From Orchestration to Choreography: Memoryless and Distributed Orchestrators

Imene Ben-Hafaiedh, Susanne Graf and **Sophie Quinton**

VERIMAG, Université Joseph Fourier

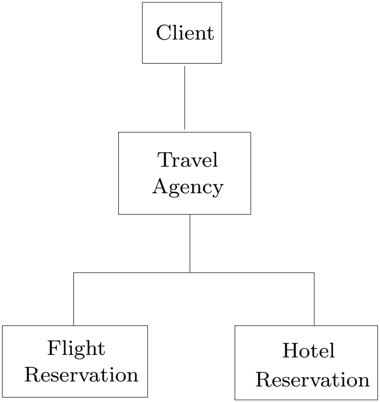
FLACOS, September 2009

Outline

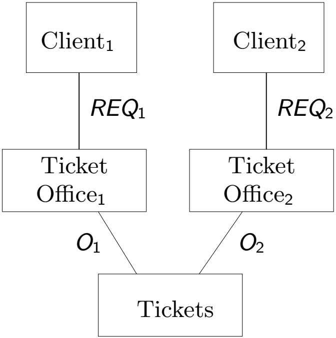
1. Motivation
2. Memoryless orchestrators
3. Distributing memoryless orchestrators
4. Conclusion

1. Motivation
2. Memoryless orchestrators
3. Distributing memoryless orchestrators
4. Conclusion

Rich interaction in Web services



n-ary rendez-vous



global priority

Orchestrators

- A contract describes how a client or a service is expected to behave.
- Syntax: $\sigma ::= 0 \mid \alpha.\,\sigma \mid \sigma + \sigma \mid \sigma \oplus \sigma \mid \texttt{rec}\, x.\,\sigma \mid x$
- A service ρ is compliant with a client σ iff $\sigma \parallel \rho$ has no deadlock.
- An orchestrator “helps” making a service compliant to a client.

Our approach

- Focus on a particular class: memoryless orchestrators (MO).
- MO can be represented as BIP connectors and priorities.
- MO can more easily be distributed.

Contracts in a fragment of CCS

Syntax

$$\sigma ::= 0 \mid \alpha.\, \sigma \mid \sigma + \sigma \mid \sigma \oplus \sigma \mid \texttt{rec}\, x.\, \sigma \mid x$$

Semantics

$$\alpha.\, \sigma \xrightarrow{\alpha} \sigma \qquad \sigma \oplus \rho \xrightarrow{\tau} \sigma \qquad \texttt{rec}\, x.\, \sigma \xrightarrow{\tau} \sigma\{{}^{\texttt{rec}\, x.\, \sigma}/_{x}\}$$

$$\frac{\sigma \xrightarrow{\tau} \sigma'}{\sigma + \rho \xrightarrow{\tau} \sigma' + \rho} \qquad \frac{\sigma \xrightarrow{\alpha} \sigma'}{\sigma + \rho \xrightarrow{\alpha} \sigma'}$$

Toy examples

A simple example that works with priorities.

Client : $\bar{a}.\, c.\, e + \bar{b}.\, d.\, e$

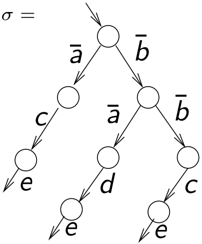
Service : $a.\, \bar{d} + b.\, \bar{d}$

A simple example that does not work with priorities.

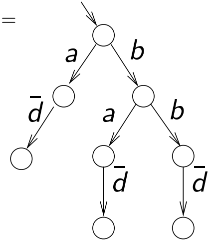
Client : $\bar{a}.\, c.\, e + \bar{b}.\, \bar{b}.\, c.\, e + \bar{b}.\, \bar{a}.\, d.\, e$

Service : $a.\, \bar{d} + b.\, b.\, \bar{d} + b.\, a.\, \bar{d}$

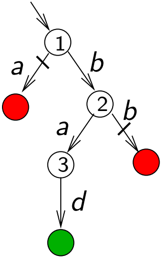
$\sigma =$



$\rho =$



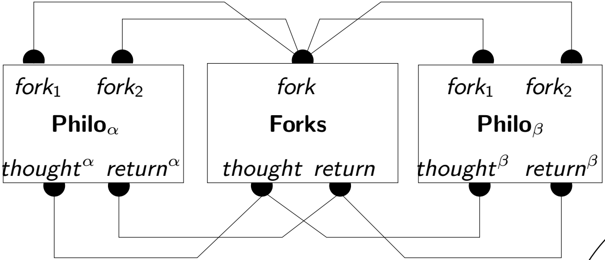
$\sigma \parallel \rho =$

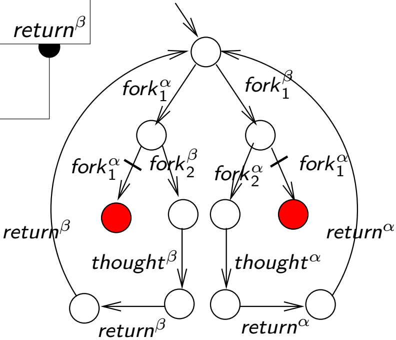


An example: the dining philosophers

Forks $= \mathtt{rec}\, x.\overline{fork}.\overline{fork}.thought.return.return.x$

Philo $= \mathtt{rec}\, x.fork_1.fork_2.\overline{thought}.\overline{return}.\overline{return}.x$





$fork_1^\beta < fork_2^\alpha$
$fork_1^\alpha < fork_2^\beta$

1. Motivation
2. Memoryless orchestrators
3. **Distributing memoryless orchestrators**
4. Conclusion

Description of the problem

- Components communicating via exchange of messages
- Global priority rules over interactions
- Restriction in this talk: only binary rendez-vous connectors

Properties expected

- Safety: in a given state, only specified interactions can be fired
- Progress: deadlock-freedom (no fairness)
- Efficiency: reduce the longest exchange of messages between two transitions

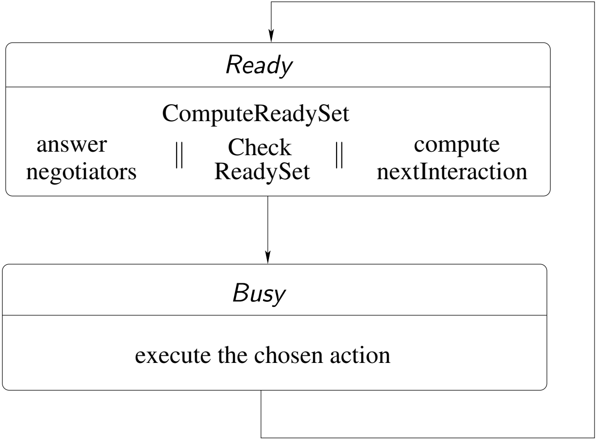
Principle of the algorithm

For a component K, an interaction is:

- possible $=$ possible for K
- ready $=$ possible for K and its counterpart
- enabled $=$ ready $+$ no ready interaction with higher priority

For each interaction α involved in a priority rule, a negotiator is chosen between the processes that participate in α.

Protocol overview



Ready

ComputeReadySet

answer negotiators || Check ReadySet || compute nextInteraction

Busy

execute the chosen action

Negotiate:

```
Require: $higherPrio(a) = \{i \mid a < i\}$
    Input: interaction $a$
    Output: $OK$ or $NOK$
toCheck ⟵ higherPrio(a)
$\forall b \in toCheck$ send $READY?(b)$
while $toCheck \neq \emptyset$ do
    if receive $READY!(b)$ then
        return $NOK$
    else if receive $NOTREADY!(b)$ then
        $toCheck \longleftarrow toCheck \setminus \{b\}$
    end if
end while
return $OK$
```

ComputeNextInteraction:

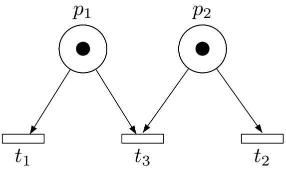
```
Require: $toNegotiate = \{i \mid negociator(i) = K\}$
    Input: set of interactions $readySet \neq \emptyset$
    Output: set of interactions $enabledSet$
Ready:
$localMax \longleftarrow readySet \backslash \{i \mid \exists j \in readySet\ s.t.\ i < j\}$
$enabledSet \longleftarrow \{i \in readySet \mid i \notin \pi\}$
for all $i \in localMax \cap toNegotiate$ do
    if $Negotiate(i) = OK$ then
        $enabledSet \longleftarrow enabledSet \cup \{i\}$
    end if
end for
if $enabledSet = \emptyset$ then
    goto Ready
else
    $enabledSetCompleted \longleftarrow true$
end if
```

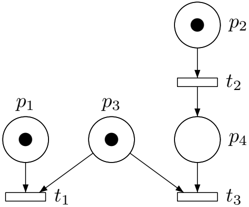
CheckReadySet:

```
Require: set of interactions $possibleSet$
    Output: interaction $i$
createNewThread ChooseNextInteraction($readySet$)
while not enabledSetCompleted do
    if receive $REFUSE(a)$ and $a \in enabledSet$ then
        kill ChooseNextInteraction
    end if
end while
choose interaction $i$ to fire among $enabledSet$
```

Safety

- t_1 and t_2 are independent when one may fire in parallel with the other.
- Otherwise t_1 and t_2 are said to be in structural conflict.
- Confusion arises if t_1 and t_2 may fire concurrently, but firing one modifies the set of transitions in actual conflict with the other.





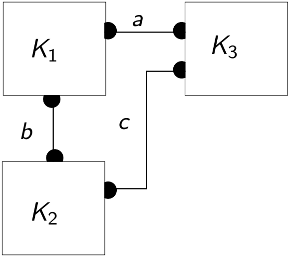
- A process can commit just one interaction at a time $\Longrightarrow$ interactions in conflict cannot be committed simultaneously.
- We do not handle confusion related to priorities.

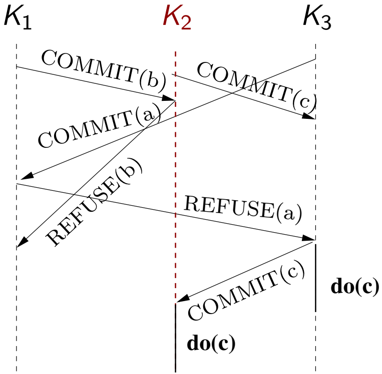
Progress

- No notion of fairness
- Avoid additional deadlocks due to negotiations
- Avoid deadlocks due to cycles: use cycle breakers

$K_1 = a + b \qquad K_3 = c + a$

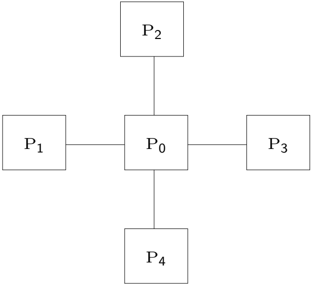
$K_2 = b + c$

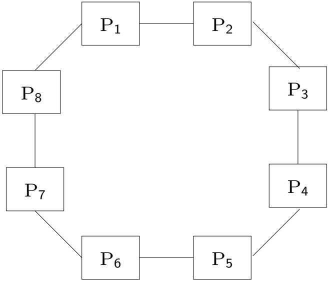




Efficiency: choice of the negociators

- Minimize the maximal number of components communicating to decide the enabledness of an interaction.
- Centralized topology $\implies$ as efficient as a central orchestrator
- Ring tobology $\implies$ at least as efficient than a central orchestrator





Conclusion and perspectives

Summary

- Memoryless orchestrators
- Concurrency and priorities can be inferred
- A distributed implementation

Future work

- Handle multiparty interactions
- Handle complex connectors: need for a new algorithm?
- Add knowledge to reduce the need for communication
- Combine this work with compositional verification
- Evaluate the approach on actual Web services