Distributed Systems – CS249

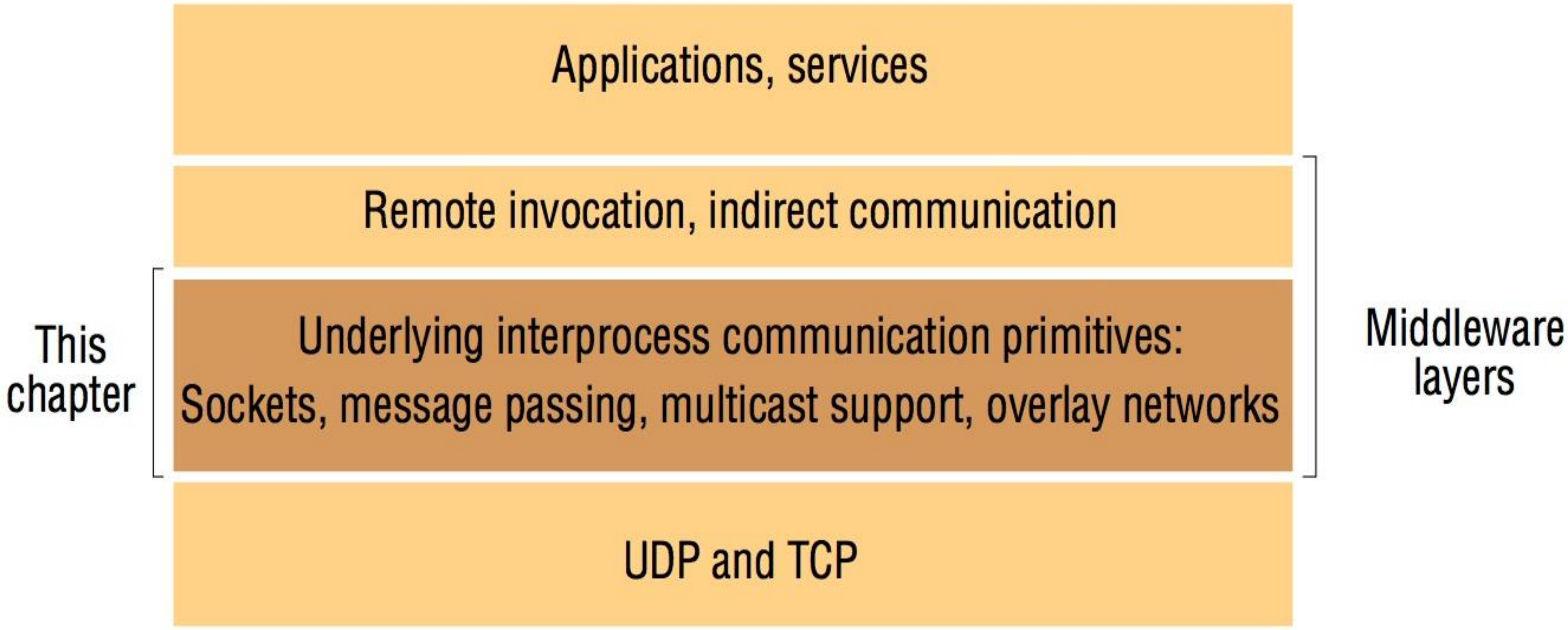
Interprocess Communication

Agenda

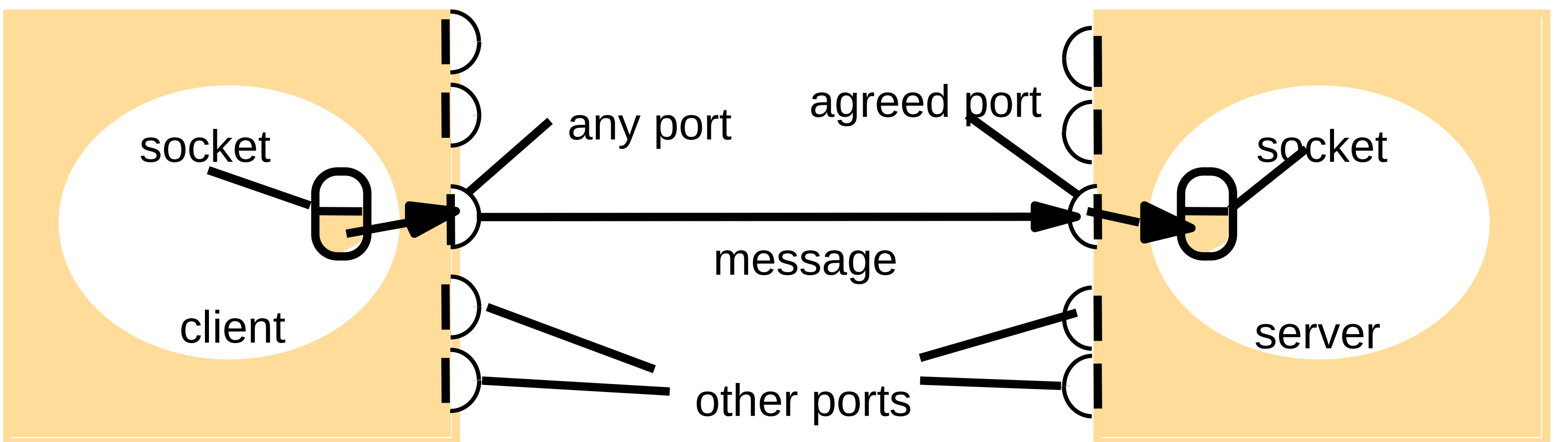
- Interprocess Communication
- Remote Invocation
- Indirect Communication
- Operating System Support
- Case Study: RPC

Interprocess Communication

Interprocess Communication: Middleware Layers



Interprocess Communication: Sockets and Ports



Internet address = 138.37.94.248

Internet address = 138.37.88.249

Interprocess Communication: UDP client sending message to 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();}
  }
}
```

Interprocess Communication: 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 – Universal Transfer Format
                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());} }
        }
}
```

Interprocess Communication: TCP server make a connection for each client and then echoes the client's request

```
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
```

Interprocess Communication: TCP server make a connection for each client and then echoes the client's request (Contd.)

```
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*/}}
        }
}
```

Interprocess Communication:
CORBA 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 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: Common Data Representation

Interprocess Communication:
CORBA CDR message

```
struct  Person {
  string  name;
  string  place;
  unsigned long  year;
};
```

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}

Interprocess Communication: Indication of java serialized form

```
struct  Person {
   int      year;
   string  name;
   string  place;
};
```

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
- Serialized form of an object is appropriate to store on disk or send in a message. Deserialization restores the state of an object from its serial form. The process of deserialization has no knowledge of the object type and as a result the serial form need to include additional info to help interpreting the serial form.
- Object may contain reference to another object; the object reference is serialized into a handle.
- To serialize an object all objects referenced must be serialized as well. Each class is given a handle, and no class is written more than once to the serial form (instead the handle is written to indicate the class)

Interprocess Communication:
XML definition of the Person structure

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

- “id” is a class attribute while <name, place, year> are elements of the class person.
- Name, place and year are tags.

Interprocess Communication:
Use of a name space 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>
```

- XML namespace provides a means to scope names. It is a set of names for collection of element types and attributes that is referenced by a URL.
- Any XML document can use an XML namespace by referring to its URL.

Interprocess Communication:
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

- Such representation is relevant only to languages that support remote objects, e.g.g., Java and CORBA.
- Remote object reference (identifier) is unique in a distributed system (space and time), i.e., even after the remote object is deleted – never reused.
- In the above representation: time is object creation time. Object number is unique local number within the host where the remote object reside.
- To allow object to migrate, you should not use object reference as the address of the remote object
- Interface of remote object includes interface name, and supported methods

Interprocess Communication: Multicast peer joins a group & sends/receive 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
```

Interprocess Communication: Multicast peer joins a group & sends/receive datagrams

```
        // 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();}
   }
}
```

Interprocess Communication:
Types of overlays (virtual networks)

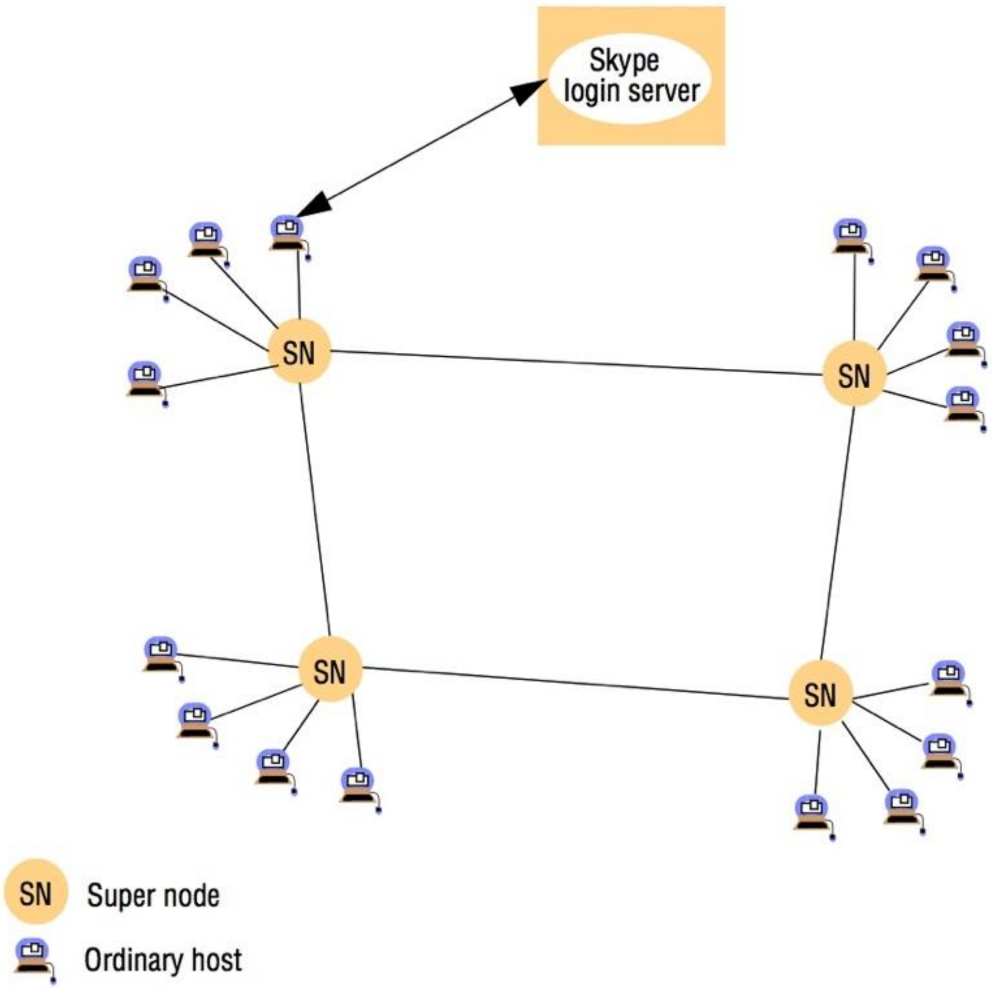
Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard 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

Interprocess Communication: Types of overlays

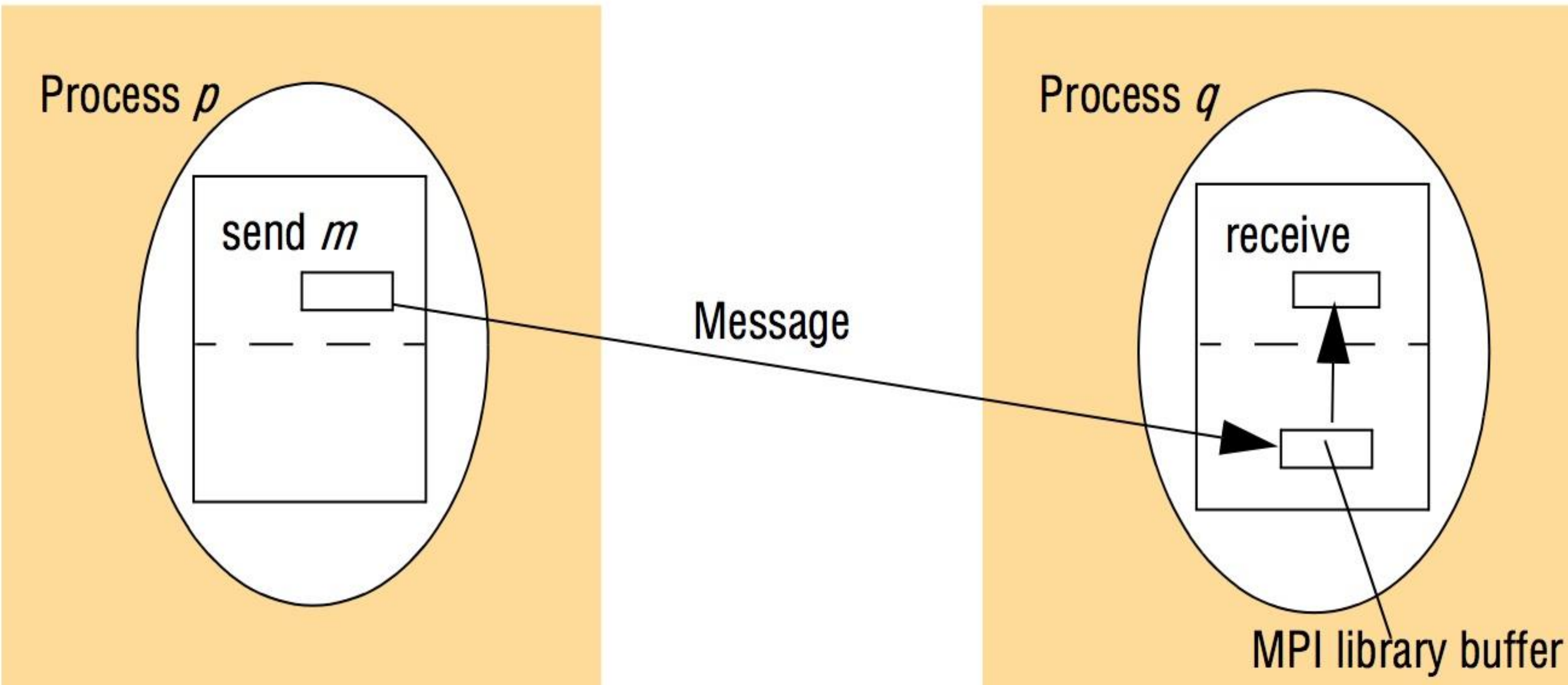
Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically 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.

Interprocess Communication:
Skype overlay architecture



- Building your own network with your desired attributes over existing networks
- Skype is P2P network offering VoIP service. It includes also: instant messaging, video conferencing, and interface to telephony service
- Skype does not require any changes to the Internet.

Interprocess Communication: An overview of point-to-point Communication in MPI



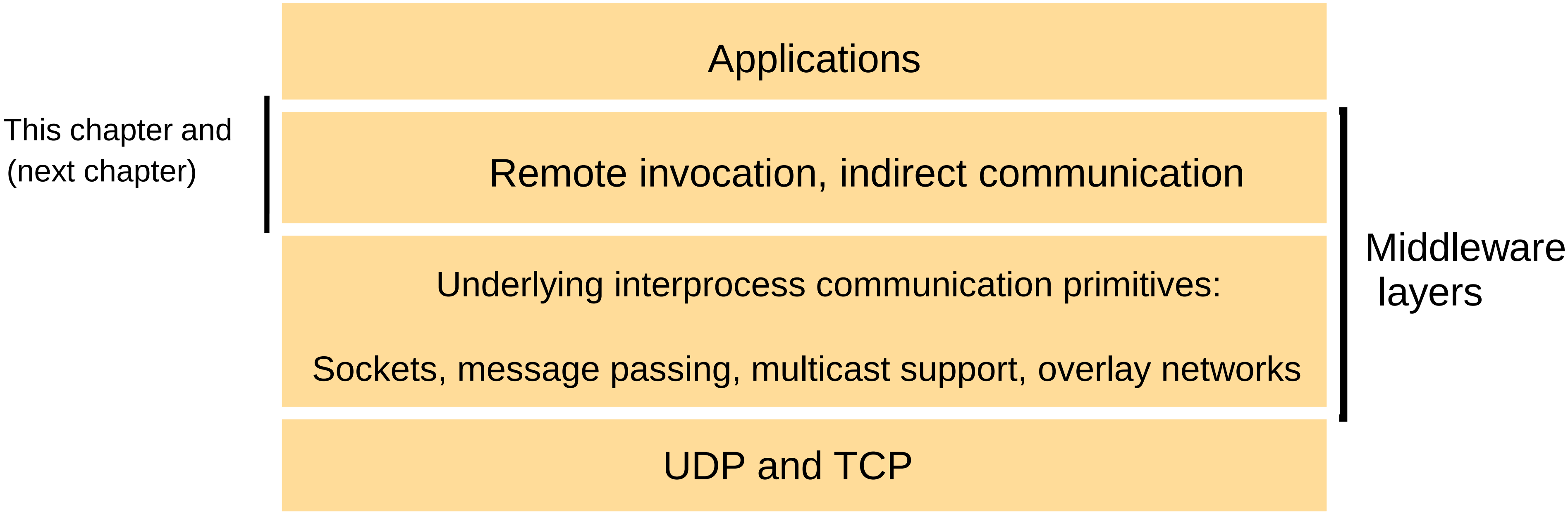
- MPI (Message Passing Interface) is lightweight MP used with High-Performance Computing for distributed programming

Interprocess Communication: Selected send operation in MPI

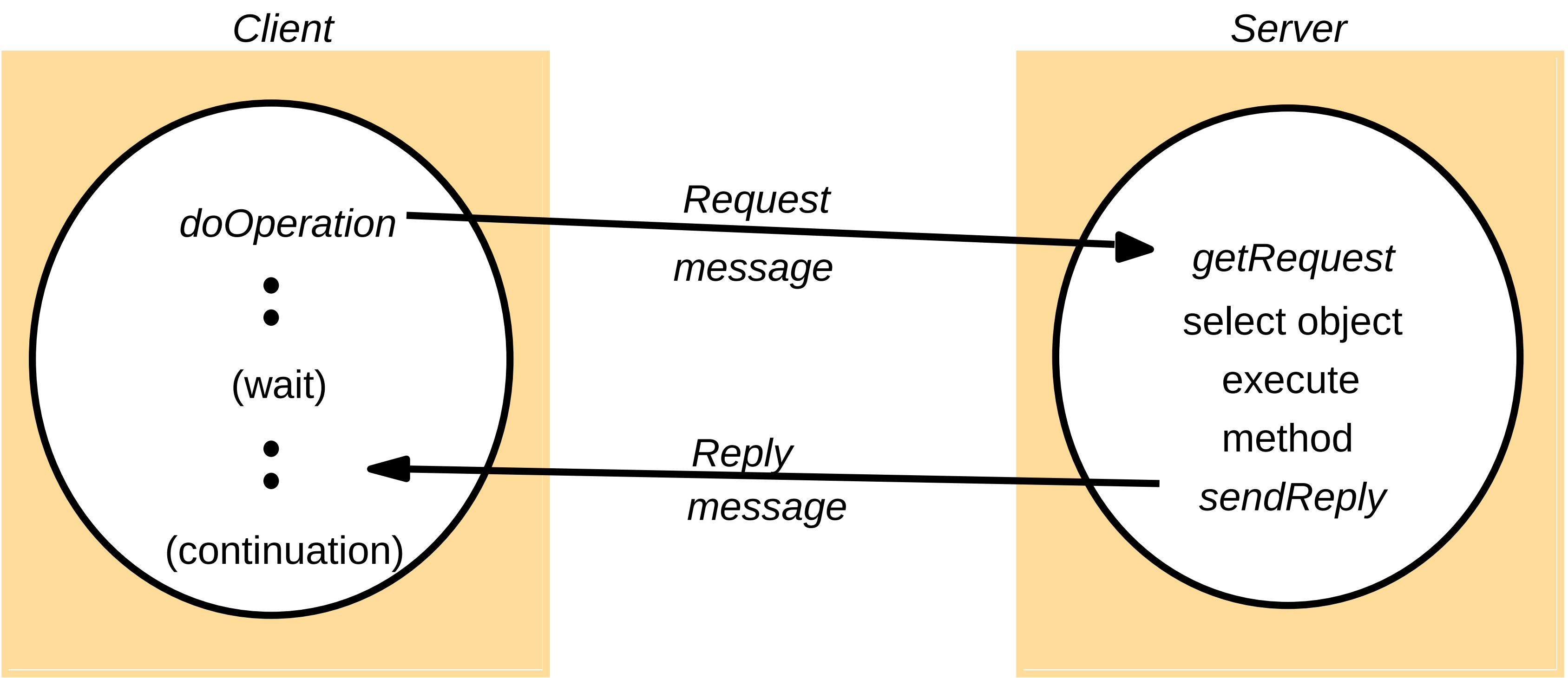
Send operations	*Blocking*	*Non-blocking*
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication 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),

Remote Invocation

Remote Invocation:
Middleware layers



Remote Invocation:
Request-reply communication



Remote Invocation:
Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)

sends a request message to the remote server and returns the reply. The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();

server acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

server sends the reply message reply to the client at its Internet address and port.

Remote Invocation:

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
Request-Id	*int*
remoteReference	*RemoteRef*
Operation-Id	*int or Operation*
arguments	*array of bytes*

Remote Invocation:
RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Remote Invocation:
HTTP request message

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Remote Invocation:
HTTP reply message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Remote Invocation:
CORBA IDL example

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void    addPerson(in Person p) ;
        void    getPerson(in string name, out Person p);
        long    number();
};
```

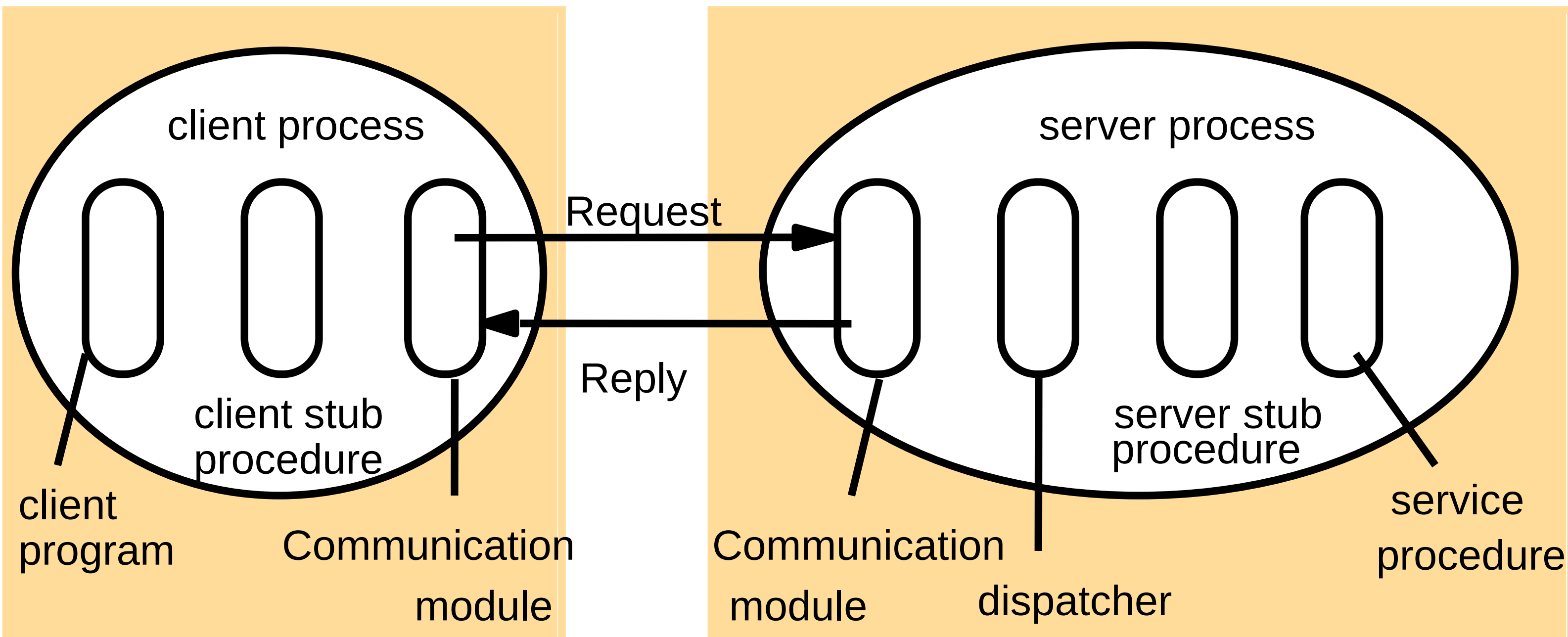
- CORBA / RPC is programming with interfaces; separate interface from implementation
- In local procedure we pass parameters by value or by reference. In RPC, we describe parameters as <In, Out, InOut>

Remote Invocation:
Call semantics

Fault tolerance measures			Call semantics
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

- **Retransmit request**: retransmit till you get reply or conclude that server is down
- **Duplicate Filtering**: whether to filter duplicate requests at the server
- **Retransmission of results**: whether to keep history of result msgs to retransmit lost replies w/o re-executing the request operation

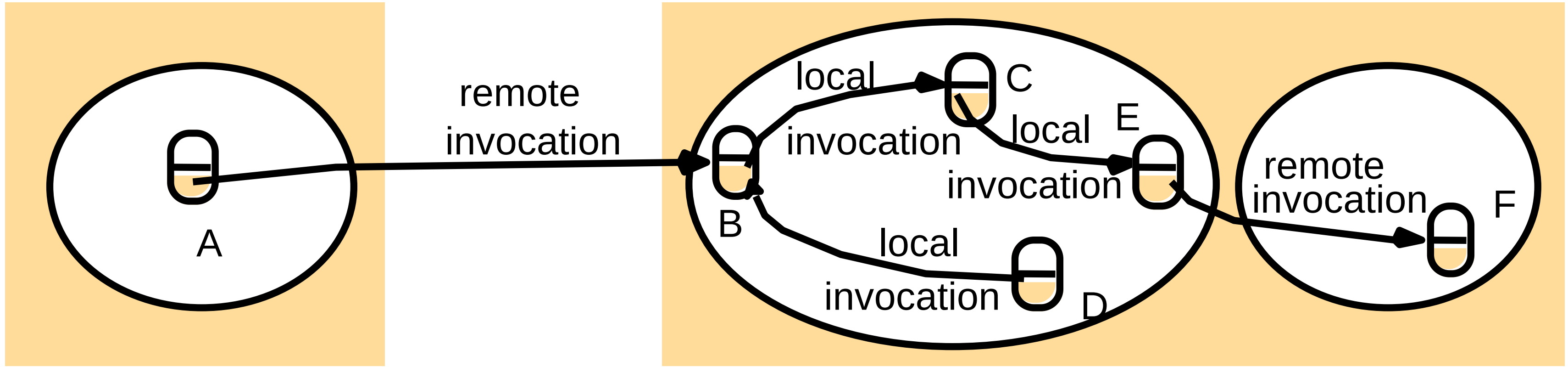
Remote Invocation:
Role of client & server stub procedure in RPC



- Client stub marshalls request and unmarshall response
- Server stub unmarshall request and marshall response

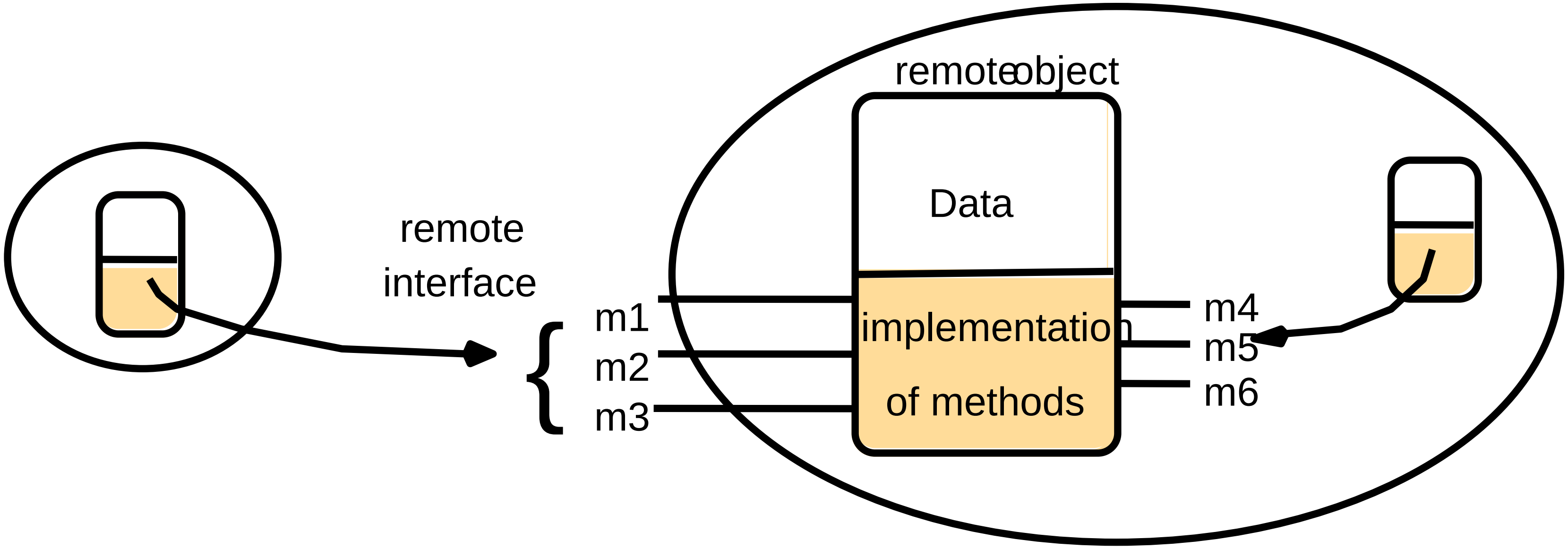
Remote Invocation:

Remote and local method invocations



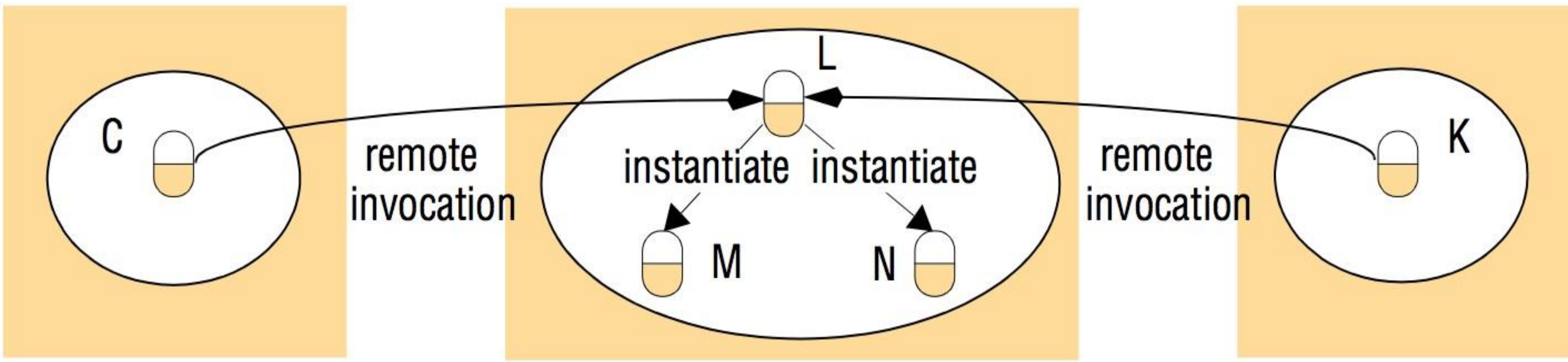
Remote Invocation:

A remote object and its remote interface



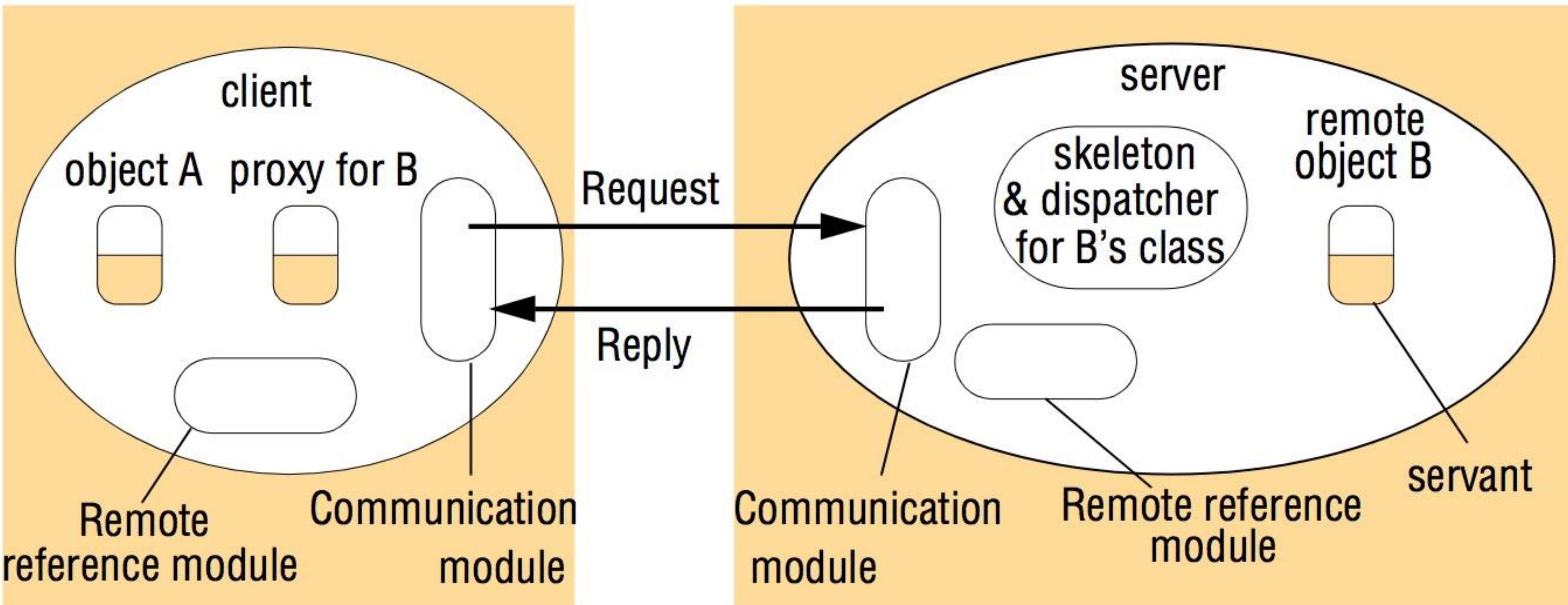
- Objects in other processes can invoke only methods in the remote interface (m1, m2, m3). Local objects can invoke both methods in the remote interface (M1, m2, m3) as well as other methods implemented by the remote object (m4, m5, m6).

Remote Invocation:
Instantiation of remote objects



- Distributed applications may provide remote objects with methods for instantiating objects that can be accessed remotely; thus effectively providing the effect of remote instantiation of objects.

Remote Invocation: The role of proxy and skeleton in remote method invocation



- **Proxy** is the client stub
- **Skeleton** is the server stub

Remote Invocation:
Java remote interfaces Shape and ShapeList

```
import java.rmi.*;
import java.util.Vector;

public interface Shape  extends  Remote {
    int                  getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;             1
}

public interface ShapeList  extends  Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;       2
    Vector allShapes() throws RemoteException;
    int       getVersion() throws RemoteException;
}
```

Remote Invocation:

The Naming class of Java RMIregistry

*void **rebind** (String name, Remote obj)*

This method is used by a server to register the identifier of a remote object by name, as shown in next page, line 2.

*void **bind** (String name, Remote obj)*

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

*void **unbind** (String name, Remote obj)*

This method removes a binding.

*Remote **lookup** (String name)*

This method is used by clients to look up a remote object by name, as shown in Figure 5.20 line 1. A remote object reference is returned.

*String [] **list**()*

This method returns an array of Strings containing the names bound in the registry.

Remote Invocation:

Java class ShapeListServer with main() method

```
import java.rmi.*;

public class ShapeListServer{
    public static void main (String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
            ShapeList aShapeList = new  ShapeListServant();          1
                Naming.rebind("Shape List", aShapeList );            2
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Remote Invocation: Java class ShapeListServant implements interface ShapeList

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
     private Vector theList;                    // contains the list of Shapes
     private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException { 1
             version++;
            Shape s = new ShapeServant( g, version);                    2
            theList.addElement(s);
            return s;
    }
    public  Vector   allShapes()  throws RemoteException{...}
    public int          getVersion() throws RemoteException { ... }
}
```

Remote Invocation:
Java client of ShapeList

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class ShapeListClient{
   public static void main (String args[]){
     System.  setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{
          aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;          1
          Vector sList = aShapeList.allShapes();                                   2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

Indirect Communication

Indirect Communication:

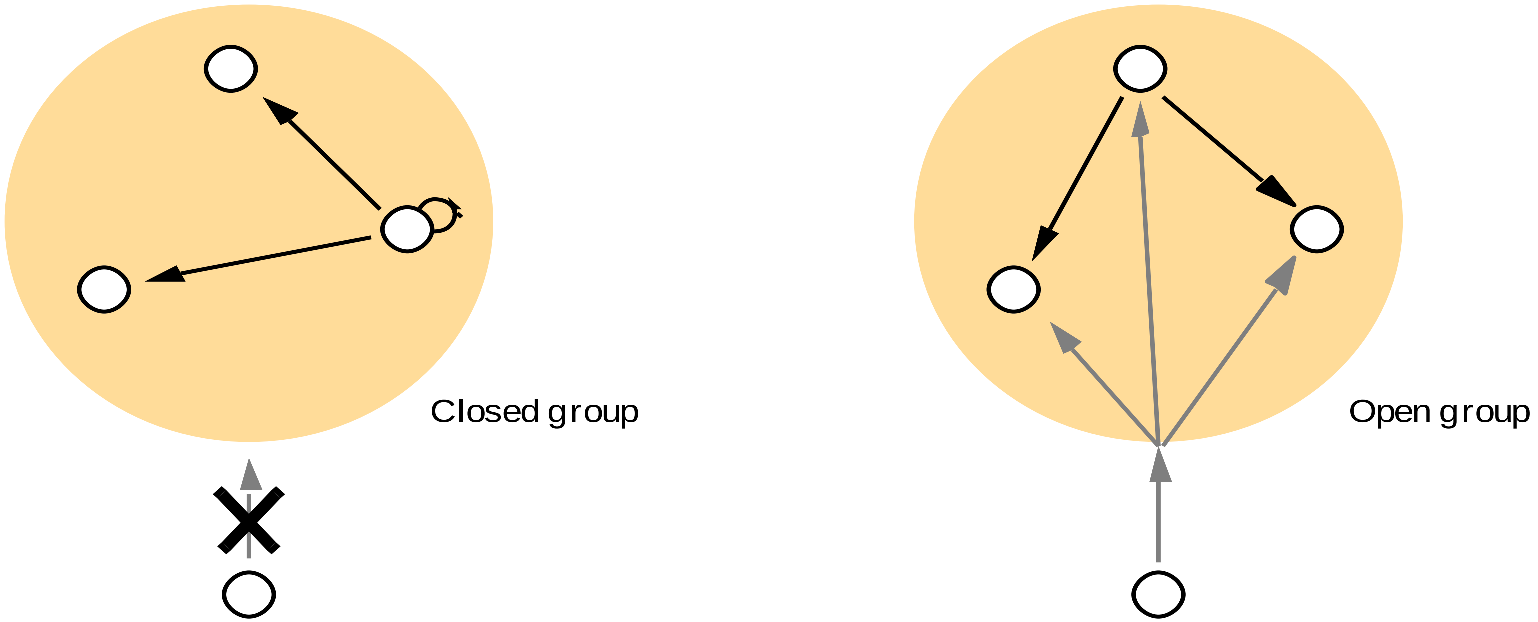
Space and time coupling in distributed systems

- **Indirect Communications** means communication through intermediary,
  hence no direct coupling between sender and receiver.
- **Space uncoupling:** sender does not know or does not need to know the identity of the receiver(s), and vice versa. This allows flexibility of change (senders and receivers) can be replaced, updated, replicated or migrated
- **Time uncoupling:** sender and receiver can have independent lifetimes. In other words, sender and receiver(s) do not need to exist at the same time to communicate
- **Example**: in mobile environment where users may rapidly connect and disconnect from the global network (time uncoupling).
- **Group communication** offers service where message is sent to a group, multicast communication. A process may join or leave the group
- **Reliability & ordering in multicast:** all members should receive any multicast msg and the delivery order

Indirect Communication:
Space and time coupling in distributed systems

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 15.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

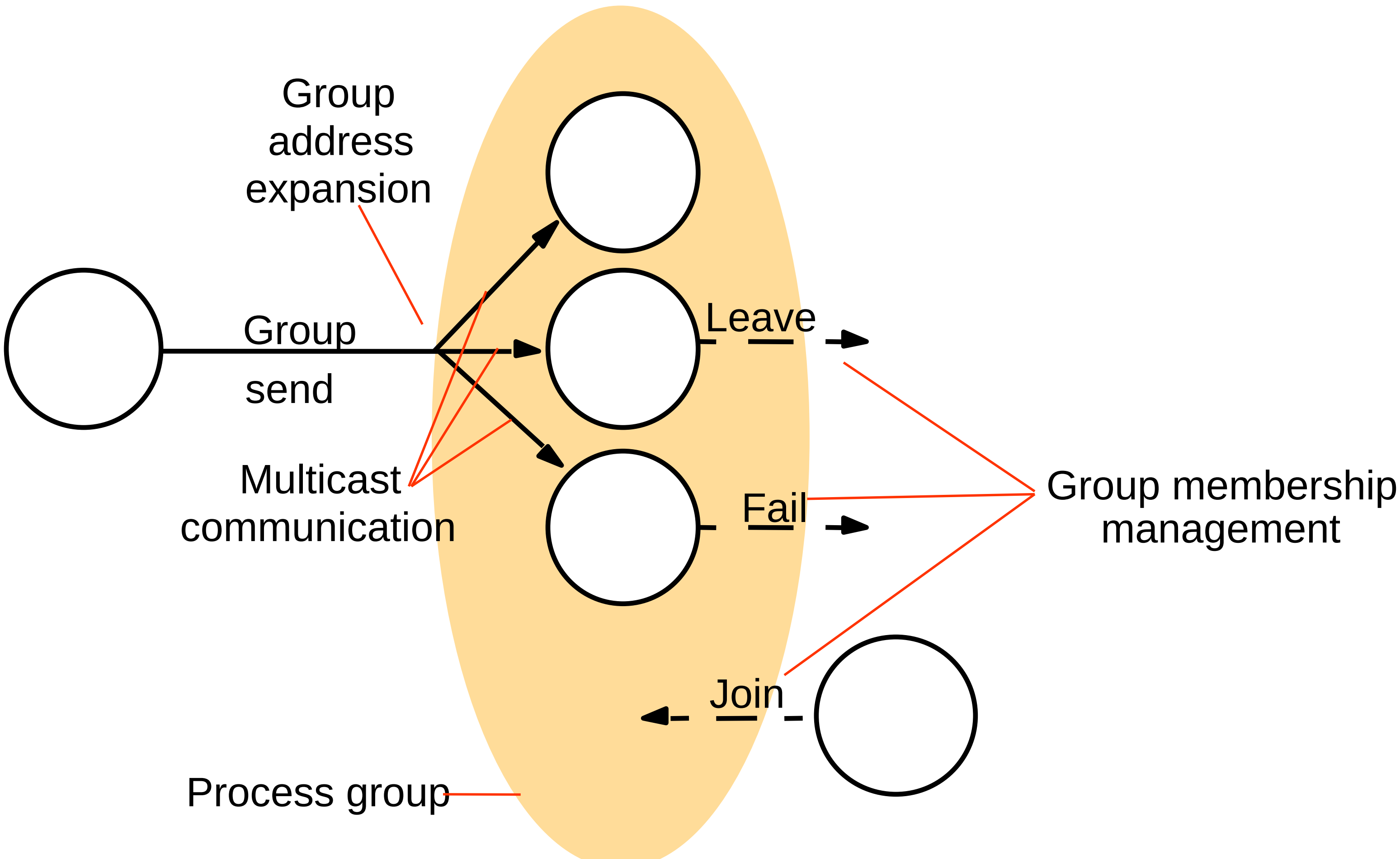
Indirect Communication:
Open and Closed groups



- Communication with all group members is broadcast. Communication with a subset is multicast. Communication with a single member is called unicast. An entity can be member of multiple groups.
- **Closed group:** only group member can send message to the group.
- **Open group:** outside process can communicate with the group.

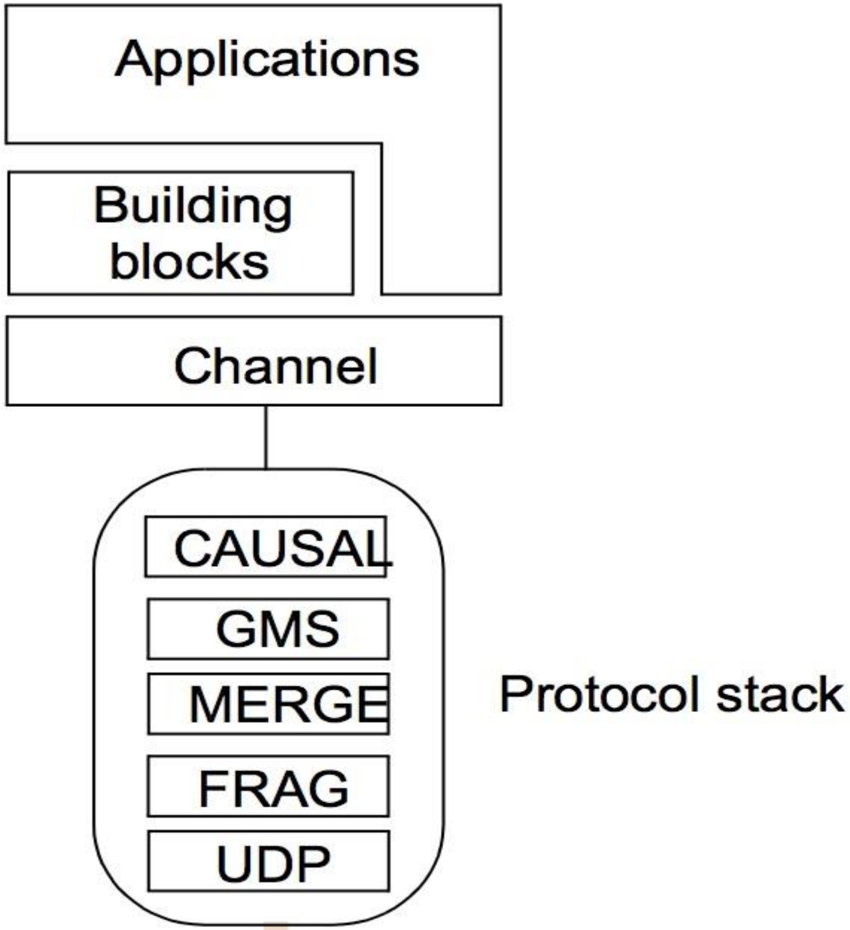
Indirect Communication:

The role of group membership management



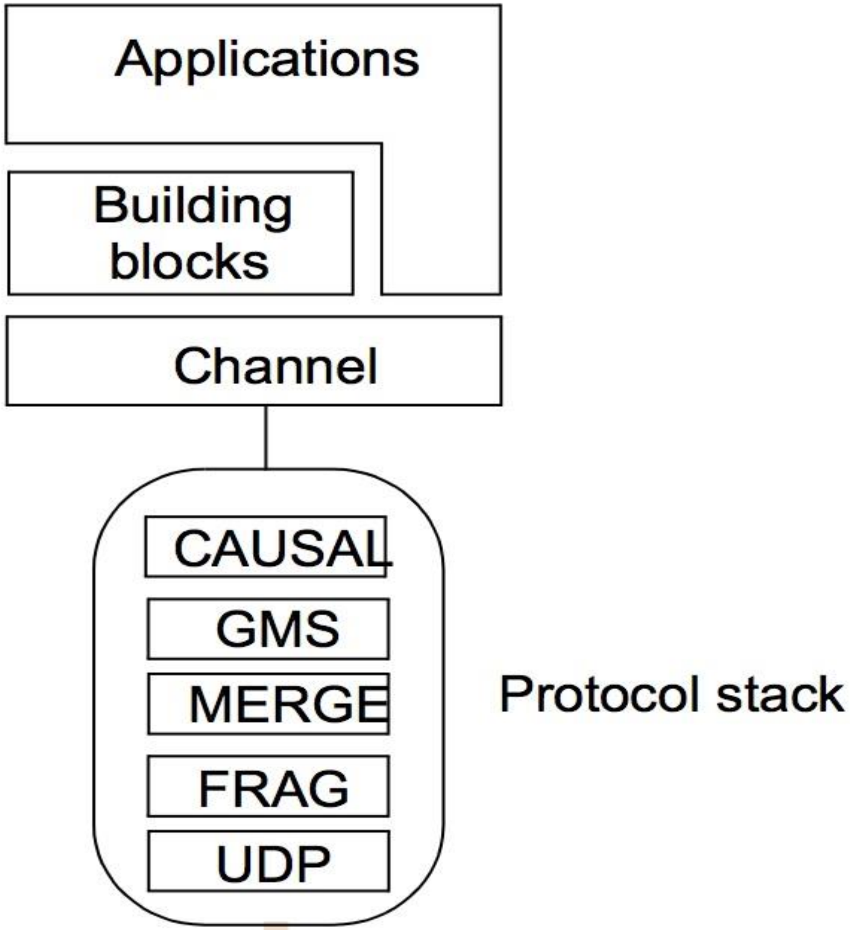
Indirect Communication:
The architecture of JGroups Toolkit www.jgroups.org

- JGroup is an open source toolkit that originated at Cornell University.
- JGroup is written in Java and it supports process groups; process can join and leaves a group + send message to the group or to single member in the group.
- JGroups supports different delivery and order guarantees.
- A process interact with a group through a channel object (handle)
- Process can connect / disconnect to a group



Indirect Communication:
The architecture of JGroups Toolkit

- **UDP:** transport layer
- FRAG: message packetization and set maximum packet size (default = 8K)
- **MERGE:** deals with network partitioning and subsequent merge
- **GMS:** Group membership protocol – maintain consistent view of Group membership across members
- **CAUSAL:** implements casual ordering



Indirect Communication:

Java class FireAlarmJG

```
import org.jgroups.JChannel;
public class FireAlarmJG {
public void raise() {
    try {
        JChannel  channel = new JChannel();
        channel.connect("AlarmChannel");
        Message msg = new Message(null, null, "Fire!");
        channel.send(msg);        // multicast to registered receivers
    }
    catch(Exception e) {
    }
}
```

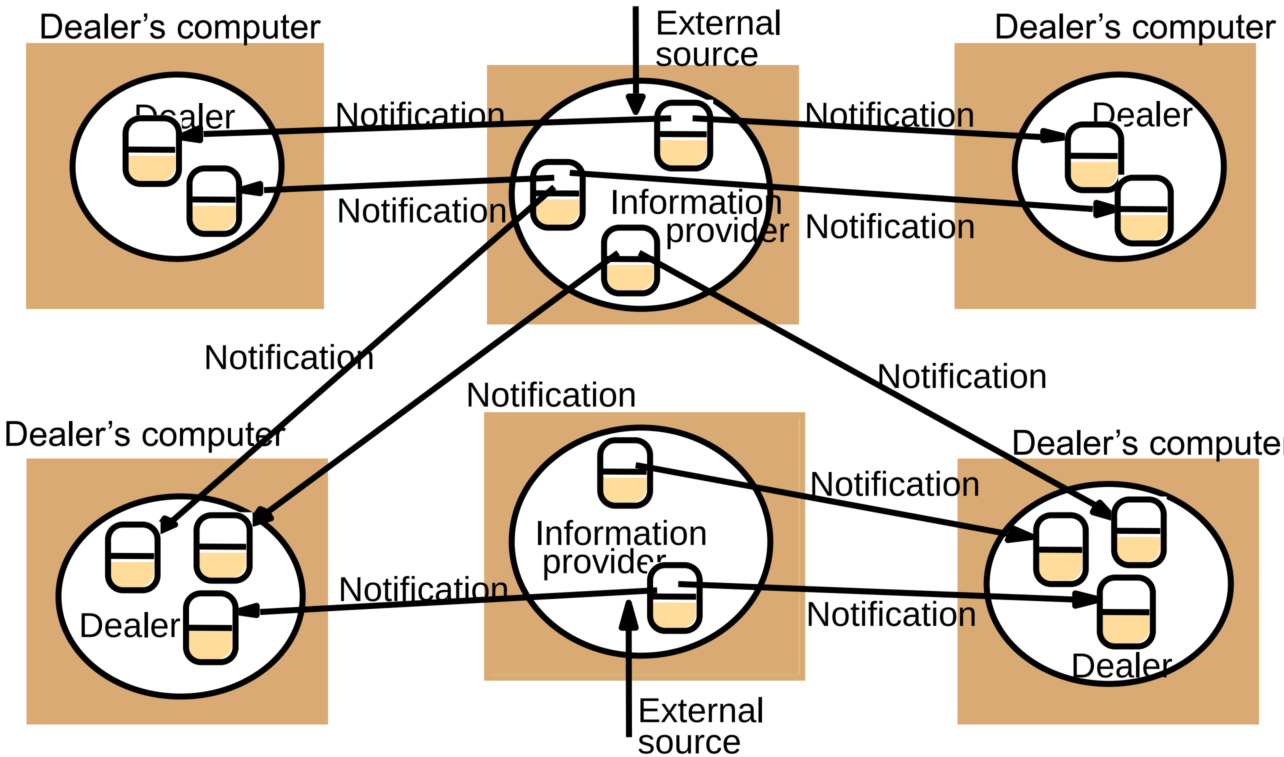
Indirect Communication:
Java class FireAlarmConsumerJG

```
import org.jgroups.JChannel;

  public  class  FireAlarmConsumerJG {
      public String await() {
     try {
           JChannel channel = new JChannel();
           channel.connect("AlarmChannel");
           Message msg = (Message) channel.receive(0);
           return (String) msg.GetObject();
     } catch(Exception e) {
           return null;
     }
  }
}
```

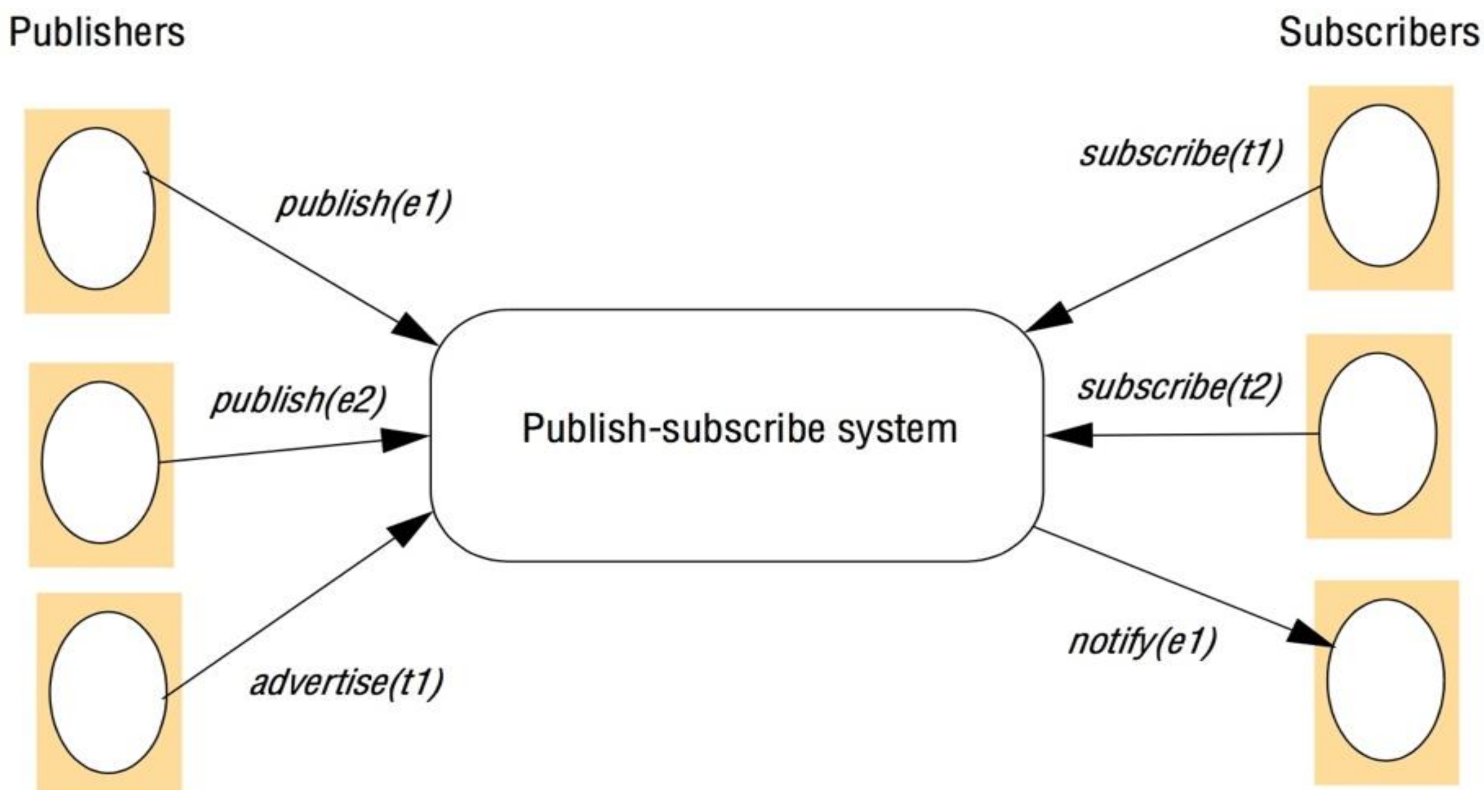
Indirect Communication:

Dealing room system – publish-subscribe system



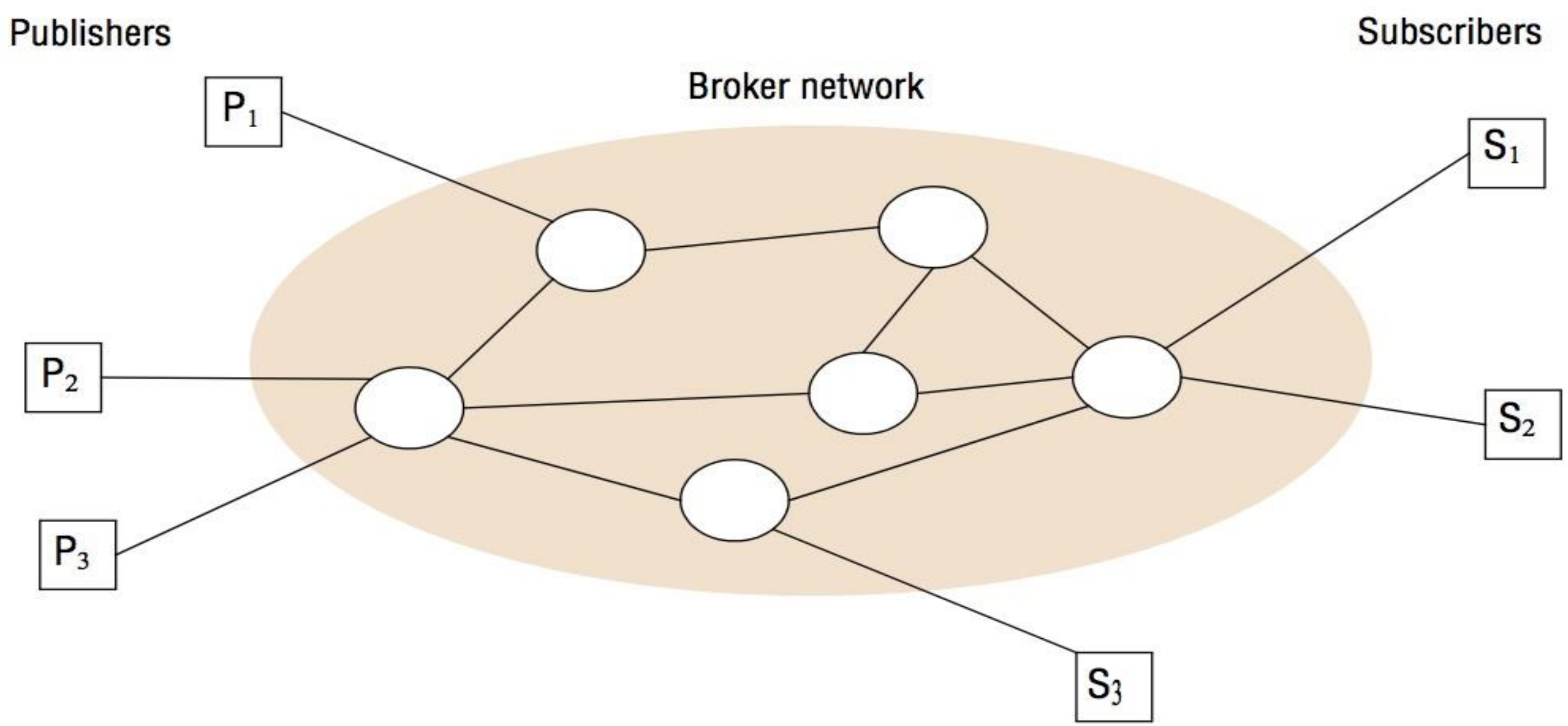
- **Dealing room system:** whose task is to allow dealers using computers to see latest information about market prices of relevant stocks. Each stock is represented as an object. Information arrives from different associated objects and arrives in the form of an update to some of the objects

Indirect Communication:
The publish-subscribe paradigm



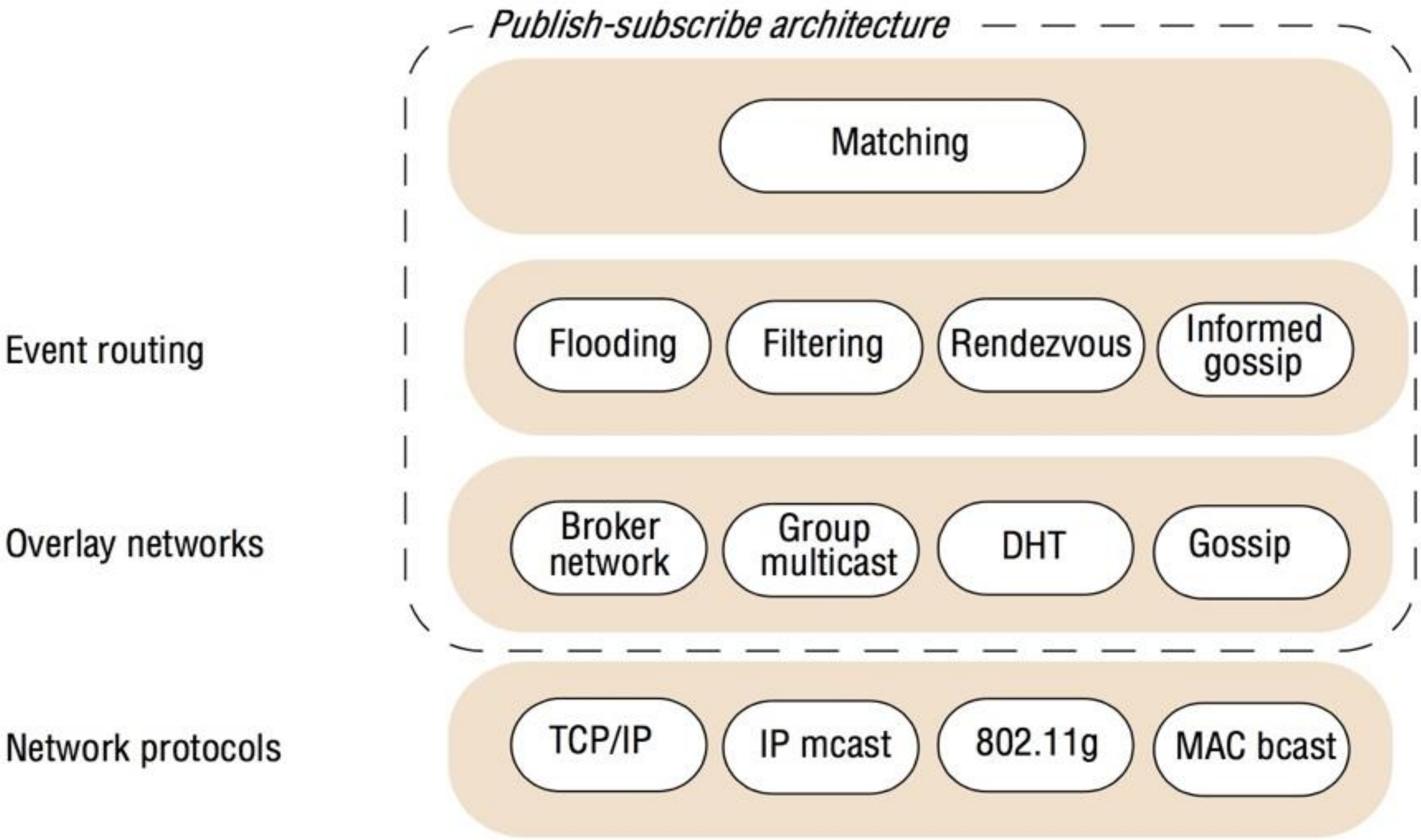
- **Publish-subscribe** is also referred to as Distributed Event-based systems. It is also one-to-many communication paradigm
- **Publish-subscribe** is a key component of Google's infrastructure – dissemination of events related to advertisements (ad clicks)

Indirect Communication:
A network of brokers



- **Publish-subscribe** can be implemented using centralized broker architecture (not scalable) or distributed where the centralized broker is replaced with network of brokers or we can go further using P2P strategy.

Indirect Communication:
The architecture of publish-subscribe systems



Indirect Communication:
Event routing layer

- **Flooding:** simplest approach; simply send event notification to all nodes in the network. Each node needs to do matching at the subscriber end
- **Filtering:** apply filtering at the network of brokers. Broker will forward notifications through the network only if path exists to a valid subscriber
- **Rendezvous:** this approach is meant to achieve load balance; partition the event space among the set of brokers in the network
- **Informed gossip:** leverage P2P to underpin event routing in publish-subscribe systems. Gossip-based approaches are used to implement efficiently multicast. They are executed by nodes periodically and probabilistically exchange events with neighboring nodes.

Indirect Communication:
Filtering-based routing

upon receive *publish(event e)* ***from*** *node x*
    matchlist := match(e, subscriptions)
    send notify(e) to matchlist;
    fwdlist := match(e, routing);
    send *publish(e) to fwdlist - x;*
upon receive *subscribe(subscription s)* ***from*** *node x*
    if *x is client* ***then***
        add x to subscriptions;
    else *add(x, s) to routing;*
    send *subscribe(s) to neighbours - x;*

Indirect Communication:
Rendezvous-based routing

upon receive *publish(event e)* ***from*** *node x* ***at*** *node i*

rvlist := EN(e);

if *i in rvlist* ***then begin***

matchlist := ***match****(e, subscriptions);*

send notify(e) to matchlist;

end

send *publish(e) to rvlist - i;*

upon receive subscribe*(subscription s)* ***from*** *node x* ***at*** *node i*

rvlist := SN(s);

if *i in rvlist* ***then***

add s to subscriptions;

else

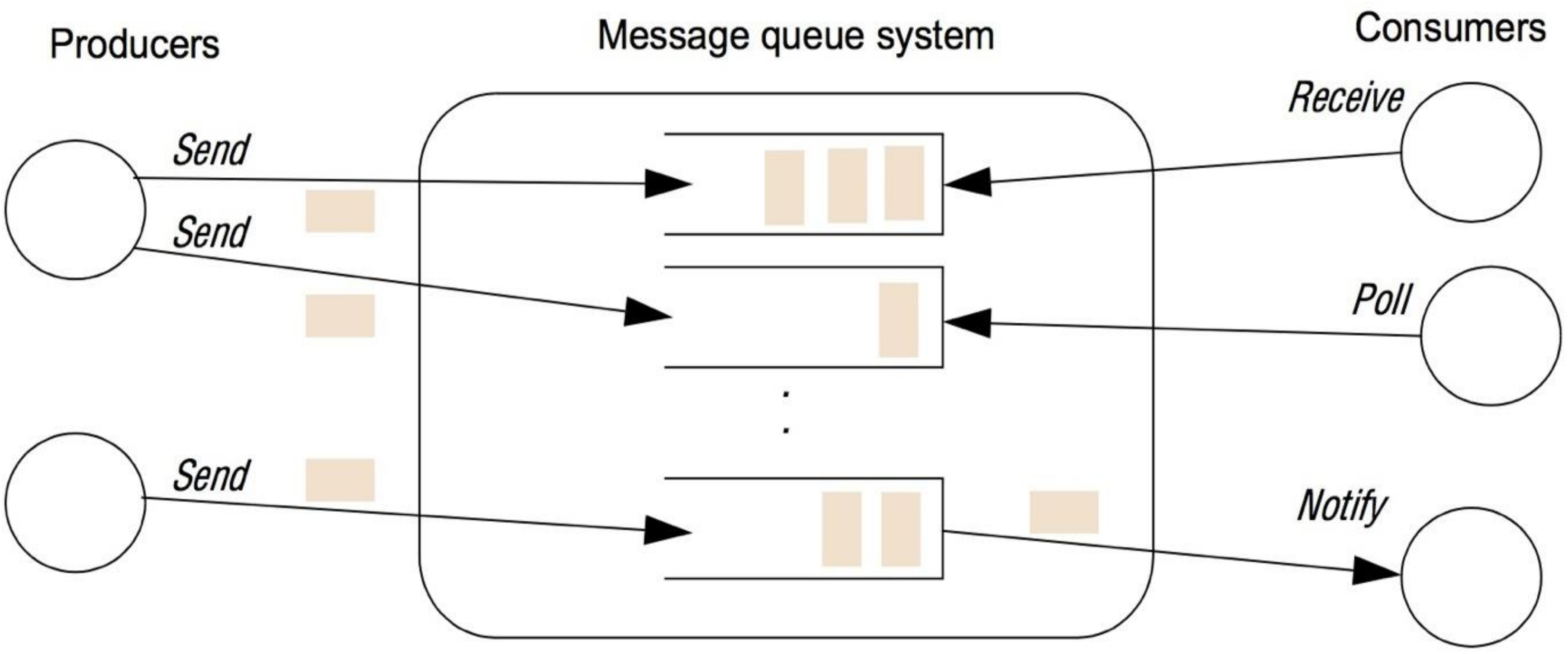
send *subscribe(s) to rvlist - i;*

Indirect Communication:

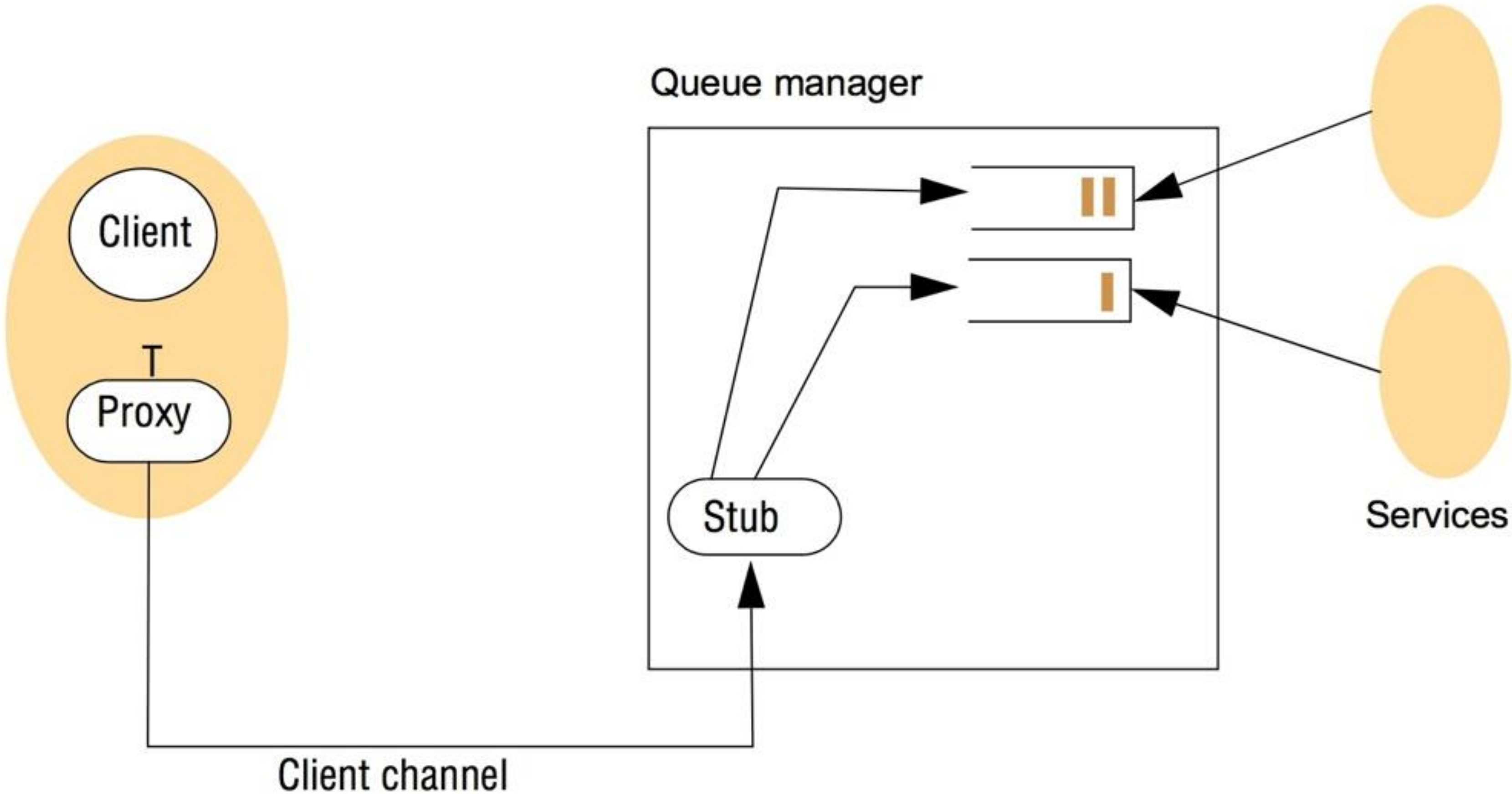
Example: publish-subscribe systems

System (and further reading)	*Subscription model*	*Distribution model*	*Event routing*
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvouz [Oki *et al.* 1993]	Topic-based	Distributed	Ffiltering
Scribe [Castro *et al.* 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni *et al.* 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga *et al.* 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	Flooding
Meghdoot [Gupta *et al.* 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni *et al.* 2005]	Content-based	Peer-to-peer	Informed gossip

Indirect Communication:
The message queue paradigm



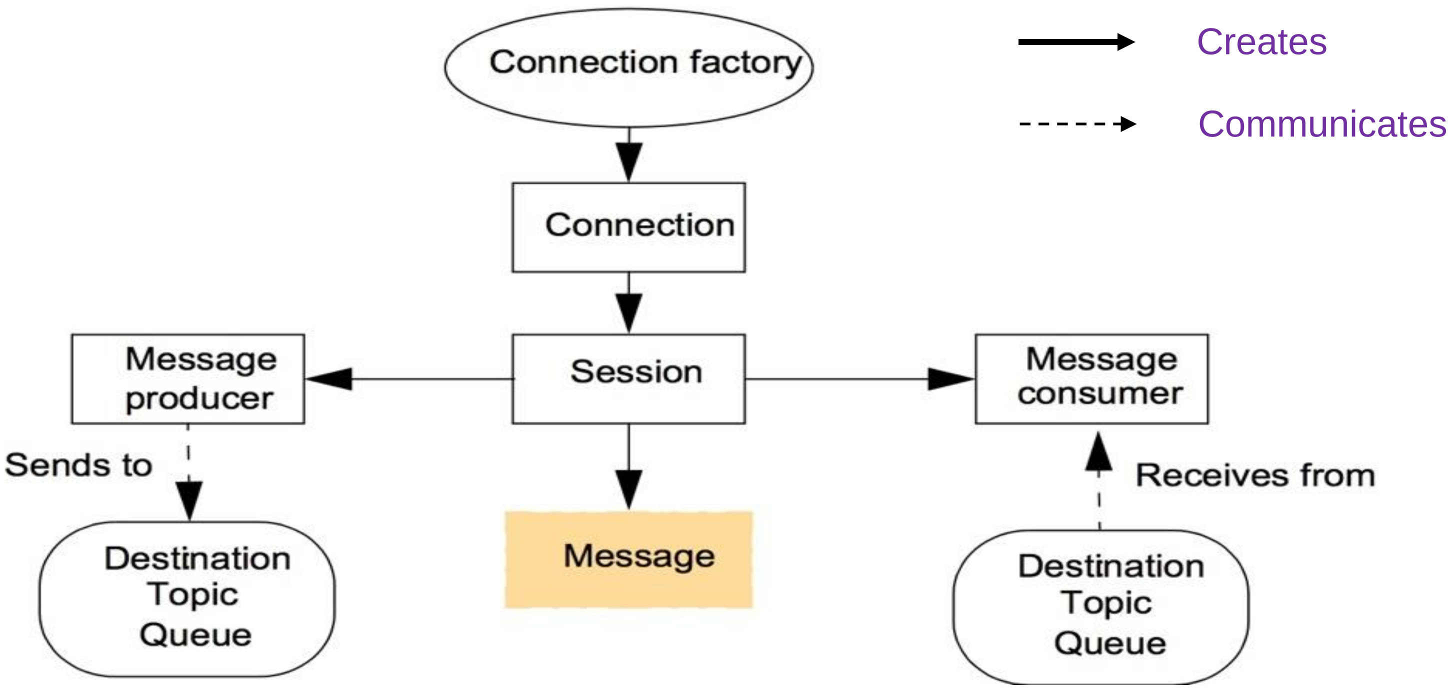
Indirect Communication:
A simple networked topology in WebSphere MQ



Indirect Communication:

The programming model offered by JMS

package: javax.jms **API:** https://docs.oracle.com/javaee/7/api/javax/jms/package-summary.html



- **JMS** unify the publish-subscribe and message queue paradigms – Indirect communication
- **JMS** supports the notion of Topics and queues as alternative destination for a message
- Variation of implementations based on JMS specs: JBoss, Apache ActiveMQ, Open MQ, etc.

Indirect Communication:
Java class FireAlarmJMS

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {

public void raise() {
      try {
            Context ctx = new InitialContext();
            TopicConnectionFactory topicFactory =
                  (TopicConnectionFactory)ctx.lookup ("TopicConnectionFactory");
            Topic   topic = (Topic)ctx.lookup("Alarms");
            TopicConnection topicConn =
                  topicConnectionFactory.createTopicConnection();
            TopicSession topicSess = topicConn.createTopicSession(false,
                  Session.AUTO_ACKNOWLEDGE);
            TopicPublisher topicPub = topicSess.createPublisher(topic);
            TextMessage msg = topicSess.createTextMessage();
             msg.setText("Fire!");
            topicPub.publish(message);
            } catch (Exception e) {
            }
}
```

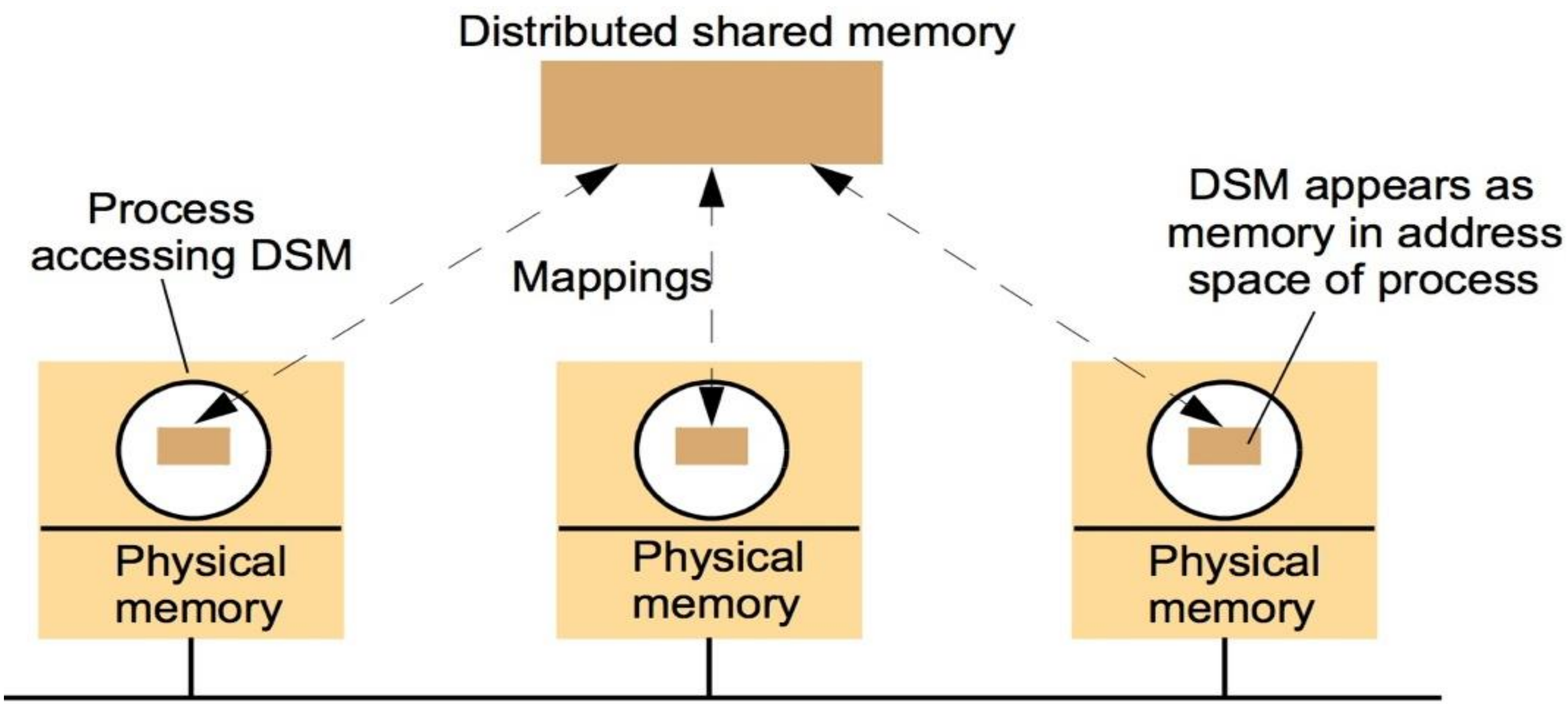
Indirect Communication:

Java class FireAlaramConsumerJMS

```
Import    javax.jms.*;       import javax.naming.*;
public  class  FireAlarmConsumerJMS
public String await() {
       try {
              Context ctx = new InitialContext();
              TopicConnectionFactory   topicFactory =
                     (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
               Topic topic = (Topic)ctx.lookup("Alarms");
               TopicConnection topicConn =
                      topicConnectionFactory.createTopicConnection();
               TopicSession topicSess = topicConn.createTopicSession(false,
                           Session.AUTO_ACKNOWLEDGE);
               TopicSubscriber topicSub = topicSess.createSubscriber(topic);
               topicSub.start();
               TextMessage msg = (TextMessage) topicSub.receive();
               return msg.getText();
               } catch (Exception e) {
                            return  null;
   }
}
```

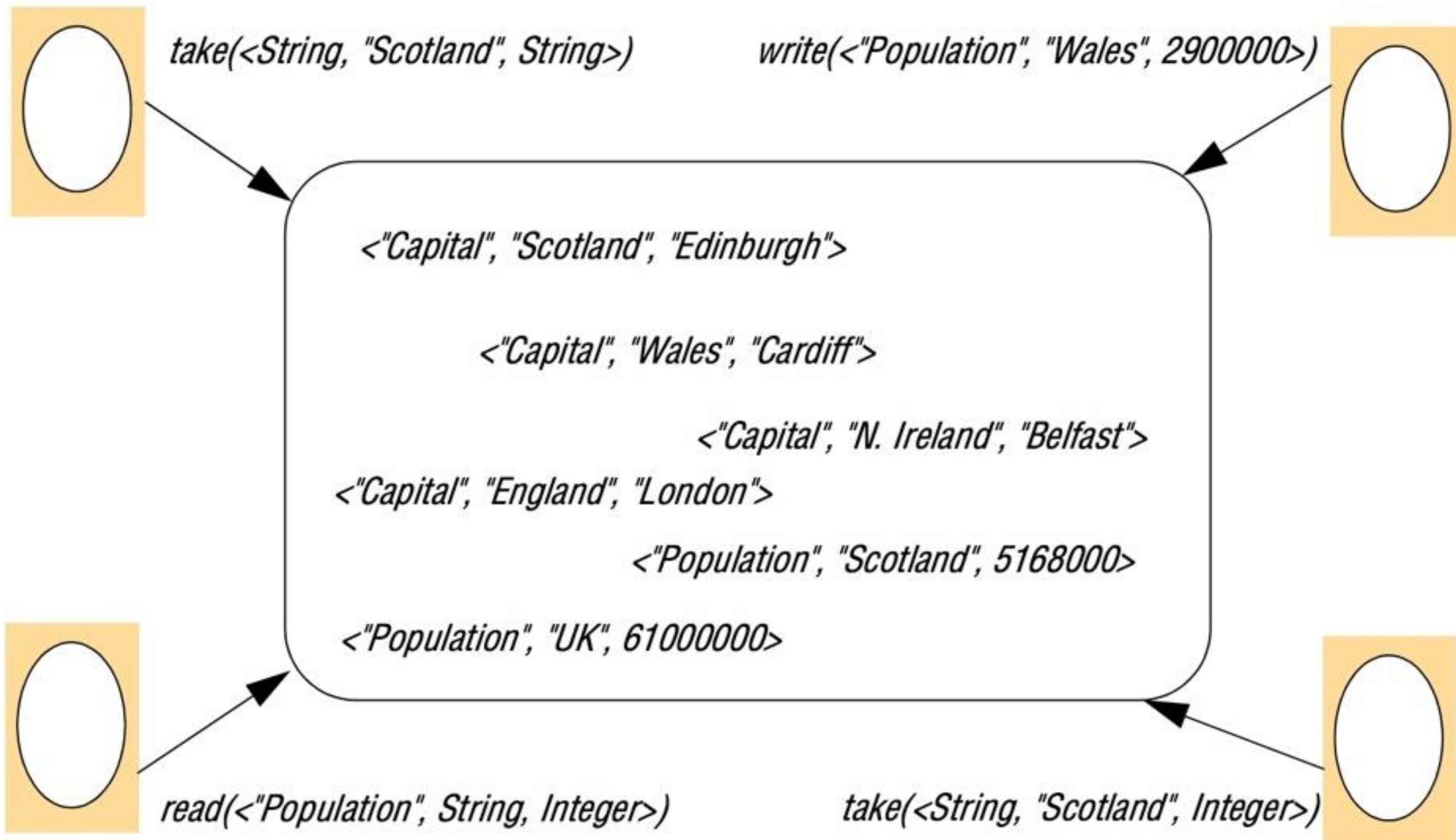
Indirect Communication:

The distributed shared memory (DSM) abstraction



- **DSM** are less appropriate for Client-Server environment as client view server held resources as an abstraction and access them through request.

Indirect Communication:
The tuple space abstraction



Indirect Communication: Replication and the tuple space operations [Xu and Liskov 1989]

Tuple space is a form of Distributed System. Processes communicate by placing tuples in a tuple space (tuple has no address to be used by other processes). Tuples are accessed using pattern matching paradigm (content addressable memory.

Write (no impact on the tuple space)

1. The requesting site multicasts the *write* request to all members of the view;
2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

Read (no impact on the tuple space)

1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others);
4. Step 1 is repeated until at least one response is received.

continued on next slide

Indirect Communication: Replication and the tuple space operations [Xu and Liskov 1989]

Take (read and remove from the tuple space): op blocks till end of phase-1

Phase 1*: Selecting the tuple to be read and removed*

1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: *Removing the selected tuple*

1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

Indirect Communication: Replication and the tuple space operations [Xu and Liskov 1989]

Enhancement:

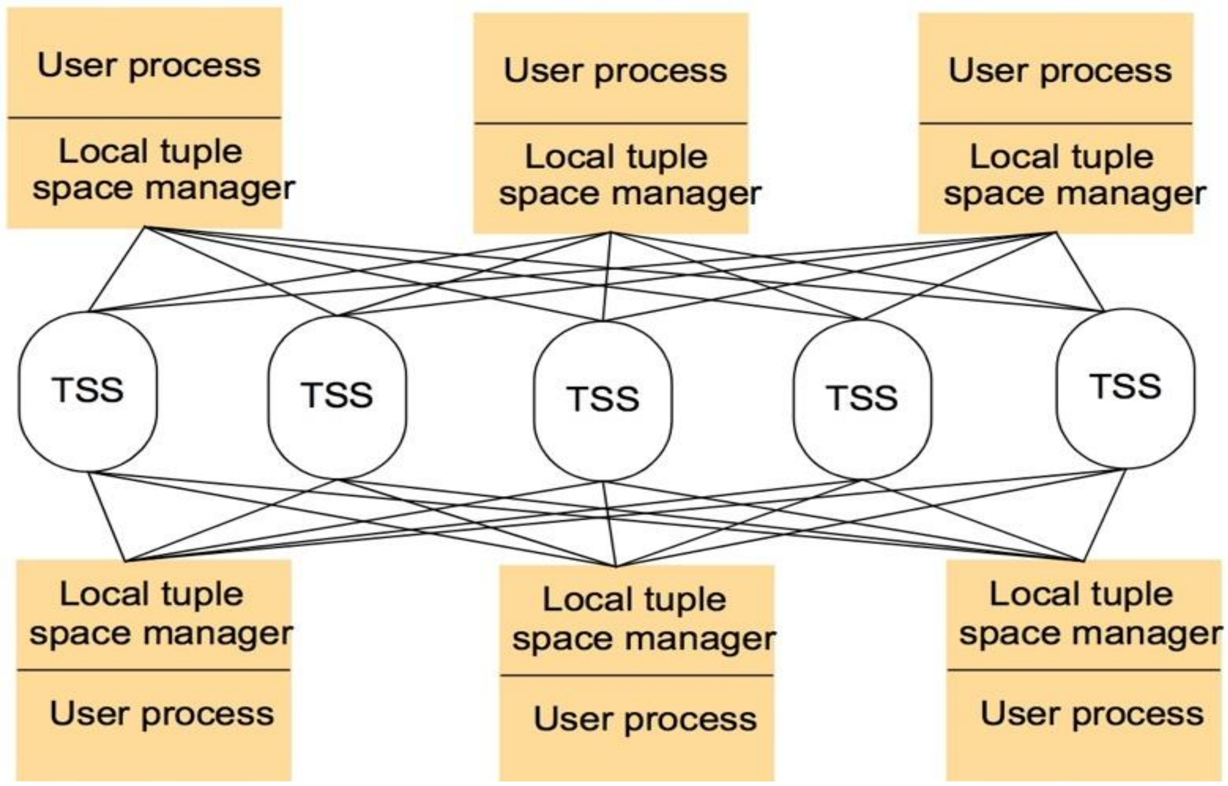
- **Tuple space** suffers from concurrency control / synchronization issues.

Authors added the following constraints as a follow up:

1. Worker Ops must be executed at each replica in the same order.
2. Write Op must not execute at any replica until all previous “Take” Ops issues by the same worker have completed at all replicas in the worker’s view

Indirect Communication:

Partitioning in the York Linda kernel [U. of York, 96]



- **TSS: Tuple Space Servers**
- Tuples are partitioned across a range of available tuple space servers (TSS); no replication of tuples. This is to increase the performance of the tuple space
- Read/Take is more complex
- Some implementations adopted P2P to provide the TSS (attractive approach)

Indirect Communication:
The JavaSpaces API

Operation	*Effect*
Lease write(Entry e, Transaction txn, long lease)	Places an entry into a particular JavaSpace
Entry read(Entry tmpl, Transaction txn, long timeout)	Returns a copy of an entry matching a specified template
Entry readIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
Entry take(Entry tmpl, Transaction txn, long timeout)	Retrieves (and removes) an entry matching a specified template
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)	Notifies a process if a tuple matching a specified template is written to a JavaSpace

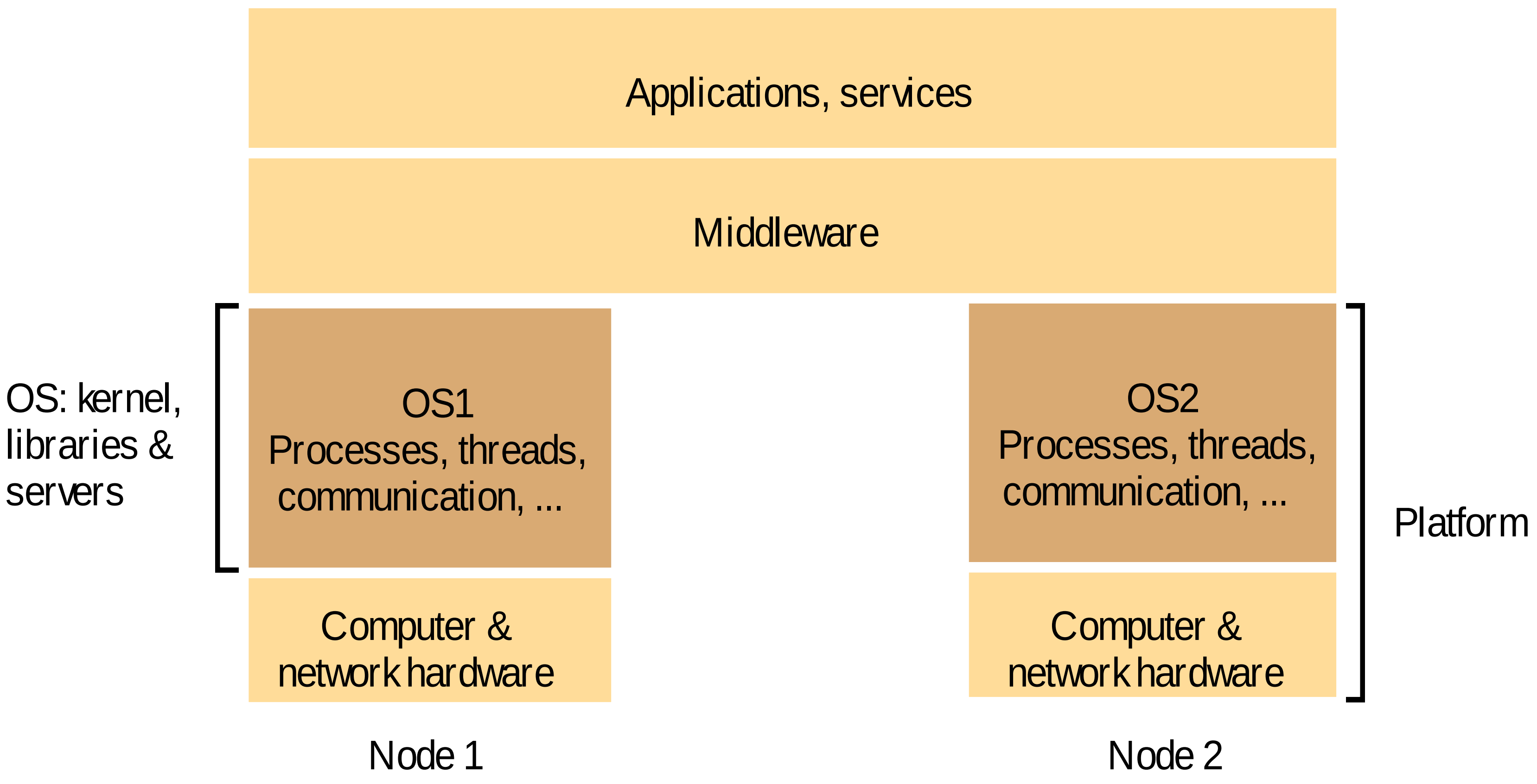
- **JavaSpaces** is a tool for tuple space communication; Sun developed the specs and let 3rd parties fo the implementation (Examples are GigaSpaces, Blitz).
- Goal is to provide platform to simplify developing distributed applications & Services

Indirect Communication:
Summary of indirect communication styles

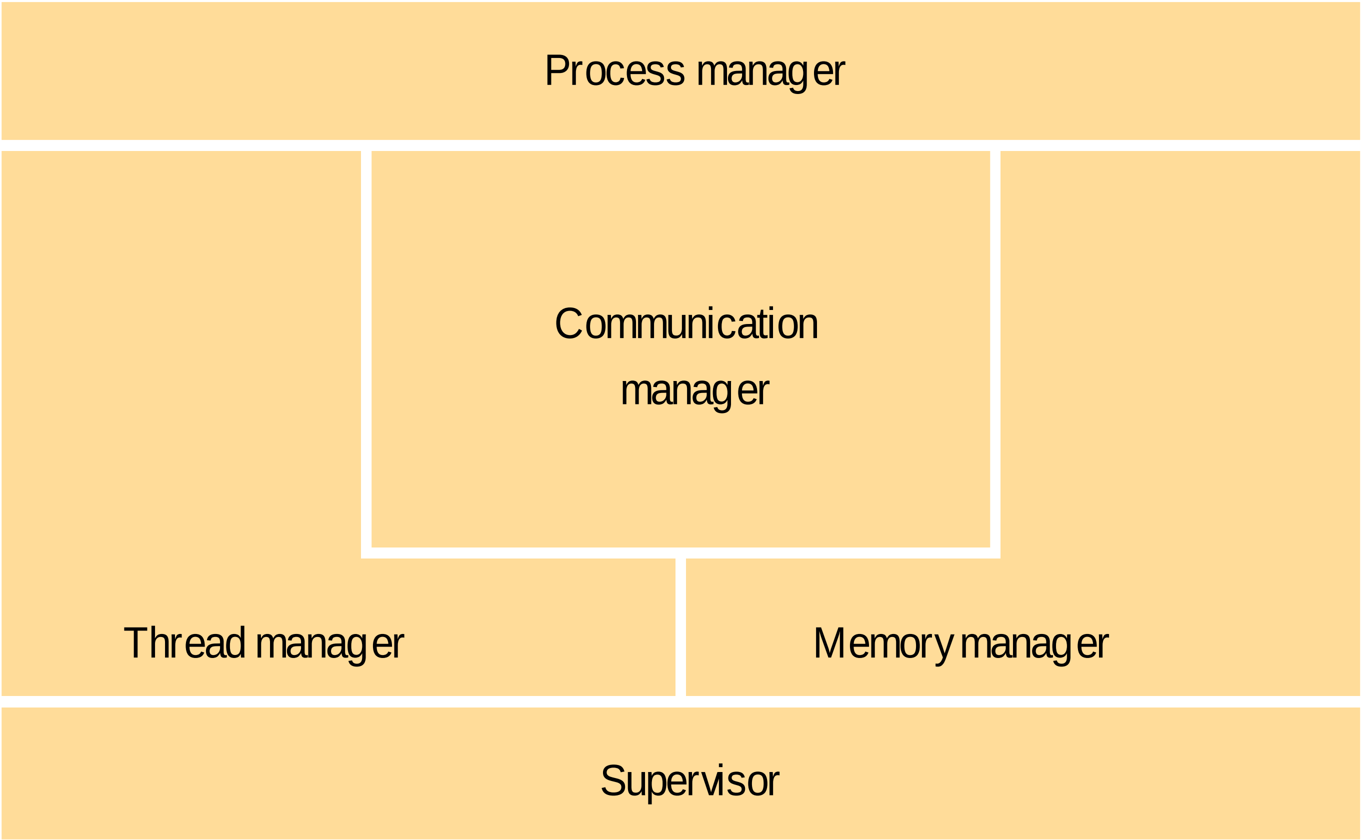
	Groups	*Publish-subscribe systems*	*Message queues*	*DSM*	*Tuple spaces*
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
Main intent	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based publish-subscribe only	No	No	Yes

Operating System Support

Operating System Support:
System layers

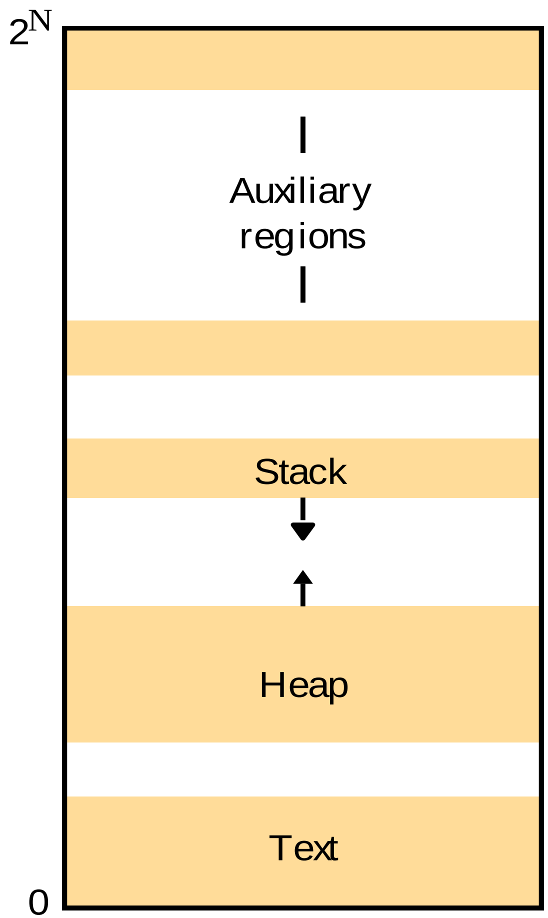


Operating System Support:
Core OS functionality

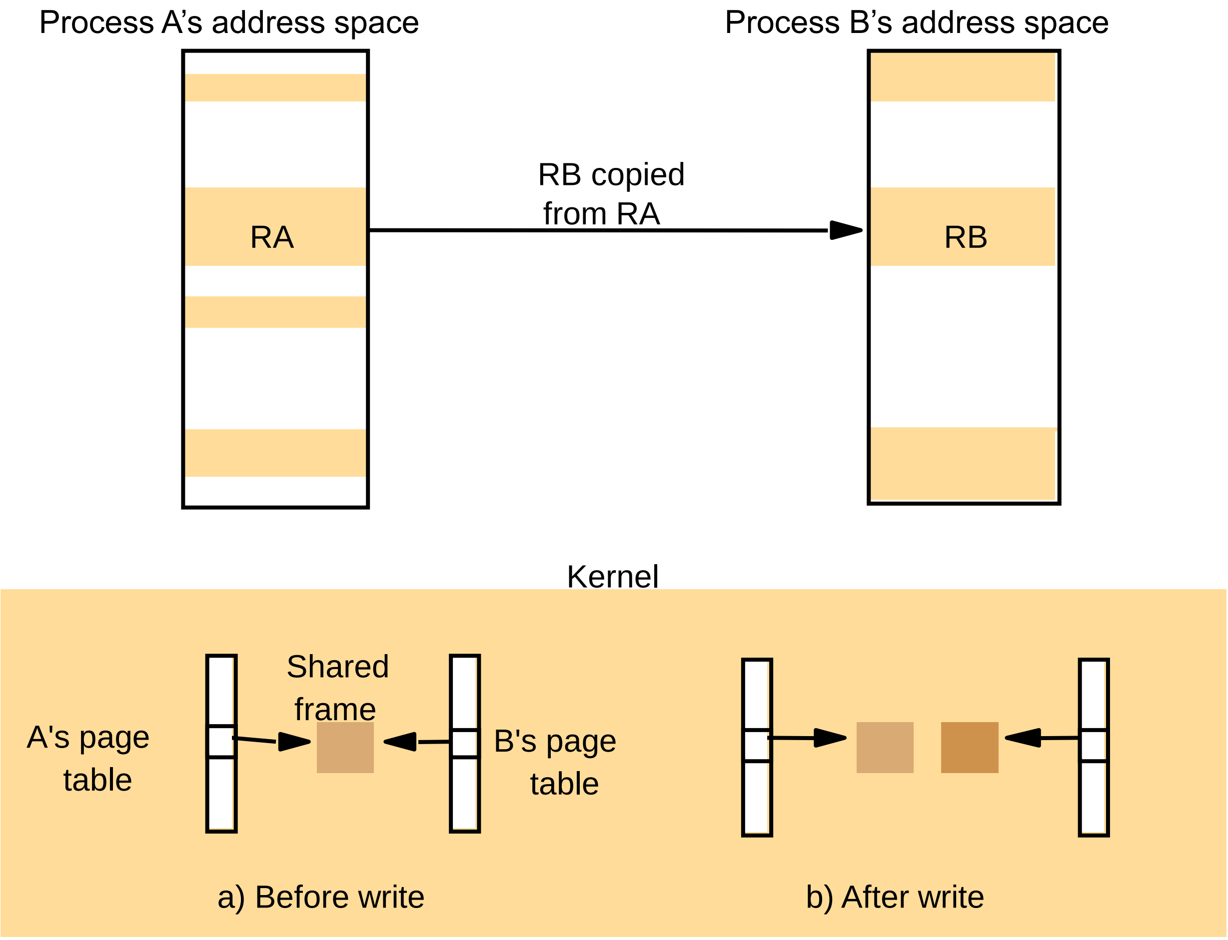


Operating System Support:
Address space

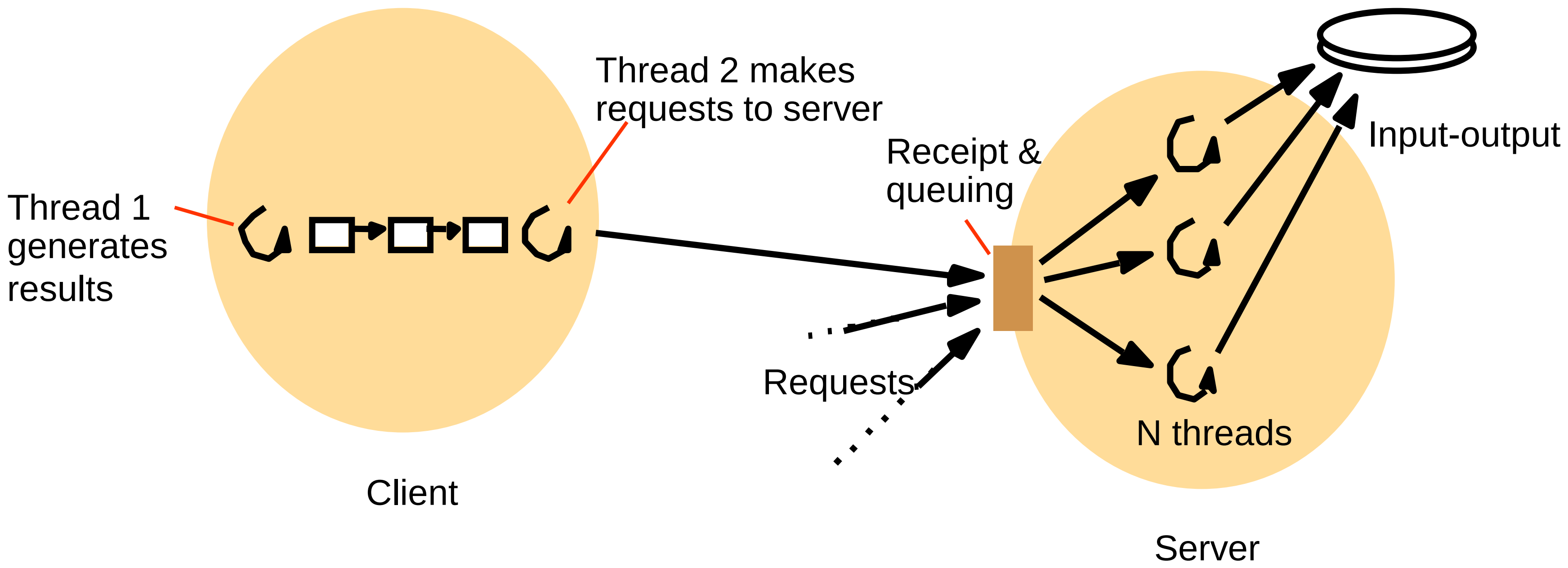
- The address space on the RHS meant to give you an idea and not necessarily an accurate representation for example in the Unix/Linux environment



Operating System Support:
Copy-on-write

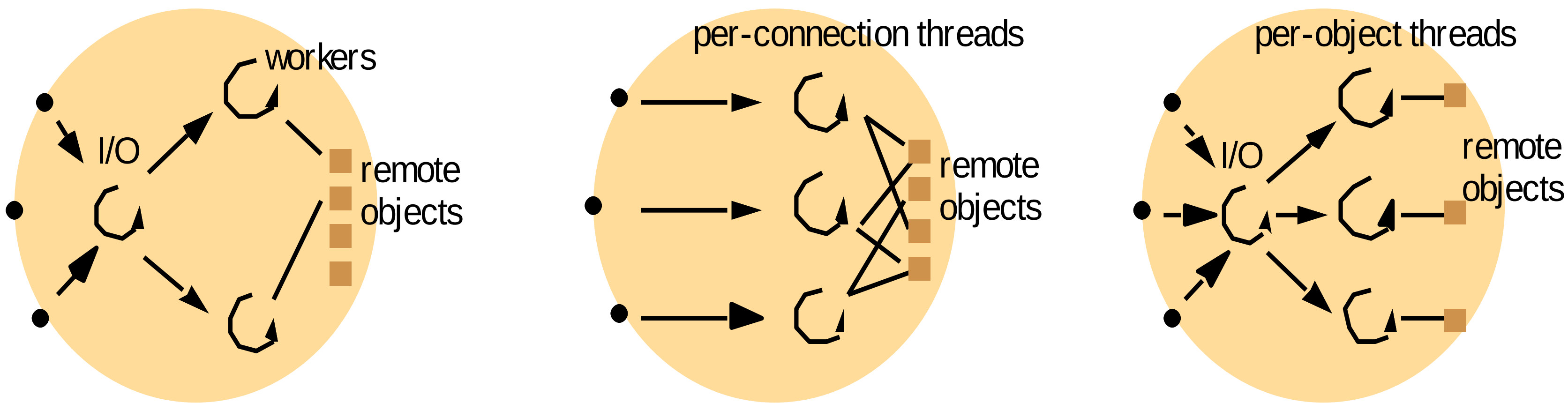


Operating System Support:
Client and server with threads



Server has multiple threads. A thread is assigned a request to serve.

Operating System Support: Alternative server threading architectures (see also pg 82)



a. Thread-per-request

b. Thread-per-connection

c. Thread-per-object

(Dispatcher thread creates a worker thread to serve the request then exits)

(client creates a connection which result in creating server thread to serve client requests. This thread is destroyed when the client closes the connection

(A thread is associated with serving one object. An IO thread receives requests and queue them for object threads.)

Operating System Support:

Java thread constructor and Management methods

Thread*(ThreadGroup group, Runnable target, String name)*

Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority*(int newPriority), getPriority()*

Set and return the thread's priority.

run*()*

A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start*()*

Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep*(int millisecs)*

Cause the thread to enter the *SUSPENDED* state for the specified time.

yield*()*

Causes the thread to enter the *READY* state and invoke the scheduler.

destroy*()*

Destroy the thread.

Operating System Support:
Java thread synchronization calls

*thread.**join**(int millisecs)*

Blocks the calling thread for up to the specified time until *thread* has terminated.

*thread.**interrupt**()*

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

*object.**wait**(long millisecs, int nanosecs)*

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

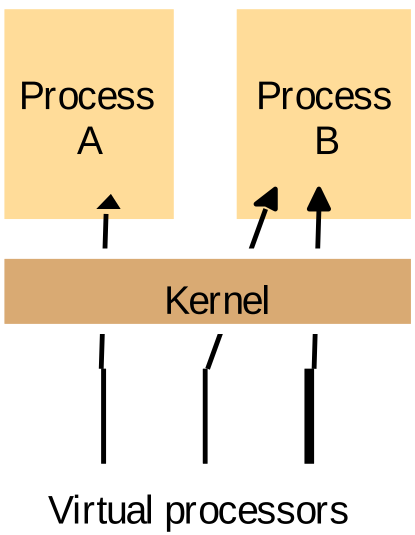
*object.**notify**(), object.notifyAll()*

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

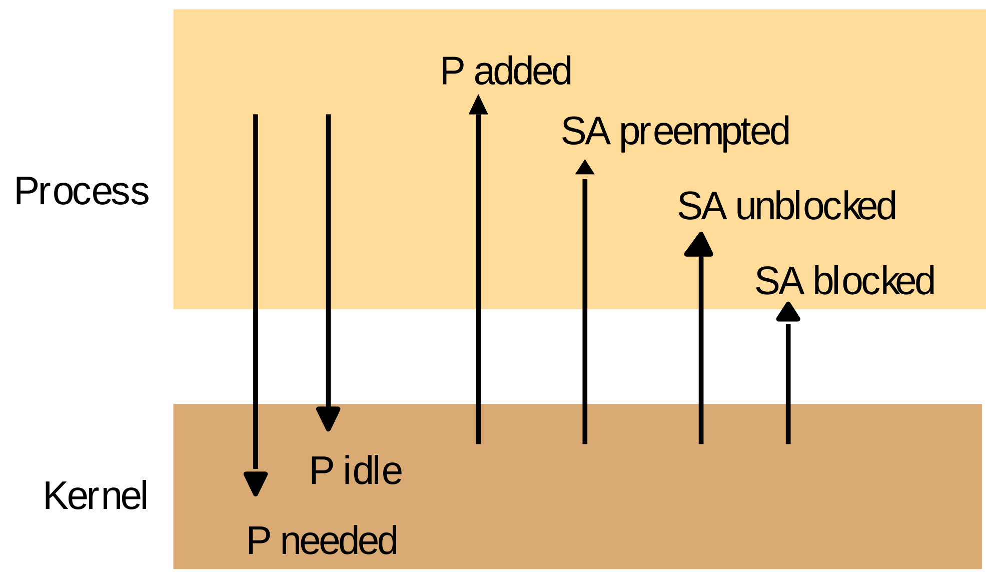
Thread Scheduling: preemptive vs. non-preemptive scheduling

Operating System Support:

Scheduler activations – kernel notifies user process' scheduler with an event



A. Assignment of virtual processors
to processes

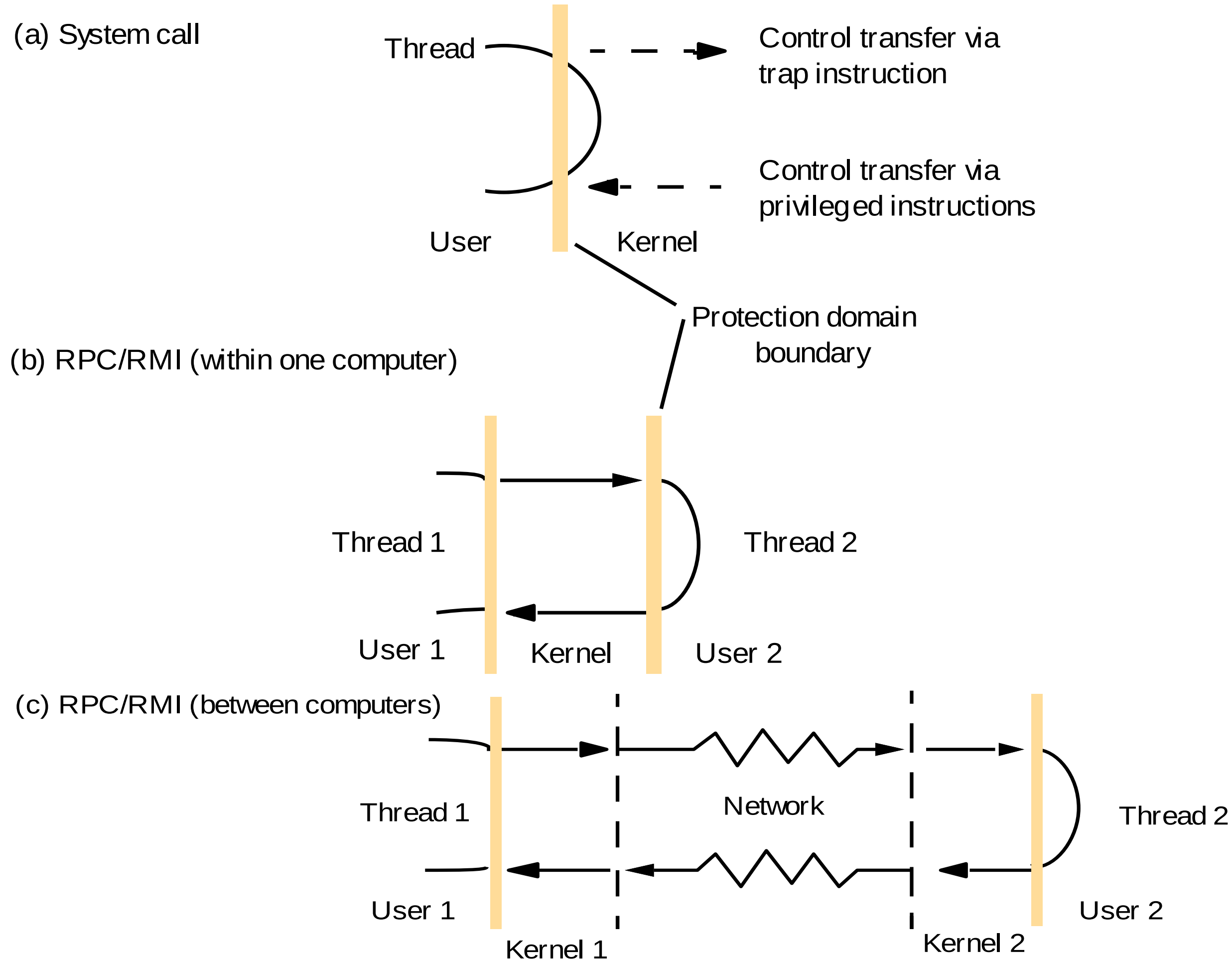


B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

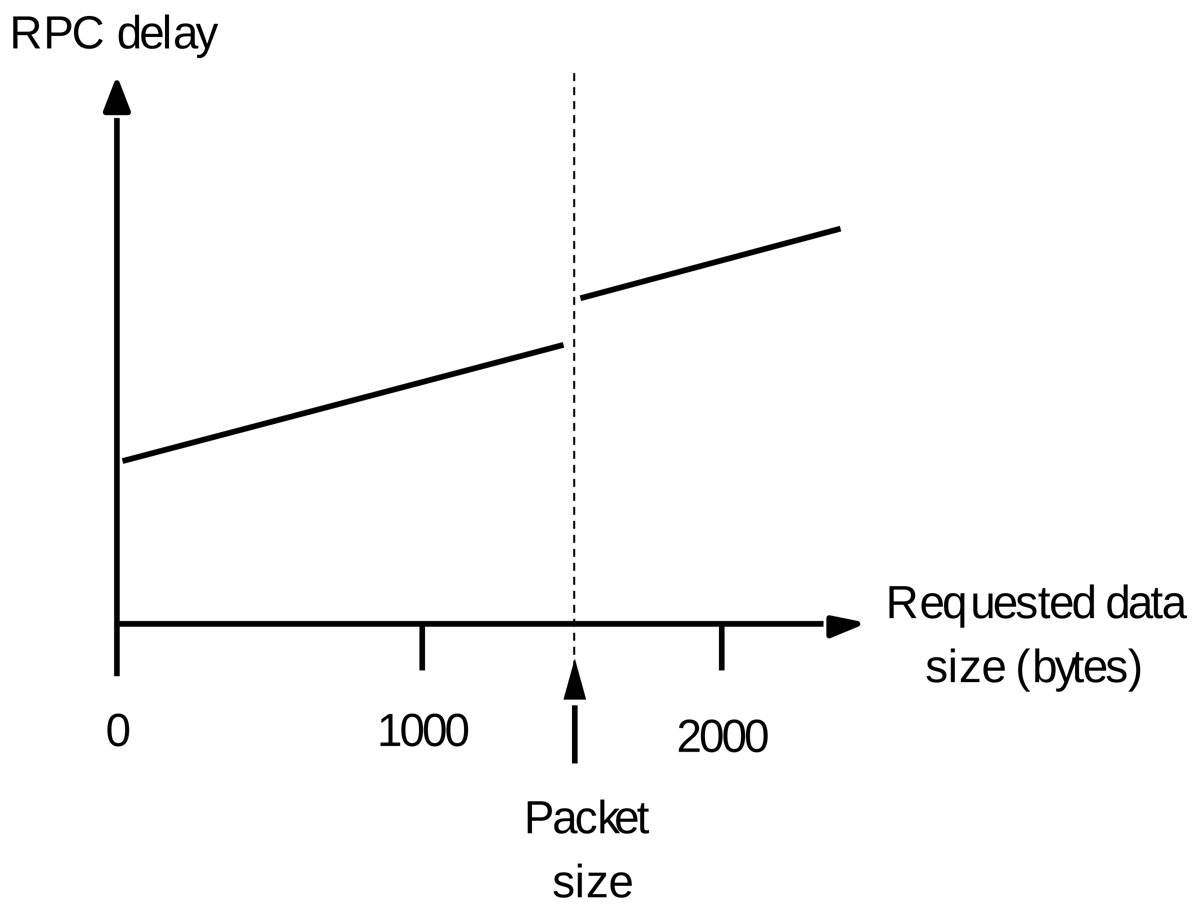
3 virtual processors are assigned to 2 processes (A, B). Virtual processor as kernel can assign different physical processor as time goes by.

Process notifies kernel when processor (P) is idle or when extra P is needed. Kernel notifies P on any of the shown 4 events occur.

Operating System Support:
Invocations between address spaces

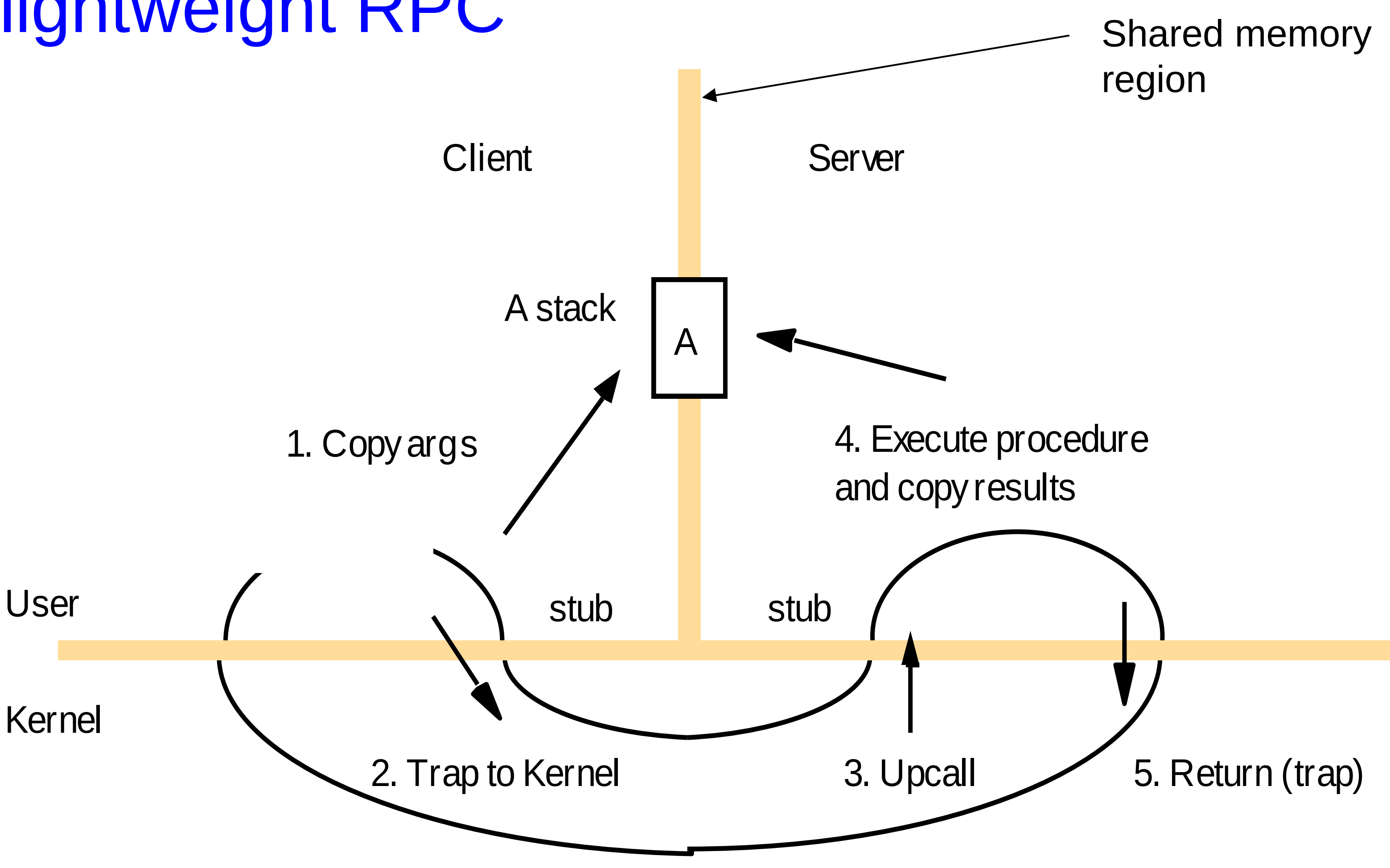


Operating System Support:
RPC delay against requested data size

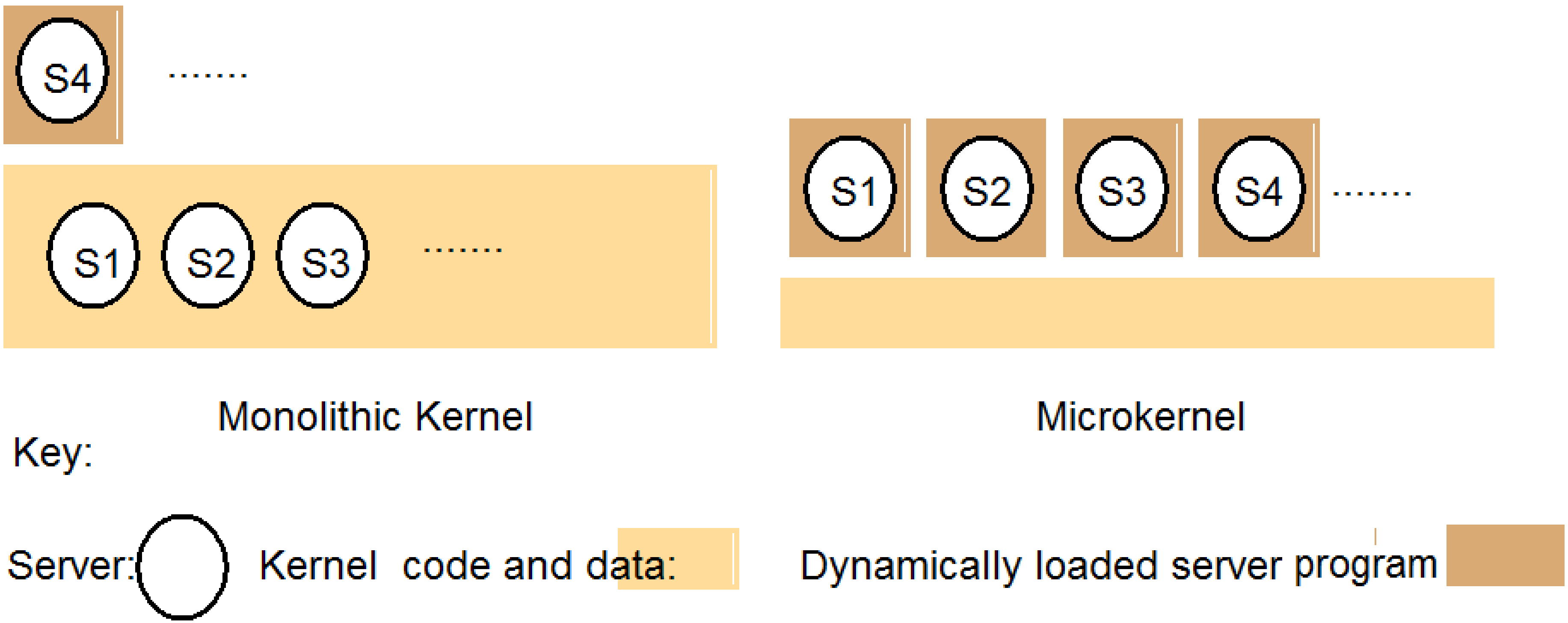


Operating System Support:

A lightweight RPC



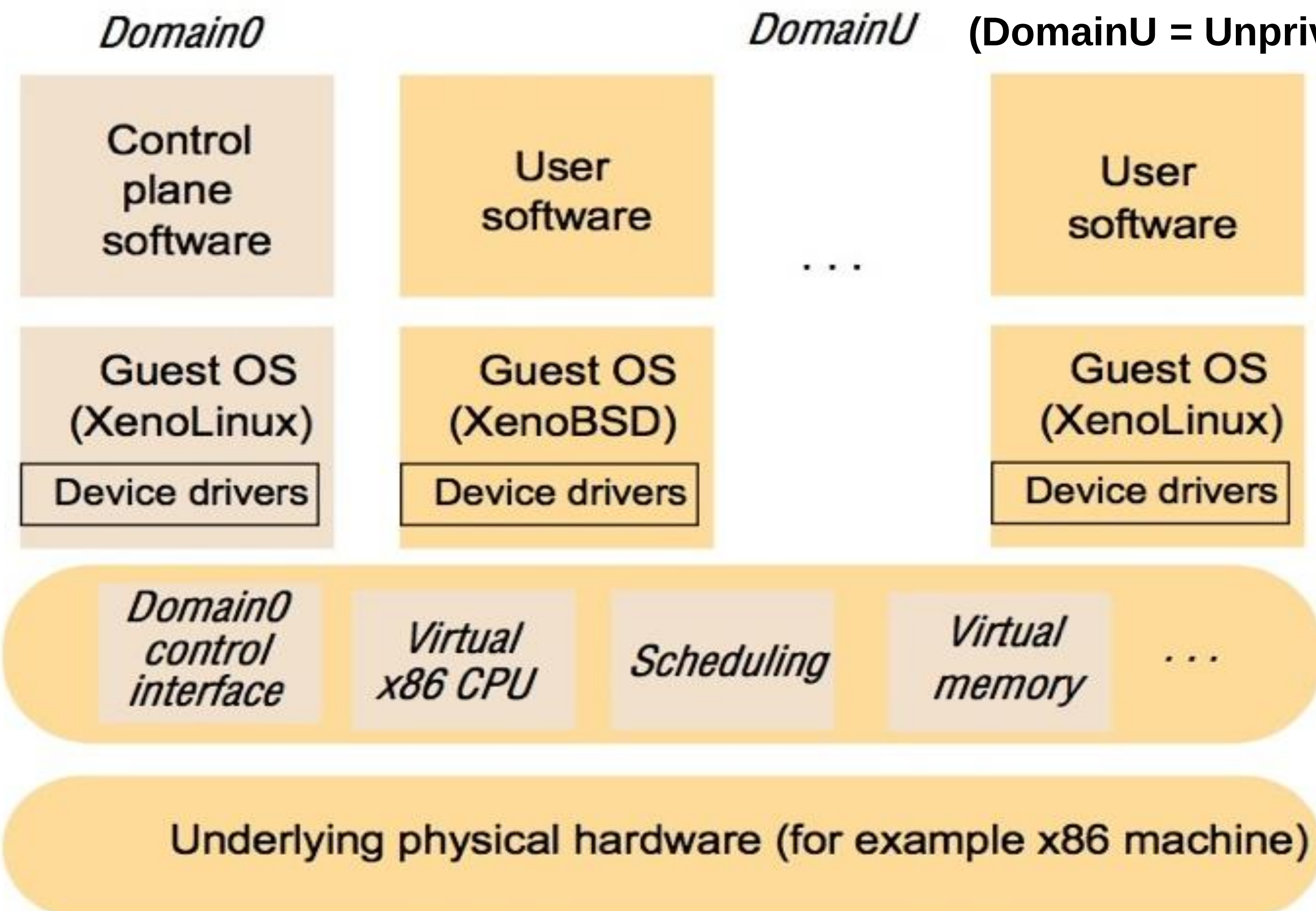
Operating System Support:
Monolithic kernel and microkernel



S1, S2, and S3 are kernel subsystems like file system, Scheduler, etc.
S4 is an application user-level process

Operating System Support:

The architecture of Xen/VM http://www.xen.org



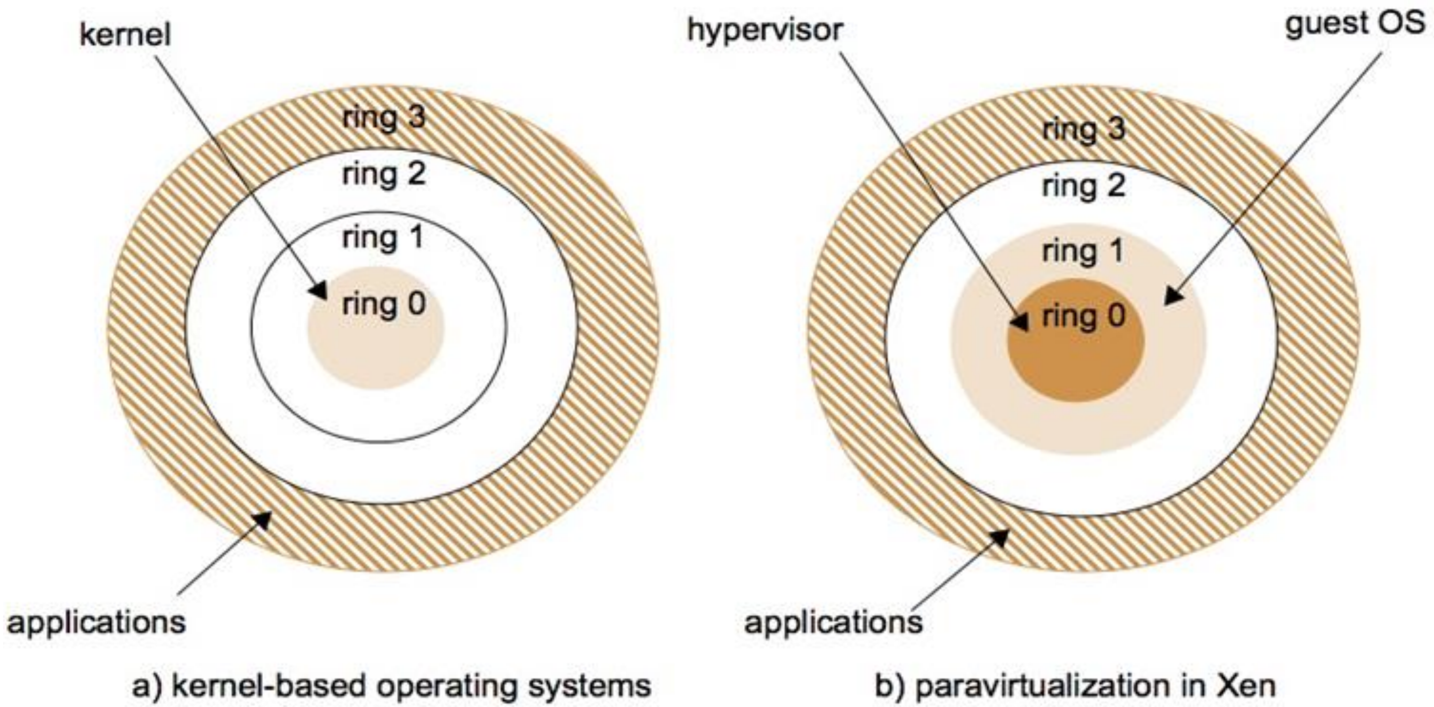
(DomainU = Unprivileged Domains)

Domain = VM

Xen hypervisor

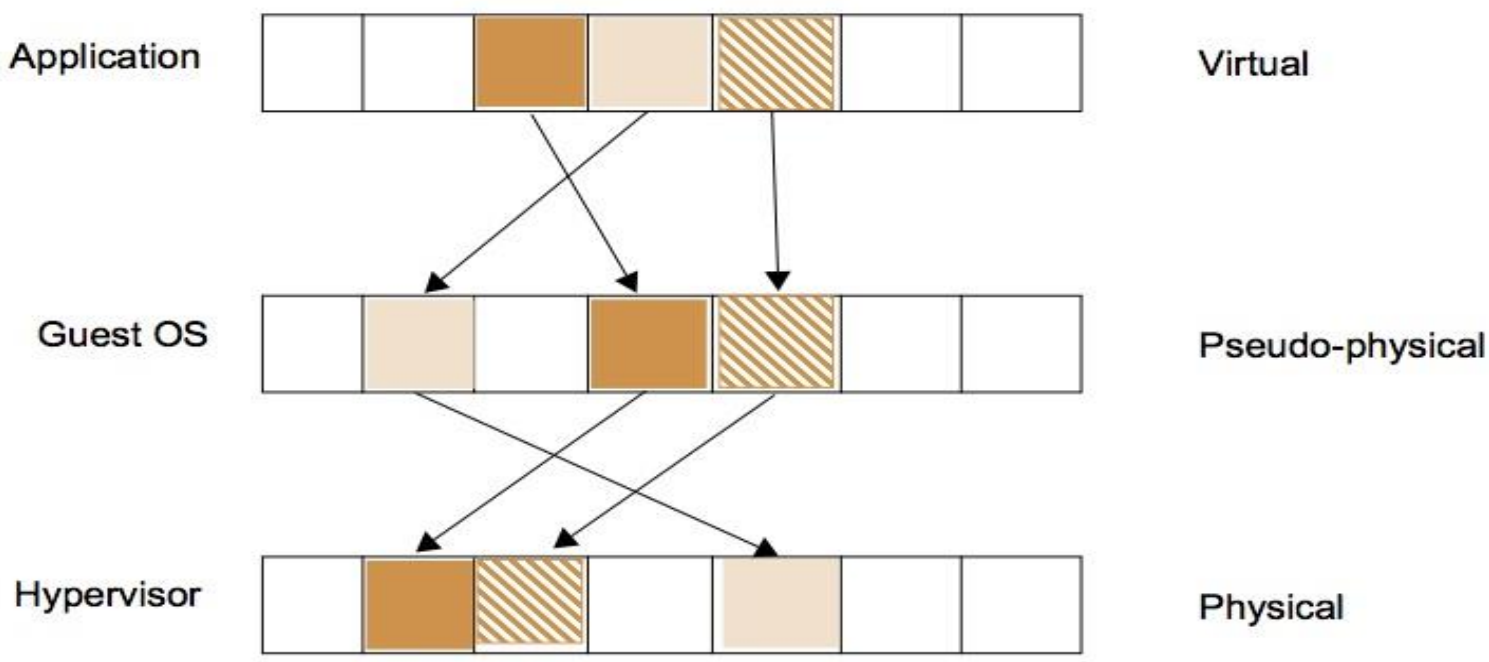
Virtual M/C Monitor (VMM)

Operating System Support:
Use of rings of privilege



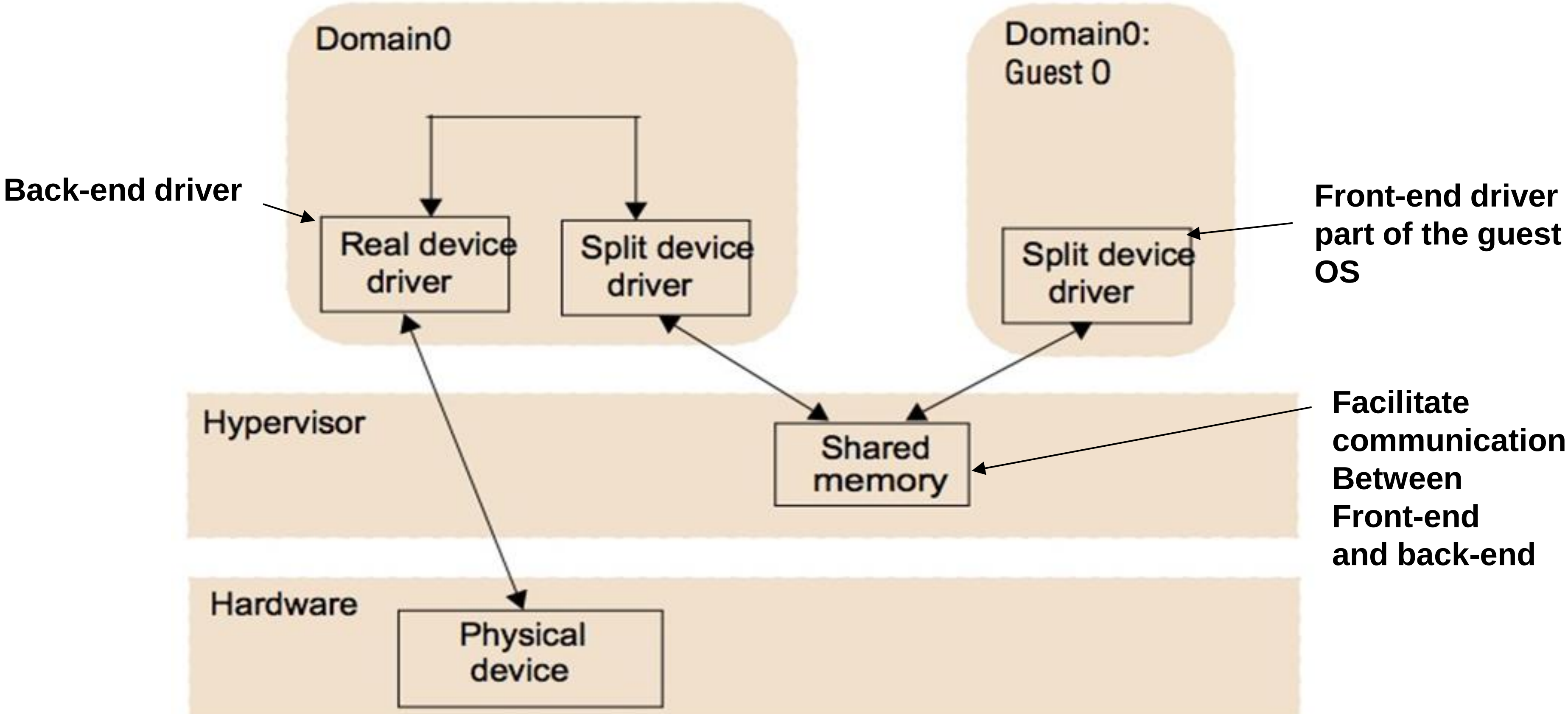
Kernel rings reflect sensitivity to the different M/C instructions (privileged vs. nonprivileged). The problem is some of these non-privileged instructions are sensitive instructions with side effect. Instead of simulating all instruction and map them into real M/C instructions which would allow us full control on all instructions, Paravirtualization, executes many instructions directly on the bare hardware and only privileged instructions are trapped and dealt with in the hypervisor. Unfortunately, this means you need to re-write the OS to deal with these sensitive non-privileged instructions.

Operating System Support:
Virtualization of memory



- **Hypervisor** is responsible for managing physical memory in terms of pages
- **Hypervisor** allocates physical page to Domains on-demand
- Pseudo-physical address space provides the abstraction of contagious pseudo-physical memory by maintaining mapping between this pseudo-physical and real physical memory. This mapping need to be managed by the guest OS and not the hypervisor.

Operating System Support:
Split device drivers



- Access to physical device is controlled exclusively by Domain0.

Case Study: RPC

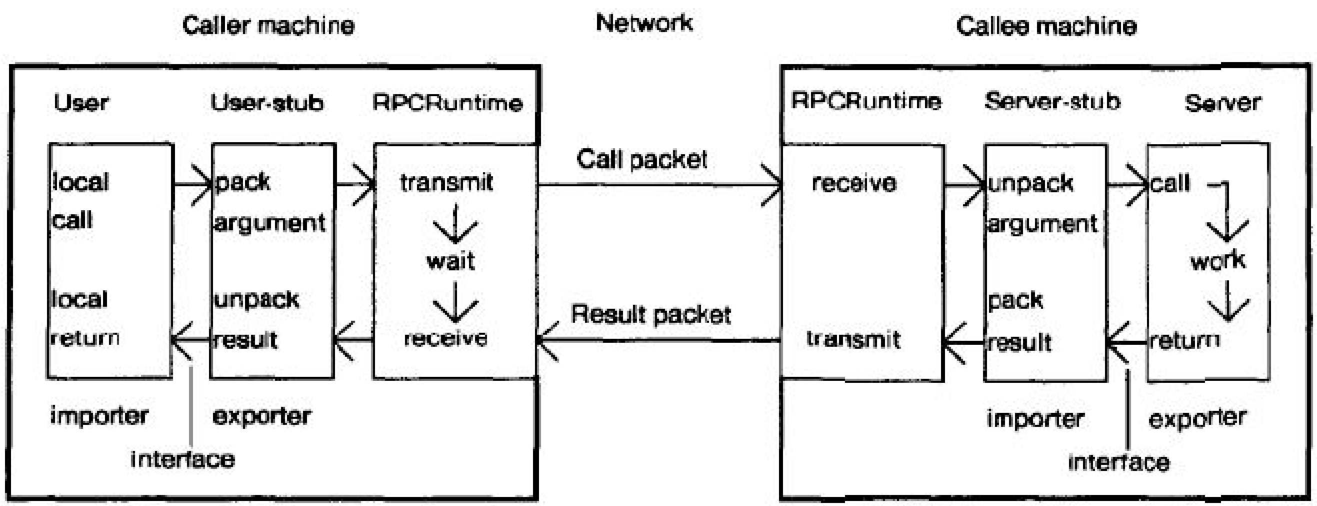
RPC: Why

- **Familiar paradigm** (local procedure call), simple semantics
- **Provides runtime distribution transparency**
- **Productive environment** as the stub compiler generates the difficult code for you such as marshalling and unmarshalling
- **General purpose:** client and server can be written in different languages with no dependencies.
- Code generated by the stub compiler will use Messages to communicate between client and server.
- **Examples**: SUN RPC or CORBA IDL

RPC: Alternatives

- **Messages like sockets:** too much work and risk of code bugs
- **Distributed shared memory:** requires hardware support and it is expensive
  - UMA is not scalable
  - NUMA is very expensive
- …..

RPC: Architecture



The components of the system, and their interactions for a simple call.

- You need to write the server interface in IDL
- **Compile the IDL file and that generates:**
  - **Client stub:** code that marshall client invocation and turn into message to be sent to the skeleton code in the server process, as well as unmarshall the response to return the result to the client code. The client stub after compilation will need to be linked with your client-side application

RPC: Architecture

- **Server Skeleton** : code that unmarshall client message request and does local function call on the server function, as well as marshall the server function output to send back to the client stub
- **Header file** that will be needed to compile both the client stub and server skeleton.

- **Client and server code** can be written in different programming languages such as Java, C, C++, etc.
- **The client stub will locate the remote receiver** (server skeleton)
- **Marshalling** is also known as serialization, and **unmarshalling** is known as de-serialization

RPC: Architecture

- **Typically RPC calls are blocking/synchronous** calls, however you can emulate asynchronous behavior by creating child thread that invokes the client stub and the parent thread can do something else – you need synchronization between the parent and child threads.
- **Communications** between stub and skeleton is connectionless.
- **Sever Skelton** supports idempotent operations via repeat/reply cache

END