Session calculus: Session-based π-calculus

Session calculus: Content

1. How to describe protocols
2. How to program protocols
3. How to check that protocols are correct
4. How to prevent errors
5. What type of errors can be prevented

A protocol in session calculus

An ATM agent offers two services: funds balance or deposit.

- If balance is chosen, then it shows a balance of the account, and recurs to the menu with the same amount.

- If deposit is chosen, then it receives a deposited amount z, and returns to the menu with the new state as their sum $y + z$.

The following is an implementation of the ATM (first try):

$$\begin{array}{lll} \mathbf{ATM}(a, y) \stackrel{\text{df}}{=} & a \rhd [& \text{balance} : \quad \overline{a}\langle y \rangle.\mathbf{ATM}\langle a, y \rangle \quad [] \\ & & \text{deposit} : \quad a(z).\overline{a}\langle y + z \rangle.\mathbf{ATM}\langle a, y + z \rangle \ ] \end{array}$$

The following is an implementation of the customer:

$$\mathbf{Customer}(a, y) \stackrel{\text{df}}{=} a \lhd \text{deposit}.\overline{a}\langle y \rangle.a(x).P$$

The interaction between these three parties is incorrect:

$$\mathbf{ATM}\langle a, 0 \rangle \,|\, \mathbf{Customer}\langle a, 100 \rangle \,|\, \mathbf{Customer}\langle a, 100 \rangle$$

Let's try again:

The following is the implementation of the ATM (second try):

$$\begin{array}{lcl} \mathbf{ATM}(a,y) & \stackrel{\text{df}}{=} & a(z).\ \mathbf{ATM_1}\langle a,y,z\rangle \\ \mathbf{ATM_1}(a,y,s) & \stackrel{\text{df}}{=} & s \rhd [\ \ \text{balance} : \overline{s}\langle y\rangle.\mathbf{ATM_1}\langle a,y,s\rangle \ \ [] \\ & & \qquad\qquad \text{deposit} : s(z).\overline{s}\langle \mathrm{y}{+}z\rangle.\mathbf{ATM_1}\langle a,y+z,s\rangle\ ] \end{array}$$

An example of the customer is:

$$\mathbf{Customer}(a,y) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.s \lhd \text{deposit}.\overline{s}\langle y\rangle.s(x).P$$

You can check the interaction between the three parties is safe:

$$\mathbf{ATM}\langle a,0\rangle \,|\, \mathbf{Customer}\langle a,100\rangle \,|\, \mathbf{Customer}\langle a,100\rangle$$

Here a is called *shared* name and it allows interference of interactions. s is called *session* name and is used for structured interactions.

Syntax

$P ::=$		processes
	$\overline{u}\langle e\rangle.P$	session request
	$u(x).P$	session accept
	$\overline{k}\langle\widetilde{e}\rangle.P$	message send
	$k(\widetilde{x}).P$	message received
	$k \rhd \{l_1 : P_1 \,[\!]\, \cdots \,[\!]\, l_n : P_n\}$	branching
	$k \lhd l.P$	selection
	$\mathbf{0}$	nil process
	$P \mid Q$	parallel composition of P and Q
	$(\nu\, k)P, (\nu\, a)P$	fresh name generation
	$\texttt{def}\ D\ \texttt{in}\ P$	recursion definition
	$X\langle\widetilde{e}\rangle$	recursion call
	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional
$u ::=$	a, b, x	shared name and variable
$k ::=$	s, x	session name and variable
$e ::=$	$v, e\ \texttt{or}\ e, e\ \texttt{and}\ e, \texttt{not}\ e$	expressions
$v ::=$	$\texttt{true}, \texttt{false}, s, a$	values
$D ::=$	$X_1(\widetilde{x_1}) = P_1,\ \ldots,\ X_n(\widetilde{x_n}) = P_n$	declaration for recursion

Free process variables

$$\mathit{fpv}(\overline{u}\langle e\rangle.P) = \mathit{fpv}(u(x).P) = \mathit{fpv}(\overline{k}\langle\widetilde{e}\rangle.P) = \mathit{fpv}(k(\widetilde{x}).P) =$$
$$\mathit{fpv}(k \rhd \{l_1 : P_1 \,[\!]\, \cdots \,[\!]\, l_n : P_n\}) = \mathit{fpv}(k \lhd l.P) =$$
$$\mathit{fpv}((\nu\, k)P) = \mathit{fpv}((\nu\, a)P) = \mathit{fpv}(P)$$

$$\mathit{fpv}(\mathbf{0}) = \emptyset$$

$$\mathit{fpv}(P \mid Q) = \mathit{fpv}(\texttt{if } e \texttt{ then } P \texttt{ else } Q) = \mathit{fpv}(P) \cup \mathit{fpv}(Q)$$

$$\mathit{fpv}(\texttt{def } D \texttt{ in } P) \qquad \text{with } D = (X_1(\widetilde{x_1}) = P_1,\ \ldots,\ X_n(\widetilde{x_n}) = P_n)$$

$$= (\mathit{fpv}(P) \cup \mathit{fpv}(P_1) \cup \ldots \cup \mathit{fpv}(P_n)) \setminus \{X_1, \ldots, X_n\}$$

$$\mathit{fpv}(X\langle\widetilde{e}\rangle) = \{X\}$$

The definitions of free names and free variables are the obvious adaptations.

Session Calculus: Semantics

- New Structural Congruence rule:

  (Cong Def) $\quad (\texttt{def}\ D\ \texttt{in}\ P) \,|\, Q \equiv \texttt{def}\ D\ \texttt{in}\ (P \,|\, Q) \quad (\mathit{fpv}(D) \cap \mathit{fpv}(Q) = \emptyset)$

- New Reduction Relation rules:

  (Def) $\quad \texttt{def}\ D\ \texttt{in}\ (X\langle \widetilde{e} \rangle \,|\, Q) \longrightarrow \texttt{def}\ D\ \texttt{in}\ (P\{\widetilde{v}/\widetilde{x}\} \,|\, Q) \quad (e_i \downarrow v_i, X(\widetilde{x}) = P \in D)$

  (IF1) $\quad \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \longrightarrow P_1 \quad (e \downarrow \texttt{true})$

  (IF2) $\quad \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \longrightarrow P_2 \quad (e \downarrow \texttt{false})$

The rest of the structural congruence and the reduction rules are the obvious adaptations.

Example: Variable Agent

A variable agent stores a value and offers the following operations:
1) read returns the stored value and recurs to the same variable;
2) write receives a different value and returns to the variable with the new state;

A Variable Agent:

$$\mathbf{Var}(a, x) \stackrel{\text{df}}{=} ?$$

Reader Process:

$$\mathbf{Reader}(a) \stackrel{\text{df}}{=} ?$$

Example: Variable Agent

A variable agent stores a value and offers the following operations:
1) read returns the stored value and recurs to the same variable;
2) write receives a different value and returns to the variable with the new state;

A Variable Agent:

$$\mathbf{Var}(a,x) \stackrel{\text{df}}{=} a(z).z \rhd [\text{read} : \overline{z}\langle x\rangle.\mathbf{Var}\langle a,x\rangle \;[]\; \text{write} : z(y).\mathbf{Var}\langle a,y\rangle]$$

Reader Process:

$$\mathbf{Reader}(a) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.s \lhd \text{read}.s(y).\mathbf{0}$$

Write Process:

$$\mathbf{Writer}(a,x) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.s \lhd \text{write}.\overline{s}\langle x\rangle.\mathbf{0}$$

Updating a value:

$$\mathbf{Var}\langle a,x\rangle \mid \mathbf{Writer}\langle a,5\rangle \longrightarrow \mathbf{Var}\langle a,5\rangle$$

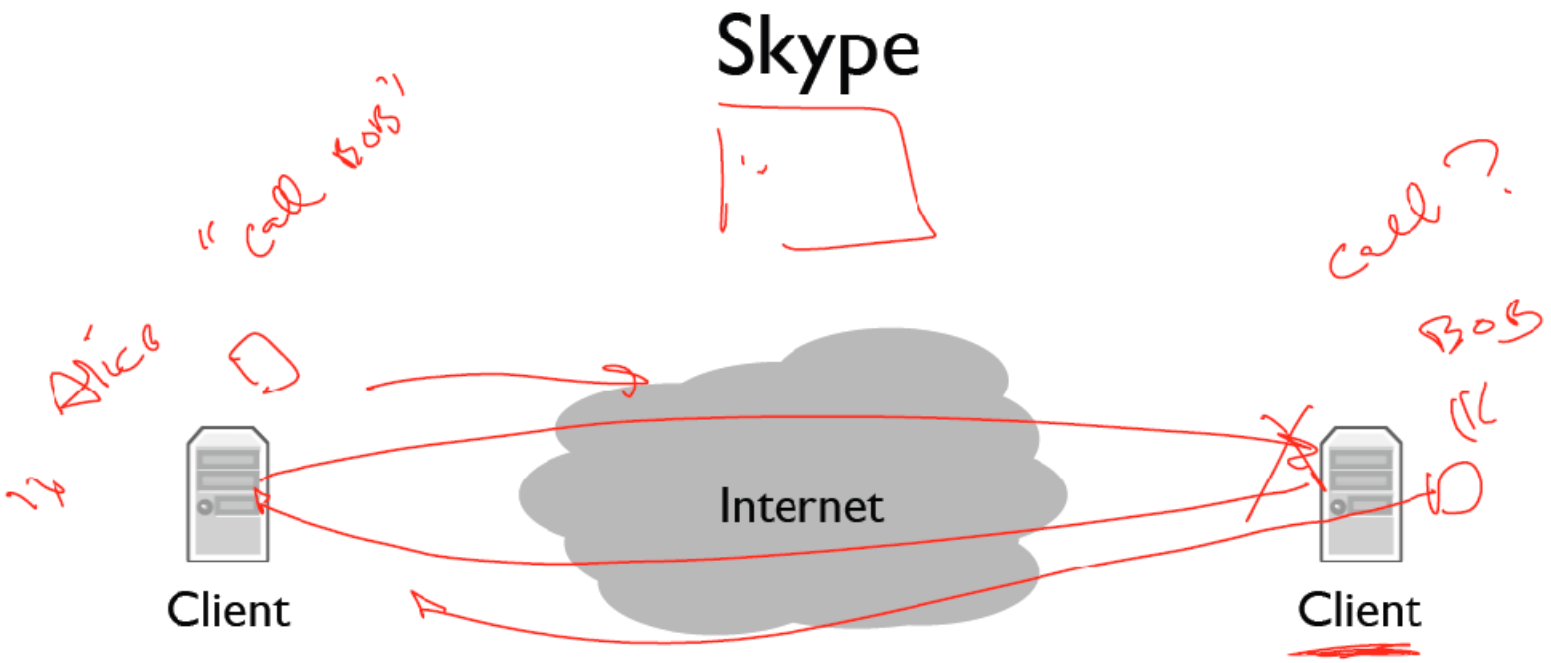
Example: Web Server

$$\mathbf{WebServer}(a) \stackrel{\mathsf{df}}{=} a(z).z(url).\ \texttt{if correct}(url)\ \texttt{then}\ z \lhd \mathsf{ok}.\ \overline{z}\langle \mathsf{pageSource}\rangle.\mathbf{WebServer}\langle a\rangle\ \texttt{else}\ z \lhd \mathsf{error}.\mathbf{WebServer}\langle a\rangle$$

$$\mathbf{Client}(a) \stackrel{\mathsf{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.\mathbf{Client}_1\langle s\rangle$$

$$\mathbf{Client}_1(s) \stackrel{\mathsf{df}}{=} \overline{s}\langle\texttt{"http://imperial..."}\rangle.\ s \rhd [\mathsf{ok} : s(page).\mathbf{Client}\langle a\rangle\ []\ \mathsf{error} : \mathbf{Client}\langle a\rangle]$$

Skype(Login Service)



Skype(Login Service)

Here is an example of a simplified chat service with authentication. Alice sends a message to Bob. Before replying to Alice, Bob sends a request to the LoginService whether Alice is in the list of his friends. If Alice is among Bob's friends, Bob replies with a greeting message, otherwise he rejects the conversation.

$$\mathbf{Alice}(a) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.\overline{s}\langle \texttt{"Alice"}\rangle.s \rhd [\mathsf{accept} : s(msg).\mathbf{0} \;[]\; \mathsf{reject} : s(reason).\mathbf{Alice}\langle a\rangle]$$

$$\begin{aligned}\mathbf{Bob}(a, b, greeting) \stackrel{\text{df}}{=}\;& a(z).z(userId).(\nu\, s')\overline{b}\langle s'\rangle.s' \lhd \mathsf{check}.\overline{s'}\langle userId\rangle.s'(x).\\ & \texttt{if } x \texttt{ then } z \lhd \mathsf{accept} : \overline{z}\langle greeting\rangle.\mathbf{Bob}\langle a, b, greeting\rangle\\ & \texttt{else } z \lhd \mathsf{reject} : \overline{z}\langle \texttt{"You are not my friend"}\rangle.\mathbf{Bob}\langle a, b, greeting\rangle\end{aligned}$$

$$\mathbf{LoginService}(b) \stackrel{\text{df}}{=} b(z).z \rhd [\mathsf{check} : z(userId).\overline{z}\langle \texttt{true}\rangle.\mathbf{LoginService}\langle b\rangle \;[]\; \ldots]$$

Web Service Protocol (Usecase from WS-CDL)

Two parties are involved: a client (Customer) and a travel agency (Agency).

1. Customer begins an *order session* s with Agency, then requests and receives the price for the desired journey.

2. Customer either accepts (label accept) an offer from Agency or decides that none of the received quotes are satisfactory.

3. if the offer is accepted, the Customer sends a delivery address and the Agency Service replies with the dispatch date for the purchased tickets. The transaction is now complete.

4. Customer retries (label retry) transactions with new journeys some number of times if Agency gave are reasonable quote.

5. Customer rejects (label reject) the transaction if no quotes were suitable after some retries and the session terminates.

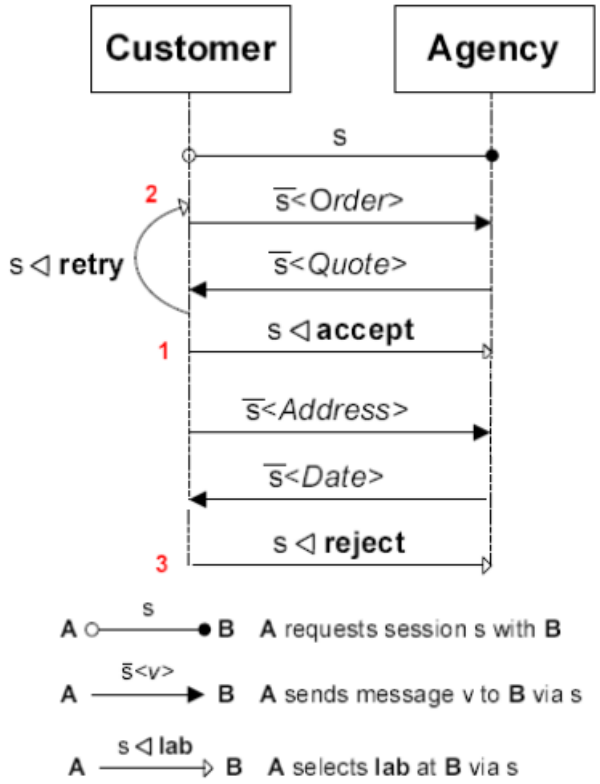
Web Service Protocols (Agency and Service)

$$\mathbf{Agency}(a,b) \stackrel{\text{df}}{=} a(y).\mathbf{Agency}_1\langle a,b,y\rangle$$

$$\begin{aligned}\mathbf{Agency}_1(a,b,s) \stackrel{\text{df}}{=}\ & s(x).\overline{s}\langle \texttt{price}(x)\rangle \\ & s \rhd [\ \ \mathsf{accept} : s(x).\overline{s}\langle \texttt{date}\rangle.\mathbf{Agency}\langle a,b\rangle \ \ [] \\ & \qquad \mathsf{retry} : \mathbf{Agency}_1\langle a,b,s\rangle \ \ [] \\ & \qquad \mathsf{reject} : \mathbf{Agency}\langle a,b\rangle]\end{aligned}$$

$$\mathbf{Customer}(a,place,i,n) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.\mathbf{Customer}_1\langle s,place,i,n\rangle$$

$$\begin{aligned}&\mathbf{Customer}_1(s,place,i,n) \stackrel{\text{df}}{=} && \overline{s}\langle place(i)\rangle.s(price). \\ &\texttt{if is\_acceptable}(price)\ \texttt{then} && s \lhd \mathsf{accept}.\overline{s}\langle \texttt{address}\rangle.s(date) \\ &\texttt{elseif}\ i \leq n\ \texttt{then} && s \lhd \mathsf{retry}.\mathbf{Customer}_1\langle s,place,i+1,n\rangle \\ &\texttt{else} && s \lhd \mathsf{reject}\end{aligned}$$



Delegation

- The original idea of delegation in object-based concurrent programming allows an object to delegate the processing of a request to another object. Its basic purpose is distributing of processing, while maintaining the transparency of name space for clients of that service.

- Can we delegate the processing of a request in the current session calculus ?

- Refresh on the syntax:

$v ::=$	value identifier
$\texttt{true}, \texttt{false}, s, a$	values

  Therefore, we can pass session channels: $\overline{s}\langle s' \rangle$

- Delegation: the ability to pass session channels

- Let's see how we can improve the Web Service Agency example using delegation.

Web Service Protocol (Usecase from WS-CDL)

Adding a third party - A Travel Service. Customer and Service are initially unknown to each other but later communicate directly through the use of *name passing*.

1. Customer begins an *order session* s with Agency, then requests and receives the price for the desired journey.

2. Customer either accepts an offer from Agency or decides that none of the received quotes are satisfactory.

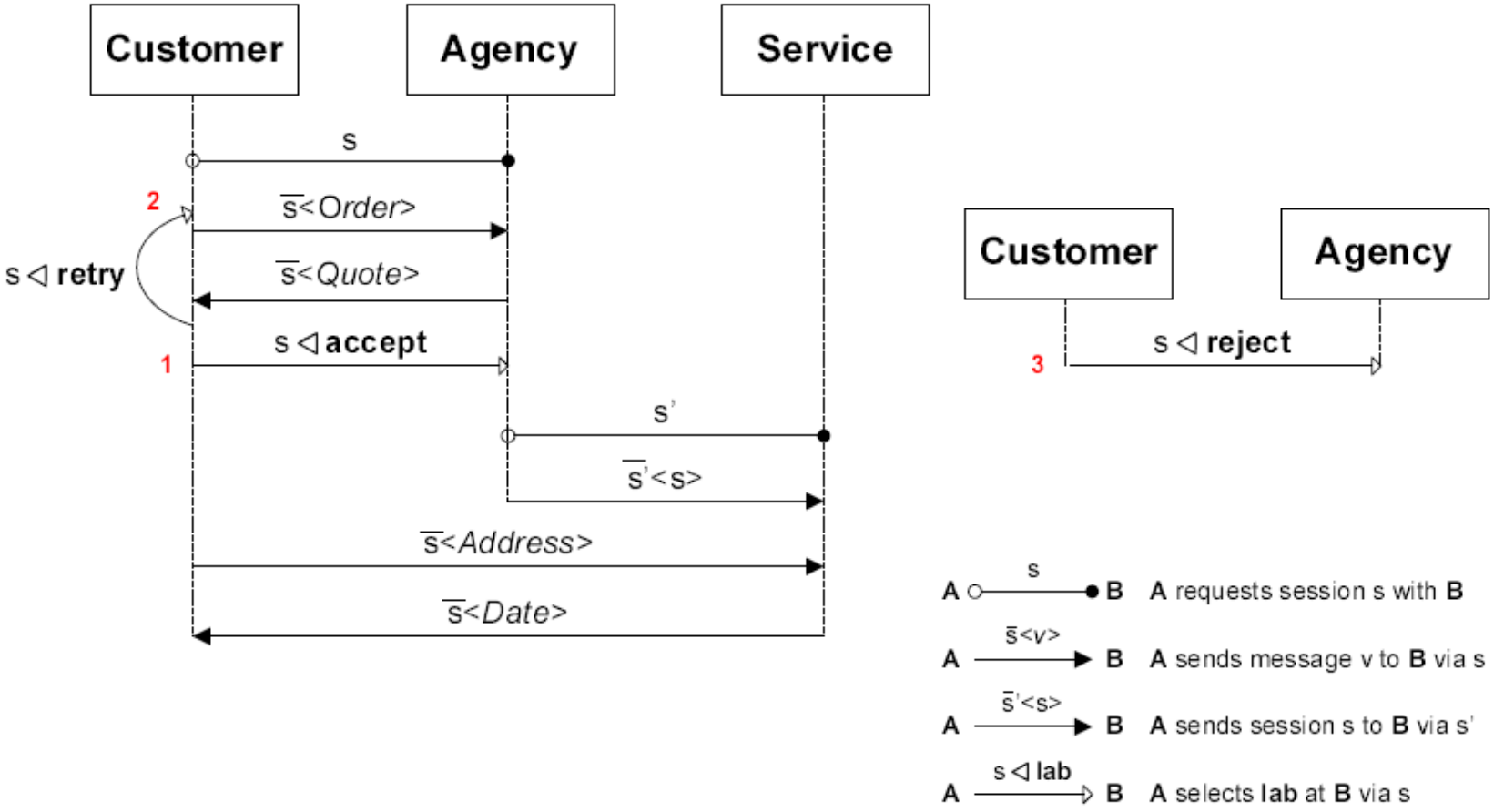
3. new: If an offer is accepted, Agency opens the session s' with Service and *delegates* to Service, through s', the interactions with Customer remaining for s.

4. new: Customer then sends a delivery address (unaware that he/she is now talking to Service), and Service replies with the dispatch date for the purchased tickets. The transaction is now complete.

5. Customer retries transactions with new journeys some number of times if Agency gave are reasonable quote.

6. Customer rejects the transaction if no quotes were suitable after some retries and the session terminates.

Web Service Protocols (Diagram)



Web Service Protocols (Agency and Service)

Agency is defined with the two nested recursions.

$$\mathbf{Agency}(a,b) \stackrel{\text{df}}{=} a(y).\mathbf{Agency}_1\langle a,b,y\rangle$$

$$\mathbf{Agency}_1(a,b,s) \stackrel{\text{df}}{=}$$

$$s(x).\overline{s}\langle \texttt{price(x)}\rangle.s \rhd [\ \begin{array}{ll} \text{accept}: & (\nu\, s')\overline{b}\langle s'\rangle.\overline{s'}\langle s\rangle.\mathbf{Agency}\langle a,b\rangle \ \ [] \\ \text{retry}: & \mathbf{Agency}_1\langle a,b,s\rangle \ \ [] \\ \text{reject}: & \mathbf{Agency}\langle a,b\rangle] \end{array}$$

The message $\overline{s'}\langle s\rangle$ delegates the interaction with Customer to Service. Service is defined as:

$$\mathbf{Service}(b) \stackrel{\text{df}}{=} b(y).y(x).\overline{x}\langle \texttt{Date}\rangle.\mathbf{Service}_1\langle x\rangle$$

where $y(x)$ receives the session with Customer so that $\mathbf{Service}_1\langle x\rangle$ can continue the interaction with Customer as if it were Agency.

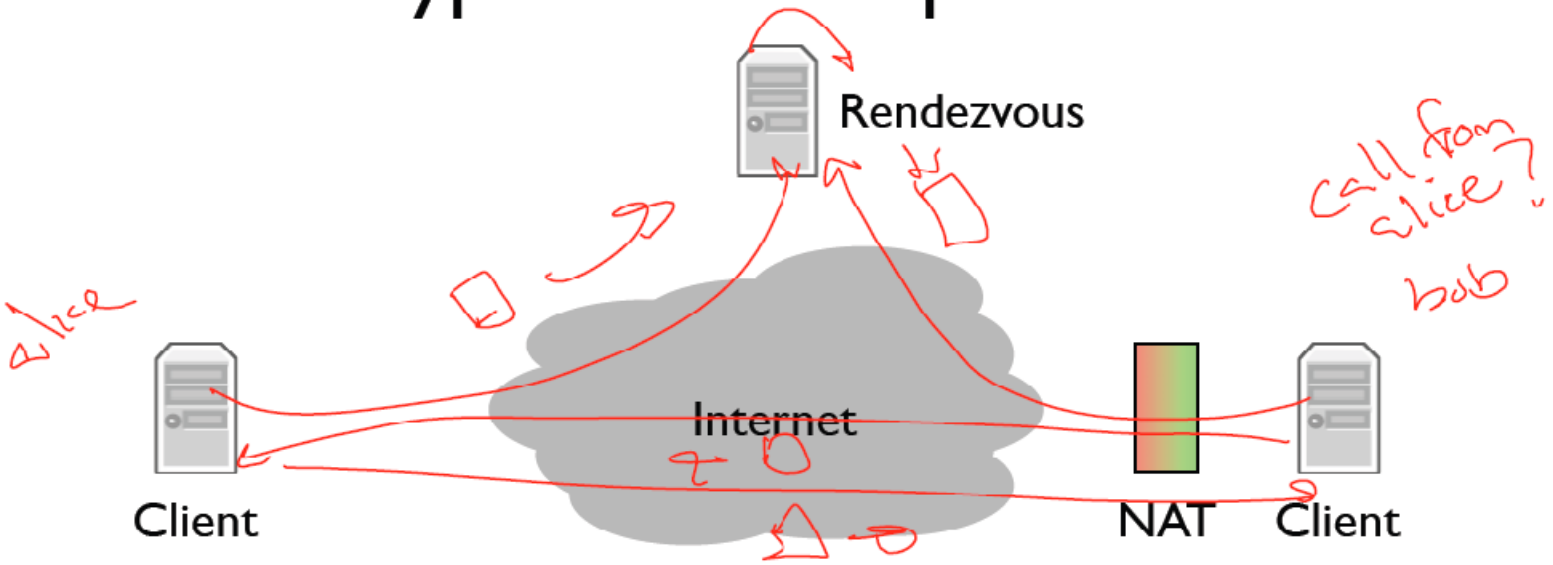
Web Service Protocols (Customer)

An example of Customer is given as:

$$\mathbf{Customer}(a, place, n) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.\mathbf{Customer}_1\langle s, place, 1, n\rangle$$

$$\begin{array}{ll}
\mathbf{Customer}_1(s, place, i, n) \stackrel{\text{df}}{=} & \overline{s}\langle place(i)\rangle.s(price). \\
\texttt{if is_acceptable}(price)\ \texttt{then} & s \lhd \mathsf{accept}.\overline{s}\langle \texttt{Address}\rangle.s(date).\mathbf{Customer}_1\langle s\rangle \\
\texttt{elseif}\ i \leq n\ \texttt{then} & s \lhd \mathsf{retry}.\mathbf{Customer}_1\langle s, place, i+1, n\rangle \\
\texttt{else} & s \lhd \mathsf{reject}
\end{array}$$

Skype with Complications



Skype with Complications

Bob is behind a firewall and Alice cannot contact him directly. Thus, Alice contacts the broker node first and it redirects her request to Bob.

$$\mathbf{Broker}(a,c) \stackrel{\text{df}}{=} a(s).\mathbf{Broker_1}\langle a,c,s\rangle$$

$$\mathbf{Broker_1}(a,c,s) \stackrel{\text{df}}{=} (\nu\, s')\overline{c}\langle s'\rangle.\overline{s'}\langle s\rangle.\mathbf{Broker}\langle a,c\rangle$$

$$\mathbf{Alice}(a) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.\overline{s}\langle\texttt{"Alice"}\rangle.s \rhd [\mathsf{accept} : s(msg) \,[]\, \mathsf{reject} : s(reason).\mathbf{Alice}\langle a\rangle]$$

$$\begin{aligned}\mathbf{Bob}(c,b) \stackrel{\text{df}}{=}\ & c(y).y(z).z(userId).(\nu\, s')\overline{b}\langle s'\rangle.s' \lhd \mathsf{check}.\overline{s'}\langle userId\rangle.s'(x)\\ & \texttt{if } x \texttt{ then } z \lhd \mathsf{accept} : s(\texttt{"Hi"})\\ & \texttt{else } z \lhd \mathsf{reject} : \overline{z}\langle\texttt{"You are not my friend"}\rangle.\mathbf{Bob}\langle a\rangle\end{aligned}$$

$$\mathbf{LoginService}(b) \stackrel{\text{df}}{=} b(z).z \rhd [\mathsf{check} : z(userId).\overline{z}\langle\texttt{true}\rangle.\mathbf{LoginService}\langle b\rangle \,[]\, \ldots]$$

Note that the Alice process is the same, the Login Service stays the same.

Error-Freedom for Protocols

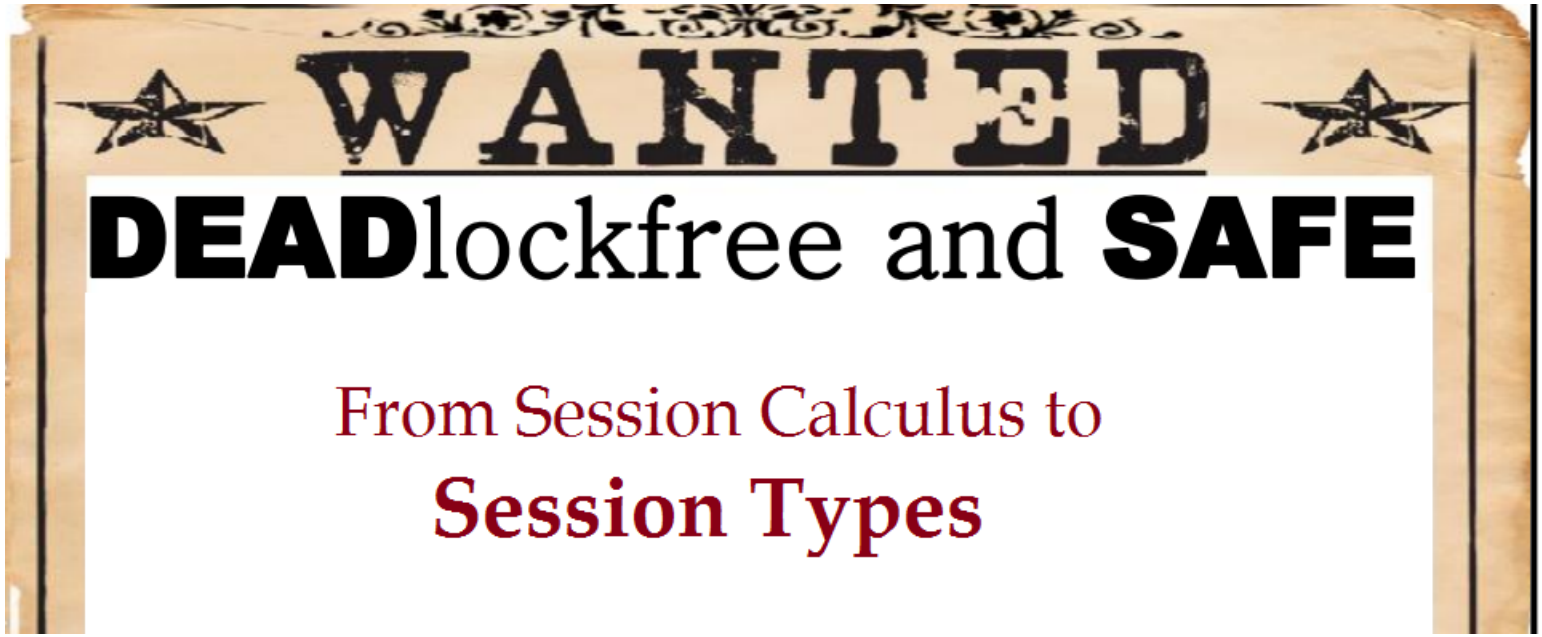
We wish to avoid the following runtime errors in various protocols.

- Base Type Error $\overline{s}\langle \texttt{Apple} \rangle.P_1 \mid s(x).\overline{s'}\langle 1 + x \rangle$
- Arity Mismatch $\overline{s}\langle 1 \rangle.P_1 \mid s(x, y).\overline{s'}\langle x + y \rangle$
- Label Undefined $s \rhd \{\mathsf{repeat} : P_1 \;[\!]\; \mathsf{reject} : P_2\} \mid s \lhd \mathsf{apple}$
- Race during Session Interaction
  Bad $s(x).P_1 \mid \overline{s}\langle v \rangle.P_2 \mid \overline{s}\langle w \rangle.P_3$
  Good $s(x).P_1 \mid \overline{s}\langle v \rangle.P_2 \mid s'(x).Q_1 \mid \overline{s'}\langle w \rangle.Q_2$
- Communication Mismatch
  Bad $s(x).\overline{s}\langle w \rangle.\mathbf{0} \mid s(y).\overline{s}\langle v \rangle.\mathbf{0}$ Good $s(x).\overline{s}\langle w \rangle.\mathbf{0} \mid \overline{s}\langle v \rangle.s(y).\mathbf{0}$

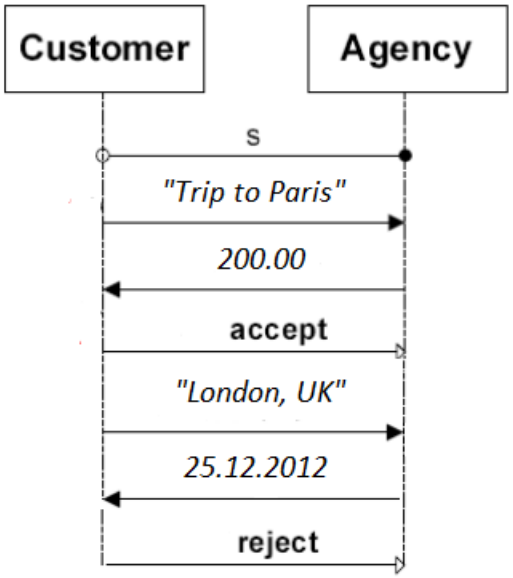
But the following should be OK since a is a shared channel.

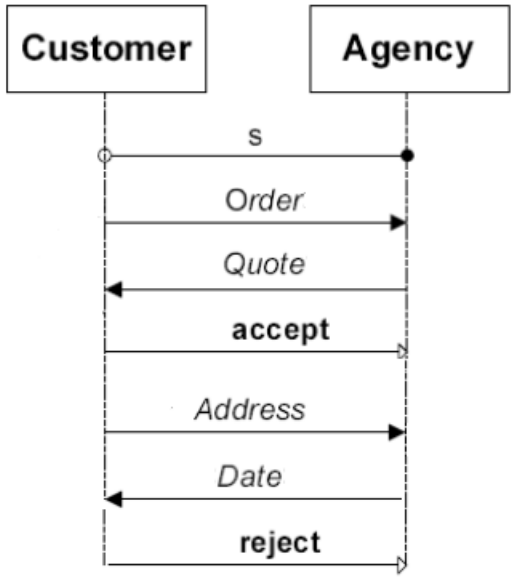
$$!a(x).P \mid (\nu\, s_1)\overline{a}\langle s_1 \rangle.P_1 \mid \cdots \mid (\nu\, s_n)\overline{a}\langle s_n \rangle.P_n$$

Can we *statically* ensure no such errors occur during communications programming!


WANTED
DEADlockfree and SAFE
From Session Calculus to
Session Types

From Session Calculi to Session Types





Session Calculus : $\overline{s}\langle place(\mathtt{i})\rangle.s(price).$ $\mathtt{if\ is\_acceptable}(price)$ $\mathtt{then}\ s \lhd \mathtt{accept}.\overline{s}\langle \mathtt{address}\rangle.s(date)$ $\mathtt{else}\ s \lhd \mathtt{reject}$

Session Types : $!\mathtt{[string]}; ?\mathtt{[double]}; \oplus\{$ $\mathtt{accept} : !\mathtt{[string]}; ?\mathtt{[nat,nat,nat]}; \mathtt{end},$ $\mathtt{reject} : \mathtt{end}\}$

Session Types Introduction

So far we have seen that using session calculus we can give a clear description of a complex interaction protocols. The more complex the interaction becomes, the more difficult it is to capture the whole interactive behaviour and to write correct programs.

The session type discipline offers a simple static checking framework to guarantee the correctness of communication patterns. Through the type system we can algorithmically check if a program has a coherent communication structure and if it contains interaction errors. The syntax for types is given below:

Session Types Syntax

S ::=		Sort
	$\texttt{bool} \mid \texttt{nat} \mid \texttt{string} \ldots$	
T ::=		Type
	$![\widetilde{S}]; T$	sending a value of type $\widetilde{S}$
	$![T]; T'$	sending a type T (delegation)
	$\&\{l_1 : T_1, \ldots, l_n : T_n\}$	branching behaviour (external choice)
	$?[\widetilde{S}]; T$	receiving a value of type $\widetilde{S}$
	$?[T]; T'$	receiving a type T (delegation)
	$\oplus \{l_1 : T_1, \ldots, l_n : T_n\}$	selection (internal choice)
	t	
	$\mu t.T$	recursive behaviour
	$\texttt{end}$	end of a session

The type $![\widetilde{S}];T$ represents the behaviour of first sending values of type $\widetilde{S}$ and then continues as specified by the type T;
$![T];T'$ represents similar behaviour, which starts with sending a channel(delegation) instead.

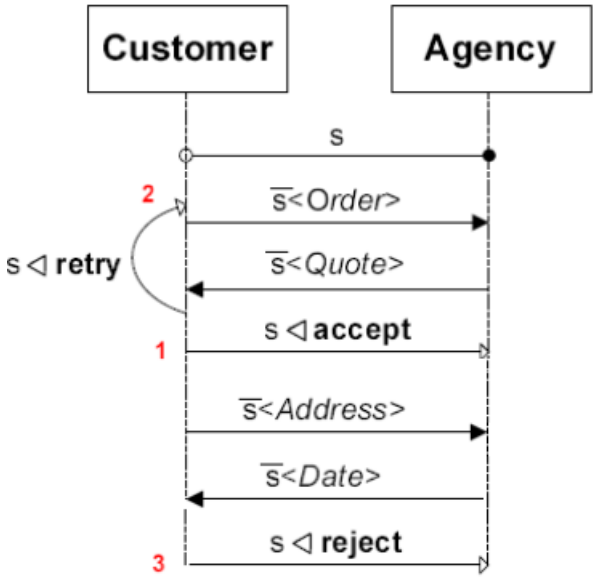
The $?[\widetilde{S}];T$ and $?[T];T'$ are the dual of $![\widetilde{S}];\overline{T}$ and $![T];\overline{T'}$ respectively, receiving values, instead of sending.

$\mu t.T$ represents recursive behaviour - start doing T, when t is encountered recur to T again.

$\&\{l_1 : T_1, \ldots, l_n : T_n\}$ shows the branching behaviour: it waits with n options, and behaves as type T_i if i-th action is selected (external choice).

$\oplus\{l_1 : T_1, \ldots, l_n : T_n\}$ then represents the behaviour which would select one of l_i and then behaves as T_i, according to the selected l_i (internal choice).

Example



$\mathbf{Customer}(a, place, i, n) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.\mathbf{Customer_1}\langle s, place, i, n\rangle$

$\mathbf{Customer_1}(s, place, i, n) \stackrel{\text{df}}{=}$

```
if is_acceptable(price) then
else if i ≤ n then
else
```

$\overline{s}\langle place(\mathtt{i})\rangle.s(price).$
$s \triangleleft \mathsf{accept}.\overline{s}\langle \mathtt{address}\rangle.s(date)$
$s \triangleleft \mathsf{retry}.\mathbf{Customer_1}\langle s, place, i+1, n\rangle$
$s \triangleleft \mathsf{reject}$

$T = \mu t.(![\mathtt{string}]; ?[\mathtt{double}]; \oplus\{ \mathsf{accept} :![\mathtt{string}]; ?[\mathtt{nat},\mathtt{nat},\mathtt{nat}]; \mathtt{end},$
$\mathsf{retry} :t,$
$\mathsf{reject} :\mathtt{end}\})$

Properties of Session Types

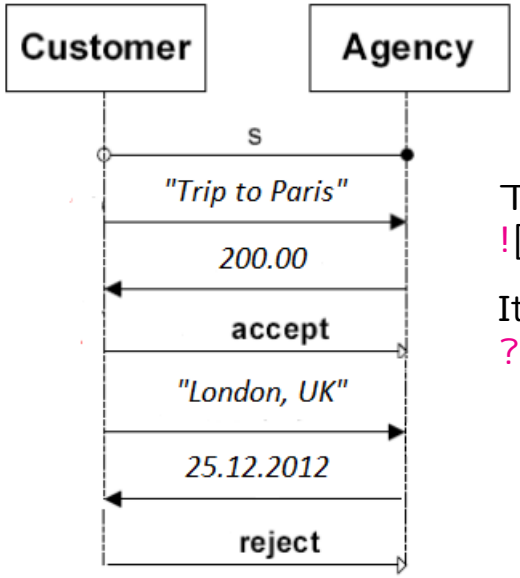
1. Communication Error-Freedom
   No communication mismatch

2. Session Fidelity
   The communication sequence in a session follows the scenario declared in the types.

3. Progress
   No deadlock/ Stuck in a session

Duality

if P has type T and Q has type $\overline{T}$ (the dual type of T) then $P \mid Q$ is typable (no communication errors).



The type for the Customer is:
$![\texttt{string}]; ?[\texttt{double}]; \oplus\{\text{accept} :![\texttt{string}]; ?[\texttt{nat,nat,nat}]; \texttt{end}, \text{reject} :\texttt{end}\}$

Its dual type is:
$?[\texttt{string}]; ![\texttt{double}]; \&\{\text{accept} :?[\texttt{string}]; ![\texttt{nat,nat,nat}]; \texttt{end}, \text{reject} :\texttt{end}\}$

Co-types (Dual Types)

For a type T, its dual or co type, written $\overline{T}$, is defined by exchanging ? and !, and & and $\oplus$. The inductive definition is given below:

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
\overline{?[\widetilde{S}];\, T} & = & ![\widetilde{S}];\, \overline{T} & \overline{\oplus\{l_i : T_i\}_{i\in I}} & = & \&\{l_i : \overline{T_i}\}_{i\in I} & \overline{?[T];\, T'} & = & ![T];\, \overline{T'} \\
\overline{![\widetilde{S}];\, T} & = & ?[\widetilde{S}];\, \overline{T} & \overline{\&\{l_i : T_i\}_{i\in I}} & = & \oplus\{l_i : \overline{T_i}\}_{i\in I} & \overline{![T];\, T'} & = & ?[T];\, \overline{T'} \\
\overline{\mathtt{end}} & = & \mathtt{end} & \overline{\mu t.\, T} & = & \mu t.\, \overline{T} & \overline{t} & = & t
\end{array}
$$

Duality is essential for checking type compatability. Compatible types mean that each common channel k is associated with complementary behaviour, thus ensuring the interactions on k to run without errors.

Exercise

Give the dual type for the following types:

1. $![\texttt{string}]; ?[\texttt{int}]$

2. $![\texttt{string}]; ![T']; \texttt{end}$

3. $\&\{\texttt{read}:?[\texttt{nat}]; \texttt{end}, \texttt{write}:![\texttt{nat}]; \texttt{end}\}$

4. $\mu t. \oplus \{\texttt{read}:?[\texttt{nat}]; t, \texttt{write}:![\texttt{nat}]; \texttt{end}\}$

Types: Variable Example

Consider the variable example from the beginning of the lecture.

$$\mathbf{Var}(a, x) = a(z).z \triangleright [\mathsf{read} : \overline{z}\langle x\rangle.\mathbf{Var}\langle a, x\rangle \,[]\, \mathsf{write} : z(y).\mathbf{Var}\langle a, y\rangle]$$

The corresponding type is

$$T = \&\{\texttt{read:!}[\texttt{nat}]; \texttt{end}, \texttt{write:?}[\texttt{nat}]; \texttt{end}\}$$

The dual type is

$$\overline{T} = \oplus\{\texttt{read:?}[\texttt{nat}]; \texttt{end}, \texttt{write:!}[\texttt{nat}]; \texttt{end}$$

Types: Variable Example (cont.)

Consider now the reader process:

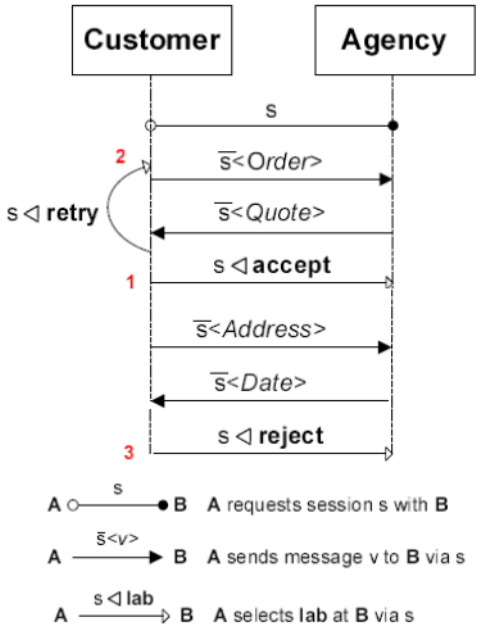
$$\mathbf{Reader}(a) = (\nu\, s)\overline{a}\langle s\rangle.s \lhd \mathsf{read}.s(y).\mathbf{0}$$

Which is a correct type for $\mathbf{Reader}(a)$?

$$(a) \quad T' = \oplus\{\mathtt{read}\mathtt{:?[nat]};\mathtt{end}\}$$

$$(b) \quad \overline{T} = \oplus\{\mathtt{read}\mathtt{:?[nat]};\mathtt{end},\mathtt{write}\mathtt{:![nat]};\mathtt{end}\}$$

Types: Web Service example



$$\mathbf{Customer}(a, place, i, n) \stackrel{\text{df}}{=} (\nu\, s)\overline{a}\langle s\rangle.\mathbf{Customer}_1\langle s, place, i, n\rangle$$

$$\mathbf{Customer}_1(s, place, i, n) \stackrel{\text{df}}{=}$$

$$\overline{s}\langle place(i)\rangle.s(price).$$

$$\texttt{if is\_acceptable}(price)\ \texttt{then}\quad s \lhd \mathsf{accept}.\overline{s}\langle \texttt{address}\rangle.s(date).\mathbf{Customer}_1\langle s\rangle$$

$$\texttt{elseif}\ i \leq n\ \texttt{then}\quad s \lhd \mathsf{retry}.\mathbf{Customer}_1\langle s, place, i+1, n\rangle$$

$$\texttt{else}\quad s \lhd \mathsf{reject}$$

Exercise: give the session type for $\mathbf{Customer}$.

Types: Web Service example with delegation

$$\mathbf{Agency}(a,b) \stackrel{\text{df}}{=} a(y).\mathbf{Agency}_1\langle a,b,y\rangle$$

$$\mathbf{Agency}_1(a,b,s) \stackrel{\text{df}}{=}$$

$$s(x).\overline{s}\langle \texttt{price(x)}\rangle.s \rhd [\quad \begin{array}{ll} \text{accept}: & (\nu\, s')\overline{b}\langle s'\rangle.\overline{s'}\langle s\rangle.\mathbf{Agency}\langle a,b\rangle \quad [\!] \\ \text{retry}: & \mathbf{Agency}_1\langle a,b,s\rangle \quad [\!] \\ \text{reject}: & \mathbf{Agency}\langle a,b\rangle] \end{array}$$

$$T \stackrel{\text{df}}{=} \mu t.(?[\texttt{string}];![\texttt{double}];\&\{\texttt{accept}:![T''];\texttt{end},\texttt{retry}:t,\texttt{quit}:\texttt{end}\})$$

$$T' \stackrel{\text{df}}{=} \mu t.(![\texttt{string}];?[\texttt{double}];\oplus\{\texttt{accept}:![\texttt{string}];?[\texttt{double}]\},\texttt{retry}:t,\texttt{quit}:\texttt{end})$$

$$T'' \stackrel{\text{df}}{=} ?[\texttt{string}];![\texttt{double}].\texttt{end}$$

Exercise: give the session type for **Customer**.

Readings

- Takeuchi, Honda, Kubo: **An Interaction-based Language and its Typing System.** PARLE 1994: 398-413

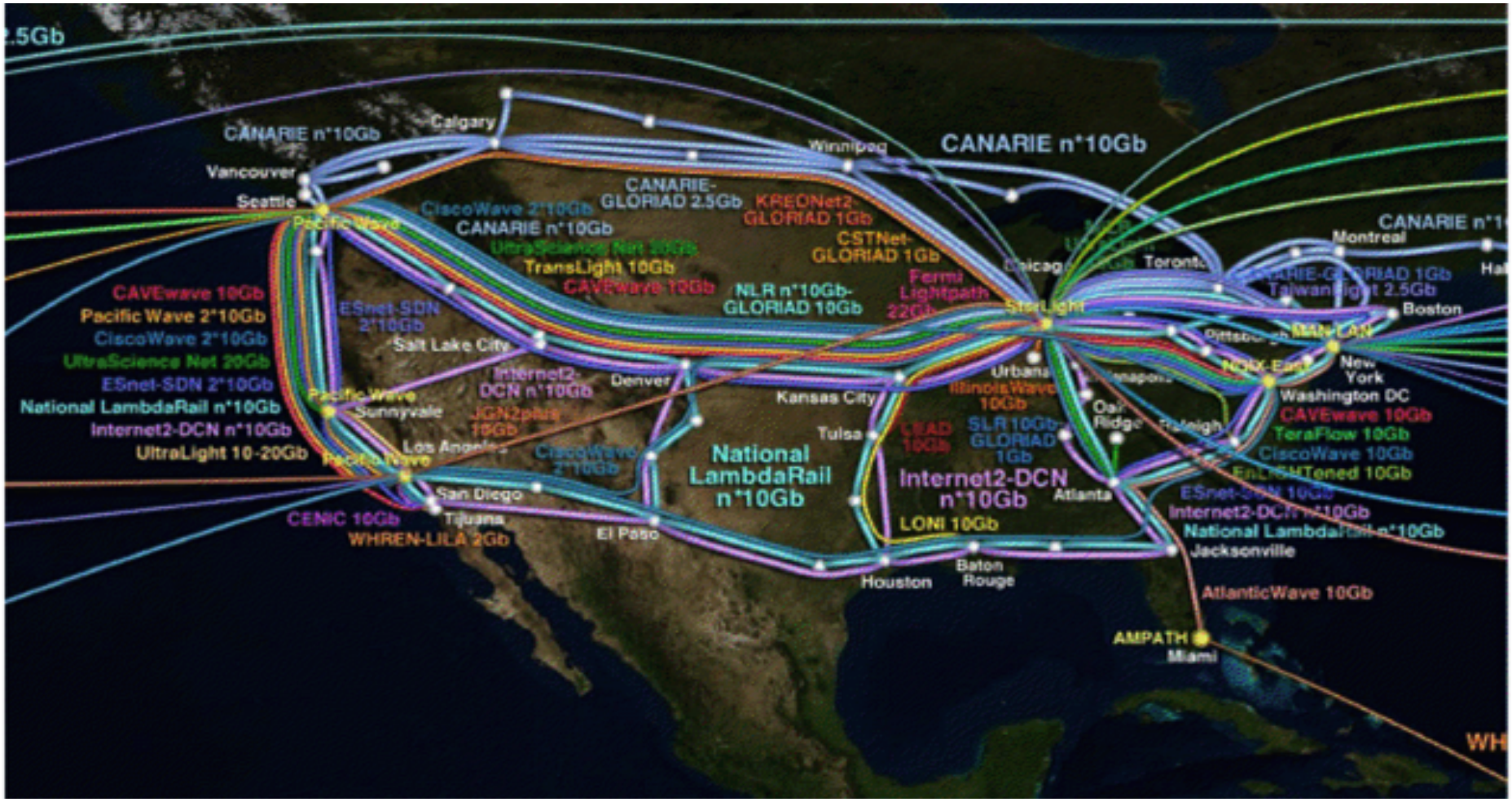
- Honda, Vasconcelos, Kubo:
Language Primitives and Type Discipline for Structured Communication-Based Programming. ESOP 1998: 122-138

- Yoshida, Vasconcelos: **Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session.** Communication. Electr. Notes Theor. Comput. Sci. 171(4): 73-93 (2007)

For the curious : Useful resources

- Session Types in Java(SJ): http://www.doc.ic.ac.uk/~rhu/
- Session Types for parallel programming (SessionC): http://www.doc.ic.ac.uk/~cn06/page/
- Session Types in Python: http://www.doc.ic.ac.uk/~rn710/mon/
- Ocean Observatory Institute: http://www.oceanobservatories.org/
- SePiConcurrent, message-passing programming language based on the π-calculus http://gloss.di.fc.ul.pt/sepi


Networks
Session Types
Types


CANARIE n*10Gb
Calgary
Vancouver
Seattle
Winnipeg
CANARIE n*10Gb
CANARIE-GLORIAD 2.5Gb
KREONet2-GLORIAD 1Gb
CiscoWave 2*10Gb
CANARIE n*10Gb
CSTNet-GLORIAD 1Gb
TransLight 10Gb
CAVEwave 10Gb
NLR n*10Gb-GLORIAD 10Gb
Fermi Lightpath 22Gb
CAVEwave 10Gb
Pacific Wave 2*10Gb
CiscoWave 2*10Gb
UltraScience Net 20Gb
ESnet-SDN 2*10Gb
National LambdaRail n*10Gb
Internet2-DCN n*10Gb
UltraLight 10-20Gb
ESnet-SDN 2*10Gb
Salt Lake City
Internet2-DCN n*10Gb
Denver
Pacific Wave
Sunnyvale
Kansas City
Tulsa
Los Angeles
CiscoWave 2*10Gb
National LambdaRail n*10Gb
Internet2-DCN n*10Gb
LONI 10Gb
SLR 10Gb-GLORIAD 1Gb
Atlanta
San Diego
Tijuana
CENIC 10Gb
WHREN-LILA 2Gb
El Paso
Houston
Baton Rouge
Jacksonville
AtlanticWave 10Gb
AMPATH
Miami
Montreal
Toronto
Boston
MAN LAN
New York
Washington DC
CAVEwave 10Gb
TeraFlow 10Gb
CiscoWave 10Gb
Urbana
Oak Ridge
Raleigh

Do you want to be a part of that?

Come and enjoy a project with us.