IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Interaction and Verification

A. Beugnard
SIIA – IV – C2
2021

Why do we communicate ?

- Coordination, cooperation, deal, …

What do we exchange ?

- Information (status, results, intentions, …)

How do we communicate ?

- By observation (1 active/1 passive)
- By sharing (canals, memory, conventions) messages (many active actors ; ex. expeditors, receivers)

- Transport protocol (shared)
- Communication language (shared)
- Interaction protocol (shared)

This course focusses on Interaction protocol.

- Whom are we communicating with
- How initiating an exchange
- (Out of scope) Effect of the communication

Classification criteria ?

- Active/passive actors
- How many actors : 2, more than 2
- Actor's roles : symmetric or not
- Who initiate the communication
- Any shared state
- Asynchronous or synchronous
- Blocking or not
- Message order ensured (FIFO) [asynchrone]
- Loss of message [fault management]

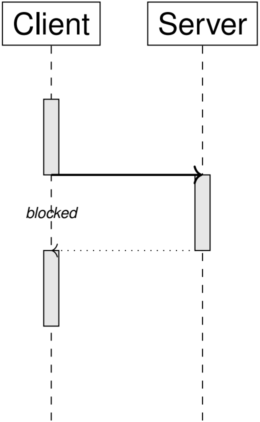
Classification criteria ?

- Let know
- Request for information
- Request for doing
- Answers
- Promisses
- Proposals
- Deals
- Choose, elect, decide
- …

2-interaction

4 models :

- Synchronous
- Asynchronous
- Future
- By necessity [Car93]



Client side

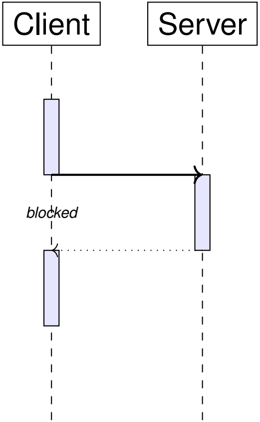
- call
- wait result
- get result
- continue

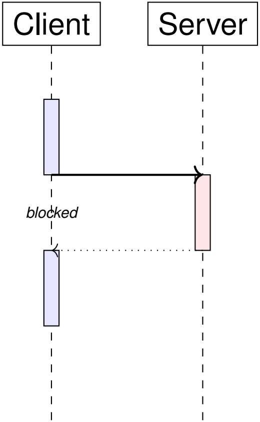
Nit visible (internal)

Serveur side

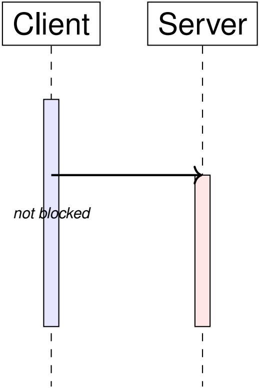
- wait call
- compute
- return result

Visible in interface





Or Rendezvous (ADA)



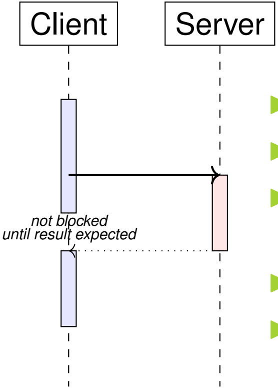
Client side

- send message
- continue

Serveur side

- wait message
- compute

We must have 2 threads . . .(either on the same machine, either on remote ones).



Client side

- call
- continue
- wait result when needed
- get result
- continue

Server side

- wait call
- compute
- return result

We must have 2 threads …(either on the same machine, either on remote ones).

Abstraction and implicit mechanism that behaves as :

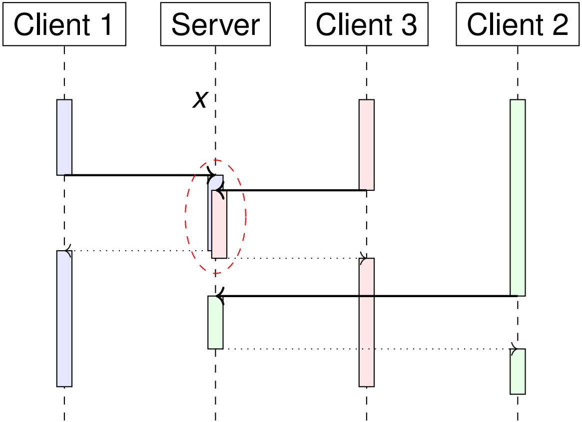
- Asynchronous ; if the result is not required
- Synchrone ; if the result is immediately required
- Future ; if the result is required later

Interactions are interesting only among active entities[1]

Passive entities are only useful for sharing states.

Sharing state is difficult ; it introduces mutual exclusion issues (safety) and deadlocks issues (vivacity).

1. With a single thread, communication can oly be synchronous.



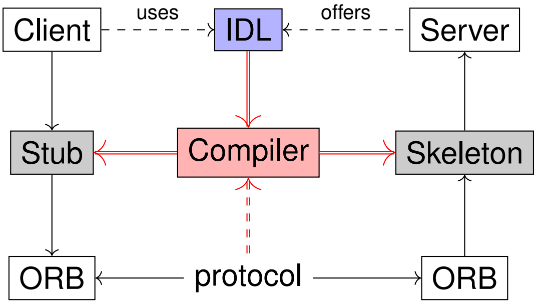
Competition fot accessing the shared state x.

Safety
: properties ensure nothing wrong happens. For instance ; invariant satisfaction or mutex . . .

Vivacity
: properties ensure that something happens. For instance, no deadlocks.

Synchronous	**Asynchronous**	**Future**	**By necessity**
ADA, C, Caml, functional, object	dart, erlang, elixir, via libraries	via libraries	ProActive [OW217], via libraries

Previous properties apply to remote (or heterogeneous) interactions.



This is the principle of RPC, CORBA, Java RMI, .NET, etc. connectors

2+-interaction

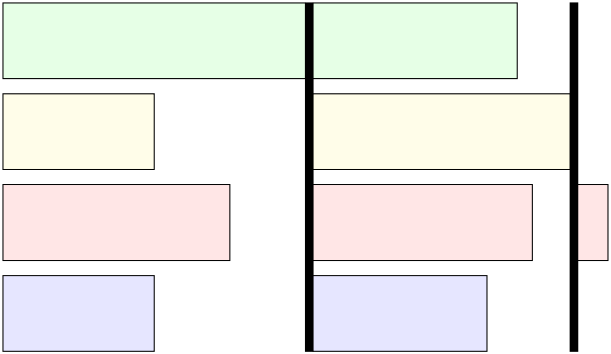
Interactions with more than 2 actors :

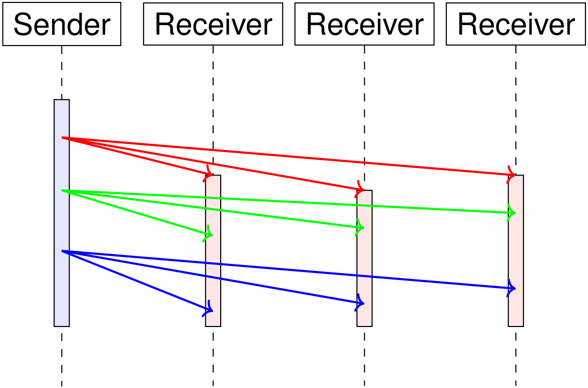
- Synchronization barrier
- Broadcast
  - Asynchronous (ex ; UDP)
  - With quaranties
- Consensus
- Group membership
- Tuple space
- Conversational language

More abstract interactions :

- Publish/subscribe [EFGK03]
- Negotiation
- Vote
- Auction
- …
- Communication abstractions

A basic coordination mechanism that ensures that entities have reached a specific point (the barrier) to continue their activity.





Properties

- No message loss
- Non message replication
- Fairness : all receive
- Atomicity : all or none
- Keep messages sequence

UDP : loss, non guaranty
reliable broadcast algorithms.
Ensure message order (red, green, blue)

Ensures that a message sent to a group is received by all or none.

Distributed systems study the interactions between processes (machine, actors, agents) by taking into account :

- The transmission delays
- The potential errors of the actors
- The potential errors of the communication channels

Distributed algorithms propose solutions to control the properties when theoretical solutions exist.

Messages are not transmitted instantly ; they can be lost.
It is impossible to distinguish between a lost message and a very long transmission time.
The notion of global time is meaningless - each actor has his own time. It is impossible to date an event in an absolute manner. event.
The notion of *causality* must be reconstructed ; lamport clock, vector clock, etc.

m_1 causally precedes m_2 ($m_1 \rightsquigarrow m_2$) iff :

- p sent m_1 before sending m_2
- p received m_1 then sent m_2
- There exists m_3 so that $m_1 \rightsquigarrow m_3 \wedge m_3 \rightsquigarrow m_2$

It must be taken into account that an actor (process, program, machine, channel, network, human, etc.) can make mistakes.

- By omission ; forgets to send a message, to reply, . . .
- Arbitrary ; sending the wrong message (voluntarily[2] or not)

For omissions, the simplest model consists of considering that an actor breaks down (crash-stop model) ; when he fails to send a message, it fails to send all the following

2. the actor is said to be malicious or Byzantine.

Perfect (or Reliable) links (PL)

- (Validity) If p_i and p_j are correct, then any messagesent by p_i to p_j is eventually delivered to p_j
- (No duplication) No message iis delivered more than once
- (No creation) No message is delivered without being sent

Reliable FIFO links (FIFO)

- Perfect links
- (FIFO) Messages are delivered in the same order as they are sent.

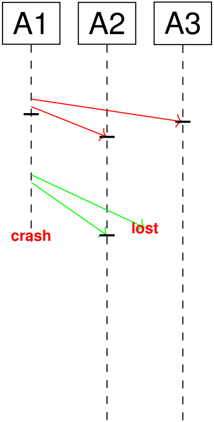
In this course, we assume channels are Perfect Links.

Best-effort Validity, No duplication, No creation (as PL)

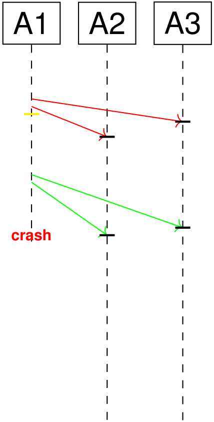
Reliable BE + Agreement : If a message *m* is delivered to a correct receiver, then all correct receivers will receive the message.

Uniform BE + Uniform agreement : For all message *m*, if a receiver get *m* then all correct receivers get it.

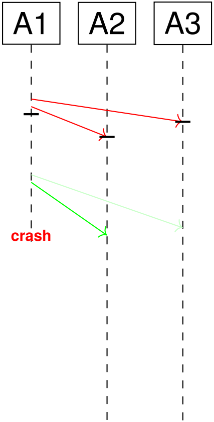
See R. Guerraoui courses for algorithms descriptions.



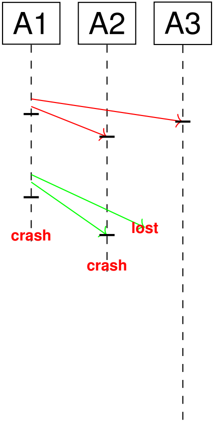
A1 do its best, but while correct A3 does not receive a message A2 received.



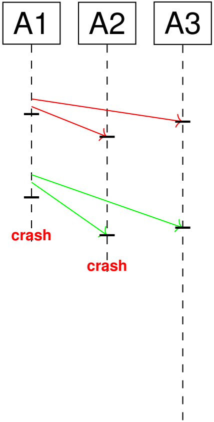
All correct processes receive the message.



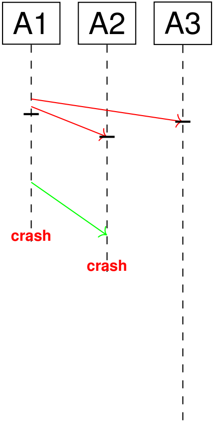
Or none.



Reliable broadcast, since A2 being not correct, *m* can be lost.

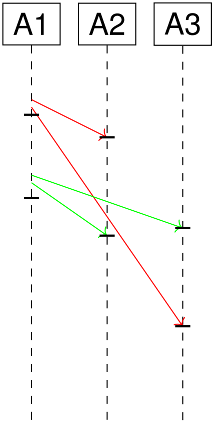


Evn if a receiver crashes, all receive…

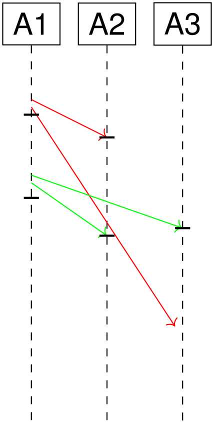


…or none.

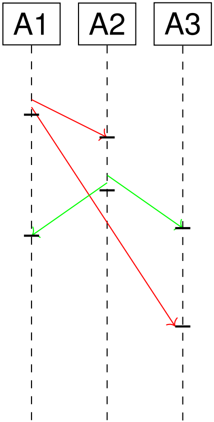
Ensure order of messages.
If p reveives m_2 then p received all m such that $m \rightsquigarrow m_2$



No.

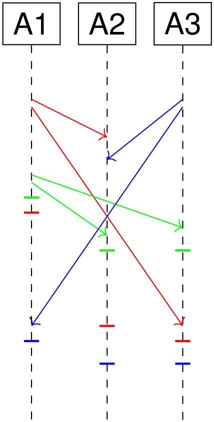


No.

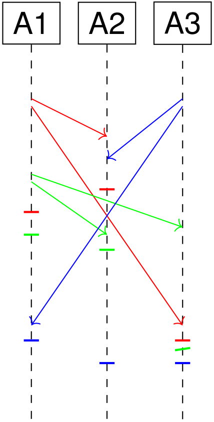


No.

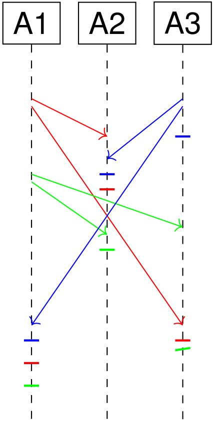
Ensure all processes see the same message order.
If the order ensure causality, it's a total causal order.



Total, not causal



Total and causal



Total and causal

Blue messages are causally independent from red and green ones.

Reliable broadcast abstractions

Best-effort broadcast

- Guarantees reliability *only if sender is correct*

Reliable broadcast

- Guarantees reliability *independent of whether sender is correct*

Uniform reliable broadcast

- Also *considers behavior of failed nodes*

Total reliable broadcast

- Reliable broadcast *with same delivery order for all correct nodes*

Causal reliable broadcast

- Reliable broadcast *with causal delivery order*

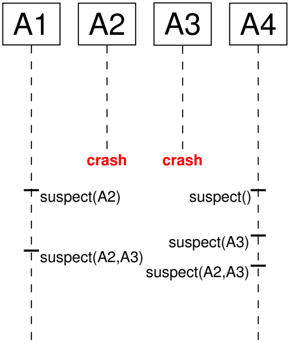
Total order can be implemented thanks to a consensus algorithm.
Consensus

- (Validity) The chosen value has been proposed
- (Uniform agreement) : Two different correct processes do the same choice
- (Termination) Any correct process eventually choose
- (Integrity) Any process choose once, at most

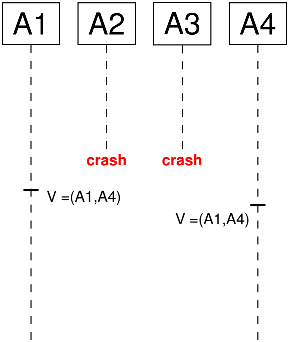
How can we know all processes involved in an interaction ?

- In case of failures
- When processes come and leave

How can we ensure all processes share the same view (list of involved processes).



No coordination



Coordination

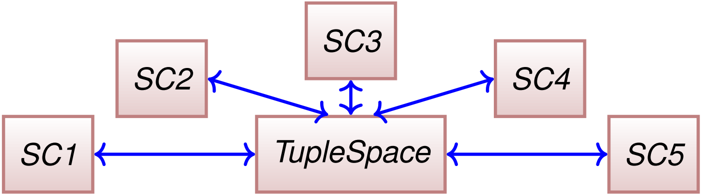
- Actors are informed of crashes, entries and exits ; actors are said to *install* views
- We assume no loss of information
- Actors install all the same sequence of views.

Taking into account only crashes (neither entries nor exits) :

- (local monotony) If an actor installs a view (j, M) after installing (k, N), then $j > k$ and $M \subsetneq N$
- (Agreement) No pairs of actors install views (j, M) and (j, M') such that $M \neq M'$
- (Completness) If an actor *a* crashes, then there exists *j* an integer such that any actor eventually install a view (j, M) such taht $a \notin M$
- (Precision) If an actor *a* installed a view (i, M) and $a \notin M$ then *a* crashed

- Coordination languages (à la Linda)
- Conversation langages (ex. RCA)

Original model : Linda [ea94]



Linda introduced 4 opérations :

- **in** read and remove atomically a tuple
- **rd** read, and keep unchanged, a tuple
- **out** add a tuple (possible replication)
- **eval** create a new process

A tuple describes a journey :

- (destination, date, duration, cost, properties).

Processes (travel agency) produce offers (out).

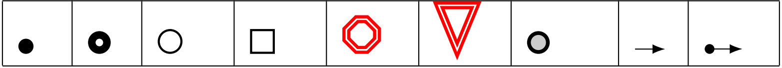
Processes (client, travel agency) consult them (rd) or book them (in).

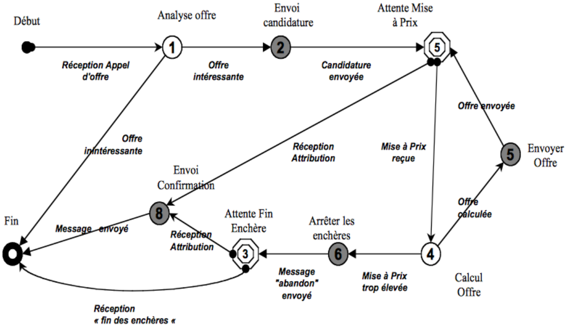
- Simple et abstract mechanism.
- No coupling among processes ; no need to know each others.
- The protocol is encoded in the tuple

Implementations : CppLinda, Erlinda, JavaSpace, PyLinda, etc.

Example : Graphes RCA [TE99]

État							Transition	
initial	final	elementary action	composite action	unbounded wait	bounded wait	communi-cation	internal	external





Good points

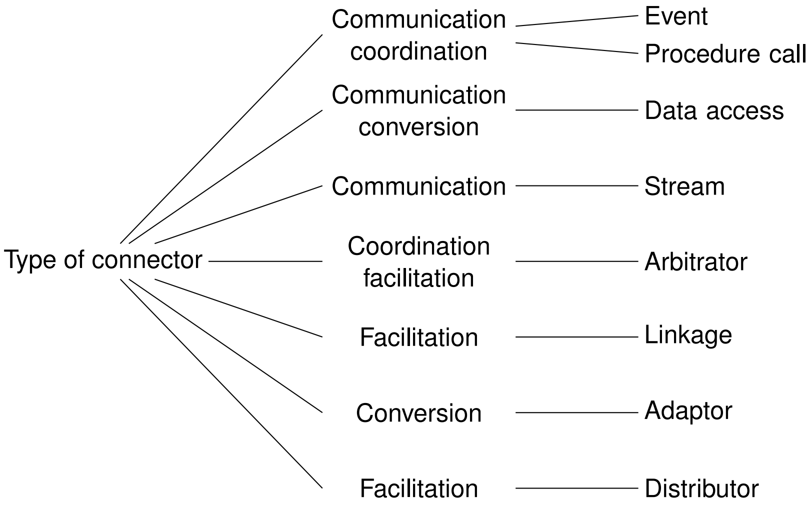
- Globale overview
- Vision of time (automata)

Negative points

- No roles
- No dynamic (numbers of actors)

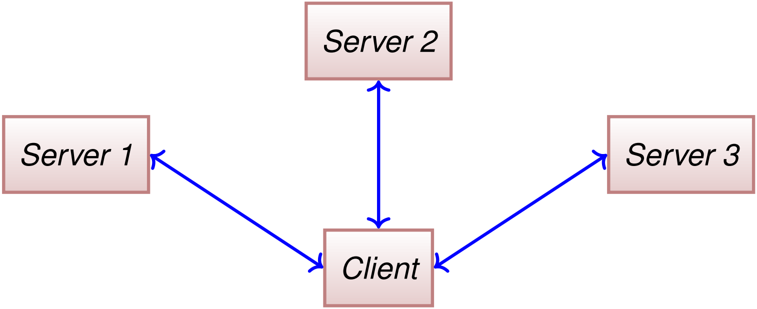
Connectors

- Number of actors, roles (*unicast, multicast, broadcast*)
- Direction
- Initiator (*push/pull*)
- Synchronous/asynchronous (blocking)
- Stream/unique
- Policy (exact, *best effort*, ACID, etc.)
- Safety, cyphered (kind of policy)
- Size, rhythm, jitter (*jitter*), bandwidth

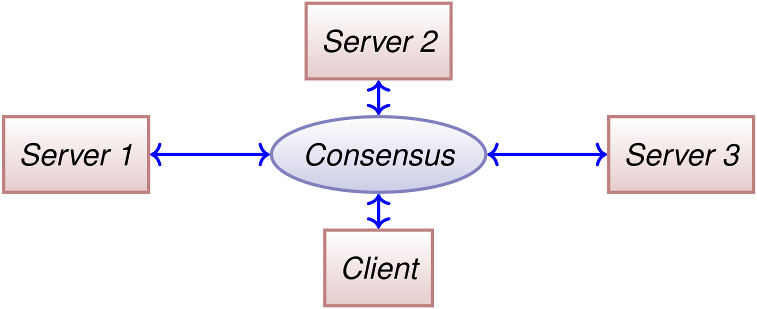

Type of connector
Communication coordination
Communication conversion
Communication
Coordination facilitation
Facilitation
Conversion
Facilitation
Event
Procedure call
Data access
Stream
Arbitrator
Linkage
Adaptor
Distributor

- Memory (register, table, stack, etc.)
- Protocol/language
- Transaction

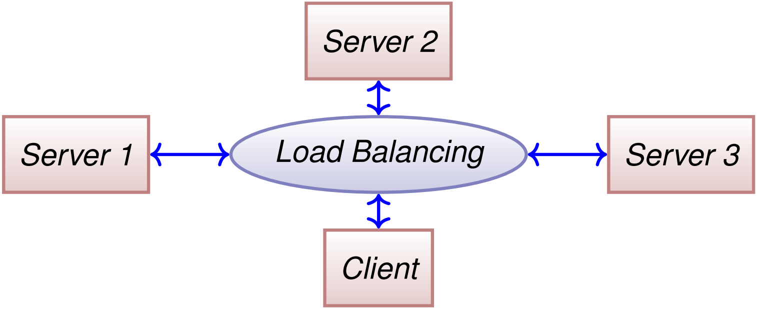
These means are interdependent (protocols and transactions use memory) ; it is the usage rules and policies that differentiate them.



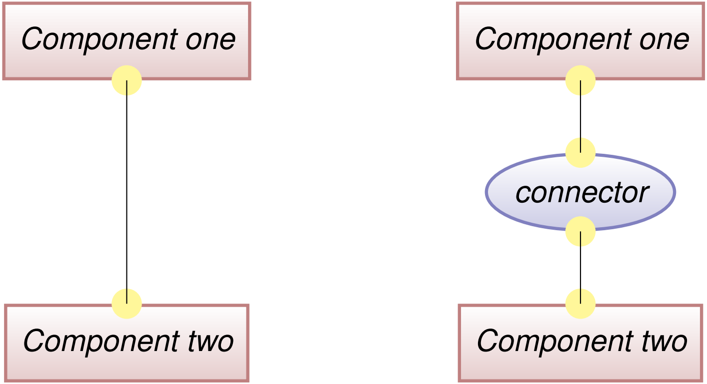
Is it a client that can choose between 3 servers, a load balancing system or a load balancing system or a redundant system with consensus ?

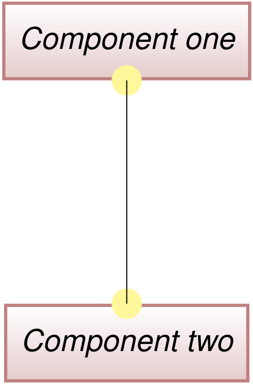


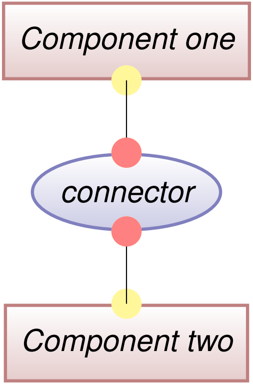
Is it a client that can choose between 3 servers, a load balancing system or a load balancing system or a redundant system with consensus ? More ambiguity.

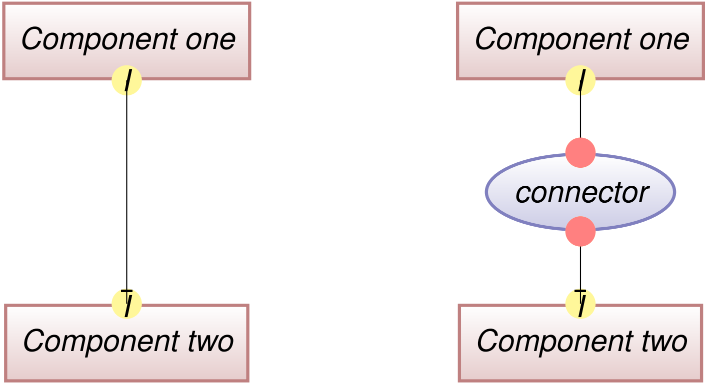


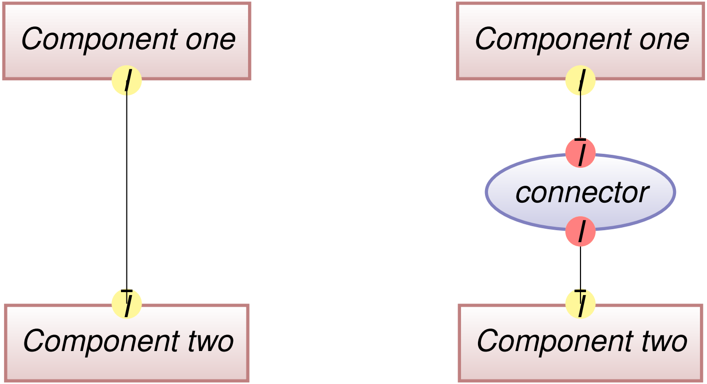
Is it a client that can choose between 3 servers, a load balancing system or a load balancing system or a redundant system with consensus ? More ambiguity.

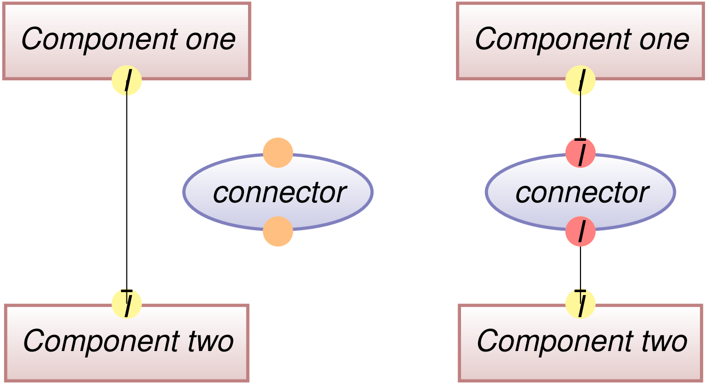

Component one
Component two
Component one
connector
Component two

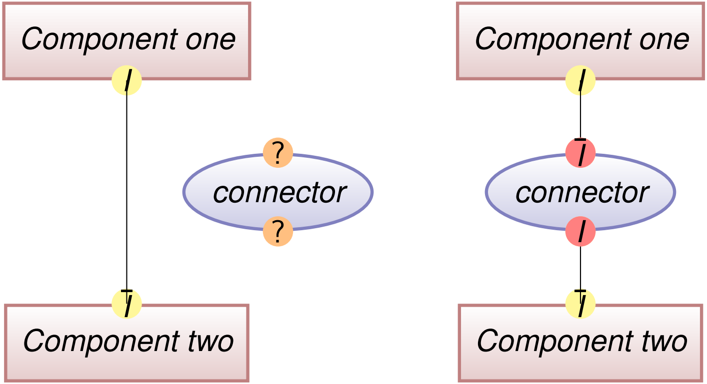

Component one
Component two


Component one
connector
Component two


Component one
Component two
Component one
connector
Component two

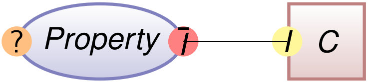

Component one
I
I
Component two
Component one
I
I
connector
I
I
Component two


Component one
connector
Component two
Component one
connector
Component two

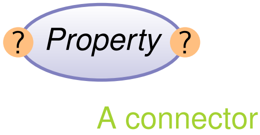

Component one
?
connector
?
Component two
Component one
connector
Component two

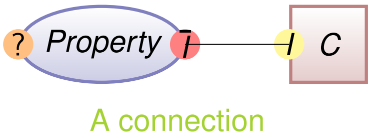
A connector

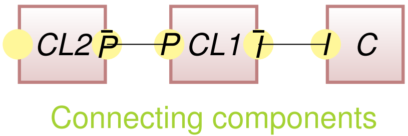
A connector

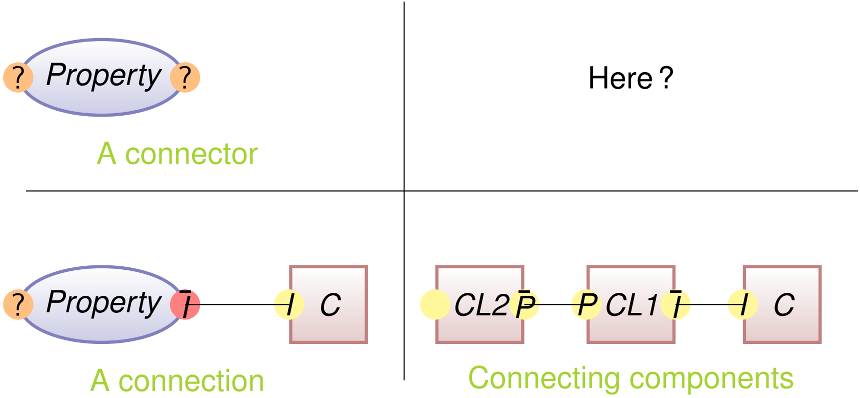


A connection


?
Property
?
A connector

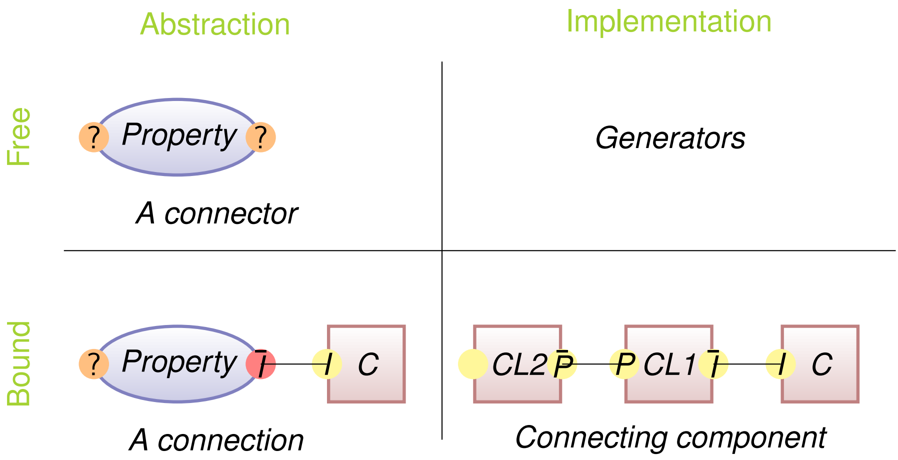

?
Property
$\bar{I}$
I
C
A connection


$CL2$
$\bar{P}$
P
$CL1$
$\bar{I}$
I
C
Connecting components


? Property ?
A connector
Here ?
? Property $\bar{I}$ — I C
A connection
CL2 $\bar{P}$ — P CL1 $\bar{I}$ — I C
Connecting components

Something that transforms a role, connected to a port – hence a socket – into a component that provides the complementary interface to the port and *ensures the property* of the connector.
Quelque chose qui *transforme* un rôle, relié à un port – donc une prise – en un composant qui offre l'interface complémentaire du port *et qui assure la propriété* du connecteur.

ex : en Corba, .NET, RPC, Java RMI. . .stub and skeleton generators
Stubs and skeletons are connecting components.


Abstraction
Implementation
Free
Bound
?
Property
?
A connector
Generators
?
Property
Ī
I
C
A connection
CL2
P̄
P
CL1
Ī
I
C
Connecting component

- Procedure calls (PC)
- Remote procedure call (RPC)
- CORBA, RMI, ...
- Client/server with load balancing
- Client/server with consensus
- etc

Many connectors exist ; sometime independant from the component model (ex. protocols), sometime associated to a model (CORBA RPC).

Protocols are“connectors” found on the shelves as components, with an explicit interface (port = API).

Using a component as connector requires to *adopt* its interface as a communicating protocol.

A connector is delivered as a generator.

D Caromel.
Toward a method of object-oriented concurrent programming.
Communications of the ACM, 1993.

Carriero et al.
The linda alternative to message-passing systems.
Parallel Computing, 2(4) :633—-655, April 1994.

P Eugster, P Felber, R Guerraoui, and A Kermarrec.
The Many Faces of Publish/Subscribe.
ACM Computing Surveys (CSUR), 2003.

N Mehta, Nenad Medvidovic, and S Phadke.
Towards a taxonomy of software connectors.
Proceedings of the 22nd international conference on Software Engineering (ICSE), pages 178–187, 2000.

OW2.
Proactive web site.
`http://proactive.activeeon.com//`, 2017.

Erwan Tranvouez and Bernard Espinasse.
Protocoles de coopération pour le réordonnancement d'atelier.
In *JFIADSMA*, 1999.