

# *OO SPMD for the Grid: an alternative to MPI and the Road to Components*

*WP 3, Oct. 14<sup>th</sup> 2005*

- 1. Asynchrony*
- 2. Group*
- 3. OO SPMD*



# Main MPI problems for the Grid

Too static in design

Too complex interface (API)

- More than 200 primitives and 80 constants

Too many specific primitives to be adaptive

- Send, Bsend, Rsend, Ssend, Ibsend, etc.

Typeless (message passing)

Manual management of complex data structures



# MPI Communication primitives

For some (historical) reasons, MPI has many com. Primitives:

MPI_Send	Std	MPI_Recv	Receive
MPI_Ssend	Synchronous	MPI_Irecv	Immediate
MPI_Bsend	Buffer	... (any) source, (any) tag,	
MPI_Rsend	Ready		
MPI_Isend	Immediate, async/future		
MPI_Ibsend, ...			

**I'd rather put the burden on the implementation, not the Programmers !**

**How to do adaptive implementation in that context ?**

Not talking about:

- the combinatory that occurs between send and receive
- the semantic problems that occur in distributed implementations

**Is Recv at all needed ? First adaptive feature: Dynamic Control Flow of Mess.**



# Main MPI problems for the GRID

Too static in design

Too complex in Interface (API)

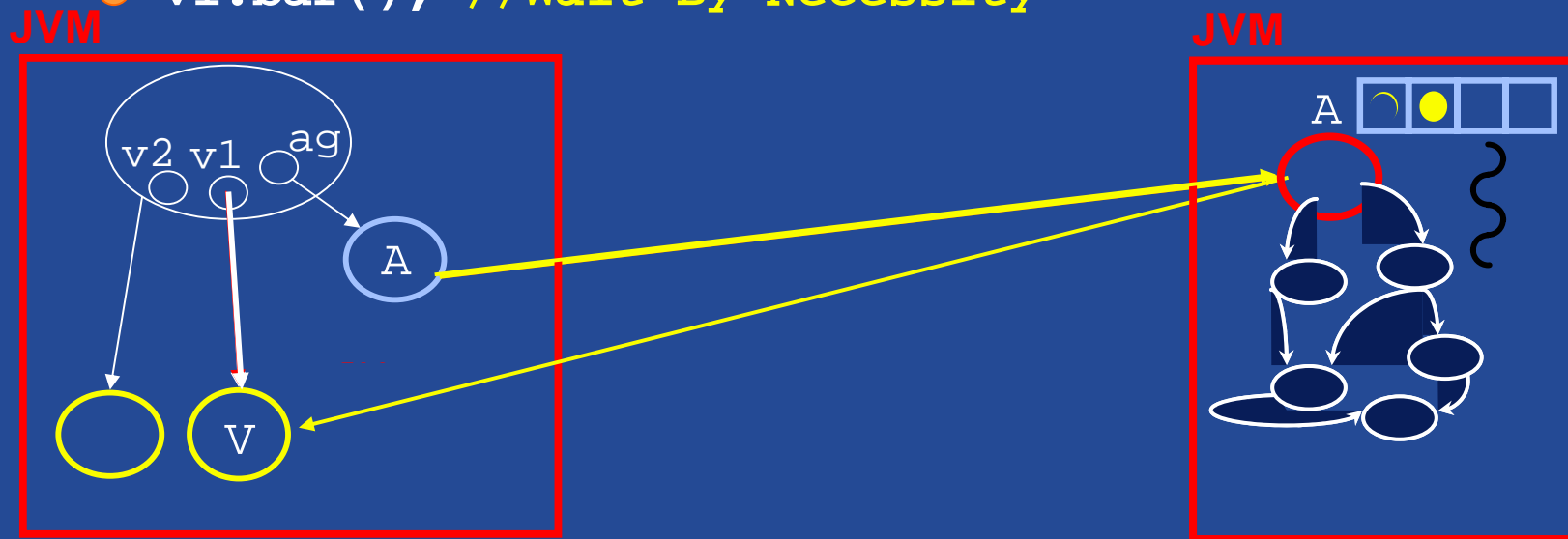
Too many specific primitives to be adaptive

Type Less



# ProActive : Active objects

- A ag = `newActive ("A", [...], VirtualNode)`
- V v1 = ag.foo (param);
- V v2 = ag.bar (param);
- ...
- v1.bar(); //Wait-By-Necessity

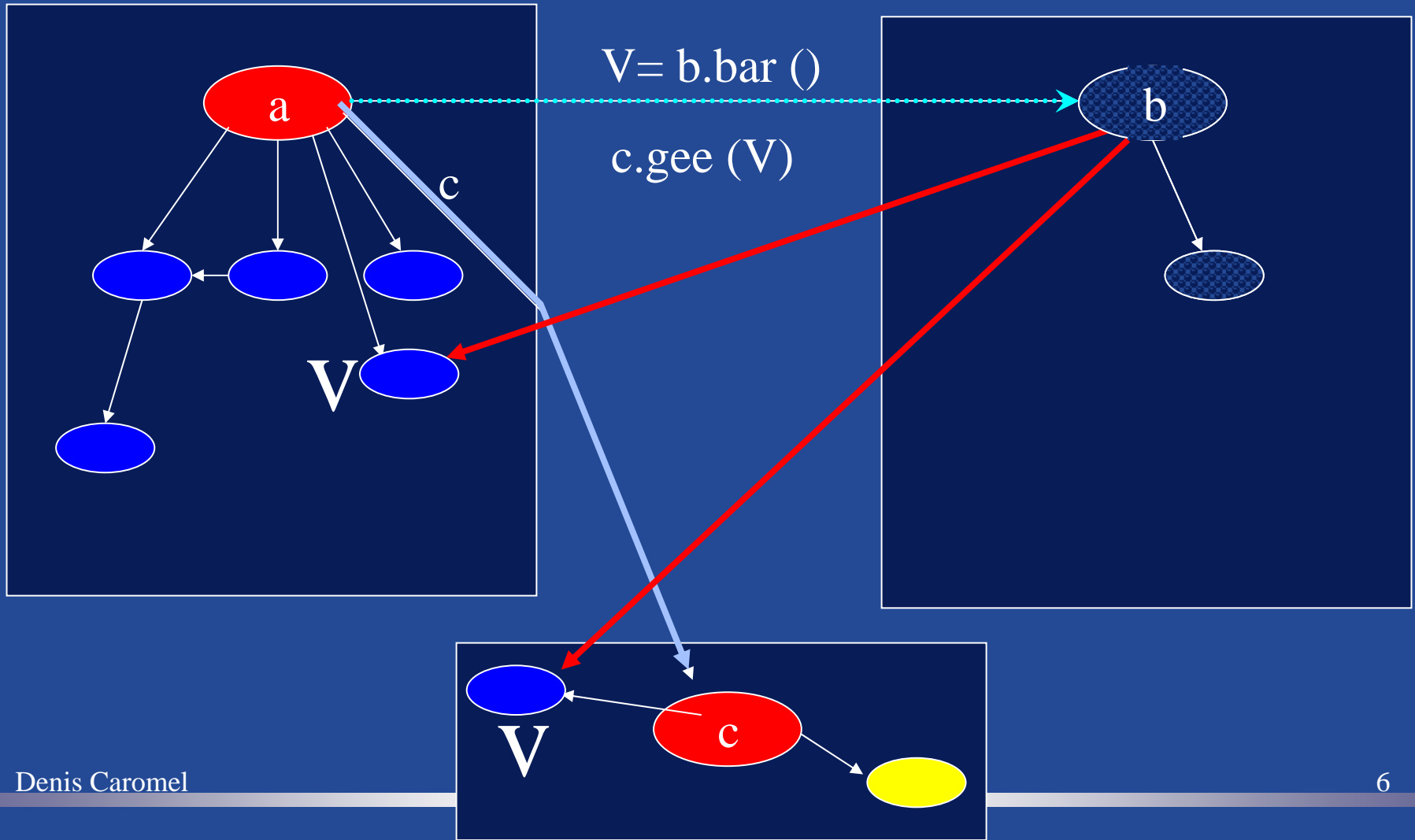


**Wait-By-Necessity**  
is a  
**Dataflow**  
**Synchronization**



# Wait-By-Necessity: First Class Futures

Futures are Global Single-Assignment Variables



---

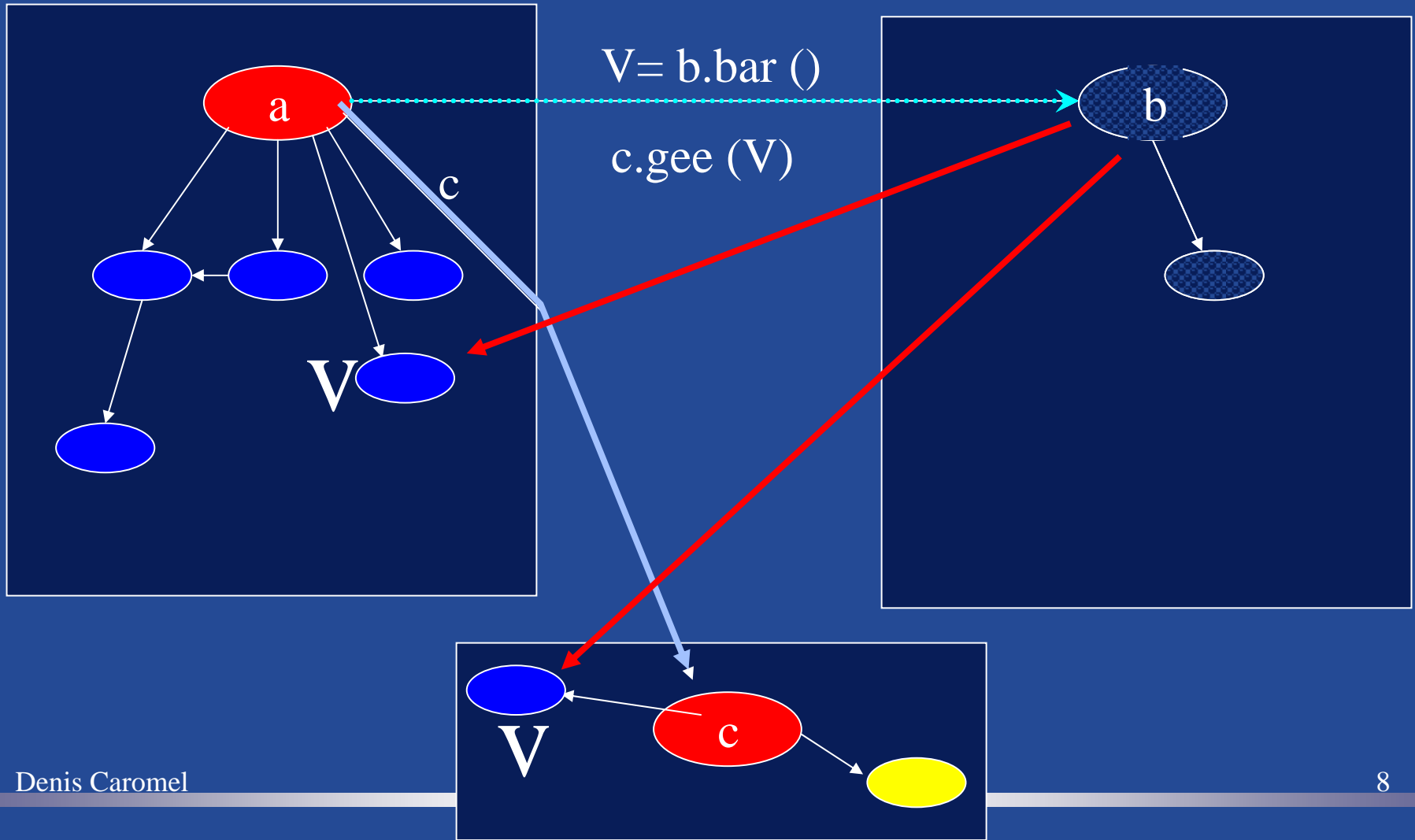
# First-Class Futures

## Update



# Wait-By-Necessity: First Class Futures

Futures are Global Single-Assignment Variables



# Future update strategies

No partial replies and requests:

- No passing of futures between activities, more deadlocks

Eager strategies: as soon as a future is computed

- Forward-based:
  - Each activity is responsible for updating the values of futures it has forwarded
- Message-based:
  - Each forwarding of future generates a message sent to the computing activity
  - The computing activity is responsible for sending the value to all

Mixed strategy:

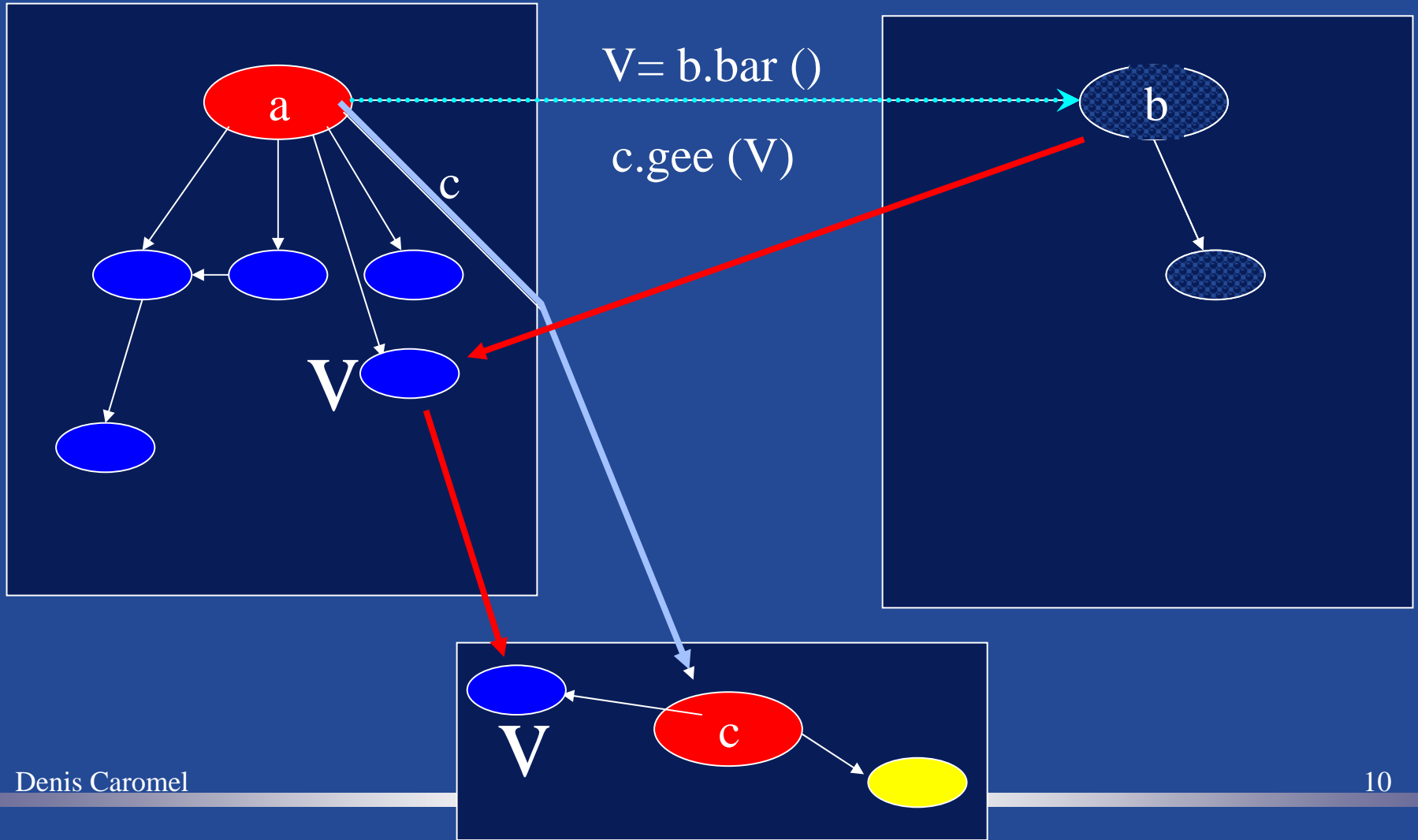
- Futures update any time between future computation and WbN

Lazy strategy:

- On demand, only when the value of the future is needed (WbN on it)

# Wait-By-Necessity: Eager Forward Based

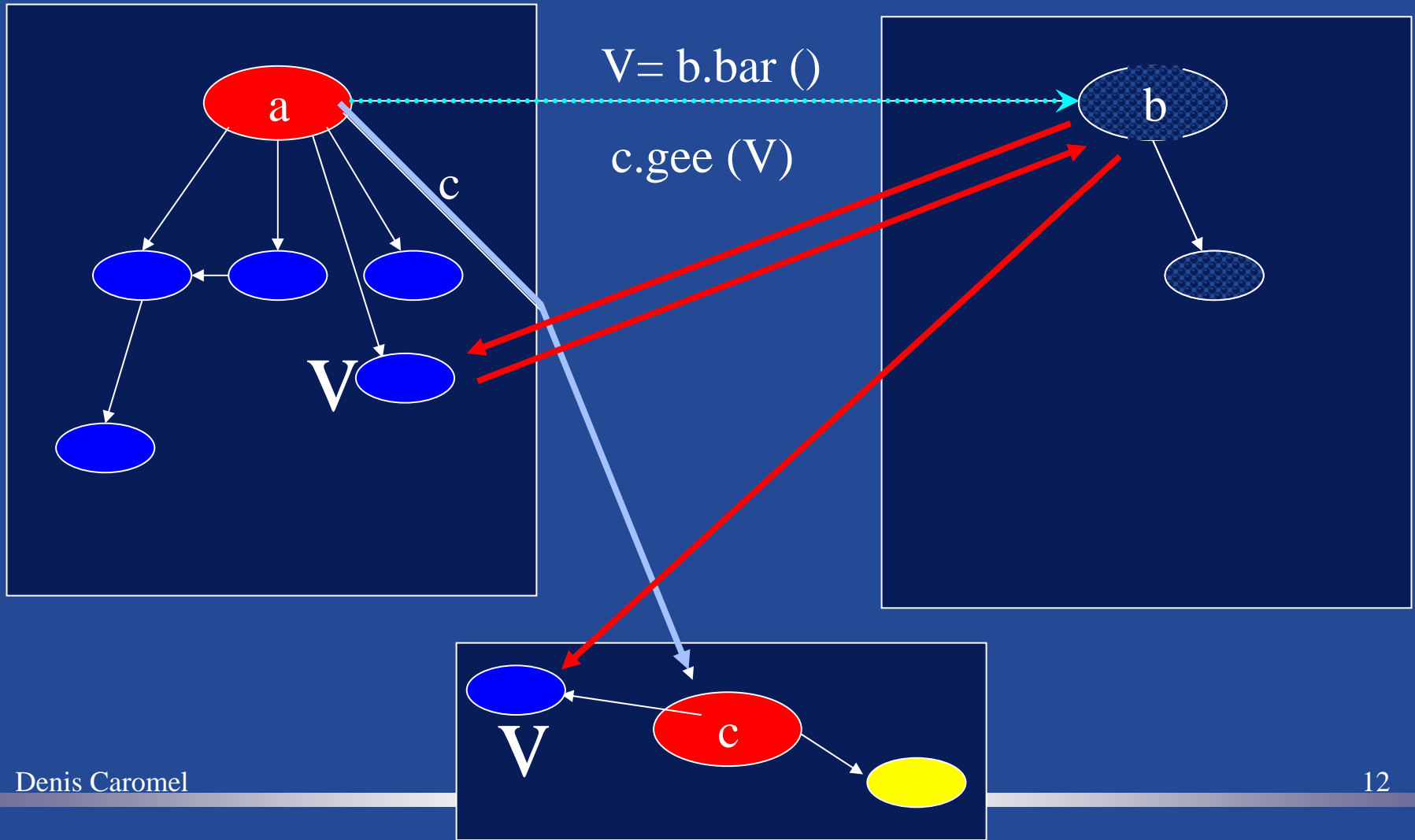
AO forwarding a future: will have to forward its value





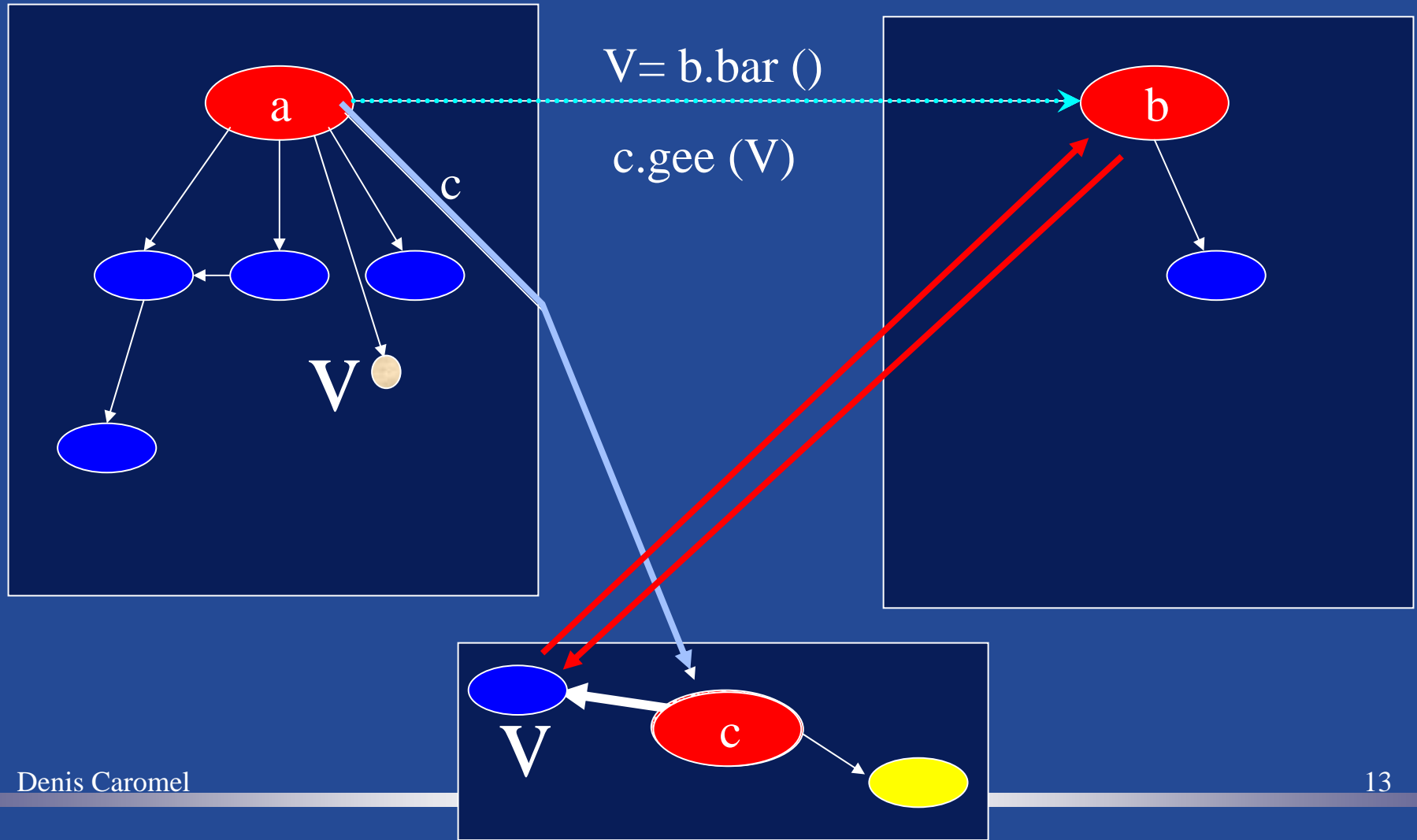
# Wait-By-Necessity: Eager Message Based

AO forwarding a future: send a message



# Wait-By-Necessity: Lazy Strategy

An Active Object requests a Future Value when needed



---

# TYPED ASYNCHRONOUS GROUPS



# Collective Communications: Groups

- Manipulate groups of Active Objects, in a simple and typed manner:
  - ➔ Typed and polymorphic Groups of active and remote objects
  - ➔ Dynamic generation of group of results
  - ➔ Language centric, Dot notation
- Be able to express high-level collective communications (like in MPI):
  - broadcast,
  - scatter, gather,
  - all to all

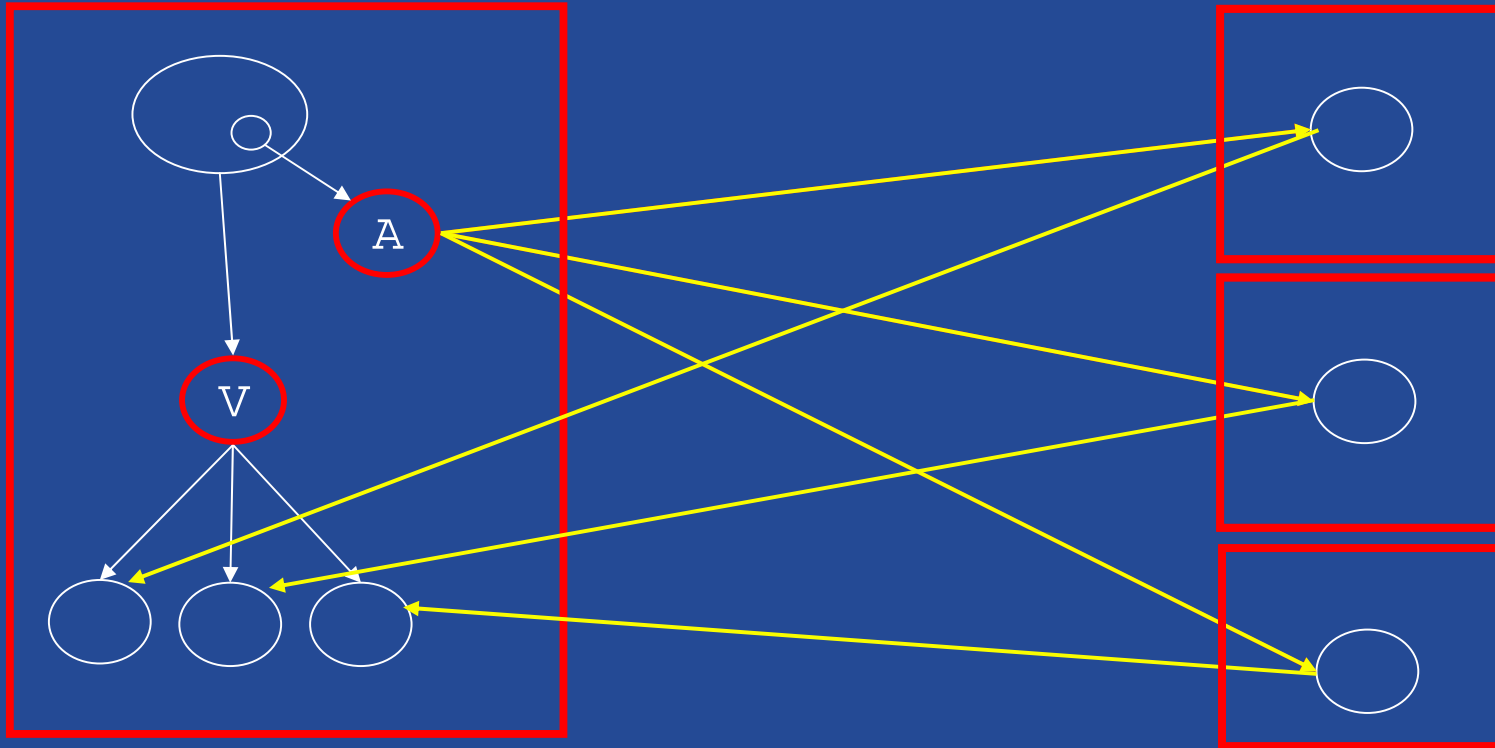
```
A ag=(A)ProActiveGroup.newActiveGroup(«A»,{{p1},...},{Nodes,..});  
V v = ag.foo(param);  
v.bar();
```



# Creating AO and Groups

- A ag = `newActiveGroup` ("A", [...], VirtualNode)
- V v = ag.foo(param);
- ...
- v.bar(); //Wait-by-necessity

JVM



○ Typed Group    ○ Java or Active Object

Group, Type, and Asynchrony  
are crucial for Cpt. and GRID



# Broadcast or scatter

Broadcast is the default behavior

Scatter is also possible

- Scatter uses a group as parameter
- Distribution relies on rank

```
ag.bar(cg, dg); // broadcast cg and dg
```

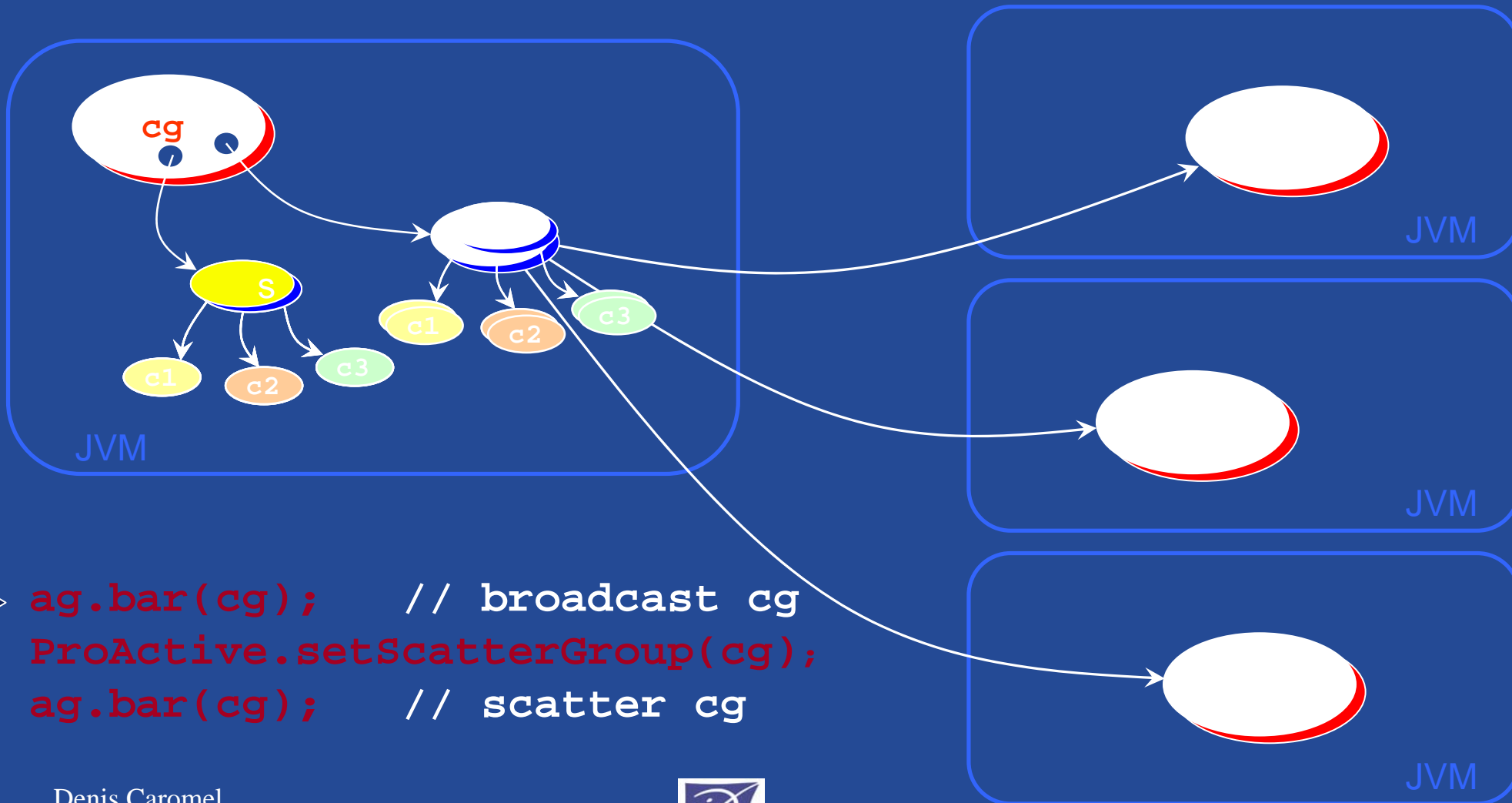
```
ProActive.setScatterGroup(cg);
```

```
ag.bar(cg, dg); // scatter cg, still broadcast dg
```

One call can both **scatter** and **broadcast**



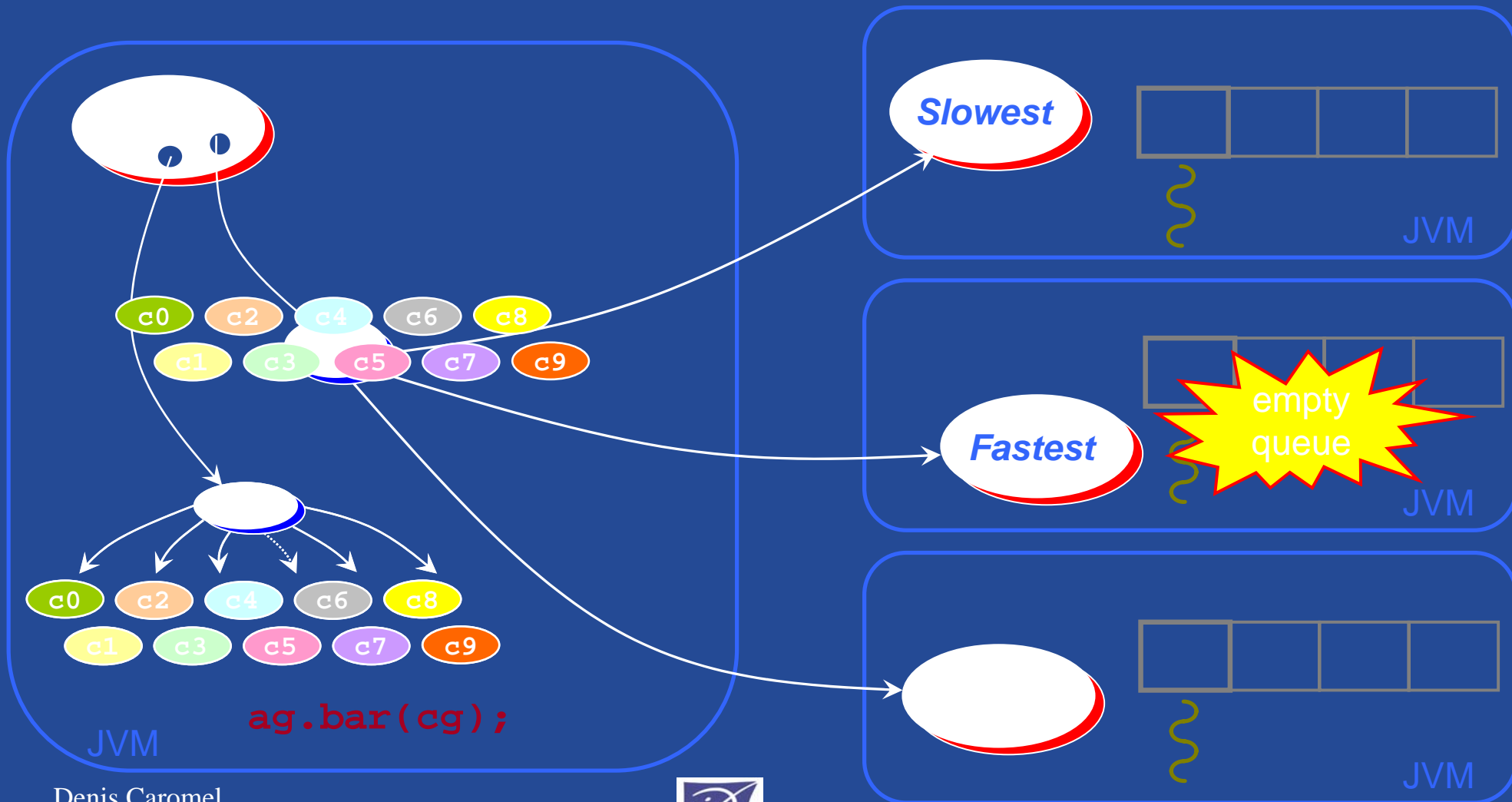
# Broadcast and Scatter



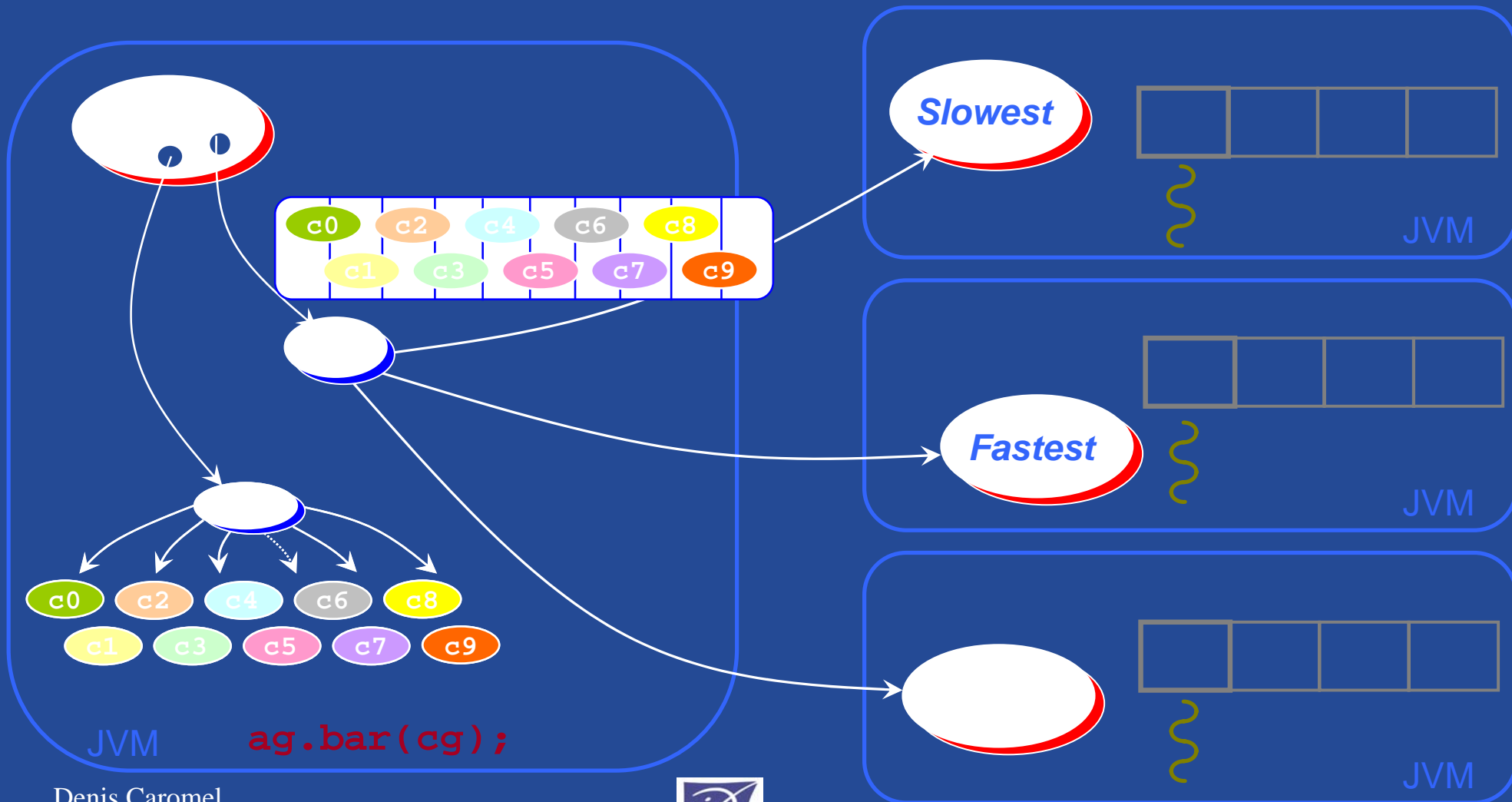
```
➔ ag.bar(cg); // broadcast cg  
ProActive.setScatterGroup(cg);  
ag.bar(cg); // scatter cg
```



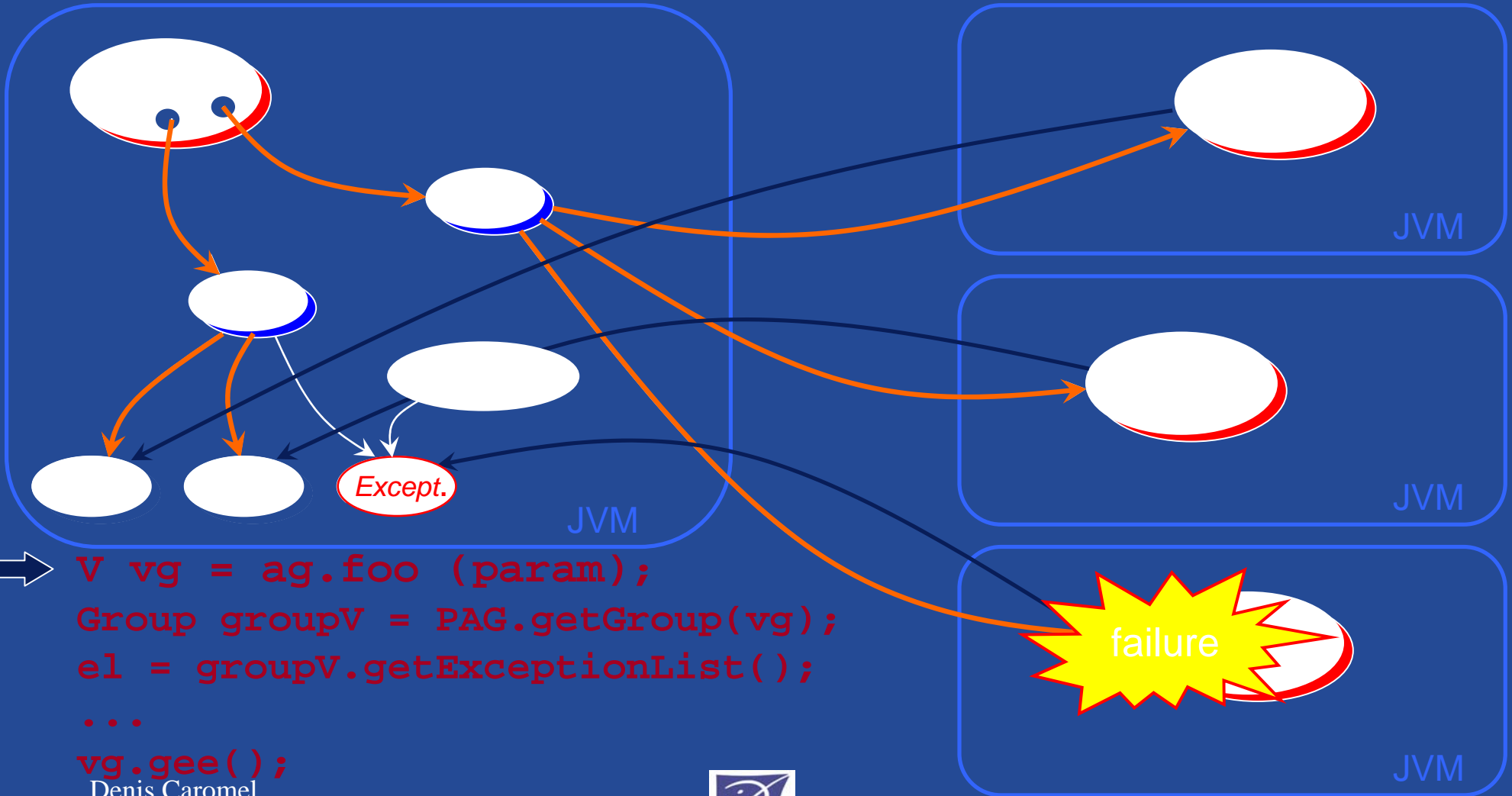
# Static Dispatch Group



# Dynamic Dispatch Group



# Handling Group Failures (2)



```
V vg = ag.foo (param);  
Group groupV = PAG.getGroup(vg);  
el = groupV.getExceptionList();  
...  
vg.gee();
```

Denis Caromel



# Group behavior manager

University of Sannio – Benevento  
(Italy)

## Definition of a group behavior

- Request mapping
- Parameters distribution
- Results gathering
- Synchronization semantic

## Dynamic configuration and binding

### IP Multicast library

- Bindings to multicast transport protocol
- TRAM, included in JRMS 1.1



# Object-Oriented SPMD

(Single Program Multiple Data)

## Motivation

- Cluster / Grid programming
- SPMD programming widely used

Already able to express most of the MPI's collective communications

- Broadcast
- Scatter
- Gather
- Reduce
- All scatter
- All gather

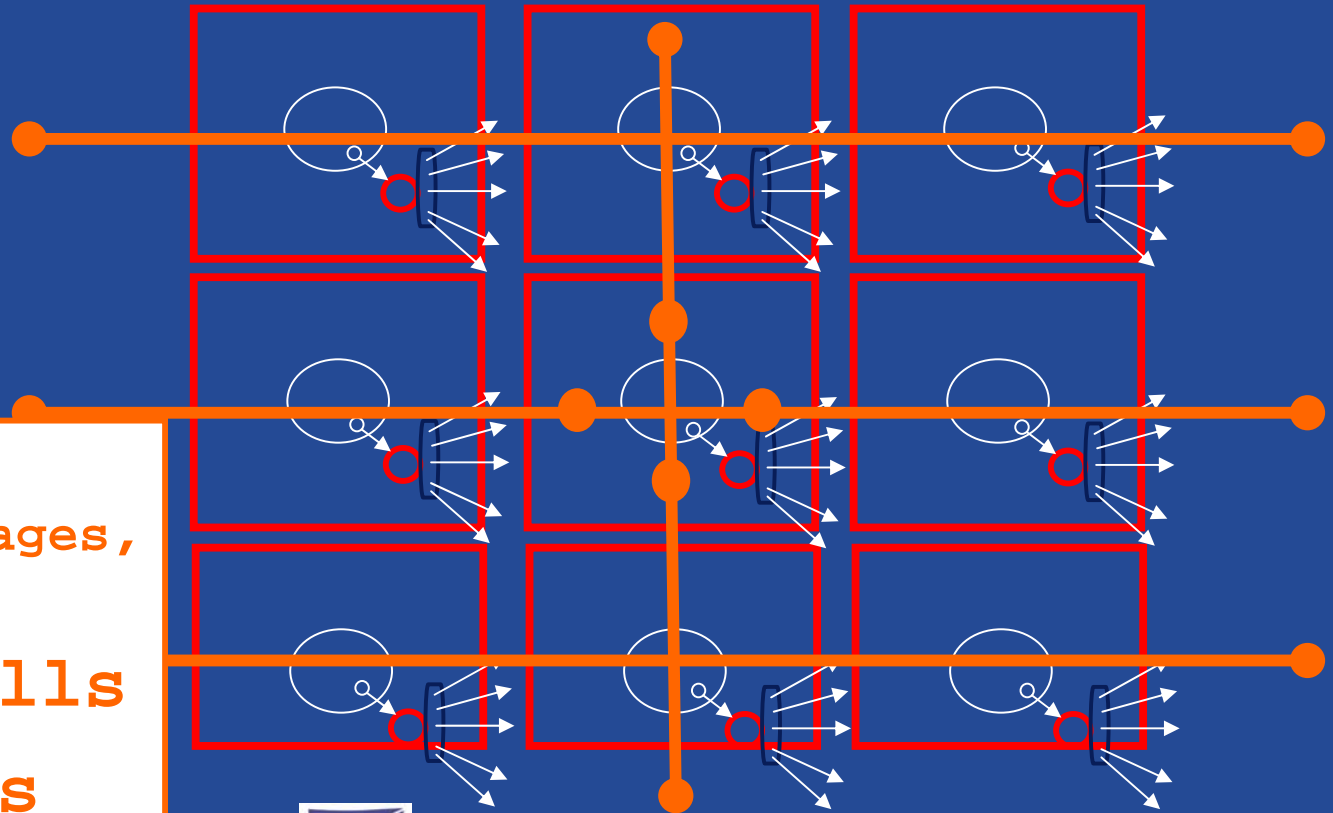
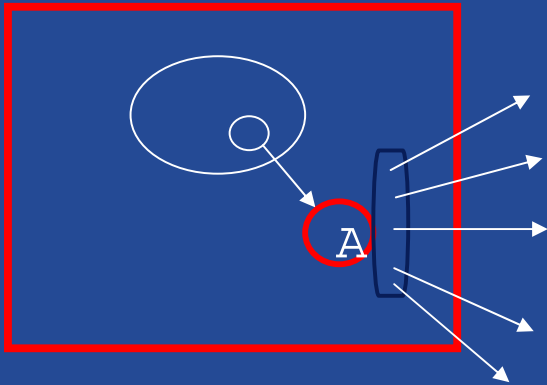
Different barriers

Topologies



# OO SPMD

- A ag = `newSPMDGroup ("A", [...], VirtualNode)`
  - // In each member
    - `myGroup.barrier ("2D"); // Global Barrier`
    - `myGroup.barrier ("vertical"); // Any Barrier`
    - `myGroup.barrier ("north","south","east","west");`



still,  
not based on raw messages,  
but  
Typed Method Calls  
==> Components



# API

## Topologies

- Table, Ring, Plan, Torus, Cube, ...
- Open API
- Neighborhood

## Barrier

- Global
- Neighbor-based
- Method-based



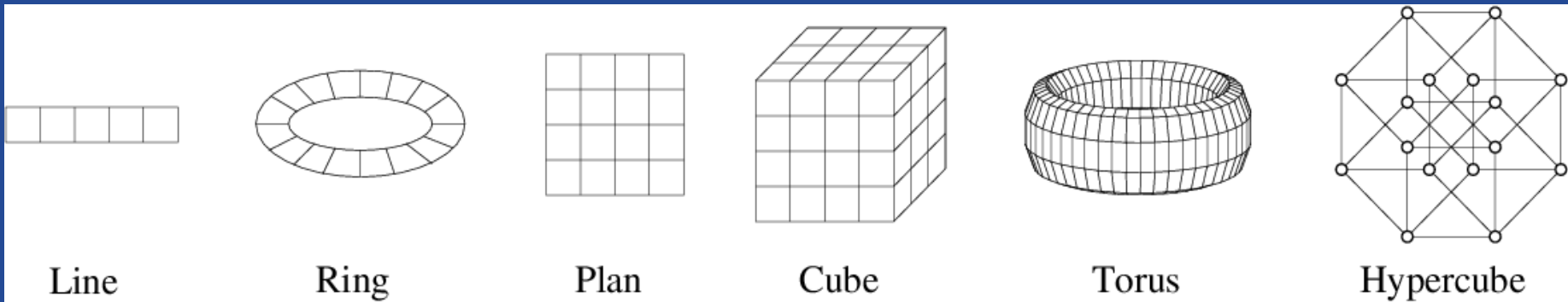
# Topologies

Topologies are typed groups

Open API

Neighborhood

Creation by extraction



```
Plan plan = new Plan(groupA, Dimensions);  
Line line = plan.getLine(0);
```

# ProActive OO SPMD

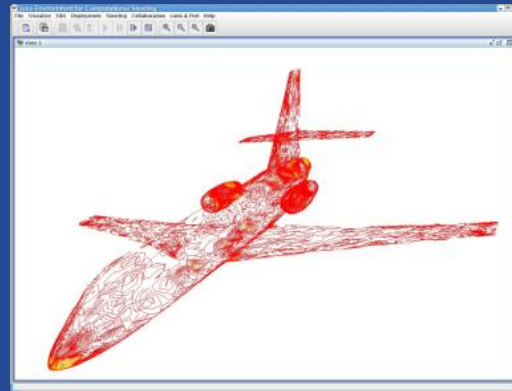
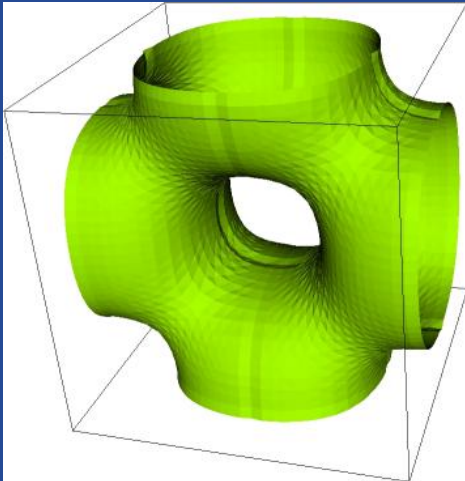
## A simple communication model

- Small API
- No “Receive” but data flow synchronization
- No message passing but RPC
- User defined data structure (Object)
- SPMD groups are dynamic
- Efficient and dedicated barrier

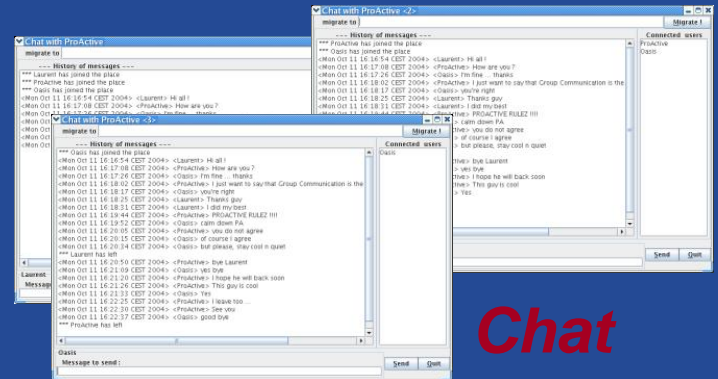


# Group communication is a key feature for the GRID

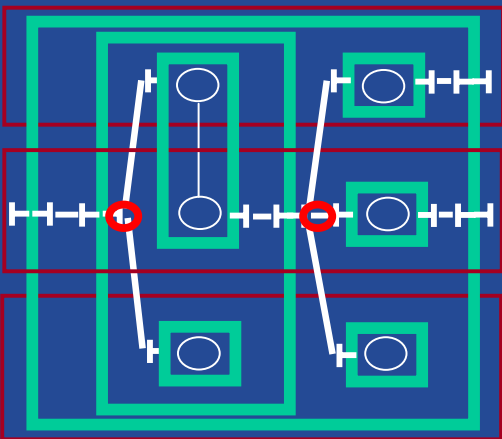
Used in many applications and other features



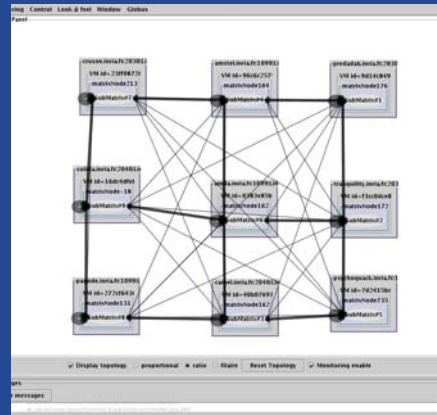
*Jem3D*



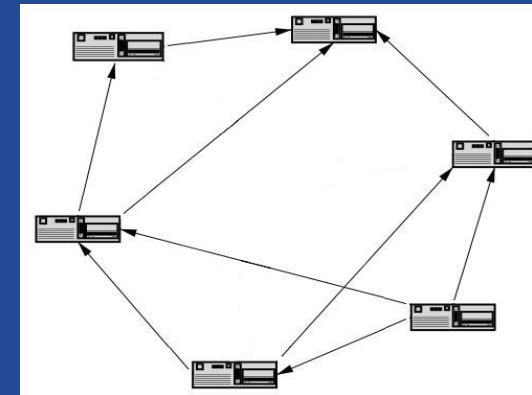
*Chat*



*Multiport component*



*Jacobi*



*Peer to peer*



# Sum up: MPI vs. OO SPMD

A simple communication model, with simple communication primitive(s):

- No RECEIVE but data flow synchronization
- Adaptive implementations are possible for:
  - // machines, Cluster, Desktop, etc.,
  - Physical network, LAN, WAN, and network conditions
  - Application behavior

**Typed Method Calls:**

**==> Towards Components**

**Reuse and composition:**

- No main loop, but asynchronous calls to myself:
  - ==> See details this afternoon in the Hands-On Session



# Single-Sided Communication

*SSC = Active Messages*

*An Hardware Reality*

*In OO SPMD: Immediate Services*



# Adaptive GRID

The need for adaptive middleware is now acknowledged, with dynamic strategies at various points in containers, proxies, etc.

Can we afford adaptive GRID ?

with dynamic strategies at various points  
(communications, groups, checkpointing, reconfiguration, ...)  
for various conditions (LAN, WAN, network, P2P, ...)

HPC vs. HPC

High Performance Components vs. High Productivity Components

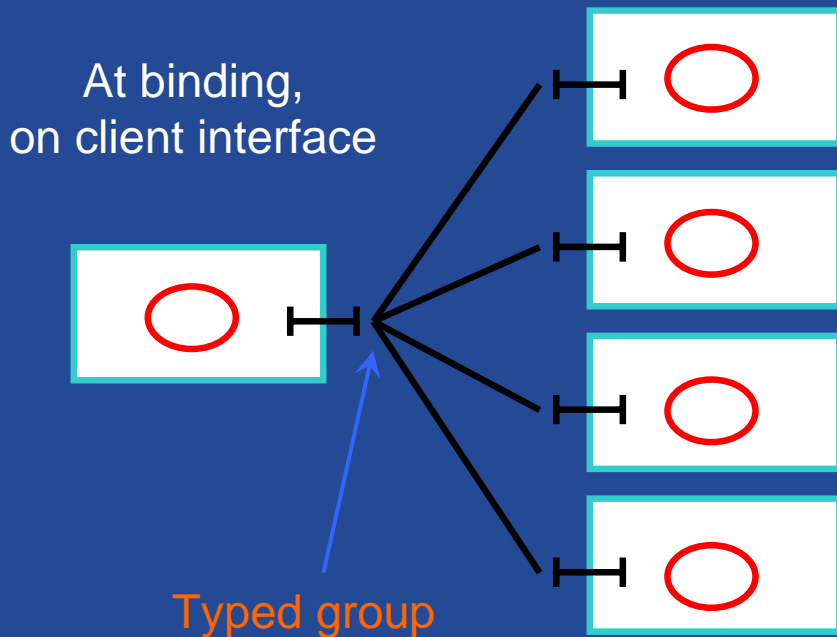


# Groups in Components

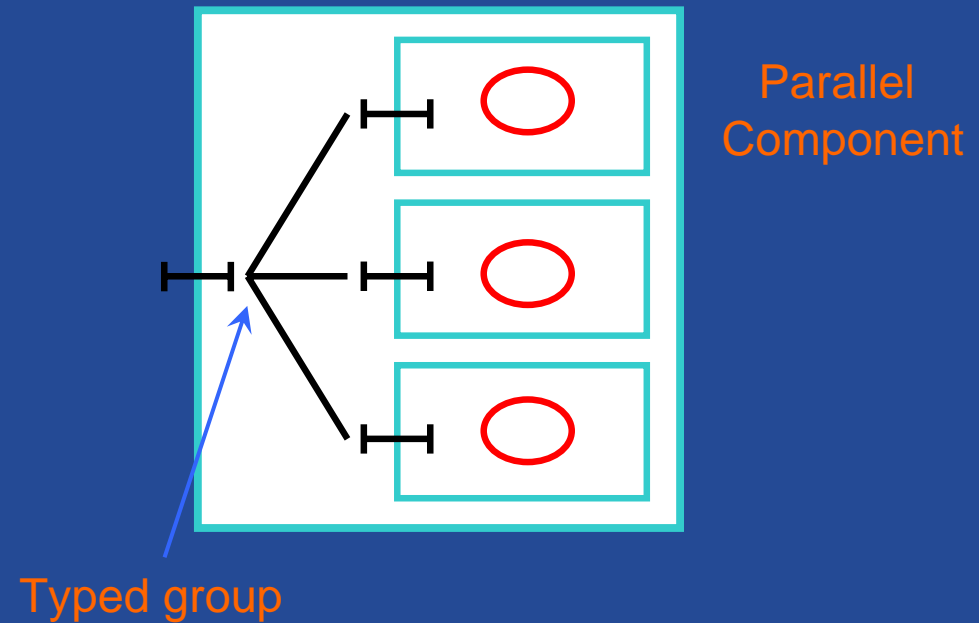
Implementation of the Fractal component model

Collective ports

Composite components

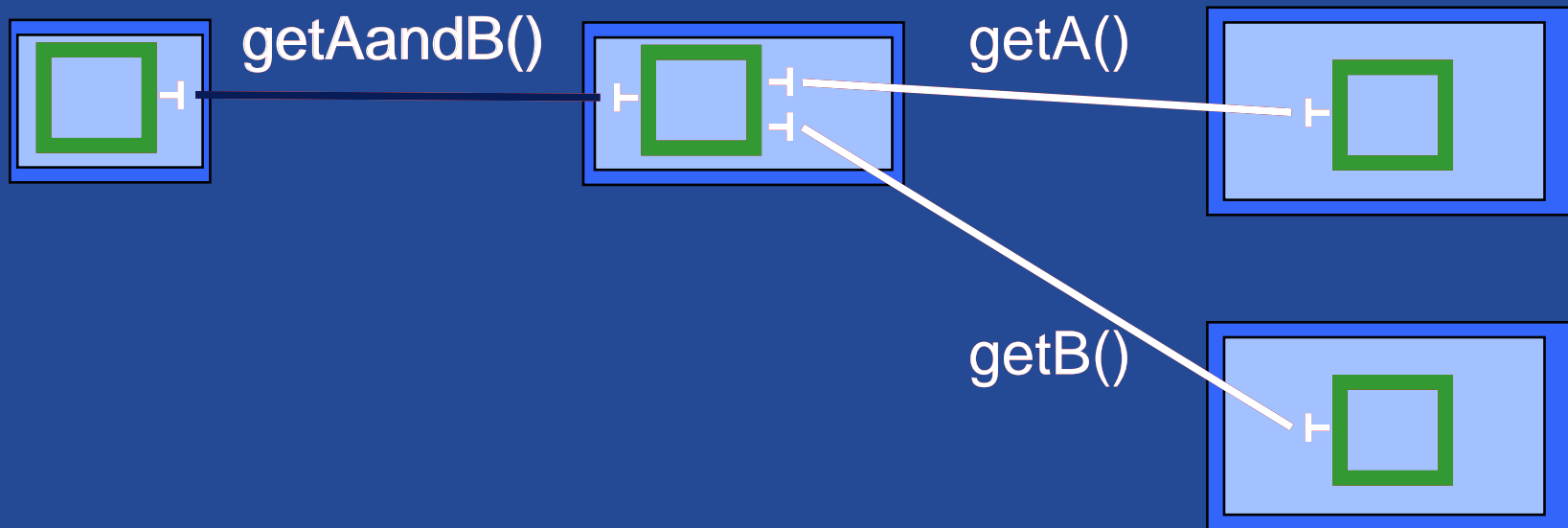


At composition,  
on composite inner server interface



# Functionalities : Without First Class Futures

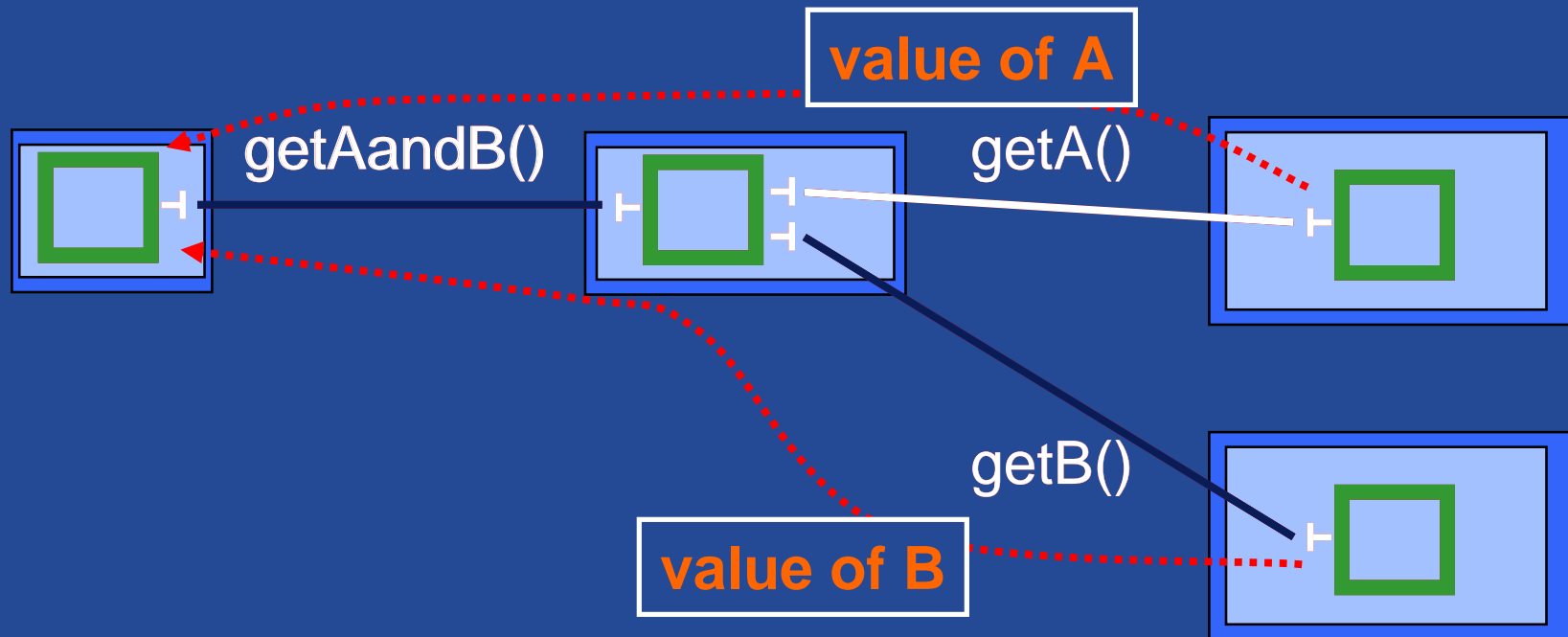
Or in the case of **Synchronous** method calls



# Functionalities : With First Class Futures

## Non-blocking method calls

Example 2 : **Asynchronous** method calls with full-fledge **Wait-By-Necessity**



**Assemblage are not blocked with Asynchrony + WbN**