JAVA Spaces

Master 2 Informatique

Frédéric Raimbault

Université de Bretagne Sud – Vannes

- JavaSpaces: Principles, Patterns, and Practice, E. Freeman and al., Addison Wesley
- JavaSpaces in Practice, P. Bishop and al., Addison Wesley

What is JavaSpaces ?

JavaSpaces is a service specification providing a distributed object exchange and coordination mechanism (which may or may not be persistent) for Java objects. It is used to store the distributed system state and implement distributed algorithms. In a JavaSpace, all communication partners (peers) communicate and coordinate by sharing state. A JavaSpace is a Jini service that stores Java objects in memory. This makes it very useful for supporting collaboration among other services, in addition to its uses as a simple shared memory.

Overview

- Principles
- Implementations
- Sharing
- Synchronizations
- Communications
- Producer-consumer pattern
- Distributed events
- Distributed implementation
- Recalls on Java RMI

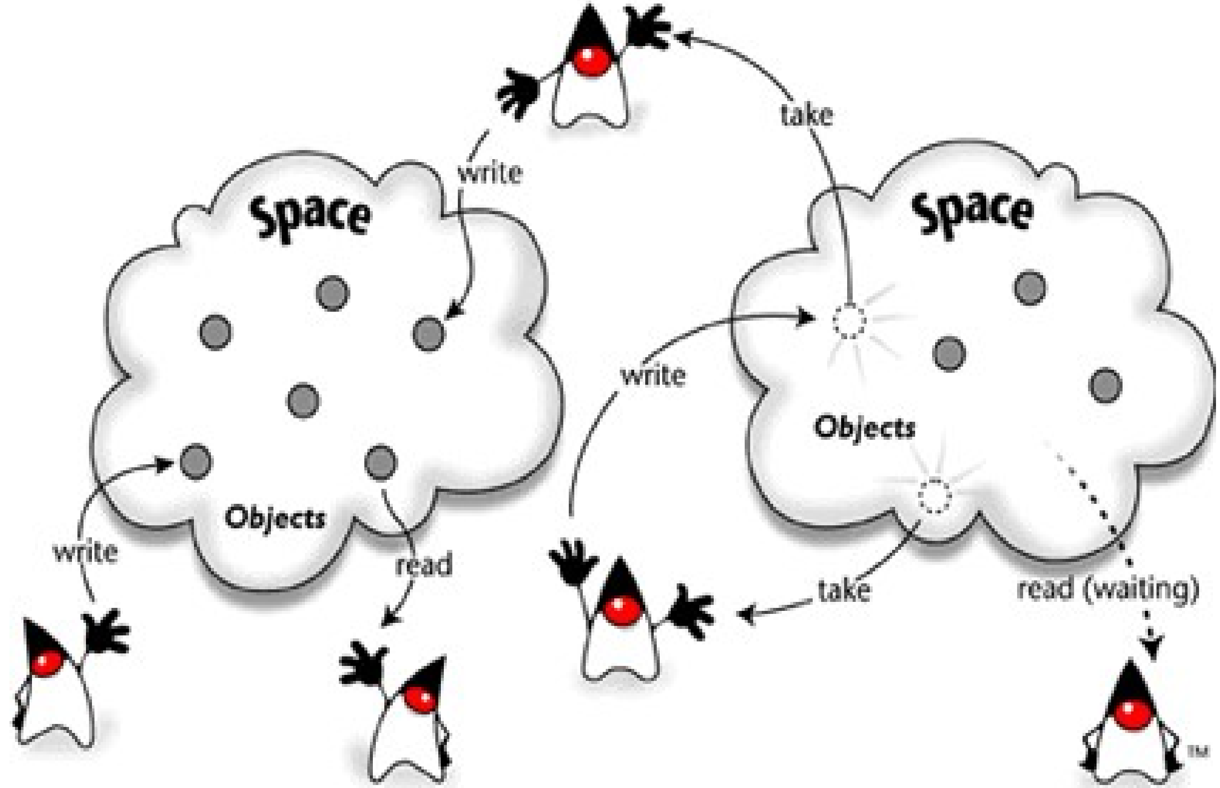
Principles: Linda

- Simple parallel programming model, language and machine independent
  - Developped at Yale by Gelernter and Carriero
  - « Linda in context », D. Gelernter and N. Carriero, CACM, Apr. 1989, vol. 32, no 4, pp. 444-458.
- Sequential processes communicating using a tuple-space
  - Computations are written in a standard language
  - Communications are expressed using primitives on a shared tuple-space.
  - Processes are not aware of each other

Tuple-Space Operation

- Shared virtually associative memory
  - Centralized (server)
  - Or physically distributed or replicated (caches)
- Tuple : ordered sequence of typed values
- 3 operations on tuple-space:
  OUT, IN, RD, (EVAL)
- A tuple is selected by pattern matching
  - Content based access
- Operations are decoupled
  - No explicit synchronization between processes

Operations



- OUT (write)
  - Put a tuple in the tuplespace
  - Non blocking
- IN (take)
  - Use a *template* to retrieve a tuple
  - <u>Blocked</u> as long as no tuple is found
  - A matching tuple is returned
  - The matching tuple is withdrawn from the tuplespace
- RD (read)
  - As IN without the withdrawal

Properties

- Non-determinism of
  - the choice of the process that performs an operation
  - the order of operations between processes
- Atomicity of insertion and deletion
- Filtering by pattern (*template*)
  - ex : IN("card", ?c, ?r)
    - selects (and removes) any card=("card",color,rank)
  - ex : IN("card", ?c, "9")
    - selects (and removes) any 9 of any color
  - ex : IN("card", "clubs", "J")
    - selects (and removes) the jack of clubs

Tuple Template

- Specifies tuples to be retrieved
- Sequence of typed fields containing
  - Values
  - And / or variables (wildcard)
- A tuple matchs a pattern if and only if it has the same
  - Number of fields
  - Values for value fields
  - Types for variable fields
- Indeterminism in the choice of the tuple among all those that match

Implementations

- GigaSpaces
  - In memory data-grid (XAP)
    - https://docs.gigaspaces.com/
- TSpaces
  - Communication middleware
    - http://www.almaden.ibm.com/cs/TSpaces/
- JavaSpaces
  - Jini/Outrigger (Sun)
  - (Apache) River
    - http://river.apache.org/

Apache River

- Toolkit for developing distributed applications in Java
  - Linux, OS X, Windows plateforms
  - Communication between clients and services by RPC: JRMP (RMI compatible) and JERI protocols
  - Above TCP, SSL, HTTP, HTTPS, Kerberos
- REGGIE : service discovery
- HTTPD : class sharing and loading
- MAHALO : transactions
- OUTRIGGER : Shared memory of tuples (JavaSpaces)
- See https://river.apache.org/

JavaSpaces vs. Linda

- Tuples are objects (instance of `Entry`)
    - Tuples may have public methods
  - Values are <u>public reference</u> fields
    - Matching is done by *bitstream* comparison
  - A template is also an instance of `Entry`
    - Fields with `null` value are « ? » equivalent
  - Tuples are polymorphic
    - May return a subtype of the type of the pattern
- Tuples have an expiration lease
- Renamed operations: `write,take,read`
- Added operation: `notify`

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

Lease

- A lease is associated with each entry
  - `Lease.FOREVER`, `Lease.ANY`, or a time in ms
- At the end of the lease, the entry is deleted
  - Useful, eg to avoid saturating / polluting the space or to avoid deadlocks
- The write operation returns a `net.jini.Lease`
  - An application can handle a lease,
    know its expiration date, renew it, cancel it etc.

Operation Parameters

- `read,take` operations have a *timeout*
  - `Long.MAX_VALUE` for blocking operation
  - `JavaSpace.NO_WAIT` for non-blocking operation
    - as `readIfExists()` or `takeIfExists()`
  - A time in ms in the case of transaction
- All operations can belong to a transaction
  - `null` parameter if no transaction
  - Need the MAHALO service in Apache River

Transactions

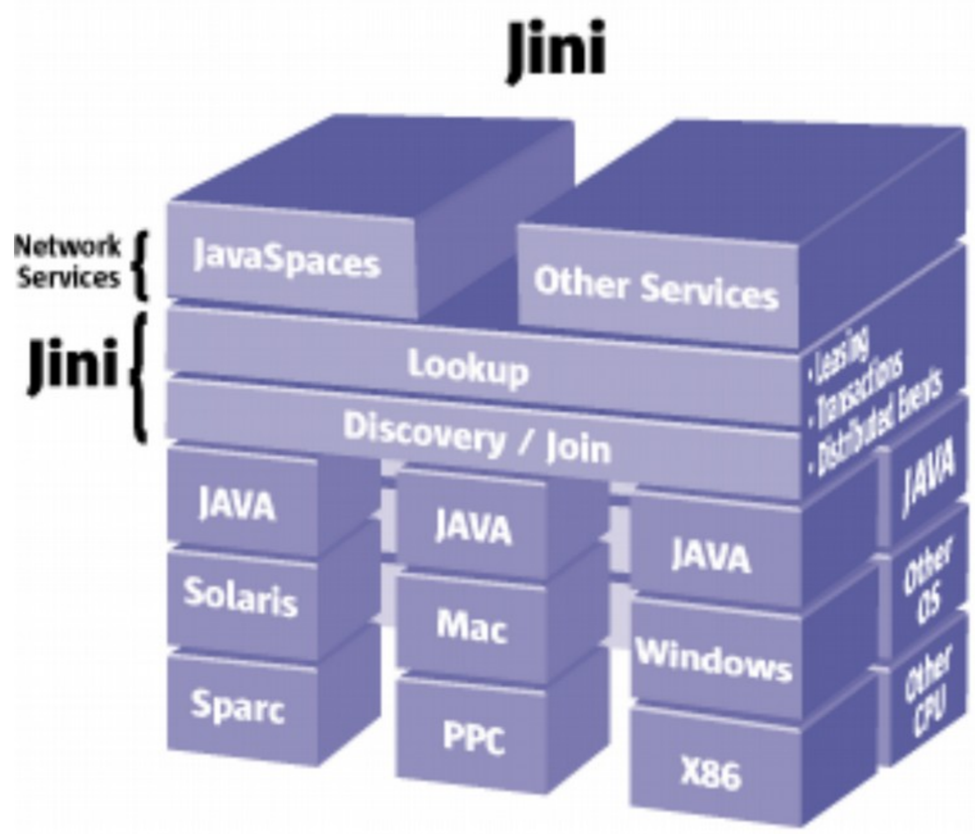
- Make it possible to group a set of operations which must either: be all executed or all canceled if at least one of them fails
  - Write: as long as the transaction has not been committed the entry is invisible for *read*, *take* or *notify* external to the transaction
  - Read: as long as the transaction has not been committed, the entry can not be removed
  - Withdrawal under transaction: if the transaction is canceled, the entry is returned to the space
- Introduce an additional (temporal cost)
  - transactions are optional

Persistence of Entries

- Data persist until they are removed or expire
  - But they are immutable
- 2 levels of space persistence:
  - Transient: inputs are only stored in RAM
  - Persist: as soon as the entry is written, it is guaranteed that it will be recovered after a stop or a crash
- Influence on performance

Jini

- Introduced by Sun in 1999
- Middleware for dynamic management of distributed services over a network
  - Service Discovery
  - Service Record
  - Service Search
  - Lease and event management
  - Lock and transaction management
- Restated in 2006 by Apache in the River project
- Used to implement JavaSpaces



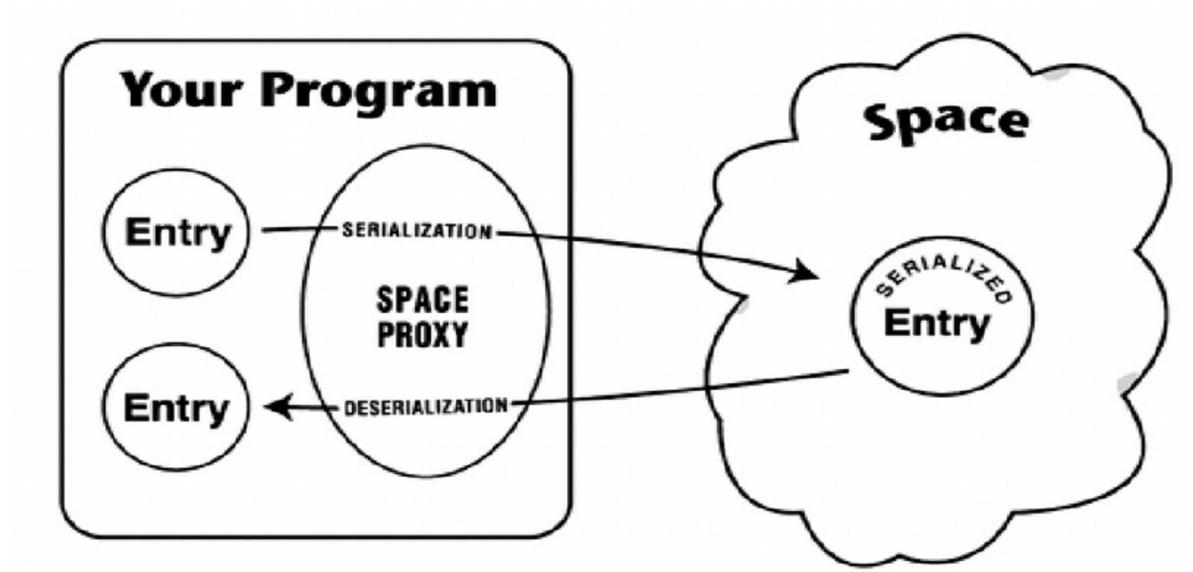
net.jini.core.entry.Entry

```
public interface Entry
      extends java.io.Serializable{
}
```

- Tuples are implementation of `Entry`
- Fields of tuples must be public objects
  - No primitive type (use a wrapper)
- Fields must be *Serializable* or declared *transient*
- Fields are serialized separately
  - Shared references are not retained

Serialization

- Serialization
  1. Class writing
  2. Public field writing
- Deserialization
  1. Reading the class
  2. Instantiation with the default constructor
  3. Reading public fields
- Possibility of avoiding useless serializations (pattern in a reading loop, for example) by memorizing them (`snapshot()` method)



Example 1

- The process A deposits an entry for a process B
- Each time an entry is read by B a counter (of the entry) is incremented
- From time to time the A displays the number of times B reads the entry (the counter value)
- A and B stop after 10 readings by B
  - See Chapter 1 of JavaSpaces Principles, Patterns and Practice (document on the ENT)

Example 1: Message Tuple

```
public class Message implements Entry {
   public String content;
   public Integer counter;
   public Message() {} //for templates
   public Message(String content, int initVal) {
      this.content = content;
      counter = new Integer(initVal);
   }
   public String toString() {
      return "read " + counter + " times.";
   }
   public void increment() {
      counter = new Integer(counter.intValue()+1);
   }
}
```

Example 1: Writer

```
public class Writer {
   public static void main(String[] args)
      throws Exception{
      Message msg = new Message("Hello World", 0);
      JavaSpace space = SpaceAccessor.getSpace();
      space.write(msg, null, 1000*60);
      Message tpl = new Message();
      while(true){
         Message m= (Message)
            space.read(tpl, null, Long.MAX_VALUE);
         System.out.println(m);
         if (m.counter.intValue() >= 10) break ;
      }
      space.take(tpl,null,Long.MAX_VALUE);
   }
}
```

Example 1: Reader

```
public class Reader {
   public static void main(String[] args)
      throws Exception {
      JavaSpace space = SpaceAccessor.getSpace();
      Entry tpl = space.snapshot(new Message());
      for (int i=0;i<10;i++) {
         Message m = (Message)
            space.take(tpl, null, Long.MAX_VALUE);
         m.increment();
         space.write(m, null, Lease.ANY);
         Thread.yield();
      }
   }
}
```

Multi-User Area: MyEntry

```
package raimbaul ;
public class MyEntry implements Entry {
  public String myId ;
  public MyEntry(){
        myId= System.getProperty ("user.name");
  }
}
```

```
package raimbaul ;
public class Message extends MyEntry {
   public String content;
   public Integer counter;
   public Message() {}
   public Message(String content, int initVal) {
      this.content = content;
      counter = new Integer(initVal);
   }...
```

Cleaning the Memory: JSClean

```
package raimbaul ;
public class MyEntry implements Entry {
...
public static void clean() throws ... {
  JavaSpace space = SpaceAccessor.getSpace();
  Entry tpl= space.snapshot(new MyEntry());
  while(true){
   MyEntry m= (MyEntry)
              space.take(tpl, null, 5*1000);
   if (m==null) break;
   System.out.println("removed : "+m.toString());
  }
```

Memory Contents View: JSList

```
public static void list() throws … {

  JavaSpace space = SpaceAccessor.getSpace();
  Entry tpl= space.snapshot(new MyEntry());
  List<MyEntry> found= new LinkedList<MyEntry>();
  while(true){
    MyEntry e= (MyEntry)
               space.take(tpl, null, 5*1000);
    if (e==null) break;
    found.add(e);
    System.out.println("found: "+e.toString());
  }
  for(MyEntry e:found){
    space.write(e, null, Lease.ANY);
  }
}
```

Example 2

- Producer / consumers
- Producer: generates N integers and places them in Javaspace. Wait until their square has been calculated and display the result.
- Consumers: read integers, compute their square, save the computed square into the integers.
- An entry: 1 integer and its associated square

Example 2: Producer

```
for(int i=1;i<n;i++){ // write integers
  IntValue val= new IntValue(i);
  space.write(val, null, Lease.ANY);
}
for(int i=1;i<n;i++){ // display squares
  IntValue tpl= new IntValue(null,true);
  IntValue val= (IntValue)
space.take(tpl,null,Lease.FOREVER);
  System.out.println(square("+val.value+")="
       +val.square);
}
```

Example 2: IntValue Entry

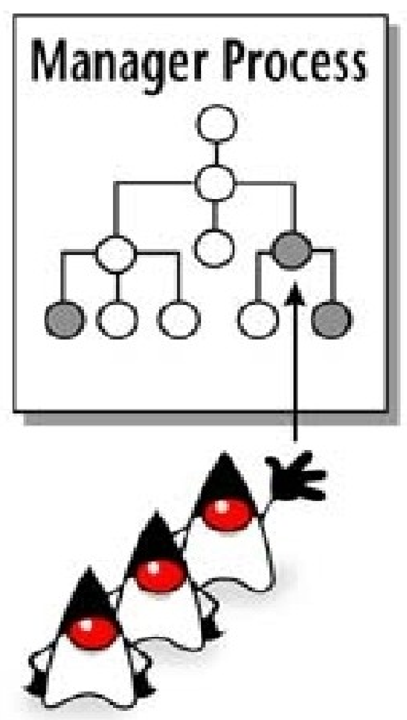
```
public Integer value, square;
public Boolean computed;
public IntValue() { // mandatory constructor
}
public IntValue(Integer v) {
  this.value = v;
  this.computed= false;
}
public IntValue(Integer v, Boolean c) {
  // useful for templates
  this.value = v;
  this.computed= c;
}
public void computeSquare() {
  this.square = new Integer(value*value);
  computed= true;
}
```

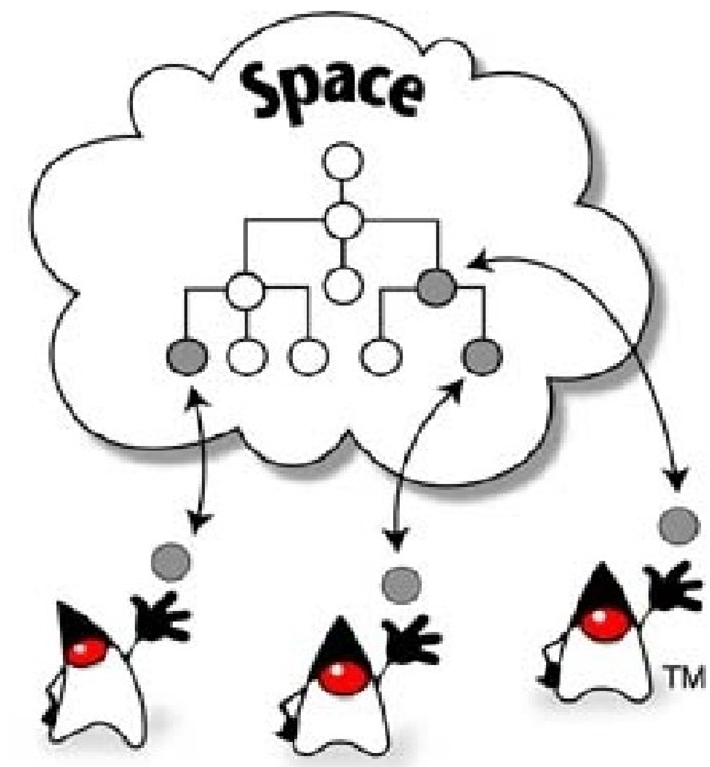
Example 2: Consumers

```java
IntValue tpl = new IntValue(null,false);
while(true){ // as long as they are found
  IntValue val = (IntValue)
                  space.take(tpl, null, 5000);
  if (val==null) break; // nothing found
  val.computeSquare();
  space.write(val, null, Lease.ANY);
}
```

Data Sharing

- JavaSpaces allows simultaneous access to <u>different</u> parts of a shared data structure
  - Avoids sequential access





Shared Variable

- Atomicity of changes: no reading while writing variable

```
public class SharedVar extends MyEntry {
    public String name;
    public Integer value;
    public SharedVar() {}
    public SharedVar(String name) { this.name = name; }
    public SharedVar(String name, int value) {
        this.name = name;
        this.value = new Integer(value);
    }
}
```

```
SharedVar myvar = new SharedVar("counter", 0);
space.write(myvar, null, Lease.ANY);
```

```
SharedVar tpl = new SharedVar("counter");
SharedVar result= (SharedVar) space.take(tpl, null, Long.MAX_VALUE);
result.value = new Integer(result.value.intValue() + 5);
space.write(result, null, Lease.ANY);
```

Shared Array

```
public class Element extends MyEntry {
    public String name;
    public Integer index;
    public Object data;
    ...
```

```
public Object readElement(int pos){

    Element tpl= new Element(name, pos, null);
    Element element= (Element)space.read(tpl, null,Long.MAX_VALUE);
    return element.data;
}
```

- Problem: how to know its size?

    => Add shared metadata to the table

    - eg. start index and end index

Synchronizations

- To manage the sharing of limited resources
- To coordinate parallel execution
- In a centralized system:
  - Role of the operating system
  - Using the main memory
- In a Distributed System:
  - No global controller
  - Cooperation by communication
- With tuple-space:
  - Exploit the (virtually) shared memory

Semaphore

- It is a counter representing the number of available instances of a resource
  - P: Blocking operation to access an instance
  - V: Non-blocking operation to release an instance
- Tuple-space Implementation: 1 entry per instance
  - P: removes an instance of space
  - V: adds an instance in space
- 2 problems:
  - A task that fails may not release its instance (a limited time lease should help)
  - No equity management

Deadlock

- The use of semaphores can lead to situations of deadlock, ex:
  - 2 resources A and B in exclusive access
  - Process P0:
    `resA.P(); resB.P(); calc(); resB.V();`
  - Process P1 :
    `resB.P(); resA.P(); calc(); resA.V();`
  - Not bound to an implementation choice
- No general solution
  - Sometimes by adding other semaphores

The Philosophers' Dinner

- Resource Sharing School Case
- 3 states:
  - thought (indefinite time)
  - is hungry
    (finite time if not, famine)
  - eat (determined time)
- Risk of deadlock
  - If they all take their right stick
  - If one dies with a stick

Dijkstra (1930-2002)

- Semaphore
- Shortest path algorithm
- Goto statement considered harmful
- Language Algol
- self-stabilization of a distributed system

Solutions

- In the case where the number of philosophers is even
  - Philosopher with even number: first take the left
  - Philosopher with odd number: first take the right
- Otherwise use semaphores (Dijkstra)
  - 1 semaphore per chopstick
    - Deadlock if all take the left
  - 1 semaphore for pre-access to the table
    - Access for all except 1)

Problem of Starvation

- There is no guarantee that a philosopher will ever have access to the table and to its chopsticks
- A general solution: numbered access ticket
  - As the ticket at the butcher shop
  - Guarantees an access to the table
- Tuple-space implementation:
  - A shared counter for ticket and number increment
  - Process wait for its turn (ie. its number)
  - Read, use, increment, write

Round Robin Synchronization

- Finite number of users who periodically access (round-robin) to the resource
  - The ticket number represents the user
  - Increment <u>modulo</u> the number of users
  - A single ticket in tuples memory
- Waiting for the ticket with its own number
- JavaSpaces implementation:

```
public class CyclicCounter extends Counter {
   public Integer numParticipants;
    ...
```

Synchronization Barrier

- Waiting for all other processes
  - Restart when all others are ready
- Tuple-space implementation: shared counter
  - Initialized to N
  - Decreased when a process reaches the barrier
  - Each process waits until the counter is 0
- Alternative implementation: one input per process
  - Each process write its P_i value,
  - then read all the other $P_{i \neq j}$

Readers / Writers

- Frequent problem, eg. online booking
- Constraints:
  - Multiple read access, single write access
  - All access must be satisfied in a finite time
- Tuple-space implementation:
  - Round-robin synchronization on access

    + A count of the number of readers
    - Reader: waits its turn, increments the ticket, increments the number of readers, reads, decrements the number of readers
    - Writer: waits his turn, waits for no reader, take the ticket, writes, write the ticket

Communications

- Tuple-space communications are weakly coupled
  - No direct interaction between processes
  - No knowledge of the identity and / or location of the correspondent
  - Anonymous communication possible
- Point-to-point or collective communication
- No simultaneous presence required
- Easy mobility support
- Increased fault tolerance

Message Passing

- Simple message

```
public class Message extends MyEntry {
   public String receiver;
   public String content;
```

- Sending

```
Message msg = new Message("fred@ubs","Hello");
space.write(msg, null, Lease.ANY);
```

- Receiving

```
Message tpl= new Message("fred@ubs",null);
Message msg= (Message)
              space.take(tpl, null, Long.MAX_VALUE);
```

- No ordering of messages

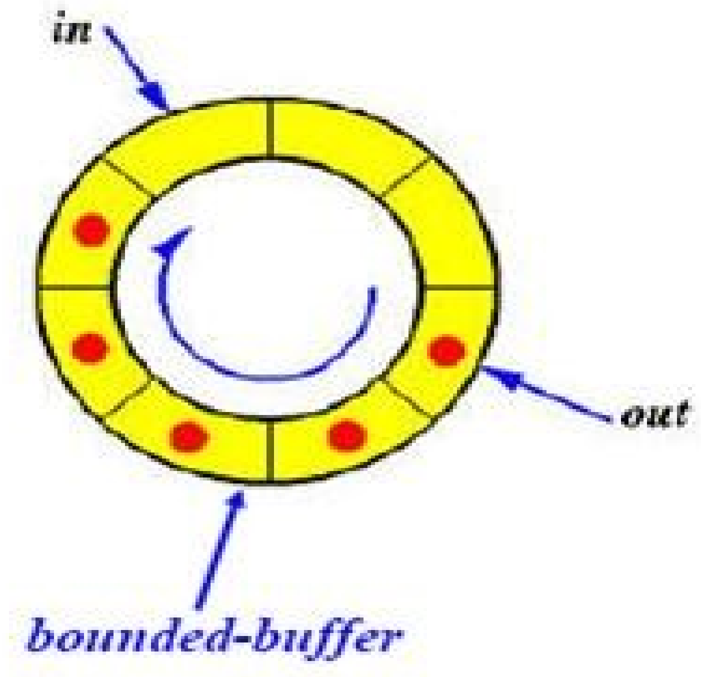
Channel

- Respects the order of messages
  - A message (a tuple) also has a position
- Position Tail in channel
- Writing: reading of Tail, deposit of message no Tail and incrementing Tail

```
public class Tail extends
  MyEntry {
    public String channel;
    public Integer position;
```

- Read: extract the next message
  - The next message is at reader's private position
- Supports multiple readers and multiple editors per channel
- Application: a multi-user chat

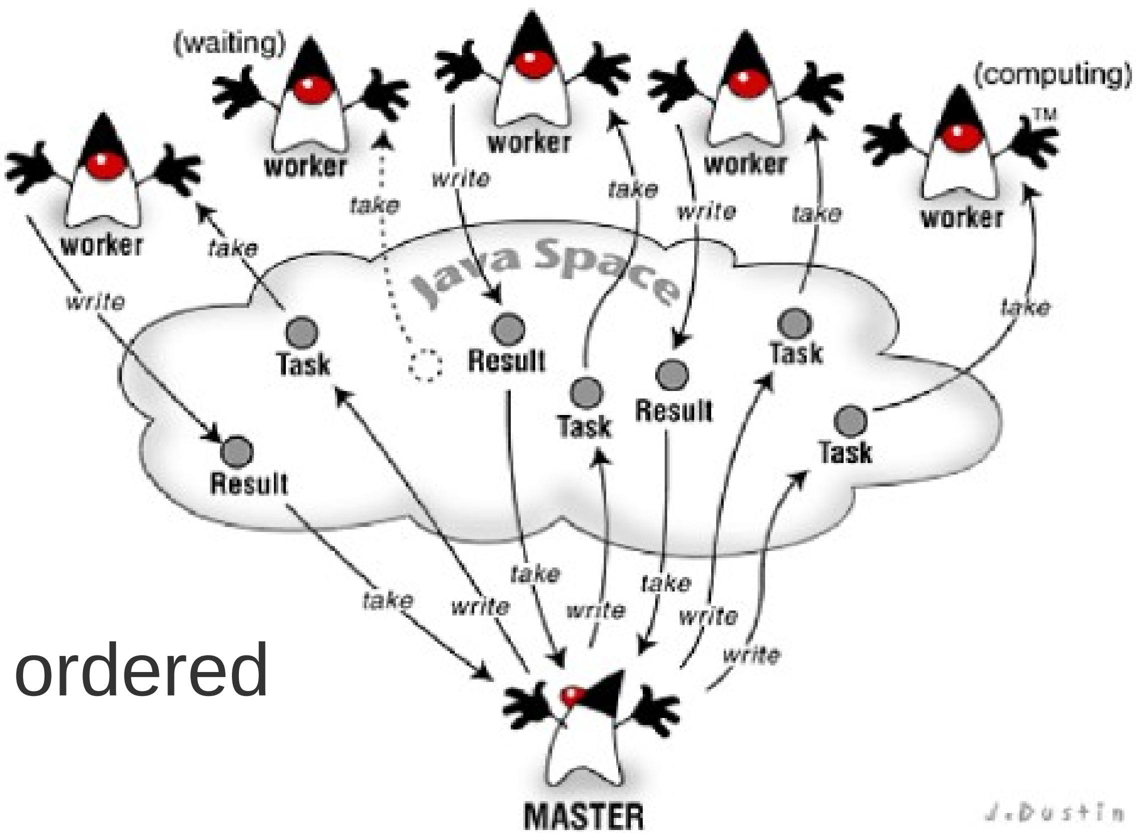
Producers / Consumers



- Queue with a head and a tail
  - Eq. channel with Start and End positions
  - Reading => removal of message from queue (at Start position)
- File bounded to control memory size
  - Adds a Status entry to store the current size
- Application: computing farm

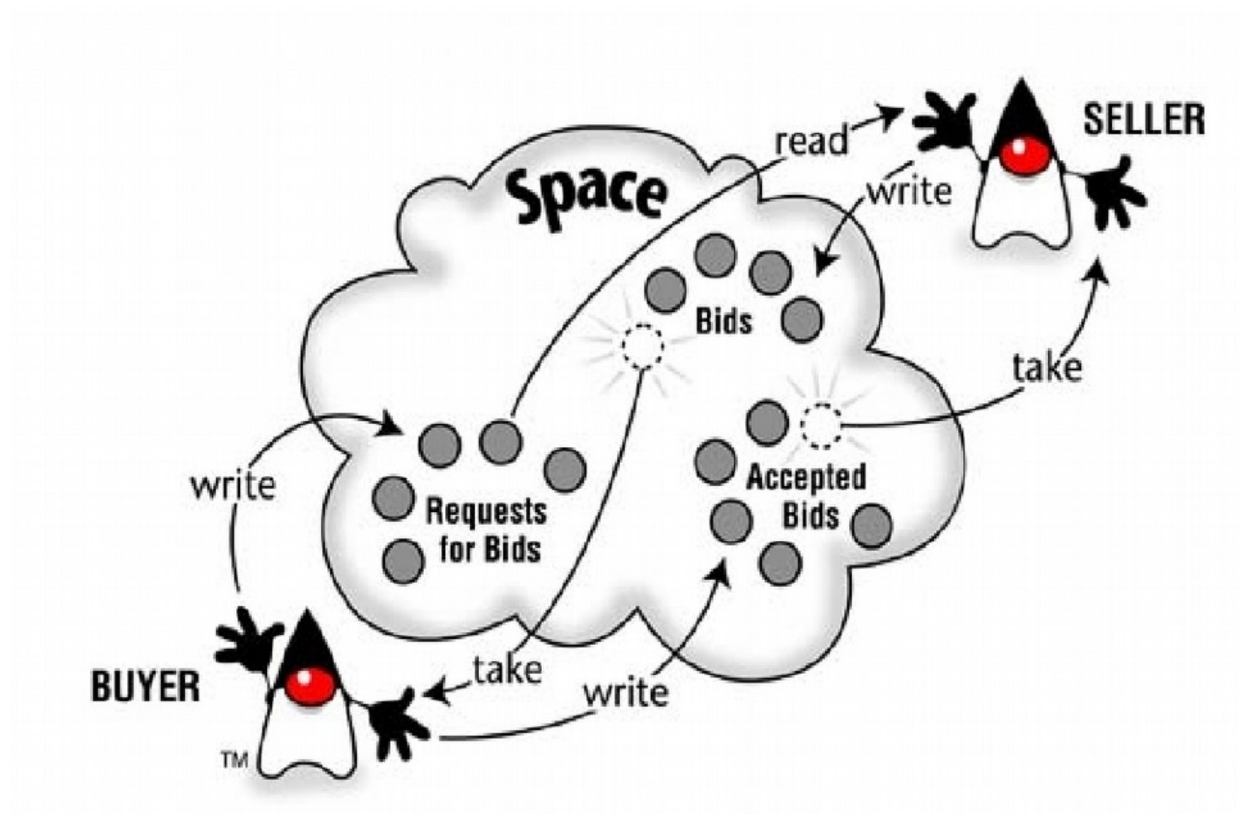
Compute-Farm Model

- Master worker / replicated-worker pattern
- Self-balancing the load
- Independent of the number of machine
  - Tasks of variable size
- Task
  - Part of a calculation
  - Method execute()
- Result
  - Part of a result
- Task and Result can be ordered
  - Queue

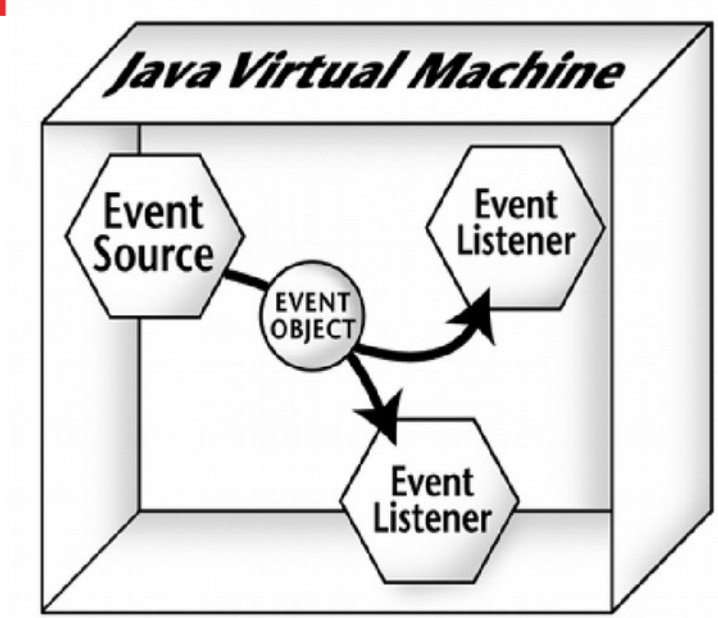


MarketPlace Model

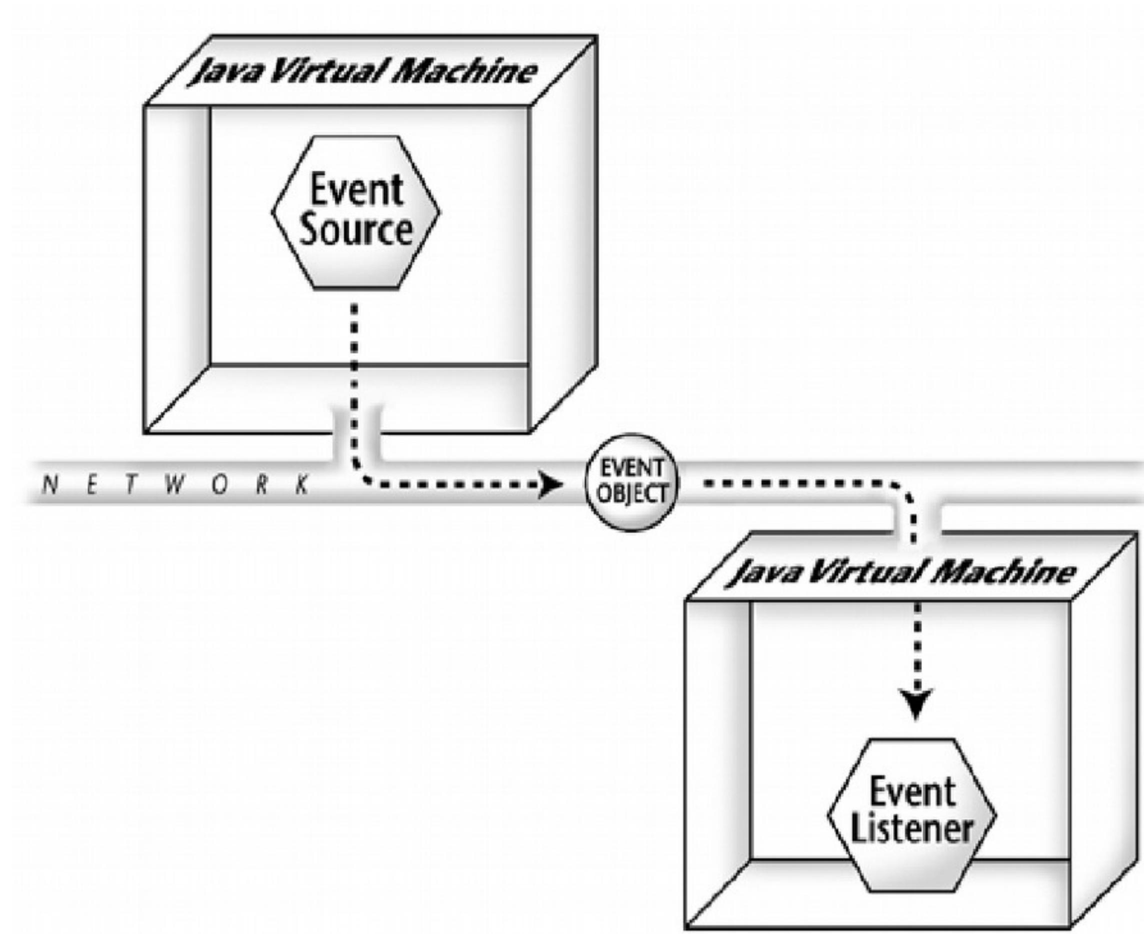
- Buyers are looking for products
- Sellers submit offers
- Buyers evaluate and accept offers
- `BidEntry`
  - `label : ''request''| ''bid'' | ''accepted-bid''`
  - `id`
  - `price`
  - `color, …`



Distributed Events



- Java: programming pattern based on event sources, listeners and event objects



- JavaSpaces uses the event template from Jini through the `notify()` operation

Event-Driven Programming

- Subscription to be notified when an entry corresponding to a template is written in space
- `Method notify()`
  - Record interest for entries that match a pattern
    - Returns a `EventRegistration`
- JavaSpace generates a `RemoteEvent` sent to the `RemoteEventListener` that triggers `notify()`
  - Based on RMI

Example: Writer (bis)

```
public class NotWriter {

    public static void main(String[] args)
        throws Exception {

        Message msg= new Message("Hello") ;
        JavaSpace sp = SpaceAccessor.getSpace();
        sp.write(msg, null, Lease.ANY) ;
        Entry tpl = new Message();
        MessageListener listener=new MessageListener();
        sp.notify(tpl,null,listener,Lease.ANY,null);
    }
}
```

Example: MessageListener

```
public class MessageListener implements
                             RemoteEventListener {
  ...
  public void notify(RemoteEvent ev) throws
                     UnknownEventException,
                      RemoteException {
    Entry tpl = new Message();
    try {
      Message m=(Message)
                sp.read(tpl,null,Long.MAX_VALUE);
      System.out.println("incremented "+m.counter+"
                         times");
      if (m.counter == 10){
             System.exit(0) ;
      }
    } catch (UnusableEntryException e) { …
```

Tuple-Space Implementation

- The Tuple-space paradigm is very convenient for programming distributed applications
- Centralized implementation is fairly easy,
  - e.g. Outrigger implementation is based on
    - Jini for registration, service discovery
    - HTTP for codebase managment
    - RMI for operation calls on a centralized space
  - Poor performance
  - Limited scaling up
- How to manage a physically distributed shared memory of tuples ?

Local Write, Remote Read

- Where to place the tuples (on which node) ?
  - Duplication => consistency problem
  - Unicity => location problem
- A strategy (among others):
  - Each node has its own local tuple space
  - It writes in his local tuple space
  - It searches (read() or take()) in his local space,
  - If not found locally, then it "asks" the others
  - Taking a remote tuple implies to remove it from the remote space

    => no tuple duplication

Abstract Tuples Memory

- A `Tuple` is a <u>list</u> of `Comparable` Objects
  - `null` is a place holder for template
  - `toBytes` and `fromBytes` serialization methods
- A `TupleMemory` is an abstract memory of `Tuple`

```
create(s), distroy(), clean(), size()
read(m), readIfExist(m),
write(t),
take(m), takeIfExists(m)
```

- Implementations :
  - Local memory: `LocalSpace`
  - Global memory: `GlobalSpace`

LocalSpace

- The tuples memory is splitted into lists according to the value of the first field using a `ConcurrentHashMap`
  - A single list of tuples would required a sequential reseach
  - The distribution of the tuples on the value of the first field favors a research of templates relating to the other fields (can be extended to other fields).
- Utility methods (return a tuple)
  - `findAndRemove(m,b)` a tuple in the memory
  - `findAndRemove(m,b,l)` a tuple in one list
  - Removing depends on the boolean parameter b

GlobalSpace

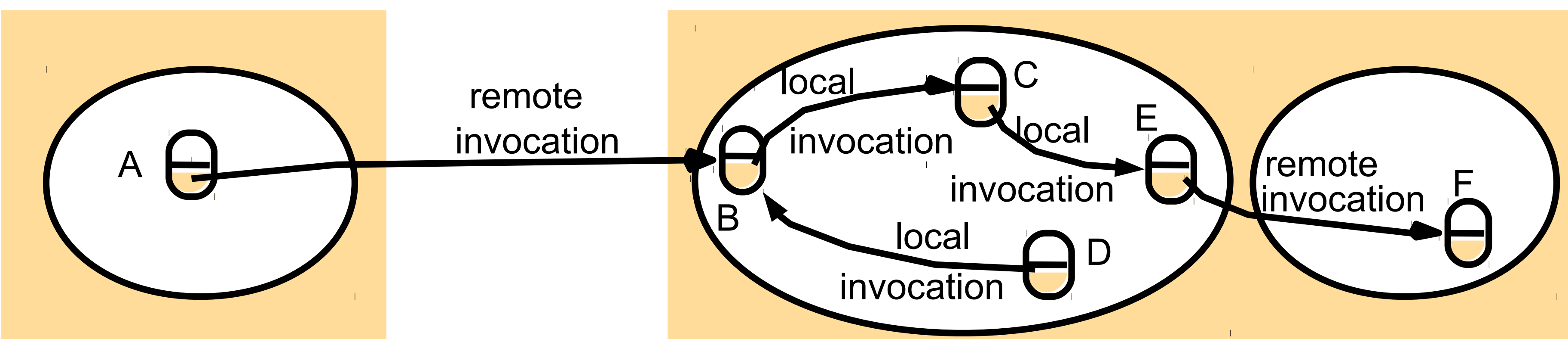
- Extends the `LocalSpace`
- Nodes are sharing their `GlobalSpace`
  - Group of participants managed with ZK (store hostnames)
  - Remote access to the others `GlobalSpace` with RMI
    - `GlobalSpace` and RMIRegistry are launched on nodes
- `GlobalSpace.write` is exactly a `write` in the `LocalSpace`
- `read` and `take` try at first to access a tuple from the `GlobalSpace` with the corresponding non-blocking operations `readIfExist` or `takeIfExist`
  - On success the found tuple is returned
  - Else a repeating access on the GlobalSpace is done until a tuple is found (received after request to other nodes)

Remote Search

- `readIfExists` consists at first in a `readIfExists` in the `LocalSpace`
  - On success the found tuple is returned
  - Otherwise, a `readIfExists` request is made to the `GlobalSpace` of the other nodes
- `takeIfExists` consists at first in a take`IfExists` in the `LocalSpace`
  - On success the found tuple is returned
  - Otherwise, a `takeIfExists` request is made to the `GlobalSpace` of the other nodes
- Tuples obtained from requests populate the `LocalSpace` (and are removed from the remote `GlobalSpace` in case of a `take` operation)

Recalls on Java RMI

- Extends the Java object model
- Same syntax to invoke methods on remote objects as for local ones
- Invoker must handle `RemoteException`
- Target must implement the `Remote` interface
  – Remote interfaces are defined in the Java language
- The semantics of parameter passing differ



Shared Whiteboard Example

- Distributed application example
- Common drawing surface containing graphical objects
- The users draw lines, circles, rectangles,….
- The server keeps a record of all the shapes and provides operations
  - Retrieves the latest shape drawn
  - Assigns a unique version number to each shape
  - Informs client about the version number of each shape

Remote Interfaces

```java
package examples.RMIShape;

import java.rmi.*;

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

```java
package examples.RMIShape;

import java.awt.Rectangle;
import java.awt.Color;
import java.io.Serializable;

public class GraphicalObject
  implements Serializable {

    ...
}
```

```java
package examples.RMIShape;

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

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

Parameter and Result Passing in Java RMI

- Parameters are assumed to be input
- Result is the single output
- Parameters and result of remote methods must implements the `Serializable` interface
- Passing remote objects as remote object reference, example: `Shape newShape(…)`
  - Remote reference received can be used to make RMI calls
- Passing non-remote objects as value, example: `newShape(GraphicalObject)` `GraphicalObject getAllState()`
  - A new object is created <u>in the receiver's process</u>

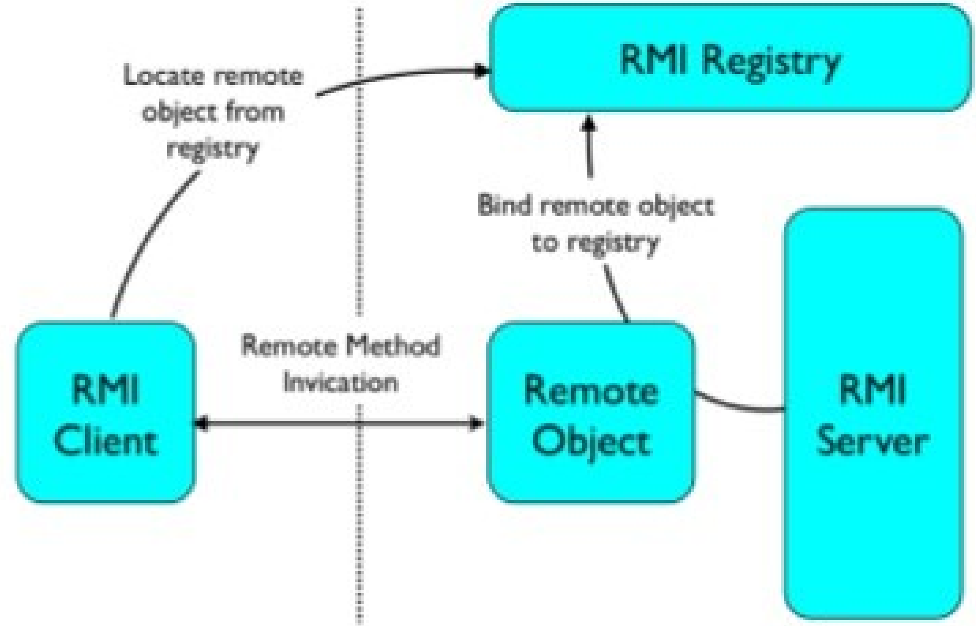
Serialization of Inputs and Output

- Arguments and return values in RMI are serialized to a stream using Java serialization but:
  - Object implementing the `Remote` interface is replaced by its remote object reference
  - Any object is serialized with the location of its class as a URL, enabling the class to be download by the receiver

Downloading of Classes

- Java is design to allow classes to be download from JVM to JVM
- Automatically download the class
  - of an object passed by value
  - Of a proxy for a remote object reference
- Allow clients and server to make transparant use of instances of new classes
  - e.g. passing parameter as a subclass with other variables / methods
  - Propagate the code of the new class to the server and to other clients

RMI Registry



- Binder for Java RMI
- Instance of `RMIregistry` should run on every host of a remote object
  - Maintains a table URL → reference to remote object
  - URL is `//hostName:port/objectName` with `hostName:port` referring to a RMIregistry
  - Accessed by methods of `Naming`
  - Clients must direct their lookup enquiries to a particular host

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 (an interface name)

- **`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)`

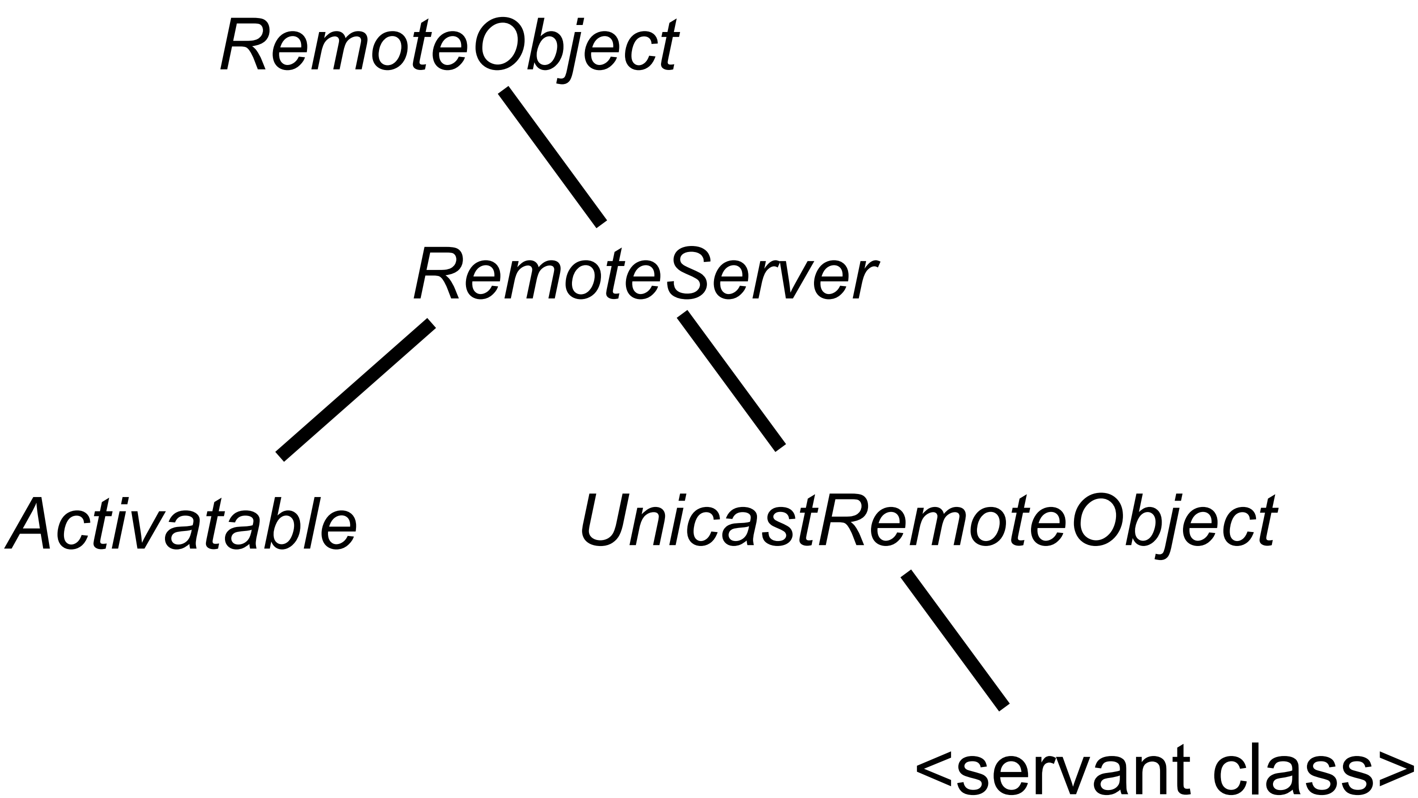
  This method is used by clients to look up a remote object by name

- `String [] list()`

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

Classes supporting RMI

- The programmer is concerned with only one class



Building Server Program

- Whiteboard server example
- The server consists of a `main` and represents
  - Each shape as a remote object instanciated by a servant `ShapeServant` that
    - Implements the `Shape` interface
    - Hold the state of a graphical object
  - The collection of shapes as a remote object instanciated by a servant `ShapeListServant` that
    - Implements the `ShapeList` interface
    - Hold a collection of shapes in a `Vector`

ShapeListServer class

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer {
     public static void main(String args[]){
        System.setSecurityManager(new SecurityManager()); // ~/.java.policy
      try{
           ShapeList aShapelist = new ShapeListServant();
           ShapeList servant =
                 (ShapeList)  UnicastRemoteObject.exportObject(aShapeList,0);
                                 // or Activatable instead of UnicastRemoteObject
           Naming.rebind("ShapeList", servant);
           System.out.println("ShapeList server ready");
       }catch(Exception e) {
           System.out.println("ShapeList server main " + e.getMessage());
       }
    }
}
```

ShapeListServant class

```
package examples.RMIShape;
import java.util.Vector;
public class ShapeListServant implements ShapeList{
   private Vector theList;
   private int version;
   public ShapeListServant() throws RemoteException{...}
   public Shape newShape(GraphicalObject g)
            throws RemoteException{
     version++;
      Shape s = new ShapeServant( g, version);
      theList.addElement(s);
      return s;
    }
    public  Vector allShapes() throws RemoteException{... }
    public int getVersion() throws RemoteException{ ... }
}
```

Building a Client Program

- Any client needs to get a binder to look up a remote object reference
- Then it sends RMIs to that object or to others discovered during its execution
- If it was implementing a whiteboard display it would use the server's `getAllState()` to retrieve each of the graphical objects and display them in a window
- Each time the user finishes drawing a graphical object it will invokes `newShape()` in the server
- It keeps a record of the later version number at the server
- From time to time (pooling) it will invoke getVersion() to find out any new shapes added by other users

ShapeListClient class

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

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

Callbacks

- Avoid the need for a client to pool the object of interest in the server
  - The server should notify its clients whenever an event occurs
- The client creates a remote object (named callback object) that implements a remote interface with a method for the server to call

```
public interface WhiteBoardCallBack extends Remote {
  void callBack(int version) throws RemoteException;
}
```

- The server provides an operation to register interested client and records these in a list

```
int register(WhiteboardCallBack callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

- When an event of interest occurs the server calls the clients in the list