

# Concurrent and Distributed Systems

Talk Seven

Distributed Programming with MPI  
&  
Tuple Space Model

**Maxim Mozgovoy**

# MPI in Clusters

MPI works **equally well** in **multicore** (multiprocessor) and **network cluster** environments. In other words, you can run any MPI-based program we considered before **on a cluster**.

MPJ Express knows how to send messages between the computers in a network.

We will not perform real experiments with computational clusters (we don't have them, unfortunately). Instead, we will discuss some of network-related problems using the same multicore mode.

# **(Just in Case) What Is a Cluster?**

A cluster is a local network of computers that work together on one computational task. Normally, clusters are homogenous (the computers have similar characteristics).

In a sense, a cluster is a “virtual supercomputer” with each computer acting as a single processor.

Cluster is not the only way to organize a distributed system...

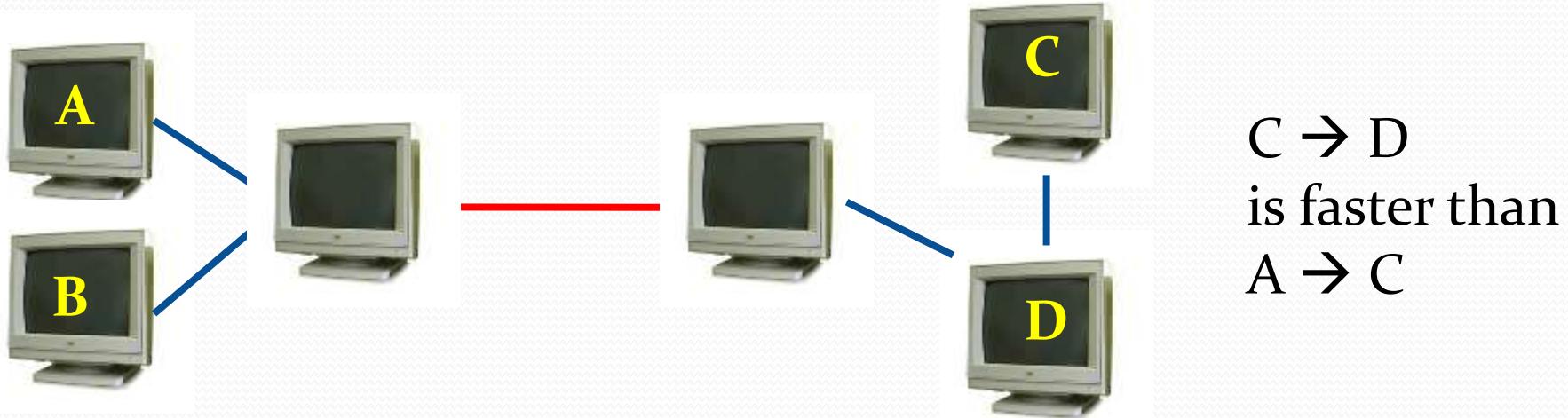
# Other Options?

- **Client-server**: a network with several *clients* that post the tasks and one *server* that executes the tasks (e.g., a database or a web server).
- **Grid / Cloud**: a decentralized heterogeneous network. The computers work together on one task, but they are independent and can be located far away from each other.
- **P2P**: a decentralized heterogeneous network of independent computers, directly communicating with each other. Mostly used for file sharing (BitTorrent) and messaging (Skype before 2012, FireChat, Tox).

So, a cluster is quite a simple model: all computers act together, they are similar and located close to each other. Also, normally there is no resource (files, printers, etc.) sharing.

# Topology and Latency

In networking, *topology* (i.e., network structure) is important. Some computers are directly connected, and some use computers in between, which means longer time for message delivery and possible bandwidth problems.



Also, we cannot ignore *latency* (message delivery takes time).

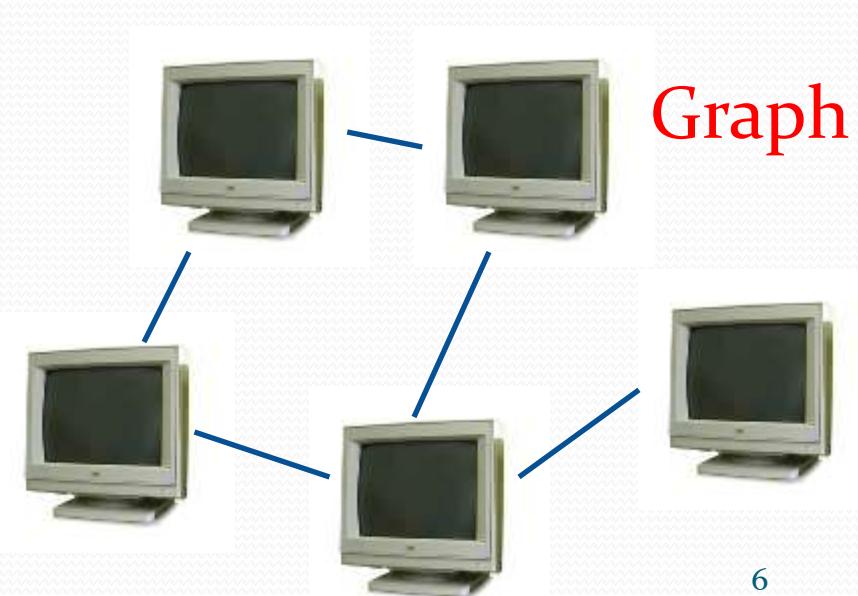
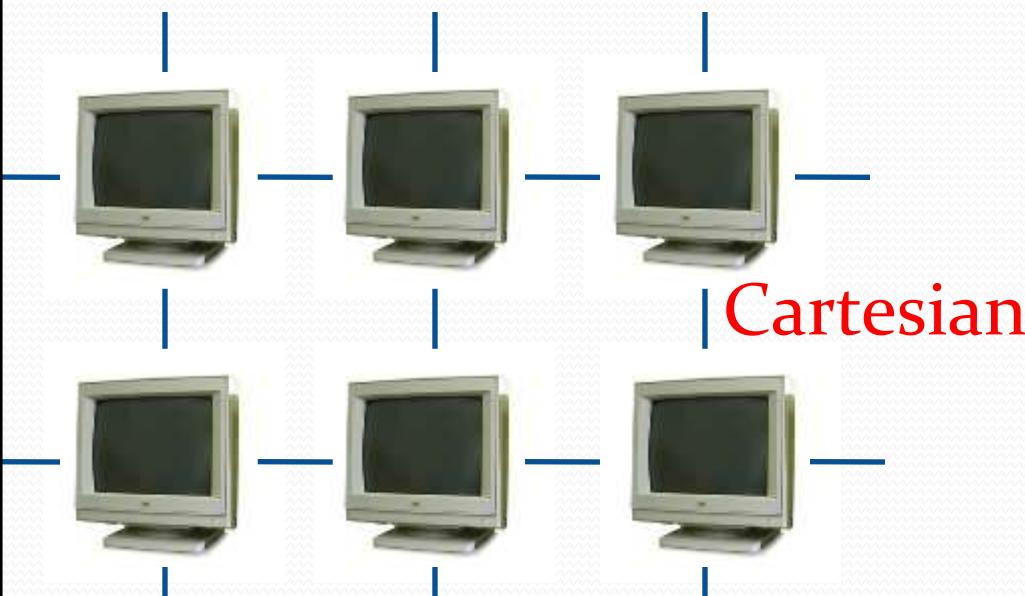
In multicore environments these problems are less important.

# Virtual Topology in MPI

MPI allows to organize a **virtual topology** by describing which computers in the network are directly connected.

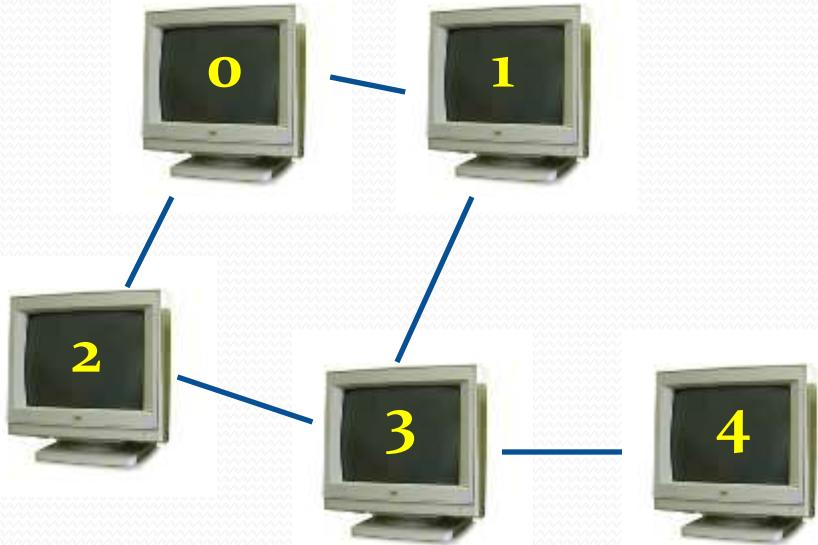
Using this knowledge, MPI can optimize message passing for your network, and you can take into account topology while designing algorithms. In case of a real cluster, this topology can be matched with the real topology of the network.

MPI supports two kinds of topologies: **Cartesian** and **Graph**.



# How to Define Topology in MPI (1)

(Let's consider a more general *graph* topology):



Prepare arrays *index* and *edges*.  
*index*[*i*]: total number of neighbors  
of the first *i* nodes.

*edges*: just a list of all edges  
(node by node)

**procID**   **neighbors**   **nbrsSum**

0	1, 2	2
1	0, 3	4
2	0, 3	6
3	1, 2, 4	9
4	3	10

$$\text{index} = \{ 2, 4, 6, 9, 10 \}$$

$$\text{edges} = \{ 1, 2, 0, 3, 0, 3, 1, 2, 4, 3 \}$$

# How to Define Topology in MPI (2)

Call [Create\\_graph\(\)](#):

```
Graphcomm gc =  
MPI.COMM_WORLD.Create_graph(index, edges,  
                                false);  
(defines whether process IDs can be reordered by the system).
```

Then you can use `gc` instead of `MPI.COMM_WORLD`:

```
gc.Send(...);  
gc.Recv(...);
```

Cartesian topology is created  
with a [Create\\_cart\(\)](#) call

You can also obtain a list of neighbors of any process in the system!

```
int [] gc.Neighbours(int rank)
```

# Blocking vs. Nonblocking Operations (1)

(Also called Synchronous / Asynchronous operations)

In networks, send / receive operations may take much time.

We can optimize algorithms if we do some useful work while sending / receiving data.

MPI supports this idea by providing **nonblocking** send and receive functions: **Irecv()**, **Irecv()**.

# Blocking vs. Nonblocking Operations (2)

Ordinary `Send()` / `Recv()` functions wait until message passing has finished.

`Irecv()` / `Irecv()` initiate data transfer, and return control to you immediately:

```
MPI.COMM_WORLD.Send(arr, 0, N, MPI.INT, 1, 0);  
doSomething(); // here arr is already sent  
...
```

```
MPI.COMM_WORLD.Isend(arr, 0, N, MPI.INT, 1, 0);  
doSomething(); // here arr is not yet sent  
// (it is still being transferred)
```

# Blocking vs. Nonblocking Operations (3)

To make sure that the data has been fully transferred, you can use `Wait()` function as follows:

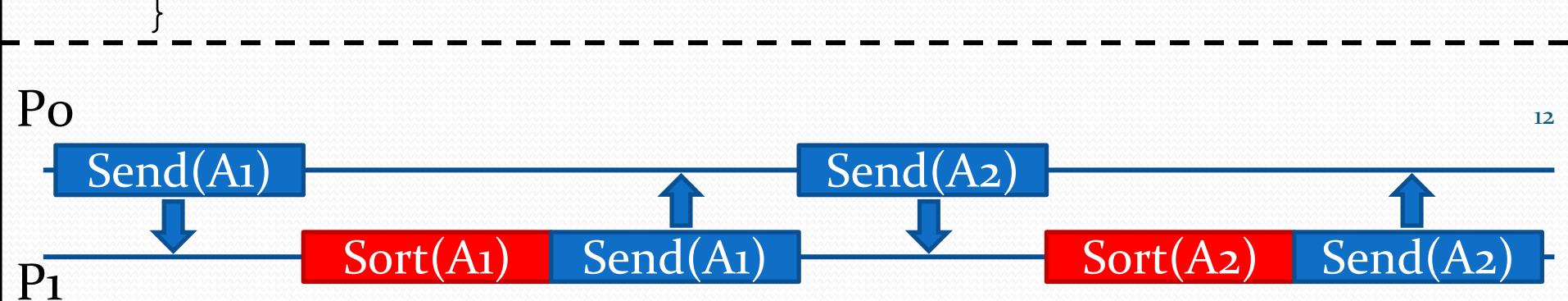
```
Request r =  
    MPI.COMM_WORLD.Isend(arr, 0, N, MPI.INT, 1, 0);  
r.Wait(); // wait for Isend() operation to complete  
  
// same for Irecv():  
Request r = MPI.COMM_WORLD.Irecv(...);  
r.Wait();
```

Nonblocking operations are **harder to use**, but they can provide better performance, if applied **correctly** and **carefully**.

# Nonblocking Operations Example (1)

**P0:** for(int i = 0; i < IT; ++i) {  
    send a block of N integers to P1  
    receive N integers from P1  
    print received data  
}

**P1:** for(int i = 0; i < IT; ++i) {  
    get an array of N integers from P0  
    sort this array  
    send the sorted array back to P0  
}



# Nonblocking Operations Example (2)

P1 spends much time for sorting the array, so we can optimize P1 as follows:

...

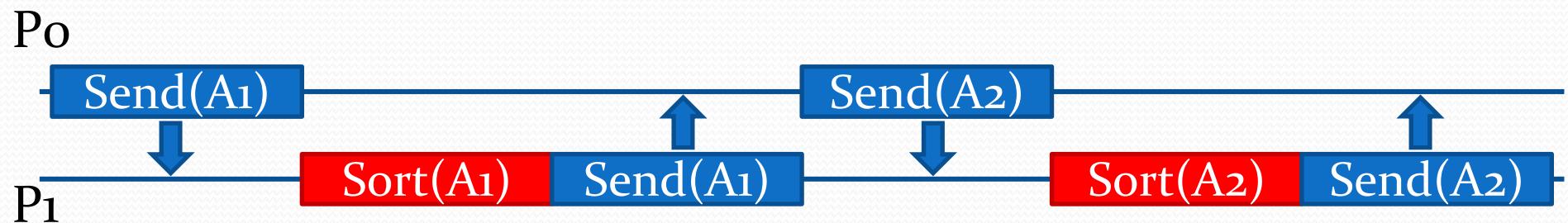
get the array (A1) of N integers from P0  
asynchronously request the next array (A2)  
while (A2) is being transferred, sort array A1

...

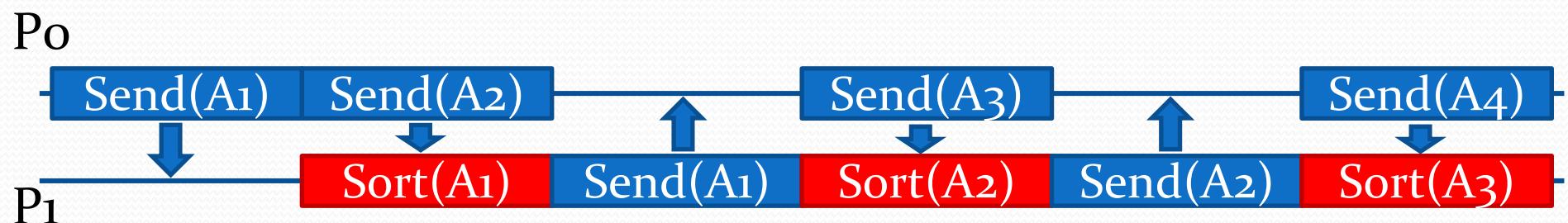
(study `w07_asyncSend.java` file and compare it  
with blocking Send/Recv-based `w07_syncSend.java`)

# Nonblocking Operations Example (3)

Send/Recv array sorting:



Irecv/Isend array sorting:



# Shared Memory in a Distributed System?

Passing messages in a distributed environment is **natural**: networked computers don't have common memory.

In some applications it is **convenient to pretend** that there is a shared memory block, and all processes can access it.

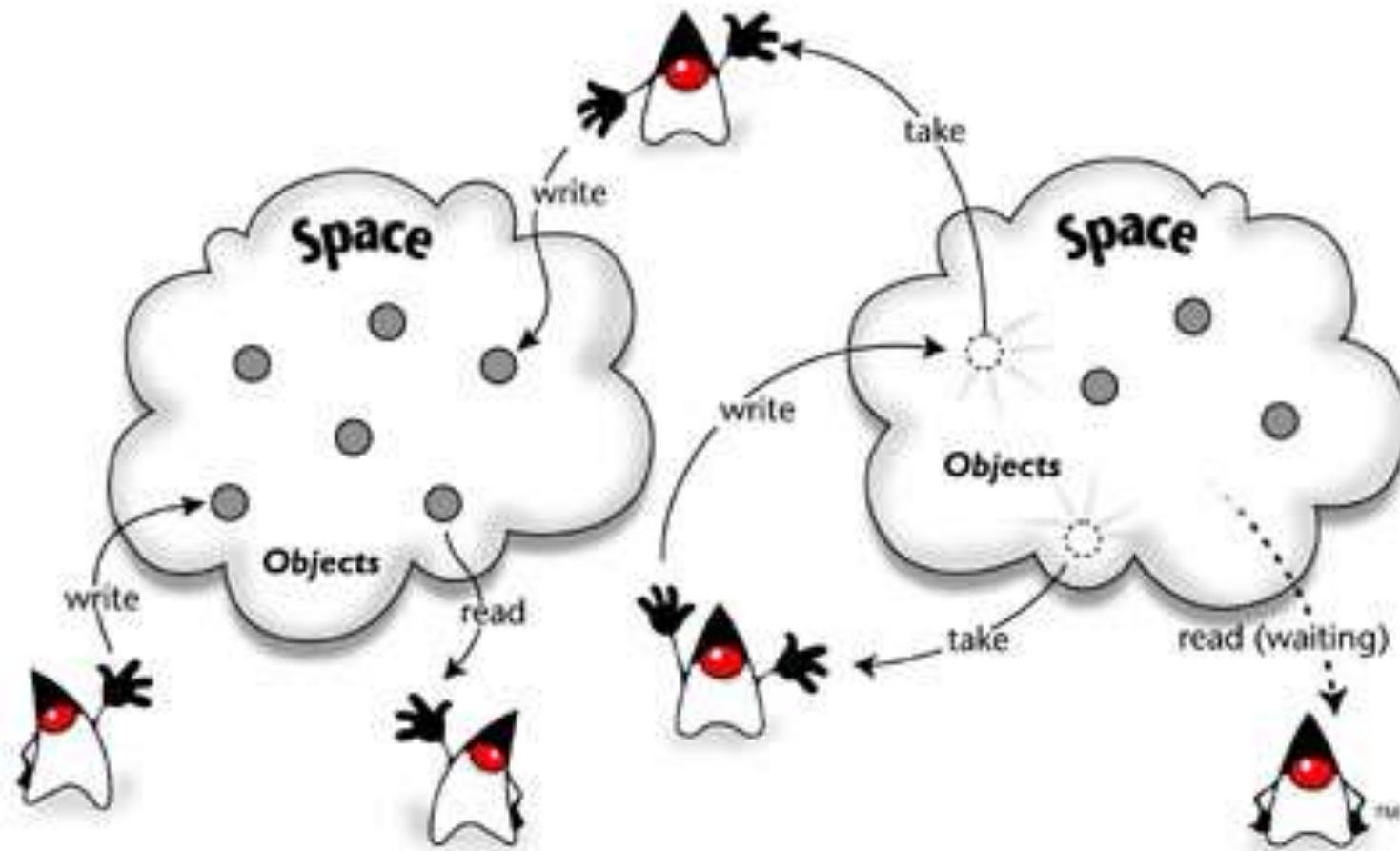
A well-known model: [\(Linda\) Tuple Space](#). Implemented in Linda language (1991). Available for other platforms, incl. Java.

**Idea**: any process can read and write objects, located on a shared blackboard (“space”)!

# Space-based (Virtual Shared Memory)

Independent processes share a **common repository** (memory) of objects and communicate by storing and retrieving them.

There is a (semi-)standard Java implementation: **JavaSpaces**.



# JavaSpaces (1)

A process can **read** or **take** (read and remove) an object from the space. It will **wait** until the object is available, if it is not there yet. A process can also **write** an object to the space.

Note the part

*“It will **wait** until the object is available, if it is not there yet”.*

This is another version of a synchronization mechanism, similar to **semaphores** or MPI synchronous **Recv()** operation.

# JavaSpaces (2)

JavaSpaces is a part of Jini project, now known as Apache River:  
<http://river.apache.org/>

# Bill Joy [chief Jini architect] on JavaSpaces:

*“a quantum leap in thinking”* (Naturally, it didn’t work)

Still a nice technology if you need shared space.

Not very popular, but used in certain areas (e.g., financial soft).

## Criticisms:

Low speed, low-level concepts, single space bottleneck.

# MozartSpaces (1)

Apache River is quite difficult to setup, so we will experiment with a simpler project called **MozartSpaces**:

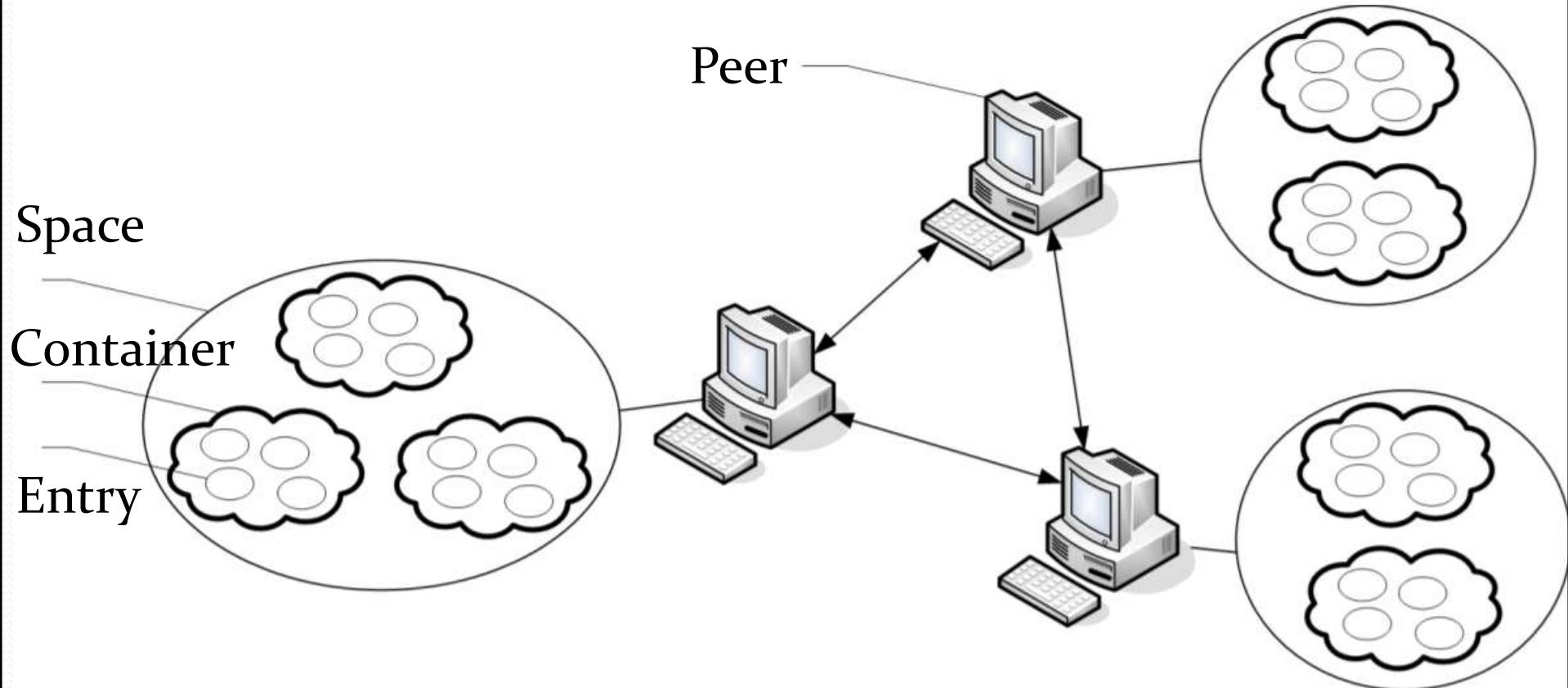
<http://www.mozartspaces.org>

In MozartSpaces, every peer (process) can create its own **shared** tuple **spaces**. You can create **containers** inside spaces, and store **entries** inside containers.

We will consider the bare minimum necessary to obtain at least some basic results with MozartSpaces.

The complete functionality of this project is quite rich.

# MozartSpaces (2)



# MozartSpaces: Setup

To use MozartSpaces types, add to you code:

```
import org.mozartspace.core.*;  
import org.mozartspace.capi3.*;  
import java.net.URI;
```

Copy provided logback.xml file to your project folder.

Compile and run your code with MozartSpaces jar file:

```
// *nix systems  
javac -cp .:mozartspace-2.3.jar <java-file>  
java -cp .:mozartspace-2.3.jar <class-file>  
  
// MS Windows  
javac -cp .;mozartspace-2.3.jar <java-file>  
java -cp .;mozartspace-2.3.jar <class-file>
```

# MozartSpaces: Cores and Spaces

Each process must have a **core**. A core can be associated with its own **space** if necessary (but not required).

The process will shutdown only after you shutdown its core:

```
// core without a space
MzsCore core =
    DefaultMzsCore.newInstanceWithoutSpace();
// core with a space
MzsCore core = DefaultMzsCore.newInstance();
// must shutdown a core to finish the process
core.shutdown(true);
```

# MozartSpaces: Creating Containers

Most interactions with spaces are conducted via a Capi (“**core API**”) object associated with a core:

```
Capi capi = new Capi(core);
```

To create a container for entities, use `createContainer()` function of Capi:

```
ContainerReference cont =
    capi.createContainer("ContainerName", null,
MzsConstants.Container.UNBOUNDED,
null, null, null);
```

# MozartSpaces: Using Existing Containers

You can also connect to an existing container located remotely.  
To do it, use `lookupContainer()`

```
Capi capi = new Capi(core);  
ContainerReference cont =  
capi.lookupContainer("ContainerName",  
URI.create("xvsm://localhost:9876"),  
MzsConstants.RequestTimeout.DEFAULT, null);
```

The second argument (URI) is an address of a **remote space** in the network. If it is located on your computer, use `xvsm://localhost:9876`

# MozartSpaces: Writing Entries

An “entry” is a value represented with a standard built-in Java type, such as an integer or a string. When you write an entry to a container, it is **added** there (old entries are kept):

```
ContainerReference cont = ...          // container  
capi.write(cont, new Entry(5));        // integer  
capi.write(cont, new Entry("hello")); // string
```

# MozartSpaces: Reading Entries (1)

The “read” operation returns all entries stored in the specified container:

```
ContainerReference cont = ...           // container
ArrayList<Serializable> entries =
                                         capi.read(cont);
```

Then you can process individual entries:

```
System.out.println(entries.get(0));
System.out.println(entries.get(1));
```

**Note:** the process will be blocked if entries are not available!

# MozartSpaces: Reading Entries (2)

When reading entries, you can specify additional options.  
For example, suppose you want to wait infinitively long  
until the container `fiveCont` contains 5 elements:

```
ContainerReference fiveCont = ...          // container  
  
entries = capi.read(fiveCont,  
                     AnyCoordinator.newSelector(5),  
                     MzsConstants.RequestTimeout.INFINITE,  
                     null);
```

**Note:** use this technique explicitly when you need to wait  
for an object to appear!

# MozartSpaces: Taking Entries

Instead of “read” operation you can use “take”, which reads the values of the entries and **removes** them from the container:

```
ArrayList<Integer> entries =  
    capi.take(myContainer);
```

Other useful features:

- Custom entry datatypes (can store user objects);
- Key/value storage: retrieve entries by key.

# Master-Worker (1)

Typical workflow in space-based systems is **master-worker** (also known as **producer-consumer**).

**Master:** prepare “units of work”  
and publish them in the repository.

**Worker:** get a unit of work from the repository,  
process it, then put back the result.

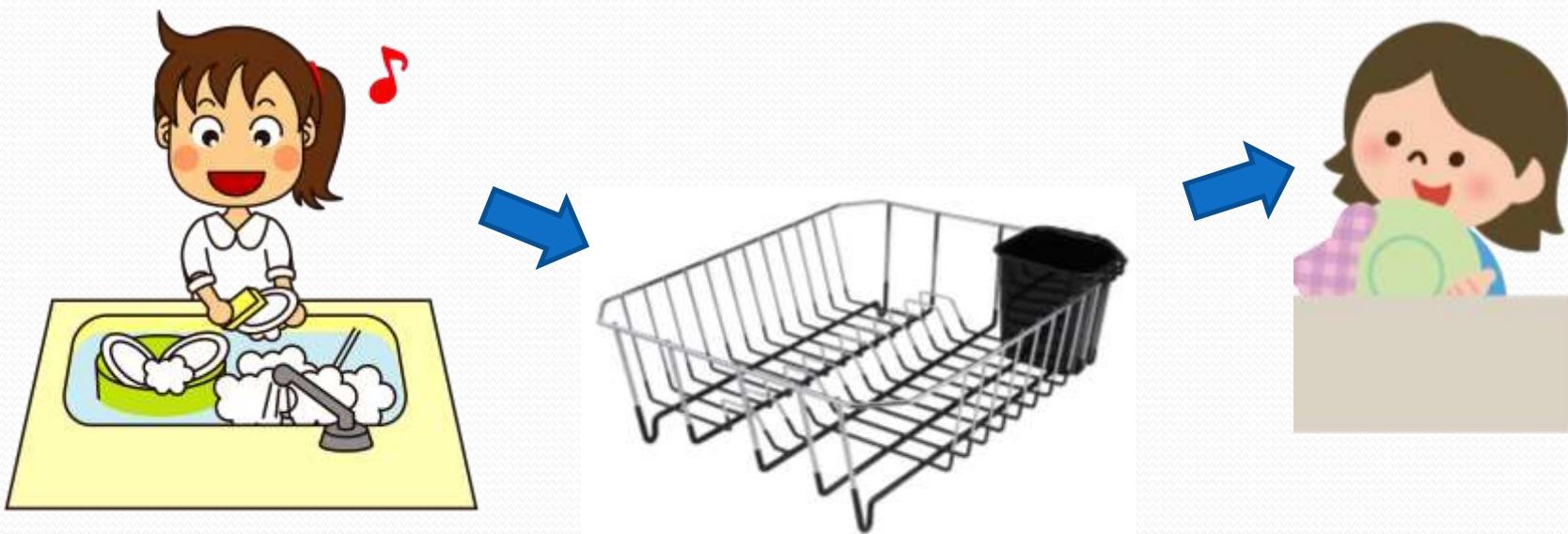
Ex.: we need to download all **PDF** files from some website.

The **master** process analyzes the website structure and publishes all found links to PDFs in the repository.

Each **worker** connects to the repository, takes the first available link and starts to download the file.

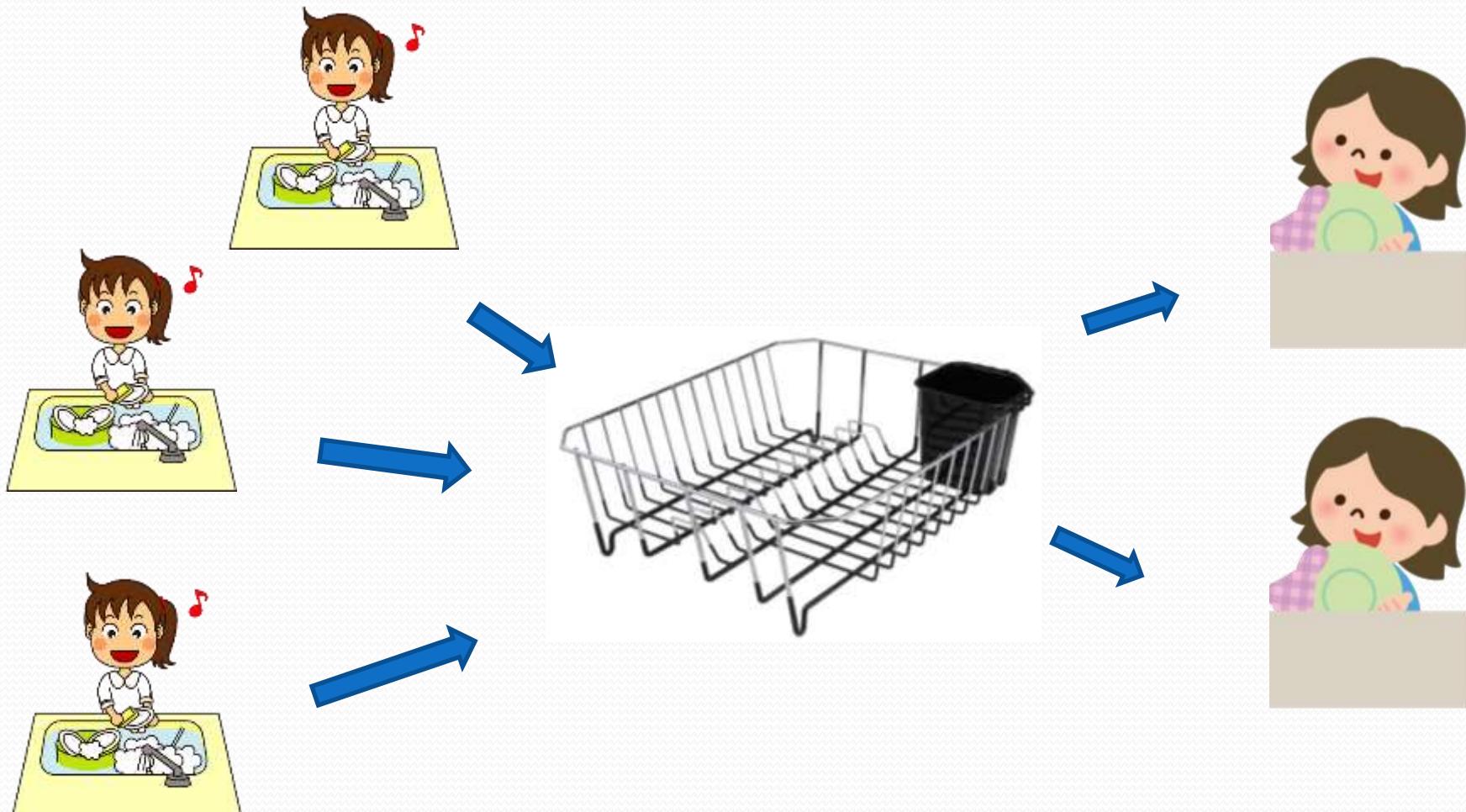
# Master-Worker (2)

A dish-washing analogy is often used. Suppose one person (producer) washes the dishes and puts them into a dish rack, while another person (consumer) dries the dishes from the rack and places them into a shelf.



# Master-Worker (3)

One attractive feature of this design is easy generalization.  
It is not hard to extend the system for the case of multiple  
producers and/or consumers.



# Conclusions

- MPI can be used both in multiprocessor (multicore) environments and in **computational clusters**.
- A **cluster** is a network of computers acting as a **single** computational device.
- Programs that execute on clusters should be aware of **topology** (how the computers are physically connected) and **latency** (passing data takes time).
- MPI helps us to experiment with different topologies by introducing **virtual topologies**.
- It is possible to speed up some algorithms by using **nonblocking** (asynchronous) send/receive operations: while data is being transferred, processes do something else.
- It is possible to **simulate distributed memory** (e.g., with JavaSpaces or MozartSpaces).