Asynchronous communications: from calculi to distributed components

Asynchonous CCS

Communication timing

Asynchronous components

ludovic.henrio@inria.fr

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 ?

2

Processes Calculi – Asynchrony in CCS

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

P, Q	$::=$	0	inaction
	$\mid$	$\boxed{\mu.P}$	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, $\bar{a}$.P, τ.P
- An asynchronous version would be to allow only a.P, and τ.P, and simply $\bar{a}$ without suffix
- $\bar{a}$.P has to be replaced by ($\bar{a}$|P)

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

Exercise: rewrite the following example in asynchronous CCS:
$(a.\bar{b}.a + c.\bar{b}.c) \mid (\bar{a}.b.\bar{a} + \bar{c}.b.\bar{c})$

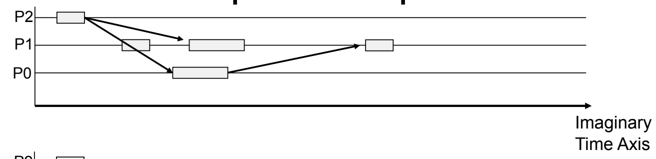
Communication Ordering; A Deeper Study

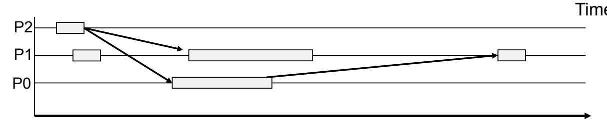
Synchronous, asynchronous, and causally ordered communication

Bernadette Charron–Bost, Friedemann Mattern, Gerard Tel

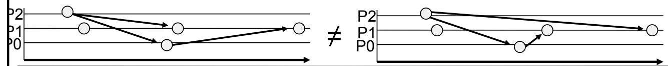
1996

Time and processes representation



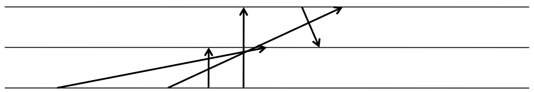


- these execution are identical -> event representation
- Only the order of message reception matters, whatever the transmission and execution duration



Happened Before Relation = Asynchronous Communication

- 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 LOCAL events)
- Global causality $\prec$, verifies at least

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

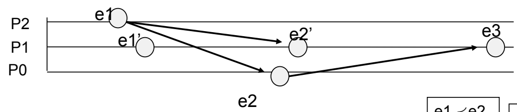
$s \prec r$ if $(s,r) \in \Gamma$

+ transitivity: If $e1 \prec e2$, and $e2 \prec e3$, then, $e1 \prec e3$

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

Happened-before relation

- Not all events are mandatorily related along $\prec$
  - Incomparable, independent, concurrent: ||
    - e1|| e2 if neither $e1 \prec e2$ nor $e2 \prec e1$
    - Non transitivity of ||

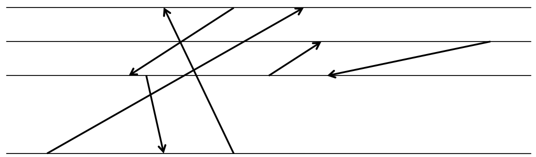


$e1 \prec e2$
$e1 \prec e2'$
$e2 \prec e3$
$e1 \prec e3$
$e1' \prec e2'$
$e2' \prec e3$
$e1' \prec e3$

e1 || e1'
e2 || e2'

e1' || e1
e2 || e1'
e2' || e2

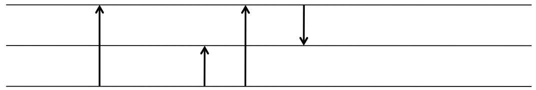
Exercise



- Why is the above execution not asynchronous?
- Make it a correct execution by changing just the red arrow
- Find 2 unrelated events

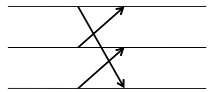
Synchronous communication

- Emission and reception is almost the same event



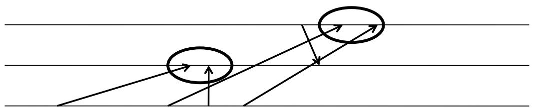
- A first characterization: Asynchronous +
  if $(s,r) \in \boldsymbol{\Gamma}$, then $a \prec s \Rightarrow a \prec r$ and $r \prec a \Rightarrow s \prec a$
  (still no cycle)
  strong common past, strong common future
- Or : in execution diagram messages can be **all** drawn vertically at the same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

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



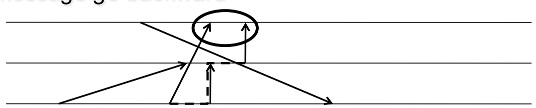
FIFO

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



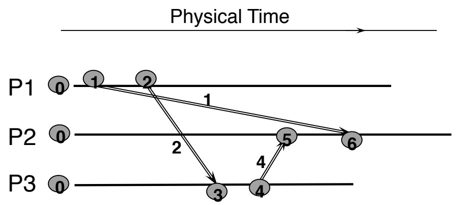
Causal Ordering

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



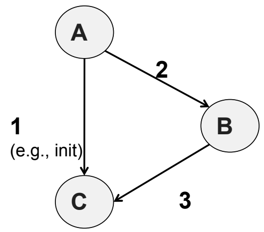
- Or no message is bypassed by a chain of messages

Causal ordering (2): Causality Violation



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

Causal ordering (3): The "triangle pattern"

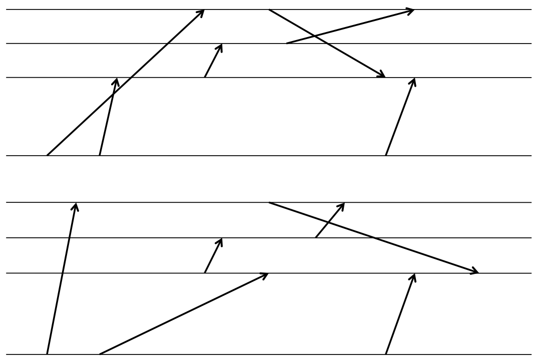


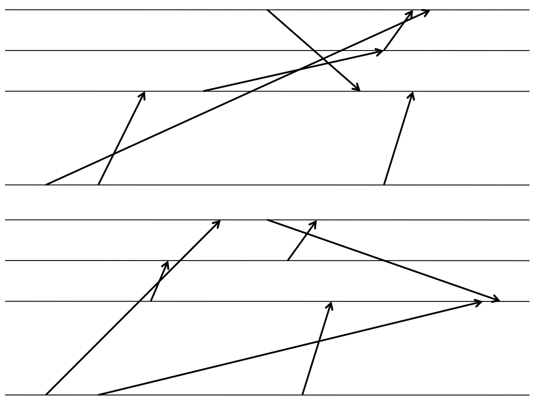
Objective: Ensure that **3** arrive at **C** after **1.**

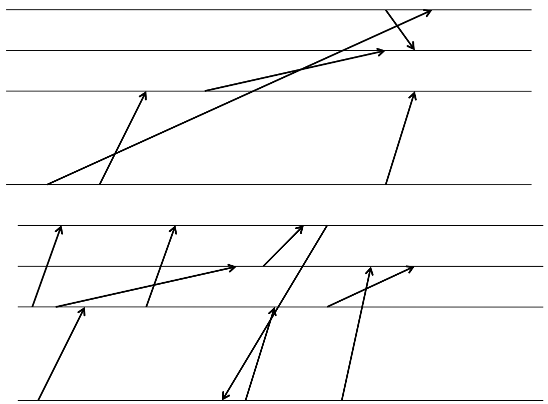
Summary of communicaiton orderings

- Asynchronous $\subset$ FIFO channels $\subset$ Causal ordering $\subset$ Synchronous
- Several characterization of communication timing (equations, diagram, …)
- 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

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







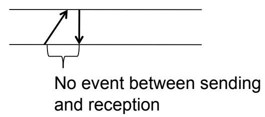
Weak common past – weak common future

- if $(s,r) \in \Gamma$, for a CO computation, then $a \prec r \Rightarrow$ NOT $s \prec a$ (WCP)
  and $s \prec a \Rightarrow$ NOT $a \prec r$ (WCF)

Exercise: find a computation that does not ensure weak common past
is it asynch FIFO CO or synch?

Exercise

- Rendez-vous:



Exercise: What does rendez-vous ensure?

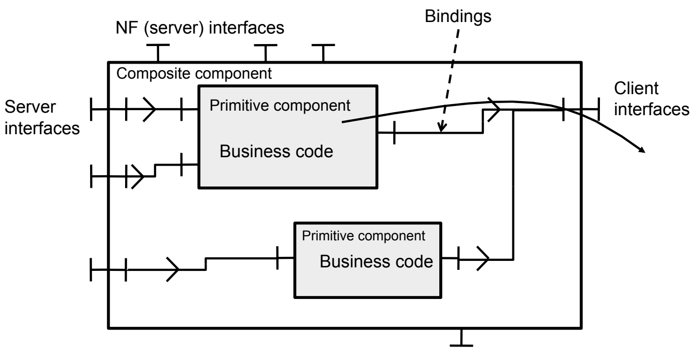
- So why is ProActive said asynchronous?

GCM: “Asynchronous” Fractal Components

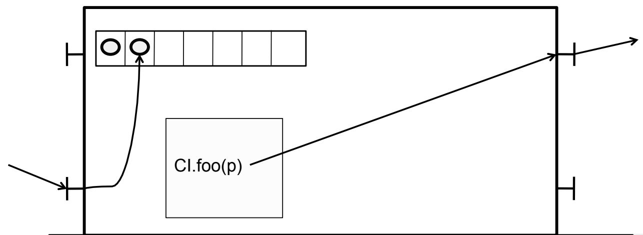
GCM – Quick Context

- 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

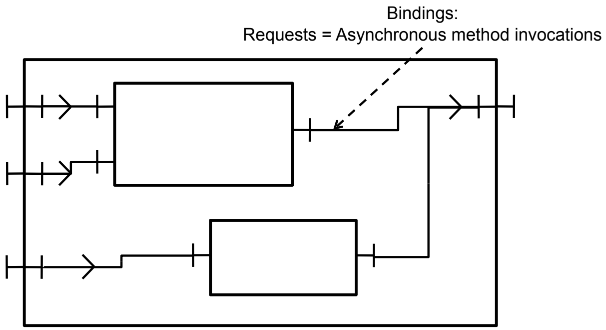


Primitive components communicating by *asynchronous* remote method invocations on interfaces (*requests*)

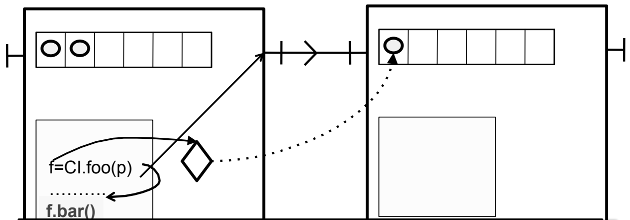
➔ **Components abstract away distribution and *concurrency***

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

Composition in GCM

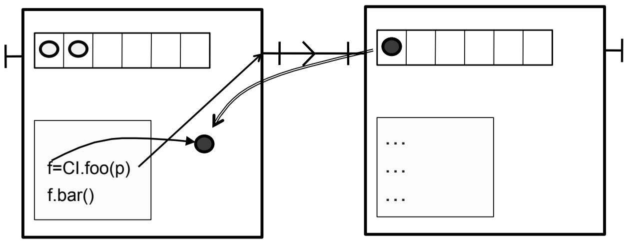


Futures for Components

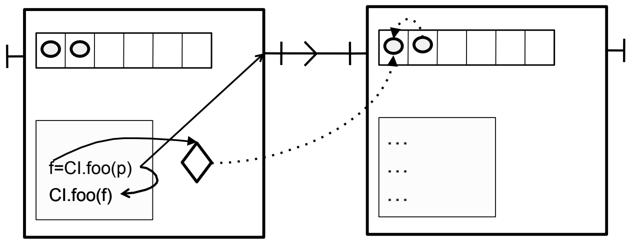


Component are independent entities
(threads are isolated in a component)
+
Asynchronous method invocations with results
↓
Futures are necessary

Replies

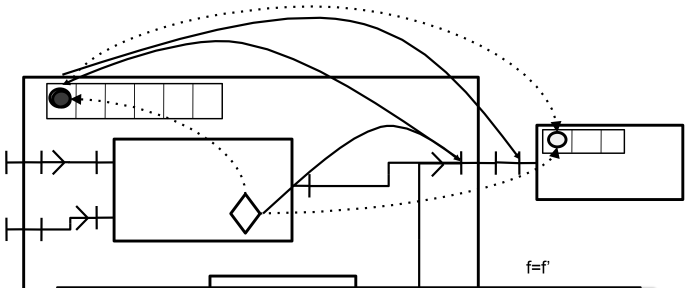


First-class Futures



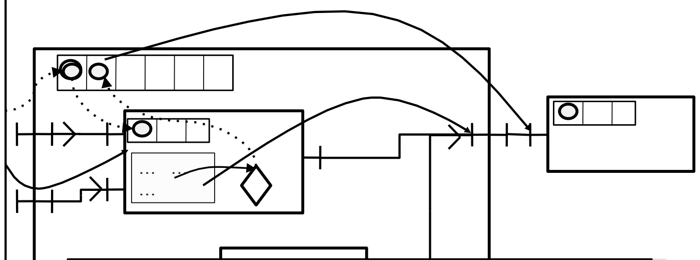
- **Only strict operations are blocking (access to a future)**
- **Communicating a future is not a strict operation**

First-class Futures and Hierarchy



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

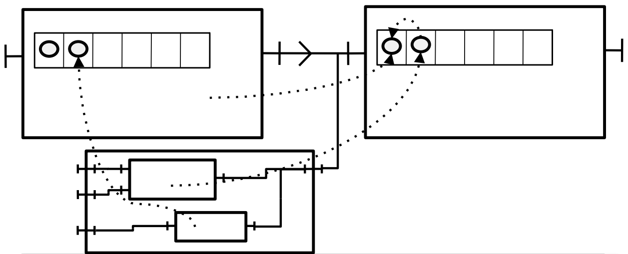
First-class Futures and Hierarchy



Almost systematic dead-lock in ProActive

A lot of blocked threads otherwise

Reply Strategies

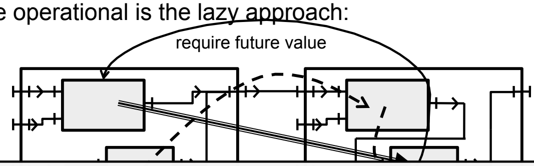


In ASP / ProActive, the result is insensitive to the order of replies (shown for ASP-calculus)

experiments with different strategies

Future Update Strategies

- How to bring future values to components that need them
- Different strategies can be envisioned
- A “naive” approach: Any component can receive a value for a future reference it holds.
- More operational is the lazy approach:



« On demand » future update

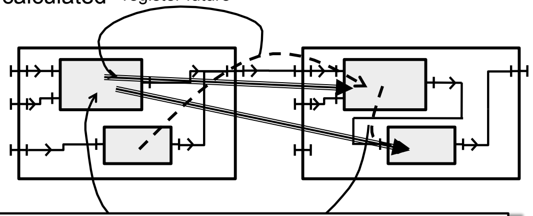
No-unnecessary transfer of values - Single step uptate

« registration delay + time for transfer »

Results stored for long term ➔ Not much operational.

Eager home-based future update

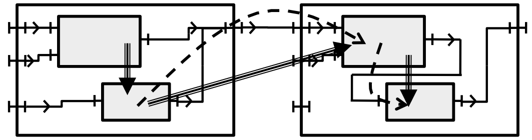
- A strategy avoiding to store future values indefinitely
- Relies on future registration and sends the value as soon as it is calculated



Results sent as soon as available - Un-necessary transfers
Every component with future reference registers
Garbage collection of computed results possible

Eager forward-based strategy

- Future updates follow the same path as future flow
- Each component remembers only the components to which it forwarded the future



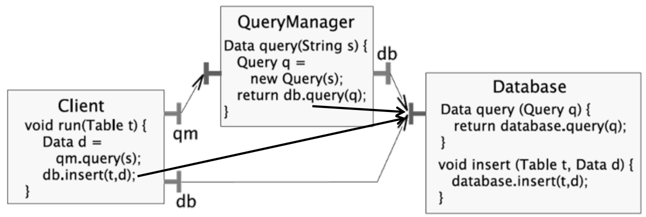
Results sent as soon as available
No registration required
Future updates form a chain ➔ intermediate components
Easy to garbage collect computed results

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:



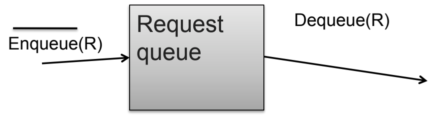
- Detecting deadlocks can be difficult ➔ behavioural specification and verification techniques (cf Eric Madelaine)

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

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

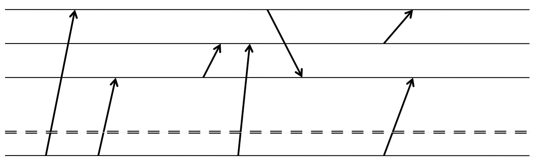
- Same thing in asynchronous CCS (without and with RDV)

Exercise 2: find a solution to the deadlock slide 37

Exercise 3: Ensuring causal ordering with a sending queue

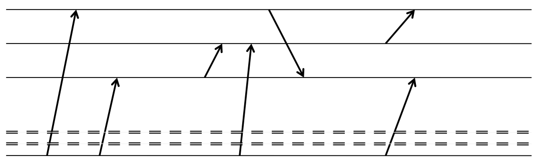
In the example below, suppose that the bottom thread has a sending queue, that is it sends all messages to an additional thread that emits the final messages.

- Draw the new message exchanges
- Suppose the communications are synchronous, what is lost by adding this new thread? what is the new overall ordering (what if CO, FIFO, or asynch?)



Exercise 4: Ensuring causal ordering with many sending queues

- Same thing but with one sending queue per destination process
  - Draw the new message exchanges
  - Suppose the communications are synchronous, what is lost by adding this new thread? what is the new overall ordering (what if CO, FIFO, or asynch?)



Pointeurs pour exposés SSDE

wikipedia Model-checking:
http://en.wikipedia.org/wiki/Model_checking

Sites

- SPIN: http://spinroot.com/
- SMV: http://www-cad.eecs.berkeley.edu/~kenmcmil/
- PSL/SuGaR: http://www.pslsugar.org/
  http://www.haifa.il.ibm.com/projects/verification/sugar/
- Ptolemy: http://ptolemy.eecs.berkeley.edu/
- Metropolis: http://www.gigascale.org/metropolis/
- Bandera: http://bandera.projects.cis.ksu.edu/
- Blast: http://www-cad.eecs.berkeley.edu/~blast/
- Slam: http://research.microsoft.com/slam/
- SPEC#: http://research.microsoft.com/specsharp/
  http://spex.projects.cis.ksu.edu/spex-jml/
- AmbientTalk: http://prog.vub.ac.be/amop/
- Fractal: http://fractal.objectweb.org/documentation.html

Sites

- SCA+ Frascati:
  http://www.davidchappell.com/articles/Introducing_SCA.pdf
  http://wiki.ow2.org/frascati/
- AltaRica/ARC:
  http://altarica.labri.fr/tools:arc
  http://altarica.labri.fr/api-docs/current/arc/arc-handbook.pdf
- Divine:
  http://divine.fi.muni.cz/page.php?page=overview
  http://divine.fi.muni.cz/page.php?page=language
- MCRL2
  http://www.mcrl2.org/mcrl2/wiki/index.php/Tool_manual_pages
  http://www.mcrl2.org/mcrl2/wiki/index.php/MCRL2_primer