Coordination Languages: Tuple-based and Event-based

Franco Zambonelli

Ottobre 2018

Problems of Distributed Programming

- In programming distributed applications:
  - Often, there are not strict pre-defined roles for the components → processes are peers from the viewpoint of their role
  - Encapsulation → would be great to enable interactions independently of the characteristics of processes
  - We need a mechanism for explicitly designing and controlling interactions between the components → rules for coordination
  - Need to minimize coupling between components → especially needed for dynamic environment (e.g., mobile computing and IoT) where procecesses might not know a priori each other

Distributed Object-based Programming

Limitations

- Based on client-server interactions with clear distinction in roles between processes
- The process structure - the number of processes and their relationships - determines the program structure → no explicit design of interactions –> no explicit design of the coordination rules/constraints
- Processes (objects) must know each other and their location and characteristics

Therefore

- Limited openness
- Hard to deal with dynamic environment
- Conceptually not scalable

The way out - Coordination

"Coordination is managing dependencies between activities."

(From Coordination Theory)

"Coordination is the process of building programs by gluing together active pieces."

(Carriero and Gelernter, Linda creators)

Coordination includes communication + synchronization

Concurrent/Distributed Programming = Computation + Coordination

(Carriero and Gelernter, Linda creators)

Coordination: Models, Languages, & Systems

- The **model** is a Triple **(E, M, L)**
  - **E** – Coordinables, or **entities** being coordinated (agents, processes, tuples)
  - **M** – Coordination media used to coordinate the entities (channels, shared variables, tuple spaces)
  - **L** – Coordination **laws** → semantic framework of the model (guards, synchr. constraints)

- **Coordination languages**
  - "The Linguistic embodiment of a coordination model"
- Coordination system
  - A middleware to support the model

Coordination Languages must Support:

- Separation between computation and coordination.
- Distribution and (if needed) transparency
  - Also decentralization
- Dynamics (no need to link the activities and the roles before runtime) and Adaptivity
- Interoperability → Openness
- Encapsulation (implementation details are hidden from other components)

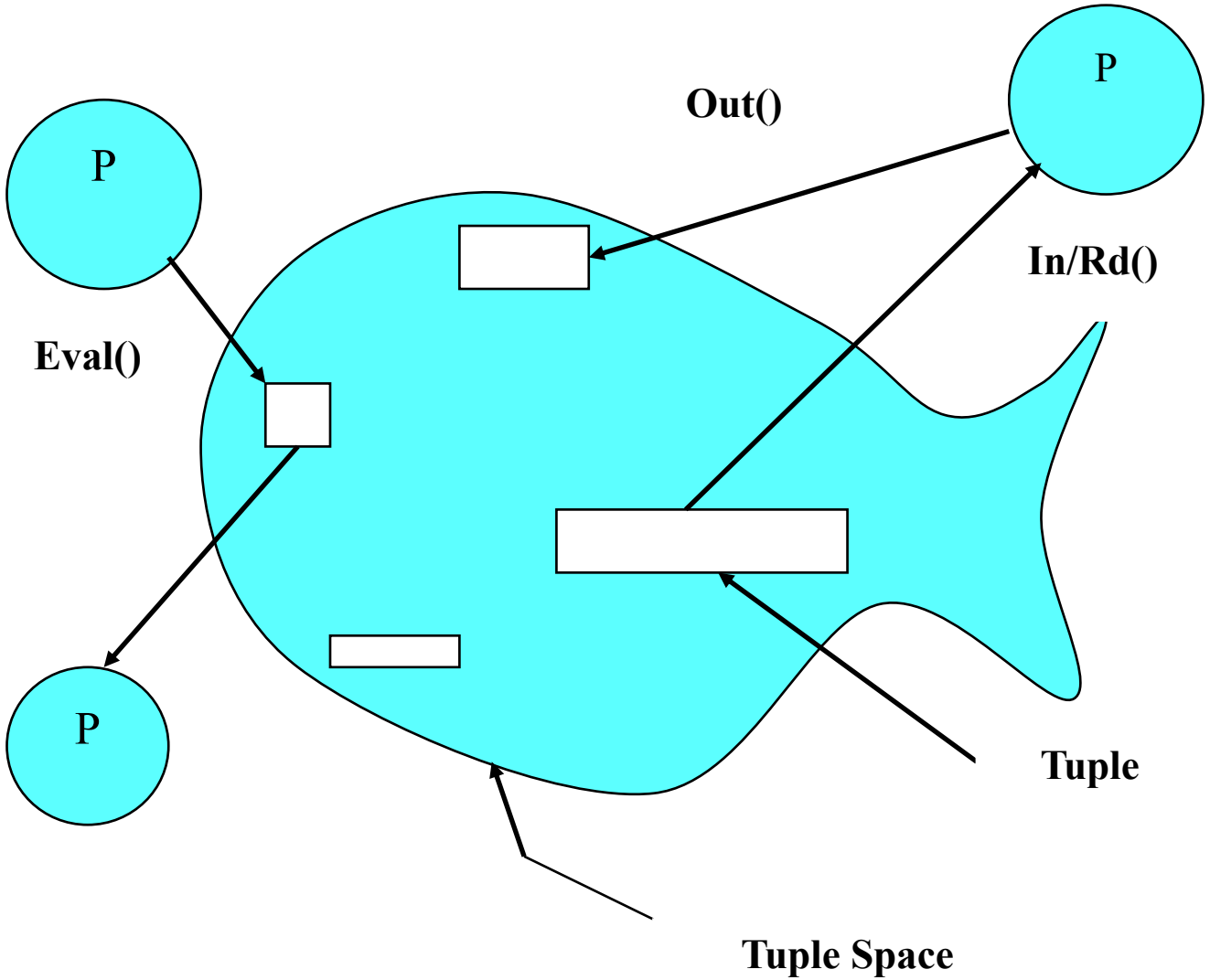
Leading Coordination Models and Languages

- Tuple Space Model (coordination through Shared Data Space)
  - Linda, TSpaces, JavaSpaces, Jini
- Event-based Models
  - We will see it as an extension to Tuple Spaces
- Multiagent Protocols
  - See Lesson on Multiagent Systems

Tuple spaces and Linda

- Based on *generative* communication:
  - Processes communicate by inserting or retrieving data objects (called Tuples) from shared Data Space (called Tuple space)
  - The tuple space is a multiset of "bag of tuples" – unstructured DB
- Interoperability is achieved: Linda places no constraints on the underlying language or platform for the processes

Tuple Space Concept



Tuples

- A tuple is a structured series of typed fields.
  - Example:
  - **(“a string”, 15.01, 17)**
  - **Int index = 5; float value = 4.45;**
  - **(“Vector”, index, value)**
  - Depending on the specific implementation we can have OO tuples or logic tuples, e.g., OO
  - `(String “hello”, Counter c, Person f)`

Operations

`out (t)` insert a tuple t into the Tuple space (non-blocking)

`in(at)` if there is a tuple **t** from the tuple space matching the template tuple **at** (often called "antituple"), it finds and removes it; **block** until a matching tuple is found – a template tuple is a normal tuple that can have some "formal" variable **?v** , i.e., fields undefined in value. These will assume the values found in the marching tuple, i.e., like **`v = value of corresponding field in t`**

`rd(at)` like in(at) but the t not removed blocking

`Bool inp(at)` tests the existence of a tuple without blocking

`eval(t)` add an active tuple t to the tuple space – basically it is the creation of a new process (not often used in real implementations, where process are created as usual)

Tuple matching

Let **at(i)** denote the $\mathbf{i_{th}}$ field in the (template) tuple **at**.

A tuple **at** given in a **in(at)** or **rd(at)** operation "matches" a tuple **t** in the tuple space iff:

1. **t** and **at** have the same number of fields, and
2. for each field

   if **at(i)** is a value then **at(i) = t(i)**

   or

   if **at(i)** is of the form **?x** then **t(i)** is a valid value for the type of variable **x**

If more than one tuple in the tuple space matches, then one is selected **nondeterministically**.

As a result of tuple matching if **at(i)** is of the form **?x**, then **x := t(i)**

Examples of Tuple matching

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

```
("V", 1, FirstElt)
("V", 2, SecondElt)
…
("V", n, NthElt)
```

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

```
rd ( "V", j, ?x )
```

To change the ith element, use

```
in ("V", i, ?OldVal)
out ("V", I, NewVal);
/* Note: impossible updating a tuple – need to remove and
   re-insert!!!!  */
```

Programming Concurrent and Distributed Applications in Linda

- With Linda
  - We can replicate most of the functionalities of message-oriented and client server systems
- However
  - We can use the tuple space as a more general means to realize dynamic and open forms of interaction

Example 1: Sending and Receiving Messages

- We can use the tuple space to simulate a communication channel between two processes

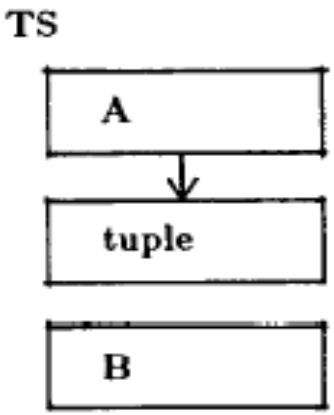


Fig. 1. To send to B, A generates a tuple. . .

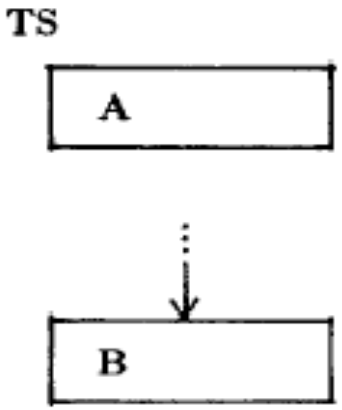


Fig. 2. . . . and B withdraws it.

Sending and Receiving messages (Cont.)

```
// Tuple structure (From, To, Index, Data)

Process p
{ int Index = 0; Process q = q_id; Anytype Data = …;
   out(p,q,Index,Data)
  Index++;
  // further messages…
}

Process q
{ Process sender, p = p_id; Anytype Data;
in(p, q, ?Index,?Data)}
// if interested only in messages from a specific p

in(?sender, q,?Index,?Data)}
// if interested in messages from anyone
}
```

Sending and Receiving messages (Cont.)

- Message-passing model contains tree separate parts:

1. Targets (to-from)
2. Tag (message Id ir index)
3. Communication

- In Linda, Tuple is a set of fields, any of which may be a tag (message Id) or target (address) or communication (data).

Example 2: Client-Server

```
client()
{ int index;
 …
  in (“server index”,?index);
  out (“server index”, index +1);
  …
  out (“server”, index, request);
  in (“response”, index, ?response);
  …
}
```

```
server()
{ int index = 1;
  out(“server index”, index);

while (1){
      in (“request”, ?index, ? req)
      …
     out (“response”, index++, response);
  }
}
```

The index is used to
Associate the client
With its answer

Example 3: Semaphores

- Semaphores

Initialize: `out("sem")`

wait or P-operation: `in("sem")`

signal or V-operation: `out("sem")`

For counting semaphore execute
`out("sem")` n times

Example 4: Concurrent Processes

Multiple processes can execute on the same task, splitted in several subtasks, by simply getting the tuple representing the task No need to know a priori how many processes

```
// create task: the element of the matrixes in the tuple space
for (i,j = 1…N)
{  Out(i,j, “A”, A[i][j]);
   Out(i,j, “B”, B[i][j]);

   In(i,j, “C”, ?C[i][j]); // read results
}
// then launch any number of concurrent processes doing this
While(inp(?i,?j, “A”, ?int)
{  in(?i,?j, “A”, ?A[i][j]);
   in(?i,?j, “B”, ?B[i][j]);
in(?i,?j, “C”, A[i][j]+B[i][j]); }
```

Concurrent Processes: Masters and Workers

- Is general:
  - A master process divides work into discrete tasks and puts them into global space (tuple space)
  - Workers retrieve tasks, execute, and put results back into global space
  - Workers notified of work completion by having met some condition → e.g. no more tuples representing task or insertion by the master of a specific tuples
  - Master gatherers results from global space

Masters and Workers (Cont.)

```
master() {
    for all tasks {
            …/* build task structure*/
           out (“task”, task_structure);  }
  for all tasks {
           in (“result”, ? &task_id, ?
            &result_structure); }
}

worker() {
      while (in (“task”, ? &task_structure) {
            …/* exec task */
            out (“result”, task_id,
           result_structure); }
}
```

Example 6: Barrier Synchronization

Each process within some group must wait at a "barrier" until all processes in the group have reached the barrier; then all can proceed.

- Set up barrier:

```
out ("barrier", n);
```

- Each process does the following:

```
in("barrier", ? val);
out("barrier", val-1);
rd("barrier", 0).
```

Example 7: Dining Philosophers Problem

```
phil(i)
{   int i;

    while (1) {
      think();
      in (“room ticket”);
      in (“chopstick”, i)
      in (“chopstick”, (i+1)% Num);
      eat ();
      out (“chopstick”, i);
      out(“chopstick”, (i+1)%Num);
      out (“room ticket”);
  }
}
```

```
initialize()
{ int i;
    for (i=0; i< Num; i++){
          out (“chopstick”, i);
          create_process( phil(i) );
          if  (i< (Num-1))
                out(“room ticket”);}
}
```

Example 8: Analyzing Sensor Data

```
sensor(id)
{
   if (new interesenting sample)
     out(“temperature”, id, value, timestamp);
}
```

```
analyzer()
{ int MAX=0, sensor_MAX, sensor_id, new_value, timestamp;

   while (1) {
      in
   (“temperature”, ?sensor_id, ?new_value,?timestamp);

      if(new_value > 0)
            MAX = new_value;
            sensor_max = sensor_id;
   }
}
```

Linda achieves:

- "Good" Nondeterminism
- Structured naming. Similar to "select" in relational DB, pattern matching in AI.

```
in(P, i:integer, j:boolean)
in(P, 2, j:boolean)
```

- **Time uncoupling** (Communication between time-disjoint processes)
- **Space uncoupling** (no need to share a process name space)

Linda: Limitations

- Events
  - How to handle asynchronous computing
  - Need to capture events without need to polling the tuple space
- The language is although intuitive, but minimal
  - Real world implementation adds several features to make it useful in real-world
- Implementation:
  - How is the tuple space to be implemented in the absence of shared memory in a distributed system?
- Security
  - Need to have access rights and confinmeent in the access to tuple spaces
- Programmability of Coordination Laws
  - It is possible to change the basic matching rules so as to support different schemes (constraints on interactions or specific composition rules)?

Handling Events

- Add a primitive operation to Linda
  - **`Sub((at), Handler)`** `-- and also` **`unsub(t)`**
  - Non blocking
- When a tuple matching at appears in the tuple space, execute the function Handler
  - In this way, a process does not have to "poll" the tuple space
  - The tuple space becomes the fact an "Event-dispatcher" to handle publish/ subscribe interaction model
- Since tuples are persistent (while events can be temporary) we can consider also adding a "timeout" (max time of life) to tuples
  - **`Out((t), timeout)`**

Example 8: Analyzing Sensor Data Event-based Approach

```
sensor(id)
{
   while(1, every 10 seconds)
     out((“sensor”, id, value, timestamp),10);
// the last parameter specify the lease time
}
```

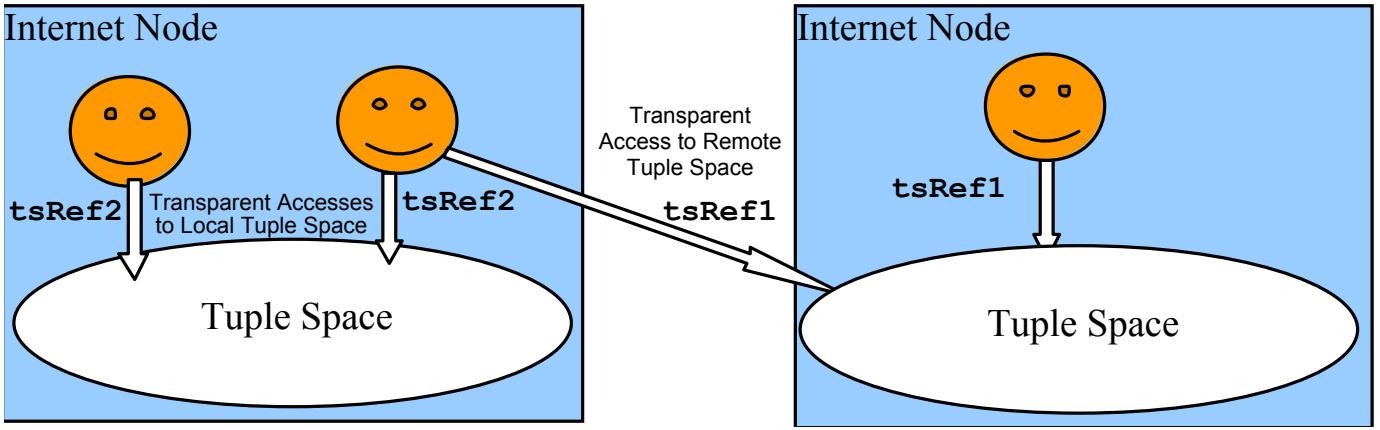
```
analyzer()
{ int MAX=0, sensor_MAX, sensor_id, new_value, timestamp;

   sub((“sensor”, ?sensor_id, ?new_value,?timestamp), f1);
   // the analyzer can do other stuff without polling the
}

void f1(t) // t is the tuple matching the subscription
{if(t.new_value > 0)
            MAX = t.new_value;
            sensor_max = t.sensor_id;
}
```

Real-world Example: Java Spaces

- Integrated in J2EE and JINI
- Is the basic engine behind the discovery services of JINI and the JNDI
- Can be used as a stand-alone service to implement tuple-based coordination

JavaSpaces: Coordination Media

- Tuple spaces as (remote) Java objects
- Possible multiple ***independent*** tuple spaces in different nodes
  - tuple spaces identified and accessed via object-references (as in RMI)
  - Accesses can be remote
- References to tuple spaces can be retrieved via the Loopup service of JINI



JavaSpaces: Tuples

- Object-oriented Linda operations
  - Tuples are called "Entry"
  - Their fields are objects
- The object tuple *tmp*:
  - subclass of `Entry` ; its instance variable are the tuple fields

```
public class MyTuple extends Entry {
public int field1;
public float field2; // a field could be any object…

public MyTuple(int field1, float field2)
{this.field1=field1;
 this.field2=field2}
```

JavaSpaces: Operations

- JavaSpaces interface
  - **`Read (Entry tmp, Tr txn, long lease)`**
  - **`Take (Entry tmp, Tr txn, long lease)`**
  - **`Write (Entry e, Tr txn, long timeout)`**
  - `txn` can specify a transactions (*all-or-nothing* semantics)
  - `lease` blocking time for operations (not infinite blocking)
  - `timeout` time to live for tuples, then they are deleted
- JavaSpaces matching
  - The template must be a class or a superclass
  - Its field with "null" value correspond to `?type` of Linda templates

JavaSpaces: Events

- Reactive event notification
- Agents can register in a space as interested in a tuple matching a template, and require to be notified about its insertion

```
notify(template, null, listener, lease.FOREVER, null)
```

  - the listener will be sent an event it has to handle when a matching tuple will be inserted in the space
- Can catch a limited set of events!

JavaSpaces: Sample Code (1)

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

// definition of the tuple class
public class MessageEntry implements Entry {
   public String content;
   public int value;

   public MessageEntry() {
   } // with no paramaters is for initializing templates

   public MessageEntry(String content, int value) {
     this.content = content;
     this.value = value;
   }

   public String toString() {
     return "MessageContent: " + content + “ " + value;
   }
}
```

JavaSpaces: Sample Code (2)

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

public class SpaceClient {
   public static void main(String argv[]) {
      try {
         MessageEntry msg = new MessageEntry(“Valore”, 5);
         // tuple created and initialized

         System.out.println("Searching for a local JavaSpace...");

         Lookup finder = new Lookup(JavaSpace.class);
        JavaSpace space = (JavaSpace) finder.getService();
         // this is the lookup service of JINI

         System.out.println("A JavaSpace has been discovered.");

// CONTINUE ON NEXT PAGE
```

JavaSpaces: Sample Code (3)

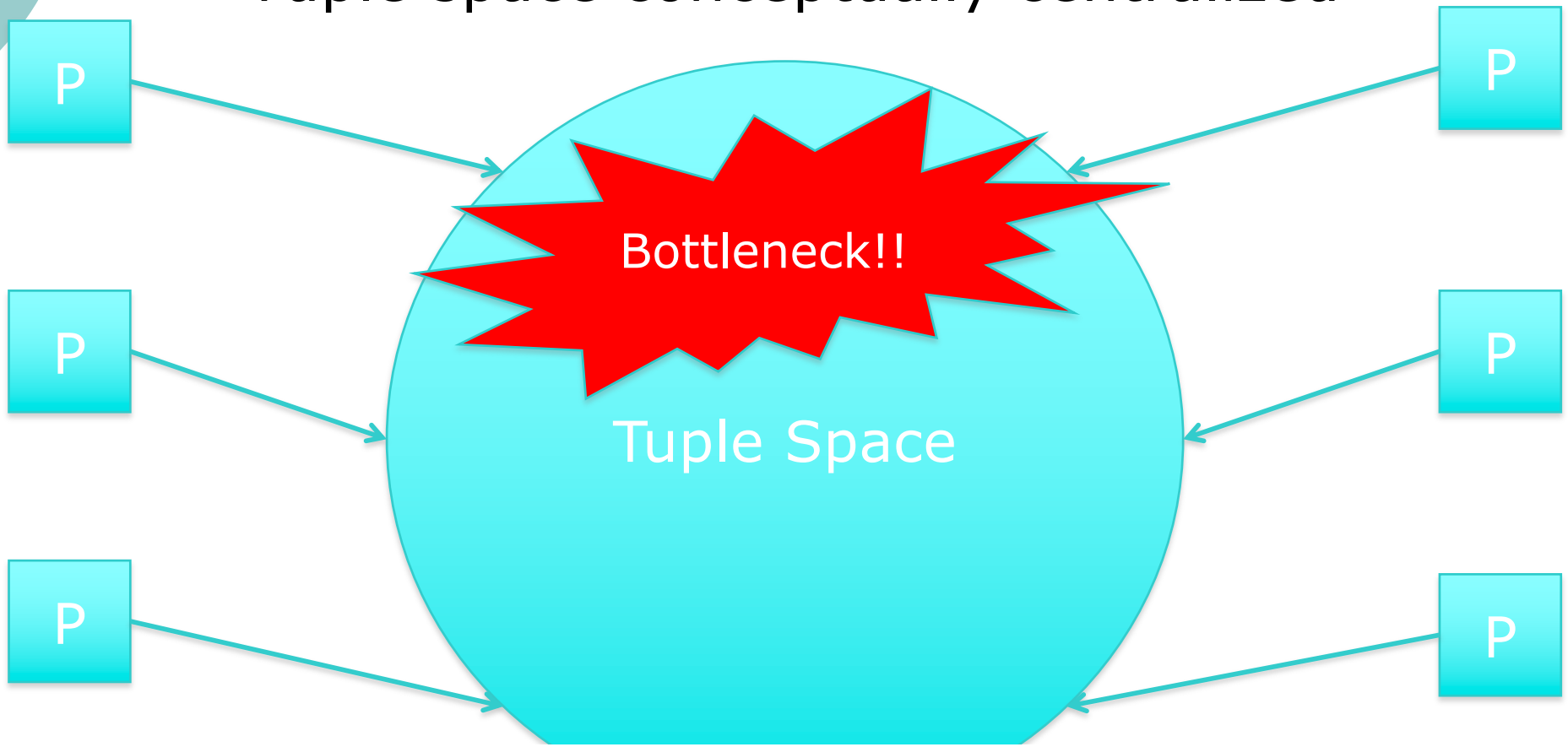
```
// CONTINUE FROM NEXT PAGE

        System.out.println("Writing a message into the space...");
        space.write(msg, null, 60*60*1000);

        MessageEntry template = new MessageEntry();
        template.content = “Valore”
        // a template has some fields null

        System.out.println("Reading a message from the space...");

        MessageEntry result =
         (MessageEntry) space.read(template, null, Long.MAX_VALUE);

         System.out.println("The message read is:”
                              + result.content);
     } catch(Exception e) {
        e.printStackTrace();
 }}}
```

Implementing Tuple Spaces

- Two main approaches
- Global (virtually unique) distributed tuple space
- Local multiple tuple spaces (set of localities)

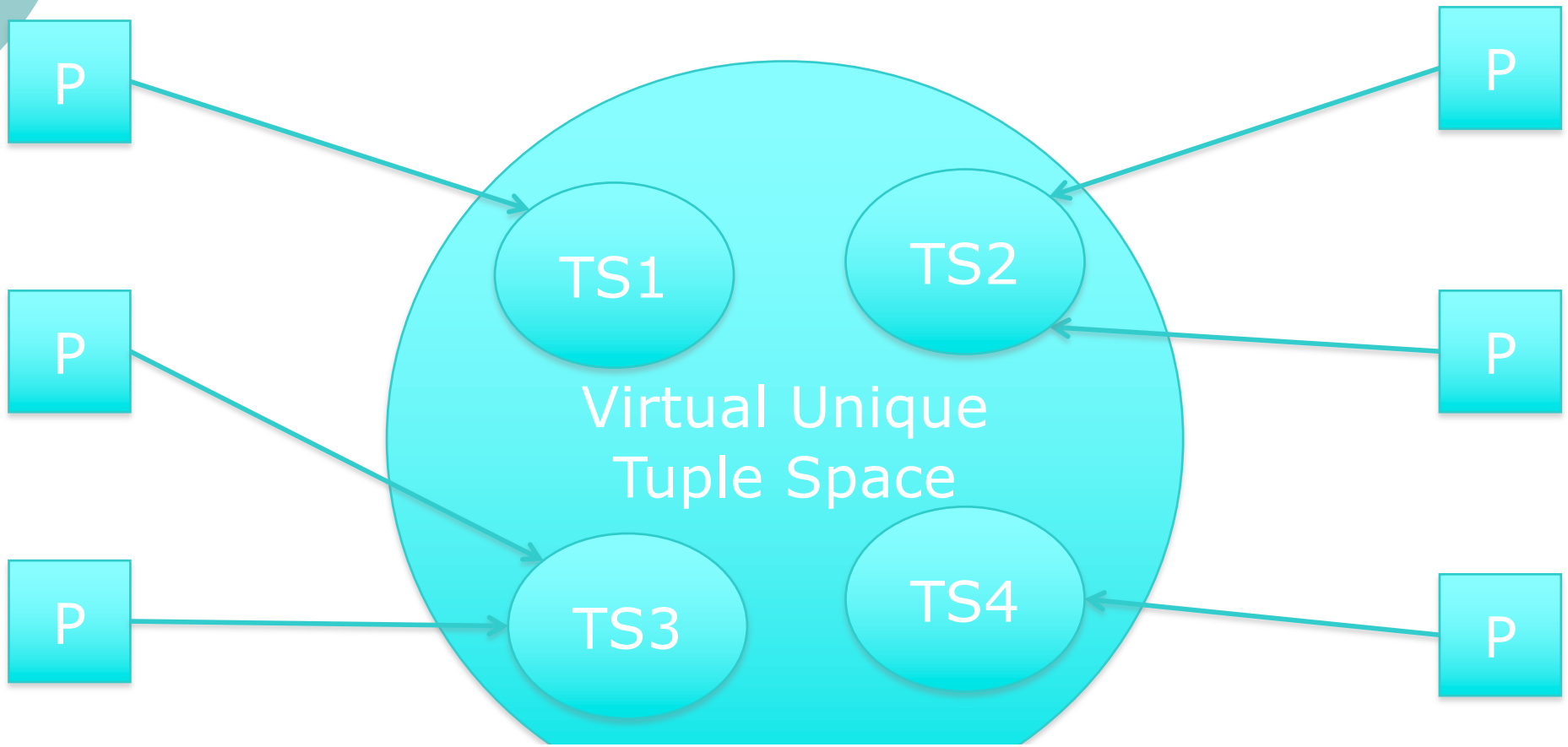
Global Tuple Space

- Dilemma
  - Processes distributed
  - Tuple space conceptually centralized



Distributed Implementation of a Virtually Unique Tuple Space

- A multiplicity of tuple spaces
- Coordinate to appear as a single one
- Locally accessed by processes

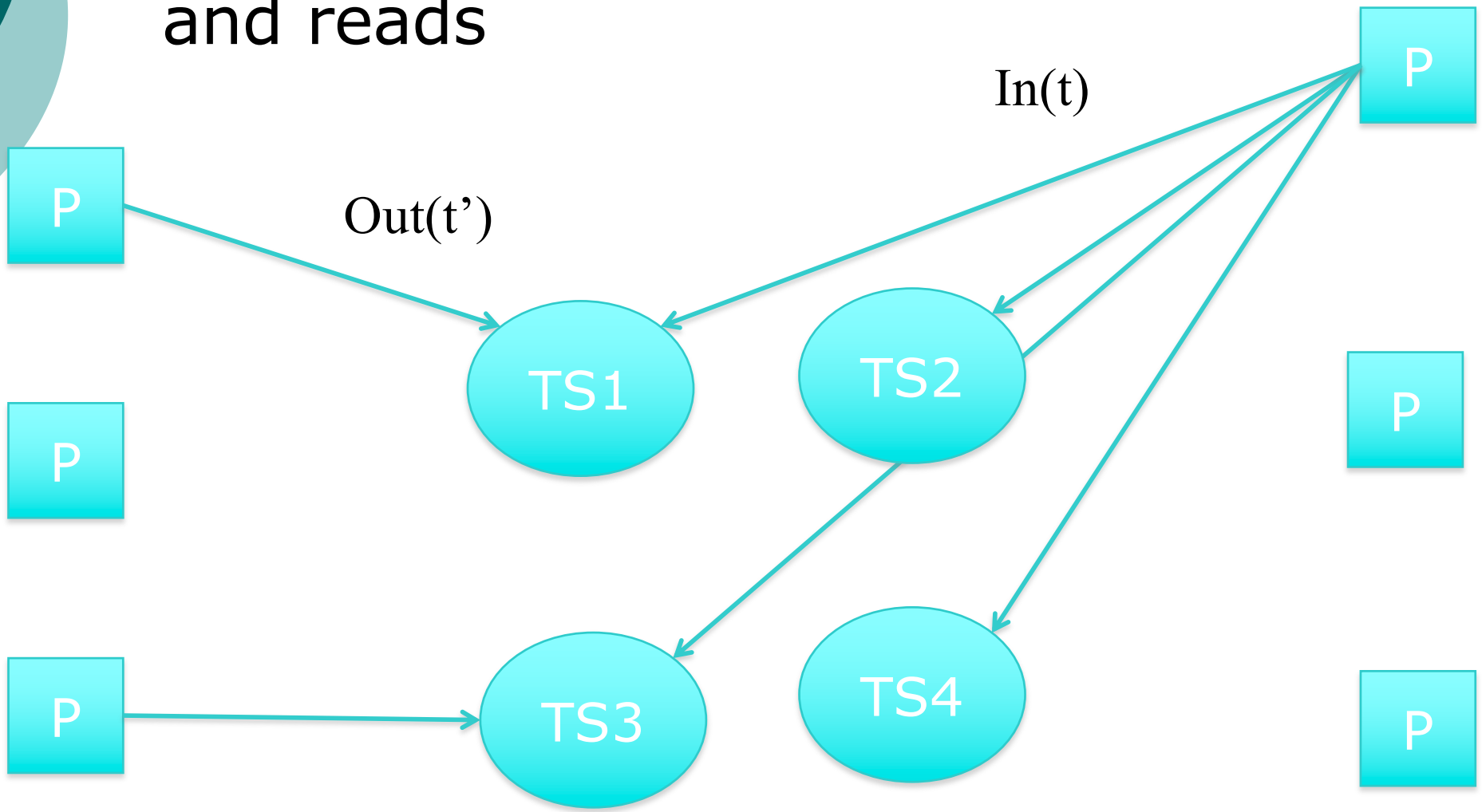


Goal of a Distributed Implementation

- Reachability
  - For any `in(t)` and `read(t)` operations
  - If there exists a tuple t' matching t – this must be found
- Consistency
  - In case of `ins` operations
  - A. Only a single instance of t' could be returned to different processes (a single tuple cannot be extracted multiple times)
  - B. process could extract a single t' with an operation

Strategies: Replicative Ins

- Local copy of outs, broadcast of ins and reads

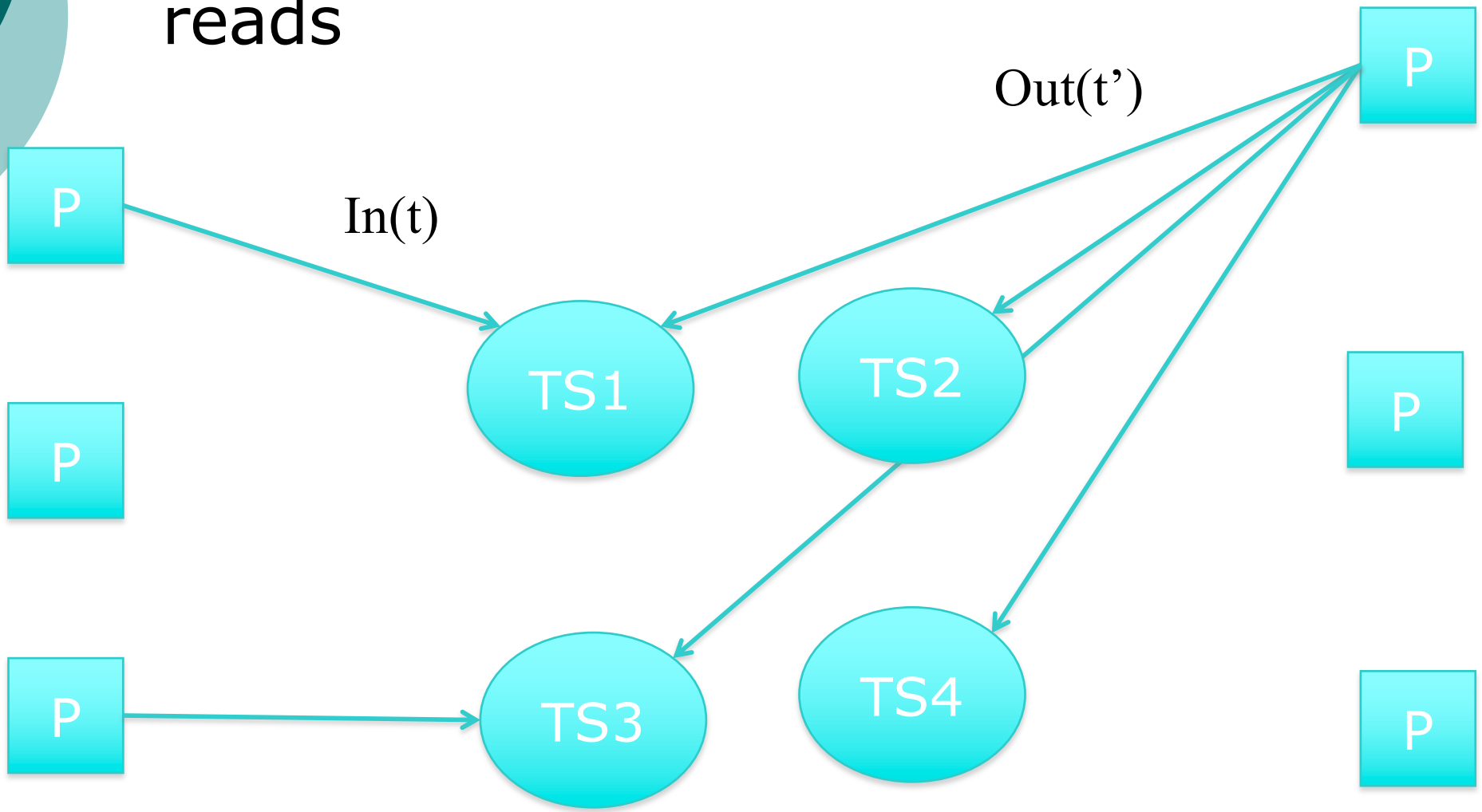


Strategies: Replicative Ins

- Reachability
  - Ensured by construction
- Consistency
  - A. Given that a tuple t' exists only on one node, it is not possible for different processes to extract the same tuple
  - B. if the same process discover multiple matching tuples on different nodes, these are re-outed on such nodes – this is true both both `ins` and `rds`. This implies executing the operation withing a **distributed transaction algorithm**.
- Important: a blocking `in(t)` implies replicating the `in(t)` request on all nodes until it is satisfied, this once a match has been found on a node all replicated `ins` have to be deleted

Strategies: Replicative Outs

- Broadcast of outs, Local of ins and reads



Strategies: Replicative Outs

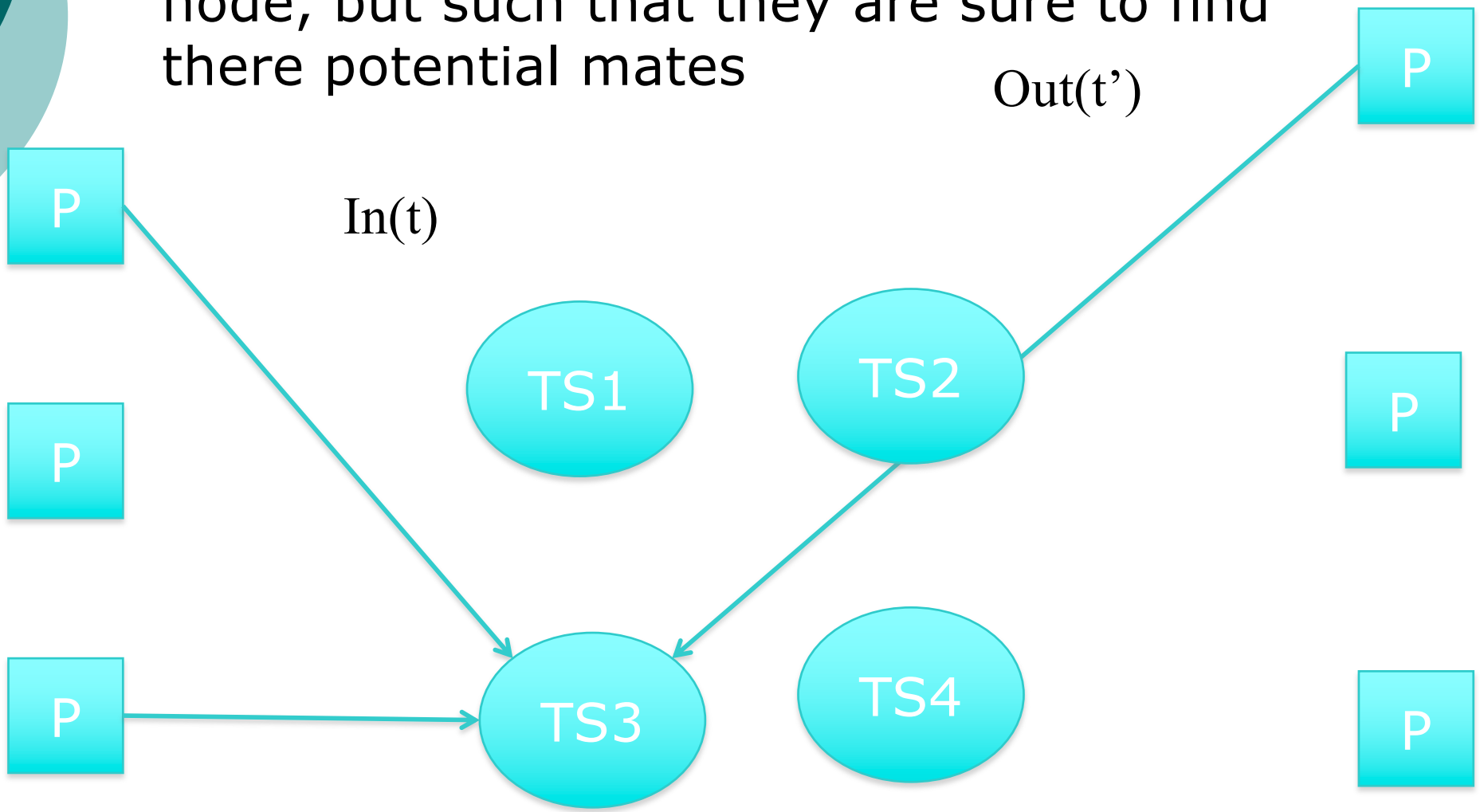
- Reachability
  - Ensured by construction
- Consistency
  - A. Once the in found a local matching tuple t', it must coordinate a process for extracting the same tuple t' on all nodes. Only if all replicas of t' have been successfully extracted t' can be delivered to a process. This is a problems of atomic acquisition of multiple resources, similar to the dining philosophers → requires **distributed mutual exclusion algorithms**
  - B. No problem of discovering multiple tuples on different nodes

Pros Cons of Replicative Policies

- Pro: Exploit locality in the accesses
  - For either ins/rds or outs a process can access the most local tuple space
- Cons:
  - Broadcasting in large-systems can be expensive
  - The transactions and mutual exclusion algorithms to be implemented can be complex

Strategies: Hash-based

- Assign tuples and templates to a single node, but such that they are sure to find there potential mates



Strategies: Hash-based

- Given N nodes that are implemeting the tuple space
- Define a function H that maps a tuple (or a template) into one of the N nodes
  - **H: T → 1..N**
- Make sure that, for any tuple t and for any potentially matching template t’
  - **H(t)=H(t’)**
- Typically, the hash function applies either on field string characterising all tuples of a given kind, or on the structure of the tuples

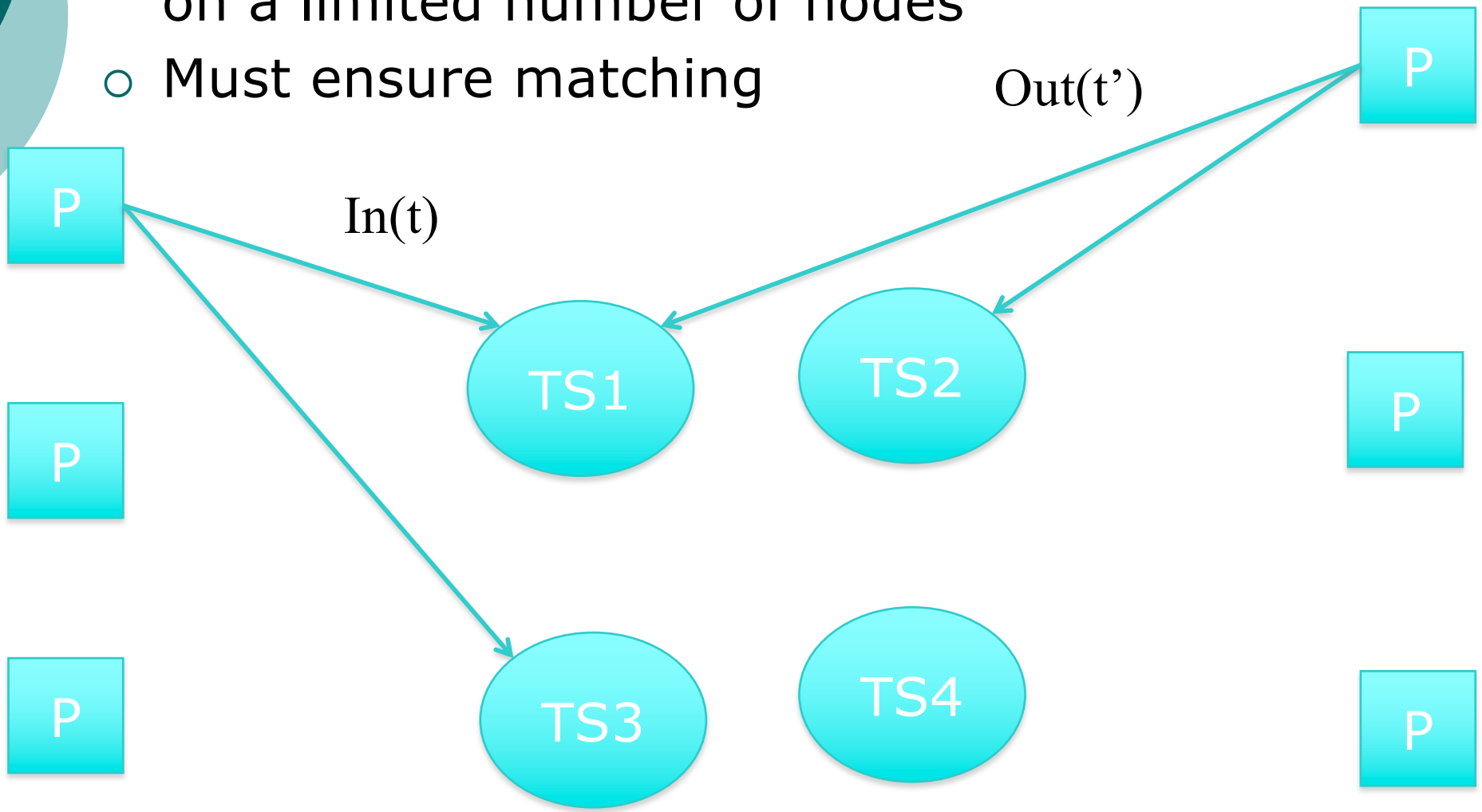
Strategies: Hash-based

- Reachability guaranteed by construction
- Consistency guaranteed because there is not replication

- Problem:
  - Do no exploit locality in the accesses
  - The node that a process accesses is not related to the location of the processes

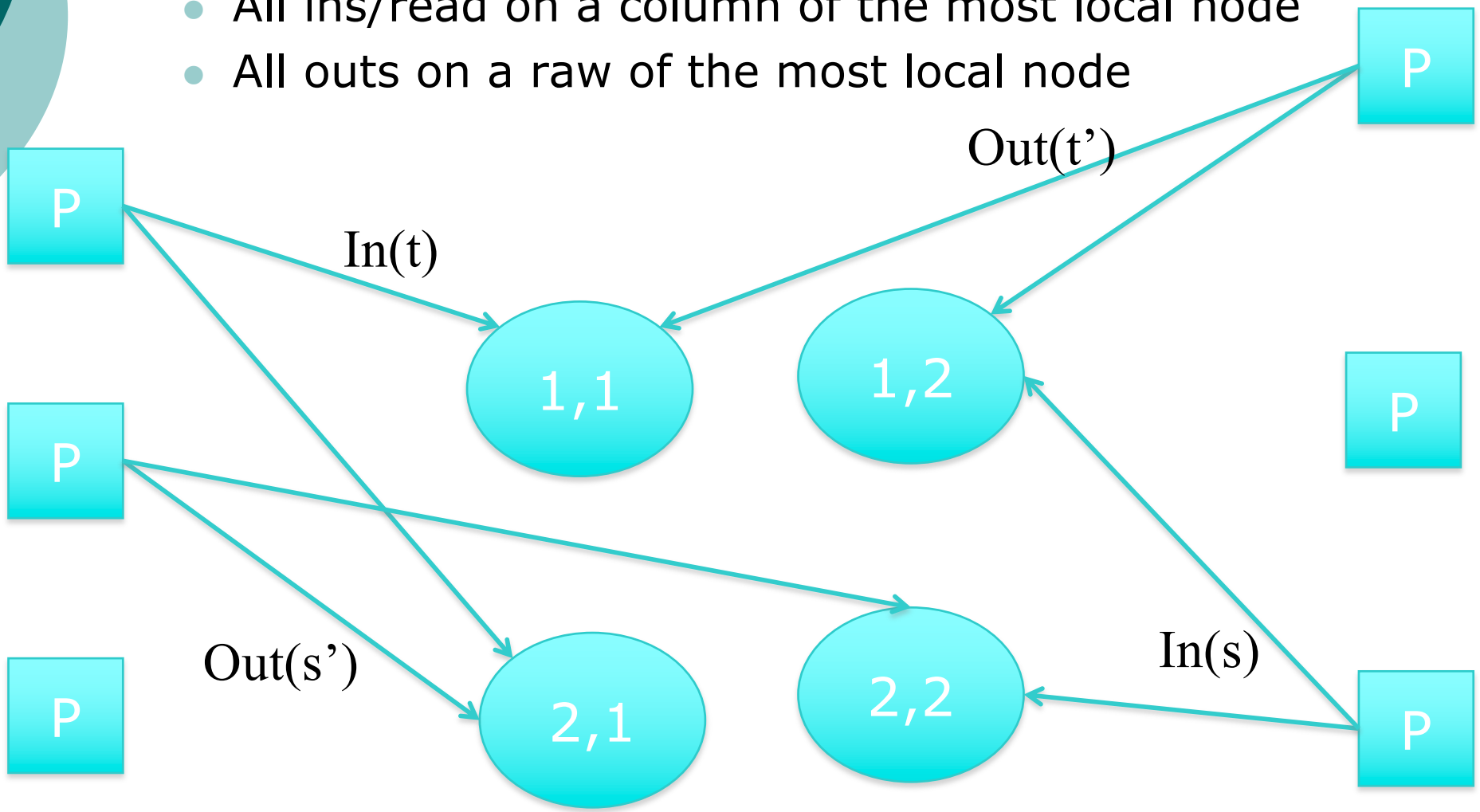
Strategies: Partitionative

- Multicast of outs, Multicast of ins and rds, on a limited number of nodes
- Must ensure matching



Strategies: Matrix Partitionative

- Organize nodes as elements of a virtual matrix
  - All ins/read on a column of the most local node
  - All outs on a raw of the most local node



Strategies: Partitionative

- Reachability
  - Must find a way to ensure it
  - Matrix scheme
  - Hierarchical scheme (F. Zambonelli, 1995); nodes organized as a hierarchy replicating in and out in a whole branch of the hierarchy, from leaves to root
- Consistency
  - Inherits both problems of replicative ins and replicative outs policies
  - But with distributed algorithms that have to execute on a much lower number of nodes
- Pros:
  - Locality in accesses
  - More scalable that fully replicative

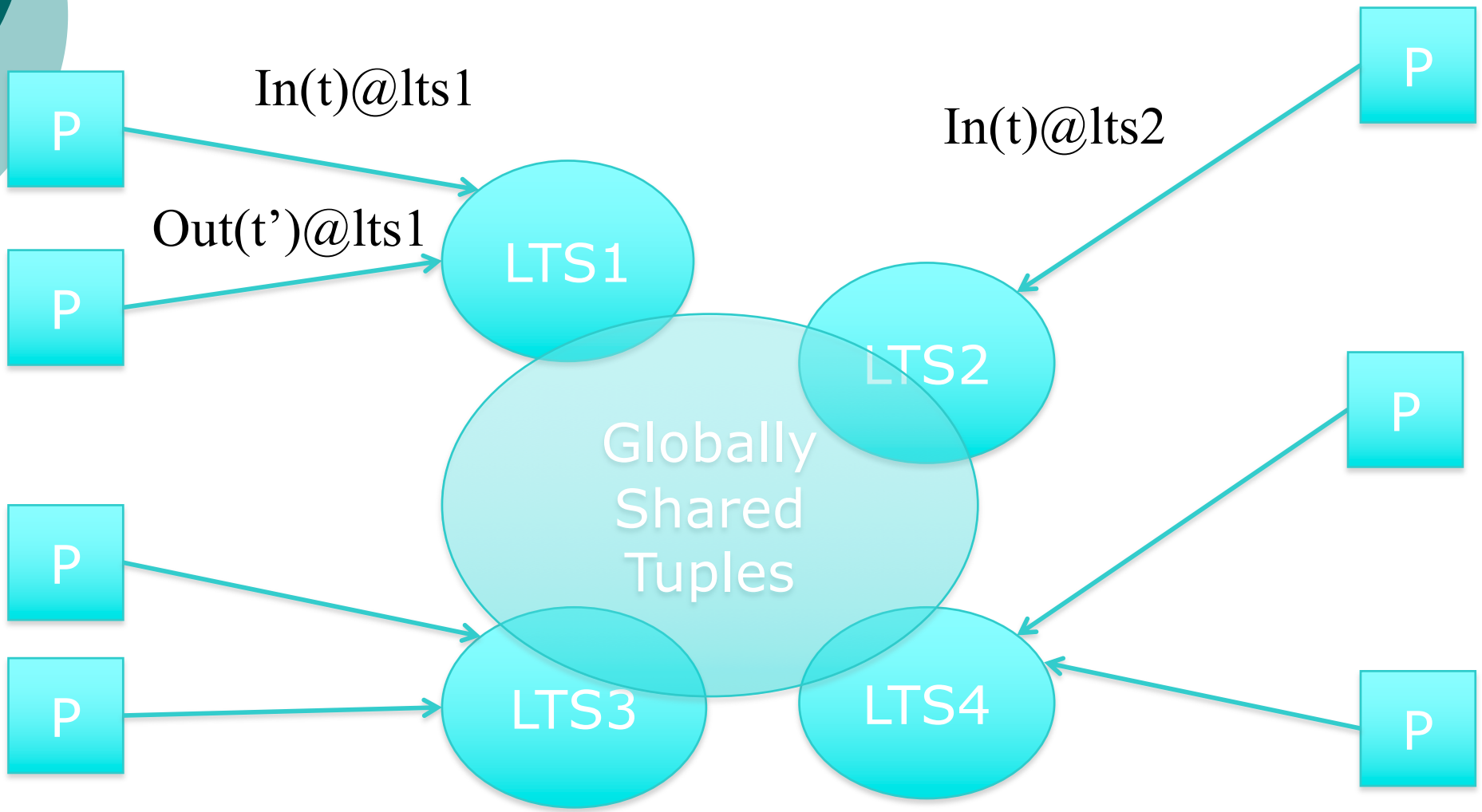
Strategies: Considerations

- A very similar problem to that of distributing names in a distributed name space, or events in a ditributed event-dispatcher system – Similar strategies have been proposed
- Most existing systems adopts the hash-based scheme or the hierarchical scheme
  - Effectively applicable also to P2P systems, where the "tuples" are the files and each client maps into a portion of the hash function
- **But….is it really always necessary to provide the abstraction of a single global tuple space???**

Local Multiple Tuple Spaces

- Rather than having a single global tuple space, go with
  - A Multiplicitly of (mostly) independent tuple spaces
  - Associated to a local logical/physical domain
  - Each with its own local tuple set
  - Possibly sharing only a limited portion of the overall tuples (that is, only a portion of the tuples are "global"
- Processes
  - Access the local tuple space for most concerns of local computation
  - And sometimes produce "global" tuples
  - And/Or sometimes access other non-local tuples spaces

The General View of Multiple Tuple Spaces



Motivations for Multiple Tuple Spaces (1)

- Separation of Concerns and Encapsulation
  - Encapsulating in a tuple space some data, and giving it access to a limited set of processes
- Security
  - Create access control policies to access to tuple spaces
  - Possibility of giving different grants to different processes in different places

Motivations for Multiple Tuple Spaces (2)

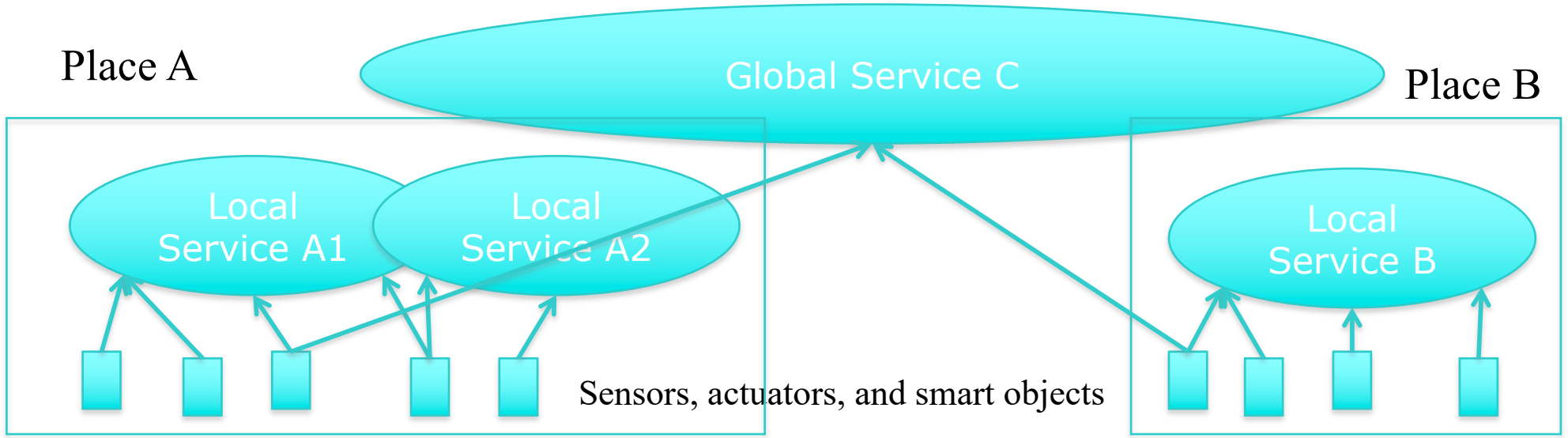
- Inherently "localized" interactions
  - In many applications, processes needs to interact on a location-based way
  - Interact with other processes physically or logically co-located → edge computing
  - Whereas a limited amount of processed needs to interact at a more global scale
- Examples:
  - Intra-organizational processes (logical co-location)
  - IoT - Sensing and acting in the physical world (physical co-location)

Example: Internet of Things (IoT)

- In the Internet of things vision, Distributed processes associated to sensors and actuators spread in the physical environment and on "smart objects" within → e.g., a smart hotel
- The objects, sensors, and actuators in a room need to interact with each other to provide services, e.g.,
  - Set up windows curtain and lights for lecture presentation in a meeting room
  - Activate sleeping mode in a guest room
  - In the case of detected fire, activate estinguishers, alert via digital signages, open security doors

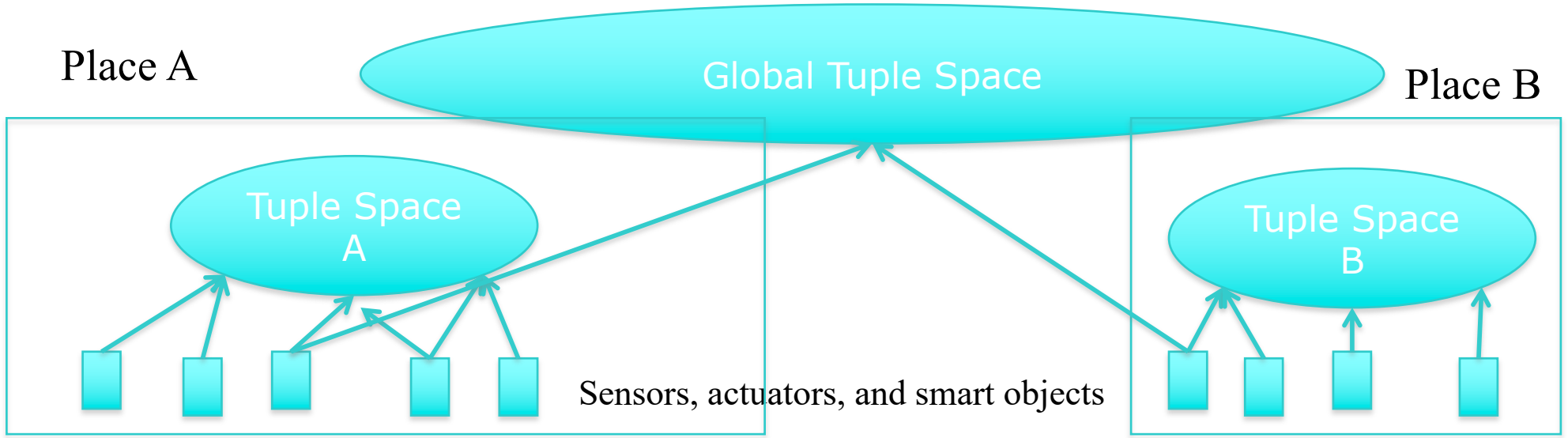
IoT: Local and Global Services

- We can clearly see, in the examples that there are
  - Local services, involving physically localized interactions among local components (e.g., meeting room and guest room services)
  - A few global services, requiring global coordination of some classes of processes at the global level (e.g., fire alarm services)



IoT: Solution with Multiple Local Tuple Spaces

- Local Services are realized by tuple-based coordination of local processes over a local tuple spaces
- Global services are realized (depending on the adopted model) by coordination of global processes
  - By having some of the tuples of interest flow also to the global tuple space, where global processes can access them
  - By having global processes also access local tuple spaces



IoT: Example of Local Service

```
initiate_conference_mode(id)
{
   localTS.out("curtain", "close");
  localTS.out ("lights", "middle_level");
  localTS.out ("projector", "on");
}
```

```
window_curtain()
{ string state, newstate;

 localTS.sub(("curtain", ?new_state), actuate);
}

void actuate(t) // t is the tuple matching the subscription
{if(t.new_state != state)
            actuator.execute(new_state);
}
```

IoT: Example of Global Service (1)

```
temperature_sensor(id)
{ int current_t, alarm_t=60;

current_t = sensor.getval();

 while(1)
   localTS.out(“temp”, current_t);
    if (current_t < alarm_t)
              global_TS.out(“temp”, id, current_t);
}
```

IoT: Example of Global Service (2)

```
digital_signage()
{ string local_message, alarm_message;

 globalTS.sub((“alarm”, “display”, ?alarm_message), actuate);

 localTS.in(“display”, ?local_message);

  signage.visualize(local_message);
}


void actuate(t) // t is the tuple matching the subscription
{
   signage.visualize(t.alarm_message);
}
```

IoT: Example of Global Service (3)

```
set_up_global_evacuation_plan()
{ int signals = 0, id, int temp;
  global_TS.sub((“temp”, ?id, ?temp), start_evacuation);
// don’t  care who or how many sensor exists
}

void start_evacuation(t)
{ signals ++; //
  if (signals > 3) // at least 3 sensors have reported

  globalTS.out(“alarm”, “display”, “Escape! doors opening”);

  globalTS.out(“security_door”, “open”, “now”);
  globalTS.out(“estinguisher”, “activate”, “now”);
  // don’t care who or how many actuator exists
}
```

Limited Flexibility of Coordination Law: Programmable Laws

- Concept invented by Omicini & Zambonelli, 1998
- A problem of tuple spaces is that the "coordination laws" in the Linda model are is always the same
  - basic pattern-matching (defined fields are primitives data types that, if actual, must have the same value)
  - object-matching (e.g., equality of Java serialised form, tuple sub-classing)
- Some times there could be need of more flexible approaches, to avoit complex sequences of ins and outs
  - `In(all the sensor tuples with a 20 < temp < 60)`
  - `In (the sensor with the max temp)`
- Also, sometimes, tuple spaces associated to different localities may have differen rules (also for security purposes)
- How can we do that?

Programmable Coordination Laws

- Programmability of the Coordination language
  - primitives overriding (the default behaviour of a primitive can be changed)
  - adding of new primitives
- Programmability of the Coordination media
  - full monitoring of access events (who, what, on which tuple)
  - which behaviour in response to which access event

Conclusions

- Linda a very effective approach to distributed system programming
  - Simple programming
  - Time and space uncoupling
  - Suited for dynamic scenarios
  - Various implementation possibilities
- Also suited for physically-situated services
  - E.g., sensor networks, Internet of things, robotics

References

- D. Gelernter, N. Carriero, "Coordination Languages and Their Significance", Communications of the ACM, Vol. 35, No. 2, pp. 96-107, February 1992.
- A. Omicini, F. Zambonelli, “Coordination for Internet Application Development”, Journal of Autonomous Agents and Multiagent Systems, Vol. 2, No. 3, Sept. 1999.
- G. Cabri, L. Leonardi, F. Zambonelli, “MARS: a Programmable Coordination Architectures for Mobile Agents”, IEEE Internet Computing, Vol.5, No. 4, July-August 2000.