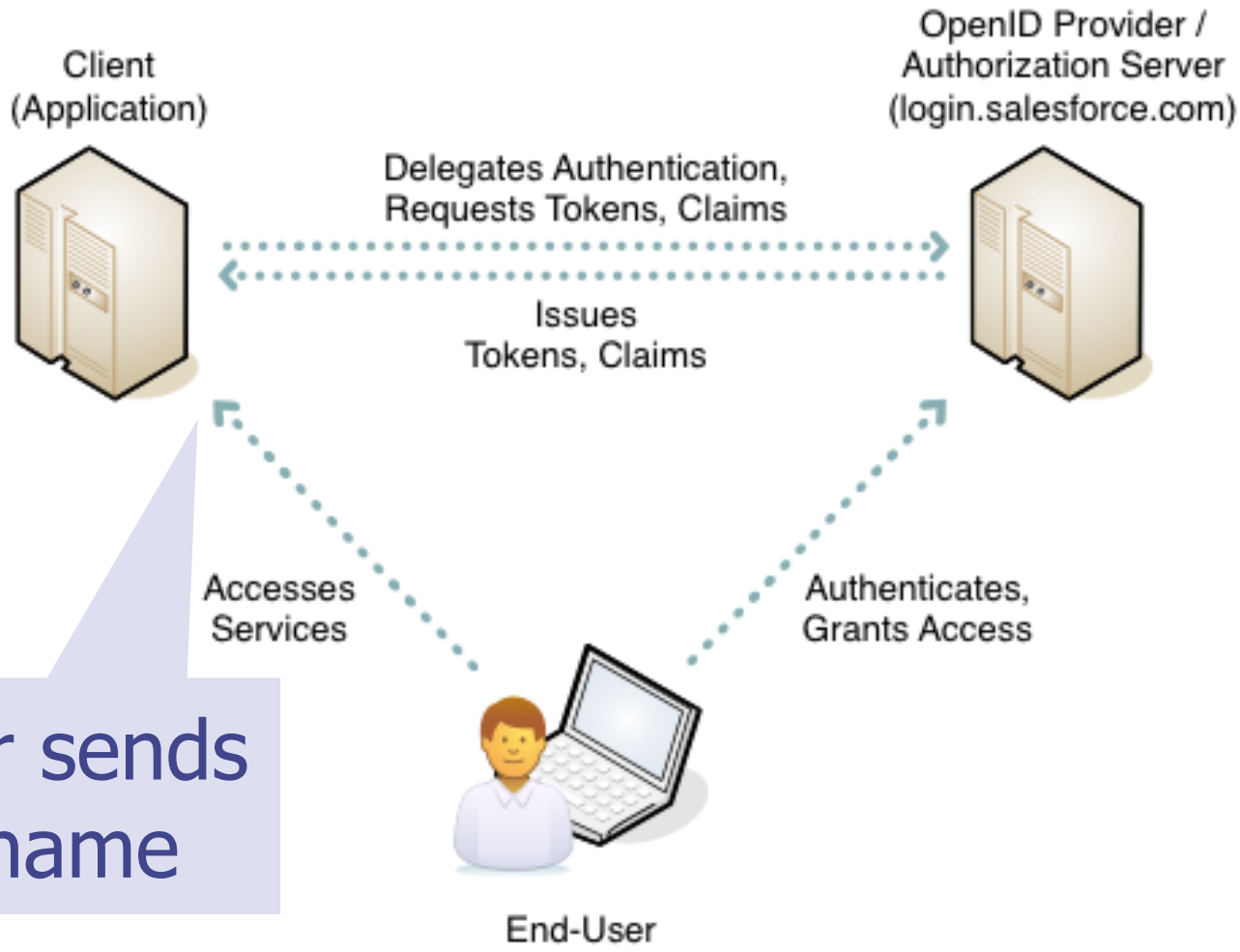
Orchestration vs Choreography

- In many cases, there is **no unique point** of invocation for the services
  - In these cases, we say that the system is a **choreography**
  - Let starts with an example:
    - Consider the popular OpenID **protocol** for distributed authentication via third party services

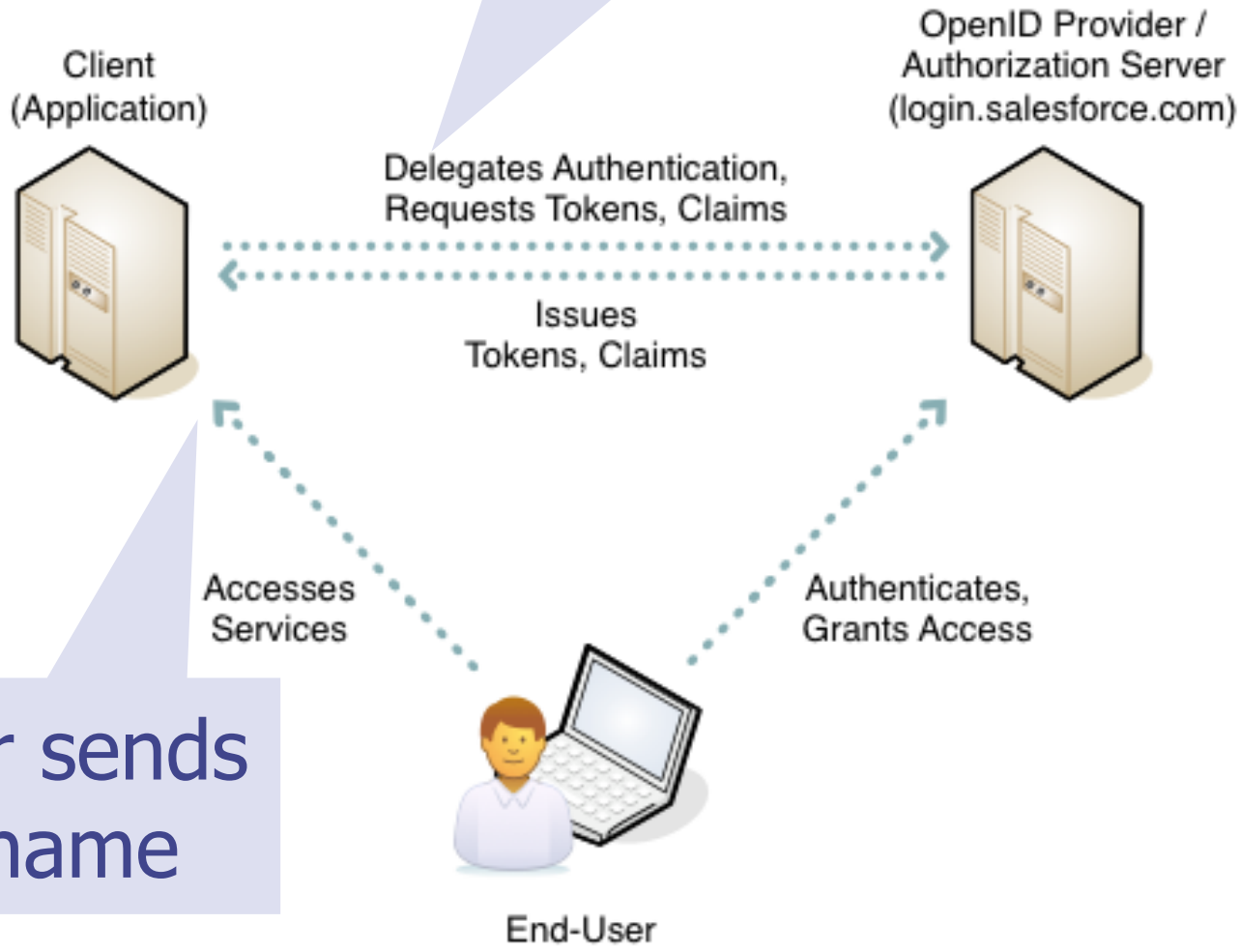
OpenID



1. User sends username

OpenID

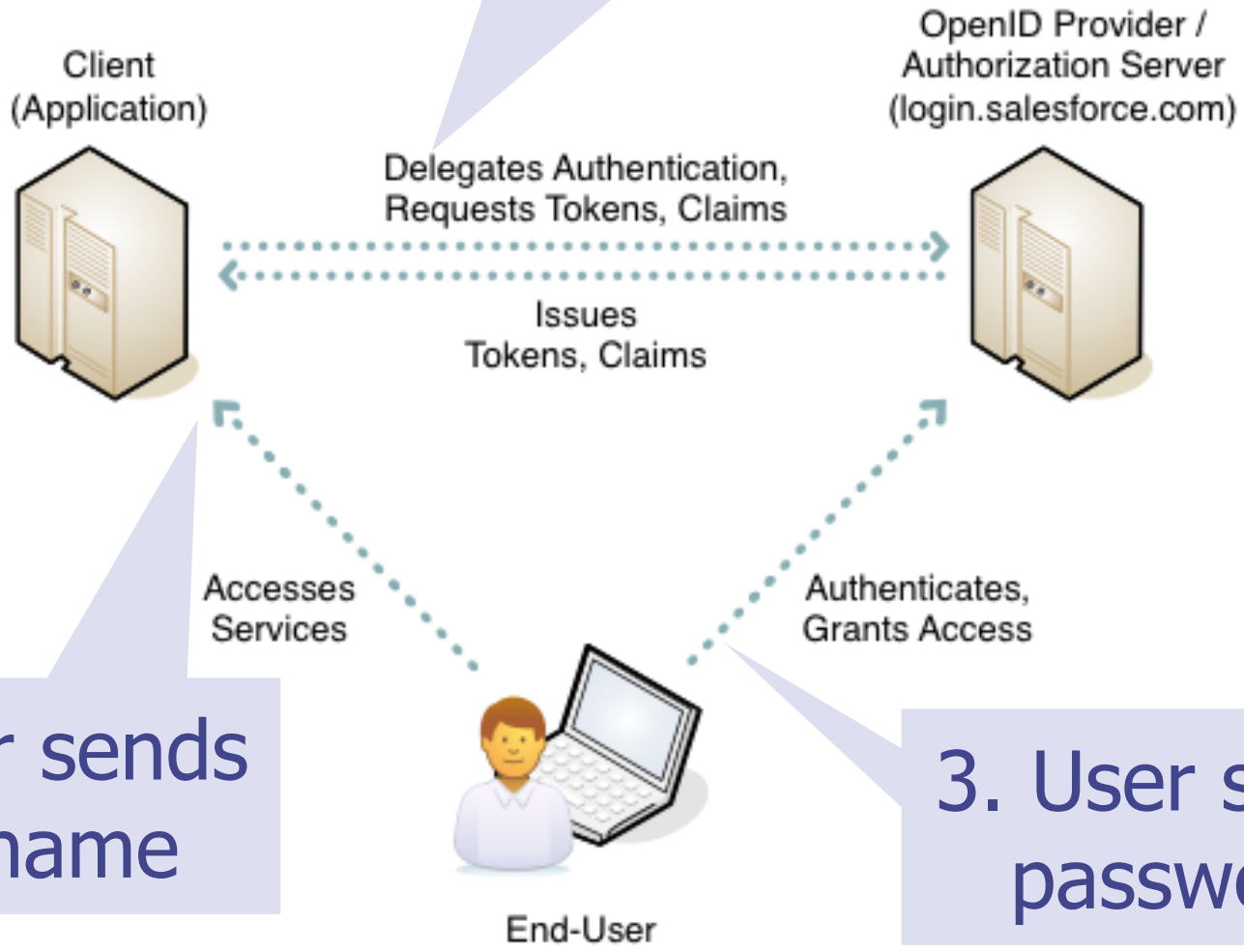
2. Username is forwarded



1. User sends username

OpenID

2. Username is forwarded

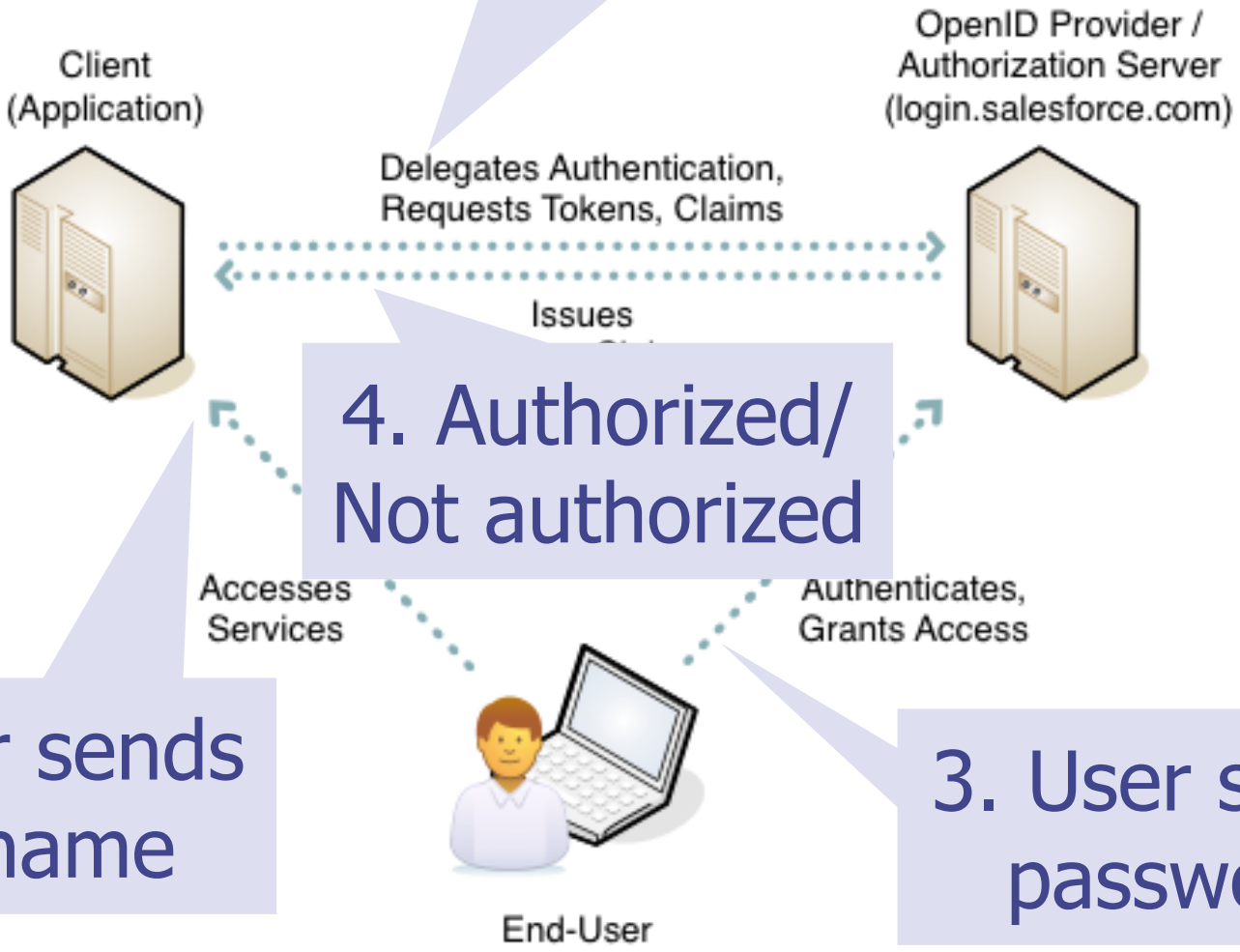


1. User sends username

3. User sends password

OpenID

2. Username is forwarded



4. Authorized/ Not authorized

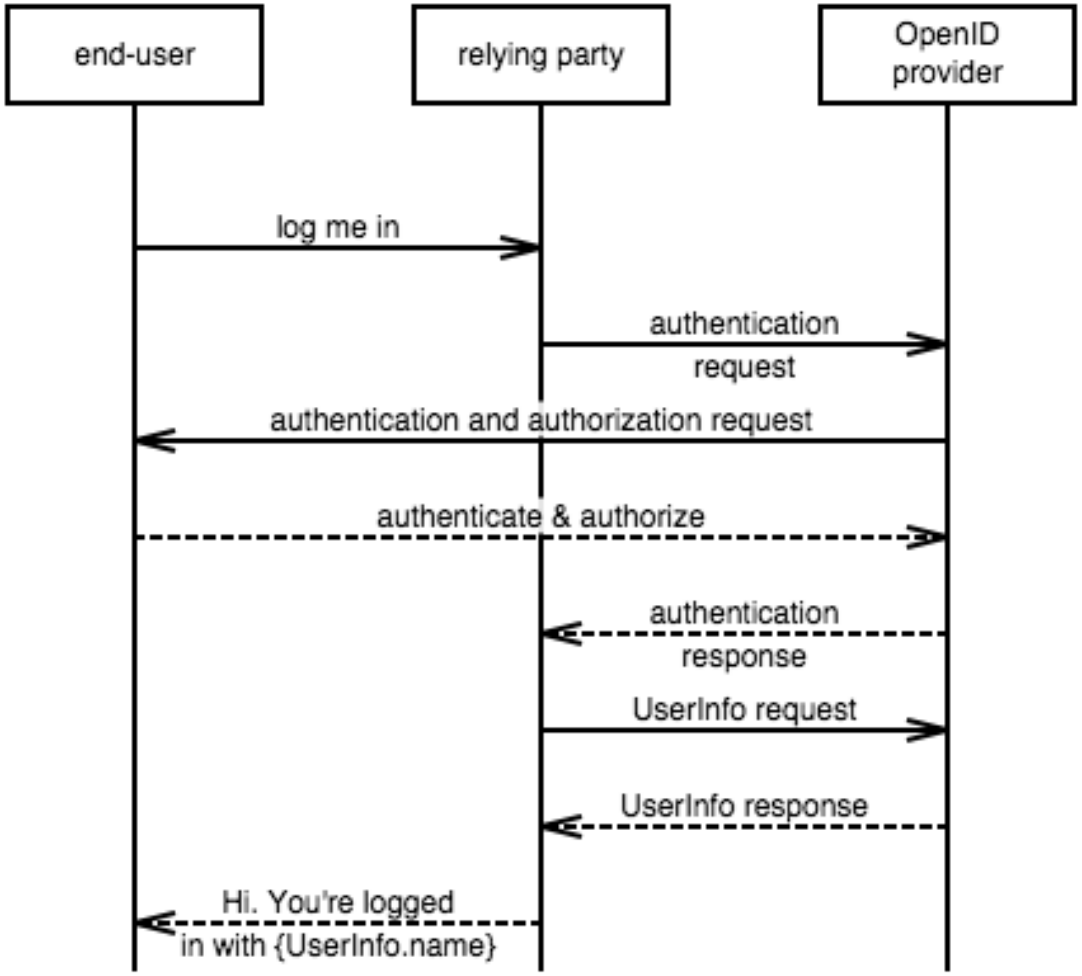
1. User sends username

3. User sends password

How to model choreographies?

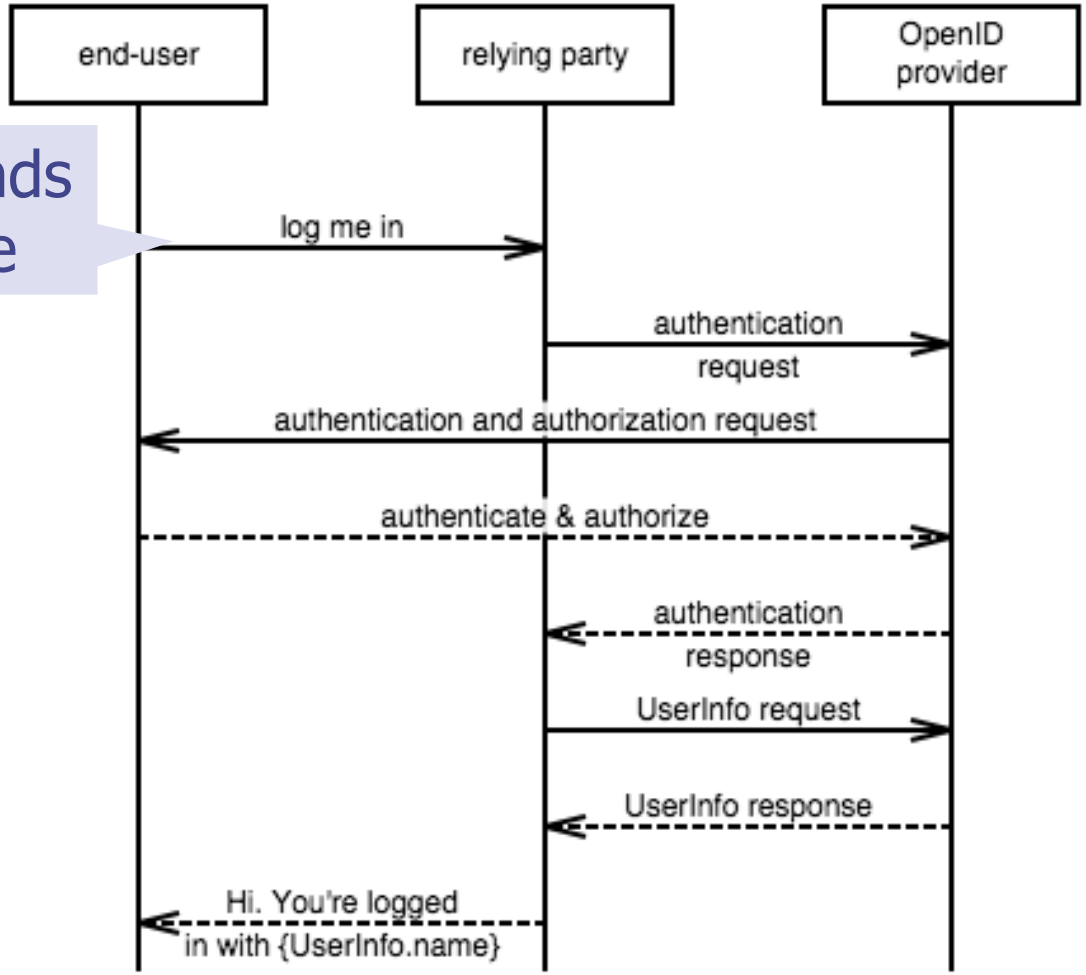
- Several approaches are usually adopted to model (and program) **choreographies**
  - MSC (Message Sequence Charts)
  - Cryptographic protocol notation
  - Open workflow nets
  - Collaboration diagrams
  - Process calculi-like languages
  - ...

Message Sequence Chart for OpenID

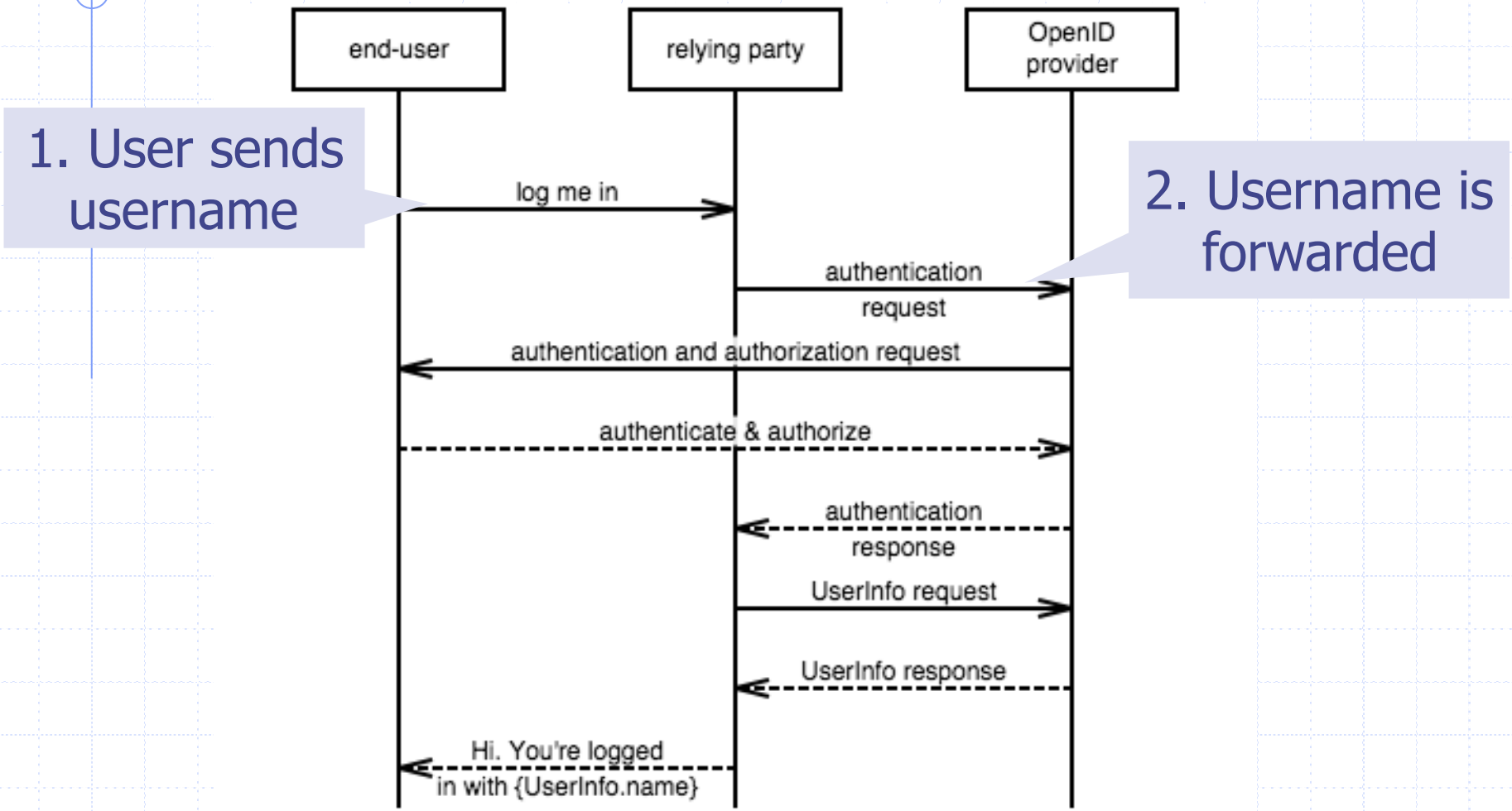


Message Sequence Chart for OpenID

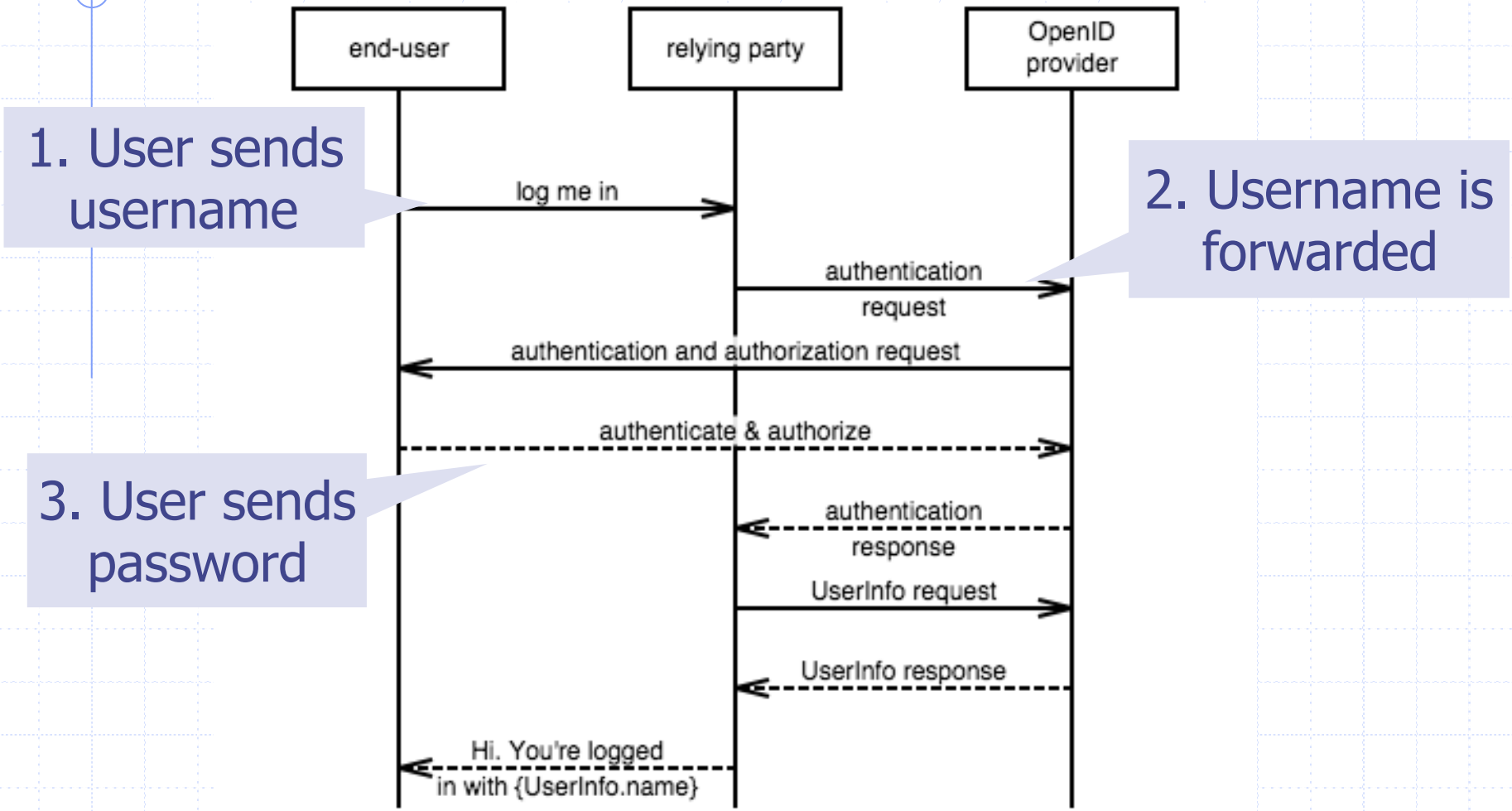
1. User sends username



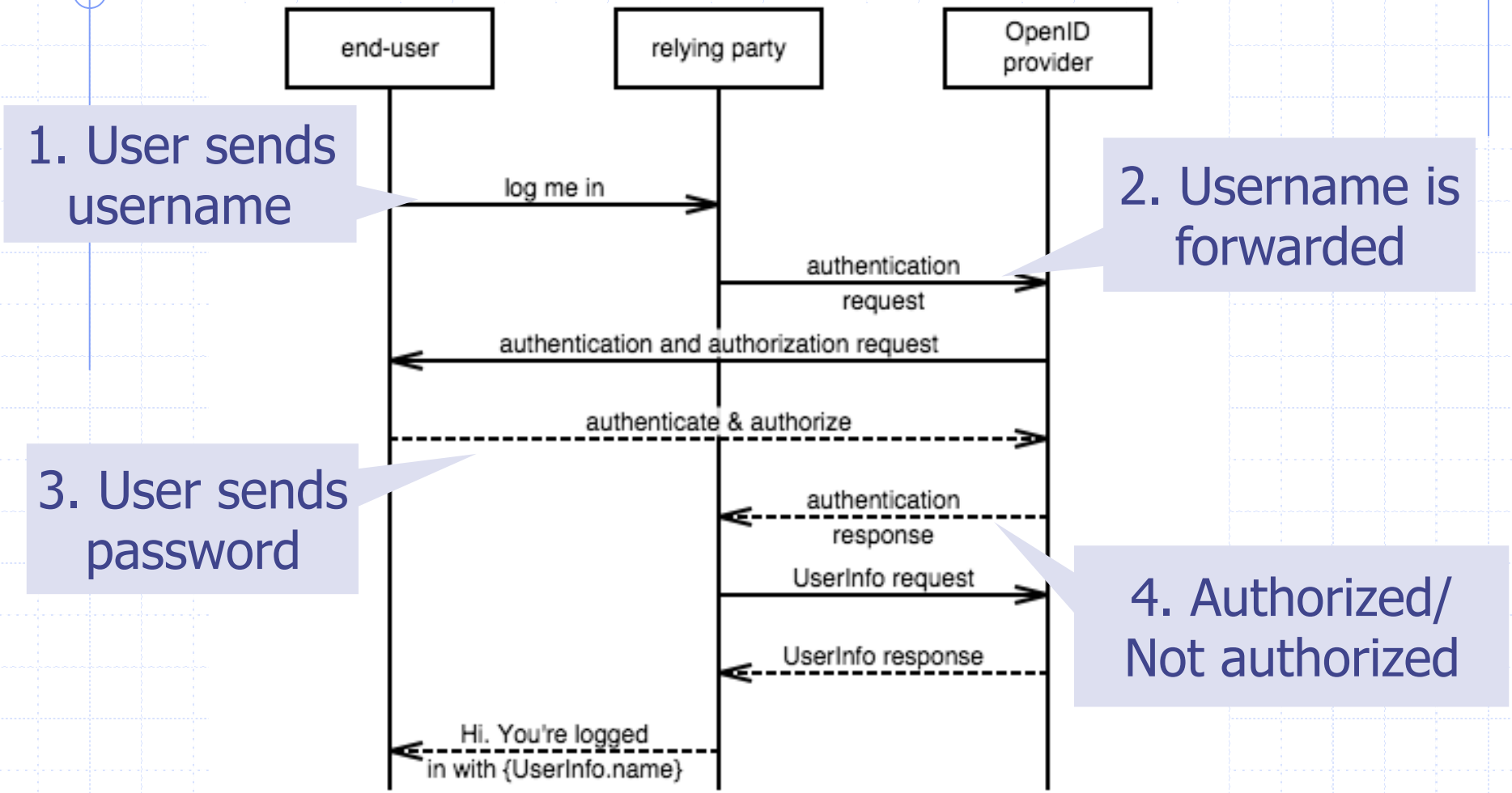
Message Sequence Chart for OpenID



Message Sequence Chart for OpenID



Message Sequence Chart for OpenID



Chryptographic protocol notation

- (Part of the) Needham-Schroeder **authentication** with public key:

$$A \rightarrow B: \{n_A, A\}_{kB}$$

$$B \rightarrow A: \{n_A, n_B\}_{kA}$$

$$A \rightarrow B: \{n_B\}_{kB}$$

Chryptographic protocol notation

- A man-in-the-middle **attack** at the Needham-Schroeder protocol:

$$A \rightarrow I: \{n_A, A\}_{kI}$$
$$I \rightarrow B: \{n_A, A\}_{kB}$$
$$B \rightarrow I: \{n_A, n_B\}_{kA}$$
$$I \rightarrow A: \{n_A, n_B\}_{kA}$$
$$A \rightarrow I: \{n_B\}_{kI}$$
$$I \rightarrow B: \{n_B\}_{kB}$$

Chryptographic protocol notation

- The **patched** Needham-Schroeder-Lowe protocol:

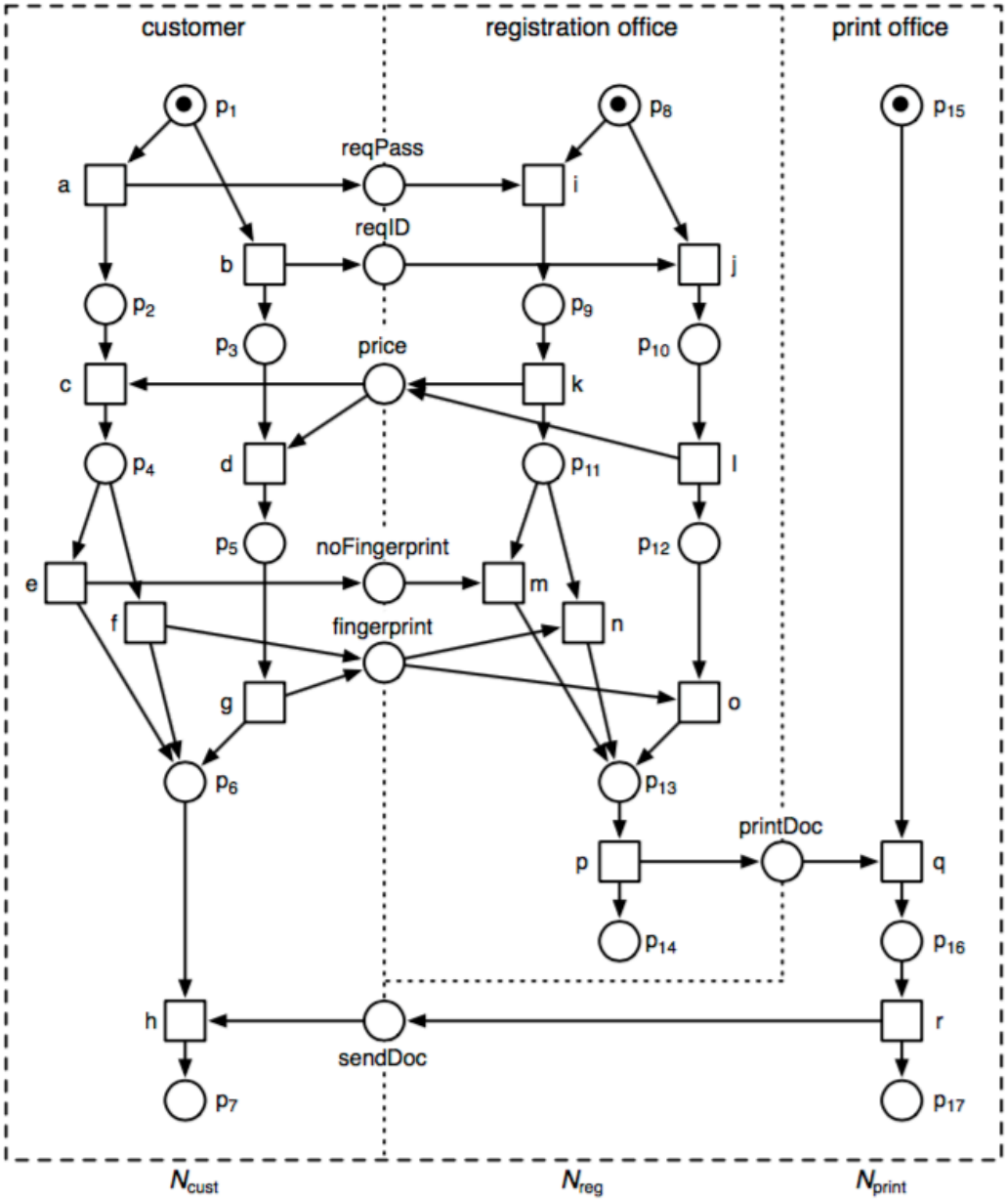
$$A \rightarrow B: \{n_A, A\}_{kB}$$

$$B \rightarrow A: \{n_A, n_B, B\}_{kA}$$

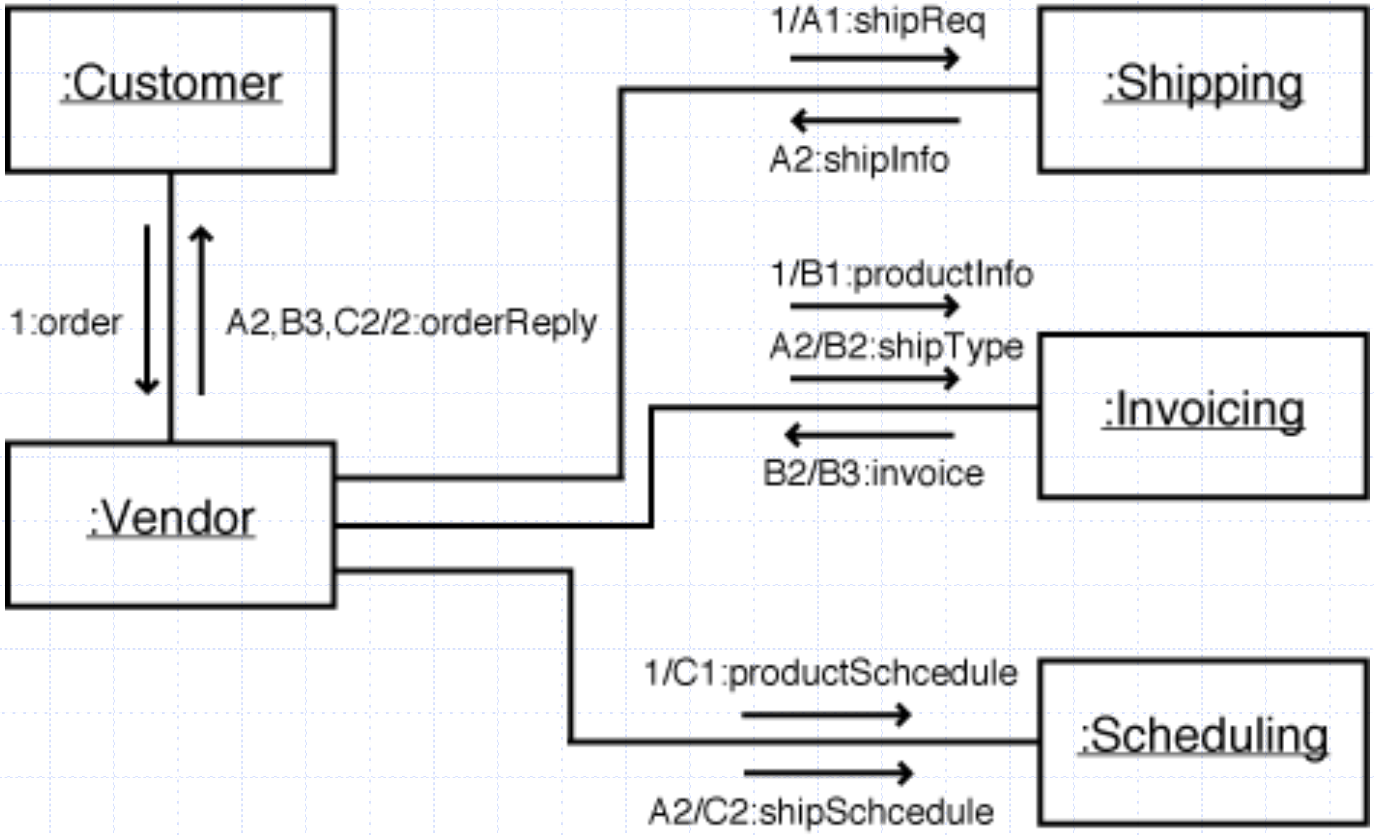
$$A \rightarrow B: \{n_B\}_{kB}$$

Open workflow nets

- Each partner is modeled by a **Petri net** with **input/output** places
  - The partners are combined by **merging** their input/output places

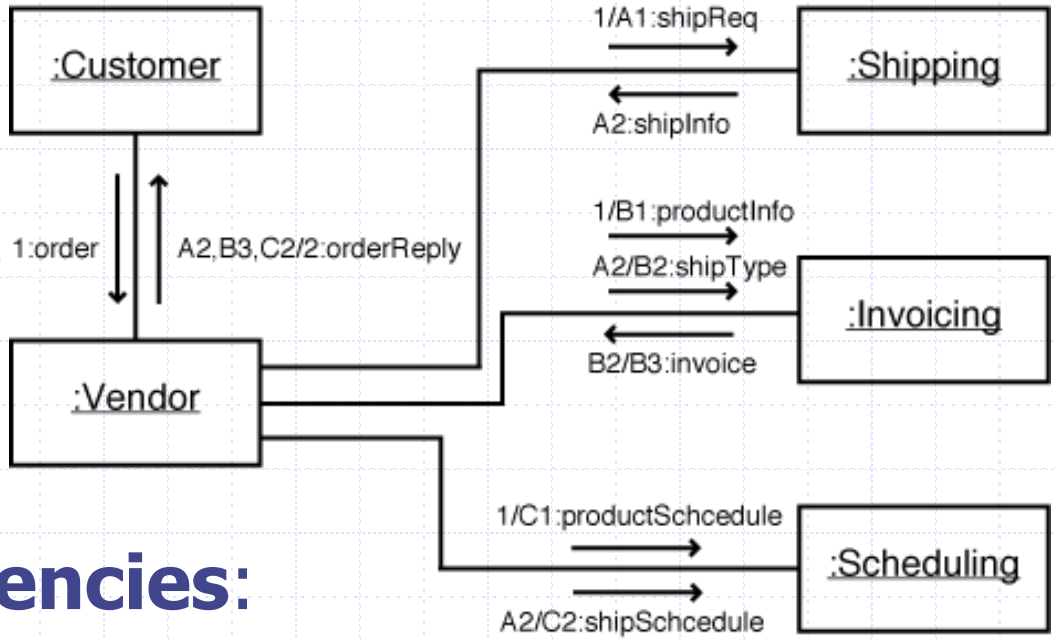


Collaboration diagram



Collaboration diagram

- Two kinds of **dependencies** among events



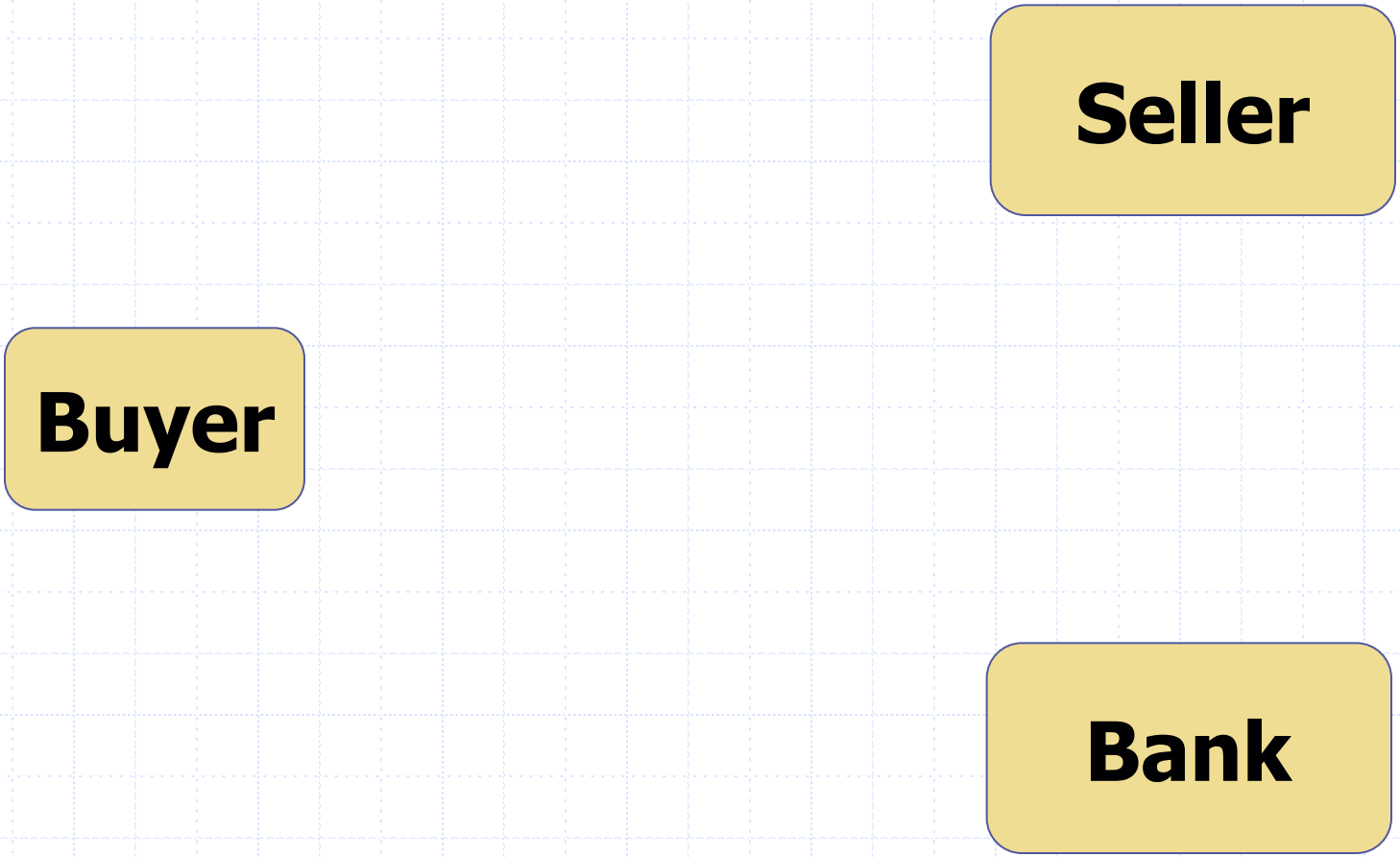
  - **Local dependencies**: total ordering of the events on the same channel (A2 follows A1)
  - **Causal dependencies**: A2/C2 indicates that C2 causally depends on A2

The process-calculus approach

- The process-calculus approach:
  - Define the basic **actions** …
  - … define operators to combine actions thus obtaining **processes**
- Approach **adopted** in the most popular choreography language in the context of Web Services:
  - **WS-CDL**: Web Service Choreography Description Language

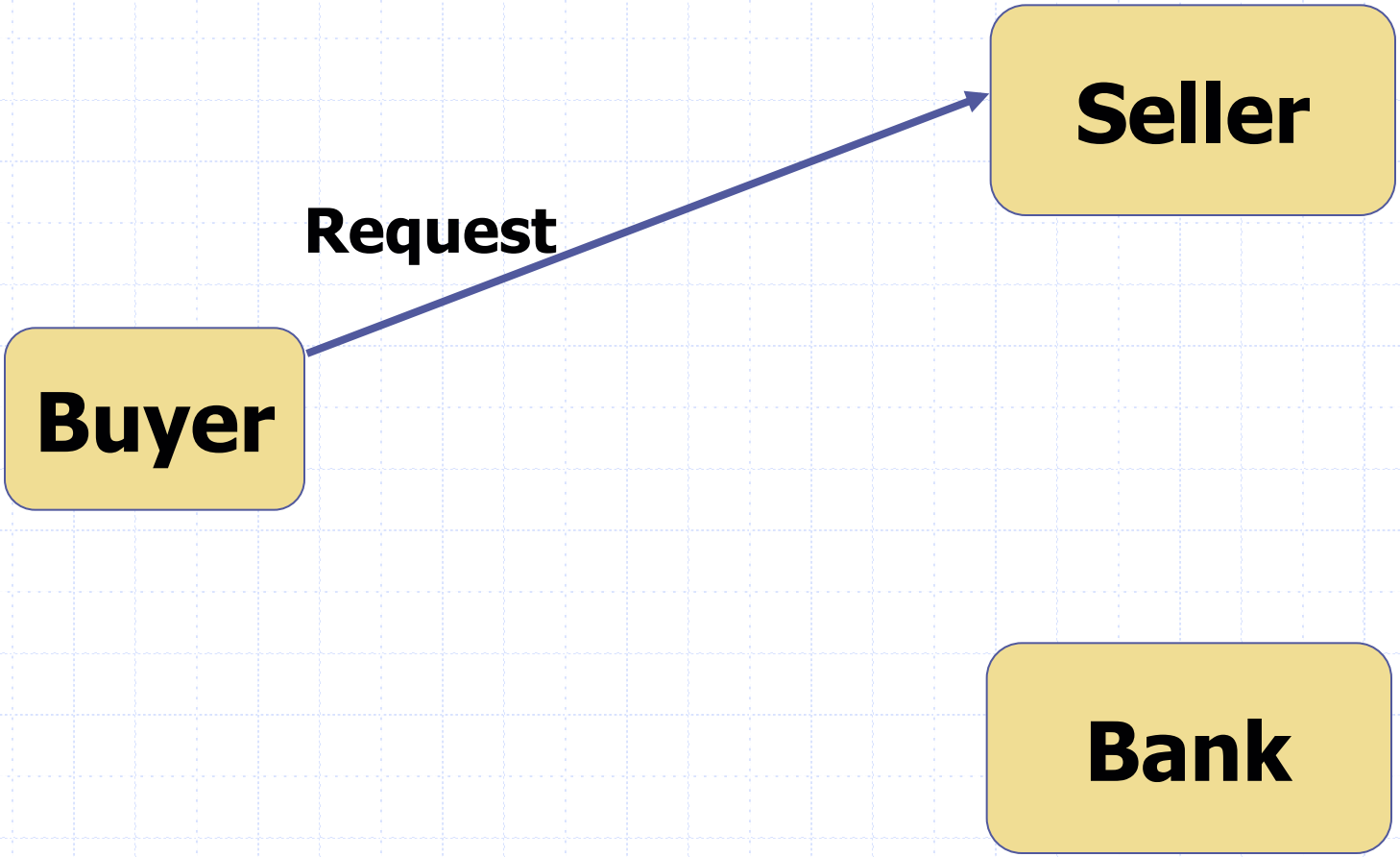
WS-CDL approach

- **Global view** of service interactions



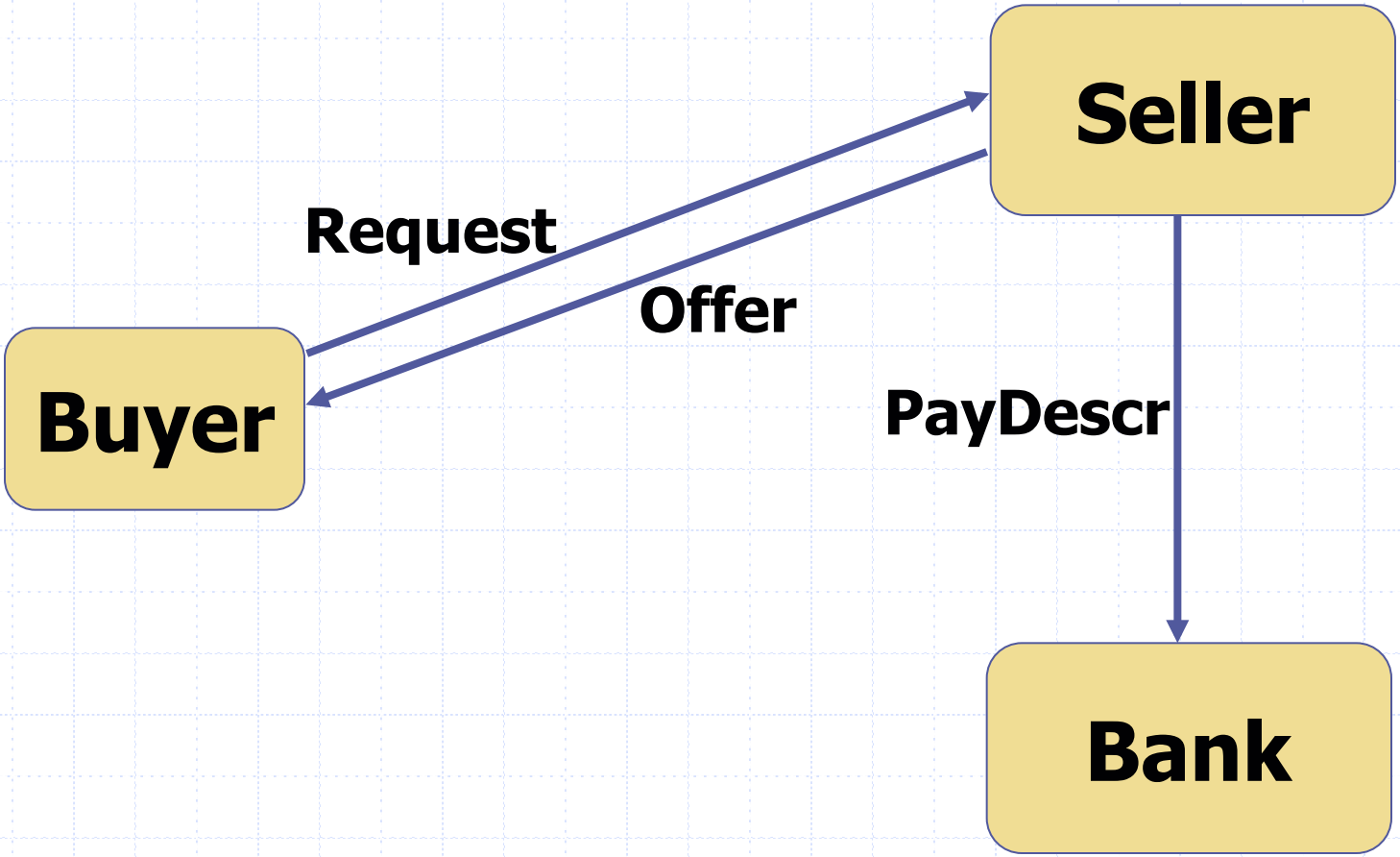
WS-CDL approach

- **Global view** of service interactions



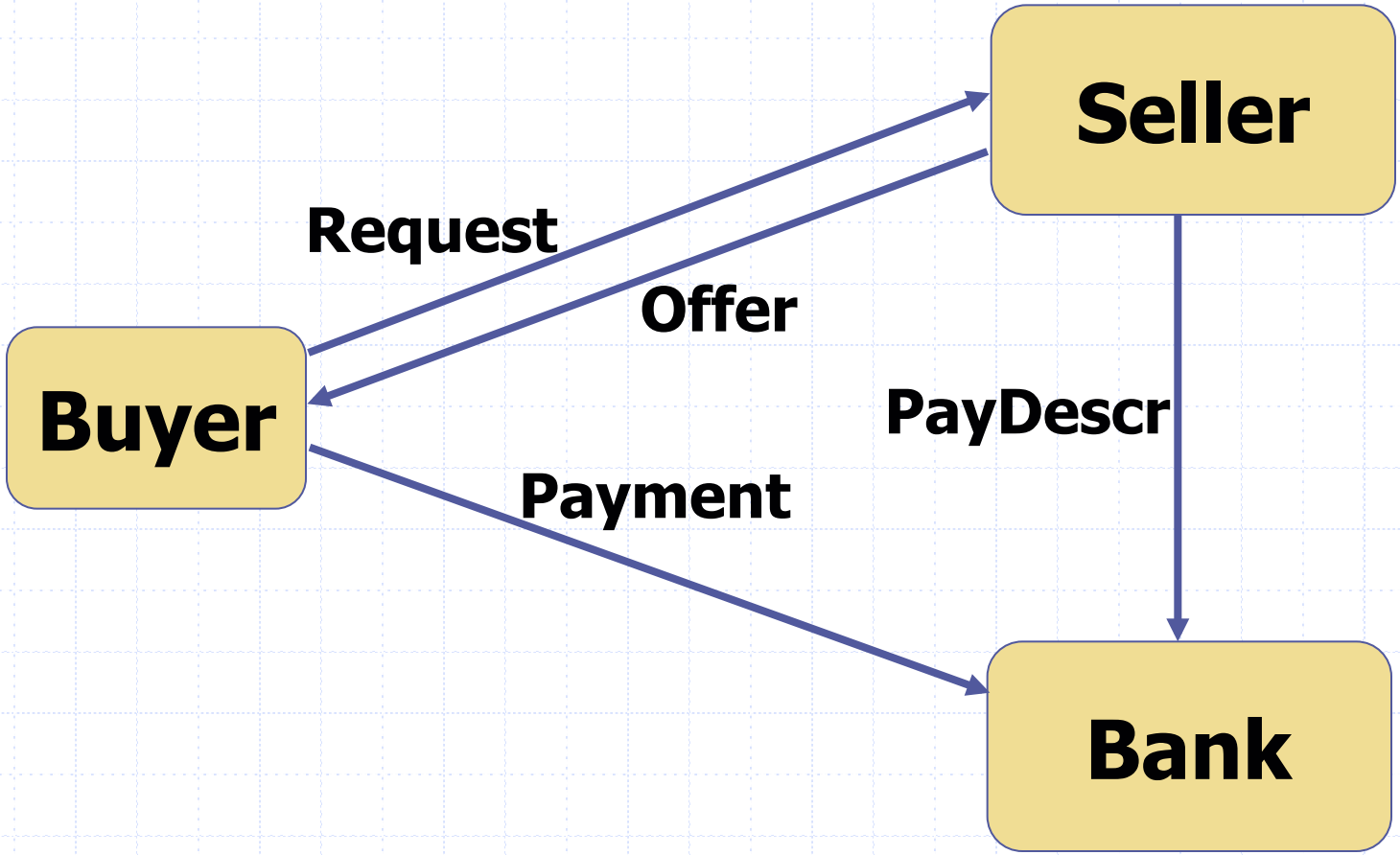
WS-CDL approach

- **Global view** of service interactions



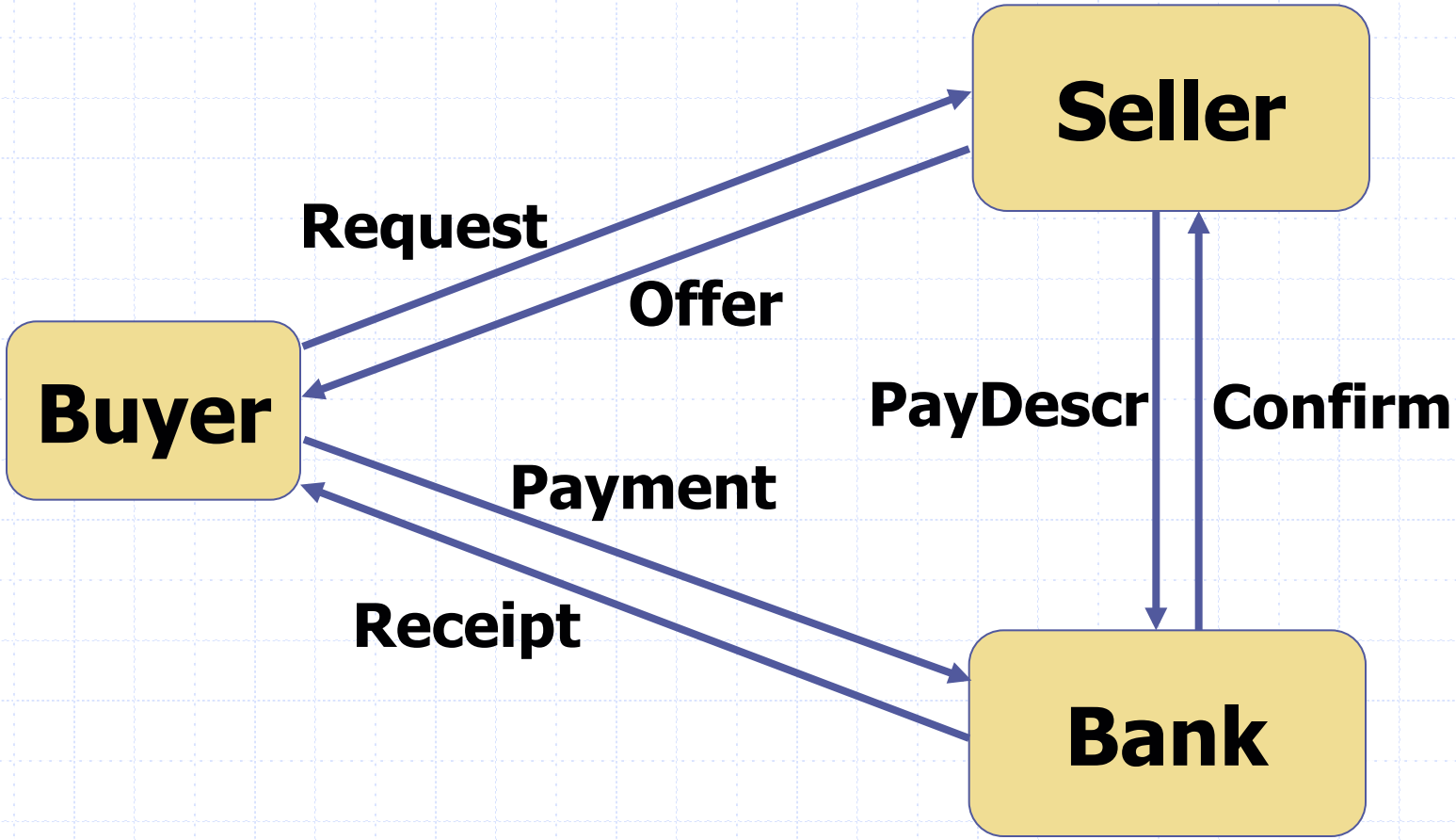
WS-CDL approach

- **Global view** of service interactions



WS-CDL approach

- **Global view** of service interactions



WS-CDL approach

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

WS-CDL approach

Basic Action
Buyer sends a message "Request" to Seller

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

WS-CDL approach

Basic Action
Buyer sends a message "Request" to Seller

Sequential composition

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ \mid \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ \mid \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

WS-CDL approach

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ \mid \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ \mid \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Basic Action
Buyer sends a message "Request" to Seller

Sequential composition

Parallel composition

Interaction Oriented Approach

- In the WS-CDL approach the basic **action** corresponds to
  - *a sender that sends a message on an operation exposed by a receiver*
  - this is the so-called **global view** approach
- Easy to formalise … we will see:

I. Lanese, C. Guidi, F. Montesi, G. Zavattaro: *Bridging the Gap between Interaction- and Process-Oriented Choreographies.* Proc. of SEFM 2008: 323-332

Choreography process calculus

- We have defined a choreography process calculus

$$H \quad ::= \quad a_{r \to s} \quad | \quad H + H \quad | \quad H; H \quad | \quad H | H \quad | \quad H^*$$

Choice

Parallel

Basic Action
r sends a message *a* to *s*

Sequential

Repetition

Choreography operational semantics

Ready to terminate

Final completion

$$a_{r\to s} \xrightarrow{a_{r\to s}} \mathbf{1} \qquad \mathbf{1} \xrightarrow{\surd} \mathbf{0} \qquad H^* \xrightarrow{\surd} \mathbf{0}$$

$$\frac{H \xrightarrow{\eta} H'}{H+L \xrightarrow{\eta} H'} \qquad \frac{H \xrightarrow{\eta} H' \quad \eta \neq \surd}{H;L \xrightarrow{\eta} H';L} \qquad \frac{H \xrightarrow{\surd} H' \quad L \xrightarrow{\eta} L'}{H;L \xrightarrow{\eta} L'}$$

$$\frac{H \xrightarrow{\surd} H' \quad L \xrightarrow{\surd} L'}{H|L \xrightarrow{\surd} H'|L'} \qquad \frac{H \xrightarrow{\eta} H' \quad \eta \neq \surd}{H|L \xrightarrow{\eta} H'|L} \qquad \frac{H \xrightarrow{\eta} H' \quad \eta \neq \surd}{H^* \xrightarrow{\eta} H';H^*}$$

A simple choreography (1)

- A **client** sends a reservation request to a **travel agency**, which contacts an **airplane company** and a **hotel**, and finally notify the client by confirming or cancelling the reservation:

$$
\begin{array}{l}
Reservation_{Client \rightarrow TravelAgency}; \\
(\ (Reserve_{TravelAgency \rightarrow AirCompany}; ConfirmFlight_{AirCompany \rightarrow TravelAgency}) | \\
\quad (Reserve_{TravelAgency \rightarrow Hotel}; ConfirmRoom_{Hotel \rightarrow TravelAgency})\ \ ); \\
Confirmation_{TravelAgency \rightarrow Client} + Cancellation_{TravelAgency \rightarrow Client}
\end{array}
$$

A simple choreography (2)

- A **client** negotiates with a **shop** the price for a product she initially indicates

$$
\begin{aligned}
&ProdID_{client \rightarrow shop};\\
&(Price_{shop \rightarrow client}; Offer_{client \rightarrow shop})^*;\\
&(Confirm_{shop \rightarrow client} + Decline_{shop \rightarrow client})
\end{aligned}
$$

Subtleties in choreographies

- Suppose the decision to **Decline** is taken by the client:

$$ProdID_{client \to shop};$$
$$(Price_{shop \to client}; Offer_{client \to shop})^*;$$
$$(Confirm_{shop \to client} + Decline_{client \to shop})$$

- Are there possible **problems** with this choreography?

Possible problems

$$ProdID_{client \to shop};$$
$$(Price_{shop \to client}; Offer_{client \to shop})^*;$$
$$(Confirm_{shop \to client} + Decline_{client \to shop})$$

- **Intuition**: problems could depend on the kind of communication
  - **Synchronous**: (no problem) client and shop synchronise on the branch to select
  - **Asynchronous**: (problem) client and shop could select different branches (by emitting inconsistent messages)

Choreography correctness

- We have performed a **formal** study of choreography correctness
  - A calculus for **choreography** (similar to the previous one, without repetition)
  - Canonical **projection**:
    - syntactic extraction from the choreography of the behaviour of the single roles
  - Check that the parallel composition of the projections **behaves** like the choreography

IOC: Interaction Oriented Choreography calculus

- We have defined an Interaction Oriented Choreography (**IOC**) calculus

$$\mathcal{I} ::= a \xrightarrow{o} b \mid \mathbf{1} \mid \mathbf{0} \mid \mathcal{I};\mathcal{I}' \mid \mathcal{I} \parallel \mathcal{I}' \mid \mathcal{I} + \mathcal{I}'$$

Sequence

Basic Action
a sends a message *o* to *b*

Parallel

Choice

IOC: Interaction Oriented Choreography calculus

- We have defined an Interaction Oriented Choreography (**IOC**) calculus

$$\mathcal{I} ::= a \xrightarrow{o} b \mid \mathbf{1} \mid \mathbf{0} \mid \mathcal{I};\mathcal{I}' \mid \mathcal{I} \parallel \mathcal{I}' \mid \mathcal{I} + \mathcal{I}'$$

Success

Failure

IOC: operational semantics

(Interaction)

$$a \xrightarrow{o} b \xrightarrow{a \xrightarrow{o} b} \mathbf{1}$$

(Sequence)

$$\frac{\mathcal{I} \xrightarrow{\sigma} \mathcal{I}' \quad \sigma \neq \surd}{\mathcal{I}; \mathcal{J} \xrightarrow{\sigma} \mathcal{I}'; \mathcal{J}}$$

(End)

$$\mathbf{1} \xrightarrow{\surd} \mathbf{0}$$

(Parallel)

$$\frac{\mathcal{I} \xrightarrow{\sigma} \mathcal{I}' \quad \sigma \neq \surd}{\mathcal{I} \parallel \mathcal{J} \xrightarrow{\sigma} \mathcal{I}' \parallel \mathcal{J}}$$

(Choice)

$$\frac{\mathcal{I} \xrightarrow{\sigma} \mathcal{I}'}{\mathcal{I} + \mathcal{J} \xrightarrow{\sigma} \mathcal{I}'}$$

(Seq-end)

$$\frac{\mathcal{I} \xrightarrow{\surd} \mathcal{I}' \quad \mathcal{J} \xrightarrow{\sigma} \mathcal{J}'}{\mathcal{I}; \mathcal{J} \xrightarrow{\sigma} \mathcal{J}'}$$

(Par-end)

$$\frac{\mathcal{I} \xrightarrow{\surd} \mathcal{I}' \quad \mathcal{J} \xrightarrow{\surd} \mathcal{J}'}{\mathcal{I} \parallel \mathcal{J} \xrightarrow{\surd} \mathcal{I}' \parallel \mathcal{J}'}$$

POC: Process Oriented Choreography calculus

- We have defined a Process Oriented Choreography (**POC**) calculus

Receive **Send**

Processes

$$P \quad ::= \quad o \mid \overline{o} \mid \mathbf{1} \mid \mathbf{0} \mid P; P' \mid \; P \mid P' \; \mid P + P'$$

$$\mathcal{S} \quad ::= \quad (P)_a \mid \mathcal{S} \parallel \mathcal{S}'$$

Choreography

Process *P* **playing role** *a*

POC: operational semantics -Processes-

(In)

$$o \xrightarrow{o} \mathbf{1}$$

(Sync-Out)

$$\overline{o} \xrightarrow{\langle o \rangle} \mathbf{1}$$

(One)

$$\mathbf{1} \xrightarrow{\surd} \mathbf{0}$$

(Sequence)

$$\frac{P \xrightarrow{\gamma} P' \quad \gamma \neq \surd}{P;Q \xrightarrow{\gamma} P';Q}$$

(Inner Parallel)

$$\frac{P \xrightarrow{\gamma} P' \quad \gamma \neq \surd}{P \mid Q \xrightarrow{\gamma} P' \mid Q}$$

(Choice)

$$\frac{P \xrightarrow{\gamma} P'}{P + Q \xrightarrow{\gamma} P'}$$

(Seq-end)

$$\frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\gamma} Q'}{P;Q \xrightarrow{\gamma} Q'}$$

(Inner Par-end)

$$\frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \mid Q \xrightarrow{\surd} P' \mid Q'}$$

POC: operational semantics -Choreographies-

(INNER)

$$\frac{P \xrightarrow{\gamma} P' \quad \gamma \neq \overline{o}, \surd}{(P)_a \xrightarrow{\gamma:a} (P')_a}$$

(SYNCHRO)

$$\frac{\mathcal{S} \xrightarrow{\langle o \rangle:a} \mathcal{S}' \quad \mathcal{S}'' \xrightarrow{o:b} \mathcal{S}'''}{\mathcal{S} \parallel \mathcal{S}'' \xrightarrow{a \xrightarrow{o} b} \mathcal{S}' \parallel \mathcal{S}'''}$$

(EXT PARALLEL)

$$\frac{\mathcal{S} \xrightarrow{\gamma} \mathcal{S}'}{\mathcal{S} \parallel \mathcal{S}'' \xrightarrow{\gamma} \mathcal{S}' \parallel \mathcal{S}''}$$

POC: the client-shop example

- Here is the behaviour of the **client** and the **shop** in the first of the two already considered choreographies:

$$\left(\overline{ProdID}; (Price; \overline{Offer})^*; (\overline{Confirm} + \overline{Decline}) \right)_{client} \| \left(ProdID; (\overline{Price}; Offer)^*; (\overline{Confirm} + \overline{Decline}) \right)_{shop}$$

POC: the client-shop example

- Here is the alternative version where the client sends the **decline** message:

$$\left(\overline{ProdID}; (Price; \overline{Offer})^*; (Confirm + \overline{Decline}) \right)_{client} \| \left(ProdID; (\overline{Price}; Offer)^*; (\overline{Confirm} + Decline) \right)_{shop}$$

- **No problem**: there are three possible choices (*Price*, *Confirm*, and *Decline*) and the client / shop **synchronize** in taking this decision!

Formal correspondence

- The IOC and the POC:

$$
\begin{array}{l}
ProdID_{client \to shop}; \\
(Price_{shop \to client}; Offer_{client \to shop})^*; \\
(Confirm_{shop \to client} + Decline_{client \to shop})
\end{array}
$$

$$
\begin{array}{l}
(\ \overline{ProdID}; (Price; \overline{Offer})^*; (\overline{Confirm} + \overline{Decline})\ )_{client} || \\
(\ ProdID; (\overline{Price}; Offer)^*; (\overline{Confirm} + Decline)\ )_{shop}
\end{array}
$$

have exactly the same **traces** of $a \xrightarrow{o} b$ transitions in the operational semantics

Canonical projection

- In the previous example, the behaviour of the roles in the POC is obtained from the IOC by means of **projection**:

$$
\begin{array}{rcl}
\mathrm{proj}(a \xrightarrow{o} b, a) & = & \overline{o} \qquad \mathrm{proj}(a \xrightarrow{o} b, b) \;=\; o \\
\mathrm{proj}(a \xrightarrow{o} b, c) & = & \mathbf{1} \;\; \mathit{if}\; c \neq a, b \\
\mathrm{proj}(\mathbf{1}, a) & = & \mathbf{1} \qquad \mathrm{proj}(\mathbf{0}, a) \;=\; \mathbf{0} \\
\mathrm{proj}(\mathcal{I}; \mathcal{I}', a) & = & \mathrm{proj}(\mathcal{I}, a); \mathrm{proj}(\mathcal{I}', a) \\
\mathrm{proj}(\mathcal{I} \parallel \mathcal{I}', a) & = & \mathrm{proj}(\mathcal{I}, a) \mid \mathrm{proj}(\mathcal{I}', a) \\
\mathrm{proj}(\mathcal{I} + \mathcal{I}', a) & = & \mathrm{proj}(\mathcal{I}, a) + \mathrm{proj}(\mathcal{I}', a)
\end{array}
$$

The OpenID Example

- The OpenId **IOC**:

$$Username_{user \rightarrow relyingParty};$$
$$Username_{relyingParty \rightarrow IdProvider};$$
$$Password_{user \rightarrow IdProvider};$$
$$(Ok_{IdProvider \rightarrow relyingParty} + Fail_{IdProvider \rightarrow relyingParty})$$

and the projected **POC**:

$$(\overline{Username}; \overline{Password})_{user} \;||$$
$$(Username; \overline{Username}; (Ok + Fail))_{relyingParty} \;||$$
$$(Username; Password; (\overline{Ok} + \overline{Fail}))_{IdProvider}$$

- No Problem! Same **traces**!

An alternative end of OpenID

- Consider the following **alternative** end:

$$
\begin{aligned}
& Password_{user \to IdProvider}; \\
& \quad (\; Ok_{IdProvider \to relyingParty}; Notify_{IdProvider \to user}; \\
& \quad\quad Connect_{user \to relyingParty} \;) \\
& +(\; Fail_{IdProvider \to relyingParty}; Notify_{IdProvider \to user} \;)
\end{aligned}
$$

and its **projection**:

$$
\begin{aligned}
& (\; \overline{Password}; (Notify; \overline{Connect}) + (Notify) \;)_{user} \; || \\
& (\; (Ok; Connect) + Fail \;)_{relyingParty} \; || \\
& (\; Password; (\overline{Ok}; \overline{Notify}) + (\overline{Fail}; \overline{Notify}) \;)_{IdProvider}
\end{aligned}
$$

An alternative end of OpenID

- Consider the following **alternative** end:

$$
\begin{aligned}
& Password_{user \to IdProvider}; \\
& \quad (\; Ok_{IdProvider \to relyingParty}; Notify_{IdProvider \to user}; \\
& \quad\quad Connect_{user \to relyingParty} \;) \\
& +(\; Fail_{IdProvider \to relyingParty}; Notify_{IdProvider \to user} \;)
\end{aligned}
$$

and its **projection**:

$$
\begin{aligned}
& (\; \overline{Password}; (Notify; \overline{Connect}) + (Notify) \;)_{user} \; || \\
& (\; (Ok; Connect) + Fail \;)_{relyingParty} \; || \\
& (\; Password; (\overline{Ok}; \overline{Notify}) + (\overline{Fail}; \overline{Notify}) \;)_{IdProvider}
\end{aligned}
$$

Choreography correctness

- Is it possible to check the **correctness** of a choreography?
  - **Yes**: by controlling three independent conditions
    - Connectedness for **sequence**
    - Unique point of **choice**
    - **Causality** safety

Connectedness for sequence

- Intuition:
  - Given the choreography *C;D* the rhs *D* should **start** only when the lhs *C* has **completed**
  - We check this condition by controlling that among the roles involved in the **completion** of *C* there is at least one role involved in the **starting** of *D*

Initial and final transitions

- The functions *transI* and *transF* return the interactions in the **starting** and **completion** of a choreography

$$\mathrm{transI}(a \xrightarrow{o} b) = \mathrm{transF}(a \xrightarrow{o} b) = \{a \xrightarrow{o} b\}$$
$$\mathrm{transI}(\mathbf{1}) = \mathrm{transI}(\mathbf{0}) = \mathrm{transF}(\mathbf{1}) = \mathrm{transF}(\mathbf{0}) = \emptyset$$
$$\mathrm{transI}(\mathcal{I} \parallel \mathcal{I}') = \mathrm{transI}(\mathcal{I} + \mathcal{I}') = \mathrm{transI}(\mathcal{I}) \cup \mathrm{transI}(\mathcal{I}')$$
$$\mathrm{transF}(\mathcal{I} \parallel \mathcal{I}') = \mathrm{transF}(\mathcal{I}) \cup \mathrm{transF}(\mathcal{I}')$$
$$\mathrm{transF}(\mathcal{I} + \mathcal{I}') = \mathrm{transF}(\mathcal{I}) \cup \mathrm{transF}(\mathcal{I}')$$
$$\mathrm{transI}(\mathcal{I}; \mathcal{I}') = \mathrm{transI}(\mathcal{I}') \text{ if } \mathcal{I} \xrightarrow{\surd}, \ \mathrm{transI}(\mathcal{I}) \text{ otherwise}$$
$$\mathrm{transF}(\mathcal{I}; \mathcal{I}') = \mathrm{transF}(\mathcal{I}) \text{ if } \mathcal{I}' \xrightarrow{\surd}, \ \mathrm{transF}(\mathcal{I}') \text{ otherwise}$$

Connectedness for sequence: formally

Definition 4.2 (Synchronous connectedness for sequence). *An IOC $\mathcal{I}$ is synchronous connected for sequence if for each subterm of the form $\mathcal{I};\mathcal{J}$ we have* $\forall a \xrightarrow{o} b \in \mathrm{transF}(\mathcal{I}), \forall c \xrightarrow{o'} d \in \mathrm{transI}(\mathcal{J}), \{a,b\} \cap \{c,d\} \neq \emptyset$.

- The last action before a sequential composition will surely **occur before** every possible intial action after
  - guaranteed by the involvement of at least **one role** in both actions

Connectedness for sequence

- Here is a trivial example of an **incorrect** choreography:

$$a \xrightarrow{o} b; c \xrightarrow{o'} d$$

having the following **projection**:

$$(\overline{o})_a \mid\mid (o)_b \mid\mid (\overline{o'})_c \mid\mid (o')_d$$

Unique point of choice

- Intuition:
  - Given the choreography *C+D* all the involved roles should **agree** on the branch to **select**:
    - When an initial action in one branch **occur**, all the initial actions in the other one should be **disabled**
    - The selected branch must be **communicated** to all partners, hence the partners involved in *C* and in *D* must be the same

Unique point of choice: formally

Definition 4.3 (Synchronous unique point of choice). *An IOC $\mathcal{I}$ has synchronous unique points of choice if for each subterm of the form $\mathcal{I} + \mathcal{J}$ we have $\forall a \xrightarrow{o} b \in \mathrm{transI}(\mathcal{I}), \forall c \xrightarrow{o'} d \in \mathrm{transI}(\mathcal{J}), \{a, b\} \cap \{c, d\} \neq \emptyset$. Furthermore* $\mathrm{roles}(\mathcal{I}) = \mathrm{roles}(\mathcal{J})$.

- Every pair of initial actions in the two branches must include a **common** role:
  - this role will be the **unique** point of choice
  - moreover, the selected branch must be communicated to **all involved** roles

Unique point of choice

- Here is a trivial example of an **incorrect** choreography:

$$(a \xrightarrow{o} b + a \xrightarrow{o'} c); a \xrightarrow{o} b$$

having the following **projection**:

$$\left((\overline{o} + \overline{o'}); \overline{o}\right)_a \parallel \left((o + \mathbf{1}); o\right)_b \parallel \left((\mathbf{1} + o'); \mathbf{1}\right)_c$$

Causality safety

- Intuition:
  - If an operation occurs **twice** in a choreography, the two interactions should be **not active contemporaneously**
  - Consider the following **choreography**:

$$(a \xrightarrow{o} b; b \xrightarrow{o'} c; c \xrightarrow{o} b) + (a \xrightarrow{o'} c; c \xrightarrow{o'} b)$$

with the **projection**:

$$\left(\overline{o} + \overline{o'}\right)_a \parallel \left((o; \overline{o'}; o) + o'\right)_b \parallel \left((o'; \overline{o}) + (o'; \overline{o'})\right)_c$$

Causality safety: formally

Definition 4.5 (Synchronous causality-safety). *An IOC is synchronous causality-safe iff for each pair of interactions i and j using the same operation, either* $s_i \leq_s r_j \wedge r_i \leq_s s_j$ *or* $s_j \leq_s r_i \wedge r_j \leq_s s_i$.

- The relation $\leq_s$ formalises **causal dependencies**
  - hence if $x \leq_s y$ the two events x and y cannot be **both ready to be executed**

Causality safety: formally

sequentiality: *for each* $\mathcal{I};\mathcal{I}'$, *if* i *is an interaction in* $\mathcal{I}$, j *is an interaction in* $\mathcal{I}'$, *and* e_i *and* e_j *are events in the same role then* $e_i \leq_s e_j$;

synchronization: *for each* i, j *if* $e_i \leq_s e_j$ *then* $\overline{e}_i \leq_s e_j$.

- The relation $\leq_s$ formalises **causal dependencies**
  - hence if $x \leq_s y$ the two events x and y cannot be **both ready to be executed**

Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

$$
\begin{aligned}
& Password_{user \rightarrow IdProvider}; \\
& \quad (\; Ok_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user}; \\
& \quad\quad Connect_{user \rightarrow relyingParty} \;) \\
& + (\; Fail_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user} \;)
\end{aligned}
$$

Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

$$
\begin{aligned}
& Password_{user \rightarrow IdProvider}; \\
& \quad (\; Ok_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user}; \\
& \quad\quad Connect_{user \rightarrow relyingParty} \;) \\
& + (\; Fail_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user} \;)
\end{aligned}
$$

Connectedness for sequence: ?

Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

$$
\begin{aligned}
& Password_{user \rightarrow IdProvider}; \\
& \quad (\; Ok_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user}; \\
& \quad\quad Connect_{user \rightarrow relyingParty} \;) \\
& + (\; Fail_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user} \;)
\end{aligned}
$$

Connectedness for sequence: ✔

Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

$$
\begin{aligned}
& Password_{user \rightarrow IdProvider}; \\
& \quad (\; Ok_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user}; \\
& \quad\quad Connect_{user \rightarrow relyingParty} \;) \\
& + (\; Fail_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user} \;)
\end{aligned}
$$

Unique point of choice: ?

Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

$$
\begin{aligned}
& Password_{user \rightarrow IdProvider}; \\
& \quad (\; Ok_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user}; \\
& \quad\quad Connect_{user \rightarrow relyingParty} \;) \\
& + (\; Fail_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user} \;)
\end{aligned}
$$

Unique point of choice: ✔

Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

$$
\begin{aligned}
& Password_{user \rightarrow IdProvider}; \\
& \quad (\; Ok_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user}; \\
& \quad\quad Connect_{user \rightarrow relyingParty} \;) \\
& + (\; Fail_{IdProvider \rightarrow relyingParty}; Notify_{IdProvider \rightarrow user} \;)
\end{aligned}
$$

Causality safety: ?

Exercise

- Which condition is not satisfied in the **wrong** alternative OpenID end?

$$
\begin{aligned}
& Password_{user \rightarrow IdProvider}; \\
& \quad (\; Ok_{IdProvider \rightarrow relyingParty}; \; Notify_{IdProvider \rightarrow user}; \\
& \quad\quad Connect_{user \rightarrow relyingParty} \;) \\
& + (\; Fail_{IdProvider \rightarrow relyingParty}; \; Notify_{IdProvider \rightarrow user} \;)
\end{aligned}
$$

Causality safety: ✗

Let's go back to the client-shop example

$$
\begin{array}{l}
ProdID_{client \to shop}; \\
(Price_{shop \to client}; Offer_{client \to shop})^*; \\
(Confirm_{shop \to client} + Decline_{client \to shop})
\end{array}
$$

$$
\begin{array}{l}
(\ \overline{ProdID}; (Price; \overline{Offer})^*; (Confirm + \overline{Decline})\ )_{client} || \\
(\ ProdID; (\overline{Price}; Offer)^*; (\overline{Confirm} + Decline)\ )_{shop}
\end{array}
$$

- Remember: possible **problems** if the client sends *Decline* while the shop *Price*
  - This could happen in **asychronously** communicating systems

Asynchronous choreography correctness

- We have defined **asychronous communication** for POCs
  - We have defined the **correspondence** between IOCs and their asynchronous execution
    - More complex because **send** and **receive** events are **disjoint** in POCs
  - We have revisited the three **correctness** conditions that guarantee correspondence

POC: Asynchronous semantics

(In)

$$o \xrightarrow{o} \mathbf{1}$$

(Out)

$$\overline{o} \xrightarrow{\overline{o}} \mathbf{1}$$

(Async-Out)

$$\langle o \rangle \xrightarrow{\langle o \rangle} \mathbf{1}$$

(One)

$$\mathbf{1} \xrightarrow{\surd} \mathbf{0}$$

(Sequence)

$$\frac{P \xrightarrow{\gamma} P' \quad \gamma \neq \surd}{P; Q \xrightarrow{\gamma} P'; Q}$$

(Inner Parallel)

$$\frac{P \xrightarrow{\gamma} P' \quad \gamma \neq \surd}{P \mid Q \xrightarrow{\gamma} P' \mid Q}$$

(Choice)

$$\frac{P \xrightarrow{\gamma} P'}{P + Q \xrightarrow{\gamma} P'}$$

(Seq-end)

$$\frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\gamma} Q'}{P; Q \xrightarrow{\gamma} Q'}$$

(Inner Par-end)

$$\frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \mid Q \xrightarrow{\surd} P' \mid Q'}$$

POC: Asynchronous semantics

(INNER)

$$\frac{P \xrightarrow{\gamma} P' \quad \gamma \neq \overline{o}, \surd}{(P)_a \xrightarrow{\gamma:a} (P')_a}$$

(MSG)

$$\frac{P \xrightarrow{\overline{o}} P'}{(P)_a \xrightarrow{\overline{o}:a} (P' \mid \langle o \rangle)_a}$$

(SYNCHRO)

$$\frac{\mathcal{S} \xrightarrow{\langle o \rangle:a} \mathcal{S}' \quad \mathcal{S}'' \xrightarrow{o:b} \mathcal{S}'''}{\mathcal{S} \parallel \mathcal{S}'' \xrightarrow{a \xrightarrow{o} b} \mathcal{S}' \parallel \mathcal{S}'''}$$

(EXT PARALLEL)

$$\frac{\mathcal{S} \xrightarrow{\gamma} \mathcal{S}'}{\mathcal{S} \parallel \mathcal{S}'' \xrightarrow{\gamma} \mathcal{S}' \parallel \mathcal{S}''}$$

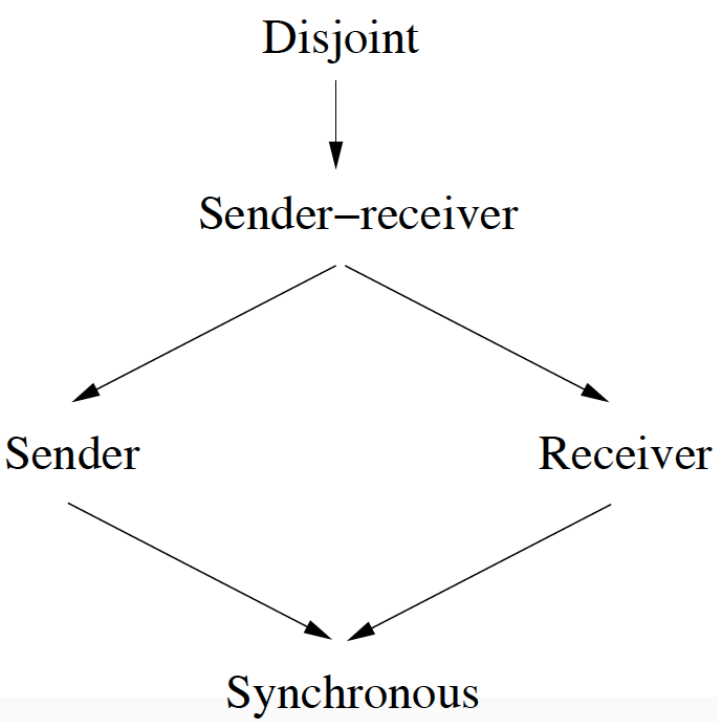
Correspondence relation

- At the level of POC the interaction coincides with the **receive event**
  - Hence, interaction correspondence guarantees correspondence of receive (**send** could not correspond)
  - See the choreography: $(a \xrightarrow{o} b; c \xrightarrow{o'} b)$

with projection: $(\overline{o})_a \mid\mid (o; o')_b \mid\mid (\overline{o'})_c$

Several correspondence relations

- The previous correspondence is named **receiver** correspondence
  - We have considered a **lattice** of correspondence relations
  - In the literature, receiver seems the **mainly considered** correspondence



Receiver connectedness for sequence

- There are **two ways** to preserve the receive events in a sequential composition *C* ;*D*
  - the **receiver** in *C* is the **emitter** in *D*

$$(a \xrightarrow{o} b; b \xrightarrow{o'} c)$$

  - the **receiver** in *C* is also the **receiver** in *D*

$$(a \xrightarrow{o} b; c \xrightarrow{o'} b)$$

Receiver connectedness for sequence

Definition 5.7 (Receiver connectedness for sequence). *An IOC $\mathcal{I}$ is receiver connected for sequence if for each subterm of the form $\mathcal{I};\mathcal{J}$ we have* $\forall a \xrightarrow{o} b \in \mathrm{transF}(\mathcal{I}), \forall c \xrightarrow{o'} d \in \mathrm{transI}(\mathcal{J}).b = c \vee b = d.$

- the **receiver** in *C* is the **emitter** in *D*

$$(a \xrightarrow{o} b; b \xrightarrow{o'} c)$$

- the **receiver** in *C* is also the **receiver** in *D*

$$(a \xrightarrow{o} b; c \xrightarrow{o'} b)$$

Asynchronous unique point of choice

Definition 5.1 (Asynchronous unique point of choice). *An IOC $\mathcal{I}$ has asynchronous unique points of choice if for each subterm of the form $\mathcal{I} + \mathcal{J}$ we have $\forall a \xrightarrow{o} b \in \mathrm{transI}(\mathcal{I}), \forall c \xrightarrow{o'} d \in \mathrm{transI}(\mathcal{J}).a = c$. Furthermore* $\mathrm{roles}(\mathcal{I}) = \mathrm{roles}(\mathcal{J})$.

- In an asynchronous setting, the **emitter** can locally decide to send a message
  - In a choice only the emitter **decide** a branch

Asynchronous causality safety

Definition 4.5 (Synchronous causality-safety). *An IOC is synchronous causality-safe iff for each pair of interactions i and j using the same operation, either $s_i \leq_s r_j \wedge r_i \leq_s s_j$ or $s_j \leq_s r_i \wedge r_j \leq_s s_i$.*

- Same definition, but with a relation $\leq_a$ that formalises **causal dependencies** for the asynchronous semantics

Asynchronous causality safety

sequentiality: *for each* $\mathcal{I};\mathcal{I}'$*, if* i *is an interaction in* $\mathcal{I}$*,* j *is an interaction in* $\mathcal{I}'$*, and* r_i *and* e_j *are respectively a receive and a generic event in the same role then* $r_i \leq_a e_j$*;*

synchronization: *for each* i*,* j *if* $r_i \leq_a e_j$ *then* $s_i \leq_a e_j$ *(here* s_i *can be an output or a message).*

- Same definition, but with a relation $\leq_a$ that formalises **causal dependencies** for the asynchronous semantics

Exercise: the initial WS-CDL example

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Exercise: the initial WS-CDL example

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Asynchronous unique point of choice: ?

Exercise: the initial WS-CDL example

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Trivial

Asynchronous unique point of choice: ✔

Exercise: the initial WS-CDL example

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Asynchronous causality safety: ?

Exercise: the initial WS-CDL example

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Trivial

Asynchronous causality safety: ✔

Exercise: the initial WS-CDL example

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Asychronous connectedness: ?

Exercise: the initial WS-CDL example

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&\mathrm{Payment}_{\mathrm{Buyer}\rightarrow\mathrm{Bank}}\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Asychronous connectedness: ✔

Exercise: try to remove the *Payment*

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Exercise: try to remove the *Payment*

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Trivial

Asynchronous unique point of choice: ✔

Exercise: try to remove the *Payment*

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ \mid \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ \mid \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Trivial

Asynchronous causality safety: ✔

Exercise: try to remove the *Payment*

$$
\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ | \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ | \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}
$$

Asychronous connectedness: ?

Exercise: try to remove the *Payment*

$$\begin{aligned}
&\mathrm{Request}_{\mathrm{Buyer}\rightarrow\mathrm{Seller}}\ ; \\
&(\ \mathrm{Offer}_{\mathrm{Seller}\rightarrow\mathrm{Buyer}}\ \mid \\
&\quad \mathrm{PayDescr}_{\mathrm{Seller}\rightarrow\mathrm{Bank}}\ )\ ; \\
&(\ \mathrm{Confirm}_{\mathrm{Bank}\rightarrow\mathrm{Seller}}\ \mid \\
&\quad \mathrm{Receipt}_{\mathrm{Bank}\rightarrow\mathrm{Buyer}}\ )
\end{aligned}$$

In an asynchronous setting there is no ordering guarantee (while it is guaranteed for synchronous communication!)

Asychronous connectedness: ✗

What to do with choreographies ?

- You are now among the greatest worldwide experts in **choreographies**!
  - so what? …
- Don't worry… you didn't lose your time
  - There is at least one **programming language** that you can use to realise your distributed systems using choreographies

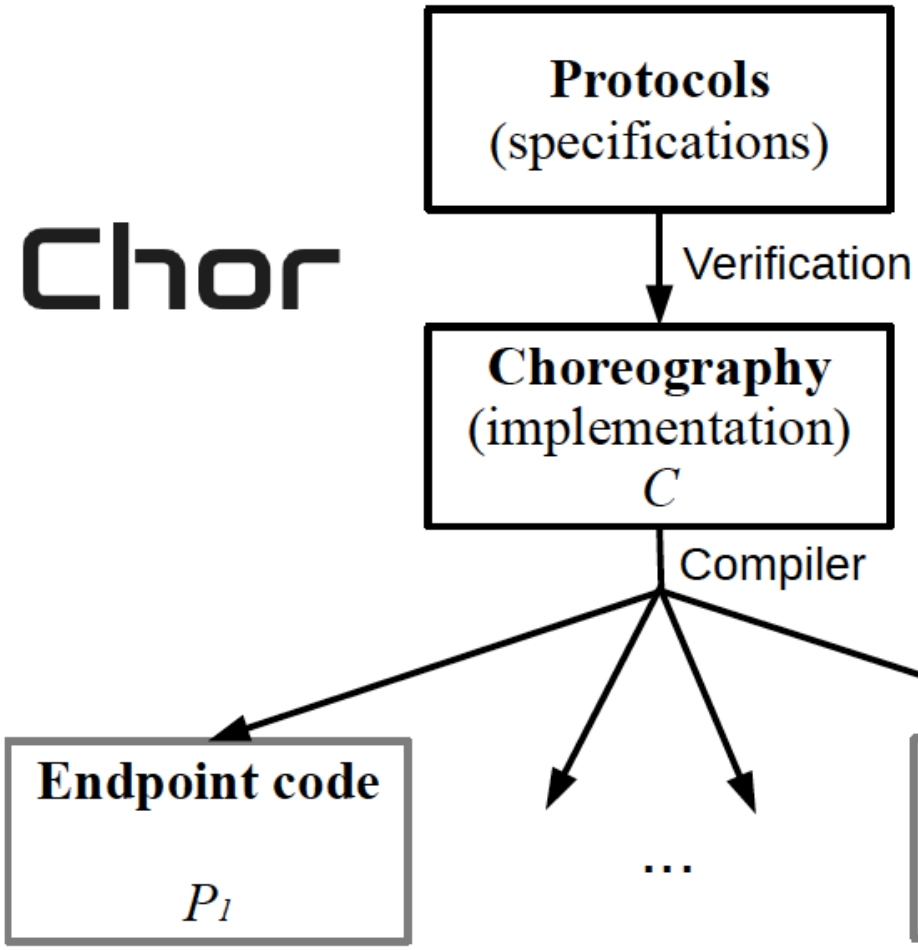
Marco Carbone, Fabrizio Montesi: *Deadlock-freedom-by-design: multiparty asynchronous global programming*. Proc. of POPL 2013: 263-274

Chor

Choreography programming

- A IOC-like language is used to specify the possible communication **protocols** used in the system
- An **enriched** choreography language (variables, if-then-else, functions, ..) is used to specify the overall system
- **End-point** programs are obtained by means of projection (written in Jolie)

The approach: graphically



Chor

Jolie

DEMO

Chor: basic ideas

- Describe the behaviour of processes in a **system**:
  - Each process has a **local state**
  - Processes take part to **multiparty conversations**, tracked as sessions
  - **Sessions** are started through public channels (e.g., URLs).
  - Both sessions and processes can be **dynamically created**

Alice and Bob buy a book together

Protocol

Choreography

Alice and Bob buy a book together

```
Buyer   → Seller  :  string;     // Ask the price
Seller  → Buyer   :  int;        // Get the price
Buyer   → Helper  :  int;        // Contribution
Helper  → Seller  :  {           // Choice
                     ok:   Seller  → Helper: string;
                           end,
                     ko:   end
                     }
```

Alice and Bob buy a book together

```
Buyer   →  Seller  :  string;     // Ask the price
Seller  →  Buyer   :  int;        // Get the price
Buyer   →  Helper  :  int;        // Contribution
Helper  →  Seller  :  {           // Choice
                      ok:   Seller  → Helper: string;
                            end,
                      ko:   end
                      }
```

- Let a be a public URL (e.g., www.amazon.com) with the protocol above.

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)
```

Active Process

Created Processes

Public name

Created Session

Alice and Bob buy a book together

```
Buyer   → Seller  : string;     // Ask the price
Seller  → Buyer   : int;        // Get the price
Buyer   → Helper  : int;        // Contribution
Helper  → Seller  : {           // Choice
                    ok:   Seller  → Helper: string;
                          end,
                    ko:   end
                    }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"          →  amazon.book          :  k
```

Sender

Receiver

Session

Alice and Bob buy a book together

```
Buyer   →  Seller  :  string;     // Ask the price
Seller  →  Buyer   :  int;        // Get the price
Buyer   →  Helper  :  int;        // Contribution
Helper  →  Seller  :  {           // Choice
                      ok:   Seller  → Helper: string;
                            end,
                      ko:   end
                      }
```

- Let a be a public URL (e.g., www.amazon.com) with the protocol above.

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)
```

Active Process

Created Processes

Public name

Created Session

Alice and Bob buy a book together

```
Buyer   → Seller  : string;     // Ask the price
Seller  → Buyer   : int;        // Get the price
Buyer   → Helper  : int;        // Contribution
Helper  → Seller  : {           // Choice
                    ok:   Seller  → Helper: string;
                          end,
                    ko:   end
                    }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"        →  amazon.book          :  k     ;
amazon.price(book)     →  alice.price          :  k
```

Sender Receiver Session

Alice and Bob buy a book together

```
Buyer   → Seller  : string;     // Ask the price
Seller  → Buyer   : int;        // Get the price
Buyer   → Helper  : int;        // Contribution
Helper  → Seller  : {           // Choice
                    ok:   Seller  → Helper: string;
                          end,
                    ko:   end
                    }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"        →  amazon.book          :  k     ;
amazon.price(book)     →  alice.price          :  k     ;
alice.(price/2)        →  bob.contrib          :  k
```

Sender

Receiver

Session

Alice and Bob buy a book together

```
Buyer   →  Seller  :  string;     // Ask the price
Seller  →  Buyer   :  int;        // Get the price
Buyer   →  Helper  :  int;        // Contribution
Helper  →  Seller  :  {           // Choice
                      ok:   Seller  → Helper: string;
                            end,
                      ko:   end
                      }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"        →  amazon.book          :  k     ;
amazon.price(book)     →  alice.price          :  k     ;
alice.(price/2)        →  bob.contrib          :  k     ;
if (contrib < 100$)@bob
```

Condition Evaluator

Alice and Bob buy a book together

```
Buyer   →  Seller  :  string;     // Ask the price
Seller  →  Buyer   :  int;        // Get the price
Buyer   →  Helper  :  int;        // Contribution
Helper  →  Seller  :  {           // Choice
                      ok:   Seller  → Helper: string;
                            end,
                      ko:   end
                      }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"          →  amazon.book         :  k     ;
amazon.price(book)       →  alice.price         :  k     ;
alice.(price/2)          →  bob.contrib         :  k     ;
if (contrib < 100$)@bob
   bob                   →  amazon              :  k[ok]
```

Sender Receiver Session

Alice and Bob buy a book together

```
Buyer   →  Seller  :  string;     // Ask the price
Seller  →  Buyer   :  int;        // Get the price
Buyer   →  Helper  :  int;        // Contribution
Helper  →  Seller  :  {           // Choice
                      ok:  Seller  → Helper: string;
                           end,
                      ko:  end
                      }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"          →  amazon.book      :  k        ;
amazon.price(book)       →  alice.price      :  k        ;
alice.(price/2)          →  bob.contrib      :  k        ;
if (contrib < 100$)@bob
   bob                   →  amazon           :  k[ok]    ;
   amazon.text(book)     →  bob.text         :  k
```

Sender Receiver Session

Alice and Bob buy a book together

```
Buyer   → Seller  :  string;     // Ask the price
Seller  → Buyer   :  int;        // Get the price
Buyer   → Helper  :  int;        // Contribution
Helper  → Seller  :  {           // Choice
                     ok:   Seller  → Helper: string;
                           end,
                     ko:   end
                     }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"          →  amazon.book      :  k        ;
amazon.price(book)       →  alice.price      :  k        ;
alice.(price/2)          →  bob.contrib      :  k        ;
if (contrib < 100$)@bob
   bob                   →  amazon           :  k[ok]    ;
   amazon.text(book)     →  bob.text         :  k
else
```

Alice and Bob buy a book together

```
Buyer   →  Seller  :  string;     // Ask the price
Seller  →  Buyer   :  int;        // Get the price
Buyer   →  Helper  :  int;        // Contribution
Helper  →  Seller  :  {           // Choice
                      ok:   Seller  → Helper: string;
                            end,
                      ko:   end
                      }
```

```
alice[Buyer], bob[Helper] start amazon[Seller] :  a(k)  ;
alice."rabbits"          →  amazon.book      :  k        ;
amazon.price(book)       →  alice.price      :  k        ;
alice.(price/2)          →  bob.contrib      :  k        ;
if (contrib < 100$)@bob
   bob                   →  amazon           :  k[ok]    ;
   amazon.text(book)     →  bob.text         :  k
else
   bob                   →  amazon           :  k[ko]
```

Sender Receiver Session