Implementation of Distribution in Process algebras

`Jean-Jacques Lévy`

INRIA Rocquencourt

August 26, 2005

Plan

π-calculus and distribution

Join-calculus

Mobile ambients and distribution

Conclusion

Plan

π-calculus and distribution

Join-calculus

Mobile ambients and distribution

Conclusion

Plan

π-calculus and distribution

Join-calculus

Mobile ambients and distribution

Conclusion

Plan

π-calculus and distribution

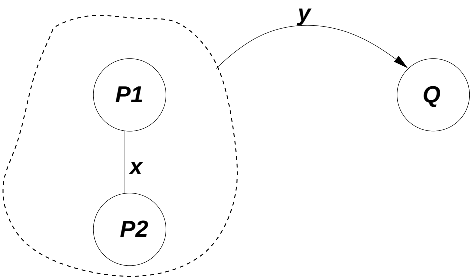
Join-calculus

Mobile ambients and distribution

Conclusion

Process algebras and Localisation (1/2)

- π-calculus is mobility of names
  - scopes $+$ scope extrusion
  - $((\nu x)\overline{y}(x).P) \mid y(z).Q \longrightarrow (\nu x)(P \mid Q[\![x/z]\!])$

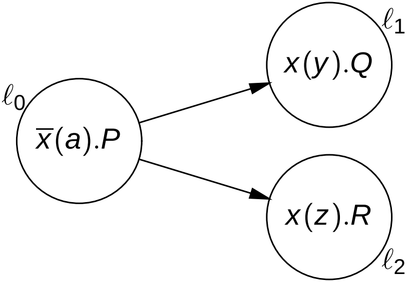


Process algebras and Localisation (2/2)

- π-calculus $\neq$ physical mobility
  - channels are not located
  - In a distributed environment, consider

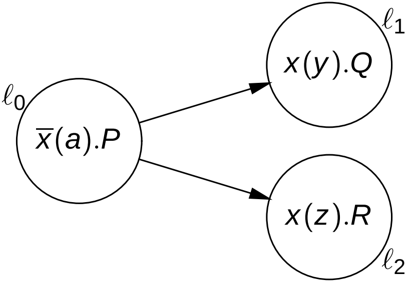
    $$\overline{x}(a).P \mid x(y).Q \mid x(z).R \;\longrightarrow\; P \mid Q[\![a/y]\!] \mid R \quad \text{or} \quad P \mid Q \mid R[\![a/z]\!]$$

  - when sender $\overline{x}$ is at location ℓ_0 and receptors x are at locations ℓ_1 and ℓ_2 ??
  - but easy to implement when $\ell_0 = \ell_1 = \ell_2$.
  - or when $\ell_1 = \ell_2$ and $P = 0$.



Process algebras and Localisation (2/2)

- π-calculus $\neq$ physical mobility
  - channels are not located
  - In a distributed environment, consider
    $$\overline{x}(a).P \mid x(y).Q \mid x(z).R \longrightarrow \begin{array}{l} P \mid Q[\![a/y]\!] \mid R \\ \text{or} \quad P \mid Q \mid R[\![a/z]\!] \end{array}$$
  - when sender $\overline{x}$ is at location ℓ_0 and receptors x are at locations ℓ_1 and ℓ_2 ??
  - but easy to implement when $\ell_0 = \ell_1 = \ell_2$.
  - or when $\ell_1 = \ell_2$ and $P = 0$.

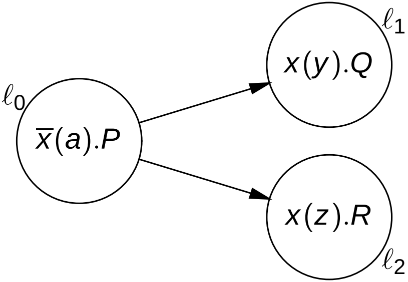


Process algebras and Localisation (2/2)

- π-calculus $\neq$ physical mobility
  - channels are not located
  - In a distributed environment, consider

$$\overline{x}(a).P \mid x(y).Q \mid x(z).R \longrightarrow \begin{array}{rl} & P \mid Q[\![a/y]\!] \mid R \\ \text{or} & P \mid Q \mid R[\![a/z]\!] \end{array}$$

  - when sender $\overline{x}$ is at location ℓ_0 and receptors x are at locations ℓ_1 and ℓ_2 ??
  - but easy to implement when $\ell_0 = \ell_1 = \ell_2$.
  - or when $\ell_1 = \ell_2$ and $P = 0$.

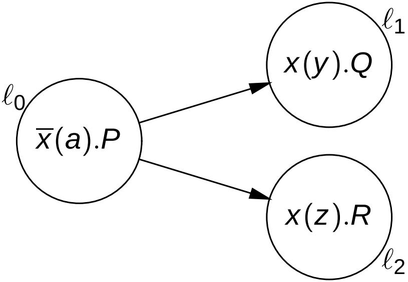


Process algebras and Localisation (2/2)

- π-calculus $\neq$ physical mobility
  - channels are not located
  - In a distributed environment, consider

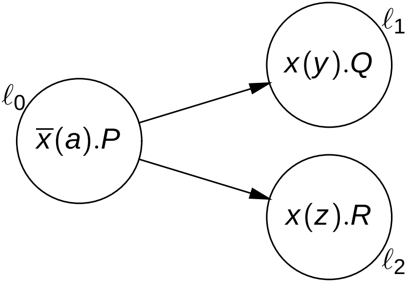
$$\overline{x}(a).P \mid x(y).Q \mid x(z).R \;\longrightarrow\; \begin{array}{rl} & P \mid Q[\![a/y]\!] \mid R \\ \text{or} & P \mid Q \mid R[\![a/z]\!] \end{array}$$

  - when sender $\overline{x}$ is at location ℓ_0 and receptors x are at locations ℓ_1 and ℓ_2 ??
  - but easy to implement when $\ell_0 = \ell_1 = \ell_2$.
  - or when $\ell_1 = \ell_2$ and $P = 0$.



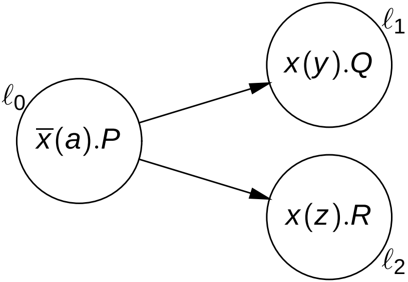
Process algebras and Localisation (2/2)

- π-calculus $\neq$ physical mobility
  - channels are not located
  - In a distributed environment, consider
    $$\overline{x}(a).P \mid x(y).Q \mid x(z).R \quad\longrightarrow\quad \begin{array}{ll} & P \mid Q[\![a/y]\!] \mid R \\ \text{or} & P \mid Q \mid R[\![a/z]\!] \end{array}$$
  - when sender $\overline{x}$ is at location ℓ_0 and receptors x are at locations ℓ_1 and ℓ_2 ??
  - but easy to implement when $\ell_0 = \ell_1 = \ell_2$.
  - or when $\ell_1 = \ell_2$ and $P = 0$.



Process algebras and Localisation (2/2)

- π-calculus $\neq$ physical mobility
  - channels are not located
  - In a distributed environment, consider
    $$\overline{x}(a).P \mid x(y).Q \mid x(z).R \;\longrightarrow\; \begin{array}{l} P \mid Q[\![a/y]\!] \mid R \\ \text{or}\quad P \mid Q \mid R[\![a/z]\!] \end{array}$$
  - when sender $\overline{x}$ is at location ℓ_0 and receptors x are at locations ℓ_1 and ℓ_2 ??
  - but easy to implement when $\ell_0 = \ell_1 = \ell_2$.
  - or when $\ell_1 = \ell_2$ and $P = 0$.



From π-calculus to distributed Join-calculus (1/3)

- asynchronous outputs (asynchronous π-calculus)
  - $\overline{x}(a).P$ implies $P = 0$
    simply written $\overline{x}(a)$.
- for each channel name, receptors are singly located
- receptors are always receptive.
  Otherwise in π-calculus :
  - $\overline{x}(a) \mid \overline{x}(b) \mid x(y).P \longrightarrow \overline{x}(b) \mid P[\![a/y]\!] \longrightarrow ?$
    when $P[\![a/y]\!]$ does not contain x.
- need of join-patterns guards for synchronization
  - $\overline{x}(a) \mid \overline{x}'(b) \mid (x(y) + x'(z)).P \longrightarrow P[\![a/y, b/z]\!]$
  - $\simeq$ polynomial π-calculus

From π-calculus to distributed Join-calculus (1/3)

- asynchronous outputs (asynchronous π-calculus)
  - $\overline{x}(a).P$ implies $P = 0$
    simply written $\overline{x}(a)$.
- for each channel name, receptors are singly located
- receptors are always receptive.
  Otherwise in π-calculus :
  - $\overline{x}(a) \mid \overline{x}(b) \mid x(y).P \longrightarrow \overline{x}(b) \mid P[\![a/y]\!] \longrightarrow ?$
    when $P[\![a/y]\!]$ does not contain x.
- need of join-patterns guards for synchronization
  - $\overline{x}(a) \mid \overline{x}'(b) \mid (x(y) + x'(z)).P \longrightarrow P[\![a/y, b/z]\!]$
  - $\simeq$ polynomial π-calculus

From π-calculus to distributed Join-calculus (1/3)

- asynchronous outputs (asynchronous π-calculus)
  - $\overline{x}(a).P$ implies $P = 0$
    simply written $\overline{x}(a)$.
- for each channel name, receptors are singly located
- receptors are always receptive.
  Otherwise in π-calculus :
  - $\overline{x}(a) \mid \overline{x}(b) \mid x(y).P \longrightarrow \overline{x}(b) \mid P[\![a/y]\!] \longrightarrow ?$
    when $P[\![a/y]\!]$ does not contain x.
- need of join-patterns guards for synchronization
  - $\overline{x}(a) \mid \overline{x}'(b) \mid (x(y) + x'(z)).P \longrightarrow P[\![a/y, b/z]\!]$
  - $\simeq$ polynomial π-calculus

From π-calculus to distributed Join-calculus (1/3)

- asynchronous outputs (asynchronous π-calculus)
  - $\overline{x}(a).P$ implies $P = 0$
    simply written $\overline{x}(a)$.
- for each channel name, receptors are singly located
- receptors are always receptive.
  Otherwise in π-calculus :
  - $\overline{x}(a) \mid \overline{x}(b) \mid x(y).P \longrightarrow \overline{x}(b) \mid P[\![a/y]\!] \longrightarrow ?$
    when $P[\![a/y]\!]$ does not contain x.
- need of join-patterns guards for synchronization
  - $\overline{x}(a) \mid \overline{x}'(b) \mid (x(y) + x'(z)).P \longrightarrow P[\![a/y, b/z]\!]$
  - $\simeq$ polynomial π-calculus

From π-calculus to distributed Join-calculus (2/3)

- in π-calculus, distinct receptors may be identified
  - x and y are distinct in P, when
    $P = z(y).(x(a).Q \mid y(b).R) \mid \overline{z}(x)$
  - $P \longrightarrow x(a).Q' \mid x(b).R'$
    and now $y = x$
- in Join, names passed on channels cannot become receptors
  - for instance $P = z(y).(x(a).Q \mid \overline{y}(b).R) \mid \overline{z}(x)$
  - $P \longrightarrow x(a).Q' \mid \overline{x}(b).R'$
  - distinct receptors remain distinct.
  - locations of receptors are easier to handle.

From π-calculus to distributed Join-calculus (2/3)

- in π-calculus, distinct receptors may be identified
  - x and y are distinct in P, when
    $P = z(y).(x(a).Q \mid y(b).R) \mid \overline{z}(x)$
  - $P \longrightarrow x(a).Q' \mid x(b).R'$
    and now $y = x$
- in Join, names passed on channels cannot become receptors
  - for instance $P = z(y).(x(a).Q \mid \overline{y}(b).R) \mid \overline{z}(x)$
  - $P \longrightarrow x(a).Q' \mid \overline{x}(b).R'$
  - distinct receptors remain distinct.
  - locations of receptors are easier to handle.

From π-calculus to distributed Join-calculus (3/3)

- π-calculus is a specification language
  - nice theory `[Sangiorgi, et al]`
  - centralized $+$ serialized implementation is trivial,
  - centralized $+$ concurrent implementation is more difficult
  - distributed implementation is mission impossible.
    (distributed consensus for nearly any communication)
- Join is motivated by distributed (asynchronous) implementation
  - more complex theory `[Fournet, Gonthier, et al]`
  - distributed implementation of Join is easy
  - JCL 1-05, Jocaml, Polyphonic C#.

From π-calculus to distributed Join-calculus (3/3)

- π-calculus is a specification language
  - nice theory `[Sangiorgi, et al]`
  - centralized $+$ serialized implementation is trivial,
  - centralized $+$ concurrent implementation is more difficult
  - distributed implementation is mission impossible.
    (distributed consensus for nearly any communication)
- Join is motivated by distributed (asynchronous) implementation
  - more complex theory `[Fournet, Gonthier, et al]`
  - distributed implementation of Join is easy
  - JCL 1-05, Jocaml, Polyphonic C#.

Example of Join programs (1/2)

- ```
  def counter(m, k) =
    def count(n) | inc() = count(n+1)
    and count(n) | get(k) = count(n) | k(n) in
    count(m) | k(get, inc) in

  def test(g,i) =
    i() | i() | g(print) in

  counter(3, test) | counter(10, test)
  ```

- prints 3-5 and 10-12.
  ```

Example of Join programs (2/2)

- syntactic sugar for continuations
  $\Rightarrow$ more direct functional style

```
def counter(m) =
  def count(n) | inc() = count(n+1) | reply to inc
  and count(n) | get(k) = count(n) | reply n to get in
  count(m) | reply (get, inc) in

def test(g,i) =
  i() | i() | print(g()) in

test(counter(3)) | test(counter(10))
```

Existing distributed implementation of π-calculus

- remote names and local names differ
  - remote names contain the location address
    $x@\ell_0 \neq x@\ell_1$
- communication is only local
- explicit routing
  - receptive distributed π-calculus,
    [Amadio, Boudol, Lousshaine]
  - remote channels are receptive
  - synchronisation with local channels
  - $P = [\ell' :: \text{go}\,\ell.\overline{x}@\ell(a).Q|Q'] \mid [\ell :: x(y).R]$
    $P \longrightarrow [\ell' :: Q'] \mid [\ell :: \overline{x}(a).Q \mid x(y).R]$
- in Nomadic Pict, remote communication is achieved by multiplexing of channel names on top of Unix sockets.

Existing distributed implementation of π-calculus

- remote names and local names differ
  - remote names contain the location address
    $x@\ell_0 \neq x@\ell_1$
- communication is only local
- explicit routing
  - receptive distributed π-calculus,
    [Amadio, Boudol, Lousshaine]
  - remote channels are receptive
  - synchronisation with local channels
  - $P = [\ell' :: \mathrm{go}\, \ell.\overline{x}@\ell(a).Q | Q'] \mid [\ell :: x(y).R]$
    $P \longrightarrow [\ell' :: Q'] \mid [\ell :: \overline{x}(a).Q \mid x(y).R]$
- in Nomadic Pict, remote communication is achieved by multiplexing of channel names on top of Unix sockets.

Existing distributed implementation of π-calculus

- remote names and local names differ
  - remote names contain the location address
    $x@\ell_0 \neq x@\ell_1$
- communication is only local
- explicit routing
  - receptive distributed π-calculus,
    [Amadio, Boudol, Lousshaine]
  - remote channels are receptive
  - synchronisation with local channels
  - $P = [\ell' :: \mathrm{go}\,\ell.\overline{x}@\ell(a).Q|Q'] \mid [\ell :: x(y).R]$
    $P \longrightarrow [\ell' :: Q'] \mid [\ell :: \overline{x}(a).Q \mid x(y).R]$
- in Nomadic Pict, remote communication is achieved by multiplexing of channel names on top of Unix sockets.

Existing distributed implementation of π-calculus

- remote names and local names differ
  - remote names contain the location address
    $x@\ell_0 \neq x@\ell_1$
- communication is only local
- explicit routing
  - receptive distributed π-calculus,
    [Amadio, Boudol, Lousshaine]
  - remote channels are receptive
  - synchronisation with local channels
  - $P = [\ell' :: \mathrm{go}\,\ell.\overline{x}@\ell(a).Q|Q'] \mid [\ell :: x(y).R]$
    $P \longrightarrow [\ell' :: Q'] \mid [\ell :: \overline{x}(a).Q \mid x(y).R]$
- in Nomadic Pict, remote communication is achieved by multiplexing of channel names on top of Unix sockets.

Mobility (1/2)

- (logical) locations may move
  $[\ell' :: \mathsf{go}\,\ell. P] \longrightarrow [\ell :: P]$
- preserving lexical scope for remote channels
  (π-calculus, Obliq, Join, Ambients, etc)
- sometimes dynamic scope
  - local ressources `[Alan Schmitt]`
  - theory of dynamic linking in prog. languages.
- syntax or type system guarantees receptivity of remote channels, or local ressources.

Mobility (1/2)

- (logical) locations may move
  $[\ell' :: \mathsf{go}\,\ell.\,P] \longrightarrow [\ell :: P]$
- preserving lexical scope for remote channels
  (π-calculus, Obliq, Join, Ambients, etc)
- sometimes dynamic scope
  - local ressources `[Alan Schmitt]`
  - theory of dynamic linking in prog. languages.
- syntax or type system guarantees receptivity of remote channels, or local ressources.

Mobility (1/2)

- (logical) locations may move
  $[\ell' :: \mathsf{go}\,\ell.P] \longrightarrow [\ell :: P]$
- preserving lexical scope for remote channels
  (π-calculus, Obliq, Join, Ambients, etc)
- sometimes dynamic scope
  - local ressources `[Alan Schmitt]`
  - theory of dynamic linking in prog. languages.
- syntax or type system guarantees receptivity of remote channels, or local ressources.

Mobility (1/2)

- (logical) locations may move
  $[\ell' :: \mathsf{go}\,\ell.P] \longrightarrow [\ell :: P]$
- preserving lexical scope for remote channels
  (π-calculus, Obliq, Join, Ambients, etc)
- sometimes dynamic scope
  - local ressources `[Alan Schmitt]`
  - theory of dynamic linking in prog. languages.
- syntax or type system guarantees receptivity of remote channels, or local ressources.

Mobility (2/2)

- flat (Dπ) or nested locations (Ambients, Join)
- explicit routing in Ambients
- implicit routing in Join
- routing in Join needs forwarders
- routing is sensitive to node failures
- programming the routing of messages is complex

Mobility in Join (1/2)

- Every process is in a location
- Every channel-name definition belongs to a unique location.
- Locations can be nested.
- Locations have (unique) names.
- Syntactic restrictions, no types
- Sub-locations can be created.
- Locations can move towards another location, carrying their contents (processes, definitions, sub-locations)

Mobility in Join (2/2)

```
def counter(m, k, There) =
  Here[ def count(n) | inc(k) = count(n+1)
        or count(n) | get(k) = count(n) | k(n)
      in go(There); count(m) | k(get,inc) ]

Client[def test = ... in counter(3,test,Client) ]
```

⇒ ppt

Coding Ambients into Join (1/5)

- The dynamic structure of ambients is coded as a doubly linked tree.
  - each node in the tree implements an ambient :
  - each node contains non-ambient processes running in parallel ;
  - each node hosts an *ambient manager* that controls the steps performed in this ambient and in its direct subambients.
  - different nodes may be running at different physical sites.
- Since several ambients may have the same name, each node is associated with a unique identifier.
- Each ambient points to its subambients and to its parent ambient.
  - The down links are used for controlling subambients,
  - the up link is used for proposing new actions.

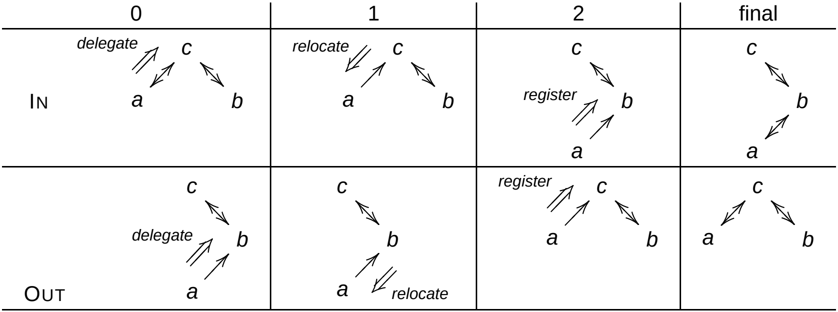
Coding Ambients into Join (1/5)

- The dynamic structure of ambients is coded as a doubly linked tree.
  - each node in the tree implements an ambient :
  - each node contains non-ambient processes running in parallel ;
  - each node hosts an *ambient manager* that controls the steps performed in this ambient and in its direct subambients.
  - different nodes may be running at different physical sites.
- Since several ambients may have the same name, each node is associated with a unique identifier.
- Each ambient points to its subambients and to its parent ambient.
  - The down links are used for controlling subambients,
  - the up link is used for proposing new actions.

Coding Ambients into Join (2/5)

- the decision to perform a step will always be taken by the parent of the affected ambient.

(Single arrows represent current links ; double arrows represent messages in transit).



Coding Ambients into Join (3/5)

In-step

$$c[\,a[\,\mathtt{in}\ b.Q\,]\ \mid\ b[\,0\,]\,] \rightarrow c[\,b[\,a[\,Q\,]\,]\,]$$

0-step : initially, *a* delegates the migration request In *b* to its current parent (here *c*) ; to this end, it uses its current up link to send a message to *c* saying that *a* is willing to move into an ambient named *b*.

1-step : the enclosing ambient *c* matches *a*'s request with *a*'s and *b*'s down links. Atomically, *a*'s request and the down link to *a* are erased, and a relocation message is sent to *a* ; this message contains the address of *b*, so that *a* will be able to relocate to *b*, and also a descriptor of *a*'s successful action, so that *a* can complete this step by triggering its guarded process.

2-step : the moving ambient *a* receives *c*'s relocation message, relocates to *b*'s site, and updates its up link to point to *b*. It also sends a message to *b* that eventually registers *a* as a subambient of *b*.

Coding Ambients into Join (4/5)

In-step

- The 1-step may preempt other actions delegated by *a* to its former parent *c*. Such actions should now be delegated to its new parent *b*.
- For that purpose, *a*'s ambient manager keeps a log of the pending actions delegated in 0-steps, and, as it completes one of these action in a 2-step, it re-delegates all other actions towards its new parent. (The log cannot be maintained by the parent, because delegation messages may arrive long after *a*'s departure)

- Moreover, in the case an ambient moves back into a former parent, former delegation messages may still arrive, and should not be confused with fresh ones. Such stale messages must be deleted.
- This is not directly possible in an asynchronous world, but equivalently each migration results in a modification of the unique identifier of the moving ambient, each delegation message is tagged with this identifier, and the parent discards every message with an old identifier.

Coding Ambients into Join (5/5)

OUT-step

- An OUT-step of *a* out of *b* corresponds to the same series of three steps. The main different is in step 1, as the enclosing ambient *b* matches *a*'s request with *a*'s down link and its own name *b*, and passes its own up link to *c* in the relocation message sent back to *a*.

Typical other problems

- integration into programming language
- distributed garbage collector
- handling of failures
- security

Each item is a huge problem

Conclusion

- distance between language design and distributed implementations
- transparency of network and routing primitives
  - easy prototyping
  - sufficient for many applications
  - network awareness
  - need for more powerful network primitives (distributed transactions, atomic broadcast, etc)
- security
  - in the programming language
  - external primitives