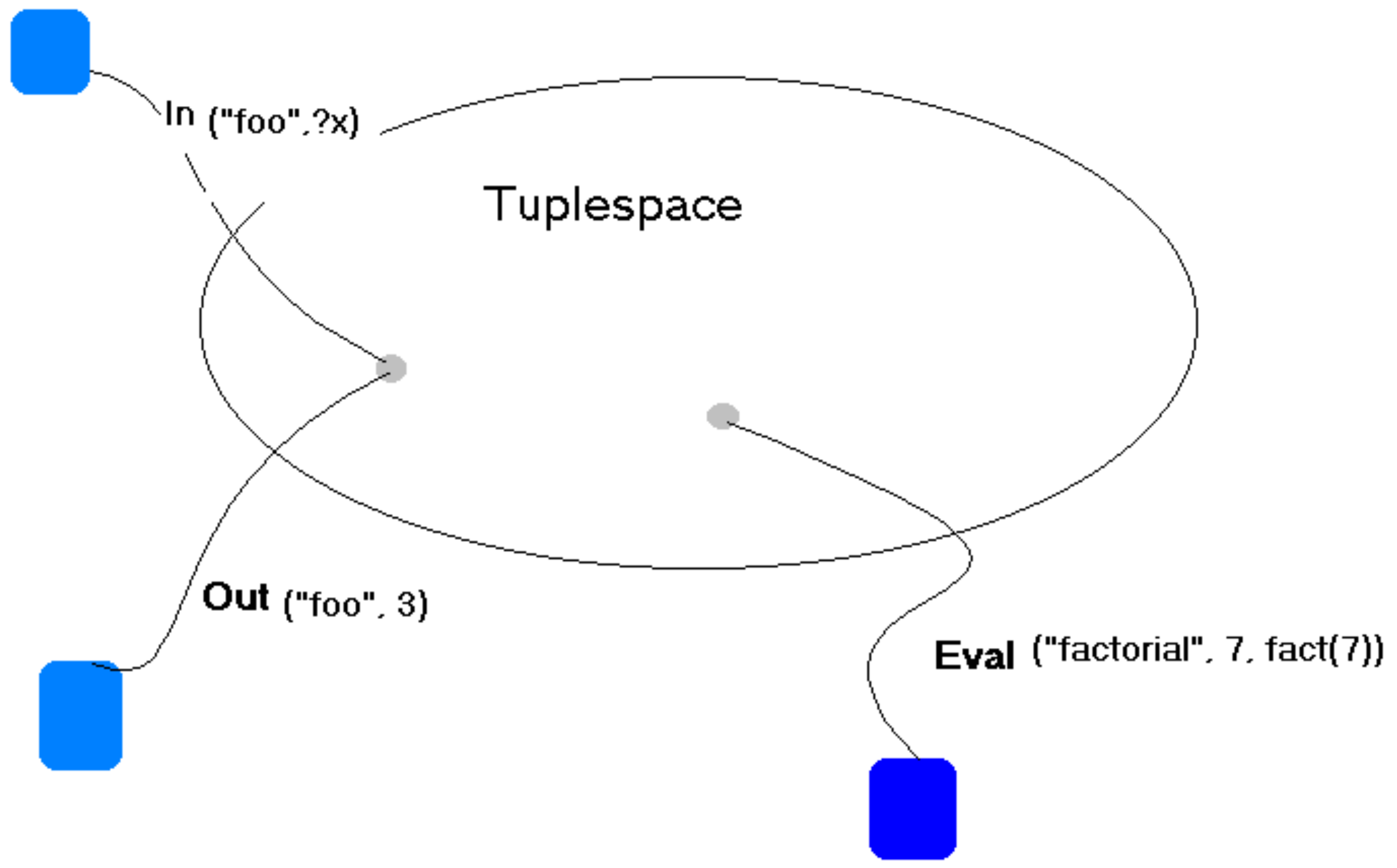
- `in(t)`: remove a tuple matching template `t`
- `rd(t)`: obtain copy of a tuple matching template `t`
- `out(t)`: add tuple `t` to the tuple space

More details

- Calling `out(t)` twice in a row, leads to storing two copies of tuple `t` $\Rightarrow$ a tuple space is modeled as a multiset.
- Both `in` and `rd` are blocking operations: the caller will be blocked until a matching tuple is found, or has become available.

Associative addressing through efficient hashing

Tuplespace



Example: Linda Tuple Space

Bob

```python
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("bob","distsys","I am studying chap 2"))
blog._out(("bob","distsys","The linda example's pretty simple"))
blog._out(("bob","gtcn","Cool book!"))
```

Alice

```python
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("alice","gtcn","This graph theory stuff is not easy"))
blog._out(("alice","distsys","I like systems more than graphs"))
```

Chuck

```python
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

t1 = blog._rd(("bob","distsys",str))
t2 = blog._rd(("alice","gtcn",str))
t3 = blog._rd(("bob","gtcn",str))
```

C-Linda Language

The C-Linda programming language is the combination of sequential C and the Linda language.

The C-Linda is one realization of the Linda model.

The C-Linda supports the six Linda functions.

C-Linda Functions

Out(t) – causes tuple t to be added to tuple space

```
Out(“a string”, 15.01, 17, x)
Out(0, 1)
```

In(s) – causes some tuple t that matches anti-tuple s to be withdrawn from tuple space.

```
In(“a string”, ?F, ?I, y)
```

Rd(s) - is the same as In(s) except that the matched tuple remains in tuple space.

C-Linda Functions

- Eval(t) is the same as out(t), except that it sends an *active* tuple into Tuplespace.

For example,

```
out(“foo”, 3, 4.3)
Eval(“factorial”, 7, fact(7))
```

C-Linda Functions

Inp and Rdp – predicate versions of in and rd:_ they attempt to locate a matching tuple and return 0 if they fail; otherwise they return 1.

Example of Tuple matching

N-element vector stored as n tuples in the Tuple space

(“V”, 1, FirstElt)

(“V”, 2, SecondElt)

…

(“V”, n, NthElt)

To read the jth element and assign it to x, use

Rd(“V”, j, ?x)

```
/* hello.clc: a simple C-Linda program */
 #include "linda.h"

int worker(int num) {
    int i;
    for (i=0; i<num; i++)
        out("hello, world");
    return 0;
}

int real_main(int argc, char *argv[]) {
    int result;
    eval("worker", worker(5));
    in("hello, world");
    in("hello, world");
    in("hello, world");
    in("hello, world");
    in("hello, world");
    in("worker", ? result);
    return 0;
}
```

Benefits of C-Linda

Anonymous and asynchronous communication

Associative addressing / pattern matching

Data persistence independent of creator

Simple API → less and easier coding

Ease of code transformation

C-Linda limitations

- High system overhead
- Designed as parallel computing model
- Lack of security model
- Lack of transaction semantics
- Language specific implementation
- Blocking calls, but no notification mechanism

JavaSpaces (Sun Micro)

- Java library (middleware) derived from Linda model
- Lightweight infrastructure for network applications
- Distributed functionality implemented through RMI
- Entries written to/from JavaSpace with “write, read”
- “notify” notifies client of incoming entries w/ timeout

JavaSpace

- Pattern Matching done to templates with class type comparisons, no comparison of literals.
- Transaction mechanism with a two phase commit model
- Entries are writtern with a “lease”, so limited persistence with time-outs.

JavaSpace see http://www.oracle.com/technetwork/articles/java/javaspaces-140665.html

Simplicity of security model

Java RMI = performance bottleneck

write(): Writes new objects into a space

take(): Retrieves objects from a space

read(): Makes a copy of objects in a space

notify: Notifies a specified object when entries that match the given template are written into a space

Both the read() and take() methods have variants: readIfExists() and takeIfExists(). If they are called with a zero timeout, then they are equivalent to their counterpart. The timeout parameter comes into effect only when a transaction is used

```
package examples.spaces;

import net.jini.impl.outrigger.binders.RefHolder;
import net.jini.lease.Lease;
import net.jini.space.JavaSpace;
import net.jini.transaction.Transaction;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class HelloWorld {
/** * Get a remote reference to a particular space */
public JavaSpace getSpace() {
   try {
      // RefHolderImpl is the remote object registered with the registry
           RefHolder rh = (RefHolder)Naming.lookup("JavaSpace");

      // Use the RefHolder's proxy method to get the space reference
      JavaSpace js = (JavaSpace)rh.proxy();
      return js;
   } catch (Exception e) {
      System.err.println(e.getMessage());
   }
   return null;
}
```

Other Tuple Space Implementations

- Java Messaging Service(JMS), a Message-Oriented Middleware (MOM) layer API, shares many of the same architectural strengths of JavaSpaces while offering some added benefits.
- Tspaces (IBM Almaden)
- PageSpace and Jada
- LIME
- XML Dataspaces
- Mobile Agent Reactive Space (MARS)

Conclusions on the Linda model and similar architectures

- “Tuple-space”-based architectures are flexible, providing most of the core infrastructure necessary to support distributed, mobile, and intelligent software agents.
- The language is intuitive, but offers minimal security.
- The original Linda is not fault-tolerant

Conclusions

- Internet transmission protocols provide two alternative building blocks from which application protocols may be constructed.
- There is a trade-off between the two protocols: UDP provides a simple message-passing facility that suffers from omission failures 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.

- CORBA choose to marshal data for use by recipients that have prior knowledge of the types of its components.
- 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.
- Another big difference is that CORBA requires a specification of the types of data items to be marshalled (in IDL) in order to generate the marshalling and unmarshalling methods, whereas Java uses reflection in order to serialize objects and deserialize their serial form.

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

Exercise

Describe a scenario in which a client could receive a reply from an earlier call.

Exercise

A server creates a port which it uses to receive requests from clients.

Which are the design issues concerning the relationship between the name of this port and the names used by clients?

Exercise

Compare and contrast web services with distributed object approaches in terms of the following:

- Marshalling and external data representation
- Interoperability
- Security
- Reliability
- Performance
- Remote references
- Full OOP
- Describe how the protocols of the internet allow for heterogeneity.
- Describe how middleware allows for heterogeneity.

Solution

The main design issues for locating server ports are:

(i) How does a client know what port and IP address to use to reach a service?

The options are:

- use a name server/binder to map the textual name of each service to its port;
- each service uses well-known location-independent port id, which avoids a lookup at a name server. The operating system still has to look up the whereabouts of the server, but the answer may be cached locally.

(ii) How can different servers offer the service at different times? Location-independent port identifiers allow the service to have the same port at different locations. If a binder is used, the client needs to reconsult the client to find the new location.

(iii) Efficiency of access to ports and local identifiers. Sometimes operating systems allow processes to use efficient local names to refer to ports. This becomes an issue when a server creates a non-public port for a particular client to send messages to, because the local name is meaningless to the client and must be translated to a global identifier for use by the client and the names used by clients.