# Models of Communication

# CO-OPERATIVE SYSTEMS

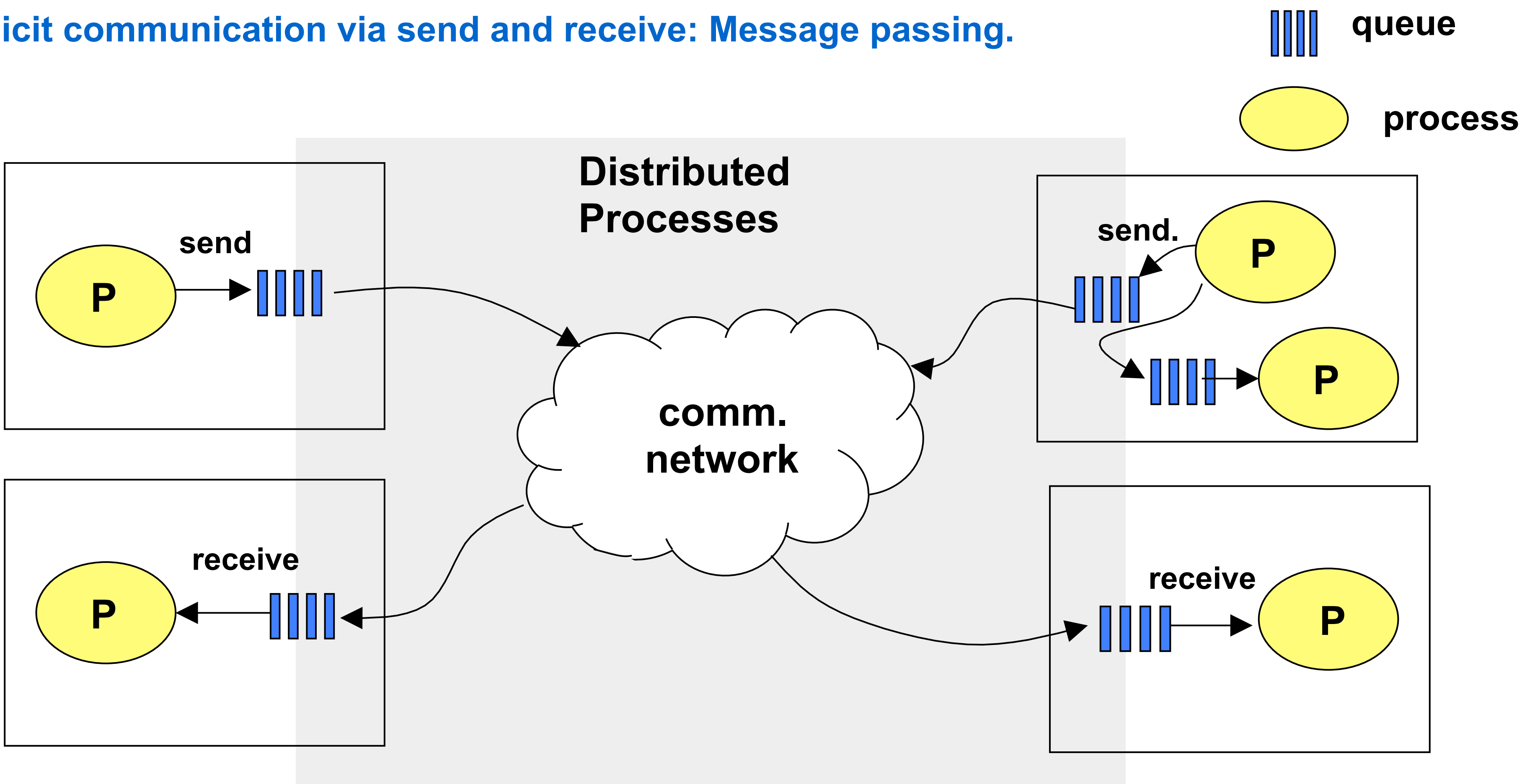
**Which model of communication?**

**What kind of addressing and routing should be supported by the network?**

**Which abstractions in the programming model?**

# Message Passing

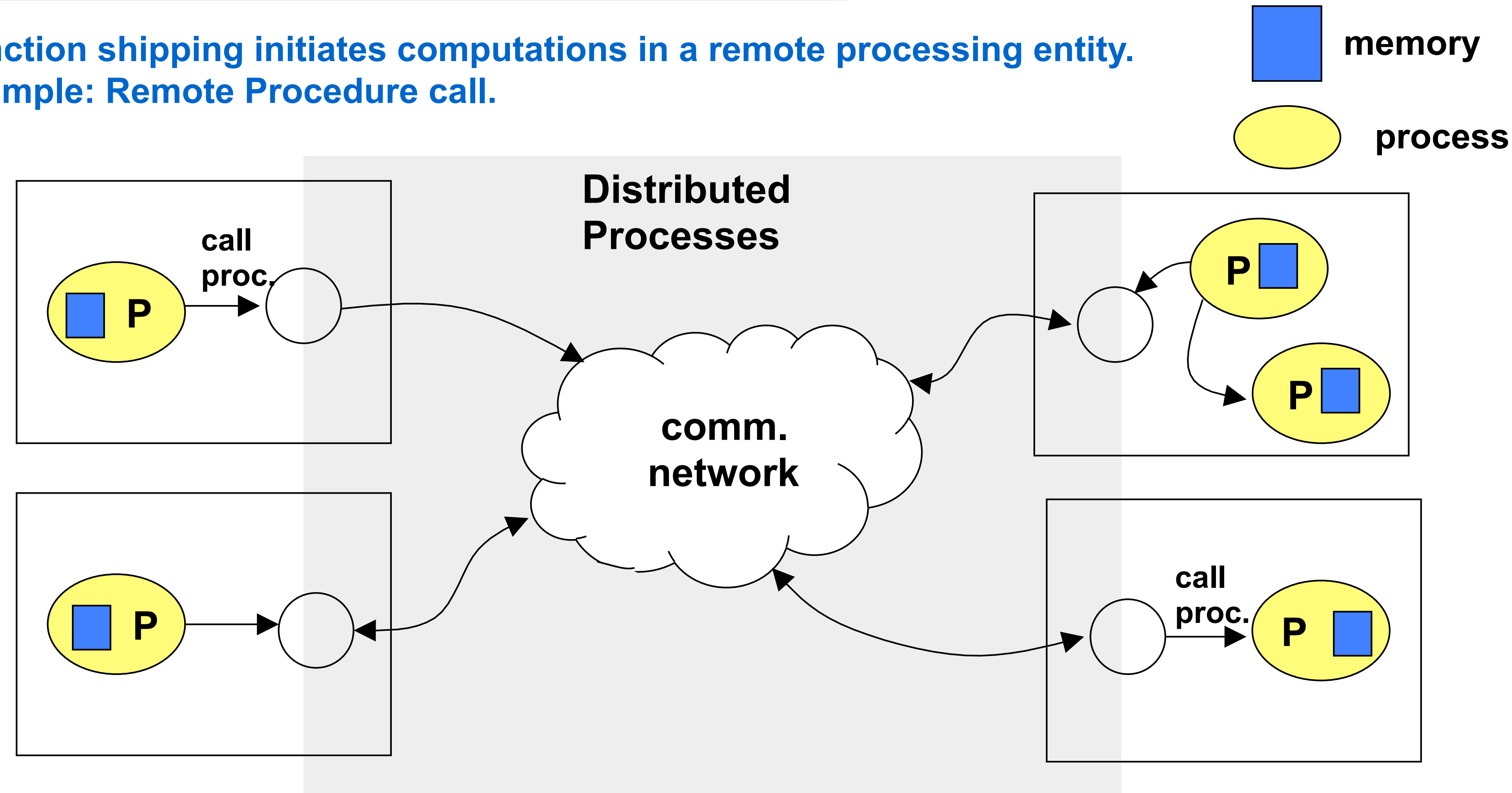
Explicit communication via send and receive: Message passing.




Problem: very low level, very general, poorly defined semantics of communication

# Remote Procedure Call

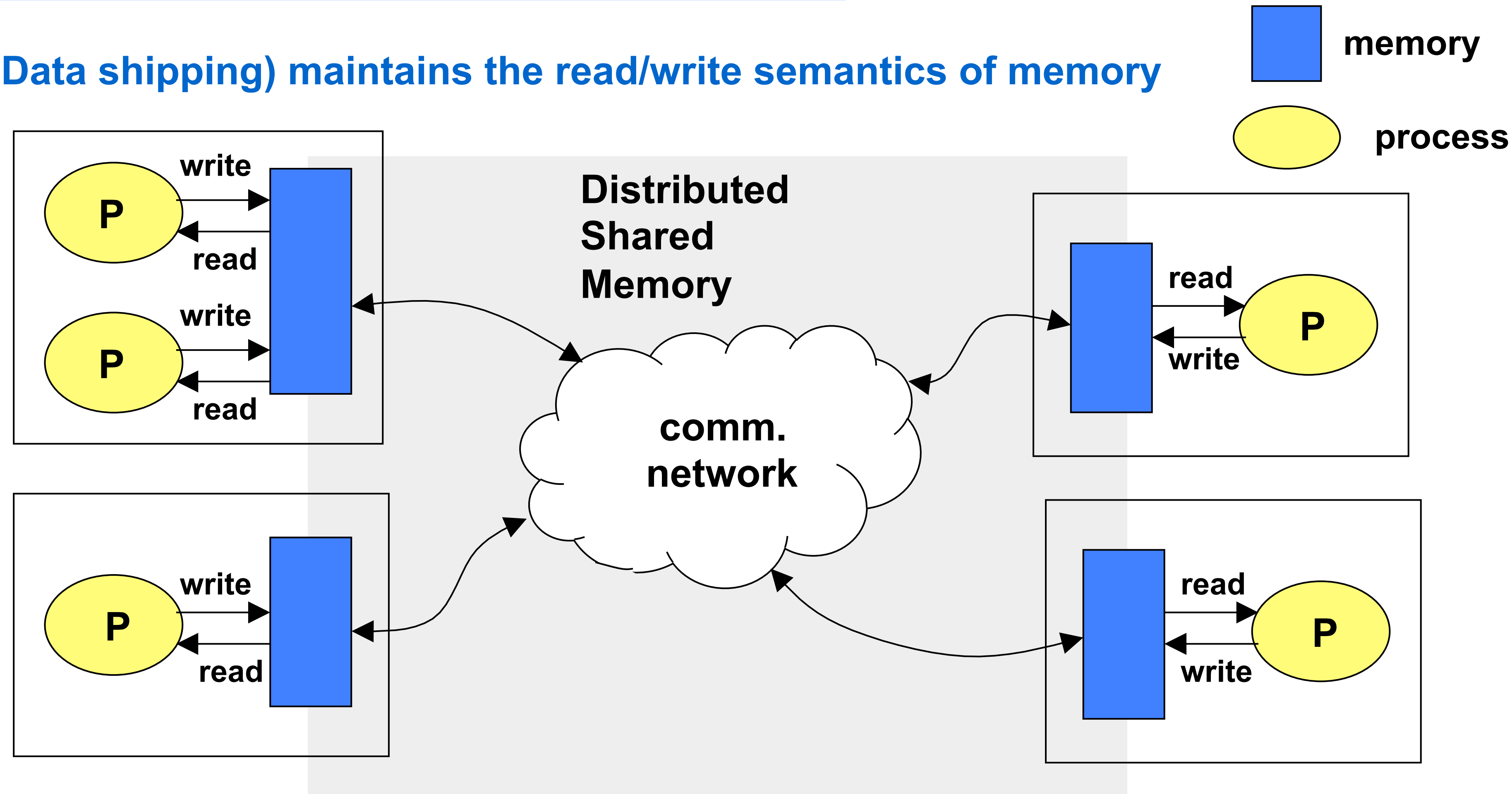
**Function shipping initiates computations in a remote processing entity.**
**Example: Remote Procedure call.**




**Problem: computation bottlenecks, fault semantics, references.**

# Distributed Shared memory

**DSM (Data shipping) maintains the read/write semantics of memory**




**Problem: Consistency in the presence of concurrency and communcation delays**

# Abstractions for Communication

- Message passing
- Remote Procedure Call
- Remote Object Invocation
- Distributed shared memory
- Notifications
- Publish Subscribe
- Shared data spaces

# Abstractions for Communication

## Dimensions of Dependencies:

### Space Coupling: References must be known

Explicit specification of the destination, i.e. producer must know where to send the message. Message contains an ID specifying an address or name.

### Flow coupling: Control transfer with communication

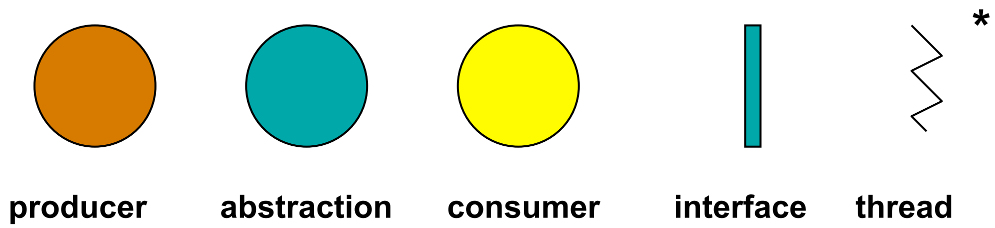
Defines whether there is a control transfer coupled with a message transfer. E.g. if the sender blocks until a message is correctly received.

### Coupling in time: Both sides must be active

Communication can only take place if all partners are up and active.

# Message passing

## Connected socket, e.g. TCP

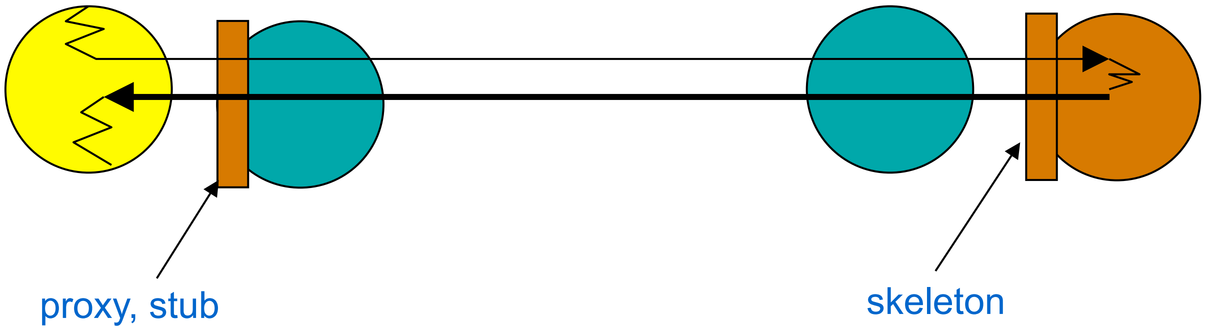




**primitives: send (), receive ()**

**Coupling: space, time**

* Notation acc. P. Eugster: Type-Based Publish Subscribe, PhD-thesis, EPFL, Nr. 2503, 2001

# Remote Procedure Call (RPC)




Relation: one-to-one

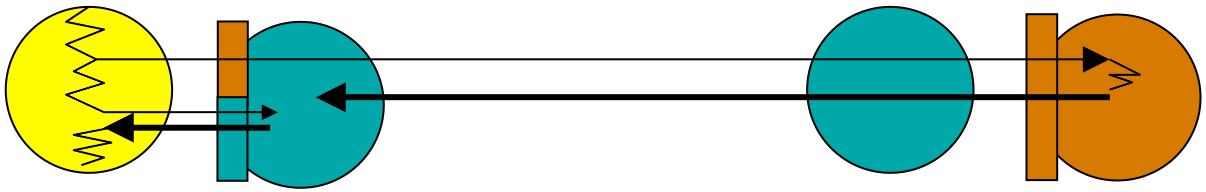
Coupling:

Space: destination is explicitely specified

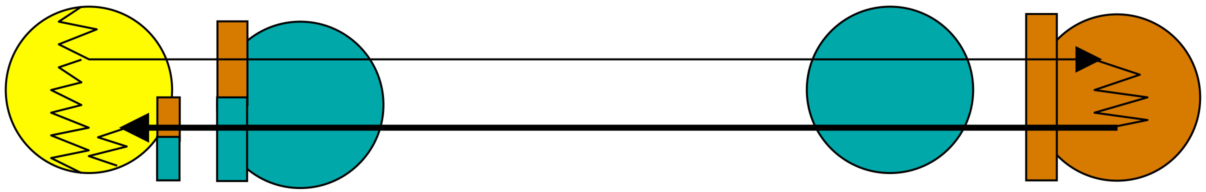
Flow: blocks until message is delivered

Time: both sides must be active

# Variations of RPC



Asynchronous RPC with pull



Asynchronous RPC with call-back

Example: Concurrent Smalltalk

Relation: one-to-one

Coupling:

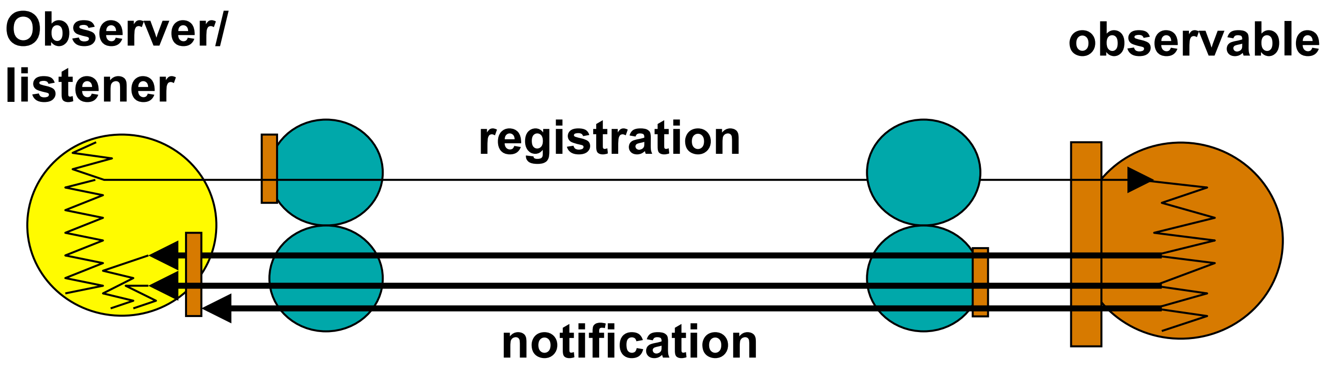
Space: destination is explicitely specified

Flow: no flow coupling

Time: both sides must be active

Example: Eiffel

# Notification



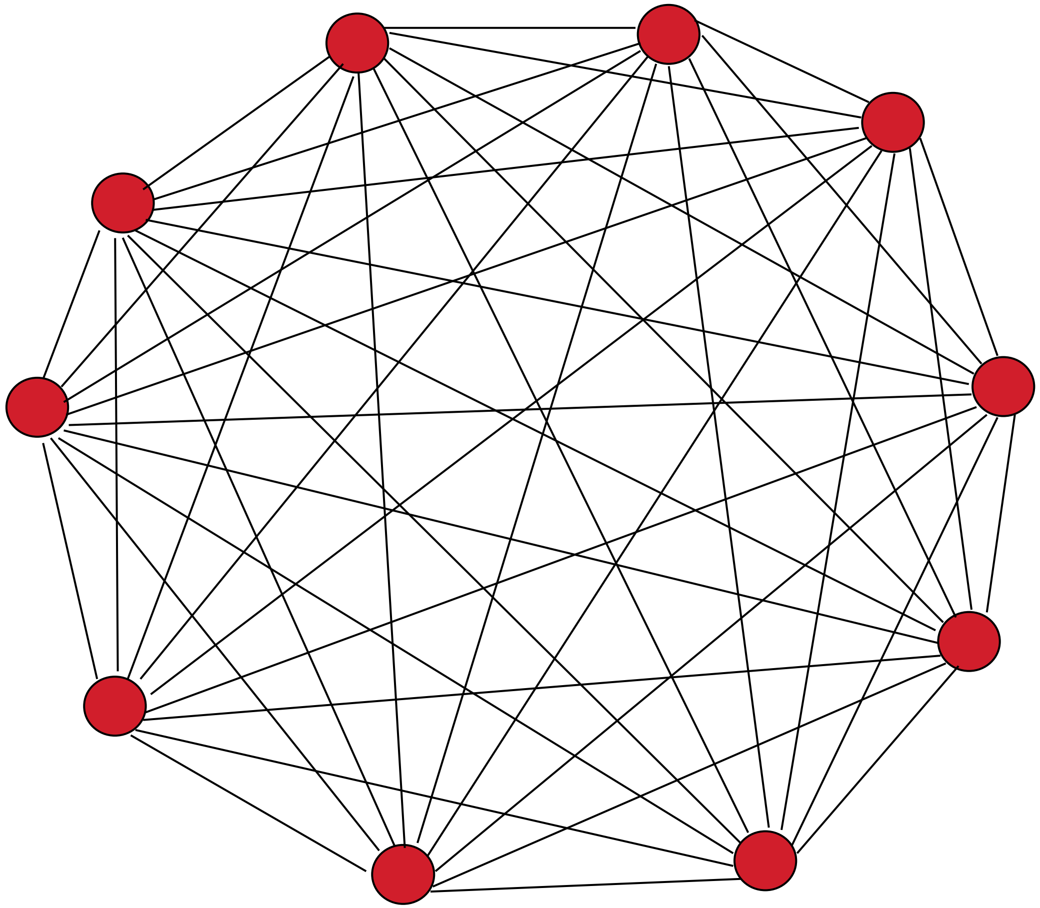

Examles:
Java

Relation: one-to-many

Coupling:
Space: Yes (Observable/Observer pattern (delegation))
Flow: none
Time: both sides must be active (notification performed by RMI)

# Interaction Structure in Co-operative Systems

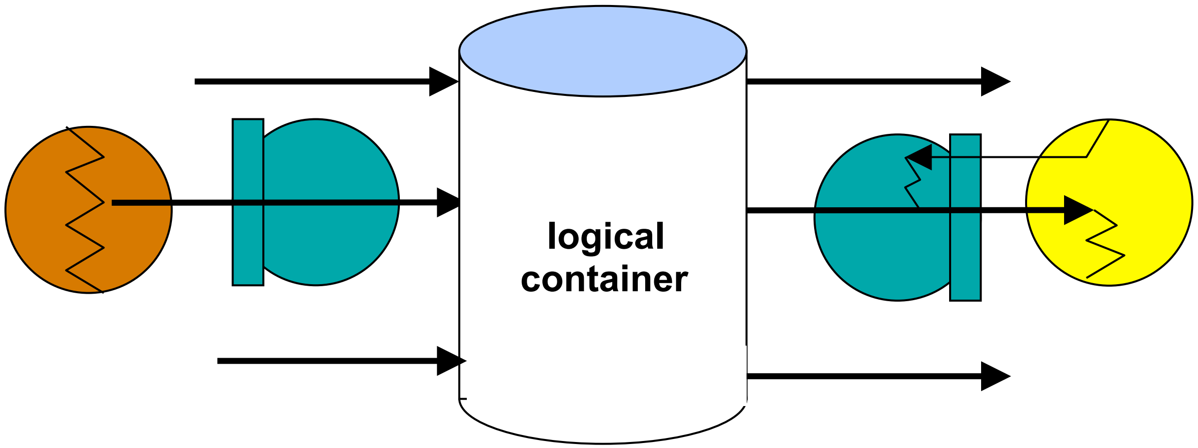
many-
to-
many



**managebility**

**goal** ➔ **sharing information and co-ordinating acitvities**

# Shared Data Spaces




Relation: many-to-many

Coupling:
Space: none
Flow: none
Time: none

Examples:
Linda Tuple Space
Java Spaces
ADS Data field

# Shared Data Spaces

Processes communicate via the "Tuple" Space,
A tuple is only data, no address, no identifier,
A tuple is a data structure similar to a struct in C,

Examples: ("3numbers", 3, 6, 7), ("matrix" , 1, 5, 3.23, 8),
("faculty", "is_member_of", "franz", "maria", "otto")

Primitives (operations) in LInda:

- op. in: takes (and removes) an element from the tuple space
- op. read: reads an element from the tuple space
- op. out: puts a tuple into the tuple space
- op. eval: allows to eveluate the fields of a tuple, results are put in the tuple space [example: ("product", mult(4,7))]

No Tuple is ever (over-) written! "out" always put a new item in the space.

# Shared Data Spaces

**Content-Based Addressing by Tuple matching:**

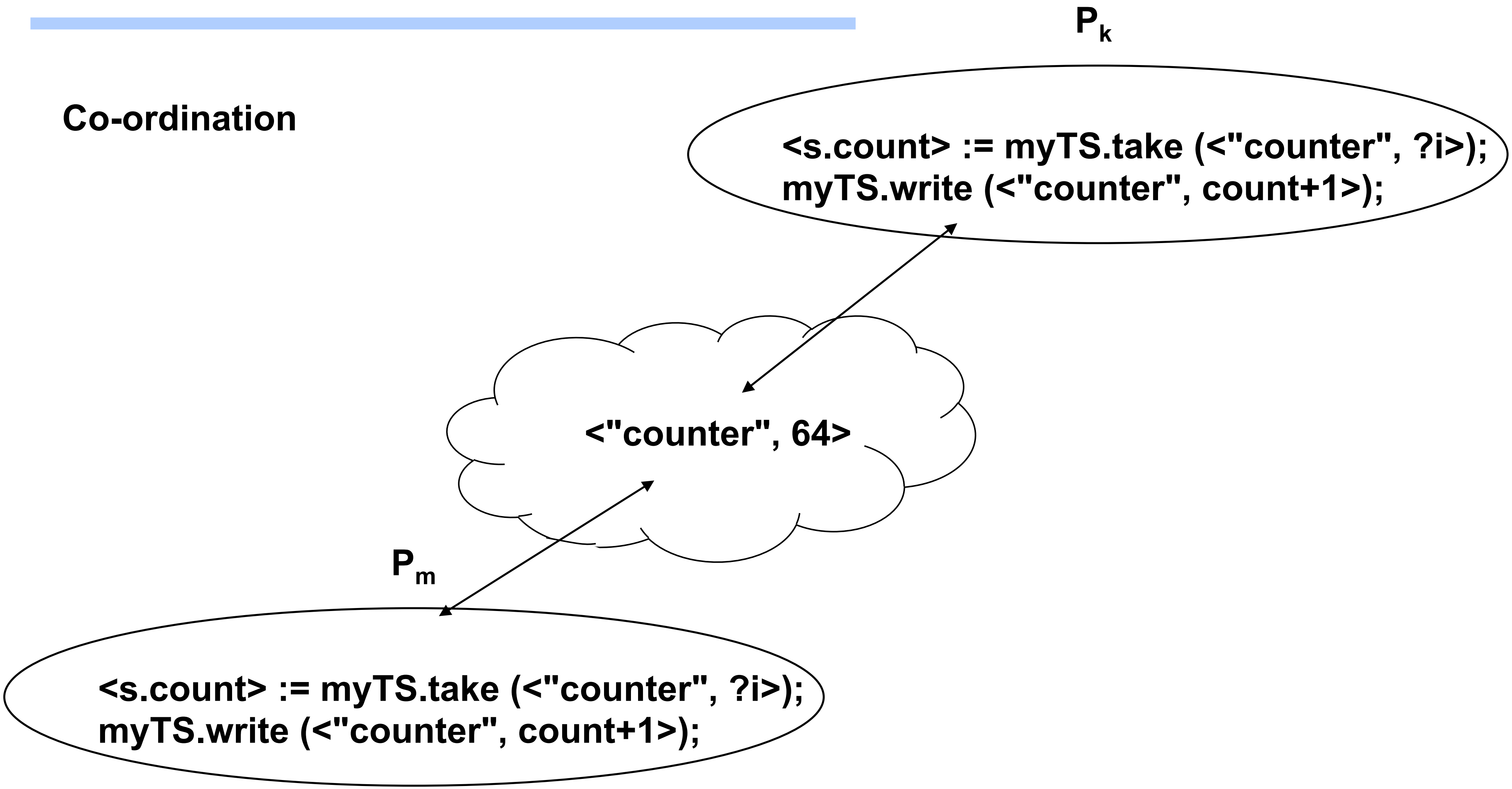
**All fields in a template are compared to all tuples.**
**A match of a template occurs if:**
**tuple has the same number of fields**
**AND types of fileds are equivalent**
**AND contents corresponds**

**Example:**
**<"distance´_sensor", "N", 23>**
**<"distance´_sensor", "E", 127>**
**<"distance´_sensor", "S", 127>**
**<"distance´_sensor", "W", 12>**

**in(<"distance_sensor", " ", ?i> : reads all distance sensors and removes their values from the space.**
**read(<"distance_sensor", S, ?i>: subsequent read blocks until new S-value has been put to the Space.**

# Shared Data Spaces

**Co-ordination**

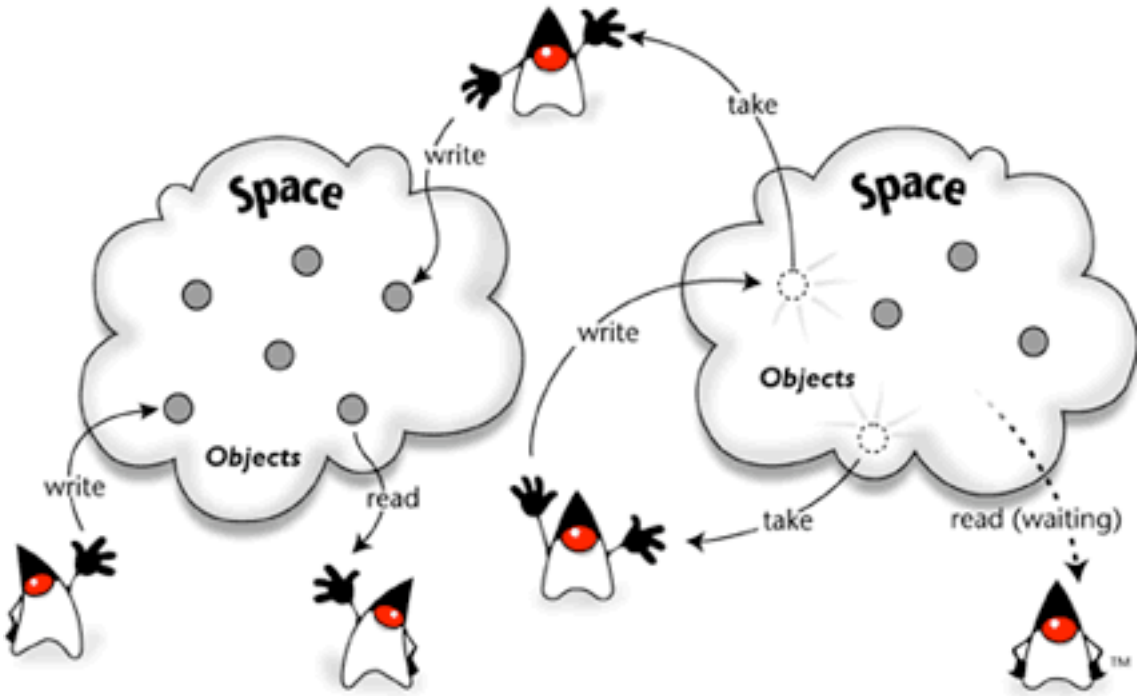


# Shared Data Spaces

## Immutable Data Storage:

- no write operation!
- "out" always adds a data element to the storage
- destructive "in" and non-destructive "read"
- consistency is preserved by ordering accesses
- examples: Linda, JavaSpaces

# Java Spaces




- **Spaces are shared**
- **Spaces are persistent**
- **Spaces are associative**
- **Spaces are transactionally secure**
- **Spaces let you exchange executable content**

**In Linda, the Space stores Tuples of simple fields, in Java Spaces the Space stores Tuples of Objects!**

Figure from: http/www.javaworld.com/javaworld/jw-11-1999/jw-11-jiniology.html

# Java Spaces

**write:** Places one copy of an entry into a space. If called multiple times with the same entry, then multiplecopies of the entry are written into the space.

**read:** (readIfExists) Takes an entry that is used as a template and returns a copy of an object in the space that matches the template. If no matching objects are in the space, then read may wait a user-specified amount of time until a matching entry arrives in the space.

**take:** (takeIfExists) Works like read, except that the matching entry is removed from the space and returned as the result of the take.

**notify:** Takes a template and an object and asks the space to notify the object whenever entries that match the template are added to the space. This notification mechanism is built on Jini's distributed event model for a reactive style of programming.

**snapshot:** Provides a method of minimizing the serialization that occurs whenever entries or templates are used; you can use snapshot to optimize certain entry usage patterns in your applications.

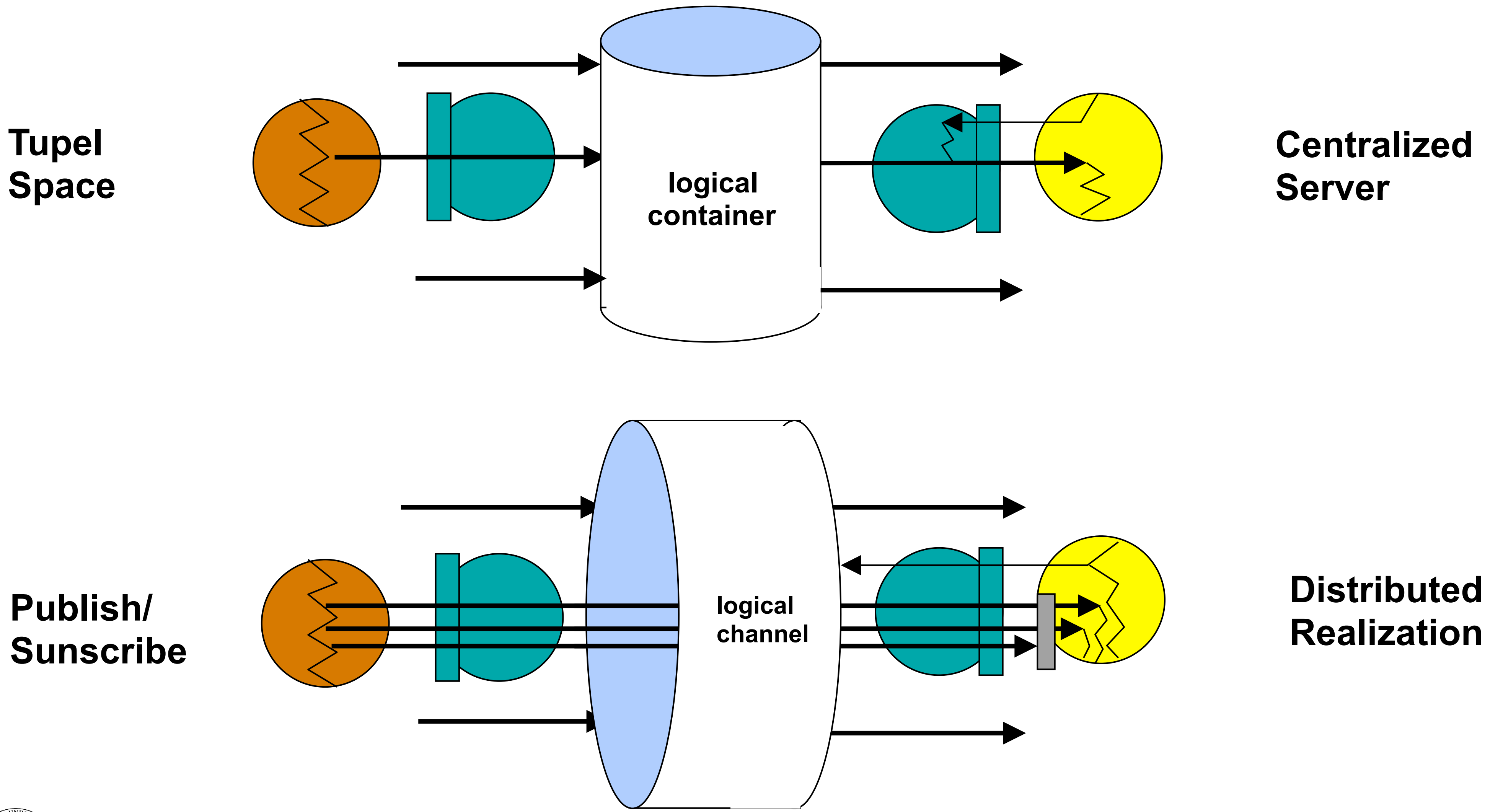
# Java Spaces

## Method Summary

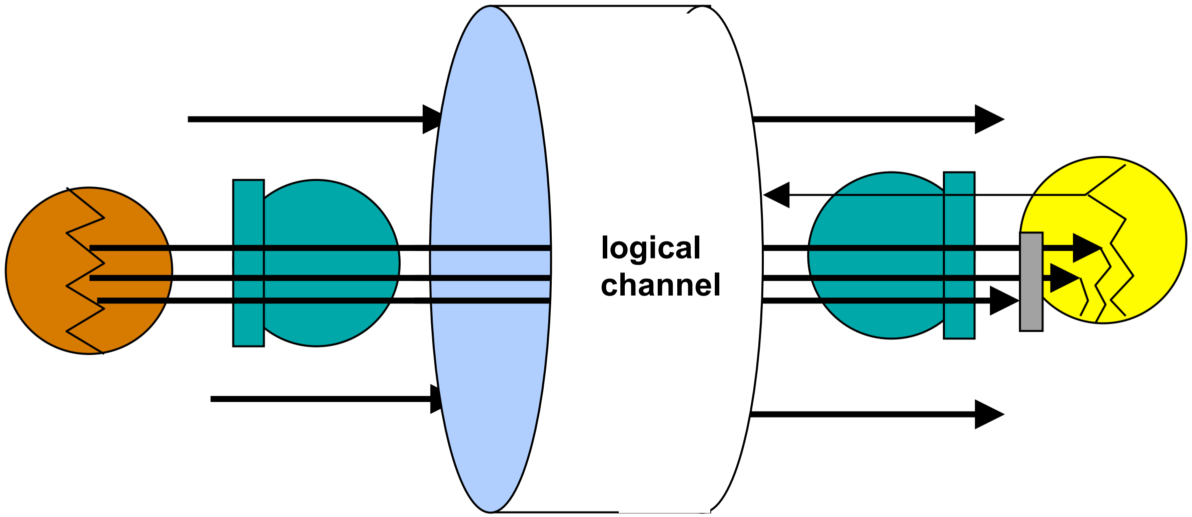
| | |
|---|---|
| EventRegistration | **notify**(Entry tmpl, Transaction txn, RemoteEventListener listener, long lease, MarshalledObject handback)<br>When entries are written that match this template notify the given `listener` with a `RemoteEvent` that includes the `handback` object. |
| Entry | **read**(Entry tmpl, Transaction txn, long timeout)<br>Read any matching entry from the space, blocking until one exists. |
| Entry | **readIfExists**(Entry tmpl, Transaction txn, long timeout)<br>Read any matching entry from the space, returning `null` if there is currently is none. |
| | **snapshot**(Entry e)<br>The process of serializing an entry for transmission to a JavaSpaces service will be identical if the same entry is used twice. |
| Entry | **take**(Entry tmpl, Transaction txn, long timeout)<br>Take a matching entry from the space, waiting until one exists. |
| Entry | **takeIfExists**(Entry tmpl, Transaction txn, long timeout)<br>Take a matching entry from the space, returning `null` if there is currently is none. |
| Lease | **write**(Entry entry, Transaction txn, long lease)<br>Write a new entry into the space. |

**http://www.gigaspaces.com/docs/JiniApi/net/jini/space/JavaSpace.html**

# Spaces vs. Notification Infrastructures



# Publish/Subscribe



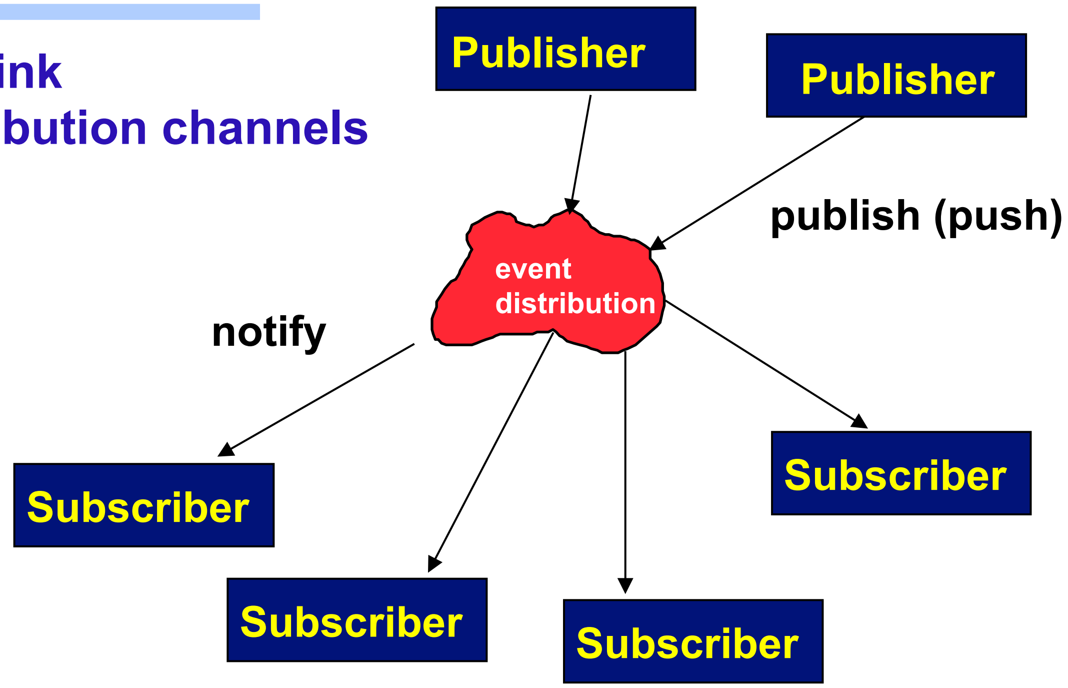

Relation: many-to-many

Coupling:
Space: none
Flow: none
Time: none

Examples:
Information Bus
NDDS
Real-Time P/S
COSMIC
....
....

# The Publisher/Subscriber Model

**Principle: Keep control local and link systems via event distribution channels**




**Information Bus** *(Oki, Pfluegl, Siegel, Skeen)*
**TIB/Rendevous**
**iBus** *(Maffeis)*
**Real-Time P/S** *(Rajkumar, Gagliardi, Sha)*
**NDDS** *(Real-Time Innovations, Inc.)*
**SIENA** *(Carzanoga, Rosenblum, Wolf)*
**Directed Diff.** *(Intanagonwiwat, Govindan, Estrin)*
**COSMIC/FAMOUSO** *(Kaiser, Schulze)*

**Many-to-many communication**

**Support for event-based spontaneous (generative) communication**

**Anonymous communication**

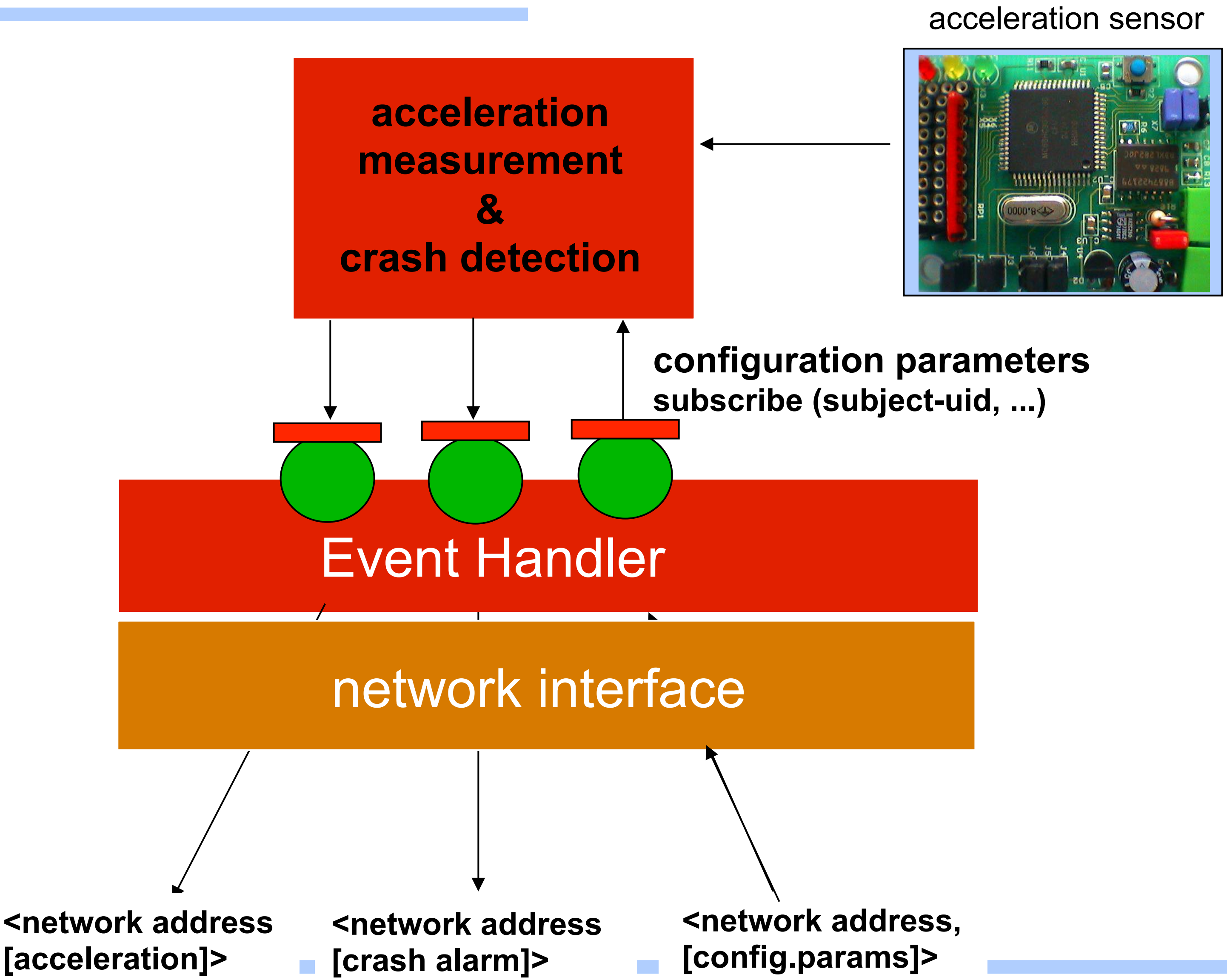
# P/S in a smart sensor application

**event: acceleration**

**publish (subject,attr., [acceleration]);**

**event: alarm**

**publish (subject, attr., [alarm]);**



# Publish/Subscribe

## Problems:

1. Routing: How comes the information from the Publisher to the Subscribers?
    - Content is used
    - Subject is used
    - Type is used

2. Filtering: How can we achieve that only those events are received that are needed?
    - How to specify filters?
    - Where to filter:
        - Sender?
        - Receiver?

# Event Specification and Attributes

**events:** **abstraction defining an individual occurence of an event**

- **treat events as time/value entities**
- **allow to describe context and quality attributes**
- **exploit event attributes by multi-level filtering**

**example:**

distance_event:= <UID, rel_pos., abs_pos., netw_zone, timestamp, validity, distance>
crash_event:= <UID, abs_pos., netw_zone,timestamp, validity, acceleration>

**event channels:** **abstraction of the infrastructure, i.e. explicit specification of the channel through which the events are disseminated**

- **provide dissemination guarantees**
- **support different synchrony classes**
- **encapsulate network configuration functions**

**example:**

distance_channel:= <UID, periodic soft real-time, period, omission degree, not_h, exc_h>
crash_channel:= <UID, periodic hard real-time, reaction_time, omission degree, exc_h>

# Overview

| Abstraction | Space Coupling | Time Coupling | Flow Coupling |
|---|---|---|---|
| • Connected Sockets | Yes | Yes | Yes |
| • Unconnected Sockets | Yes | Yes | Consumer |
| • RPC | Yes | Yes | Consumer |
| • Oneway RPC | Yes | Yes | No |
| • async (Pull) | Yes | Yes | No |
| • async (Callback) | Yes | Yes | No |
| • Implicit Future | Yes | Yes | No |
| • Notifations (Observer Design Pattern) | Yes | Yes | No |
| • Tuple Spaces (Pull) | No | No | Consumer |
| • Message Queues (Pull) | No | No | Consumer |
| • Subject-Based P/S | No | may be | No |
| • Content-Based P/S | No | may be | No |

# What are the options?

| Communication model | Communication abstraction | Communication relation | Routing mechanism | Binding Time |
|---|---|---|---|---|
| message based | message | symmetric | address | design time |
| Remote procedure Call | invocation | client-server | address | design time |
| Distributed shared memory | memory cell | central | address | design time |
| Shared Data Spaces | object,tupel | central | contents | run time |
| Publish-Subscribe | event | Producer-consumer | contents/ subject | run time |