DAD
Desenvolvimento de Aplicações Distribuídas

Message Queues

Agenda

- **Introduction**
- **Programming Model**
- **Case-study:**
  - Websphere MQ
  - Java Messaging Service

Introduction

- **Indirect Communication**
  - loose-coupling
  - queues are first-class entitites
  - no explicit end-point address of senders/receivers
- ***point-to-point* service**
  - not one-to-many as groups and pub-sub
- **used heavily in:**
  - Enterprise Application Integration (EAI) scenarios
  - *commercial transaction processing systems*

Programming Model

- **What is a Queue:**
  - communication end-point
  - contents:
    - ordered set of persistent messages
    - multiple producers and consumers
    - messages only consumed by one process

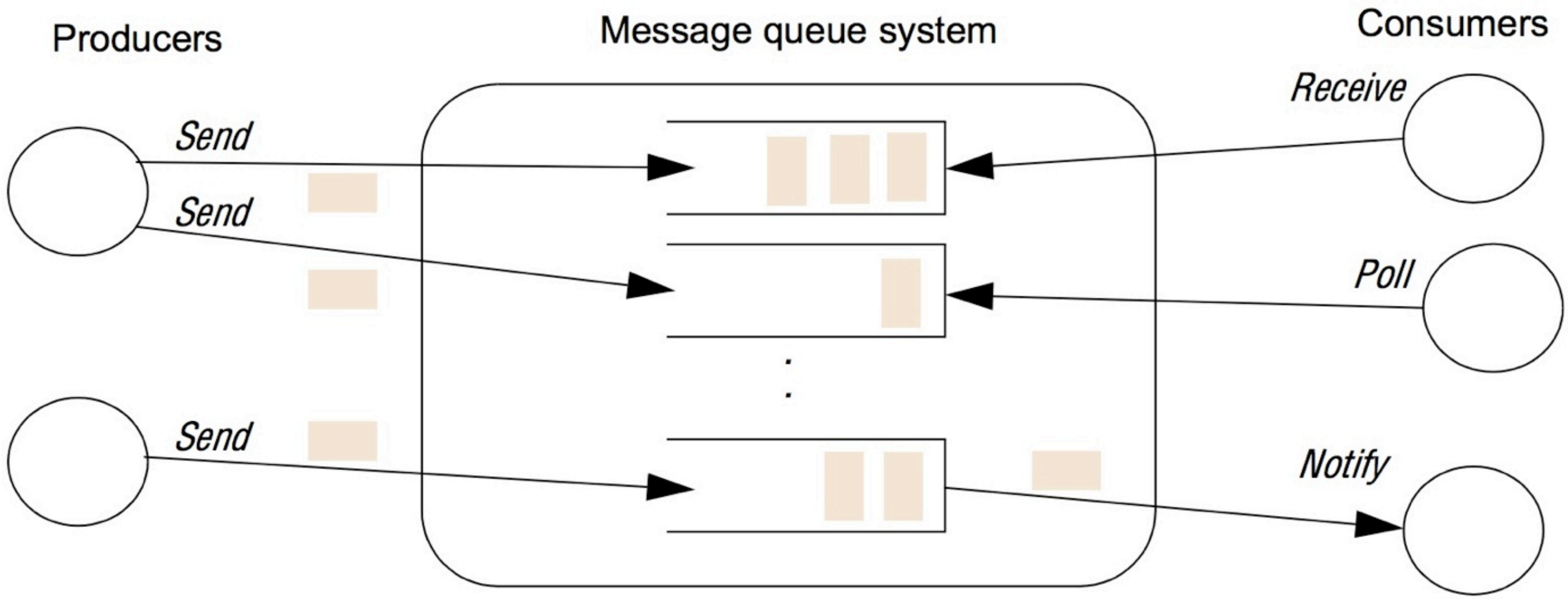
Programming Model

- **Operations on Queues:**
  - send (q, msg, ...)
  - msg = receive (q,...)
  - possibly integrated into transactions
    - atomicity of all operations and messages sent

- **Receive semantics:**
  - blocking (receive)
    - until appropriate message available
  - non-blocking (poll)
    - return message if available, otherwise return *not_available*
  - notify
    - raise event when message is available in queue

Programming Model

- **The message queue paradigm**



Programming Model

- **Queueing policy:**
  - order by which messages are consumed
    - FIFO, most common
    - priority-based
    - based on message properties
- **Message structure:**
  - destination (queue)
  - metadata (e.g., priority, delivery mode)
  - body (opaque to system)
    - can be a row or set of rows in a database

Properties

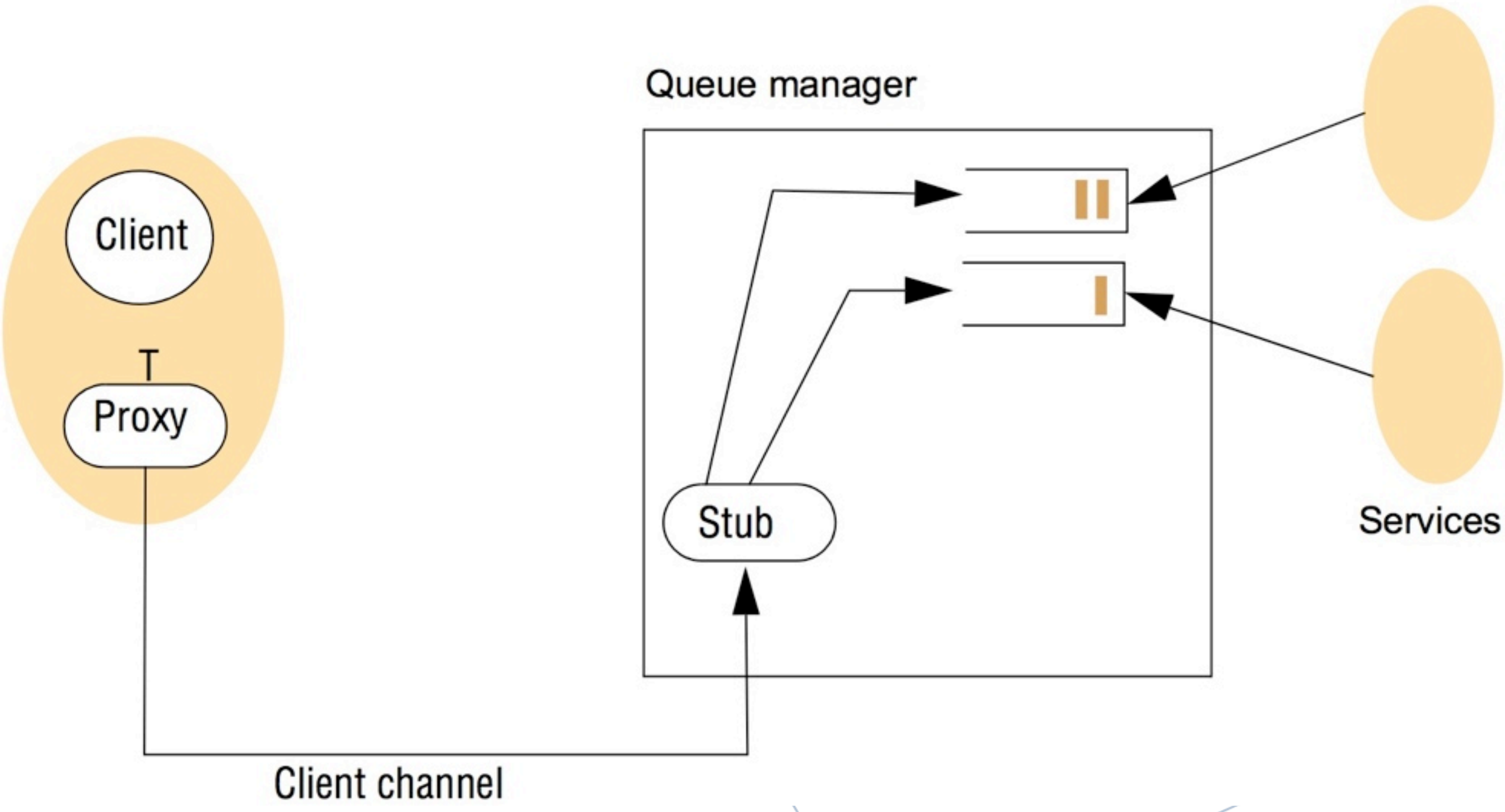
- **Reliable Delivery:**
  - messages are persistent
    - stored on disk indefinitely until consumed
    - including transactional support, journalling, logging, ...
- **Validity:**
  - any message sent is eventually received
- **Integrity:**
  - message received identical to the one sent
  - no message is delivered twice

Websphere MQ (IBM middleware)

- ***queue managers***
  - host and manage queues, enforce reliability
- **queue operations**
  - defined by a Message Queue Interface (MQI)
    - connect /disconnect (MQCONN / MQDISC)
    - send message (MQPUT)
    - receive message (MQGET)
- **channels:**
  - asynchronous *message channels* between *queue managers*
    - unidirectional, managed by 2 message channel agents (MCA)
  - client channels (senders/receivers to queue managers)

Websphere MQ

- **simple networked topology in Websphere MQ**



Websphere MQ

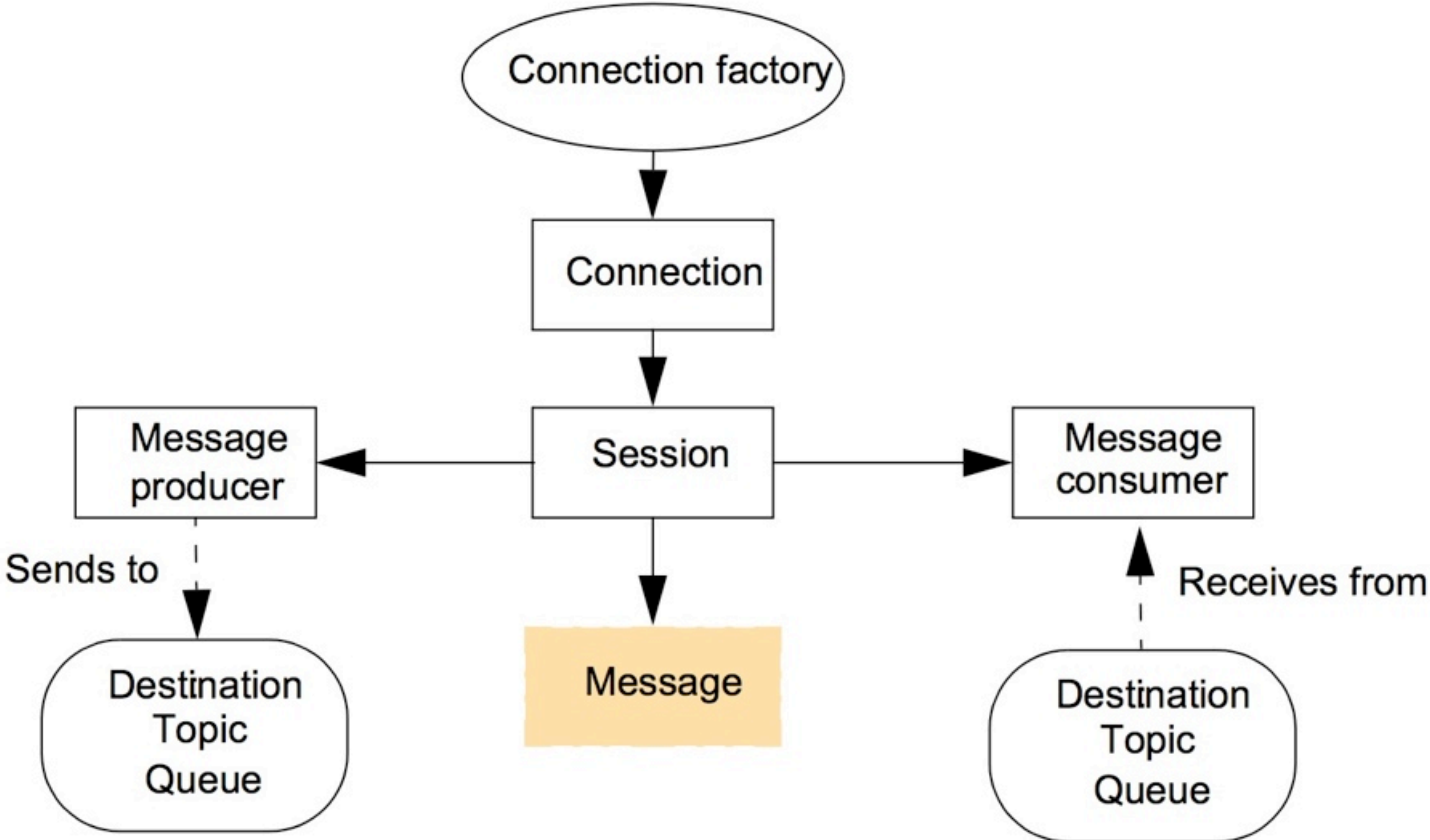
- ***Networks of Queue Managers:***
  - queue managers linked in federated structure
    - with message channels
    - routing tables at each queue manager
  - arbitrary topologies possible: trees, meshes, bus-based...
- ***Hub-and-Spoke Approach*:**
  - one QM is the hub
    - hosts most services, executed upon message reception
  - other, several QM are designated as spokes (*relays*)
    - placed around the network for geo-coverage and load-balancing
    - clients connect only to (near-by) spokes
  - widely used but hub potential bottleneck and single-failure
    - use *queue manager clusters* (for QM replication)

JMS: Java Messaging Service

- **Java standard specification**
  - indirect communication
  - attempt to partial unify pub-sub and message queues
  - many implementations;
    - Joram/OW2, JBoss, Sun OpenMQ, Apache ActiveMQ, OpenJMS
    - also Wepshere MQ.
- **Key roles:**
  - client (sender/receiver program)
  - provider (system that implements JMS, e.g., a server)
  - message (an object sent to the queue).
- **Programming model:**
  - Connections, Sessions, Topics, Queues, Transactions

JMS: Java Messaging Service

- **The programming model offered by JMS**



JMS: 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) {
        }
}
```

JMS: class *FireAlarmConsumerJMS*

```
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;
    }
}
```

Summary

- **Introduction**
- **Programming Model**
- **Case-study:**
  - Websphere MQ
  - Java Messaging Service

DAD

Desenvolvimento de Aplicações Distribuídas

Shared Memory Approaches

Agenda

- **Introduction**
- **Distributed Shared Memory**
- **Tuple-space Communication**
  - other approaches: replication, partitioning
- **Case-study:**
  - JavaSpaces
- **Indirect Communication Summary**

Introduction

- **previous Indirect Communication based on *message-passing***
  - *group communication*
  - *publish-subscribe*
  - *message queues*

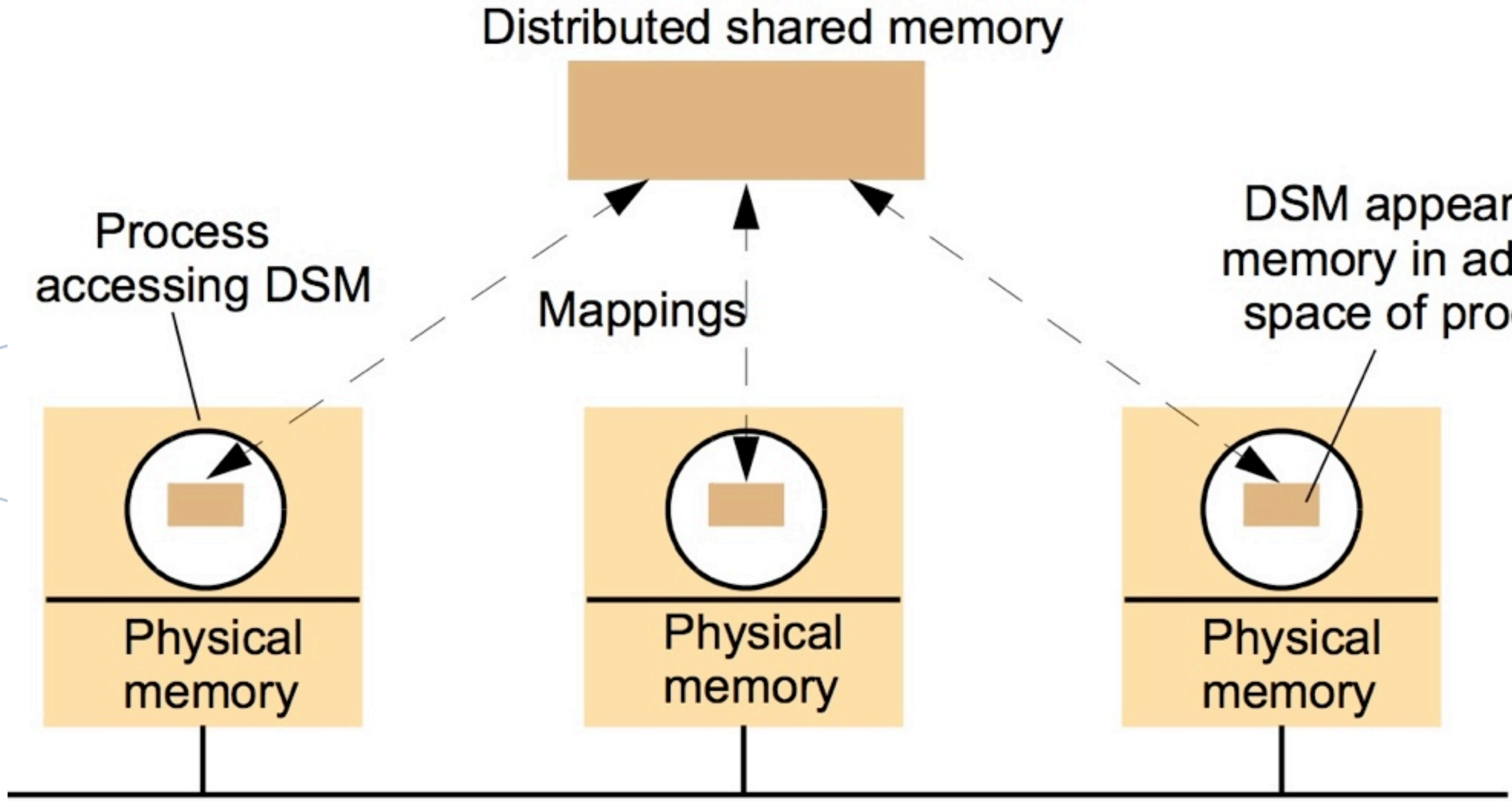
- **Shared Memory approaches**
  - abstraction of shared address/data space
  - accessed with reading and writing operations
  - located by *memory address* or by *content (associative)*

Distributed Shared Memory (DSM)

- **abstraction for sharing data**
  - between computers with separate physical memory
  - shared memory regions
    - mapped in each process as *ordinary* memory in address space
  - data accessed/modified by read/write CPU instructions
  - supported transparently by runtime system

- **DSM processes manipulate shared logical global address space**
  - while in fact physical memory is distributed

Distributed Shared Memory



Distributed Shared Memory (DSM)

- **Main Goal**
  - free programmers from concerns with message-passing
  - common use in parallel and distributed applications
    - acessing shared data items individually and directly (i.e., variables, objects, arrays, ...)
    - easier to express parallel algorithms
    - not appropriate for client-server, lacks modularity and protection
- **Implementation Issues**
  - message passing cannot be avoided in distribution
  - DSM runtime has to exchange updates among processes
  - DSM systems manage local copies of replicated data for access speed
  - evolution: shared-memory multiprocessors, non-uniform memory access (NUMA), distributed-memory multiprocessors, clusters

Tuple Space Communication

- **abstraction for sharing data**
  - unlike DSM, no direct addressing,
  - data located by pattern matching on content
    - early form of content-addressable memory
  - organized as *tuples* (akin to database records)
  - tuples retrieved and created, never modified
    - makes synchronization easier/unnecessary

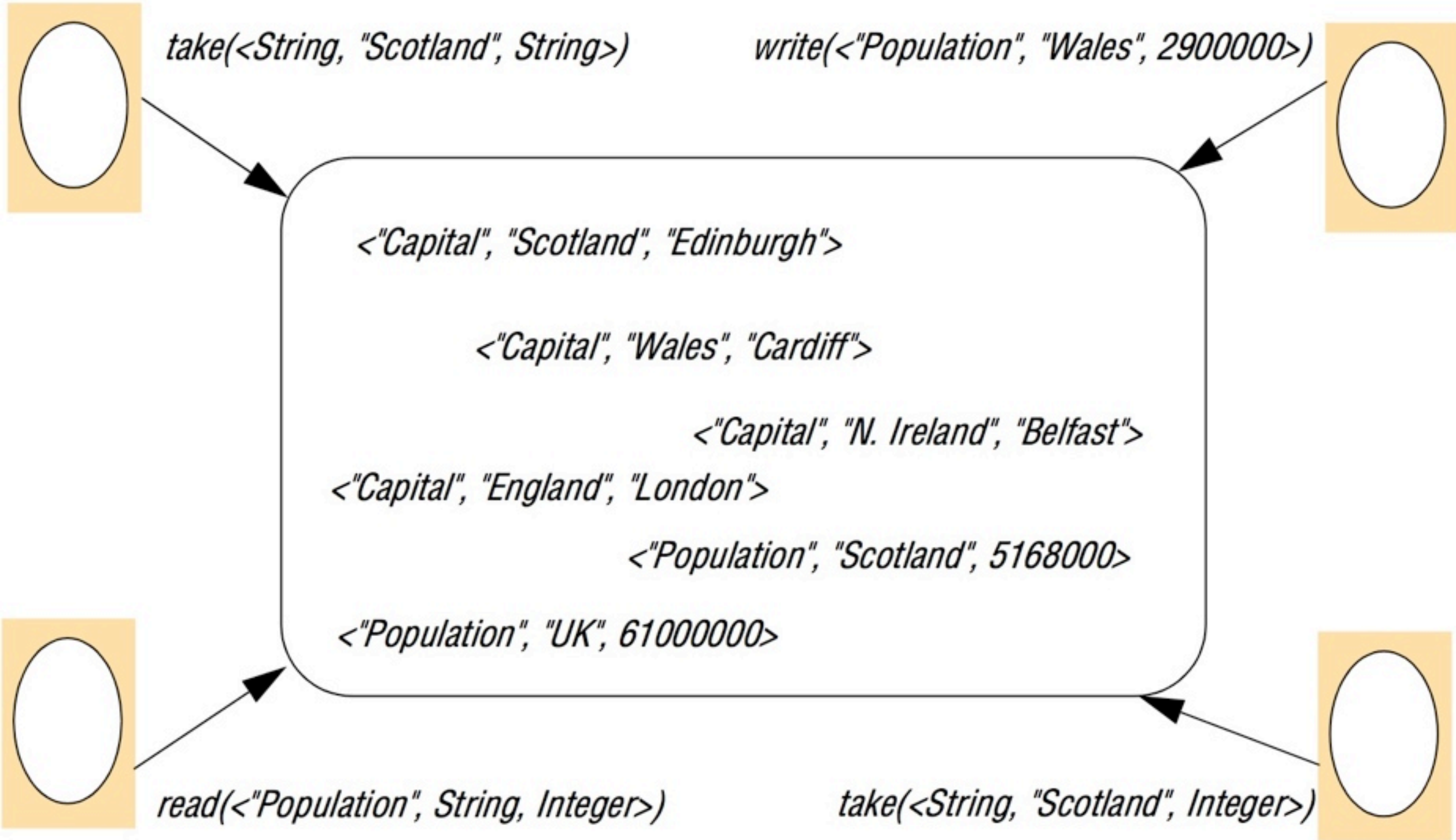
- **relevant examples**
  - Linda, IBM Tspaces, Sun JavaSpaces

Tuple Space Communication

- **Programming Model**
  - data space
    - shared collection of tuples (sequence of typed data fields):
      - e.g., <“fred”, 1958>, <“sid”, 1964>, <4, 9.8, “yes”>
    - processes shared data by accessing the same tuple space
  - operations
    - **write (out):** creates and adds tuples to the tuple space
      - without affecting any other tuple
    - **read (rd):** retrieves content of one tuple
      - without affecting any other tuple, tuple may be read several times
    - **take (in):** retrieves one tuple and removes it from the tuple space
      - each tuple is *consumed* by only one *take* operation
  - tuple specification (e.g., <*String*, “Scotland”, *String*>
    - *template* to find *associatively,* any matching tuple in the space

Tuple Space Communication

- Programming model and examples of tuple operations



Tuple Space Communication

- **Synchronization**
  - read and take operations are blocking
    - until there is matching tuple in the space
  - no direct access to tuples in the tuple space
    - processes have to replace tuples instead of modifying them
  - e.g.,
    - shared counter <“counter”, 64> manipulated by several processes
    - to increment counter, a process performs
    - <s, count> := myTS.take(<“counter”, integer>);
    - myTS. write(<“counter”, count+1>);

Tuple Space Communication

- **Important Properties:**
  - space uncoupling
    - tuple in space may originate from any process and may be delivered to any number of processes
  - time uncoupling
    - tuple placed in the tuple space will remain until removed (potentially undefinitely), hence sender and receiver need not overlap in time

- **Variations**
  - multiple tuple spaces, per-user, per-application, system
  - distributed implementations (original ones were centralized)
  - objects (methods, attributes) stored as tuple fields

Tuple Space (Other Approaches)

- **Replicated Tuple Spaces**
  - goal: fault-tolerance and scalability
- **State-machine approach**
  - assume tuple space behaves as state-machine
  - to ensure consistency, replicas must:
    - 1. begin with the same data (empty tuple space)
    - 2. execute events in the same order
    - 3. react deterministically to each event
  - property (2) can be ensured by total order multicast (coming soon!)

Tuple Space (Other Approaches)

- **Replicated Tuple Spaces**
  - goal: fault-tolerance and scalability
- **Xu and Liskov**
  - optimizes replication strategy, and uses partitioning
  - leverages semantics of particular tuple space operations
  - processes:
    - set of workers that perform computation on the tuple space
    - set of tuple space replicas
    - worker and replica may be combined in a single physical node
  - network assumed to fail:
    - may lose, duplicate, delay messages or delivered them out of order
  - operations executed in the context of current *view*
  - ***view:*** agreed set of replicas of the tuple space

Replicated Tuple Spaces: Xu and Liskov (operation algorithms)

- **Write:**
  - 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

Replicated Tuple Spaces: Xu and Liskov (operation algorithms)

- **Read**
  - 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.

Replicated Tuple Spaces: Xu and Liskov (operation algorithms)

- **Take (*Phase 1: Selecting the tuple to be removed)***
  - **1.**Requesting site multicasts the take request to all members of 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.

Replicated Tuple Spaces: Xu and Liskov (operation algorithms)

- **Take *(Phase2: 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.

Replicated Tuple Spaces: Xu and Liskov (operation latency)

- Minimize delay given the **semantics** of the three tuple space operations:
  - read operations only block until the first replica responds to the request
  - write operations can return immediately.
  - take operations block until the end of phase 1, when the tuple to be deleted has been agreed.

Replicated Tuple Spaces: Xu and Liskov

- **Issues with concurrency:**
  - e.g., **read** operation may access tuple that should have been deleted in the 2nd phase of **take** operation
- **Solution: enforce additional constrains**
  - operations of each worker must be executed at each replica in the same order they were issued by worker
    - FIFO order (or client order)

Replicated Tuple Spaces: Xu and Liskov

- **Issues with concurrency:**
  - e.g., **read** operation may access tuple that should have been deleted in the 2nd phase of **take** operation
- **Solution: enforce additional constrains**
  - operations of each worker must be executed at each replica in the same order they were issued by worker
    - FIFO order (or client order)
  - a **write** operation must not be executed by any replica until
    - all previous **take** operations issued by the same worker have completed at all replicas in the worker's *view.*

Partitioned Tuple Spaces

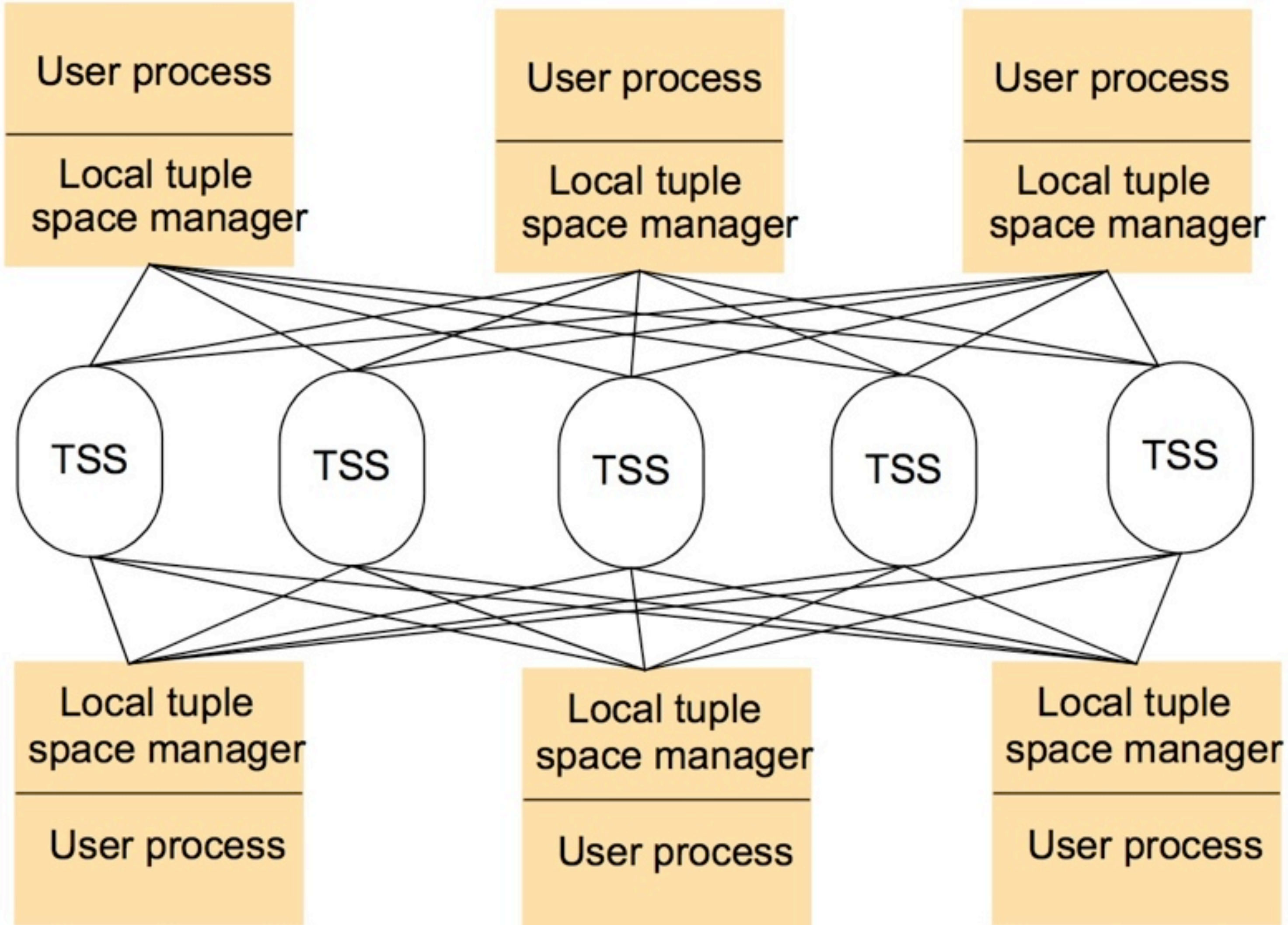
- **Partitioned Tuple Spaces**
  - Linda kernel at Univ. York
  - tuples partitioned across Tuple Space Servers (TSS)
  - no replication, sole goal to increase performance

  - tuple specifications specify type or values
  - tuple location decided by hashing algorithm
    - generates set of possible servers that may contain matching tuples
    - local linear searching

  - e.g, of peer-to-peer implementation: PeerSpaces

Partitioned Tuple Spaces

- **Architecture of Linda Kernel Univ. York**



JavaSpaces (Jini / Apache River)

Programming Model

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 example

```
import net.jini.core.entry.*;
public class AlarmTupleJS implements Entry {
    public String alarmType;
        public AlarmTupleJS() { }
    }
    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;}
    }
}
```

JavaSpaces example

```
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) {
    }
   }
}
```

JavaSpaces example

```
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;
            }
      }
}
```

Indirect Communication (Summary)

	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

Summary

- **Introduction**
- **Distributed Shared Memory**
- **Tuple-space Communication**
- **Case-study:**
  - JavaSpaces
- **Indirect Communication Summary**