XPi: a typed process calculus for XML messaging

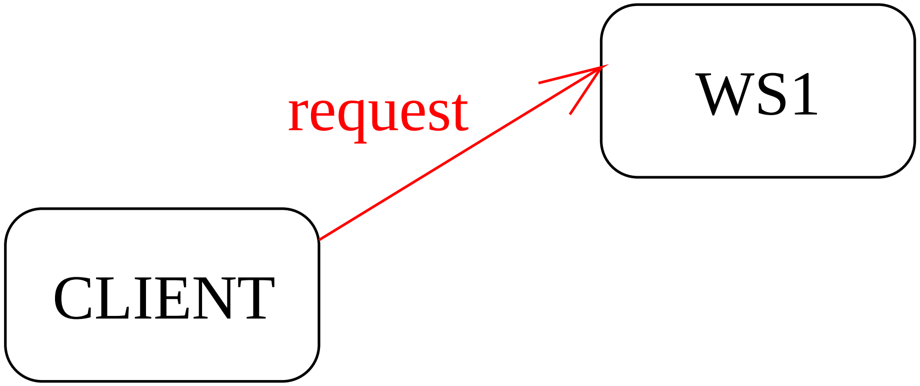
Lucia Acciai - Michele Boreale

D.S.I. - University of Firenze

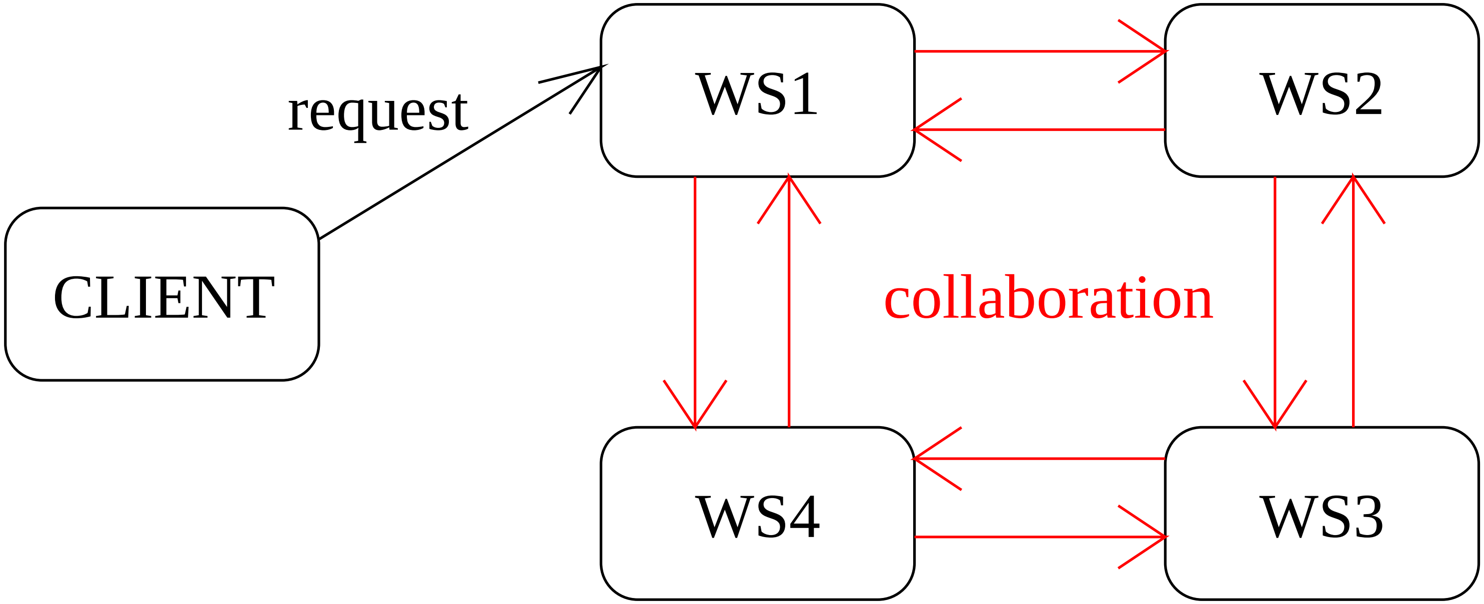
Message passing

- The design of globally distributed application (WS, B2B) is centered around asynchronous message passing in the form of XML documents.
- The choise of message passing is due to:
  - its conceptual simplicity;
  - minimal infrastructural requirements;
  - neutrality with respect to platforms and back-ends of services.

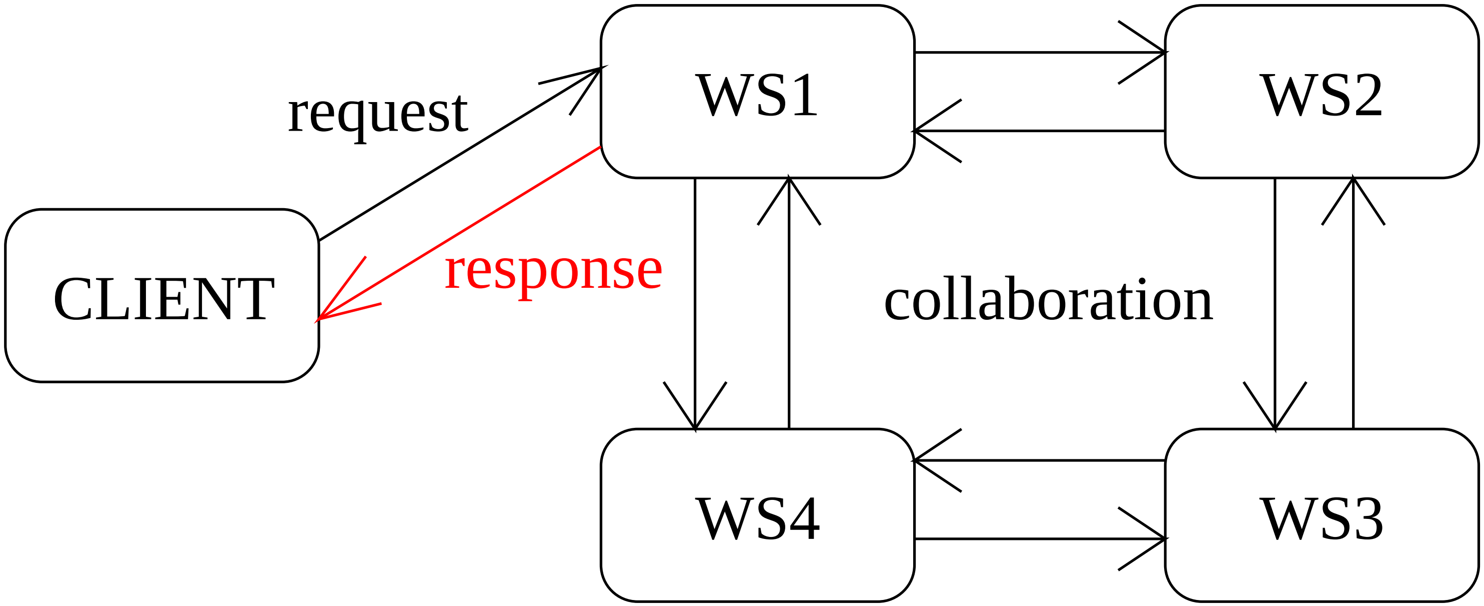
Example: WS



Example: WS



Example: WS



Languages for WS

- There are several languages and standard for WS (WSDL, BPEL4WS, . . .), some of them draw their inspiration from the π-calculus.
- Among these languages we can recognize two extreme:
  - WSDL: which says very little about behaviour;
  - BPEL4WS, BizTalk, . . . : which are hardly amenable to formal analisys.

WS and process calculi

We propose an asynchronous version of the π-calculus where:

- names represent addresses on the net;
- messages passed around are XML documents;
- we generalize ordinary inputs with filtering;
- static (dynamic) typing ensures the run time safety property.

XPi = XML + π-Calculus

XPi is a process calculus based on the asynchronous π-Calculus where:

- messages are represented as nested and tagged lists;
- we generalize the ordinary input actions with queries, which are open messages with no abstractions;
- a type system and a type inference system are provided;
- a notion of barbed equivalence allows to validate interesting equations.

Example: messages

```
<addrbook>
 <person>
  <name>John Smith</name>
  <tel>12345</tel>
  <emailaddrs>
   <email>john@smith</email>
   <email>smith@john</email>
  </emailaddrs>
 </person>
 <person>
  <name>Eric Brown</name>
  <tel>678910</tel>
  <emailaddrs></emailaddrs>
 </person>
</addrbook>
```

Example: messages

```
<addrbook>
 <person>
  <name>John Smith</name>
  <tel>12345</tel>
  <emailaddrs>
   <email>john@smith</email>
   <email>smith@john</email>
  </emailaddrs>
 </person>
 <person>
  <name>Eric Brown</name>
  <tel>678910</tel>
  <emailaddrs></emailaddrs>
 </person>
</addrbook>
```

```
addrbook([
 person([
   name(“John Smith”),
   tel(12345),
   emailaddrs([
      email(“john@smith”),
      email(“smith@john”)
      ])
    ]),
 person([
   name(“Eric Brown”),
   tel(678910),
   emailaddrs([])
    ])
 ])
```

Example: queries

An input process $a.(Q_{\tilde{x}})P$ is a channel a followed by an abstraction $(Q_{\tilde{x}})P$.

The following query extracts the content of the tag `name` from the two `person` elements:

$$\begin{aligned} Q_{\{x,y\}} = (\,&\texttt{addrbook[person[name(x),}\\ &\qquad\qquad\qquad\qquad\quad \texttt{\_],}\\ &\qquad\qquad\quad \texttt{person[name(y),}\\ &\qquad\qquad\qquad\qquad\quad \texttt{\_]}\,)_{\{x,y\}} \end{aligned}$$

Syntax of messages

Message	$M ::=$	v	*Value*
		$\mid x$	*Var*
		$\mid f(M)$	*Tag*
		$\mid LM$	*List*
		$\mid A$	*Abstraction*
Query	$Q ::=$	$\ldots$	
List	$LM ::=$	$[\,]$	*Empty list*
		$\mid x$	*Var*
		$\mid M \cdot LM$	*Concatenation*
Abstraction	$A ::=$	$(Q_{\tilde{x}})P$	*Query and Continuation*
		$\mid x$	*Var*

Example: processes

Consider the query $Q_{\{x,y\}}$ previously defined, and consider the following query:

$$Q'_{\{x\}} = \texttt{(addrbook[person[name(x),\_]])}_{\{x\}}$$

We can define the following process:

$$P = (\ \overline{a}\langle M\rangle \mid (a.(Q'_{\{x\}})P_1 + a.(Q_{\{x,y\}})P_2)\ )\ \text{else}\ P_3$$

Syntax of processes

Process	$P ::=$	$\overline{u}\langle M\rangle$	*Output*
		$\mid \sum_{i\in I} a_i.A_i$	*Guarded Summation*
		$\mid P$ else R	*Else*
		$\mid P_1 \mid P_2$	*Parallel*
		$\mid\, !P$	*Replication*
		$\mid (\nu a)P$	*Restriction*

Example: derived constructs

- **Application:**

$$(Q_{\tilde{x}})P \bullet M \;=\; (\nu c)(\overline{c}\langle M\rangle | c.(Q_{\tilde{x}})P)$$

- **Case:**

$$\text{if } M \text{ of } Q_{\tilde{x}} \text{ then } P_1 \text{ else } P_2 \;=\; ((Q_{\tilde{x}})P_1 \bullet M) \text{ else } P_2$$

- **Decomposition:**

$$\text{if } M \text{ of } Q_{\tilde{x}} +\!+\, Q'_{\tilde{y}} \text{ then } P_1 \text{ else } P_2 \;=\; R([[\,], M]) \qquad \text{where:}$$

$$\begin{aligned} R([l,x]) = \text{if } x \text{ of } (y \cdot w)_{\{y,w\}} \text{ then } \big(&\text{if } l@y \text{ of } Q_{\tilde{x}} \\ &\text{then } \big(\text{if } w \text{ of } Q'_{\tilde{y}} \quad \text{then } P_1 \\ &\qquad\qquad\qquad\qquad \text{else } R([l@y, w])\big) \\ &\text{else } R([l@y, w])\big) \\ \text{else } P_2 \end{aligned}$$

Example: a streaming audio server

- Consider a web service WS that offers two different services:
  - an audio streaming service, offered at channel $stream$;
  - a download service offered at $download$.
- Clients that request the first service must specify a channel for the streaming and its capacity.
- Clients that request download must specify only a channel.

Example: a streaming audio server

The process WS may be the following:

$$\begin{aligned} WS \triangleq !(\quad & stream.(\texttt{req\_stream[bandwidth(``low''),channel(}x\texttt{)]}_{\{x\}}) \\ & \quad \overline{x}\langle \mathsf{v}_{low} \rangle \\ + \ & stream.(\texttt{req\_stream[bandwidth(``high''),channel(}y\texttt{)]}_{\{y\}}) \\ & \quad \overline{y}\langle \mathsf{v}_{high} \rangle \\ + \ & download.(\texttt{req\_down(}z\texttt{)}_{\{z\}})\overline{z}\langle Player \rangle). \end{aligned}$$

The abstraction $Player$:

$$\begin{aligned} Player \triangleq \quad & (\texttt{req\_stream[bandwidth(}x\texttt{),channel(}y\texttt{)]}_{\{x,y\}}) \\ & \quad (\text{Case } x \text{ of } \ \texttt{``low''} \Rightarrow \overline{y}\langle \mathsf{v}_{low} \rangle \\ & \qquad\qquad\quad \texttt{``high''} \Rightarrow \overline{y}\langle \mathsf{v}_{high} \rangle). \end{aligned}$$

Reductions semantic

(COM)

$$\frac{j \in I \quad a_j = a, \quad A_j = (Q_{\tilde{x}})P, \quad \mathsf{match}(M, Q, \sigma)}{\overline{a}\langle M\rangle \mid \sum_{i\in I} a_i.A_i \rightarrow P\sigma}$$

(STRUCT)

$$\frac{P \equiv P', \quad P' \rightarrow R', \quad R' \equiv R}{P \rightarrow R}$$

(CTX)

$$\frac{P \rightarrow P'}{(\nu\tilde{a})(P|R) \rightarrow (\nu\tilde{a})(P'|R)}$$

(ELSE$_1$)

$$\frac{P \rightarrow P'}{P\ \mathsf{else}\ R \ \rightarrow P'}$$

(ELSE$_2$)

$$\frac{P \nrightarrow}{P\ \mathsf{else}\ R \ \rightarrow \ R}$$

Reductions semantic

(COM) $$\frac{j \in I \quad a_j = a, \quad A_j = (Q_{\tilde{x}})P, \quad \mathsf{match}(M, Q, \sigma)}{\overline{a}\langle M \rangle \mid \sum_{i \in I} a_i.A_i \to P\sigma}$$

Type Checking

- We define a typed calculus with annotated queries:

  $A_i = ((Q_i)_{\tilde{x}_i} : \Gamma_{Q_i})P_i$, where $dom(\Gamma_{Q_i}) = \tilde{x}_i$.
- We define a notion of types.
- We define a subtyping relation;
- To every channel we associate a capacity, that is the type of the messages that the channel can transport; $a : ch(\tau)$ indicates that a channel a can transports messages of type τ.

Types

Type	$\tau ::=$	bt	*Basic type* (bt $\in \mathcal{BT}$)
		$\mid$ **T**	*Top*
		$\mid$ **⊥**	*Bottom*
		$\mid f(\tau)$	*Tag* $(f \in \mathcal{F})$
		$\mid$ LT	*List*
		$\mid \tau + \tau$	*Union*
		$\mid$ (τ)Abs	*Abstraction*
List	LT $::=$	$[\,]$	*Empty*
		$\mid * \tau$	*Star*
		$\mid \tau \cdot$ LT	*Concatenation*

Subtyping

The subtyping relation is defined syntactically:

- $f(\mathsf{int}) < *f(\mathsf{int}) + *g(\mathsf{string})$;
- subtyping is contravariant on channel: for every type τ $ch(\bot) > ch(\tau)$; that is $ch(\bot)$ is the type of every channel;
- consider the type: $\tau = f[g[\top], \top]$; the channel that can transport documents of some type $\tau' < \tau$ is $ch(f[g[\bot], \bot])$.

Type Safety

The type system guarantees that well typed processes satisfy a safety property:

Safety: Let P be an annotated closed process. P is *safe* if and only if for each name $a \in ch(\tau)$:

1. whenever $P \equiv (\nu\,\tilde{h})(\overline{a}\langle M\rangle \mid R)$ then $M : \tau$;
2. suppose τ is consistent. Whenever $P \equiv (\nu\,\tilde{h})(\sum_{i\in I} a_i.((Q_i)_{\tilde{x}_i})P_i \mid R)$ and $a_i = a$ then Q_i is τ-consistent.

Type Safety

The type system guarantees that well typed processes satisfy a safety property:

Safety: Let P be an annotated closed process. P is *safe* if and only if for each name $a \in ch(\tau)$:

1. whenever $P \equiv (\nu\,\tilde{h})(\overline{a}\langle M\rangle \mid R)$ then $M : \tau$;
2. suppose τ is consistent. Whenever $P \equiv (\nu\,\tilde{h})(\sum_{i \in I} a_i.((Q_i)_{\tilde{x}_i})P_i \mid R)$ and $a_i = a$ then Q_i is τ-consistent.

That is:

- services receiving only requests they can understand;
- services offered at a given channel will comply with the type declared for that channel.

Conclusions

XPi is a core calculus for XML messaging, featuring:

- asynchronous communications;
- ML-like pattern matching;
- name and code mobility;
- integration of static typing;
- integration of a type inference system;
- the introduction of dynamic abstractions;
- also we have defined a notion of barbed equivalence.

Related works

- **XDuce (Hosoya, Pierce), CDuce (Castagna et al.):** typed (functional) languages for XML document processing;
- **TQL (Cardelli, Ghelli):** logic and query language for XML, based on a spatial logic for the Ambient calculus;
- **π-Duce (Meredith et al.):** language that features asynchronous communication and code/name mobility;
- **Semantic subtyping for the π-calculus (Castagna, De Nicola, Varracca):** language that is the π-calculus enriched with a rich form of semantic subtyping and pattern matching.