Coordination-based Distributed Systems

Vania Marangozova-Martin

ids.forge.imag.fr

Motivation

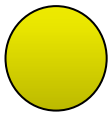
- Many distributed systems are centered around data
  - files
  - objects
  - web pages
  - …
- They are extensions to centralized systems
  - allow the remote access to data
- Coordination-based systems consider that
  - the components are inherently distributed
  - the challenge is to make them work together
    - control, coordinate

Key Idea

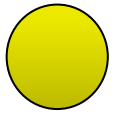
- Clear separation between computation and coordination
  - Computation = the application functionality implemented by the different components (processes)
  - Coordination = communication

Example:
A « simple » distributed system

- Goal: compute the total number of votes from all voting sections scattered all over the country



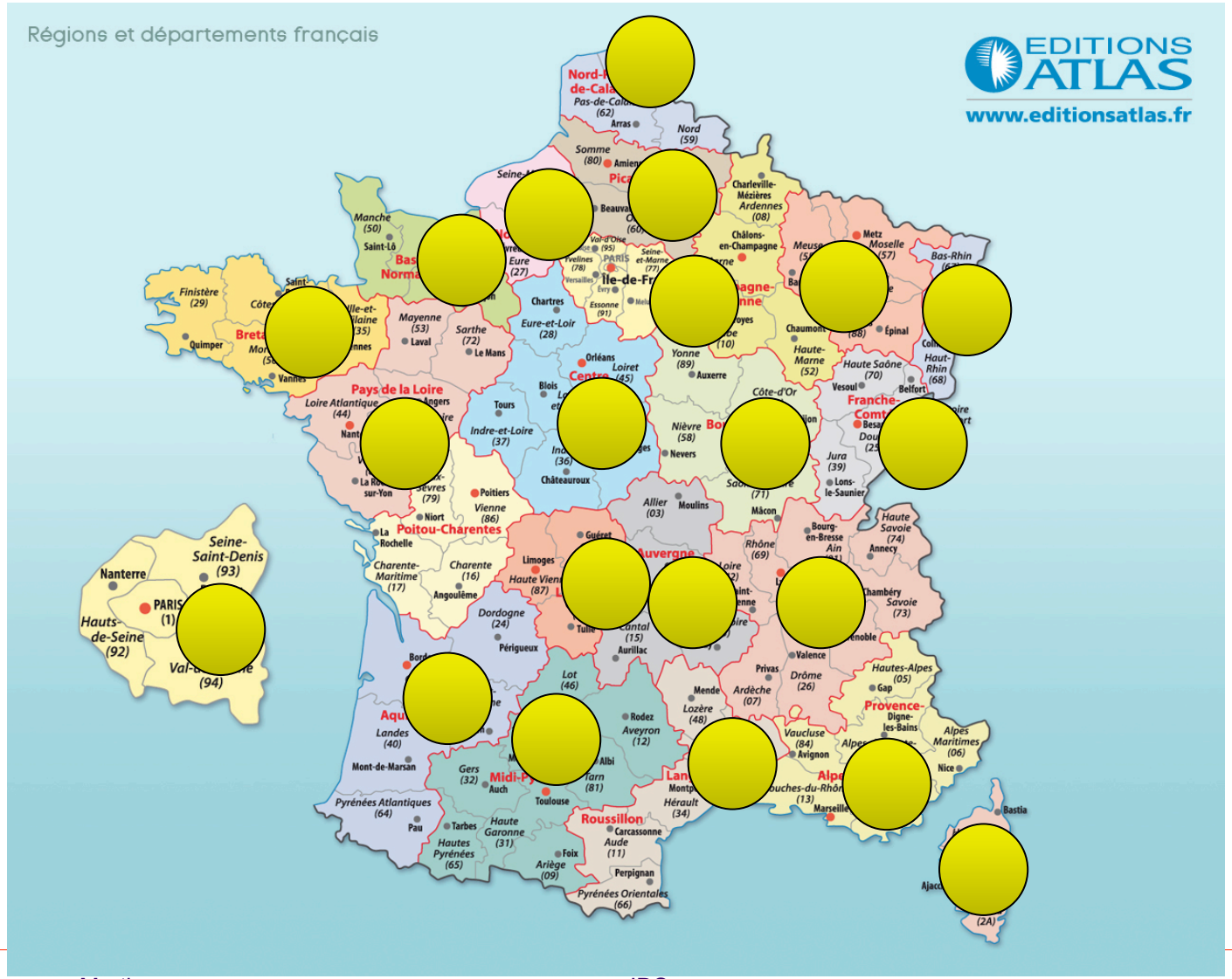
Voting section



localV = number of local votes

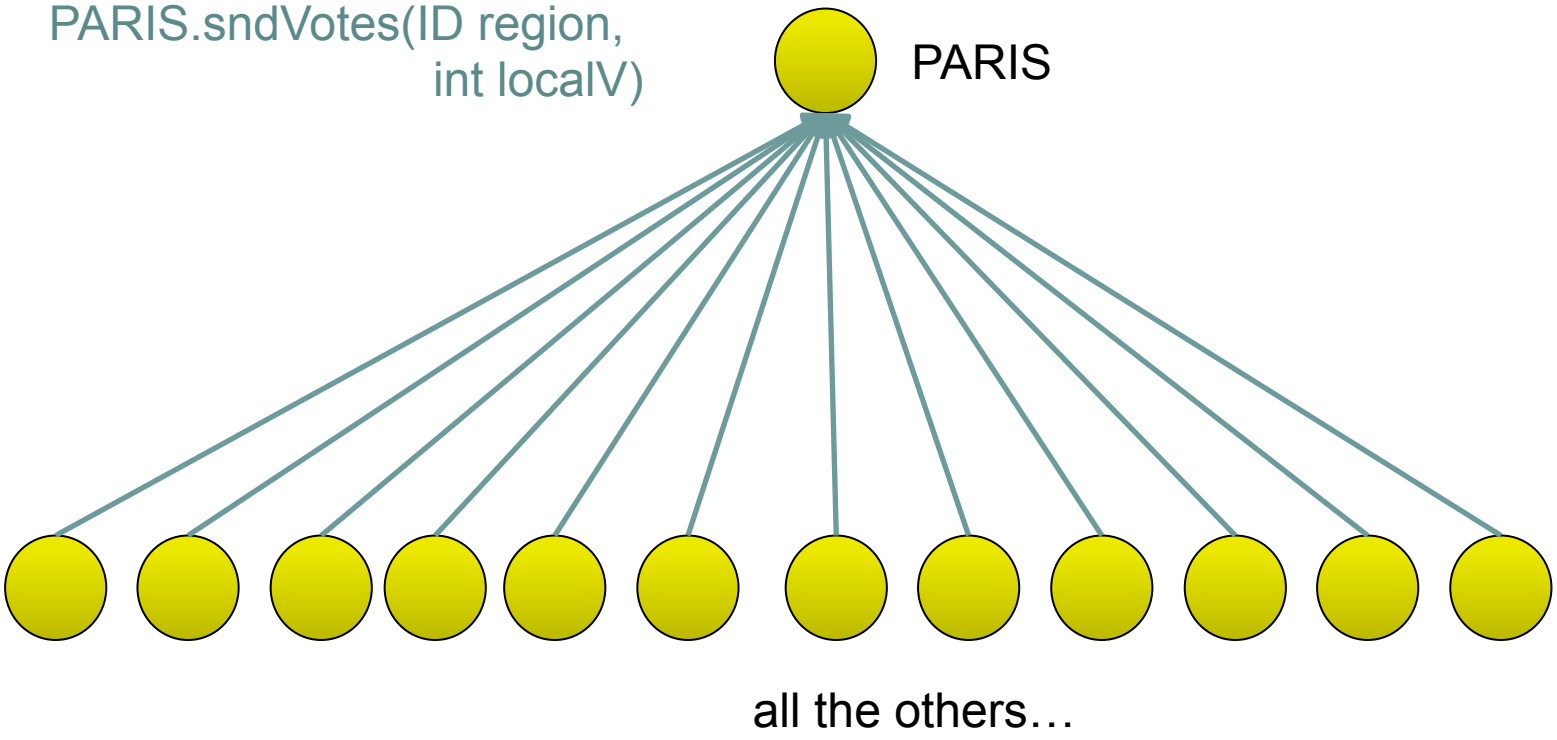
- Compute the sum of localV

Example:
A « simple » distributed system (2)



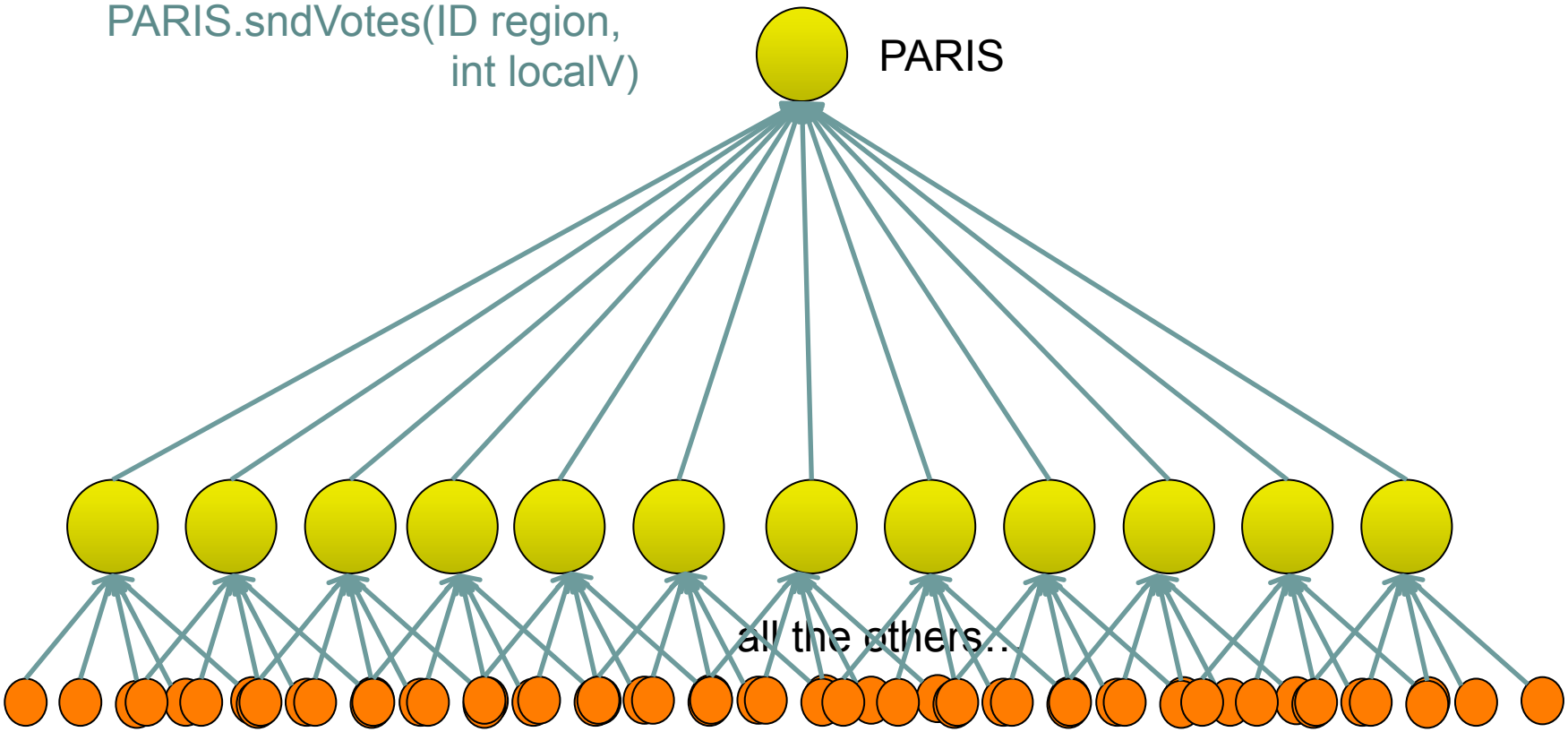
Example:
A « simple » distributed system (3)

- A client-server implementation

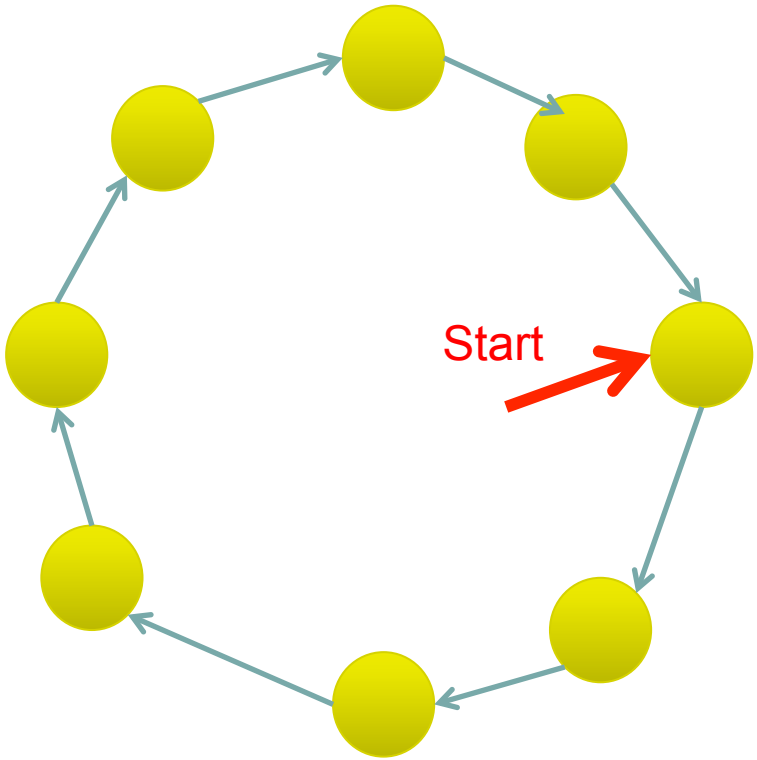


Example:
A « simple » distributed system (4)

- A client-server implementation in a tree



Example:
A « simple » distributed system

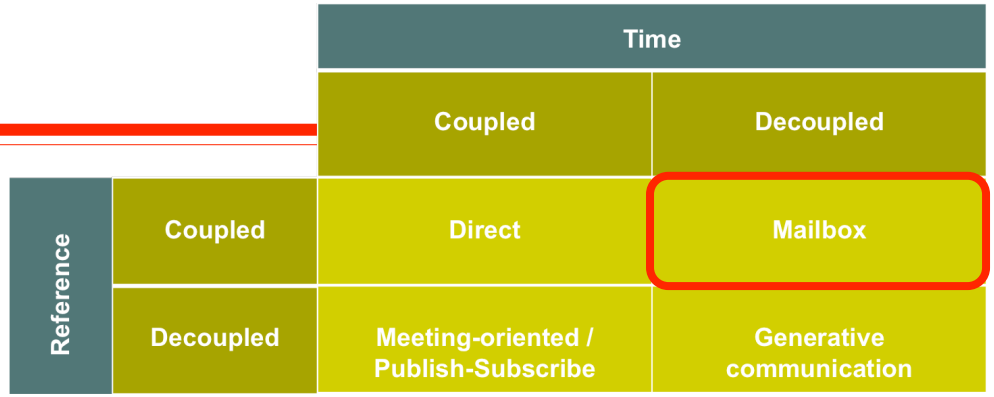



```
LOCAL NODE ALGORITHM
//Start computation
START ? Start =>
    OUT.Send(<me, 0>);

//Receive a msg and the
//computation should go on
IN.Recv(Msg m = <id I, int x>)
&& I!=me =>
    OUT.send(x+local);

//Finish the computation
IN.Recv(Msg m = <id I, int x>)
&& I==me =>
    TOTAL = x+local;
```

Coordination Models

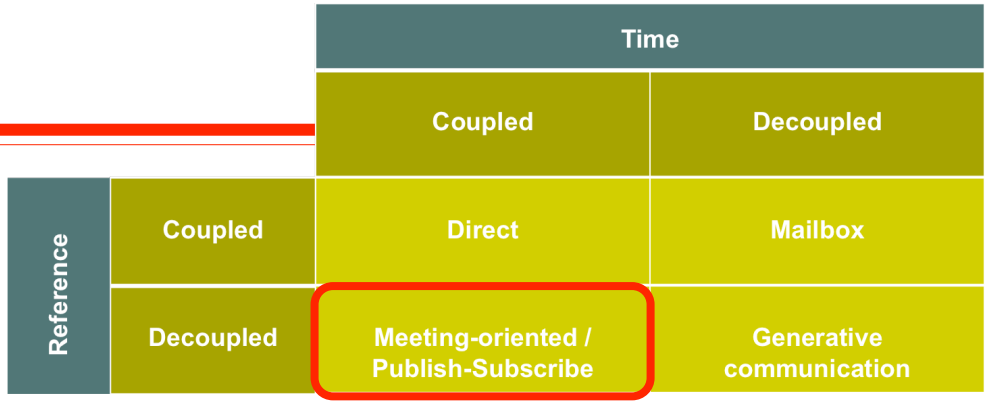
		Time	
		Coupled	Decoupled
Reference	Coupled	Direct	Mailbox
	Decoupled	Meeting-oriented / Publish-Subscribe	Generative communication

Mailbox



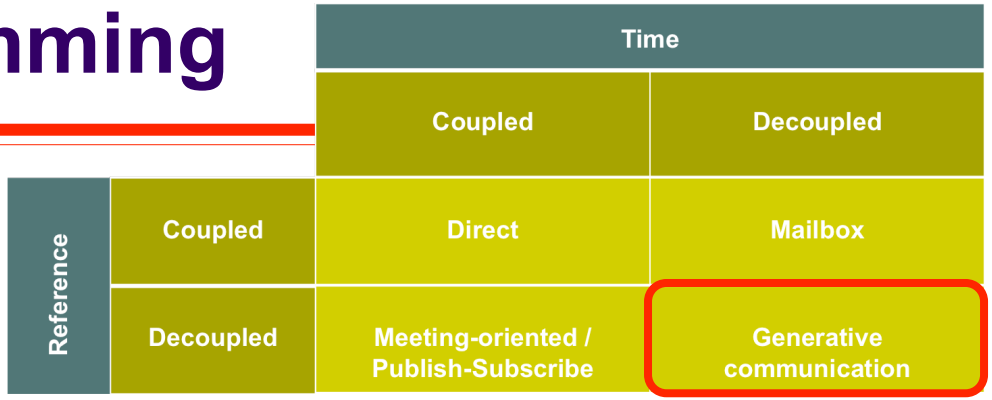
- Processes do not need to be all present (active) to communicate
  - A process puts messages in a mailbox
  - Later, another process may read the messages
  - The first process may be inactive
    - Need for a « mailbox » that exists outside the processes
    - Processes need to know (name) the mailbox

Meeting-Oriented



- Components do not know each other
  - The members of the communicating group may evolve dynamically
- They need to meet to coordinate
  - The meeting is established on the basis of « commun interests »

Generative Programming



- Processes put tagged data tuples in a shared dataspace
- Other processes may read/write the tuples later
- The tag is used to represent different types of information
- *Introduced by the Linda system (1985)*

Why Decoupling?

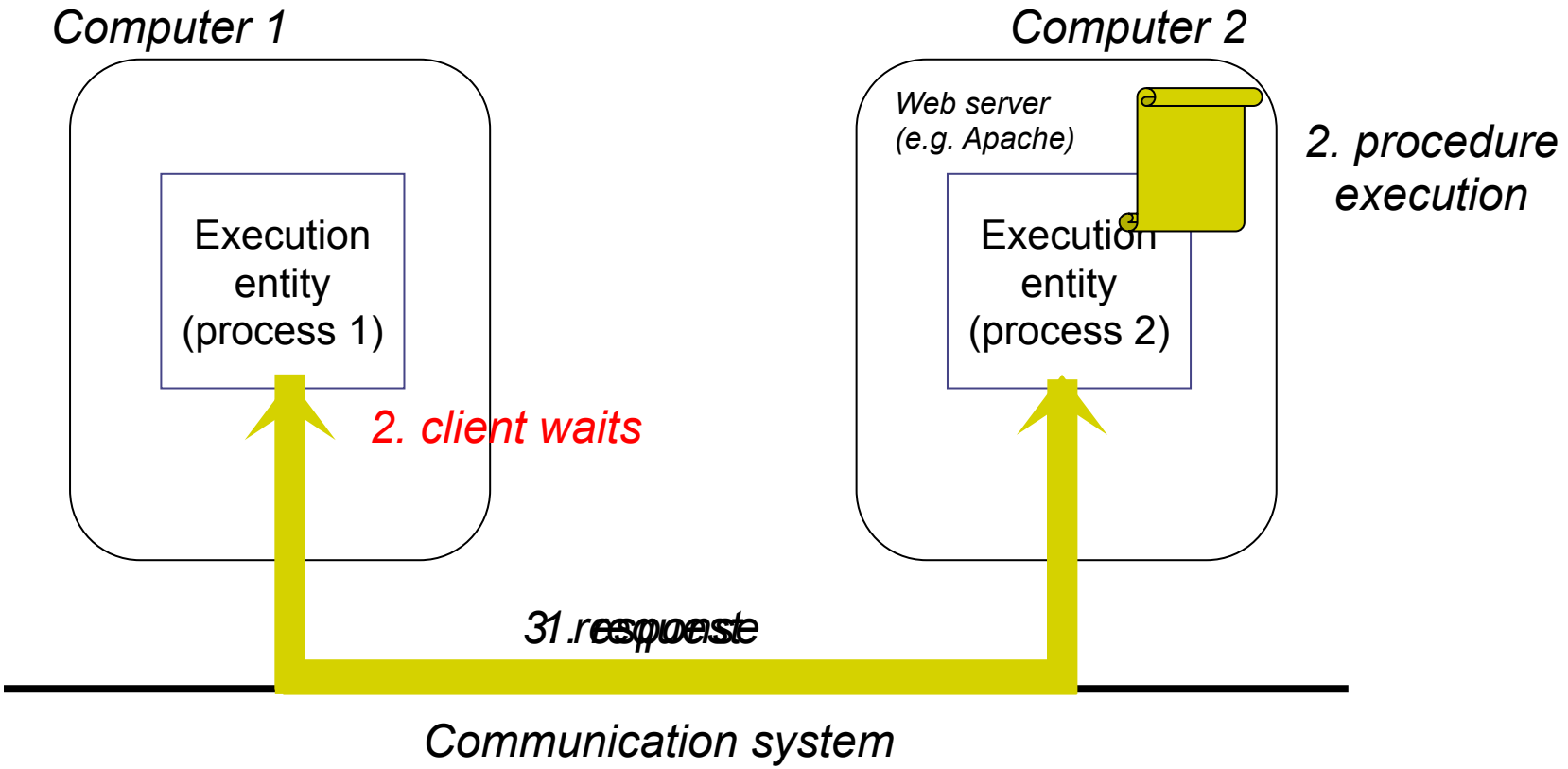
- To deal with change
- Examples of problems encountered in a tightly-coupled environment
  - In a traditional client-server system, it is difficult to update the server's software in a transparent way, without disturbing the functionality of the client
  - Similarly, if the server fails, this directly affects the client, which must explicitly deal with the failure.

Indirect Communication

- Communication between entities in a distributed system
  - through an intermediary
  - with no direct coupling between the sender and the receiver(s)
- *Space uncoupling*
  - The sender does not know or need to know the identity of the receiver(s), and vice versa.
- *Time uncoupling*
  - The sender and receiver(s) can have independent lifetimes.

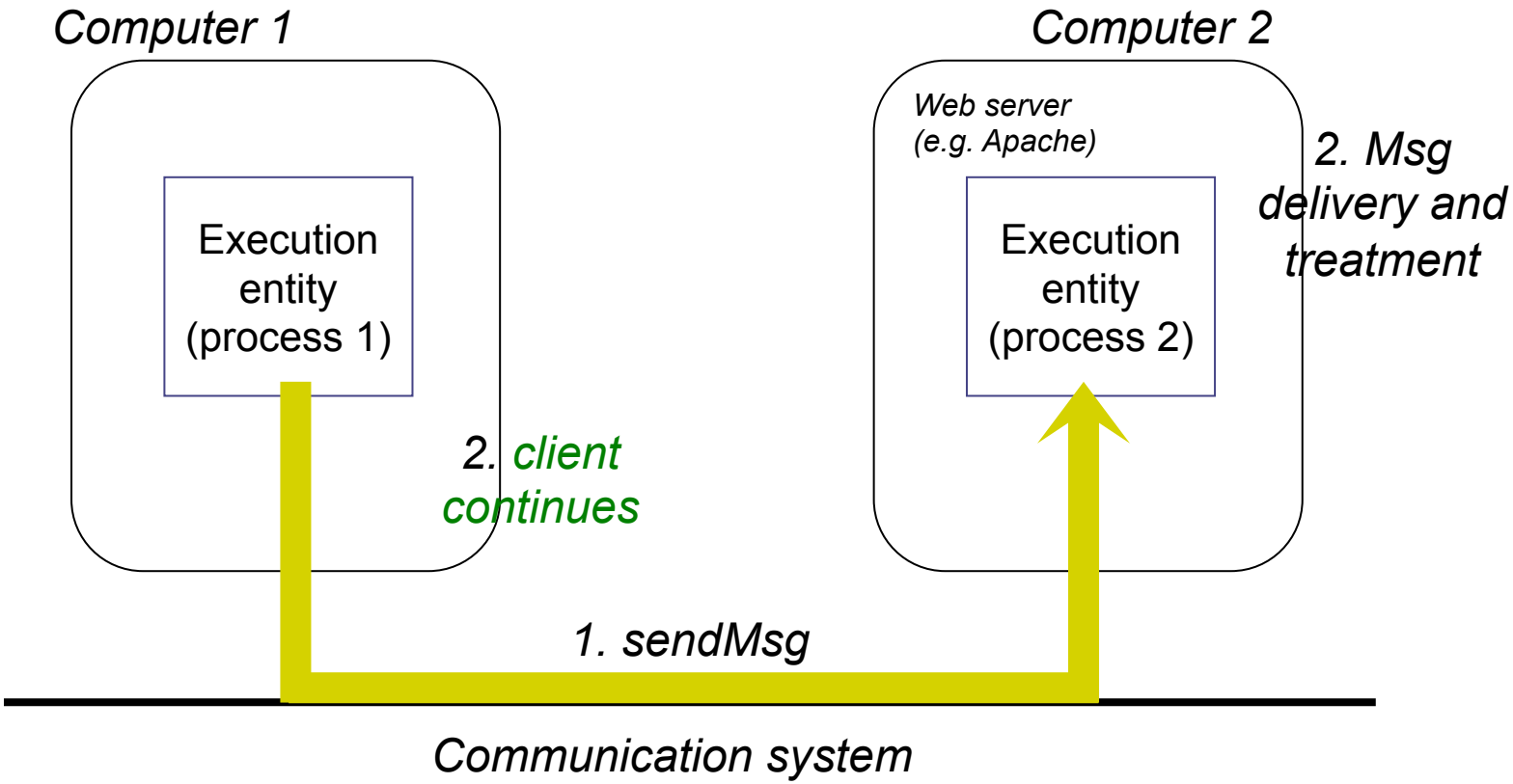
Indirect Communication versus Asynchronous Communication (1)

- Synchronous communication: reminder



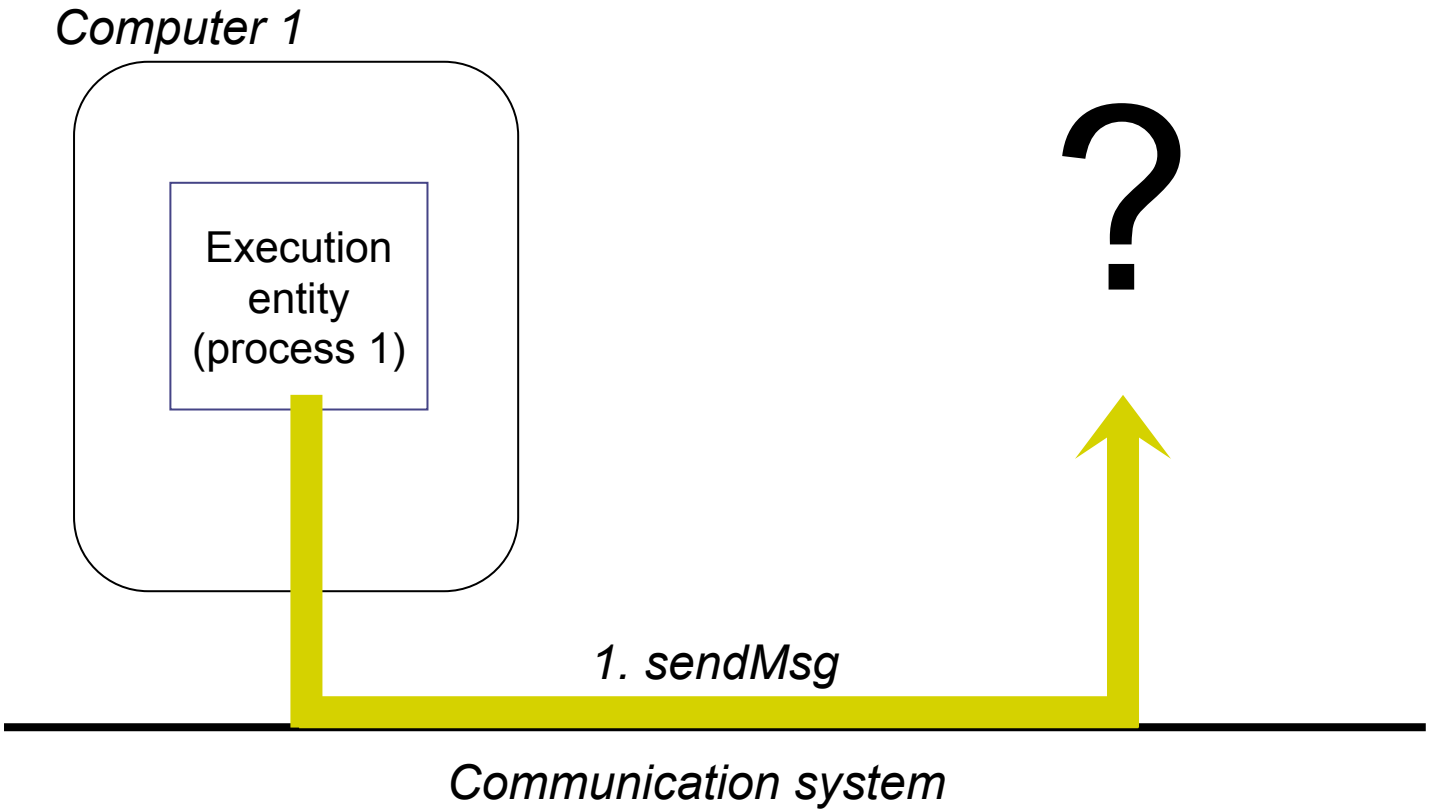
Indirect Communication versus Asynchronous Communication (2)

- Asynchronous communication: reminder



Indirect Communication versus Asynchronous Communication (3)

- Indirect communication

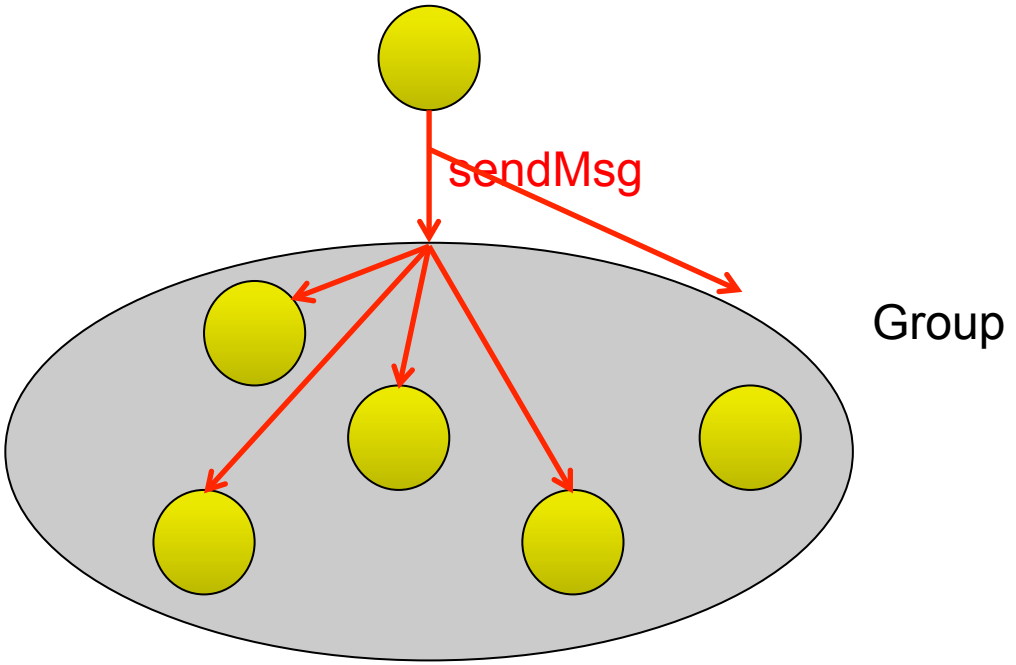


Indirect Communication Schemes

- Group Communication
- Publish-Subscribe
- Message Queues
- Distributed Shared Memory

Group Communication

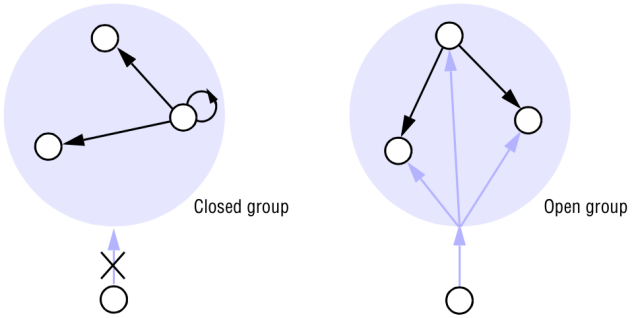
- A sender sends a message to a group
  - without knowing who the members are



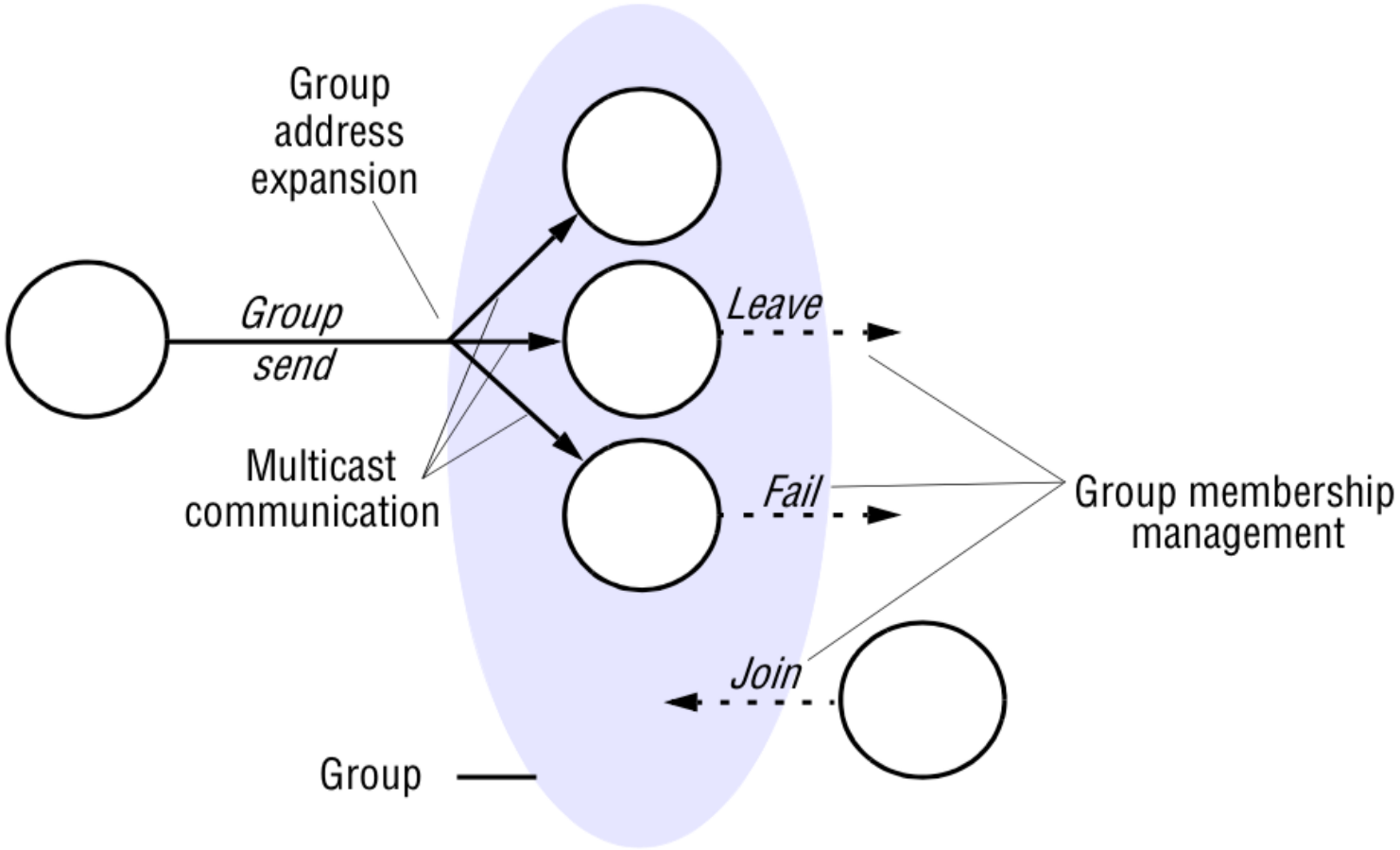
- Important building block in *reliable* distributed systems

Group Communication

- An abstraction over network multicast communication
- Provides additions in terms of
  - member management
  - failure management
  - reliability in message delivery
  - order guarantees
    - Ex. If Sender1 sends M1 and then M2, is it guaranteed that all the members of the group receive M1 first and M2 second?

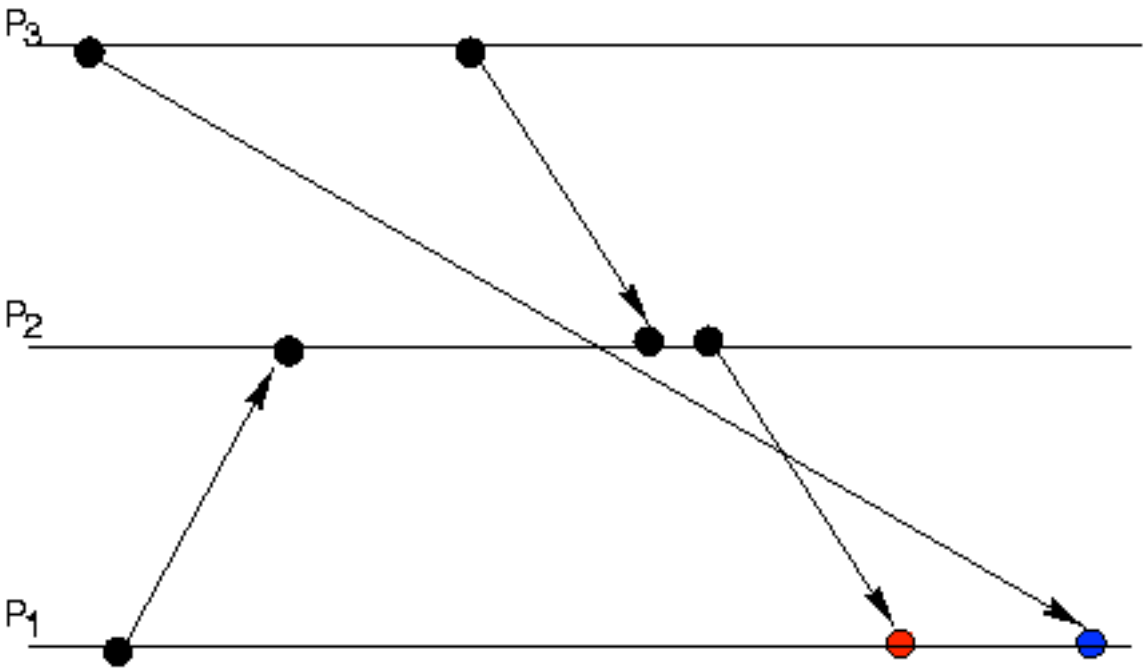


Group Management



Order in Communication

- Three distributed sites communicating through messaging : PB at site P1



Case study: the JGroups toolkit

- http://www.jgroups.org/
- A toolkit for reliable group communication written in Java
  - process groups
  - join/leave a group
  - send a msg to one or all members of a group
  - receive a msg from a group

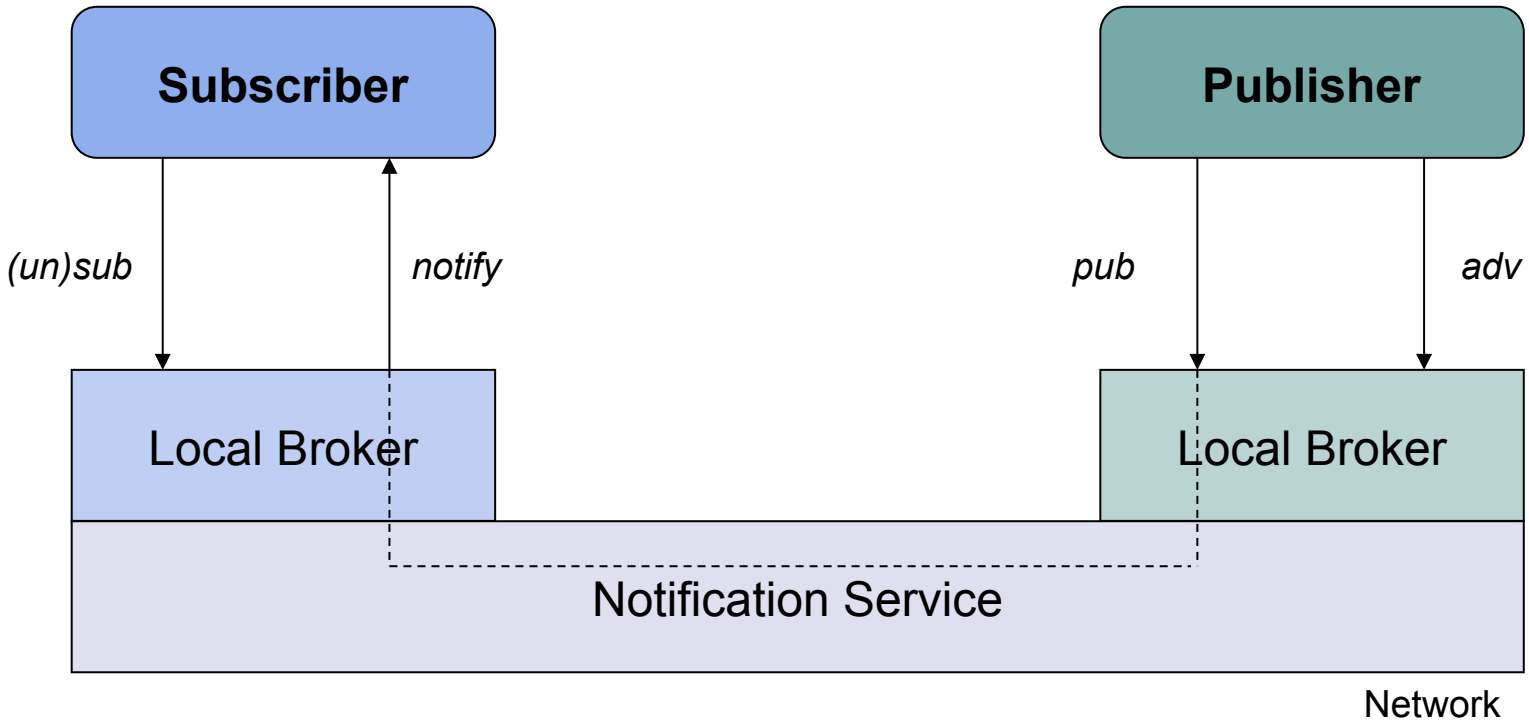
Indirect Communication Schemes

- Group Communication
- **Publish-Subscribe**
- Message Queues
- Distributed Shared Memory

		Time	
		Coupled	Decoupled
Reference	Coupled	Direct	Mailbox
	Decoupled	Meeting-oriented / Publish-Subscribe	Generative communication

Publish-Subscribe

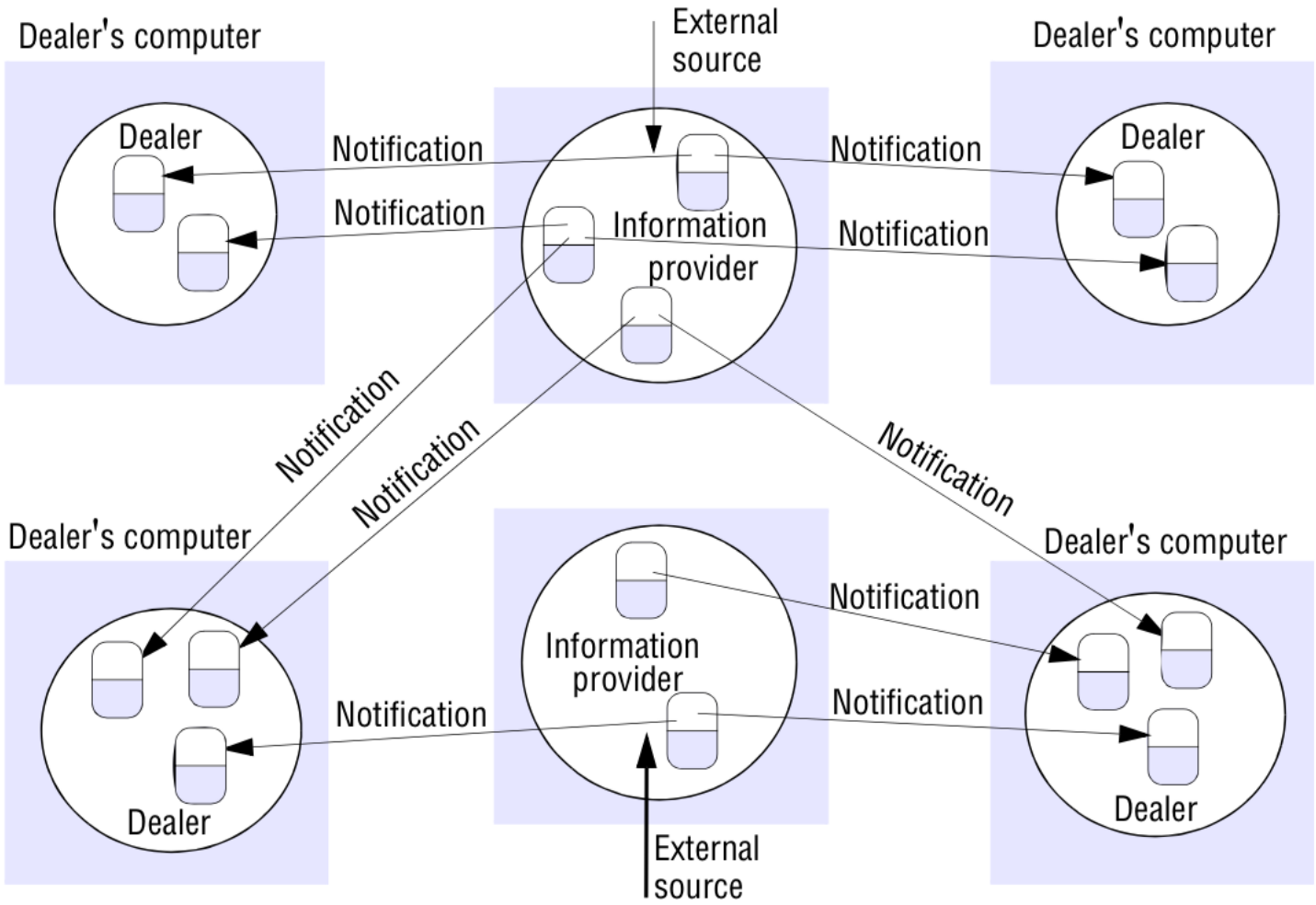
- *Publishers* publish structured events to an event service
- *Subscribers* express interest in particular events
  - *subscriptions* can be arbitrary patterns over the structured events



Applications

- Multiuser games: disseminate events for a consistent view among players
- Financial systems: disseminate real-time financial information (Nasdaq…) to financial institutions
- Consistent management of replicated data
- System monitoring and management

Dealing Room Example

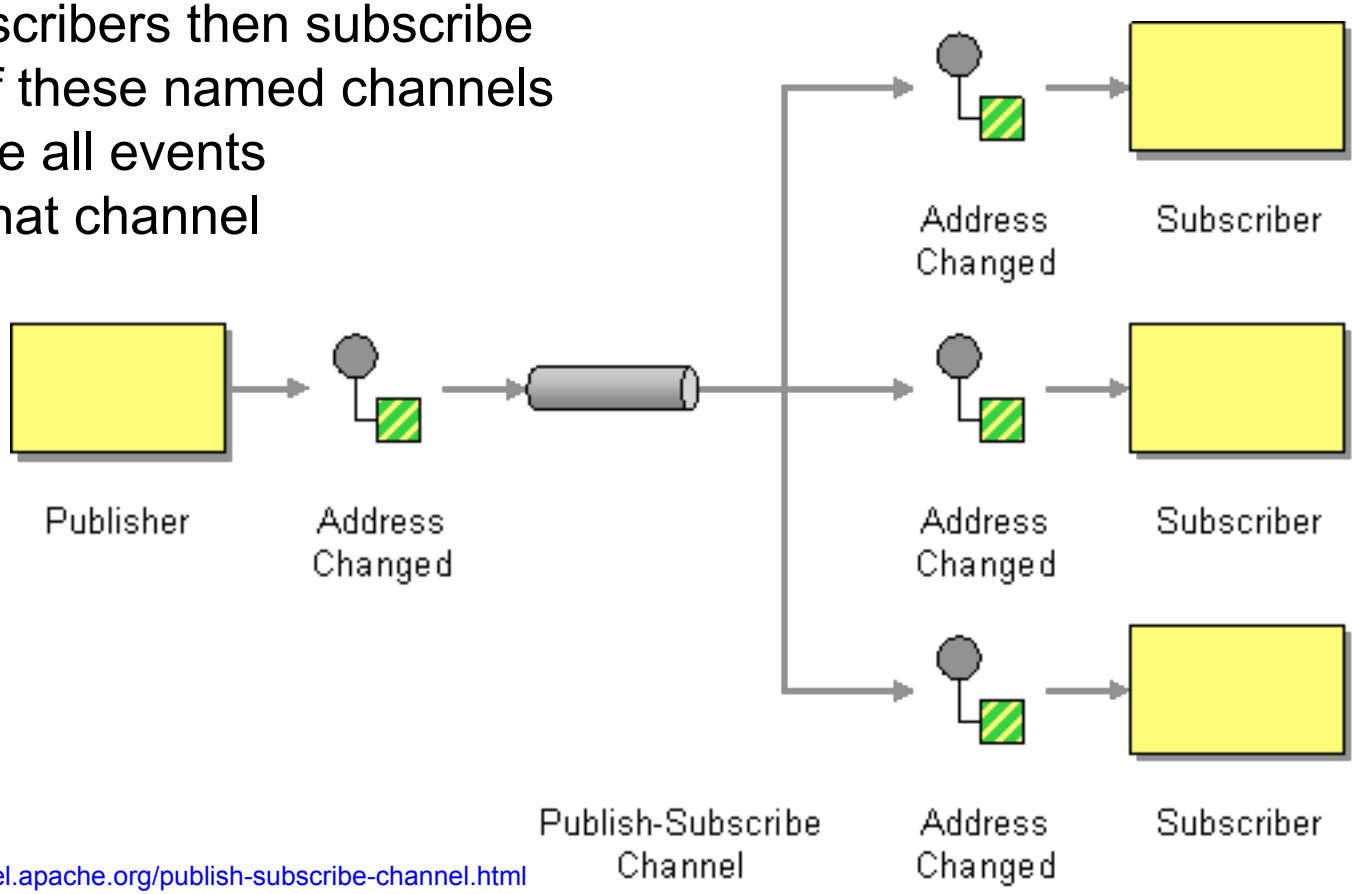


Characteristics of publish-subscribe systems

- *Heterogeneity*
  - When event notifications are used as a means of communication, components in a distributed system that were not designed to interoperate can be made to work together
- *Asynchronicity*
  - Notifications are sent asynchronously to prevent publishers needing to synchronize with subscribers
    - publishers and subscribers need to be decoupled.

Subscription (filter) Model (1)

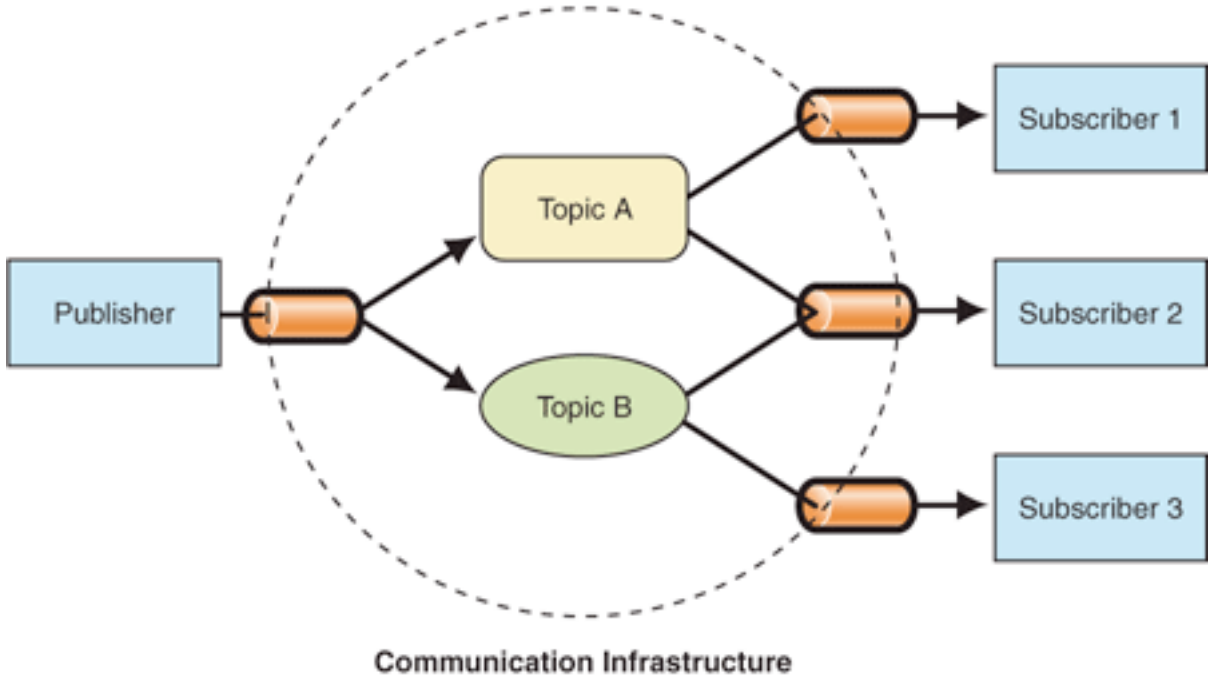
- *Channel-based*
  - Publishers publish events to named channels and subscribers then subscribe to one of these named channels to receive all events sent to that channel



http://camel.apache.org/publish-subscribe-channel.html

Subscription (filter) Model (2)

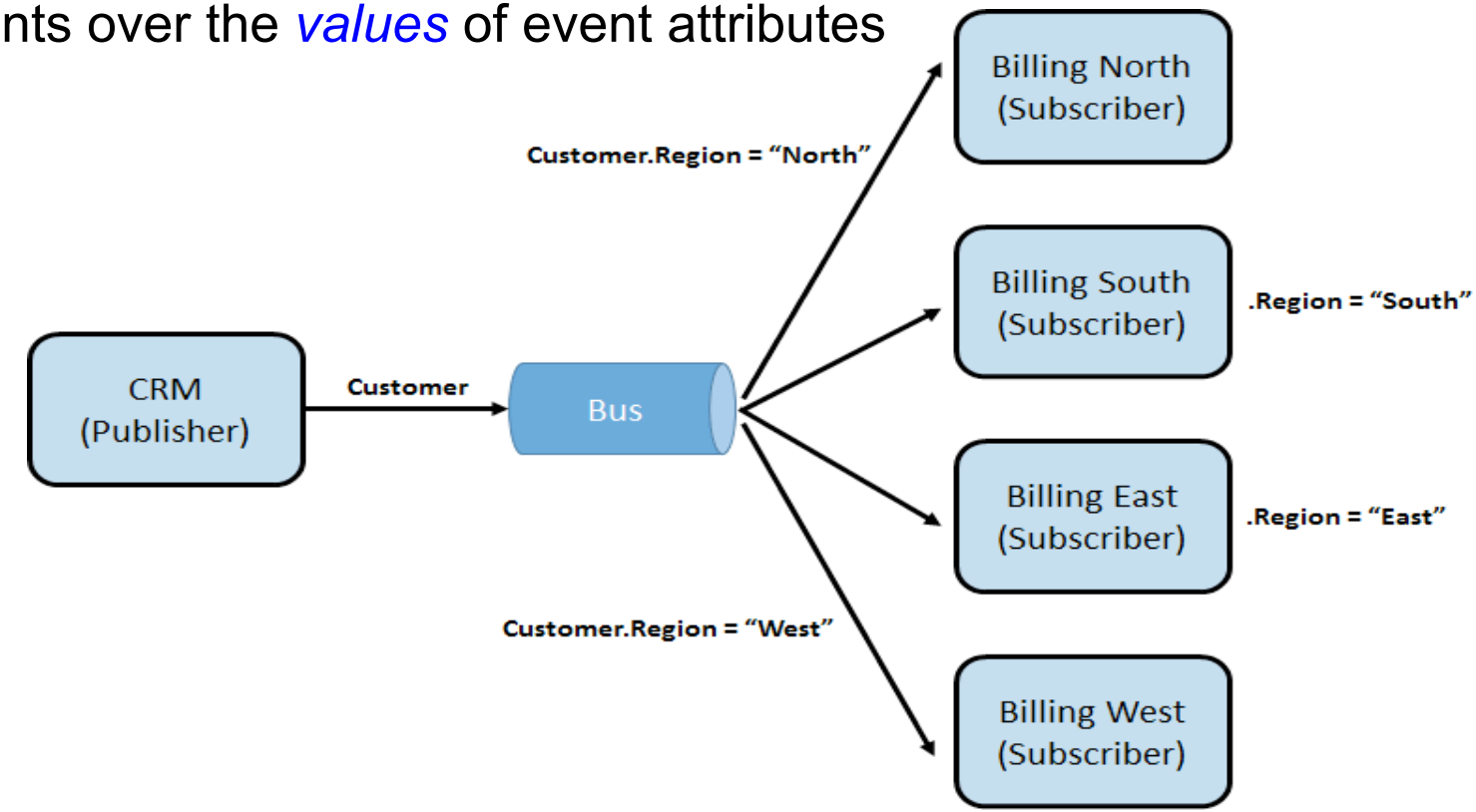
- *Topic-based*
  - Each notification is expressed in terms of a number of fields, with one field denoting the topic.
  - Subscriptions are then defined in terms of the topic of interest.



https://msdn.microsoft.com/en-us/library/ff649664.aspx

Subscription (filter) Model (3)

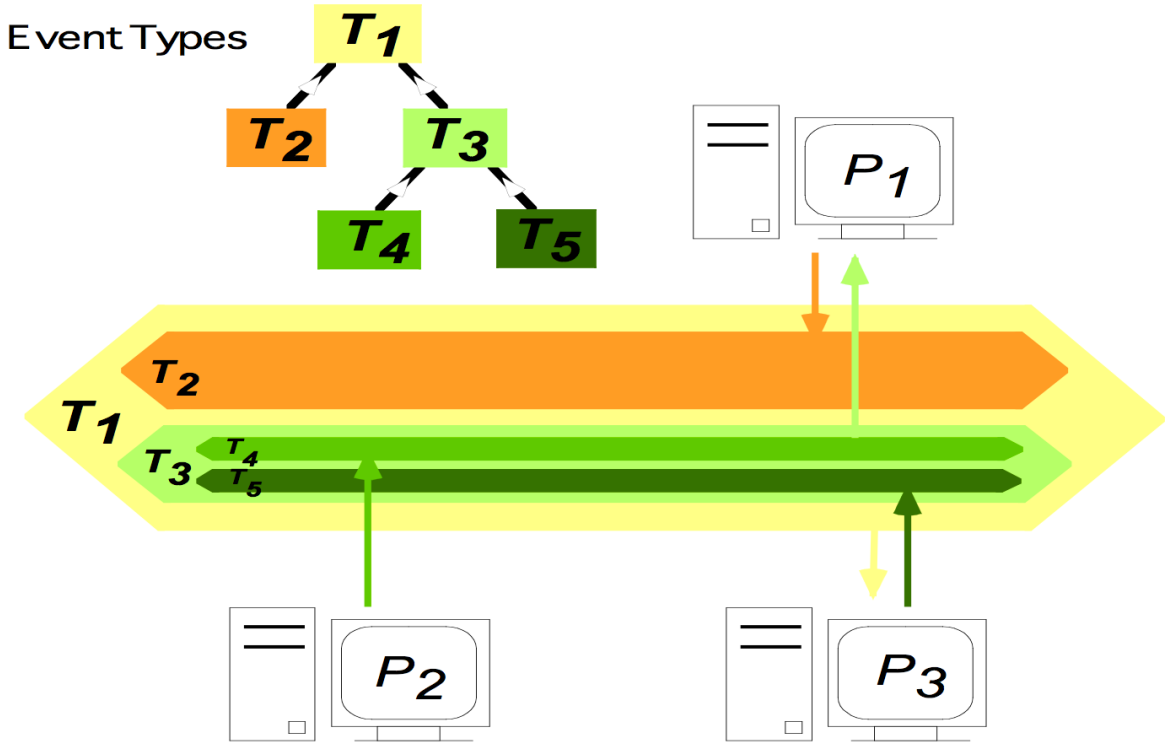
- *Content-based*
  - Generalization of topic-based.
  - A content-based filter is a query defined in terms of compositions of constraints over the *values* of event attributes



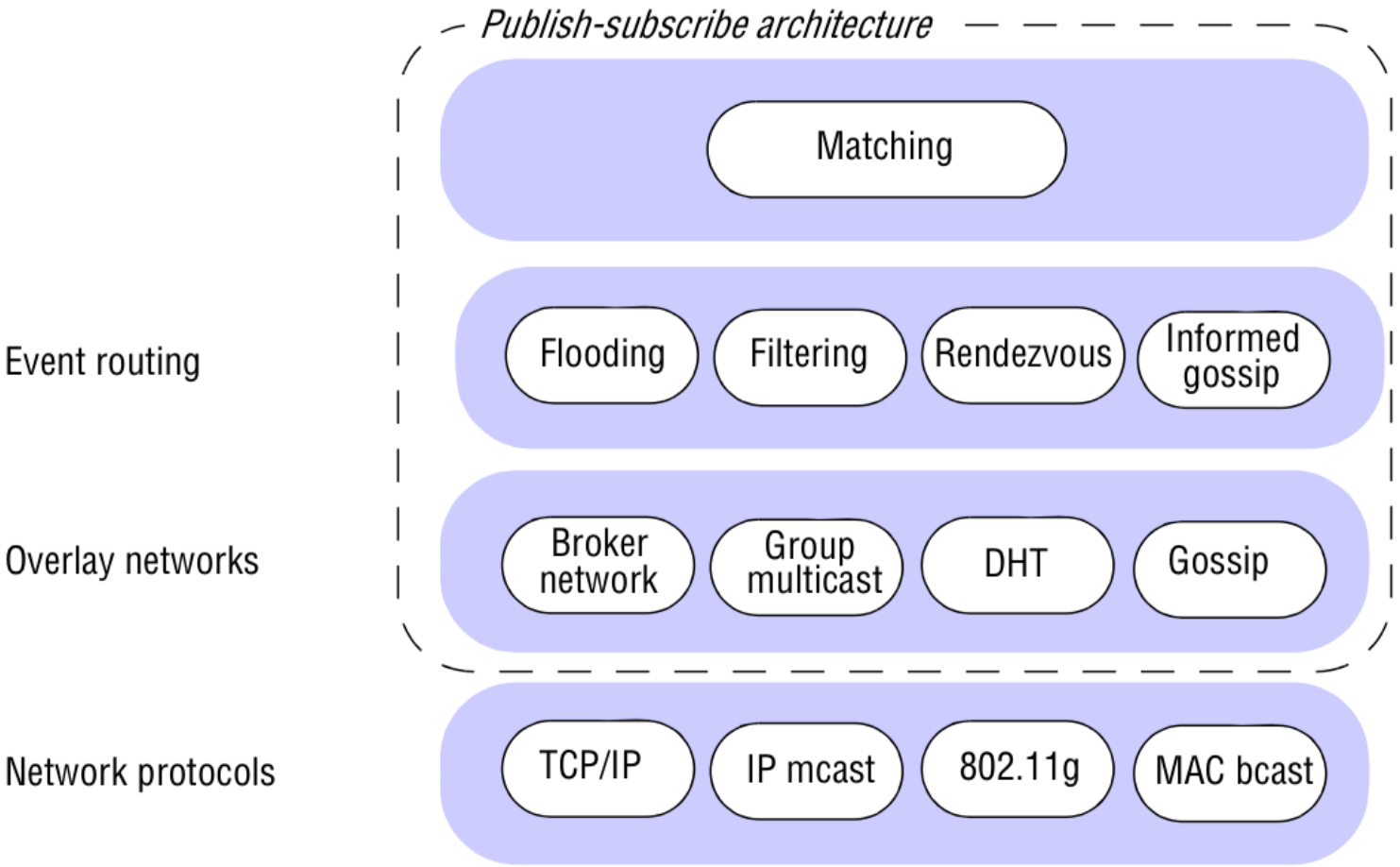
http://blogs.msdn.com/b/dynamics-coe/archive/2013/03/05/simple-publish-subscribe-integration-with-dynamics-crm-using-azure-service-bus-topics-subscriptions.aspx

Subscription (filter) Model (4)

- *Type-based*
  - Intrinsically linked with object-based approaches where objects have a specified type.
  - Subscriptions are defined in terms of types of events and matching is defined in terms of types or subtypes of the given filter.



The Architecture of Publish-Subscribe Systems

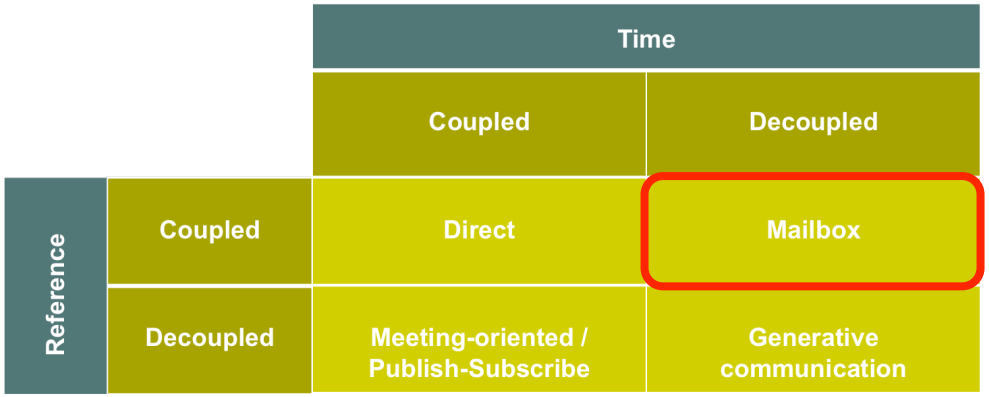


Example publish-subscribe systems

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

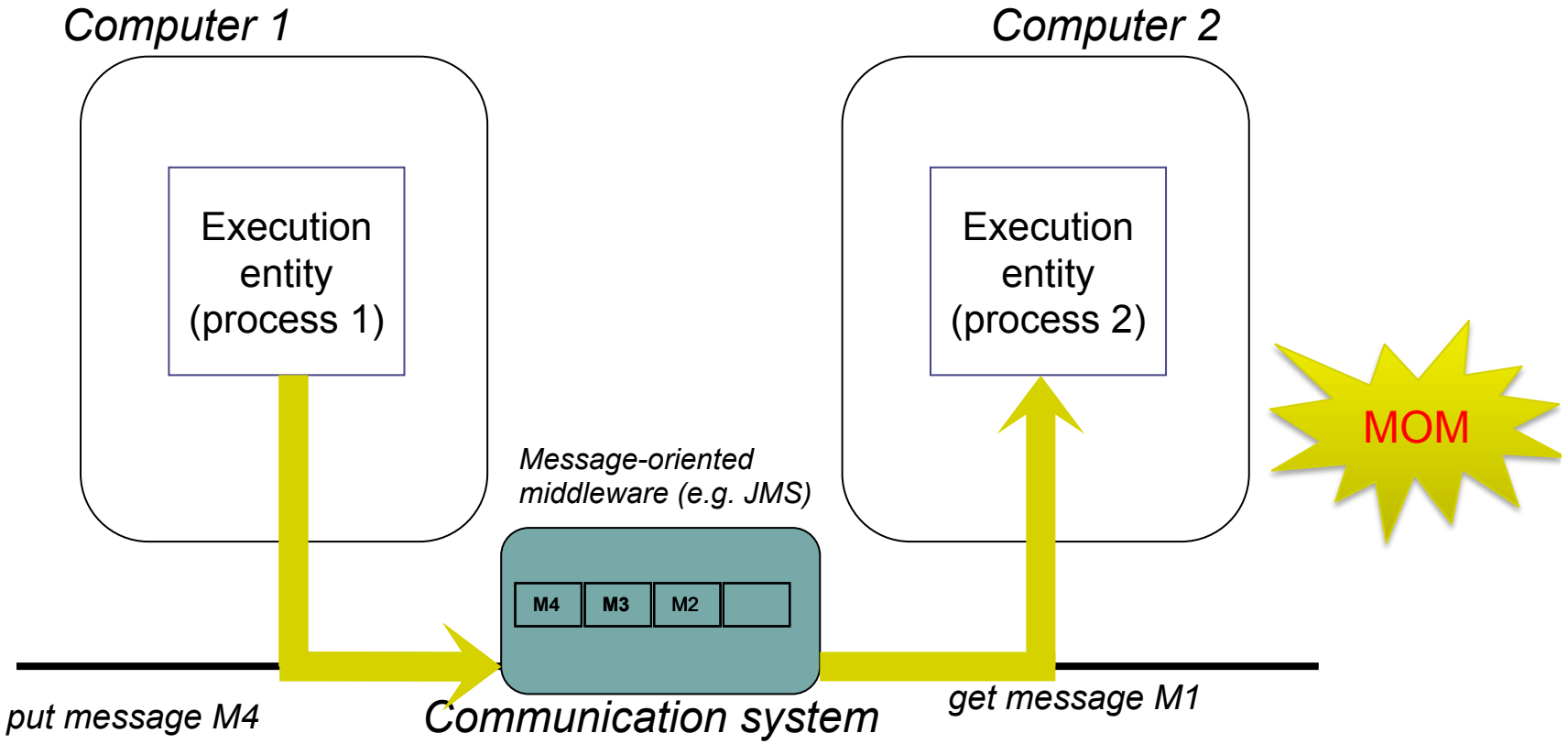
Indirect Communication Schemes

- Group Communication
- Publish-Subscribe
- **Message Queues**
- Distributed Shared Memory

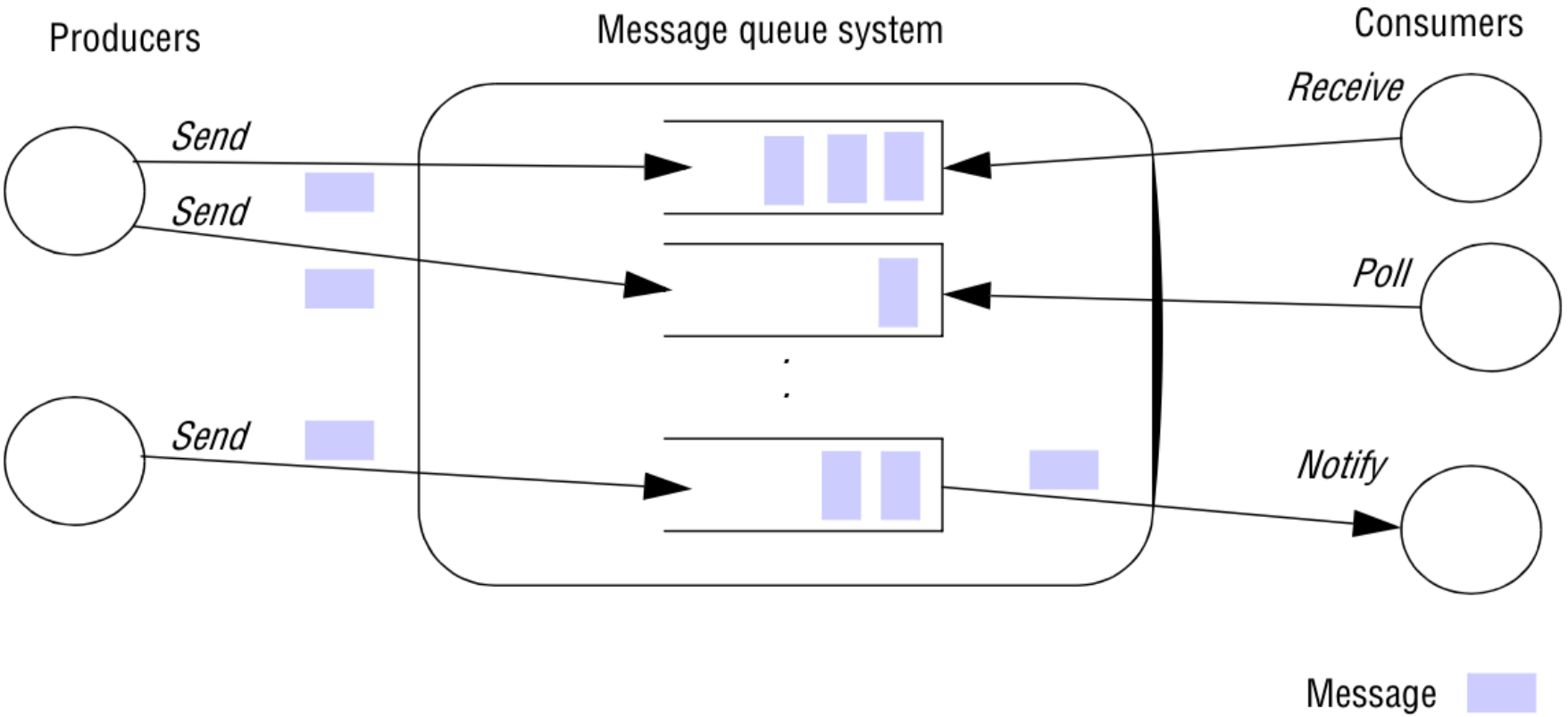


Messague Queue

- Message queues provide a *point-to-point* service using the concept of a message queue as an indirection, thus achieving the desired properties of space and time uncoupling



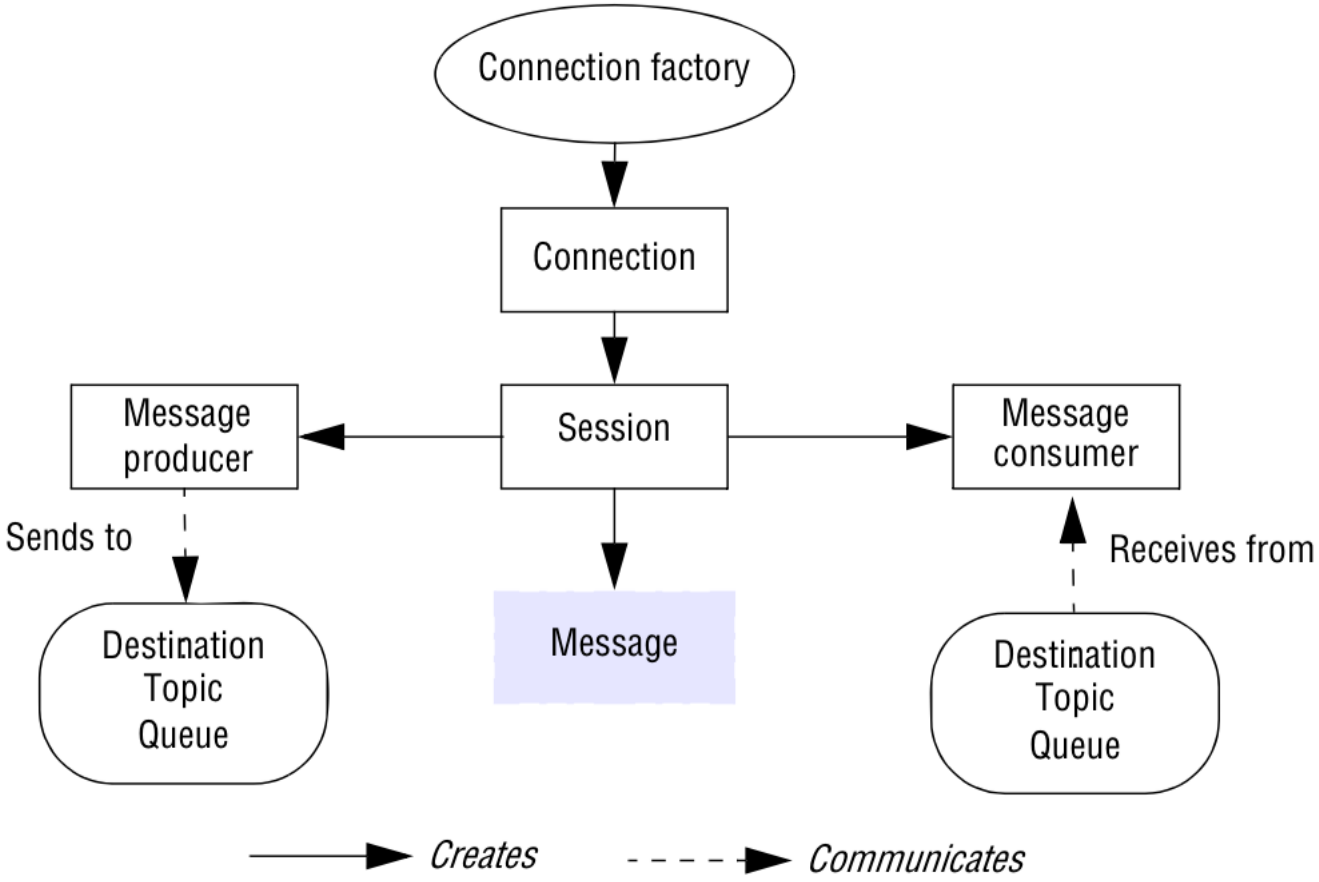
Primitives


Producers
Message queue system
Consumers
Send
Send
Send
Receive
Poll
Notify
Message

Persistency

- Messages are *persistent*
  - message queues will store the messages indefinitely (until they are consumed) and
  - will also commit the messages to disk to enable *reliable delivery*.
- Transactions
  - Most commercially available systems provide support for the sending or receiving of a message to be contained within a *transaction*.
- Transformation
  - An arbitrary transformation can be performed on an arriving message.
    - E.g. Handling heterogenous message formats

Example: Java Messaging Service



Example

Message Producer

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmJMS {

    public void raise() {
    try {                                                              1
        Context ctx = new InitialContext();                            2
        TopicConnectionFactory topicFactory =                          3
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory"); 4
        Topic topic = (Topic)ctx.lookup("Alarms");                     5
        TopicConnection topicConn =                                    6
            topicConnectionFactory.createTopicConnection();            7
        TopicSession topicSess = topicConn.createTopicSession(false,   8
                                        Session.AUTO_ACKNOWLEDGE);     9
        TopicPublisher topicPub = topicSess.createPublisher(topic);    10
        TextMessage msg = topicSess.createTextMessage();               11
        msg.setText("Fire!");                                          12
        topicPub.publish(message);                                     13
        } catch (Exception e) {                                        14
```

Example

Message Consumer

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmConsumerJMS {
    public String await() {
    try {                                                                    1
        Context ctx = new InitialContext();                                  2
        TopicConnectionFactory topicFactory =                                3
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");    4
        Topic topic = (Topic)ctx.lookup("Alarms");                           5
        TopicConnection topicConn =                                          6
            topicConnectionFactory.createTopicConnection();                  7
        TopicSession topicSess = topicConn.createTopicSession(false,         8
                                        Session.AUTO_ACKNOWLEDGE);           9
        TopicSubscriber topicSub = topicSess.createSubscriber(topic);        10
        topicSub.start();                                                    11
        TextMessage msg = (TextMessage) topicSub.receive();                  12
        return msg.getText();                                                13
        } catch (Exception e) {                                              14
            return null;                                                     15
```

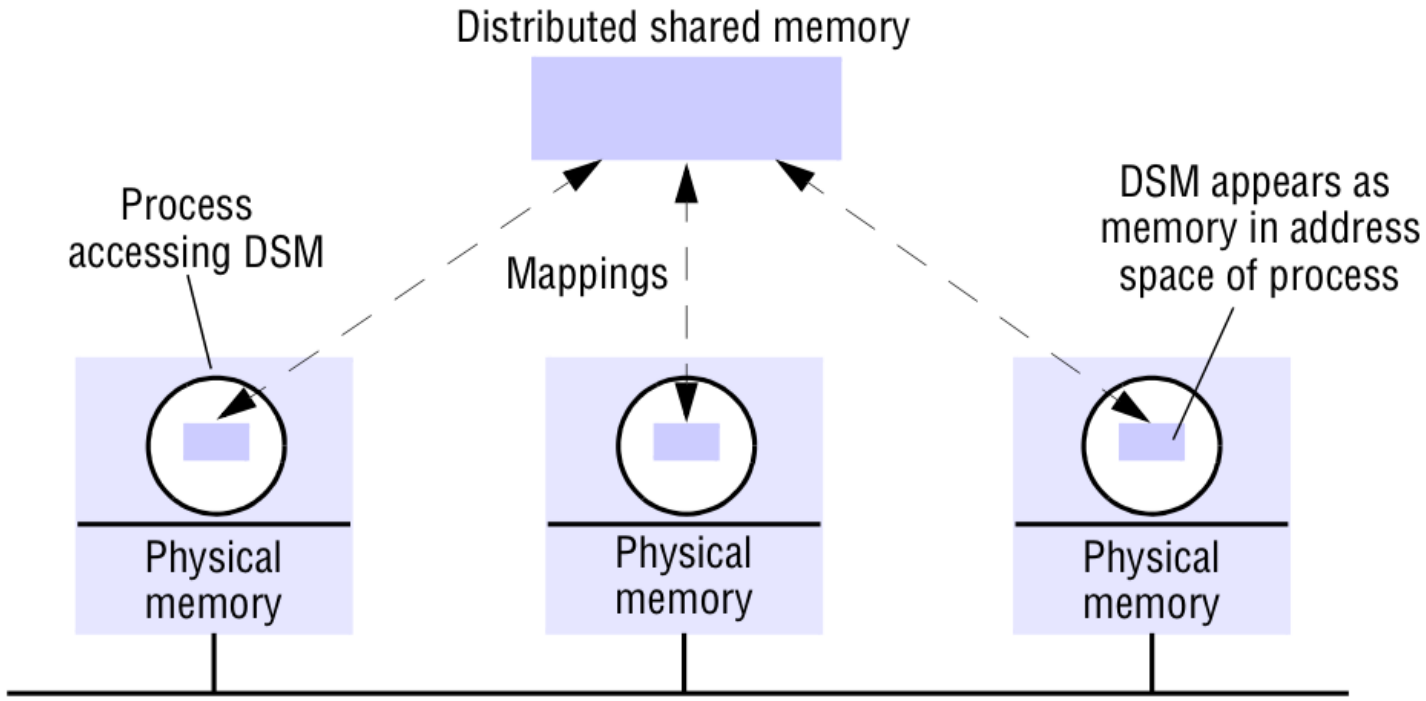
Indirect Communication Schemes

- Group Communication
- Publish-Subscribe
- Message Queues
- **Distributed Shared Memory**

		Time	
		Coupled	Decoupled
Reference	Coupled	Direct	Mailbox
	Decoupled	Meeting-oriented / Publish-Subscribe	Generative communication

Distributed shared memory

- Abstraction used for sharing data between computers that do not share physical memory

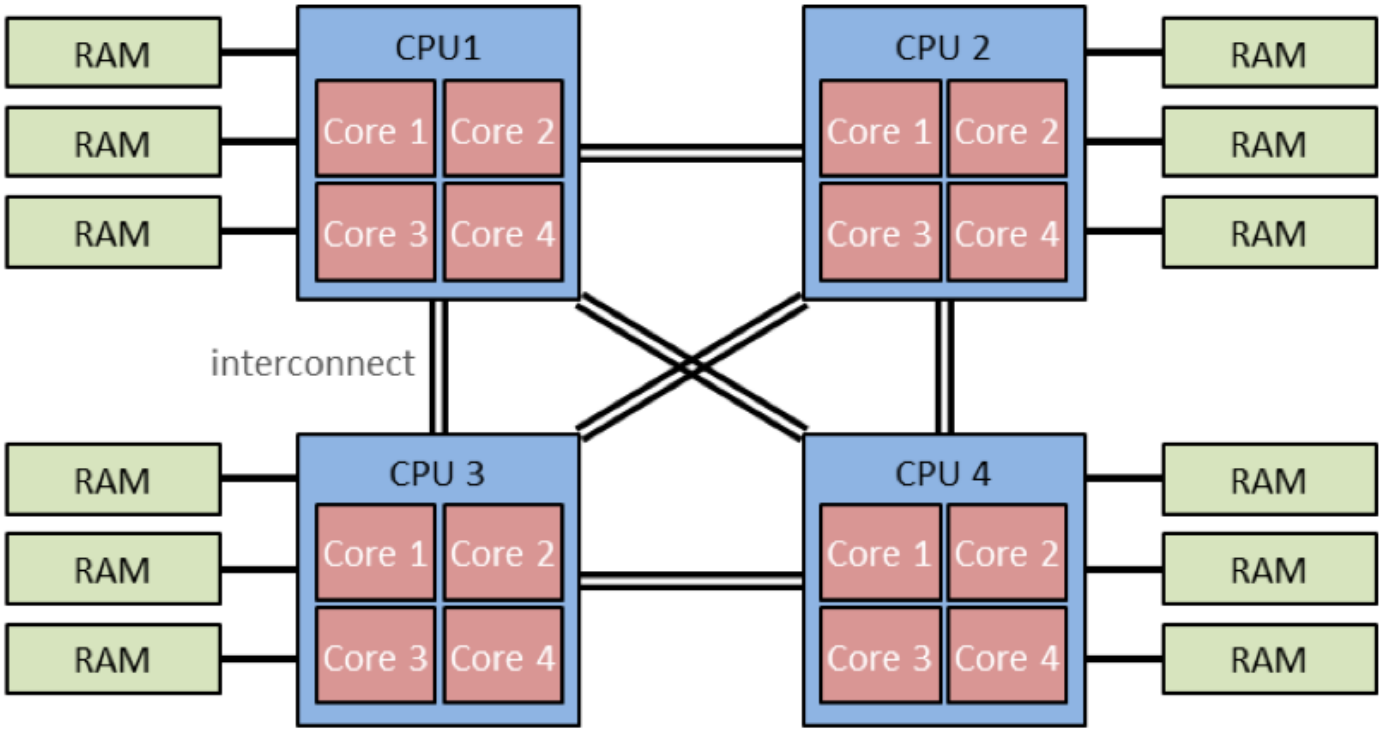


Principles

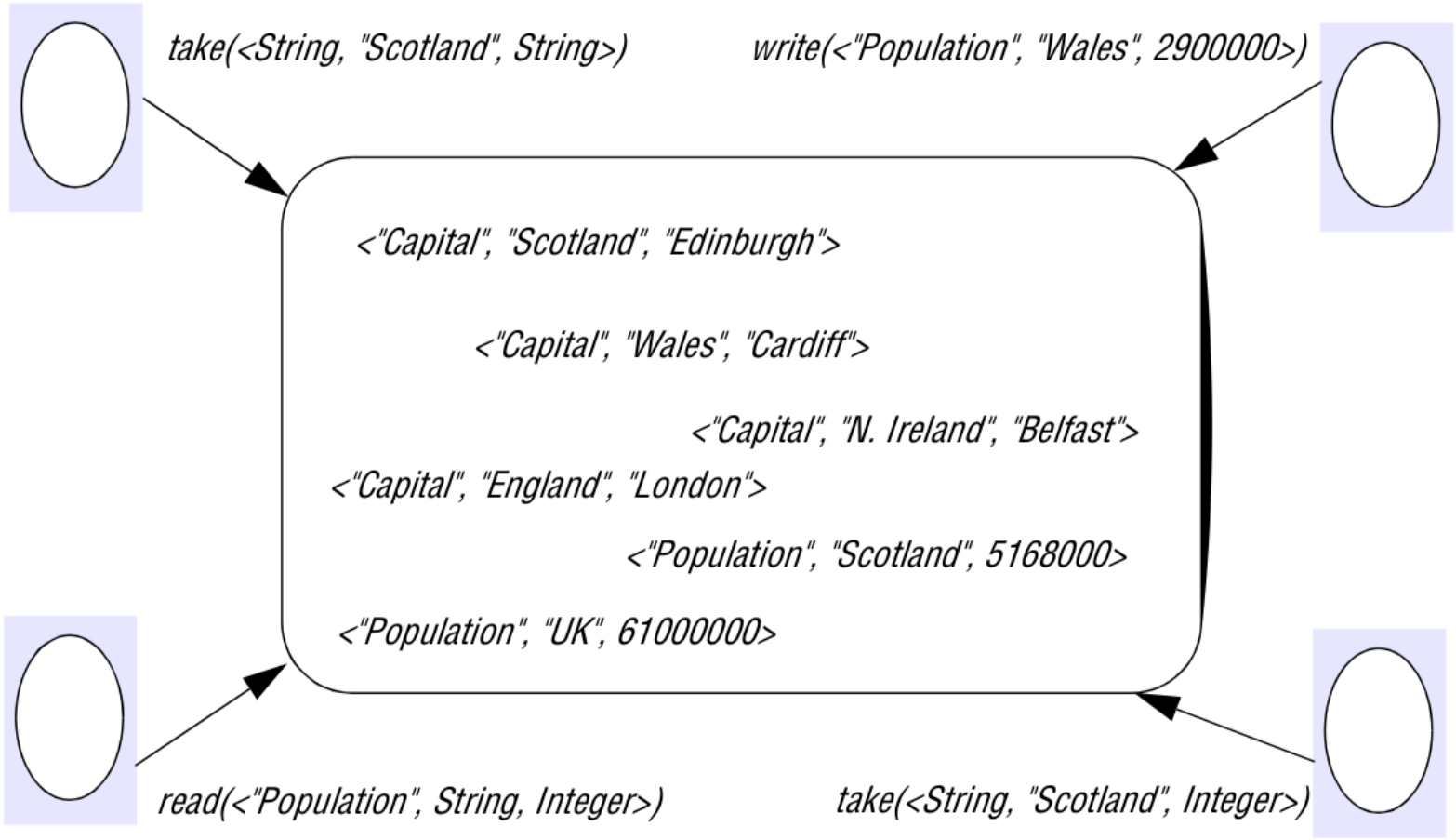
- It spares the programmer the concerns of message passing
- However, the DSM runtime support has to send updates in messages between computers
  - manage replicated data
    - each computer has a local copy (cache) of recently accessed data items stored in DSM, for speed of access

Application

- NUMA Architecture



Tuple-space Programming



Tuples are *immutable*

Case study: JavaSpaces

- Tool for tuple space communication
- Goals
  - Simplify the design of distributed applications and services
  - Small footprint to allow the code to run on resource-limited devices (such as smart phones)
  - Replicated implementations of the specification

Example

Tuple

Java class *AlarmTupleJS*

```
import net.jini.core.entry.*;

public class AlarmTupleJS implements Entry {
    public String alarmType;

    public AlarmTupleJS() {
    }

    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;
    }
}
```

Example cont.

Producer

Java class *FireAlarmJS*

```
import net.jini.space.JavaSpace;

public class FireAlarmJS {

    public void raise() {
        try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
        catch (Exception e) {
        }
    }
}
```

Example cont.

Consumer

Java class *FireAlarmReceiverJS*

```
import net.jini.space.JavaSpace;

public class FireAlarmConsumerJS {

    public String await() {
        try {
            JavaSpace space = SpaceAccessor.findSpace();
            AlarmTupleJS template = new AlarmTupleJS("Fire!");
            AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                                                Long.MAX_VALUE);
            return recvd.alarmType;
        }
        catch (Exception e) {
            return null;
        }
    }
}
```

Example Questions

- How would you implement a semaphore using a tuple space?
- Explain in which respects DSM is suitable or unsuitable for client-server systems.
- In publish-subscribe systems, explain how channel-based approaches can trivially be implemented using a group communication service?
- …

References

- This lecture is based on
  - George Coulouris, Jean Dollimore and Tim Kindberg, **Distributed Systems Concepts and Design**