Asynchronous Components

Asynchronous communications: from calculi to distributed components

Synchronous and asynchronous languages

- Systems build from communicating components : parallelism, communication, concurrency
- Asynchronous Processes
  - Synchronous communications (rendez-vous)

    Process calculi: CCS, CSP, Lotos
  - Asynchronous communications (message queues)

    SDL modelisation of channels
- Synchronous Processes (instantaneous diffusion)

  Esterel, Sync/State-Charts, Lustre

Question on D. Caromel course: how do you classify ProActive ?

Asynchrony in CCS

Processes Calculi – what is asynchrony?

- A proposal in π-calculus: Asynchronous π-calculus
- No consequence of output actions
- Equivalent in CCS:

P, Q	$::=$	0	inaction
	$\mid$	$\mu.P$	prefix
	$\mid$	$P \mid Q$	parallel
	$\mid$	$P + Q$	(external) choice
	$\mid$	$(\nu a)P$	restriction
	$\mid$	$\mathrm{rec}_K P$	process P with definition $K = P$
	$\mid$	K	(defined) process name

Processes Calculi – what is asynchrony? (2)

- μ.P can be a.P, $\overline{a}$.P, τ.P
- An asynchronous version would be to allow only a.P, and τ.P, and simply $\overline{a}$ without suffix
- $\overline{a}$.P has to be replaced by ($\overline{a}$|P)

- A very simple notion but sufficient at this level
- Same expressivity, but simple synchronisation can become more complex

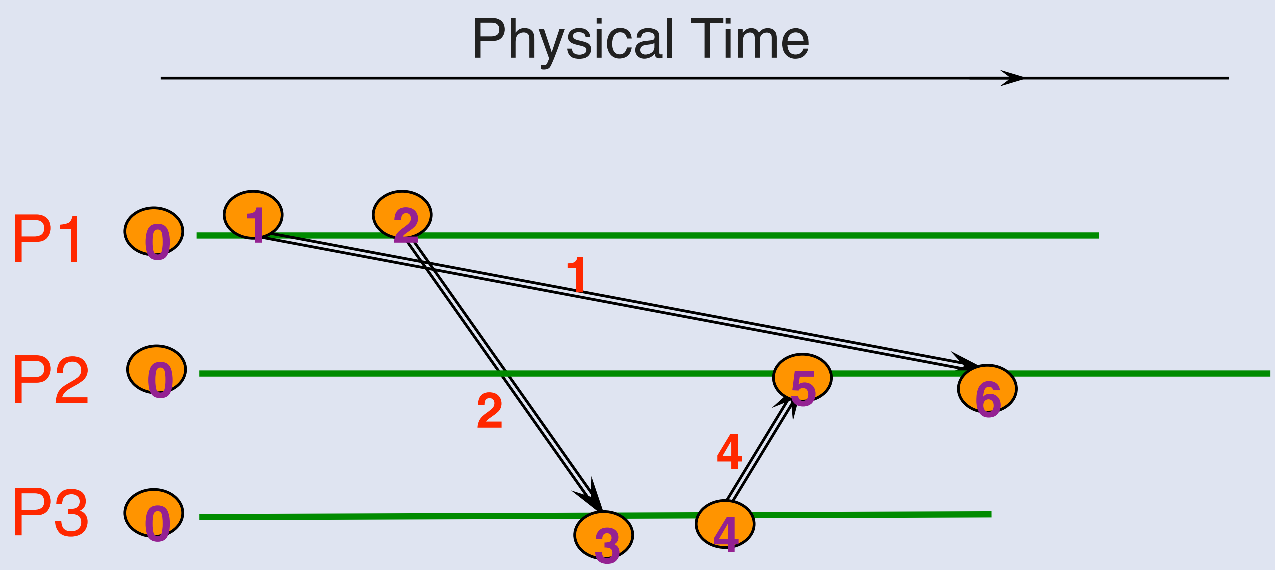
Communication Ordering; A Deeper Study

Synchronous, asynchronous, and causally ordered communication

Bernadette Charron–Bost, Friedemann Mattern, Gerard Tel

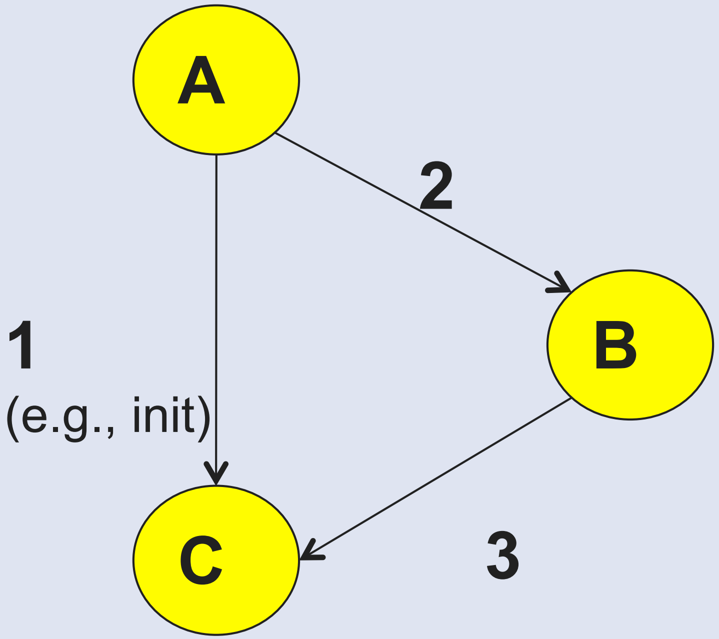
1996

Causality Violation



- Causality violation occurs when order of messages causes an action based on information that another host has not yet received.

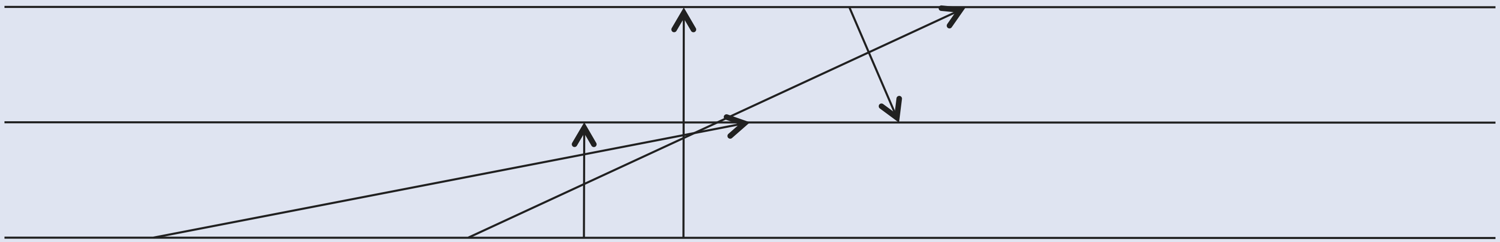
The “triangle pattern”



Objective: Ensure that **3** arrive at **C** after **1.**

Mattern: Communication is not only synchronous or asynchronous

- asynchronous communications, any order is valid (provided messages are received after being sent)



- (s,r) $\in \Gamma$ a communication
- $\prec_i$ local causality relation (total order on events)
- Global causality $\prec$, verifies at least

$a \prec_i b \Rightarrow a \prec b$

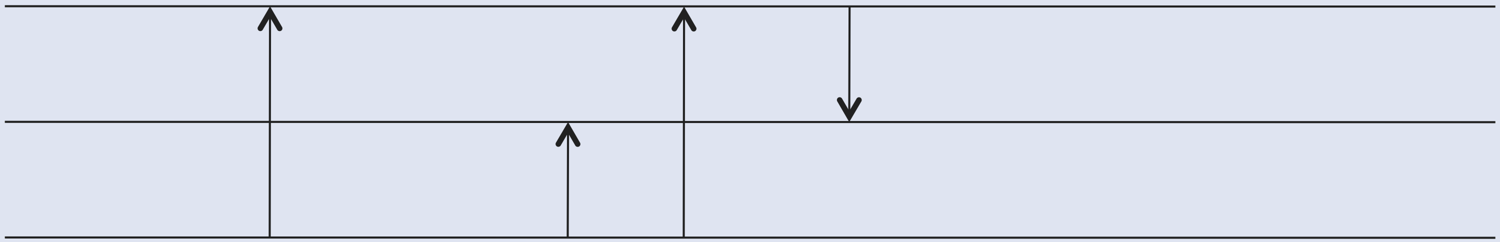
$s \prec r$ (if $(s,r) \in \Gamma$)

+ transitivity

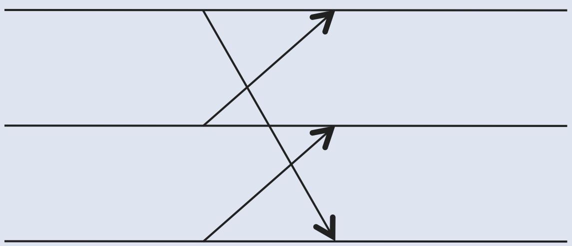
If $\prec$ is a partial order (antisymetric) then it represents a valid asynchronous communication
i.e. there must be no cycle of different events

Synchronous communication

- Emission and reception is almost the same event



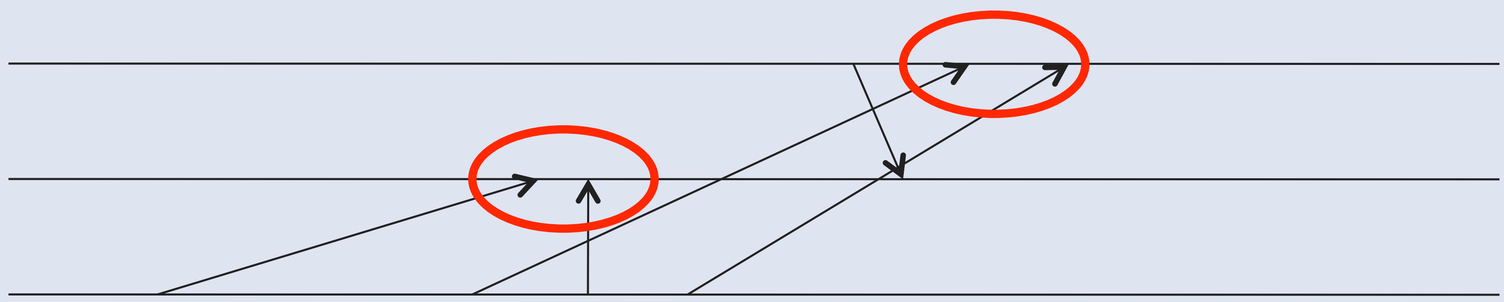
- A first characterization: Additionally
  if $(s,r) \in \Gamma$, then $a \prec s \Rightarrow a \prec r$ and $r \prec a \Rightarrow s \prec a$
  (still no cycle) – strong common past, strong common future
- Or : messages can be all drawn vertically at the same time
- OR: no crown



($s1 \prec r2$ and $s2 \prec r3$ and … $sn \prec r1$)

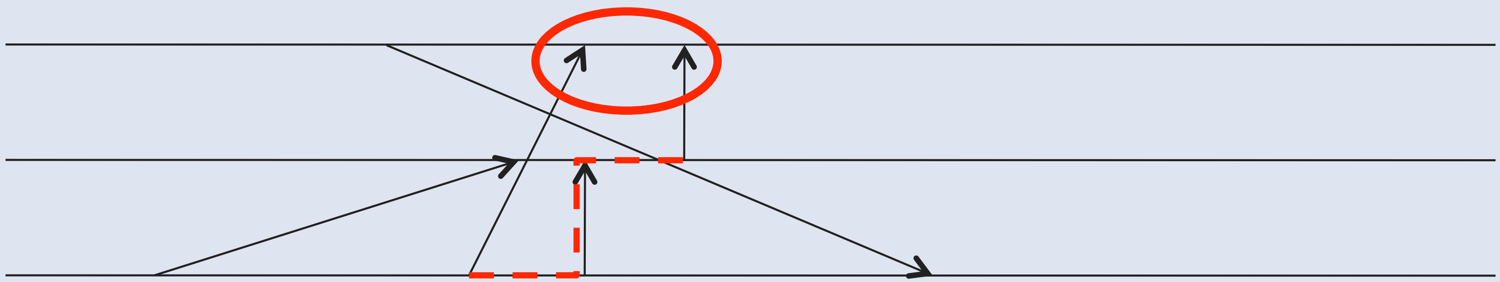
FIFO

- Order of messages sent between two given processes is guaranteed (reception order is the sending order)
- Let a~b if and b on the same process
- Asynchronous +
  if $(s,r) \in \Gamma$, $(s',r') \in \Gamma$, $s \sim s'$ and $r \sim r'$
  then $s \prec s' \Rightarrow r \prec r'$
  (still no cycle)



Causal Ordering

- More general than FIFO
- Asynchronous +
  if $(s,r) \in \Gamma$, $(s',r') \in \Gamma$, and $r \sim r'$
  then $s \prec s' \Rightarrow r \prec r'$
  (still no cycle)
- A nice characterization: for each message the diagram can be drawn with m as a vertical arrow and no other message go backward



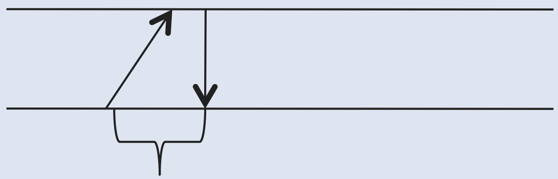
Applications

Such characterizations are useful for

- Identifying coherent states (states that could exist)
- Performing fault-tolerance and checkpointing
- Study which algorithms are applicable on which communication orderings
- Might be useful for debugging, or replaying an execution

A “few” communication orderings

- Synchronous
- FIFO channels
- Causal ordering
- Synchronous



No event beteen sending and reception

- What is rendez-vous?
  What does rendez-vous ensure?

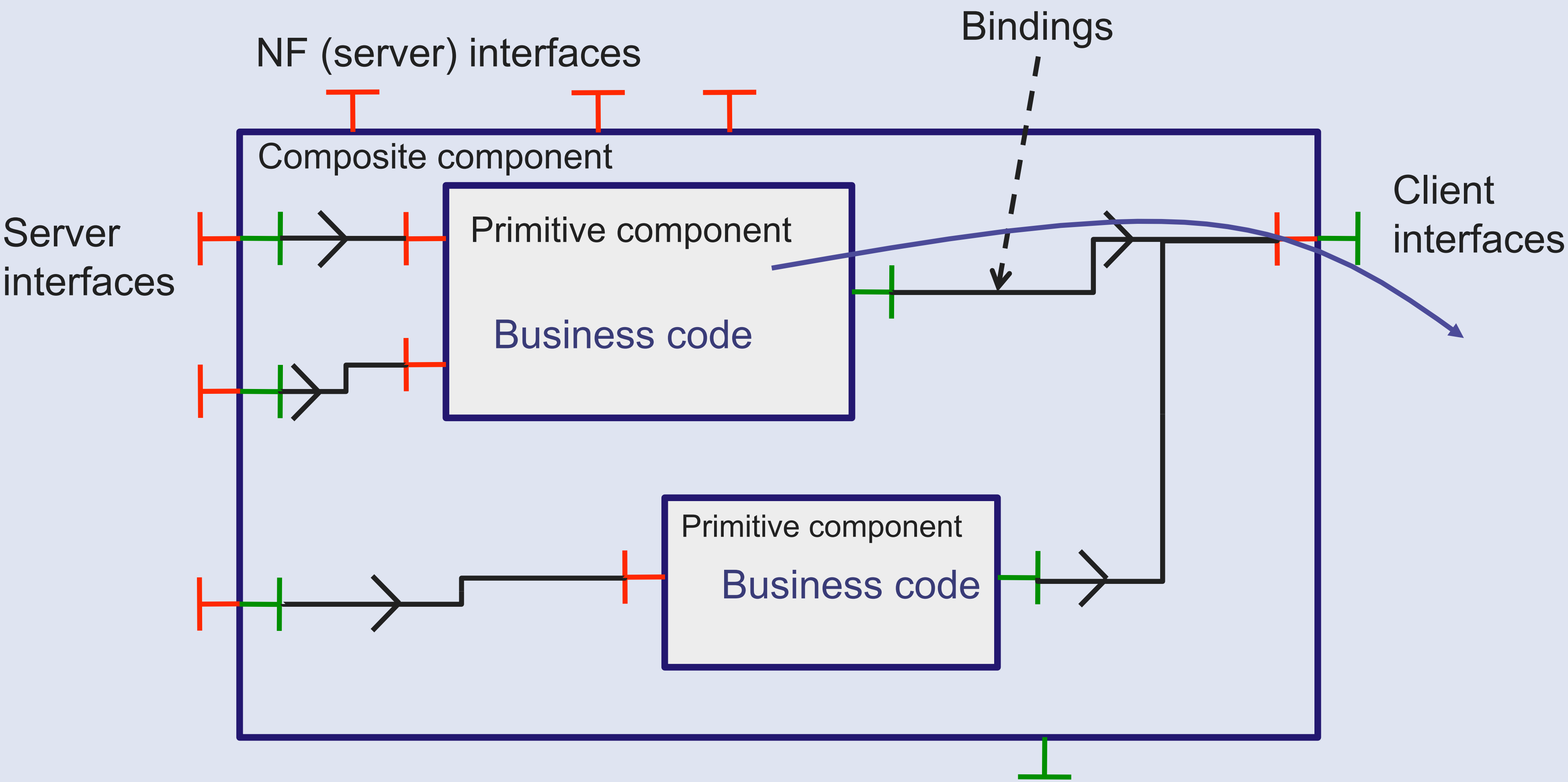
- So why is ProActive said asynchronous?

GCM: “Asynchronous” Fractal Components

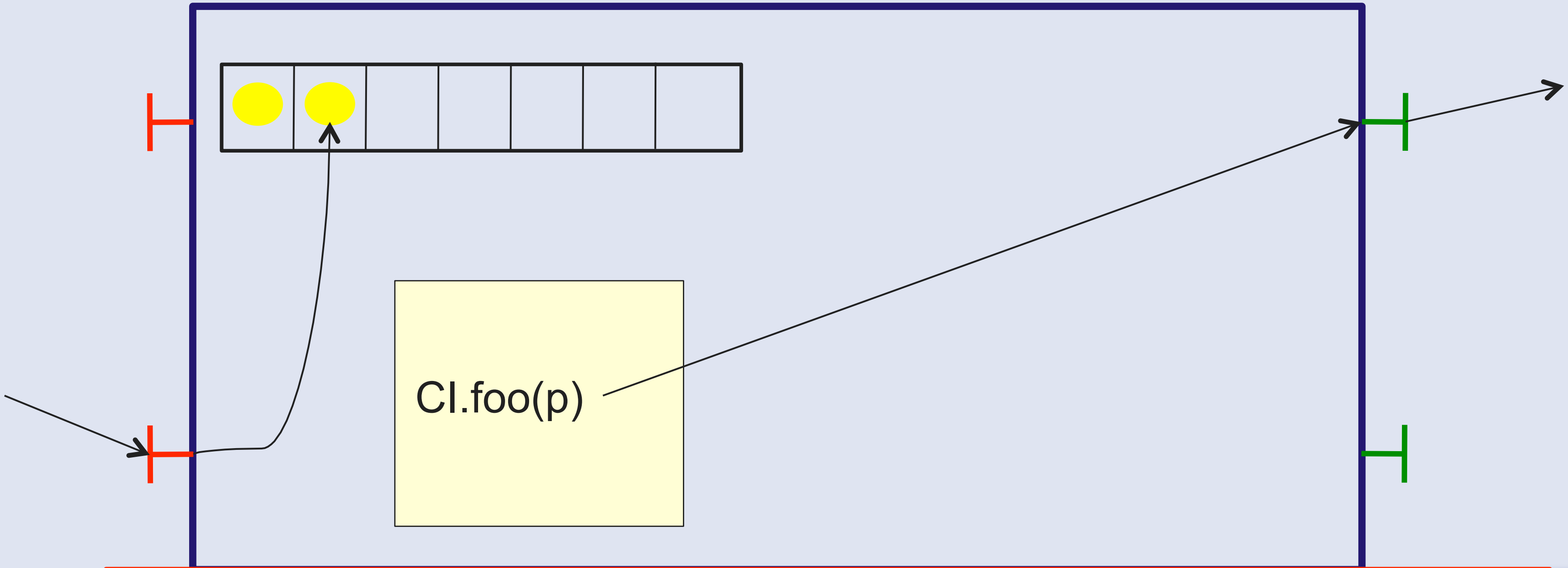
GCM – Quick Context

- Designed in the CoreGrid Network of Excellence, Implemented in the GridCOMP European project
- Add distribution to Fractal components
- OUR point of view in OASIS:
  - No shared memory between components
  - Components evolve asynchronously
  - Components are implemented in ProActive
  - Communicate by request/replies (Futures)
- A good context for presenting asynchronous components futures and many-to-many communications

What are (GCM/Fractal) Components?



A Primitive GCM Component

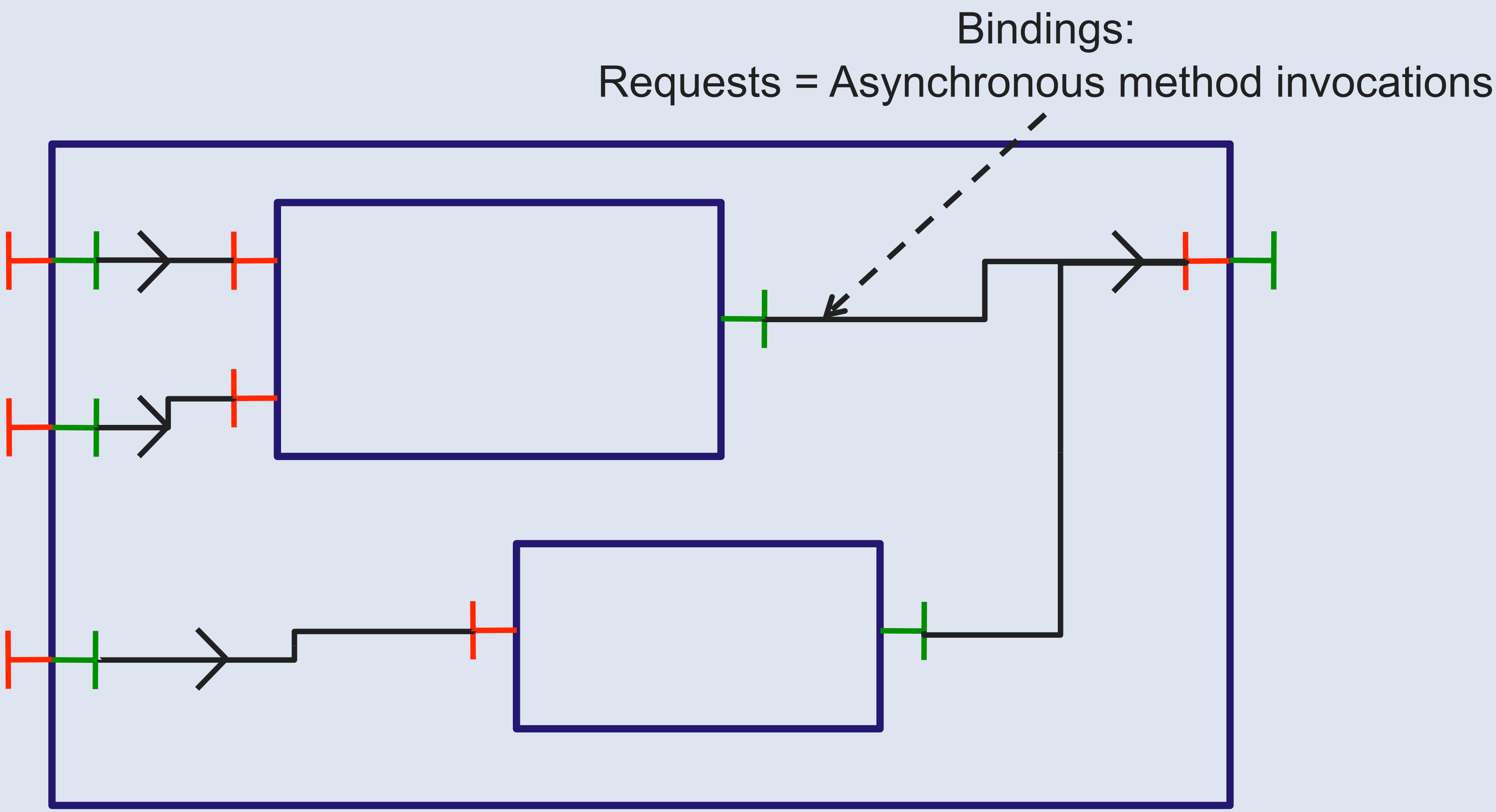


Primitive components communicating by *asynchronous* remote method invocations on interfaces (*requests*)

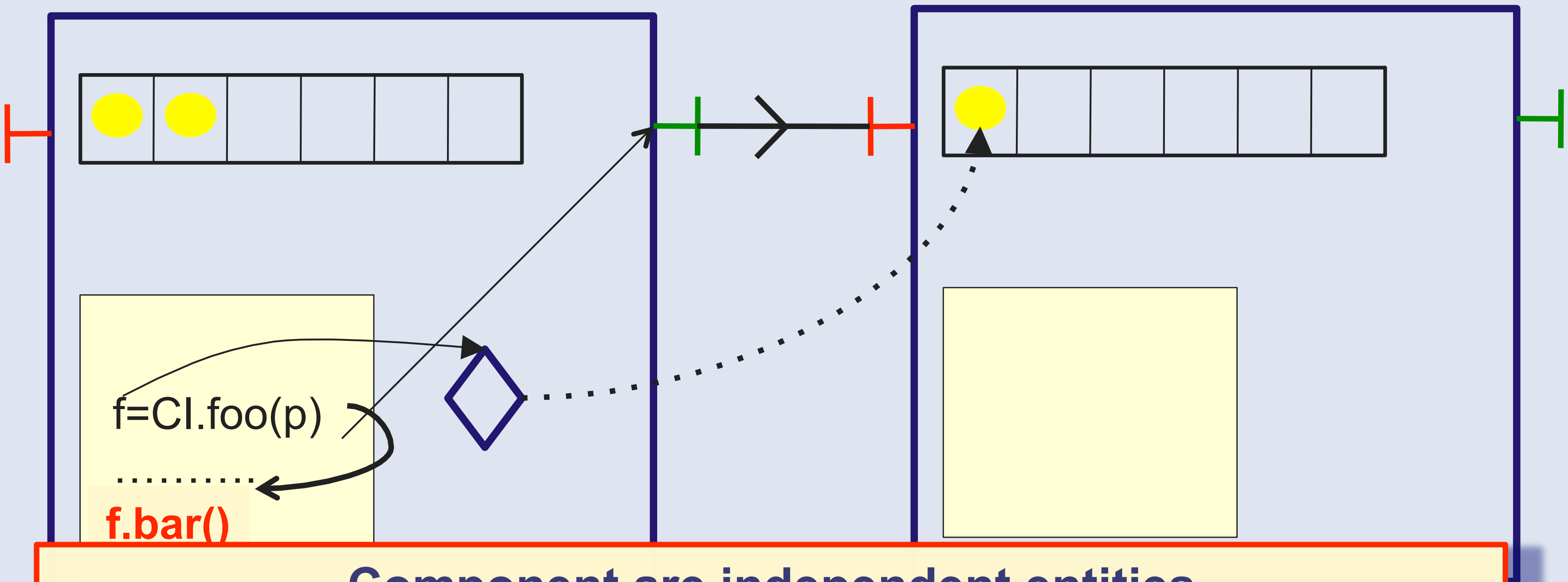
➔ Components abstract away distribution and *concurrency*

in ProActive components are mono-threaded
➔ **simplifies concurrency** but can create **deadlocks**

Composition in GCM

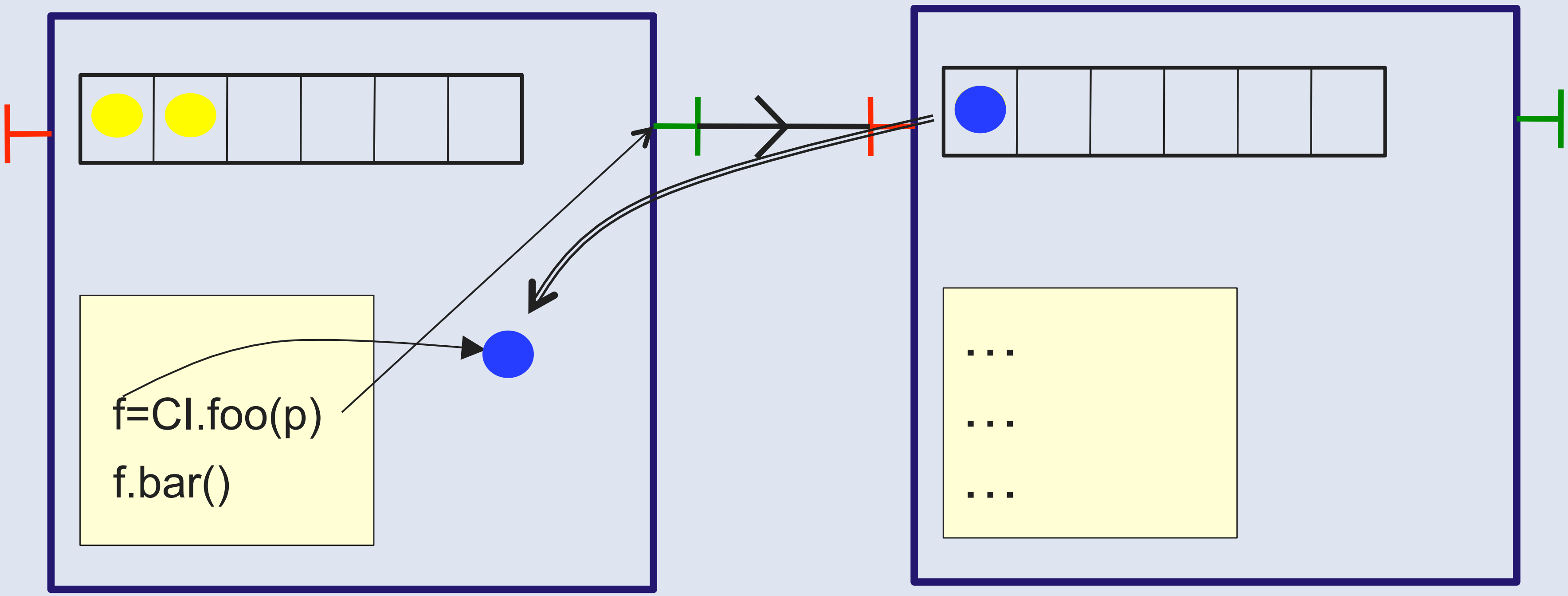


Futures for Components

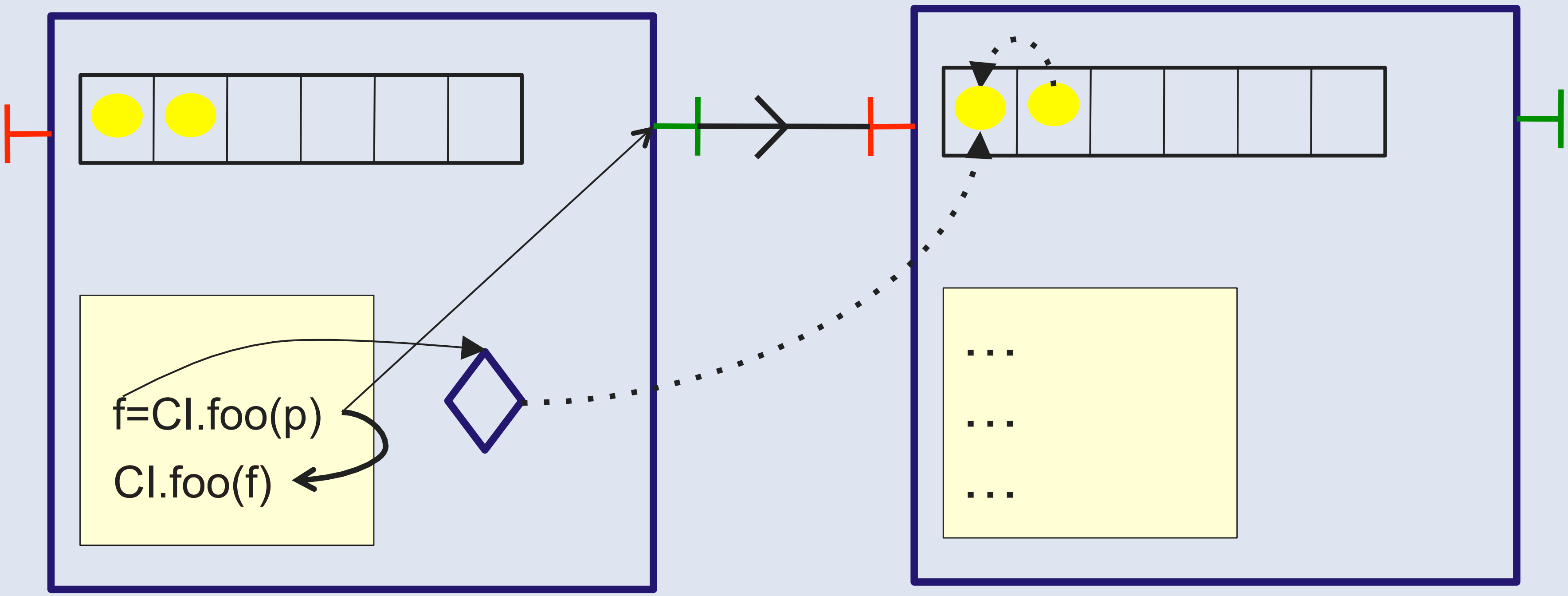


Component are independent entities
(threads are isolated in a component)
+
Asynchronous method invocations with results
⬇
Futures are necessary

Replies

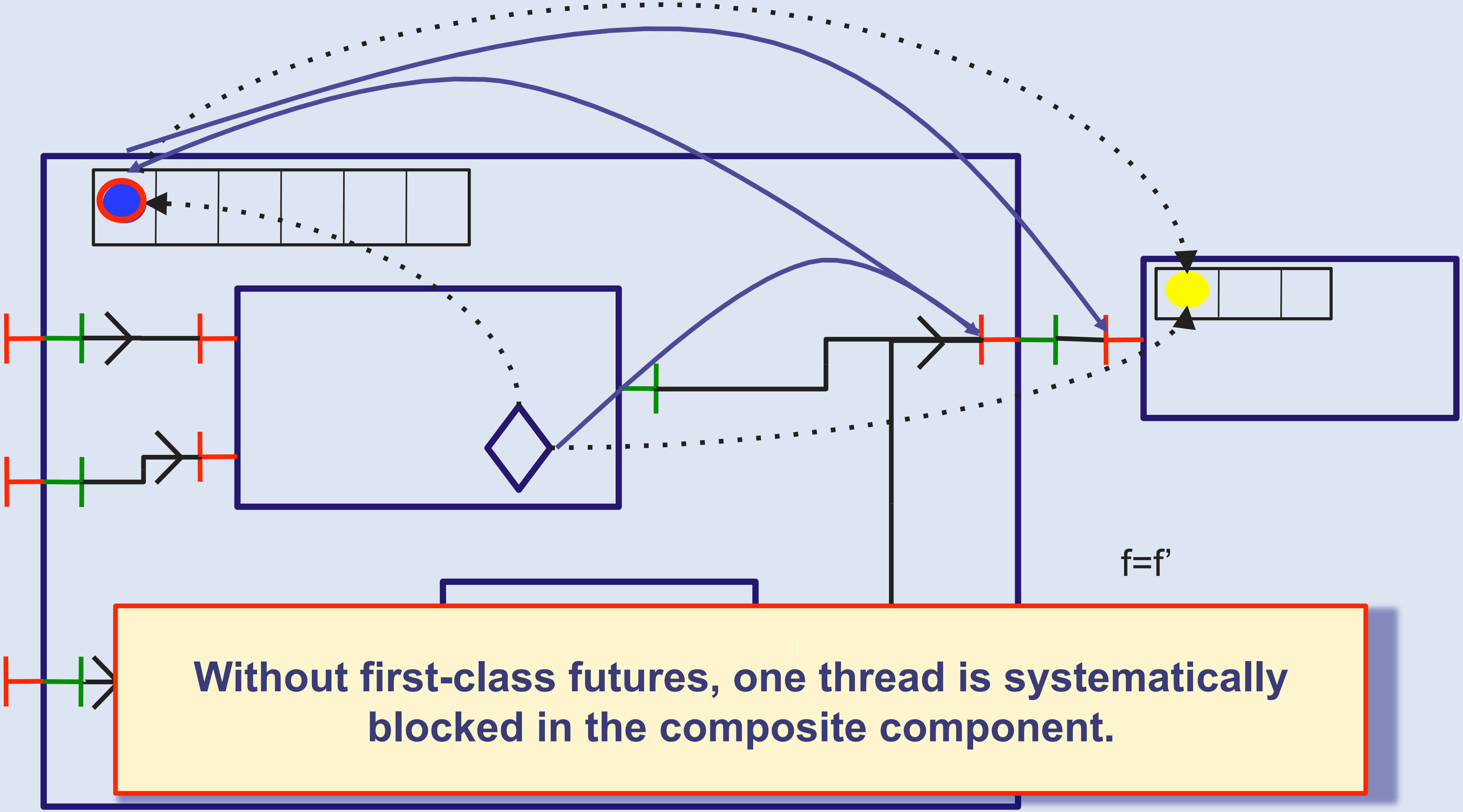


First-class Futures



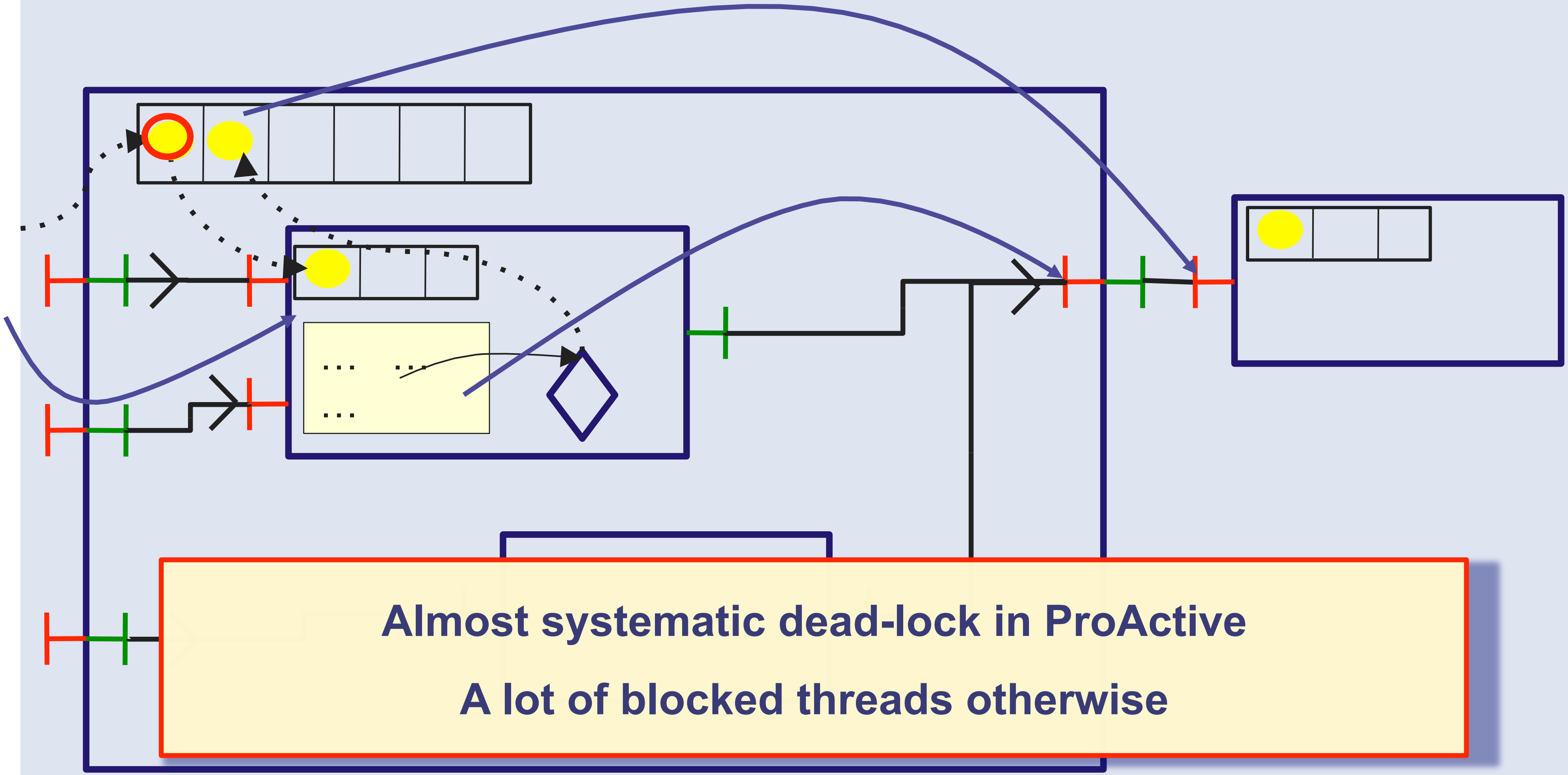
- Only strict operations are blocking (access to a future)
- Communicating a future is not a strict operation

First-class Futures and Hierarchy



Without first-class futures, one thread is systematically blocked in the composite component.

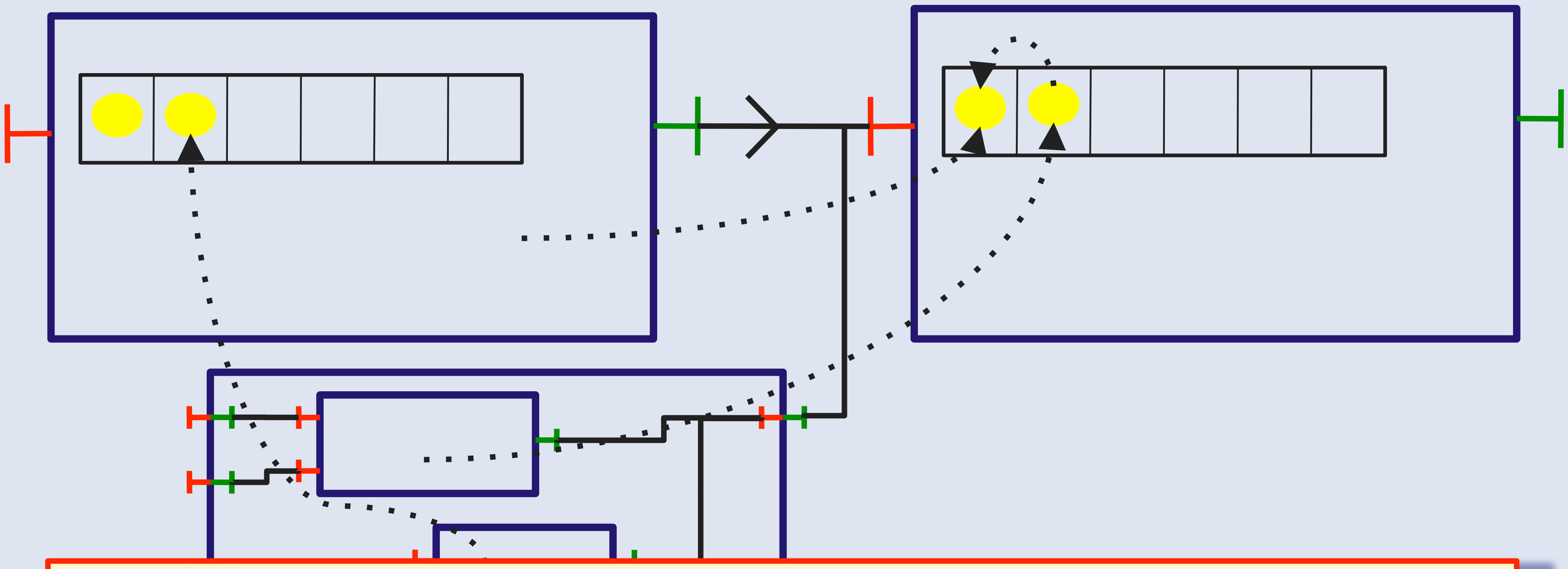
First-class Futures and Hierarchy



Almost systematic dead-lock in ProActive

A lot of blocked threads otherwise

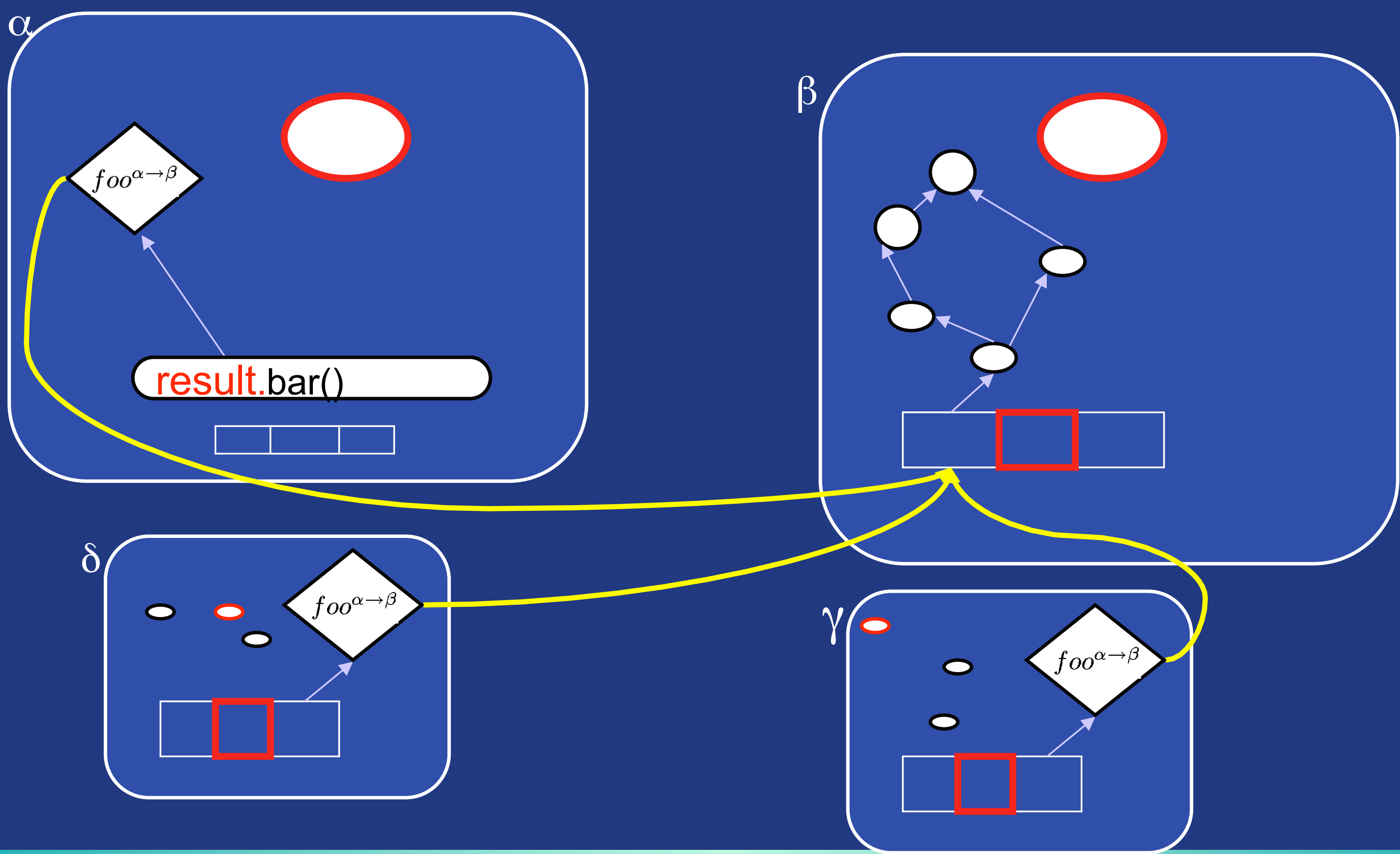
Reply Strategies



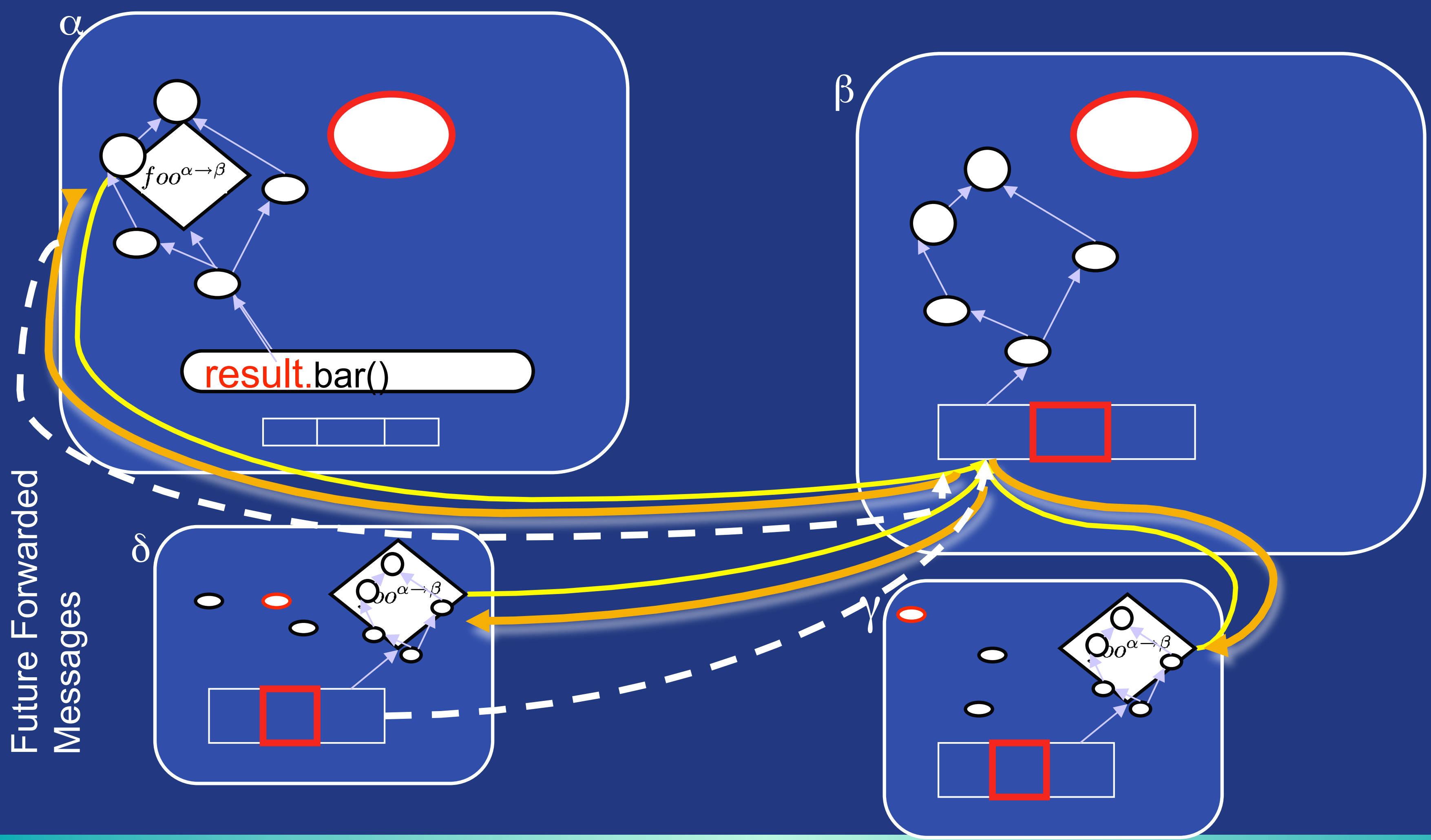
In ASP / ProActive, the result is insensitive to the order of replies (shown for ASP-calculus)

experiments with different strategies

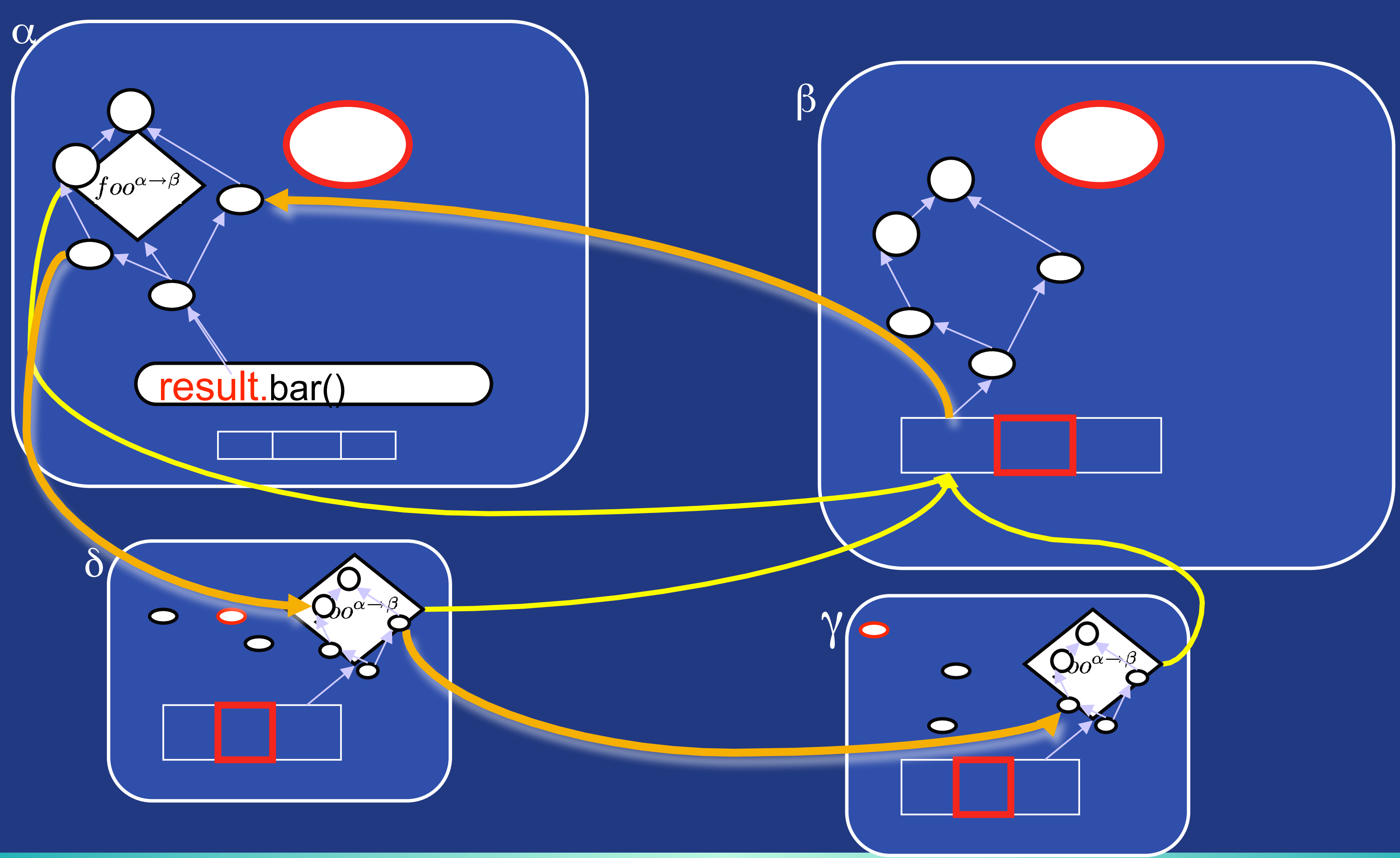
Future Update Strategies



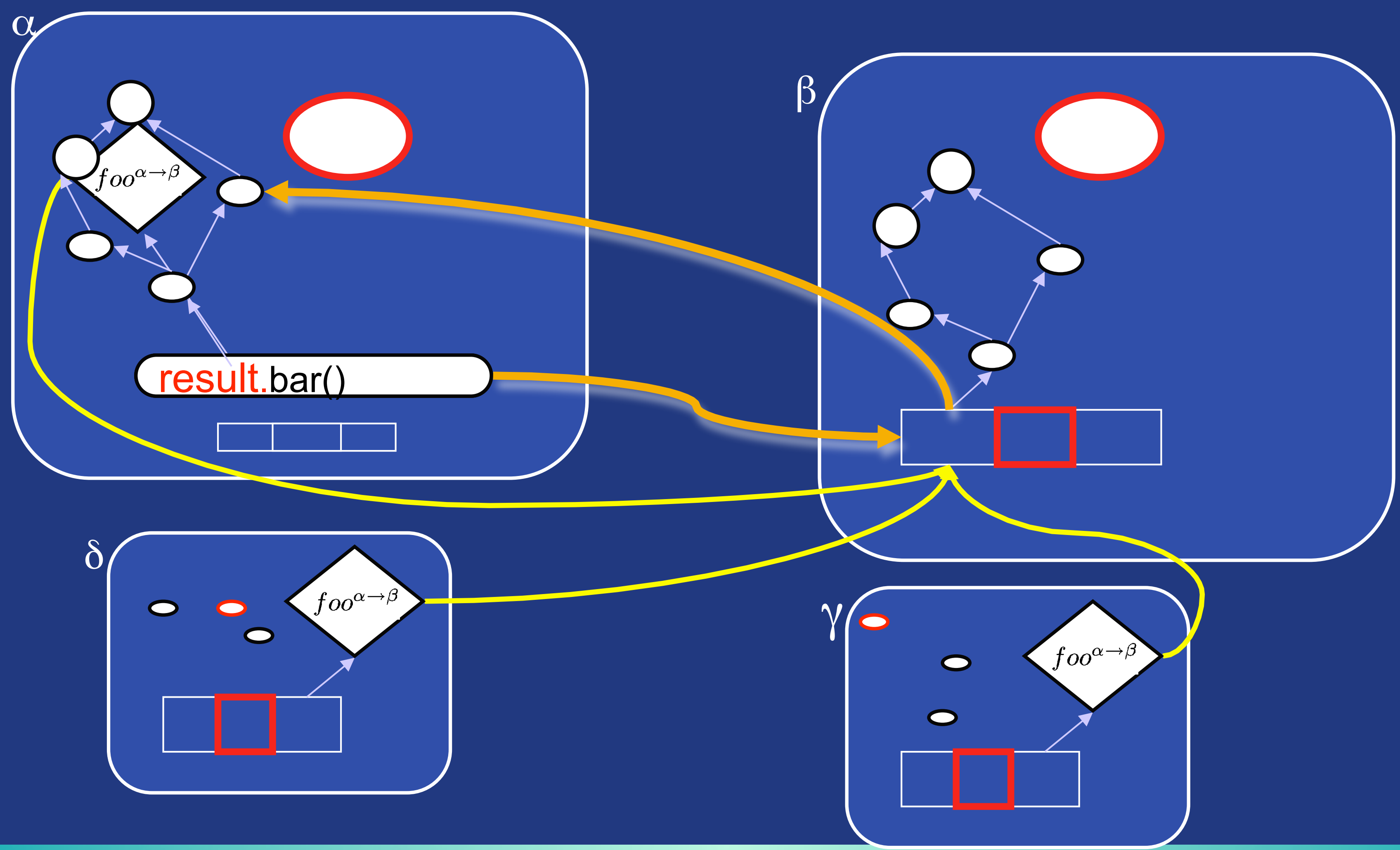
Future Update Strategies: Message-based



Future Update Strategies: Forward-based



Future Update Strategies: Lazy Future Updates

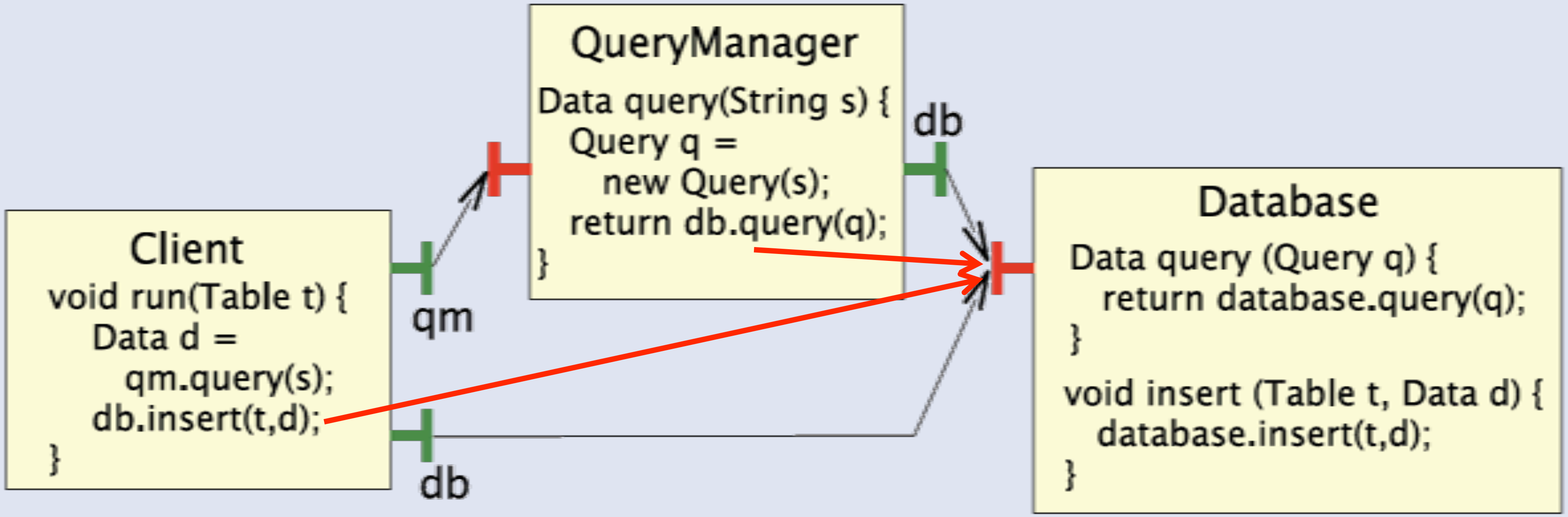


A Distributed Component Model with Futures

- Primitive components contain the business code
- Primitive components act as the unit of distribution and concurrency (each thread is isolated in a component)
- Communication is performed on interfaces and follows component bindings
- Futures allow communication to be asynchronous requests
- **Futures are transparent can lead to optimisations and are a convenient programming abstraction but …**

What Can Create Deadlocks?

- A race condition:



- Detecting deadlocks can be difficult ➔ behavioural specification and verification techniques (cf Eric Madelaine)

Collective Communications

Communications are not necessarily one-to-one:

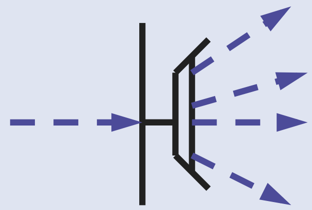
- One-to-many
- Many-to-One
- M by N

Collective Communications

- Simple type system
- Component type = types of its interfaces
- Interface type :
  - Name
  - Signature
  - Role
  - Contingency
  - Cardinality extended to support multicast / gathercast

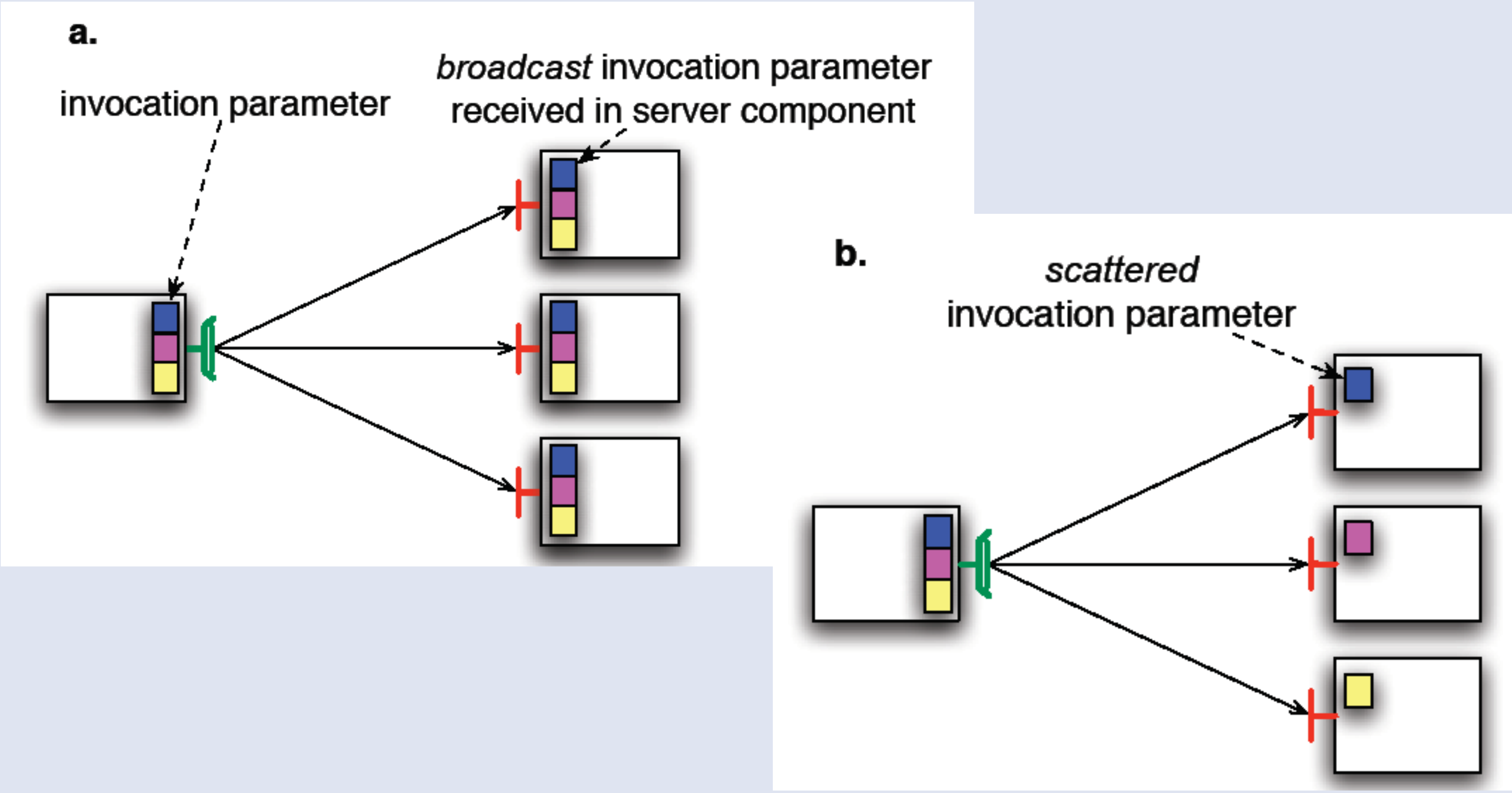
Fractal type-system

Multicast interfaces



Transform a single invocation into a list of invocations

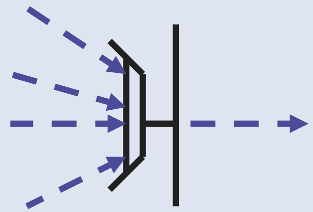
- Multiple invocations
  - Parallelism
  - Asynchronism
  - Dispatch
- Data redistribution (invocation parameters)
  - Parameterisable distribution function
  - Broadcast, scattering
  - Dynamic redistribution (dynamic dispatch)
- Result = list of results


a.
invocation parameter
broadcast invocation parameter
received in server component
b.
scattered
invocation parameter

Ordering and Multicast

- <u>FIFO ordering</u>: If a correct process issues *multicast(i,m) and then multicast(i,m'), then every correct process that delivers m' will deliver m before m'.*
- <u>Causal ordering</u>: If *multicast(i,m) precedes multicast(i',m') with i abd i' containing the same elements then any correct process that delivers m' will deliver m before m'.*
- <u>Totally ordering</u> (determinism): If a correct process delivers message *m before m', then any other correct process that delivers m' will deliver m before m'.*

Gathercast interfaces



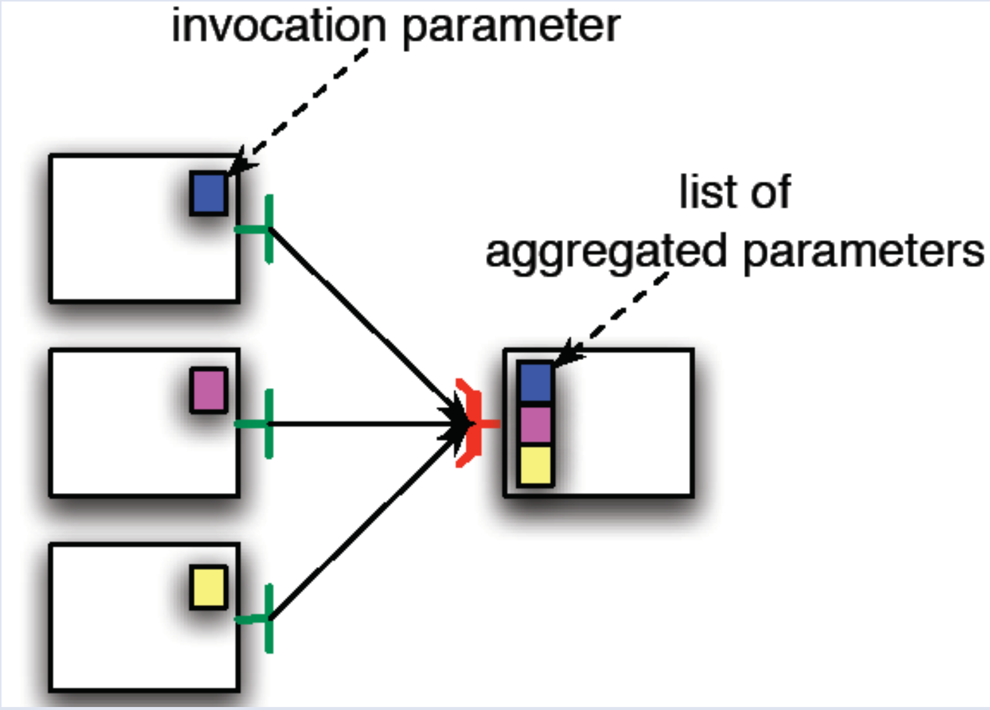
Transform a list of invocations into a single invocation

- Synchronization of incoming invocations
  - ~ “join” invocations
  - Timeout / drop policy
  - Bidirectional bindings (callers ⇔ callee)
- Data gathering

  Aggregation of parameters into lists

- Redistribution of results

  Redistribution function



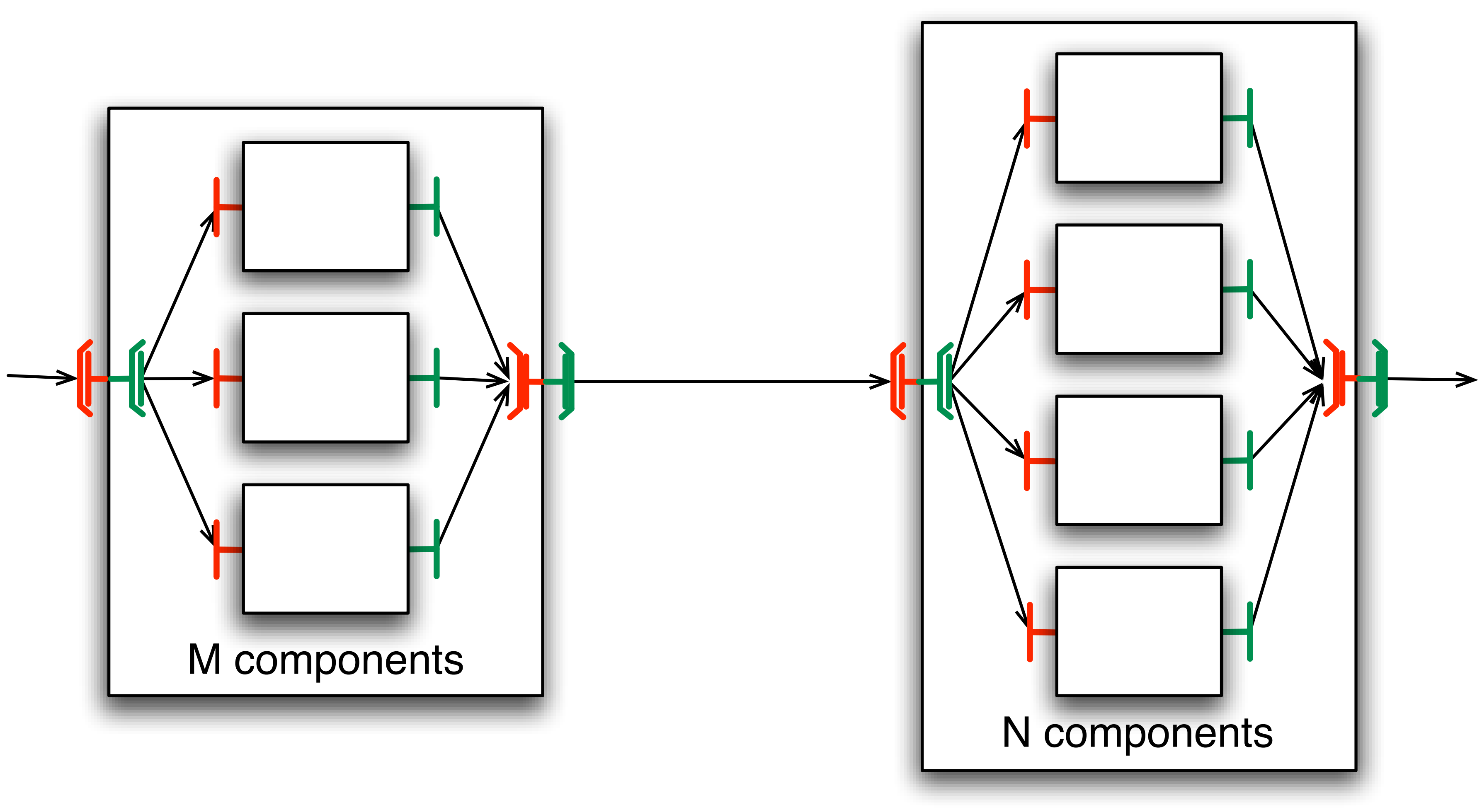
Collective interfaces

- Specific API ➔ manage collective interfaces and reconfigure them (add client, change policy, …)

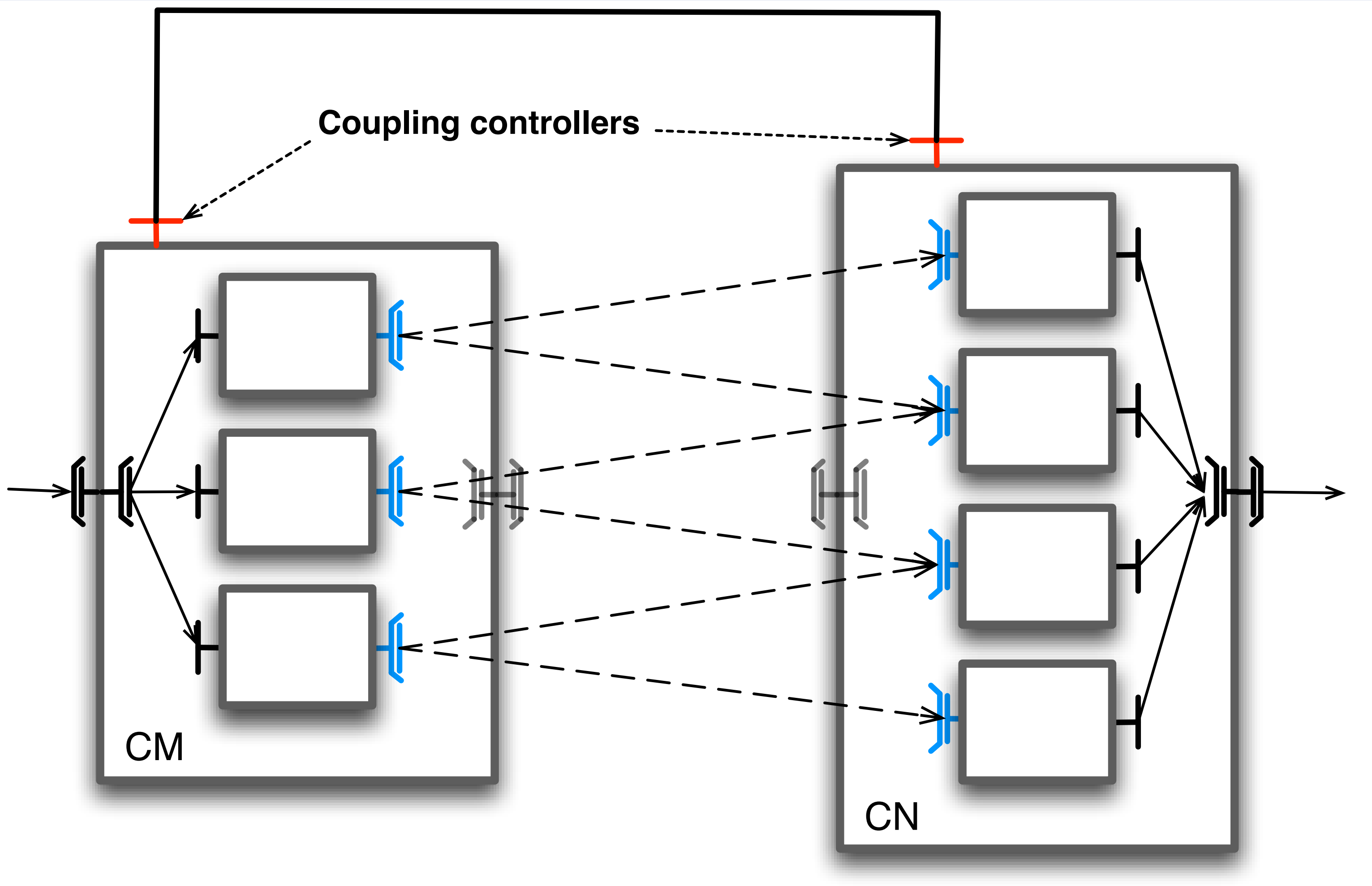
➡ Allow MxN communications:

Redistribution and direct communications for many-to-many communications

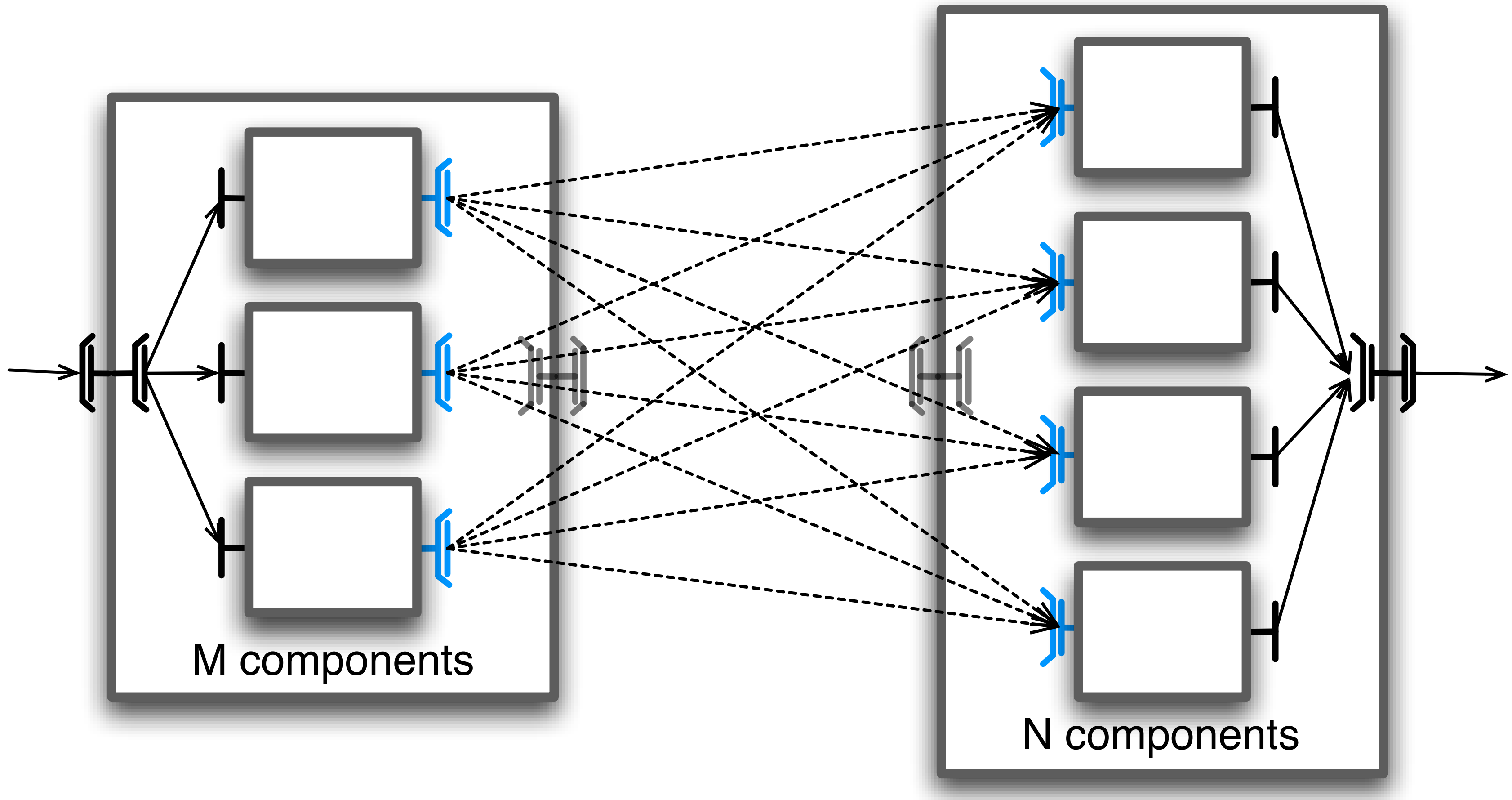
The MxN Problem (1)



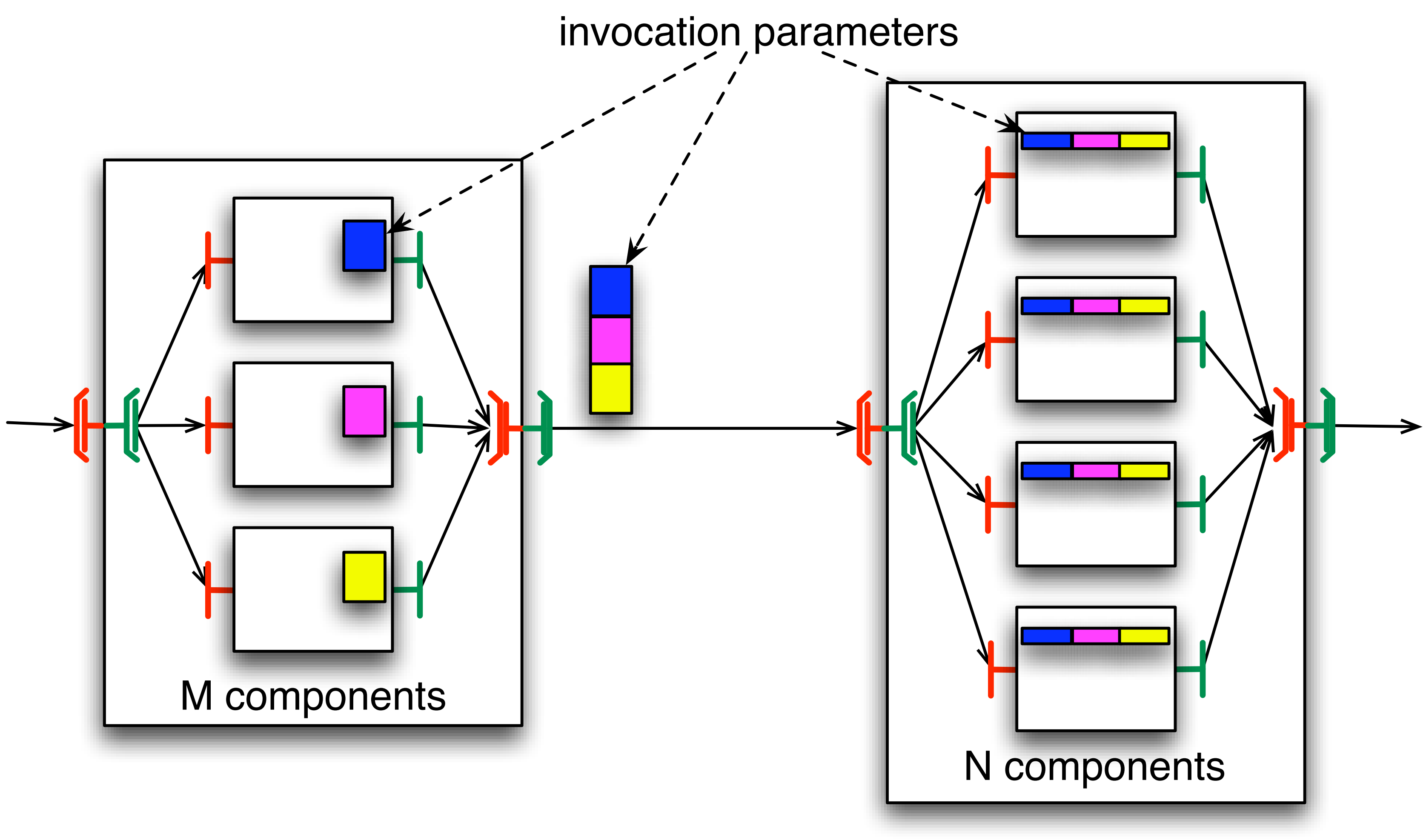
The MxN Problem (2)



The MxN Problem (3)



The MxN Problem (4): data distribution



Summary of Collective Communications

- Simple way of specifying collective operations
- + definition at the level of the interfaces ➔ better for verification and specification
- Rich high levels spec of synchronisation (especially gathercast)
- Easier to optimize
  - The MxN case: synchronisation issues, complex distribution policies avoid bottleneck

A few things we did not cover

- SPMD programming and Synchronization Barriers, cf gathercast???
- Group communications ~ Multicast
- Purely synchronous models -> Robert de Simone
- Shared memory models

- … and a lot of more complex communication models

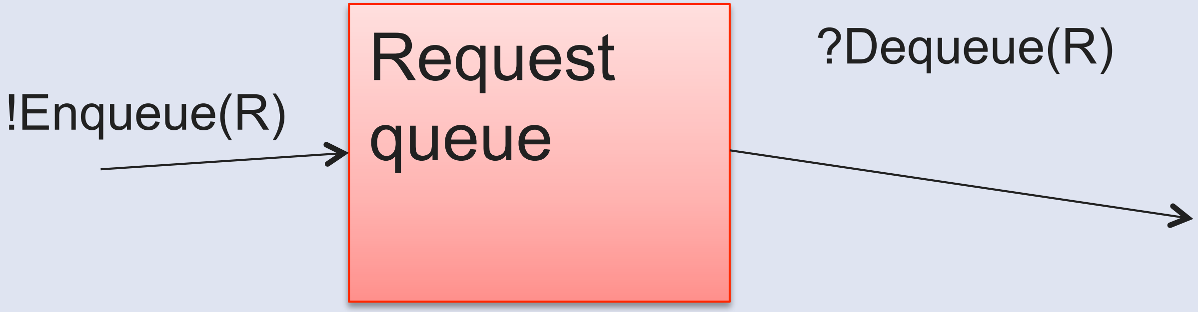
Conclusion

- An overview of asynchronism and different communication timings
- Applied to components with richer language constructs (futures, collective interfaces, …)
- Still a lot of other distributed computing paradigms exist (Ambient Talk, creol, X10 for example)
- A formalism for expressing communication ordering

Exercises

Exercise 1: Request queue

- In CCS with parameters (a value can be a request)
  - Express a request queue:

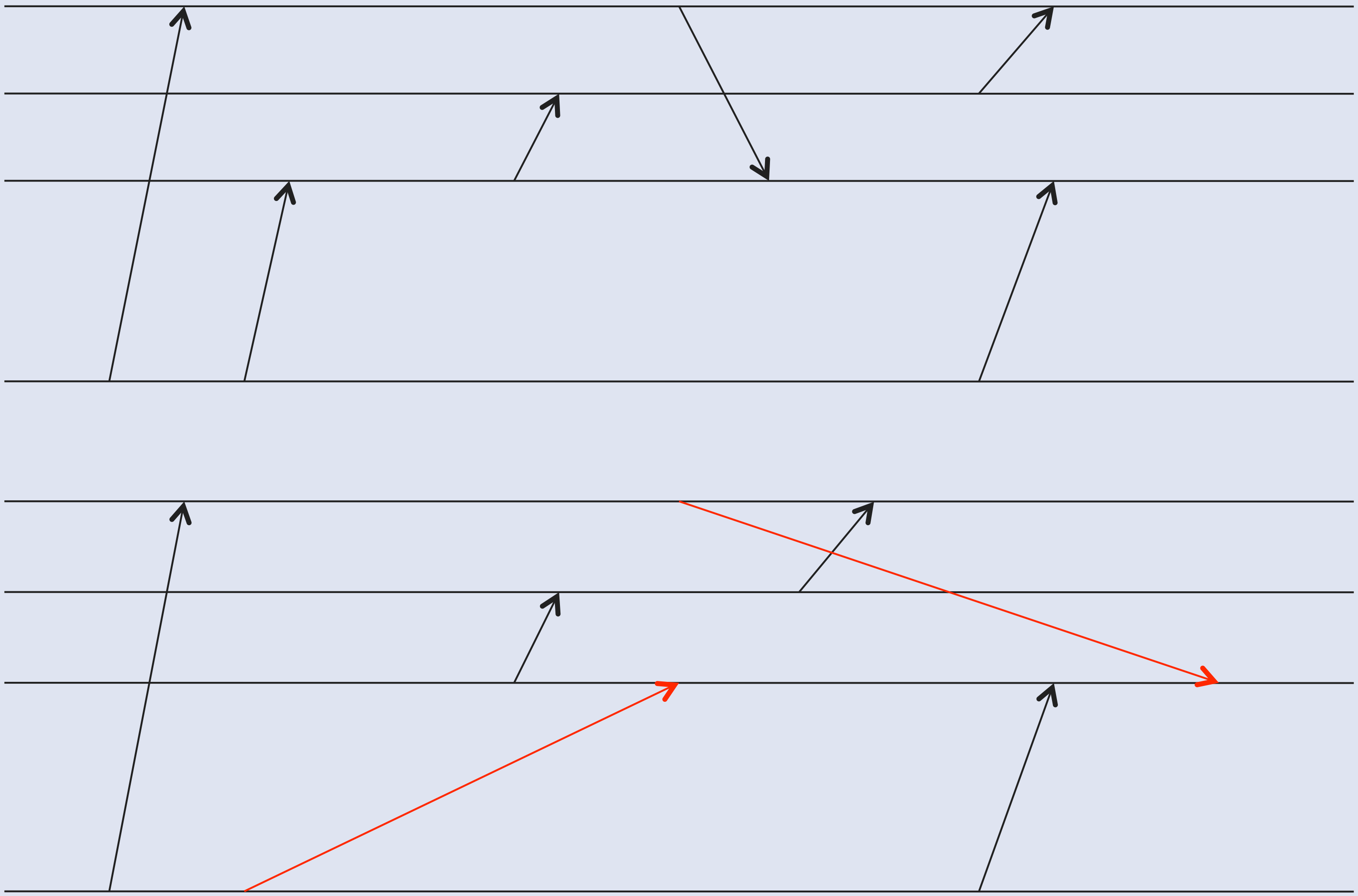


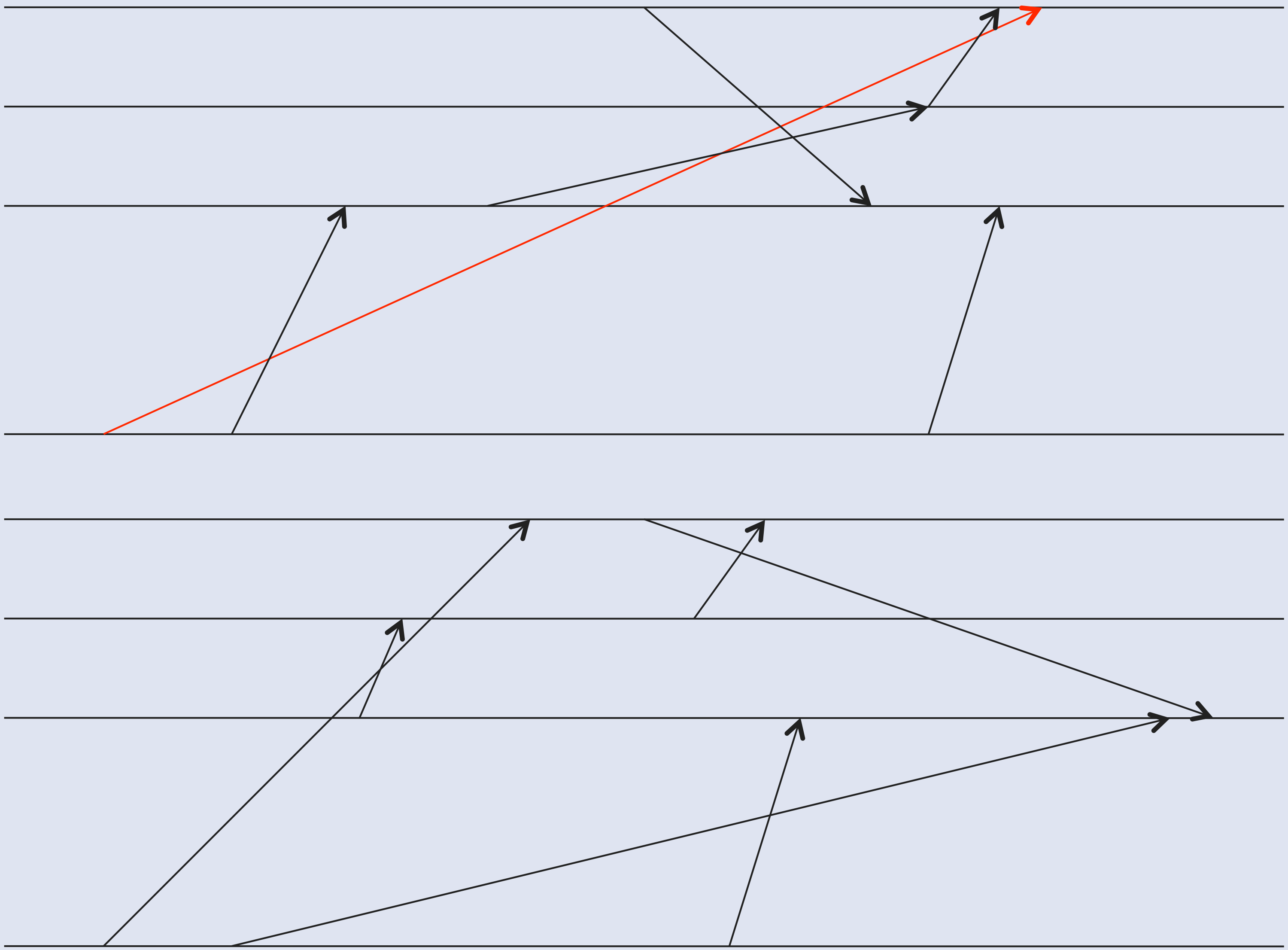
  - Also express 2 simple processes accessing it

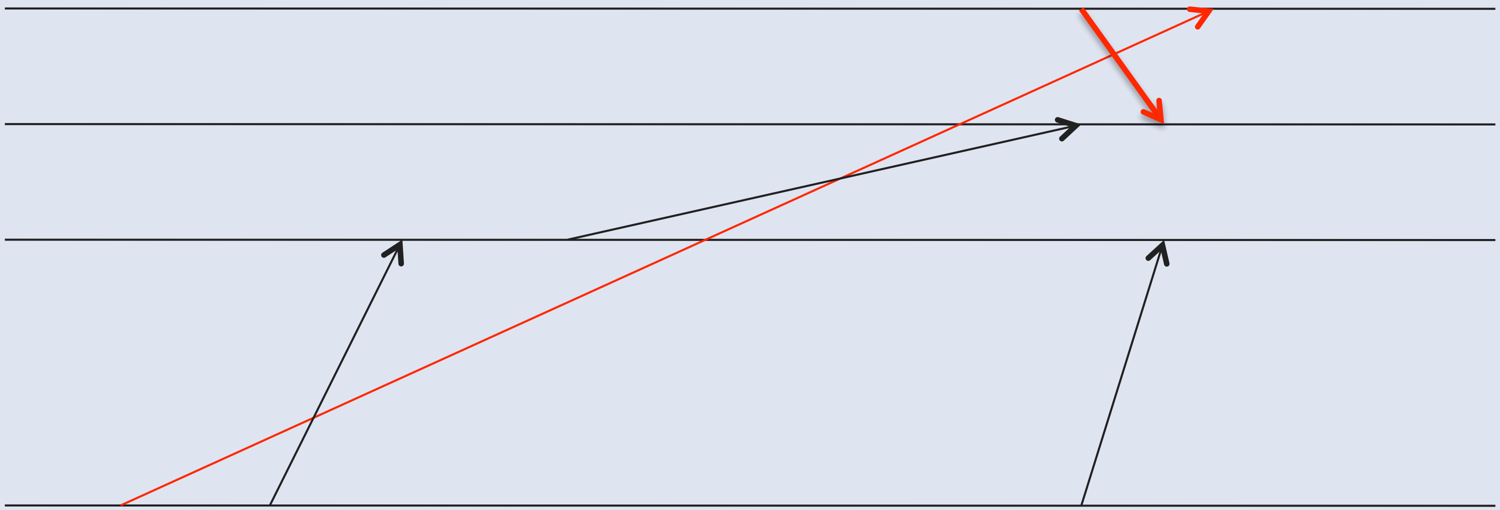
Hint from last course: $$Reg_i = \overline{read}(i).Reg_i + write(x).Reg_x$$

- Same thing in asynchronous CCS (without and with RDV)

Exercise 2: Are the execution CO, synchronous, asynchronous or FIFO?





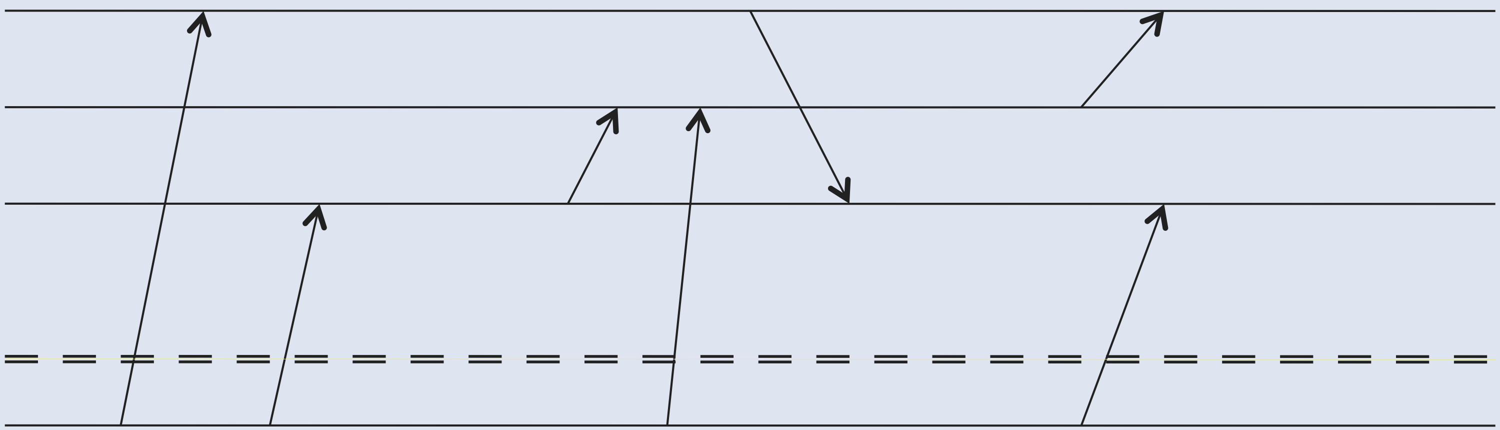


Exercise 3: find a solution to the deadlock slide 31

Exercise 4: Ensuring causal ordering with a sending queue

In the example below, suppose that the bottom thread has a sending queue, that is it sends all messages to an additional thread that emits the final messages.

- Draw the new message exchanges
- Is causal ordering still ensured?
- FIFO ?



Exercise 5: Ensuring causal ordering with many sending queues

- Same thing but with one sending queue per destination process
  - Draw the new message exchanges
  - Is causal ordering still ensured?
  - FIFO ?