The π-calculus

The π-calculus was first presented in 1989 by Milner, Parrow and Walker, based on an extension of CCS by Engberg and Nielsen.

It is useful for building models of concurrent/distributed/mobile systems and study their properties, like Turing Machines and the λ-calculus are useful for studying sequential computation.

The impact of π-calculus on industry and academia:

- **Message-passing programming** (SJ, Session C, MPI, Go, Scala, Erlang, ...).
- **Distributed programming** (JBoss Red Hat Scribble and Savara projects and the Ocean Observatories Initiative)
- **Web services and IoT** (e.g. W3C Web Service Choreography Description Languages) Financial protocols; and Block-chains (RChain)
- Systems Biology and Security: the asynchronous π-calculus is used to specify and verify security protocols
- Active field of research, especially in Europe, in the UK, and at Imperial

References

- What you need for the exam is on these slides (and the tutorial sheets)
- **Introductory book:**

  Communicating and Mobile Systems: the π-calculus (Milner 1999)
- Advanced books:

  The π-calculus: a Theory of Mobile Processes (Sangiorgi, Walker 2001)
  Distributed Pi-Calculus (Hennessy 2007)

About the π-calculus

- There is no "canonical" π-calculus. For each application domain, there are alternative notations and lots of specialised variants:
  - we fix one notation and stick to it (once you know one notation, it's easy to understand the others)
  - we start with the simplest variant (the asynchronous π-calculus) and explore various extensions
- An interaction happens by *message passing* rather than *synchronisation*.
- In the asynchronous π-calculus, the communication is *asynchronous* rather than *synchronous*.
- The π-calculus evolved from (value-passing) CCS, but it is more expressive:
  - channel mobility: send and receive *channel names* as messages
  - restriction is interpreted as new (private, secret) channel generation

Some Picture

Overview of the π-calculus lectures

- A simple version of asynchronous π-calculus: syntax and semantics
- The asynchronous π-calculus: syntax and semantics
- Joyful Hacking in asynchronous π-calculus
- Protocols and Session-calculus

The asynchronous π-calculus

The asynchronous π-calculus is a subset of the π-calculus presented independently by Honda and Tokoro (1991), and by Boudol (1992).

Communication is asynchronous: the output process $\overline{a}\langle b\rangle$ represents a message which is in the communication layer waiting to be picked up by a receiver (it does not have a continuation P like the synchronous process $\overline{a}\langle b\rangle.P$). Several messages can be in the communication layer at the same time, and their order is not preserved. Asynchronous communication is common in distributed systems, and can be used to simulate synchronous communication (handshake) when needed.

The asynchronous π-calculus is easier and more efficient to implement than the full π-calculus. It is widely used as the basis for building more complicated calculi, by adding primitives for distributed or object-oriented programming.

As a start, we consider a subset of asynchronous π-calculus, called *asynchronous* CCS.

Syntax of CCS

	$a, b, c, ..$	name
$P, Q ::=$		processes
	$\mathbf{0}$	nil process
	$P \mid Q$	parallel composition of P and Q
	$(\nu\, a)P$	restriction of a (scope) in P
	$\overline{a}$	output on channel a
	$a.P$	input on channel a, with continuation P

Notation: $\begin{cases} \widetilde{a} \overset{\text{df}}{=} a_1, \ldots, a_n & \text{when we don't care about each } a_i \\ (\nu\, a_1, \ldots, a_n)P \overset{\text{df}}{=} (\nu\, a_1) \ldots (\nu\, a_n)P & \end{cases}$

Abbreviation: $\begin{cases} \overline{a} \overset{\text{df}}{=} \overline{a}.\mathbf{0} & \text{when we don't have a continuation} \\ a \overset{\text{df}}{=} a.\mathbf{0} & \text{when we don't have a continuation} \end{cases}$

Examples of CCS

- $\mathbf{0}$ means nothing and $\mathbf{0} \,|\, \mathbf{0}$ is as same as one $\mathbf{0}$, hence nothing
- $\overline{a}$ is one message to a; and $\overline{a} \,|\, \overline{a}$ means two messages to a.
- $\overline{a} \,|\, \overline{b} \,|\, \overline{c}$

  One message to a, one message to b and One message to c
- $\overline{a} \,|\, \overline{b}$ is as the same as $\overline{b} \,|\, \overline{a}$.
- $a.b.c$ inputs from a, then inputs from b and then inputs from c
- $a.b$ does not mean $b.a$
- $a.\overline{b}$ inputs from a then outputs to b, and $a.(\overline{b} \,|\, \overline{c} \,|\, \overline{d})$ inputs from a, then outputs to b, c and d.
- $a.b.(\overline{c} \,|\, \overline{a})$
- $a.(b.\overline{c} \,|\, d.\overline{e})$

Bad Syntax of CCS

- $\overline{a}.\overline{b}$ and $\overline{a}.b$
- $a.(b\,|\,c).d$ and $a.(b\,|\,c).\overline{d}$
- $\mathbf{0}.\mathbf{0}$

Reduction of CCS: Informally

We write $P \longrightarrow Q$ if P reduces to Q, like $8+2 \longrightarrow 10$.

- $\overline{a} \,|\, a.\mathbf{0} \longrightarrow \mathbf{0}$
- $\overline{a} \,|\, a.\overline{b} \longrightarrow \overline{b}$
- $\overline{a} \,|\, a.\overline{b} \,|\, \overline{c} \,|\, c.\overline{d}$ (hint: $(10+2)-(2+4)$)
- $\overline{a} \,|\, a.\overline{b} \,|\, a.\overline{c}$ reduces to either $\overline{b} \,|\, a.\overline{c}$ or $a.\overline{b} \,|\, \overline{c}$, that is

$$\overline{a} \,|\, a.\overline{b} \,|\, a.\overline{c} \longrightarrow \overline{b} \,|\, a.\overline{c} \quad \text{or} \quad \overline{a} \,|\, a.\overline{b} \,|\, a.\overline{c} \longrightarrow a.\overline{b} \,|\, \overline{c}$$

- $\overline{a} \,|\, \overline{a} \,|\, a.\overline{b} \,|\, a.\overline{c}$
- $\overline{a} \,|\, \overline{b} \,|\, a.b.\overline{e} \,|\, b.a.\overline{d}$

Name Restriction of CCS: Informally

- $(\nu\, a)(\overline{a} \,|\, a.\overline{b}) \,|\, a.\overline{c}$ means $(\nu\, d)(\overline{d} \,|\, d.\overline{b}) \,|\, a.\overline{c}$

  Similar with $f(x) = x + 2$ means $f(y) = y + 2$

- $(\nu\, a)(\overline{a} \,|\, a.\overline{b}) \,|\, a.\overline{c} \longrightarrow (\nu\, a)\overline{b} \,|\, a.\overline{c} \equiv \overline{b} \,|\, a.\overline{c}$

- but $(\nu\, a)(\overline{a} \,|\, a.\overline{b}) \,|\, a.\overline{c} \not\longrightarrow a.\overline{b} \,|\, \overline{c}$

Names and Variables

We separate channel names, which are like constants in a programming language, from variables, which are used to instantiate messages received in input. Consequently, we have two sorts:

$a, b, c \in \mathcal{N}$	Channel Names
$x, y, z \in \mathcal{V}$	Variables

This distinction is not mandatory to build the theory, but is convenient when the calculus is used to model actual systems.

Syntax of asynchronous π-calculus

$u, v ::=$		identifiers
	$a, b, c, ..$	name
	$x, y, z, ..$	variable
$P, Q ::=$		processes
	$\mathbf{0}$	nil process
	$P \mid Q$	parallel composition of P and Q
	$(\nu\, a)P$	generation of a with scope P (also called *restriction*)
	$!P$	replication of P, i.e. infinite parallel composition $P \mid P \mid P \mid \ldots$
	$\overline{u}\langle v\rangle$	output of v on channel u
	$u(x).P$	input of *distinct* variables x on u, with continuation P

Notation: $\begin{cases} \widetilde{u} \overset{\text{df}}{=} u_1, \ldots, u_n & \text{when we don't care about each } u_i \\ (\nu\, a_1, \ldots, a_n)P \overset{\text{df}}{=} (\nu\, a_1) \ldots (\nu\, a_n)P & \end{cases}$

Later on, we will consider other operators such as choice, output continuation, recursive definitions.

Free variables and free names

In order to understand the formal semantics of the π-calculus, it is important to know exactly what are the free variables fv and the free names fn of each term.

$$
\begin{aligned}
\mathit{fv}(x) &= \{x\} & \mathit{fn}(x) &= \emptyset \\
\mathit{fv}(a) &= \emptyset & \mathit{fn}(a) &= \{a\} \\
\mathit{fv}(\mathbf{0}) &= \emptyset & \mathit{fn}(\mathbf{0}) &= \emptyset \\
\mathit{fv}(P \mid Q) &= \mathit{fv}(P) \cup \mathit{fv}(Q) & \mathit{fn}(P \mid Q) &= \mathit{fn}(P) \cup \mathit{fn}(Q) \\
\mathit{fv}((\nu\, a)P) &= \mathit{fv}(P) & \mathit{fn}((\nu\, a)P) &= \mathit{fn}(P) \setminus \{a\} \\
\mathit{fv}(!P) &= \mathit{fv}(P) & \mathit{fn}(!P) &= \mathit{fn}(P) \\
\mathit{fv}(\overline{u}\langle v\rangle) &= \mathit{fv}(u) \cup \mathit{fv}(v) & \mathit{fn}(\overline{u}\langle v\rangle) &= \mathit{fn}(u) \cup \mathit{fn}(v) \\
\mathit{fv}(u(x).P) &= \mathit{fv}(u) \cup (\mathit{fv}(P) \setminus \{x\}) & \mathit{fn}(u(x).P) &= \mathit{fn}(u) \cup \mathit{fn}(P)
\end{aligned}
$$

Both $u(-)$ and $(\nu\, -)$ are called *binders*. A term is *closed* if it has no free variables, it is *open* otherwise.

In the process $P = (\nu\, b)a(x).(\overline{x}\langle z\rangle \mid \overline{x}\langle b\rangle)$, we have highlighted all the *free occurrences* of names and variables. In particular, $\mathit{fv}(P) = \{z\}$ and $\mathit{fn}(P) = \{a\}$. Above, the first occurrences of b and x are called *binding occurrences*, whereas the second occurrence of b and the second and third occurrences of x are called *bound*.

α-conversion

α-conversion is the meta-operation of renaming consistently (i.e. avoiding clashes) the bound names or variables of a process. If P is obtained from Q by α-conversion, we say that P and Q are α-equivalent, and we write $P =_\alpha Q$. For example:

$$(\nu\, a)(\overline{a}\langle b\rangle \,|\, (\nu\, c)\overline{c}\langle a\rangle) =_\alpha (\nu\, d)(\overline{d}\langle b\rangle \,|\, (\nu\, c)\overline{c}\langle d\rangle)$$

But we cannot replace a with b:

$$(\nu\, a)(\overline{a}\langle b\rangle \,|\, (\nu\, c)\overline{c}\langle a\rangle) \neq_\alpha (\nu\, b)(\overline{b}\langle b\rangle \,|\, (\nu\, c)\overline{c}\langle b\rangle)$$

Can we replace a with c?

$$(\nu\, a)(\overline{a}\langle b\rangle \,|\, (\nu\, c)\overline{c}\langle a\rangle) \neq_\alpha (\nu\, c)(\overline{c}\langle b\rangle \,|\, (\nu\, c)\overline{c}\langle c\rangle)$$

Not naively! We can if, for example, we first α-convert the name c to e.

$$(\nu\, a)(\overline{a}\langle b\rangle \,|\, (\nu\, c)\overline{c}\langle a\rangle) =_\alpha (\nu\, c)(\overline{c}\langle b\rangle \,|\, (\nu\, e)\overline{e}\langle c\rangle)$$

The intuition is that α-conversion preserves each difference between names. We will use α-conversion very often, without explicit mention.

Substitution

A *substitution* $\{^{a}/_{x}\}$ applied to a process P (denoted by $P\{^{a}/_{x}\}$), has the effect of replacing all the free occurrences of x in P with a.

The substitution avoids clashes with any bound names by implicitly using α-conversion (*capture-avoiding substitution*). For example:

$$((\nu\, d)(\overline{a}\langle b\rangle \mid \overline{a}\langle d\rangle \mid \overline{a}\langle x\rangle))\{^{d}/_{x}\} \;=\; (\nu\, c)(\overline{a}\langle b\rangle \mid \overline{a}\langle c\rangle \mid \overline{a}\langle d\rangle))$$

where we have avoided the capture of the external d by α-converting $(\nu\, d)$ to $(\nu\, c)$.

Given a substitution σ and an open process P, if $P\sigma$ is closed (i.e. $\mathit{fv}(P\sigma) = \emptyset$) we say that σ is a *closing substitution*. For example, $\{^{b}/_{x}\}$ is closing for $a(y).\overline{x}\langle y\rangle$:

$$a(y).\overline{x}\langle y\rangle\{^{b}/_{x}\} = a(y).\overline{b}\langle y\rangle \qquad \mathit{fv}(a(y).\overline{b}\langle y\rangle) = \emptyset$$

As a shorthand, we use the notation $\{^{v_1,\dots,v_n}/_{x_1,\dots,x_n}\} \overset{\text{df}}{=} \{^{v_1}/_{x_1}\}\dots\{^{v_n}/_{x_n}\}$.

Quiz: α-conversion and substitution

1. Correct?

   $(\nu\, a)(\overline{a}\langle b\rangle \mid \overline{c}\langle a\rangle) =_\alpha (\nu\, d)(\overline{d}\langle b\rangle \mid \overline{c}\langle d\rangle)$

2. Correct?

   $(\nu\, a)(\overline{a}\langle b\rangle \mid (\nu\, a)\overline{c}\langle a\rangle) =_\alpha (\nu\, d)(\overline{d}\langle b\rangle \mid (\nu\, a)\overline{c}\langle a\rangle)$

3. Correct?

   $(\nu\, a)(\overline{a}\langle b\rangle \mid (\nu\, a)\overline{c}\langle a\rangle) =_\alpha (\nu\, d)(\overline{d}\langle b\rangle \mid (\nu\, a)\overline{c}\langle d\rangle)$

4. Correct?

   $a(y).\overline{x}\langle y\rangle\{^{b}/_{y}\} = a(b).\overline{x}\langle b\rangle$

5. Correct?

   $a(y).(\overline{x}\langle y\rangle \mid (\nu\, a)\overline{c}\langle a\rangle)\{^{a}/_{x}\} = a(y).(\overline{a}\langle y\rangle \mid (\nu\, a)\overline{c}\langle a\rangle)$

6. Correct?

   $a(y).(\overline{x}\langle y\rangle \mid (\nu\, a)\overline{c}\langle x\rangle)\{^{a}/_{x}\} = a(y).(\overline{a}\langle y\rangle \mid (\nu\, a)\overline{c}\langle a\rangle)$

Quiz: Answers: α-conversion and substitution

Structural congruence (1)

The reduction semantics of the π-calculus is inspired by the Chemical Abstract Machine (CHAM) of Berry and Boudol. Processes ”float around” like molecules in a solution using structural congruence ($\equiv$) and ”react” using a reduction relation ($\longrightarrow$).

The intuition is that if $P \equiv Q$ then we consider P and Q completely interchangeable.

Structural congruence is an equivalence relation, is preserved by all the syntactic operators, and contains α-equivalence.

$P \equiv P$	(Eq Reflexivity)
$P \equiv Q \Longrightarrow Q \equiv P$	(Eq Symmetry)
$P \equiv R \ \mathit{and} \ R \equiv Q \Longrightarrow P \equiv Q$	(Eq Transitivity)
$P \equiv Q \Longrightarrow (\nu\, a)P \equiv (\nu\, a)Q$	(Cong Res)
$P \equiv Q \Longrightarrow P \,\vert\, R \equiv Q \,\vert\, R$	(Cong Par)
$P \equiv Q \Longrightarrow u(x).P \equiv u(x).Q$	(Cong In)
$P \equiv Q \Longrightarrow !P \equiv !Q$	(Cong Rep)
$P =_\alpha Q \Longrightarrow P \equiv Q$	(α-equivalence)

Structural congruence satisfies also additional rules specific to the π-calculus. . .

Structural congruence (2)

The set of all processes, with the operation of parallel composition, and with the nil process as the neutral element are a *commutative monoid*.

$P \mid (Q \mid Q') \equiv (P \mid Q) \mid Q'$	(Associativity)
$P \mid Q \equiv Q \mid P$	(Commutativity)
$P \mid \mathbf{0} \equiv P$	(Zero)

That is, we can exchange freely the position of parallel processes and we can insert or delete the $\mathbf{0}$ process at will. For example:

$$\mathbf{0} \mid \mathbf{0} \mid P \mid Q \mid R \equiv Q \mid \mathbf{0} \mid R \mid P$$

Replication can be folded or unfolded as many times as needed.

$!P \equiv P \mid !P$	(Rep)

Thanks to this rule, the π-calculus can express infinite computations.

Structural congruence (3)

The last rules that complete the definition of $\equiv$ state that the scope of a restricted name can be extended (or contracted) across terms which do not contain free occurrences of that name.

$(\nu\, a)\mathbf{0} \equiv \mathbf{0}$ (Res Nil)

$(\nu\, a)(\nu\, b)P \equiv (\nu\, b)(\nu\, a)P$ (Res Res)

$a \notin \mathit{fn}(P) \Longrightarrow P \,|\, (\nu\, a)Q \equiv (\nu\, a)(P \,|\, Q)$ (Res Par)

For example:

$$(\nu\, a, b)(a(x).\overline{x}\langle c\rangle \,|\, \overline{a}\langle b\rangle) \equiv (\nu\, a)(a(x).\overline{x}\langle c\rangle \,|\, (\nu\, b)(\overline{a}\langle b\rangle))$$

$$(\nu\, a, b)(c(x).\overline{c}\langle x\rangle \,|\, \overline{c}\langle d\rangle) \equiv c(x).\overline{c}\langle x\rangle \,|\, \overline{c}\langle d\rangle$$

It is very important to gain familiarity with structural congruence, we will use it very often during the rest of the course.

Question 1 *Prove* $(\nu\, c)P \equiv P$ *if* $c \notin \mathit{fn}(P)$.

Question 2 *Prove if* $P \equiv Q$ *then* $\mathit{fn}(P) = \mathit{fn}(Q)$ *and* $\mathit{fv}(P) = \mathit{fv}(Q)$.

Reduction relation

Reduction describes how processes interact by exchanging messages. It is the smallest *partial relation* between processes satisfying the rules given below.

$\overline{a}\langle v\rangle \mid a(x).P \;\longrightarrow\; P\{^{v}/_{x}\}$ (Comm)

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$ (Par)

$$\frac{P \longrightarrow P'}{(\nu\, a)P \longrightarrow (\nu\, a)P'}$$ (Res)

$$\frac{P \equiv Q \longrightarrow Q' \equiv P'}{P \longrightarrow P'}$$ (Struct)

As a shorthand notation, we write $P \overset{*}{\longrightarrow} Q$ if $P \equiv Q$ or $P \longrightarrow \ldots \longrightarrow Q$.

Reduction Examples

Communication:

$a(x).\overline{x}\langle c\rangle \mid \overline{a}\langle b\rangle \longrightarrow \overline{b}\langle c\rangle$ by (Struct), (Commutativity) and (Comm)

$$\begin{array}{lll} & a(x).\overline{x}\langle c\rangle \mid \overline{a}\langle b\rangle & \\ \equiv & \overline{a}\langle b\rangle \mid a(x).\overline{x}\langle c\rangle & \text{by (Struct), (Commutativity)} \\ \longrightarrow & \overline{b}\langle c\rangle & \text{by (Comm)} \end{array}$$

Scope extrusion:

$(\nu\, b)(\overline{a}\langle b\rangle) \mid a(x).\overline{c}\langle x\rangle \longrightarrow (\nu\, b)\overline{c}\langle b\rangle$ by (Struct), (Res Par), (Res) and (Comm)

$$\begin{array}{lll} & (\nu\, b)(\overline{a}\langle b\rangle) \mid a(x).\overline{c}\langle x\rangle & \text{note } b \notin \mathit{fn}(a(x).\overline{c}\langle x\rangle) \\ \equiv & (\nu\, b)(\overline{a}\langle b\rangle \mid a(x).\overline{c}\langle x\rangle) & \text{by (Struct), (Res Par)} \\ \longrightarrow & (\nu\, b)\overline{c}\langle b\rangle & \text{by (Res) and (Comm)} \end{array}$$

Infinite behaviour and Sorts

We use the macros for process definition $\mathbf{A}(\widetilde{x})$ and usage $\mathbf{A}\langle\widetilde{v}\rangle$ *only informally*. To represent infinite behaviour, we use the *replication* operator $!P$, stating that there are as many copies of P as needed, all running in parallel. For example:

$$a(x).P \,|\, a(x).Q \,|\, !\overline{a}\langle b\rangle \longrightarrow P\{^b/_x\} \,|\, a(x).Q \,|\, !\overline{a}\langle b\rangle \longrightarrow P\{^b/_x\} \,|\, Q\{^b/_x\} \,|\, !\overline{a}\langle b\rangle$$

Replication is very simple, yet combined with channel generation it can encode recursive definitions (later on we will see how).

Reduction Examples

Infinite behaviour:

$$\overline{a}\langle b\rangle \mid !a(x).\overline{a}\langle x\rangle \longrightarrow \overline{a}\langle b\rangle \mid !a(x).\overline{a}\langle x\rangle$$ by (Struct), (Rep), (Par) and (Comm)

$$\begin{array}{ll} & \overline{a}\langle b\rangle \mid !a(x).\overline{a}\langle x\rangle \\ \equiv & \overline{a}\langle b\rangle \mid a(x).\overline{a}\langle x\rangle \mid !a(x).\overline{a}\langle x\rangle \\ \longrightarrow & \overline{a}\langle b\rangle \mid !a(x).\overline{a}\langle x\rangle \end{array}$$

by (Struct), (Rep)

by (Par) and (Comm)

Nondeterminism:

$$\overline{a}\langle b\rangle \mid \overline{a}\langle d\rangle \mid a(x).\overline{c}\langle x\rangle \begin{array}{l} \nearrow \overline{a}\langle b\rangle \mid \overline{c}\langle d\rangle \\ \searrow \overline{a}\langle d\rangle \mid \overline{c}\langle b\rangle \end{array}$$

by (Par) and (Comm)

using also (Struct), (Commutativity)

Atoms for Name Passing

Small Agents (1)

Forwarder $\mathbf{FW}(a,b) \overset{\mathsf{df}}{=} a(z).\bar{b}\langle z\rangle$

A *forwarder* from channel a to channel b, is a process that forwards a message for a on b.

$$\mathbf{FW}\langle a,b\rangle \,|\, \overline{a}\langle d\rangle \longrightarrow \bar{b}\langle d\rangle$$

$$(\nu\, b)(\mathbf{FW}\langle a,b\rangle \,|\, \mathbf{FW}\langle b,c\rangle) \,|\, \overline{a}\langle d\rangle \longrightarrow\longrightarrow \overline{c}\langle d\rangle$$

Duplicator $\mathrm{D}(a,b,c) \overset{\mathsf{df}}{=} a(z).(\bar{b}\langle z\rangle \mid \overline{c}\langle z\rangle)$

A *duplicator* from channel a to b and c, is a process that duplicates a message for a to b and c.

$$\mathbf{D}\langle a,b,c\rangle \,|\, \overline{a}\langle d\rangle \longrightarrow (\bar{b}\langle d\rangle \,|\, \overline{c}\langle d\rangle)$$

$$(\nu\, b)(\mathbf{D}\langle a,b,c_1\rangle \,|\, \mathbf{D}\langle b,c_2,c_3\rangle) \,|\, \overline{a}\langle d\rangle \longrightarrow\longrightarrow (\overline{c_1}\langle d\rangle \,|\, \overline{c_2}\langle d\rangle \,|\, \overline{c_3}\langle d\rangle)$$

Killer $\mathrm{K}(a) \overset{\mathsf{df}}{=} a(z).\mathbf{0}$

A *killer* at channel a is a process that kills a message from a.

$a(z).(P \,|\, Q)$ can be decomposed as $(\nu\, c_1,c_2)(\mathbf{D}\langle a,c_1,c_2\rangle \,|\, c_1(z).P \,|\, c_2(z).Q)$.

For example, $a(z).(\bar{b}\langle z\rangle \,|\, \mathbf{0})$ is the same as $(\nu\, c_1,c_2)(\mathbf{D}\langle a,c_1,c_2\rangle \,|\, \mathbf{FW}\langle c_1,b\rangle \,|\, \mathbf{K}\langle c_2\rangle)$

Small Agents (3)

Identity Receptor $\mathrm{I}(a) \overset{\text{df}}{=} !\mathbf{FW}\langle a, a\rangle$

An *identity receptor* at channel a is a process that forwards messages for a on a.

$$\overline{a}\langle d\rangle \,|\, \mathbf{I}\langle a\rangle \longrightarrow \overline{a}\langle d\rangle \,|\, \mathbf{I}\langle a\rangle \longrightarrow \overline{a}\langle d\rangle \,|\, \mathbf{I}\langle a\rangle \longrightarrow \overline{a}\langle d\rangle \,|\, \mathbf{I}\langle a\rangle \ldots$$

Equator $\mathrm{EQ}(a, b) \overset{\text{df}}{=} !\mathbf{FW}\langle a, b\rangle \,|\, !\mathbf{FW}\langle b, a\rangle.$

An *equator* between two channels a and b, is a process that forwards all messages for a on b, and viceversa, making a and b in some sense "equivalent". For example,

$$\overline{a}\langle d\rangle \,|\, \mathbf{EQ}\langle a, b\rangle \longrightarrow \overline{b}\langle d\rangle \,|\, \mathbf{EQ}\langle a, b\rangle \longrightarrow \overline{a}\langle d\rangle \,|\, \mathbf{EQ}\langle a, b\rangle \longrightarrow \ldots$$

Omega $\Omega \overset{\text{df}}{=} (\nu\, a)(!\mathbf{FW}\langle a, a\rangle \,|\, \overline{a}\langle a\rangle)$

An *omega* is a process that continues infinite reductions by himself.

$$\Omega \longrightarrow \Omega \longrightarrow \Omega \longrightarrow \cdots$$

New Name Generator $\mathrm{NN}(a) \overset{\text{df}}{=} !a(x).(\nu\, b)\overline{x}\langle b\rangle$. A *new name generator* is a process that creates a new name infinitely when it is asked.

$$\overline{a}\langle c\rangle \,|\, \overline{a}\langle d\rangle \,|\, \mathbf{NN}\langle a\rangle \longrightarrow (\nu\, b)\overline{c}\langle b\rangle \,|\, \overline{a}\langle d\rangle \,|\, \mathbf{NN}\langle a\rangle \longrightarrow (\nu\, b)\overline{c}\langle b\rangle \,|\, (\nu\, b')\overline{d}\langle b'\rangle \,|\, \mathbf{NN}\langle a\rangle$$

Quiz: Name Generateor

1. Is $\mathrm{NN}(a) \overset{\mathsf{df}}{=} !a(x).(\nu\, b)\overline{x}\langle b\rangle$ structural congruent with $\mathrm{NN}_1(a) = (\nu\, b)!a(x).\overline{x}\langle b\rangle$?

2. Is $\mathrm{NN}(a) \overset{\mathsf{df}}{=} !a(x).(\nu\, b)\overline{x}\langle b\rangle$ structural congruent with $\mathrm{NN}_2(a) =!(\nu\, b)a(x).\overline{x}\langle b\rangle$?

Quiz: Answer: Name Generateor

Quiz: Small Agents

Joyful Hacking in π-calculus

Channel mobility example

Internet connection: a $\mathbf{Client}$ and a $\mathbf{Server}$ talk on dedicated communication ports, set up using the channel a:

$$\mathbf{Client}(a, c) \overset{\text{df}}{=} (\overline{a}\langle c\rangle \mid c(x).\mathbf{Client}_1\langle c, x\rangle)$$

$$\mathbf{Server}(a, s) \overset{\text{df}}{=} a(y).(\overline{y}\langle s\rangle \mid \mathbf{Server}_1\langle y, s\rangle)$$

Difference with CCS: the variable y is used as the name of a channel.

$$\begin{array}{rl} & \mathbf{Client}\langle a, c\rangle \mid \mathbf{Server}\langle a, s\rangle \\ \longrightarrow & c(x).\mathbf{Client}_1\langle c, x\rangle \mid \overline{c}\langle s\rangle \mid \mathbf{Server}_1\langle c, s\rangle \\ \longrightarrow & \mathbf{Client}_1\langle c, s\rangle \mid \mathbf{Server}_1\langle c, s\rangle \end{array}$$

After the two communication steps, client and server know each other's port.

Channel generation example

We can make the Internet connection example more flexible. To avoid external interferences, the client and server exchange newly generated port names:

$$\mathbf{Client}(a) \overset{\text{df}}{=} (\nu\, c)(\overline{a}\langle c\rangle \mid c(x).\mathbf{Client_1}\langle c, x\rangle)$$

$$\mathbf{Server}(a) \overset{\text{df}}{=} a(y).(\nu\, s)(\overline{y}\langle s\rangle \mid \mathbf{Server_1}\langle y, s\rangle)$$

No other process knows about c and s because they are inside the name restriction: you can imagine that they will be created at run-time by the ν operator (pronounced "new").

$$\begin{array}{rl} & \mathbf{Client}\langle a\rangle \mid \mathbf{Server}\langle a\rangle \\ \longrightarrow & (\nu\, c)(c(x).\mathbf{Client_1}\langle c, x\rangle \mid (\nu\, s)(\overline{c}\langle s\rangle \mid \mathbf{Server_1}\langle c, s\rangle)) \\ \longrightarrow & (\nu\, c, s)(\mathbf{Client_1}\langle c, s\rangle \mid \mathbf{Server_1}\langle c, s\rangle) \end{array}$$

Note that $\mathbf{Client_1}$ and $\mathbf{Server_1}$ are now within the scope of the ν operator. This phenomenon, called *scope extrusion*, is a distinguishing feature of the π-calculus.

Secure client-server communication

We can now represent secure client-server communication. The client creates a new (secret) channel c before contacting the server on the public channel a:

$$\mathbf{SClient_i}(a) \overset{\text{df}}{=} (\nu\, c)(\overline{a}\langle c\rangle \mid c(x).\mathbf{PClient_i}\langle c, x\rangle)$$

The server is a replicated process, ready to spawn a *session* for each client request:

$$\mathbf{SServer}(a) \overset{\text{df}}{=} !a(y).(\nu\, s)(\overline{y}\langle s\rangle \mid \mathbf{PServer}\langle y, s\rangle)$$

The server can interact with multiple clients at the same time:

$$\begin{aligned}
&\mathbf{SClient_1}\langle a\rangle \mid \mathbf{SClient_2}\langle a\rangle \mid \mathbf{SServer}\langle a\rangle \;\overset{*}{\longrightarrow}\\
&\quad (\nu\, s_1, c_1)(\mathbf{PClient_1}\langle c_1, s_1\rangle \mid \mathbf{PServer}\langle c_1, s_1\rangle)\\
&\mid (\nu\, s_2, c_2)(\mathbf{PClient_2}\langle c_2, s_2\rangle \mid \mathbf{PServer}\langle c_2, s_2\rangle)\\
&\mid \mathbf{SServer}\langle a\rangle
\end{aligned}$$

Each session is protected from the external environment by the restriction on the client and server communication ports, which can be used to exchange data without external interferences.