Specifying *Stateful Asynchronous* Properties for Distributed Programs

Tzu-chun Chen and Kohei Honda

Queen Mary College, University of London

September 2, 2012

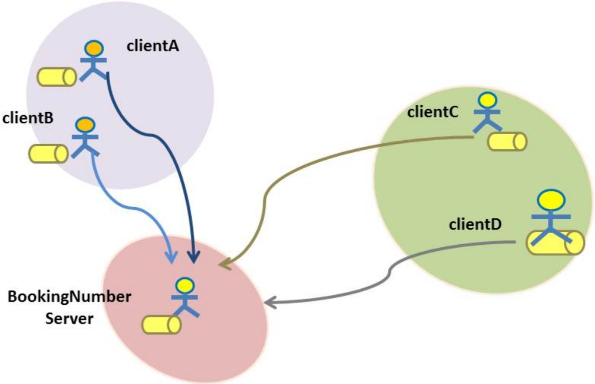
The Challenges in Distributed Computing

- To maintain the systems's safe and reliable.
- To understand the interaction behaviours for design, implementation, maintenance and other purposes
- To identify effective specification methods for programs and applications so that trusted parties (e.g. system monitors) can judge the correctness of behaviours.

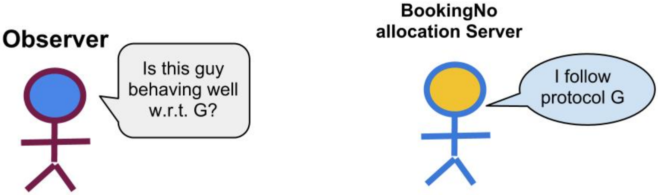
Dynamic Checking in Asynchronous Environments

- Dynamic checking is needed during runtime because not all parties in large-scale distributed systems can be statically checked.
- To carry out dynamic checking in *asynchronous environments*, we consider:
  1. complication by the effects of *states* of remote endpoints.
  2. subtleties of *asynchronous observations of actions* (i.e. the order of actions is not preserved).

Motivating Example: How to Assign Booking Numbers?



Motivating Example: How to Assign Booking Numbers?



1. Specification defined by protocol G

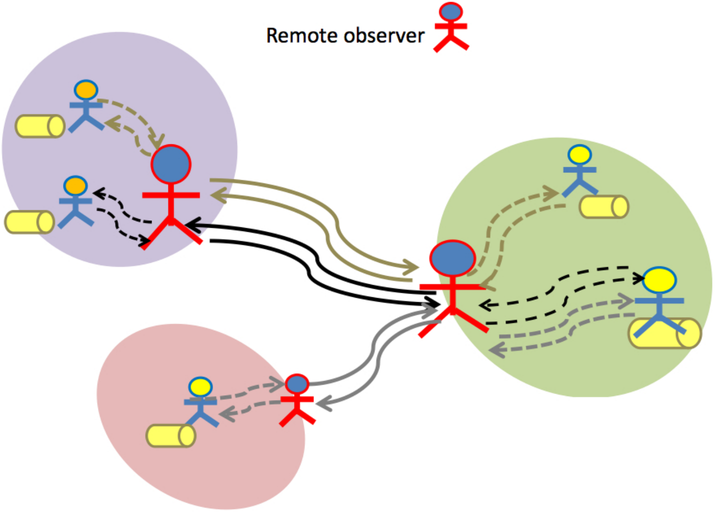
2. Having states of observee to mimic it.

Motivating Example: How to Assign Booking Numbers?

$$
\begin{array}{lcl}
G_{\mathsf{sync}} & = & \mathtt{client} \to \mathtt{server} : \mathsf{req}(\varepsilon). \\
 & & \mathtt{server} \to \mathtt{client} : \mathsf{ans}(x)\langle x = \mathbf{c};\ \mathbf{c} := \mathbf{c} + 1\rangle\langle \mathsf{true}; \varepsilon\rangle. \\
 & & \mathsf{end}
\end{array}
$$

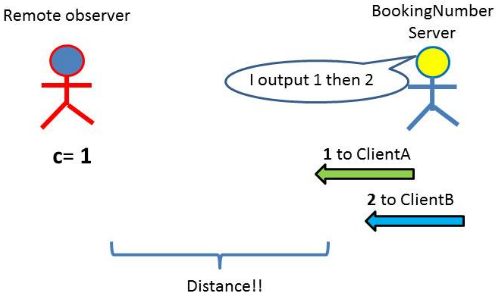
Motivating Example: How to Assign Booking Numbers?

- But, G_{sync} does not work well for a remote observer......



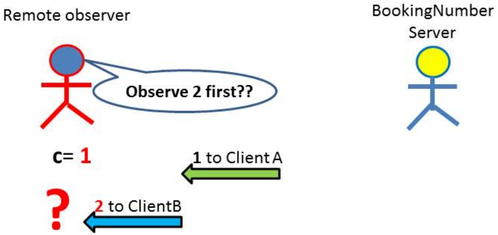
Motivating Example: How to Assign Booking Numbers?

- But, G_{sync} does not work well for a remote observer.....



Motivating Example: How to Assign Booking Numbers?

- But, G_{sync} does not work well for a remote observer.....



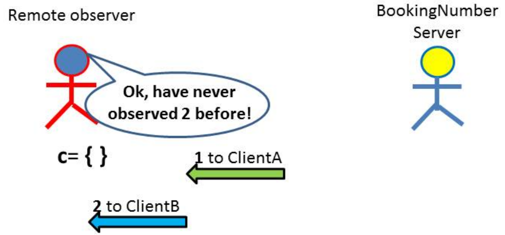
Motivating Example: How to Assign Booking Numbers?

- An asynchronous specification can work for both synchronous and asynchronous observations.

$$
\begin{array}{rcl}
G_{\text{async}} & = & \texttt{client} \to \texttt{server} : \mathsf{req}(\varepsilon). \\
 & & \texttt{server} \to \texttt{client} : \mathsf{ans}(x)\langle x \notin \mathbf{c};\ \mathbf{c} := \mathbf{c} \cup \{x\}\rangle\langle \mathsf{true}; \varepsilon\rangle. \\
 & & \mathsf{end}
\end{array}
$$

Motivating Example: How to Assign Booking Numbers?

- G_{async} works well for a remote observer......



Motivating Example: How to Assign Booking Numbers?

- G_{async} works well for a remote observer......

Motivating Example: How to Assign Booking Numbers?

- G_{async} works well for a remote observer......

Capturing Causality By Using Sets

- G_{assign} is a refinement of G_{async}.

$$
\begin{array}{rcl}
G_{\text{assign}} & = & \texttt{client} \to \texttt{server} : \mathsf{req}(\varepsilon)\langle \mathsf{true}; \varepsilon\rangle\langle \mathsf{true}; \mathbf{t} := \mathbf{t} + 1, \mathbf{c} := \mathbf{c} \uplus \{\mathbf{t}\}\rangle. \\
 & & \texttt{server} \to \texttt{client} : \mathsf{ans}(x)\langle x \in \mathbf{c};\ \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\langle \mathsf{true}; \varepsilon\rangle. \\
 & & \mathsf{end}
\end{array}
$$

Some valid traces of asynchronous interactions (I)

Traces of permitted actions with the corresponding state change

cases	1st	2nd	3rd	4th
(I) actions	req from clientA req from clientB req from clientA req from clientB	req from clientB req from clientA req from clientB req from clientA	1 to clientA 1 to clientA 1 to clientB 1 to clientB	2 to clientB 2 to clientB 2 to clientA 2 to clientA
(I) states	$\mathbf{t} \mapsto 1, \mathbf{c} \mapsto \{1\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{1, 2\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{2\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{\}$

1. If `server` receives n requests, then (assuming the server issues the booking numbers starting from 1) as a whole the numbers which can be issued are among $\{1, 2, .., n\}$.
2. If `server` issues a number from this set, the remaining numbers are what it can issue.

Some valid traces of asynchronous interactions (II)

Traces of permitted actions with the corresponding state change

cases	1st	2nd	3rd	4th
(II) actions	req from clientA	req from clientB	2 to clientA	1 to clientB
	req from clientB	req from clientA	2 to clientA	1 to clientB
	req from clientA	req from clientB	2 to clientB	1 to clientA
	req from clientB	req from clientA	2 to clientB	1 to clientA
(II) states	$\mathbf{t} \mapsto 1, \mathbf{c} \mapsto \{1\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{1, 2\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{1\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{\}$

1. If `server` receives n requests, then (assuming the server issues the booking numbers starting from 1) as a whole the numbers which can be issued are among $\{1, 2, .., n\}$.
2. If `server` issues a number from this set, the remaining numbers are what it can issue.

Some valid traces of asynchronous interactions (III)

Traces of permitted actions with the corresponding state change

cases	1st	2nd	3rd	4th
(III) actions	req from clientA req from clientB	1 to clientA 1 to clientB	req from clientB req from clientA	2 to clientB 2 to clientA
(III) states	$\mathbf{t} \mapsto 1, \mathbf{c} \mapsto \{1\}$	$\mathbf{t} \mapsto 1, \mathbf{c} \mapsto \{\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{2\}$	$\mathbf{t} \mapsto 2, \mathbf{c} \mapsto \{\}$

1. If S receives n requests, then (assuming the server issues the booking numbers starting from 1) as a whole the numbers which can be issued are among $\{1, 2, .., n\}$.
2. If S issues a number from this set, the remaining numbers are what it can issue.

Stateful Protocols (SP) Grammar

global stateful protocol	local stateful protocol
$G ::= \mathtt{p} \to \mathtt{q} : \{l_i(x_i : S_i)\langle A_i; E_i\rangle\langle A'_i; E'_i\rangle.G_i\}_{i\in I}$	$T ::= \mathtt{p}!\{l_i(x_i : S_i)\langle A_i; E_i\rangle.T_i\}_{i\in I}$
$\mid\ G_1 \mid G_2 \qquad (G_1 \wedge G_2 = \emptyset)$	$\mid\ \mathtt{p}?\{l_i(x_i : S_i)\langle A_i; E_i\rangle.T_i\}_{i\in I}$
$\mid\ \text{end}$	$\mid\ \text{end}$

$\Theta = \langle \Gamma\ ;\ \Delta\ ;\ D\rangle$ endpoint spec

$\Gamma ::= \emptyset \mid \Gamma, a : \mathtt{I}(G[\mathtt{p}]) \mid \Gamma, a : \mathtt{O}(G[\mathtt{p}])$ shared environment

$\Delta ::= \emptyset \mid \Delta, s[\mathtt{p}] : T$ session environment

$D ::= \emptyset \mid D, \mathbf{f} \mapsto e$ data storage

$$
\begin{array}{lcl}
A & ::= & e_1 = e_2 \mid e_1 > e_2 \mid e_1 \in e_2 \mid A_1 \wedge A_2 \mid \neg A \mid \forall x.\ A \\
E & ::= & E, \mathbf{f} := e \mid \text{if } A \text{ then } E_1 \text{ else } E_2 \\
S & ::= & \text{nat} \mid \text{bool} \mid \text{string..} \mid S_1 \times S_2 \mid \text{set} \mid \text{map}(S_1, S_2) \\
e & ::= & x \mid v \mid \mathbf{f} \mid op(e_1, ..., e_n)
\end{array}
$$

- **f**: state, an attribute/ a field stored at a location (e.g. database) that a process can access / update.

The LTS of Specifications (output)

$$\ell ::= \overline{a}(s[\mathrm{p}] : G) \mid \overline{a}\langle s[\mathrm{p}] : G\rangle \mid a(s[\mathrm{p}] : G) \mid s[\mathrm{p},\mathrm{q}]!l(v) \mid s[\mathrm{p},\mathrm{q}]?l(v)$$

[REQ-INI]
$$\frac{s \notin \mathrm{dom}(\Delta),\ \mathrm{role}(G) = \{\mathrm{p}_i\}_{i\in I}}{\langle \Gamma, a : \circ(G[\mathrm{p}_j]);\ \Delta, \{s[\mathrm{p}_i] : G \upharpoonright \mathrm{p}_i\}_{i\in I};\ D\rangle \xrightarrow{\overline{a}(s[\mathrm{p}_j]:G)} \langle \Gamma, a : \circ(G[\mathrm{p}_j]);\ \Delta, \{s[\mathrm{p}_i] : G \upharpoonright \mathrm{p}_i\}_{i\in I\setminus\{j\}};\ D\rangle}$$

[REQ]
$$\frac{s \in \mathrm{dom}(\Delta),\ \mathrm{p}_j \in \mathrm{role}(G)}{\langle \Gamma, a : \circ(G[\mathrm{p}_j]);\ \Delta, s[\mathrm{p}_i] : G \upharpoonright \mathrm{p}_i;\ D\rangle \xrightarrow{\overline{a}\langle s[\mathrm{p}_j]:G\rangle} \langle \Gamma, a : \circ(G[\mathrm{p}_j]);\ \Delta;\ D\rangle}$$

[SEL]
$$\frac{T = \mathrm{q}!\{l_i(x_i : S_i)\langle A_i; E_i\rangle.T_i'\}_{i\in I},\quad \Gamma \vdash v : S_j,\quad \Gamma \models A_j\{v/x_j\}}{\langle \Gamma;\ \Delta, s[\mathrm{p}] : T;\ D\rangle \xrightarrow{s[\mathrm{p},\mathrm{q}]!l_j(v)} \langle \Gamma;\ \Delta, s[\mathrm{p}] : T_j'\{v/x_j\};\ D\ \mathtt{after}\ E_j\{v/x_j\}\rangle}$$

Definition of Trace

Definition (trace)

A *trace* $(s, s', \ldots)$ is a sequence of actions which satisfy the standard binding conventions.

Definition (valid traces of Θ)

We define trace(Θ) *a set of* valid traces *of* Θ, $\forall s \in \text{trace}(\Theta), \exists \Theta'$ *such that* $\Theta \xrightarrow{s} \Theta'$.

- $\text{trace}(\Theta_{\text{sync}}) \subset \text{trace}(\Theta_{\text{assign}}) \subset \text{trace}(\Theta_{\text{async}})$
- Both Θ_{assign} and Θ_{async} can verify
  - The number of inputs is more than the number of outputs.
  - Every assigned value is unique.
- But only Θ_{assign} can verify
  - At every moment, the assigned value is smaller than the current number of inputs.

Legal Unit Permutation

Definition (legal unit permutation)

$\ell_1 \ell_2$ is *legally permuted* to $\ell_2 \cdot \ell_1$ when

1. ℓ_1 and ℓ_2 are both input session actions from different senders.
2. ℓ_1 and ℓ_2 are both output session actions to different receivers.
3. ℓ_1 is an output session action and ℓ_2 is an input one.
4. One of them is $\overline{a}(s[\mathrm{p}] : G)$ or $\overline{a}\langle s[\mathrm{p}] : G\rangle$ or $a(s[\mathrm{p}] : G)$, and ℓ_1 does not bind ℓ_2.

Example

permutable	*non-permutable*
$s[\mathrm{p}, \mathrm{q}_1]!l_1(v_1) \cdot s[\mathrm{p}, \mathrm{q}_2]!l_2(v_2)$	$s[\mathrm{q}, \mathrm{p}]?l_1(v_1) \cdot s[\mathrm{p}, \mathrm{q}]!l_2(v_2)$
$s[\mathrm{p}, \mathrm{q}]!l_1(v_1) \cdot s[\mathrm{q}, \mathrm{p}]?l_2(v_2)$	$\overline{a}(s[\mathrm{p}] : G) \cdot s[\mathrm{p}, \mathrm{q}]!l(v)$

Process and Synchronous v.s. Asynchronous Observables

Definition (process)

A *process* (P, Q, ..) is a set of prefix-closed traces.

- $\mathsf{Obs}_s(P)$: the set of traces *synchronously* observed from P.
- $\mathsf{Obs}_a(P)$: the set of traces *asynchronously* observed from P.

Definition (synchronous and asynchronous observables)

1. $\mathsf{Obs}_s(P) \stackrel{\text{def}}{=} P$.
2. $\mathsf{Obs}_a(P)$ is the set of all legal permutation variants of the traces in P.

Example

- If $\ell_1 \cdot \ell_2 \in \mathsf{Obs}_s(P)$ and $\ell_1 \cdot \ell_2 \curvearrowright \ell_2 \cdot \ell_1$, then $\ell_1 \cdot \ell_2 \in \mathsf{Obs}_a(P)$ and $\ell_2 \cdot \ell_1 \in \mathsf{Obs}_a(P)$.

Satisfaction

Definition (satisfaction up to observables)

$P \models_{\texttt{sync}} \Theta$, when the two conditions hold:

1. (safety) $\mathsf{Obs}_s(P) \subset \mathsf{trace}(\Theta)$.
2. (input consistency [Bodei 98])
   Whenever $\mathtt{s} \in \mathsf{Obs}_s(P)$ and $\mathtt{s} \cdot \ell \in \mathsf{trace}(\Theta)$ where ℓ is an input, if P is capable to input ℓ after $\mathtt{s}$, $\mathtt{s} \cdot \ell \in \mathsf{Obs}_s(P)$.

$P \models_{\texttt{async}} \Theta$, if the same conditions as above (neglecting the orange part) holds replacing each $\mathsf{Obs}_s(P)$ with $\mathsf{Obs}_a(P)$.

Asynchronous Verifiable

Definition (asynchronously verifiable specification)

We say Θ is *asynchronously verifiable* or simply *asynchronous* when $s \in \mathsf{trace}(\Theta)$ and $s \curvearrowright s'$ imply $s' \in \mathsf{trace}(\Theta)$.

Proposition

Θ is asynchronous iff, for each P, $P \models_{\mathtt{sync}} \Theta$ implies $P \models_{\mathtt{async}} \Theta$.

Proposition

If $P \approx_{\mathtt{async}} Q$ and $P \models_{\mathtt{async}} \Theta$ then $Q \models_{\mathtt{async}} \Theta$.

Commutativity (I): Predicate and Update Functions

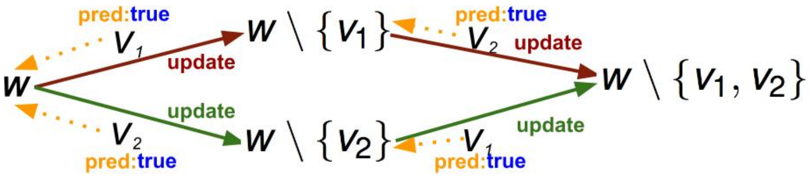
For an obligation ξ defined in Θ on role `server`

$$s[\texttt{server}] \quad = \quad \texttt{client}!\mathsf{ans}(x : \mathsf{int})\langle x \in \mathbf{c}\,;\ \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle$$

$$\xi \quad \stackrel{\text{def}}{=} \quad \texttt{client}!\mathsf{ans}(x : \mathsf{int})\langle x \in \mathbf{c}; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle$$

$$\texttt{pred}(\xi) \stackrel{\text{def}}{=} \lambda x, \mathbf{c}.(x \in \mathbf{c}) \qquad \texttt{upd}(\xi) \stackrel{\text{def}}{=} \lambda x, \mathbf{c}.\langle \mathbf{c} \setminus \{x\}\rangle$$

ξ is commutative over itself:



Commutativity(II)

Definition (commutativity)

Given Θ, let $\xi_1, .., \xi_n$ be all the obligations usable in Θ. Then we say Θ is *commutative* if the following conditions hold.

1. For (possibly identical) ξ'_1 and ξ'_2 from $\{\xi_1, .., \xi_n\}$, if both are inputs or both are outputs, then ξ'_1 and ξ'_2 are commutative.
2. For distinct ξ'_1 and ξ'_2 from $\{\xi_1, .., \xi_n\}$, if ξ'_1 is an output and ξ'_2 is an input then ξ'_1, ξ'_2 is semi-commutative.

Commutativity Implies Asynchronous

Θ is asynchronous when all obligations used in the specifications for the target process commute over each other up to legal permutations.

Proposition (Commutativity implies asynchronous)

If Θ is commutative then it is asynchronous.

Note that the other way round is not right.

Example

Assume the initial values of states $\mathbf{c}$ and $\mathbf{t}$ are both 0, and we have following local SP for role p in session s is:

$$s[\mathrm{p}] : \mathrm{q}_1!\mathrm{ans}(\varepsilon)\langle \mathsf{true}; \mathbf{c} = 50\rangle \, . \, \mathrm{q}_2!\mathrm{ans}(\varepsilon)\langle \mathsf{true}; \mathbf{t} = \mathbf{c} + 10\rangle$$

It is asynchronous because it checks nothing . But it is not commutative. Consider $\ell_1 = s[\mathrm{p}, \mathrm{q}_1]!\mathrm{ans}(\varepsilon)$ and $\ell_2 = s[\mathrm{p}, \mathrm{q}_2]!\mathrm{ans}(\varepsilon)$. $\ell_1 \cdot \ell_2$ makes state $\mathbf{t}$ be 60, but $\ell_2 \cdot \ell_1$ makes state $\mathbf{t}$ be 10.

Specifications in Examples

Proposition

1. Θ_{async} and Θ_{assign} are commutative thus are asynchronous.
2. Θ_{sync} is not asynchronous.

Proof.

That Θ_{assign} (resp. Θ_{async}) is commutative can be easily proved by proving every pair ξ_i, ξ_j in Θ_{assign} (resp. Θ_{async}). □

Proposition

With the stateful protocol (SP) grammar restricting operations on integers to be the addition, subtraction and the sets, then the commutativity of specifications is decidable.

Proof.

Based on [Zarba 02]. □

Related works

- There are many work on asynchronous equivalences and calculi such as
  - [de Boer, Kok, Palamidessi and Rutten 1991],
  - [Jifeng, Josephs and Hoare 1990],
  - [Honda and Tokoro 1991]
  - [Amadio, Castellani and Sangiorgi 1996]
  - ...

  but they do not treat the problem of asynchronous *stateful specifications*.
- [Bocchi, Demangeon and Yoshida TGC 12]'s work for the proof system with stateful logics based on the π-calculus.
- So far the same technical problem of *semantics* has not been discussed in the literature, as far as we know.

Future works

- Find out the necessary and sufficient sound characterisation for asynchronous specifications.
- Propose promising approaches to automatically translate a synchronous suitable stateful specification to an asynchronous suitable one.
- Deeper analysis of a Θ by identifying and formalising the elements of Θ as properties on traces. Then we can:
  1. describe the components ("requirements") of Θ,
  2. compare the expressiveness of Θ concretely,
  3. design a semantically well-founded policy language under asynchronous-stateful environments.
- Apply the theory to specifications and runtime monitoring in a real-world distribtued software infrastructure.

Thank you !

Appendix: the Standard Binding Conventions

A sequence of actions that satisfy standard binding conventions is:

1. for $\mathrm{s} \cdot \overline{a}(s[\mathrm{p}] : G)$ or $\mathrm{s} \cdot a(s[\mathrm{p}] : G)$, s has no action $\overline{a}\langle s[\mathrm{p}] : G\rangle$ or $a(s[\mathrm{p}] : G)$ or $s[\mathrm{p}, \mathrm{q}] \dagger l(v), \dagger \in \{?, !\}$.
2. for $\mathrm{s} \cdot \overline{a}\langle s[\mathrm{p}] : G\rangle$, s has action $\overline{a}(s[\mathrm{p}] : G)$, but no action $a(s[\mathrm{p}] : G)$ or $s[\mathrm{p}, \mathrm{q}] \dagger l(v), \dagger \in \{?, !\}$.
3. for $\mathrm{s} \cdot s[\mathrm{p}, \mathrm{q}] \dagger l(v), \dagger \in \{?, !\}$, s has actions $\overline{a}\langle s[\mathrm{p}] : G\rangle$ and $\overline{a}(s[\mathrm{p}] : G)$, or $a(s[\mathrm{p}] : G)$.

For Example: Θ_{assign}

Θ_{assign} is the local specification of role `server` projected from G_{assign}

$$\begin{array}{l}\langle \Gamma, \mathtt{ser} : \mathtt{I}(G_{\text{assign}}[\mathtt{server}]); \\ \Delta, \{s_i[\mathtt{server}] : \mathtt{client}?\mathrm{req}(\varepsilon)\langle \mathrm{true}\ ; \mathbf{t} := \mathbf{t}+1, \mathbf{c} := \mathbf{c} \uplus \{\mathbf{t}\}\rangle. \\ \qquad\qquad \mathtt{client}!\mathrm{ans}(x)\langle x \in \mathbf{c}; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\}_{i \in \{1,2\}}; \\ D, \mathbf{c} \mapsto \{\}, \mathbf{t} \mapsto 0\rangle\end{array}$$

For Example: Θ_{assign}

Θ_{assign} is the local specification of role `server` projected from G_{assign}

$$
\begin{array}{l}
\langle \Gamma, \mathtt{ser} : \mathtt{I}(G_{\text{assign}}[\mathtt{server}]); \\
\;\Delta, s_1[\mathtt{server}] : \mathtt{client}?\mathrm{req}(\varepsilon)\langle \mathrm{true}\,; \mathbf{t} := \mathbf{t}+1, \mathbf{c} := \mathbf{c} \uplus \{\mathbf{t}\}\rangle. \\
\qquad\qquad\quad \mathtt{client}!\mathrm{ans}(x)\langle x \in \mathbf{c}; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle, \\
\quad\; s_2[\mathtt{server}] : \mathtt{client}?\mathrm{req}(\varepsilon)\langle \mathrm{true}\,; \mathbf{t} := \mathbf{t}+1, \mathbf{c} := \mathbf{c} \uplus \{\mathbf{t}\}\rangle. \\
\qquad\qquad\quad \mathtt{client}!\mathrm{ans}(x)\langle x \in \mathbf{c}; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\}_{i\in\{1,2\}}; \\
\;D, \mathbf{c} \mapsto \{\}, \mathbf{t} \mapsto 0\rangle
\end{array}
$$

$$\xrightarrow{s_1[\mathtt{client},\mathtt{server}]?\mathrm{req}(\varepsilon)}$$

$$
\begin{array}{l}
\langle \Gamma, \mathtt{ser} : \mathtt{I}(G_{\text{assign}}[\mathtt{server}]); \\
\;\Delta, s_1[\mathtt{server}] : \mathtt{client}!\mathrm{ans}(x)\langle x \in \mathbf{c}; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle, \\
\quad\; s_2[\mathtt{server}] : \mathtt{client}?\mathrm{req}(\varepsilon)\langle \mathrm{true}\,; \mathbf{t} := \mathbf{t}+1, \mathbf{c} := \mathbf{c} \uplus \{\mathbf{t}\}\rangle. \\
\qquad\qquad\quad \mathtt{client}!\mathrm{ans}(x)\langle x \in \mathbf{c}; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle; \\
\;D, \mathbf{c} \mapsto \{1\}, \mathbf{t} \mapsto 1\rangle
\end{array}
$$

Appendix: Θ_{assign} Approaches Θ_{sync} more than Θ_{async}.

Example

$$\mathrm{s}' = s_1[\mathtt{client}, \mathtt{server}]?req(\varepsilon) s_1[\mathtt{server}, \mathtt{client}]!ans(1) s_1[\mathtt{client}, \mathtt{server}]?req(\varepsilon)$$

$\mathrm{s}' \in \mathrm{Obs}_s(P)$ *whenever* $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$.

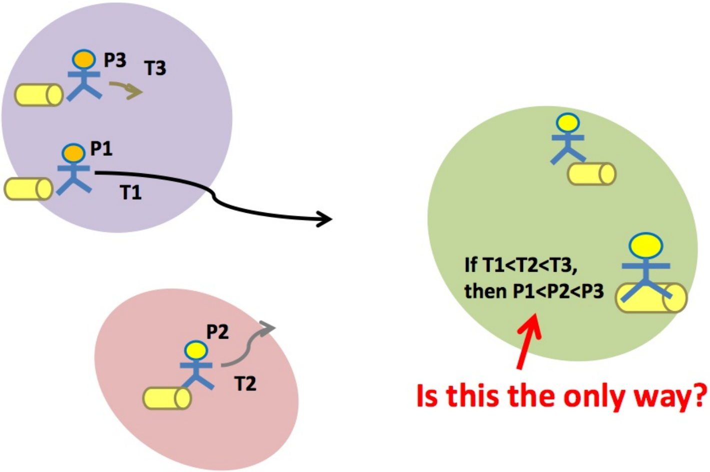
But Θ_{async} is not.

Example

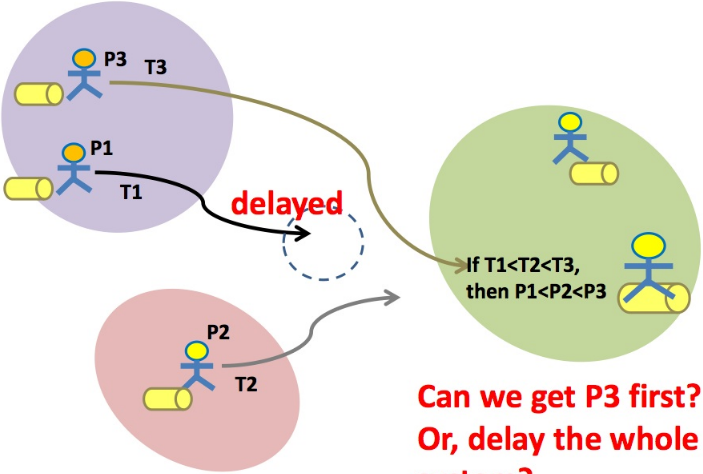
$$\mathrm{s} = s_1[\mathtt{client}, \mathtt{server}]?req(\varepsilon) \cdot s_1[\mathtt{server}, \mathtt{client}]!ans(3) \cdot s_1[\mathtt{client}, \mathtt{server}$$

But $\mathrm{s} \notin \mathrm{Obs}_s(P)$ *whenever* $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$.

Appendix: Timestamp? (Lamport, 1978)



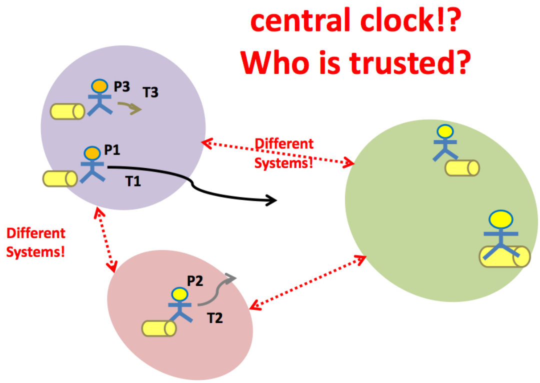
Appendix:Timestamp?



Can we get P3 first?
Or, delay the whole system?

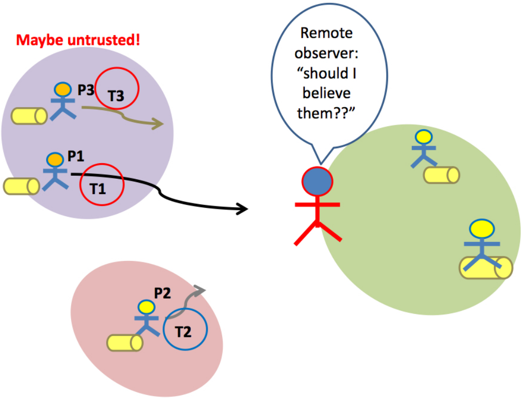
Appendix:Timestamp?

- Why don't we use *timestamp*?
  - to timestamp the messages from *different systems* is an issue.



Appendix:Timestamp?

- Why don't we use *timestamp*?
  - some sub-systems may not be trusted.



Appendix: Previous Works for Specification for Static and *Runtime* Checking for Distributed Computing

- The specification for runtime verification is needed but not trivial, especially when asynchrony is considered.
  1. Consistent property for specifications of sender and receiver is not trivial because interaction messages may be still floating around the network:

$$G = \texttt{client} \to \texttt{server} : \mathsf{req}(x : \mathsf{string}).\texttt{server} \to \texttt{client} : \mathsf{ans}(y : \mathsf{int}).\epsilon$$

$$T_{\texttt{client}} = \texttt{server}!\mathsf{req}(x : \mathsf{string}).\texttt{server}?\mathsf{ans}(y : \mathsf{int}).\mathsf{end}$$
$$T_{\texttt{server}} = \texttt{client}?\mathsf{req}(x : \mathsf{string}).\texttt{client}!\mathsf{ans}(y : \mathsf{int}).\mathsf{end}$$

Appendix: Previous Works for Specification for Static and *Runtime* Checking for Distributed Computing

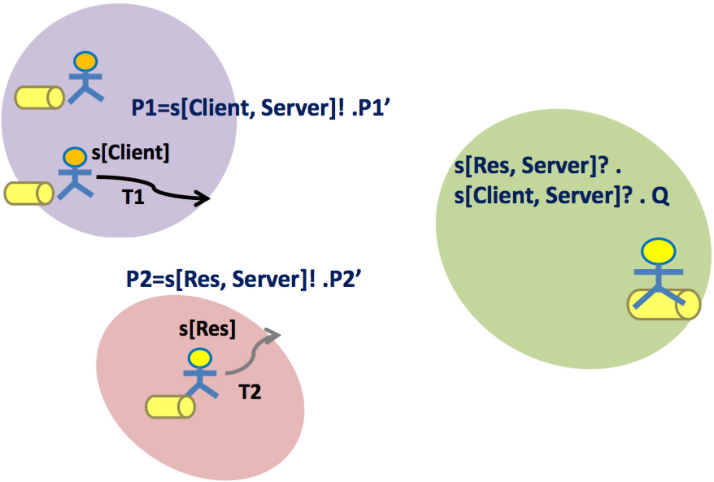
- The specification for runtime verification is needed but not trivial, especially when asynchrony is considered.
  1. Consistent property for specifications of sender and receiver is not trivial because interaction messages may be still floating around the network:

$$G = \texttt{client} \to \texttt{server} : \mathsf{req}(x : \mathsf{string}).\texttt{server} \to \texttt{client} : \mathsf{ans}(y : \mathsf{int}).$$

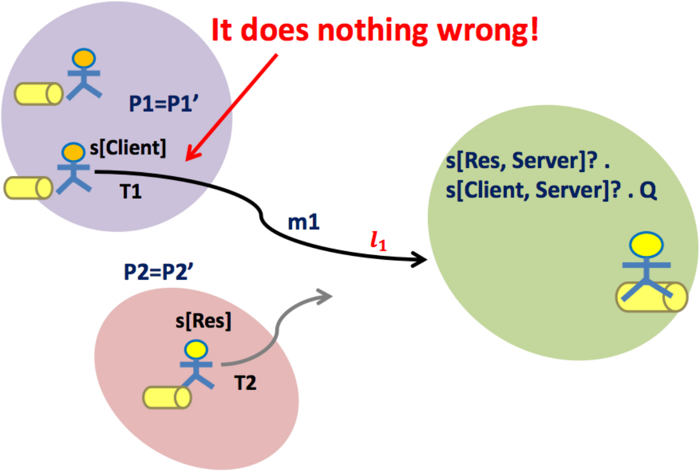
$$T_{\texttt{client}} = \texttt{server}?\mathsf{ans}(y : \mathsf{int}).\mathsf{end}$$
$$T_{\texttt{server}} = \texttt{client}?\mathsf{req}(x : \mathsf{string}).\texttt{client}!\mathsf{ans}(y : \mathsf{int}).\mathsf{end}$$

  because `server` has not yet received message $s\langle \texttt{client}, \texttt{server}, \mathsf{ans}\langle v\rangle\rangle$

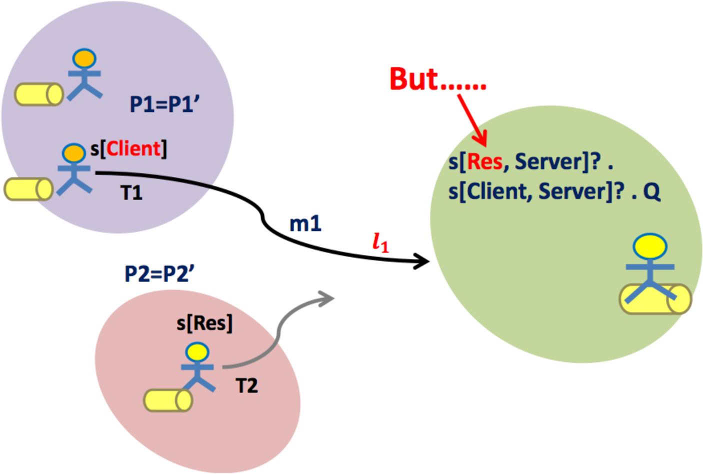
Appendix: Permutation Mechanism for Asynchrony



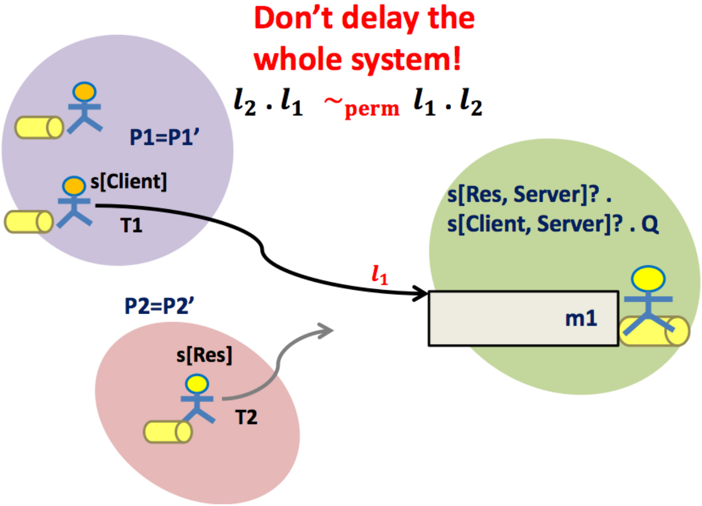
Appendix: Permutation Mechanism for Asynchrony



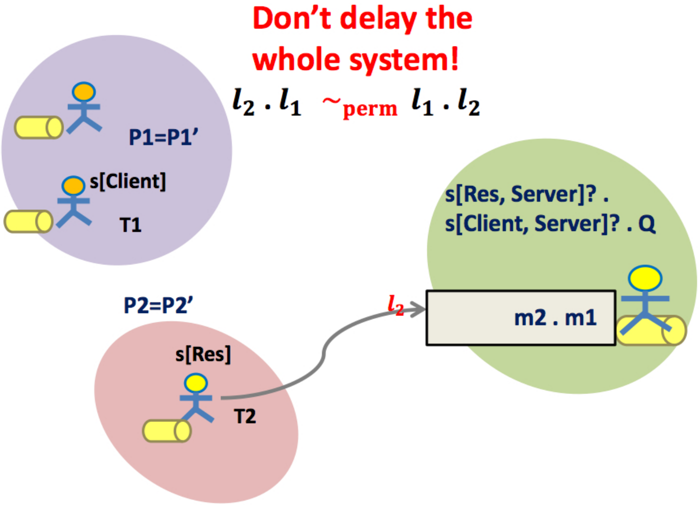
Appendix: Permutation Mechanism for Asynchrony



Appendix: Permutation Mechanism for Asynchrony



Appendix: Permutation Mechanism for Asynchrony



Appendix: Permutation Mechanism for Asynchrony