Programming services with correlation sets

Fabrizio Montesi

joint work with Marco Carbone

IT University of Copenhagen, Denmark

Introduction

Our solution

Demo

Conclusions

Service-Oriented Computing (SOC)

- A design paradigm for **distributed systems**.
- A **service-oriented** system is a network of **services**.
- Services communicate through message passing.

- Messages are tagged with **operations** (similar to method names in OO).
- **Interfaces** are obtained by typing operations with message types.
- Reference technology: Web Services.
  - Based on XML;
  - WS-BPEL (BPEL for short) for programming composition.

Sessions

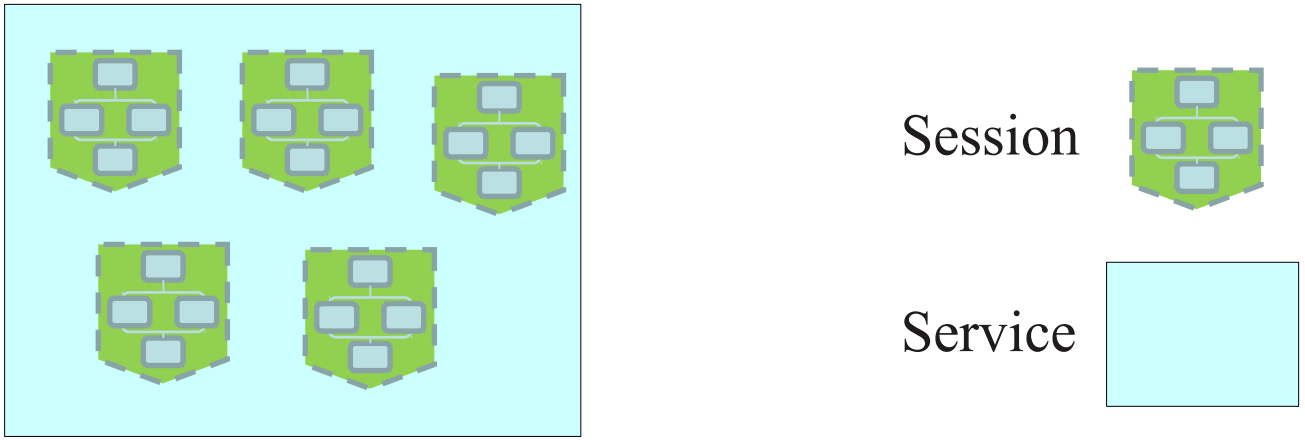
- A service may engage in different **separate conversations** with other parties.
  - Example: a chat server may manage different chat rooms.

- Each conversation needs to be supported by a private execution state.
  - Example: each chat room needs to keep track of the posted messages.

- We call this support **session**.

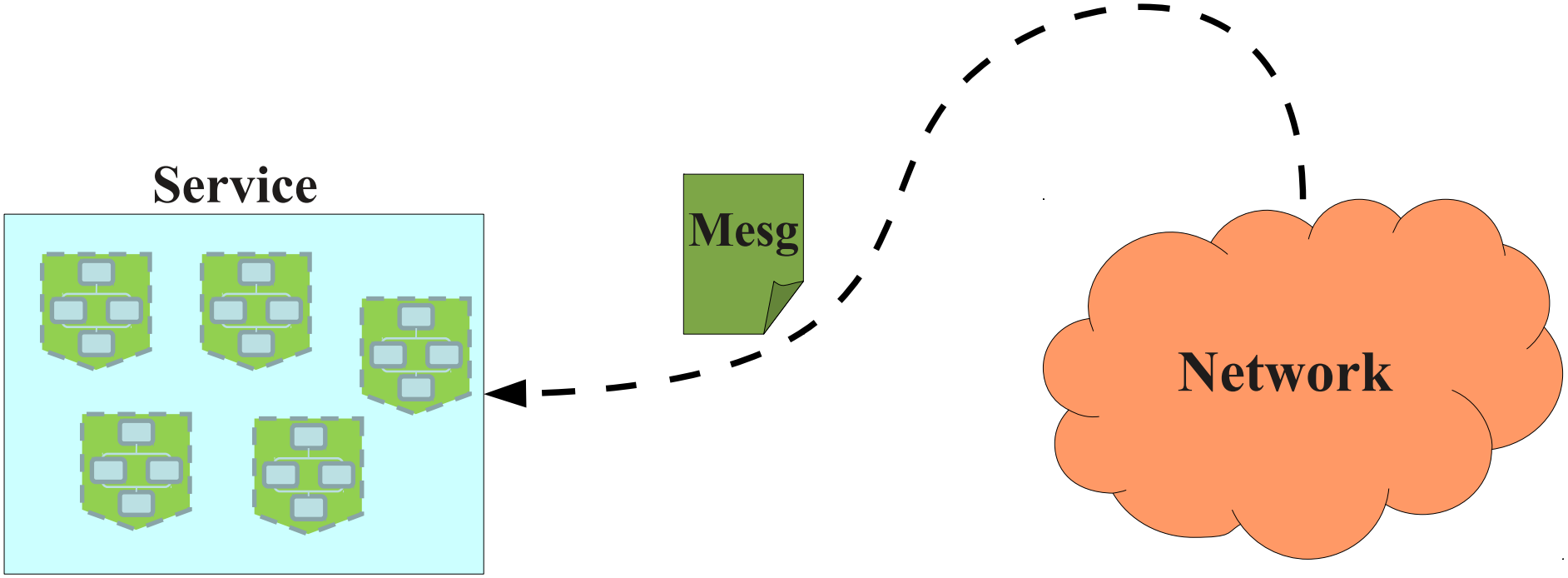
- Sessions are independent of each other: they run in parallel.
  - Some call them **threads** equipped with a **private state**.

- Therefore, a service has many parallel sessions running inside of it:



Message routing

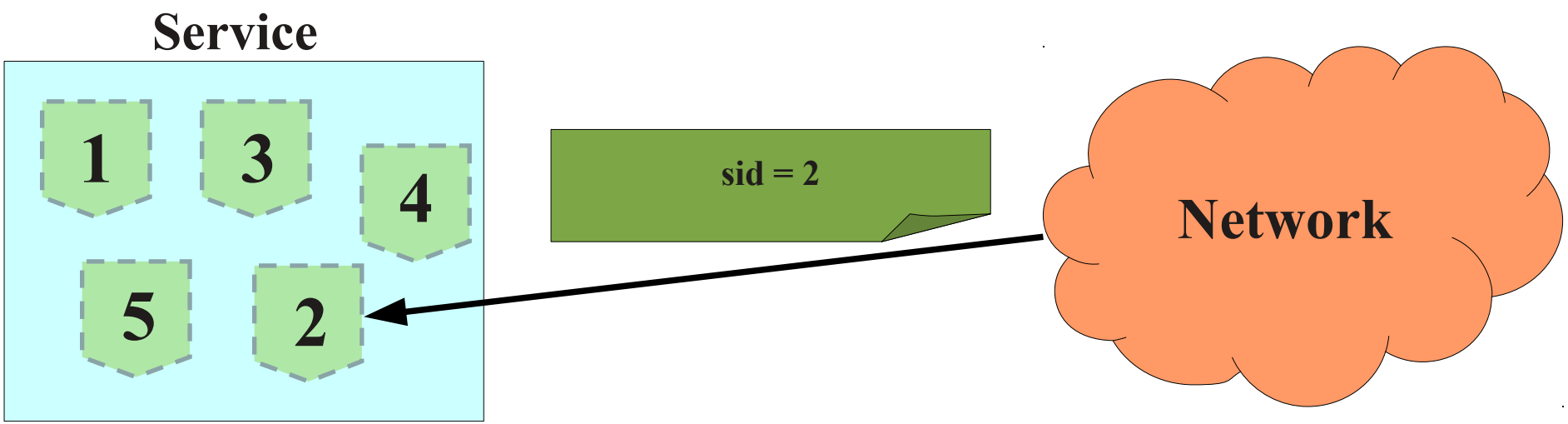
- What happens when a service receives a message from the network?
- We need to assign the message to a session!



- How can we establish which session the message is meant for?

Session identifiers

- A widely used mechanism for routing messages to sessions.
- Each session has a **session identifier** (sid).
- All received messages contain an sid.
- The service gives the message to the session with the same sid.

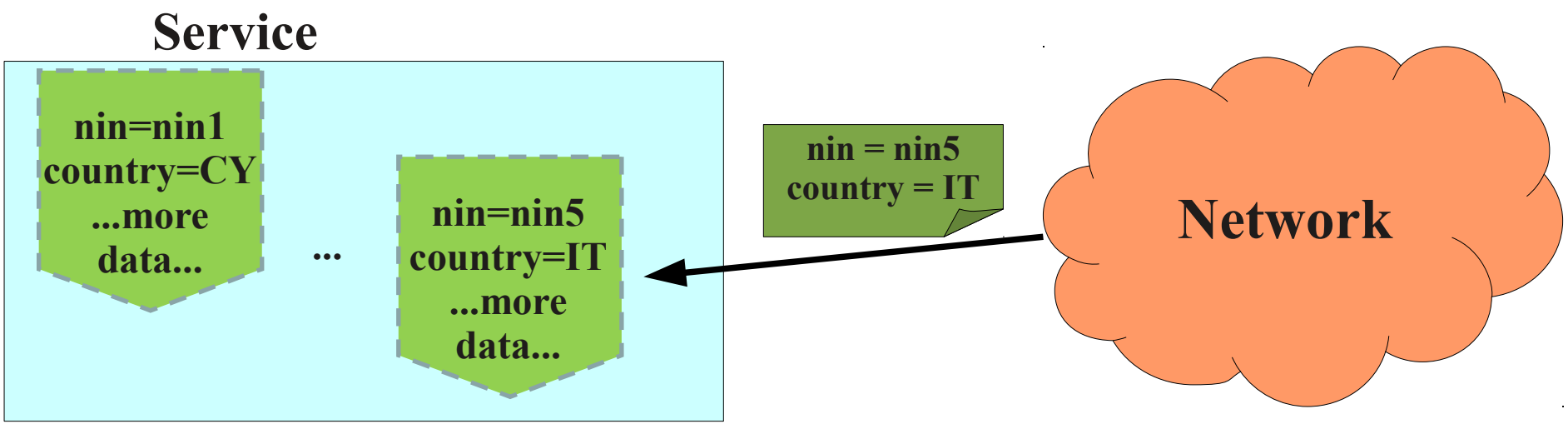


Correlation sets

- A *generalisation* of session identifiers. Introduced in WS-BPEL.

- A session is identified by the **values** of some of its variables.
  - These variables form a **correlation set** (or **cset**).
  - Similar to unique keys in relational databases.

- Example:
  - in a service where we have a session for every person in the world a correlation set could be formed by the national identification number and the country.



Session identifiers VS correlation sets

Session identifiers

- Pros
  - Usually handled by the middleware: hard to make mistakes.
- Cons
  - All clients must send the sid as expected: no support for integration.

Correlation sets

- Pros
  - Programmability of correlation can be used for **integration**.
  - Each cset is a different way of identifying a session: support for **multiparty interactions**.
- Cons
  - Almost totally controlled by the programmer: easy to make mistakes.

Can we reach a compromise?

- In this work we propose a compromise between the two approaches.

- First, we identify some desirable properties for correlation-based programs.

- Then we offer...
  - a **formal model** (process calculus) for a correlation-based programming language, in which we formalise the properties;
  - a **type system** that checks programs for respecting the properties;
  - a **reference implementation** of our framework;
  - a nontrivial real-world **application** inspired by the OpenID protocol.

Introduction

Our solution

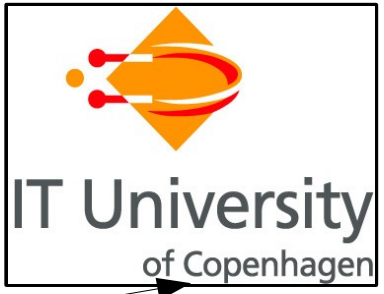
Demo

Conclusions

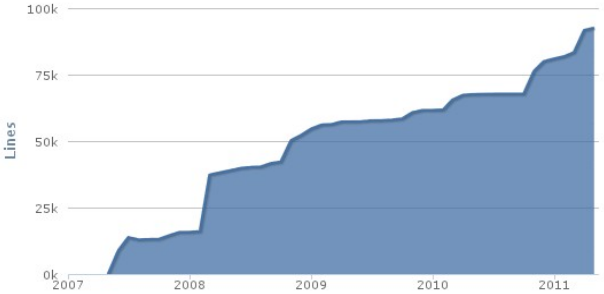
Jolie: Java Orchestration Language Interpreter Engine

- Our work is based on Jolie, a service-oriented programming language.
- Nice logo:

FOCUS Research Team



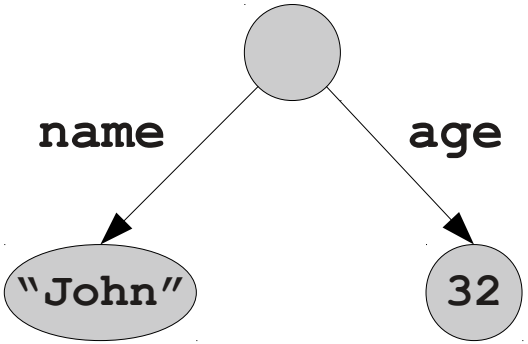
- *Formal foundations* from the Academia.
- Tested and used in the *real world*: italianaSoftware Horsa™ We think. machines do.
- *Open source* (`http://www.jolie-lang.org/`), with a well-maintained code base:



Data and interfaces

- Data is structured as trees.

```
person.name = "John";
person.age = 32;
```



- Data can be typed:

```
type Person {
    .name:string
    .age:int
}
```

- Interfaces are created by associating data types to operations:

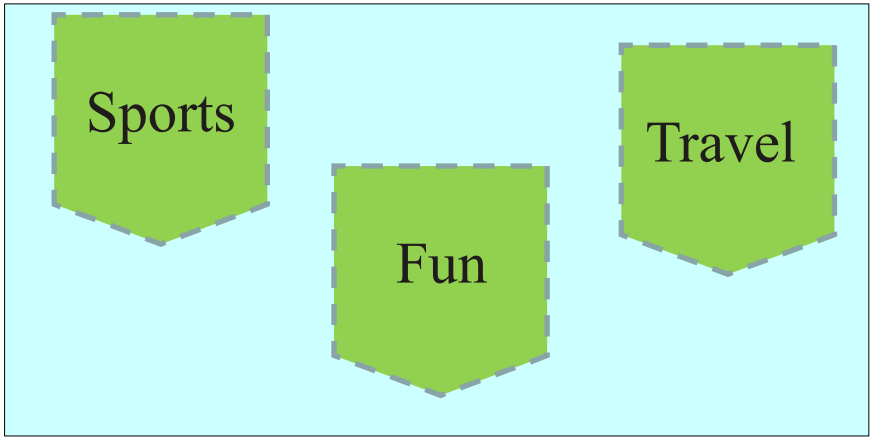
```
interface MyInterface {
RequestResponse:
    getPerson(string)(Person)
}
```

Example: chat service

- We model a chat service handling separate chat rooms. Each room is a session.

```
interface ChatInterface {
RequestResponse:
    openRoom(OpenRequest)(OpenResponse)
OneWay:
    publish(PublishMesg),
    close(CloseMesg)
}
```

Chat service



Session starter

```
main
{
    openRoom( openRequest )( response ) {
        // Create the chat room...
    }; run = true;
    while ( run ) {
        [ publish( message ) ] { println@Console( message.content )() }
        [ close( closeRequest ) ] { run = false }
    }
}
```

Correlating chats

- We want:
  - to publish messages in the right rooms; 1
  - to let the room creator close it, but only her! 2

- So we create two correlation sets:

```
interface ChatInterface {
RequestResponse: openRoom(OpenRequest)(OpenResponse)
OneWay: publish(PublishMesg), close(CloseMesg)
}

cset { name: OpenRequest.room PublishMesg.roomName }   1
cset { adminToken: CloseMesg.adminToken }   2
```

```
main
{
    openRoom( openRequest )( csets.adminToken ) {
        csets.adminToken = new
    }; run = true;
    while ( run ) {
        [ publish( message ) ] { println@Console( message.content )() }
        [ close( closeRequest ) ] { run = false }
    }
}
```

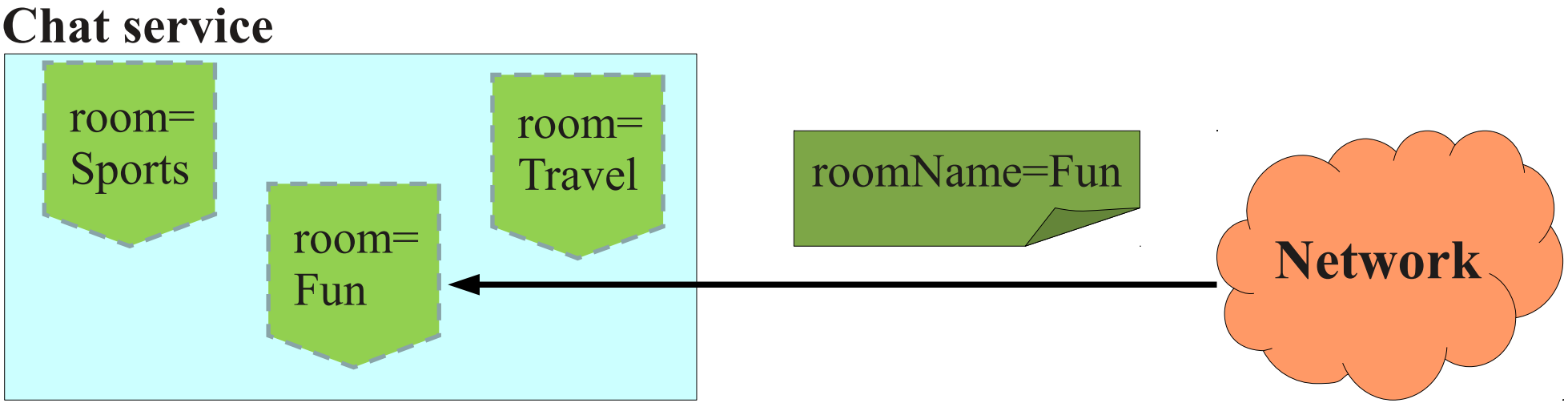
Fresh value generator

Design choices

- **Typed correlation:** correlation sets are declaratively defined on interfaces.
- **Correlation data** is manipulated directly from the program.
- **Aliasing:** a correlation variable may be in different parts of a message.

```
interface ChatInterface {
RequestResponse: openRoom(OpenRequest)(OpenResponse)
OneWay: publish(PublishMesg), close(CloseMesg)
}

cset { name: OpenRequest.room PublishMesg.roomName }
cset { adminToken: CloseMesg.adminToken }
```



Safe chats

- Two properties that we would like to get in our chat service.
- **No ambiguity**: there are *never* two chat rooms sharing name or admin token.
- **No correlation deadlock**: correlation values for an operation must be defined before trying to receive a message for it.

```
main {
    openRoom( openRequest )( csets.adminToken ) {
        csets.adminToken = new      csets.adminToken = 42
    }; run = true;
    while ( run ) {
        [ publish( message ) ] { println@Console( message.content )() }
        [ close( closeRequest ) ] { run = false }
    }
}
```

Ambiguity!!

```
main {
    openRoom( openRequest )( csets.adminToken ) {
        csets.adminToken = new
    }; run = true;
    while ( run ) {
        [ publish( message ) ] { println@Console( message.content )() }
        [ close( closeRequest ) ] { run = false }
    }
}
```

Deadlock!!

General safety properties

- Our formal model and type system guarantee...

- **No ambiguity**: when a message is received, it always correlates with *at most one session*.

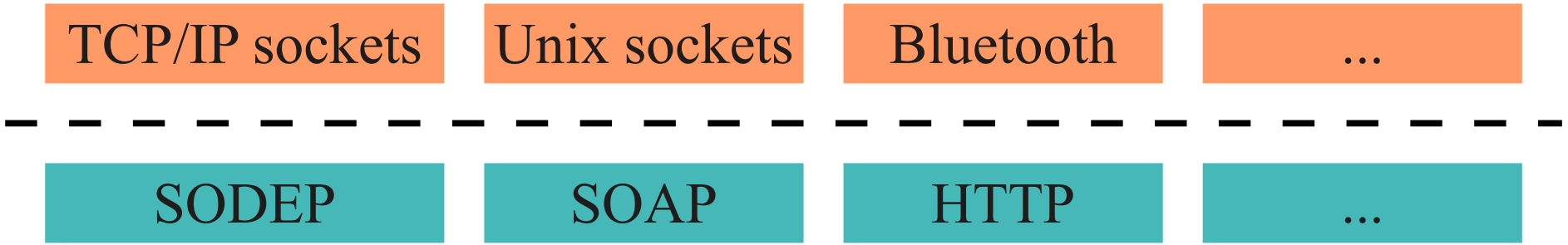
- **Correlation value instantiation**: when a session waits for an input on an operation, the corresponding correlation values have already been instantiated.

- **Message well-formedness**: all message types are linked to a correlation set, and contain all the necessary data specified by it.

- The last two properties give...
- **No correlation deadlock**: correlation does not introduce any new deadlock into the system.

Communication abstraction

- Jolie supports many different communication mediums and data protocols.



- Our implementation exploits this feature to allow correlation values to appear in different places:
  - inside XML documents;
  - as a header of an HTTP message (cookies);
  - etc...

- A same correlation definition can be reused with different protocols transparently!

Introduction

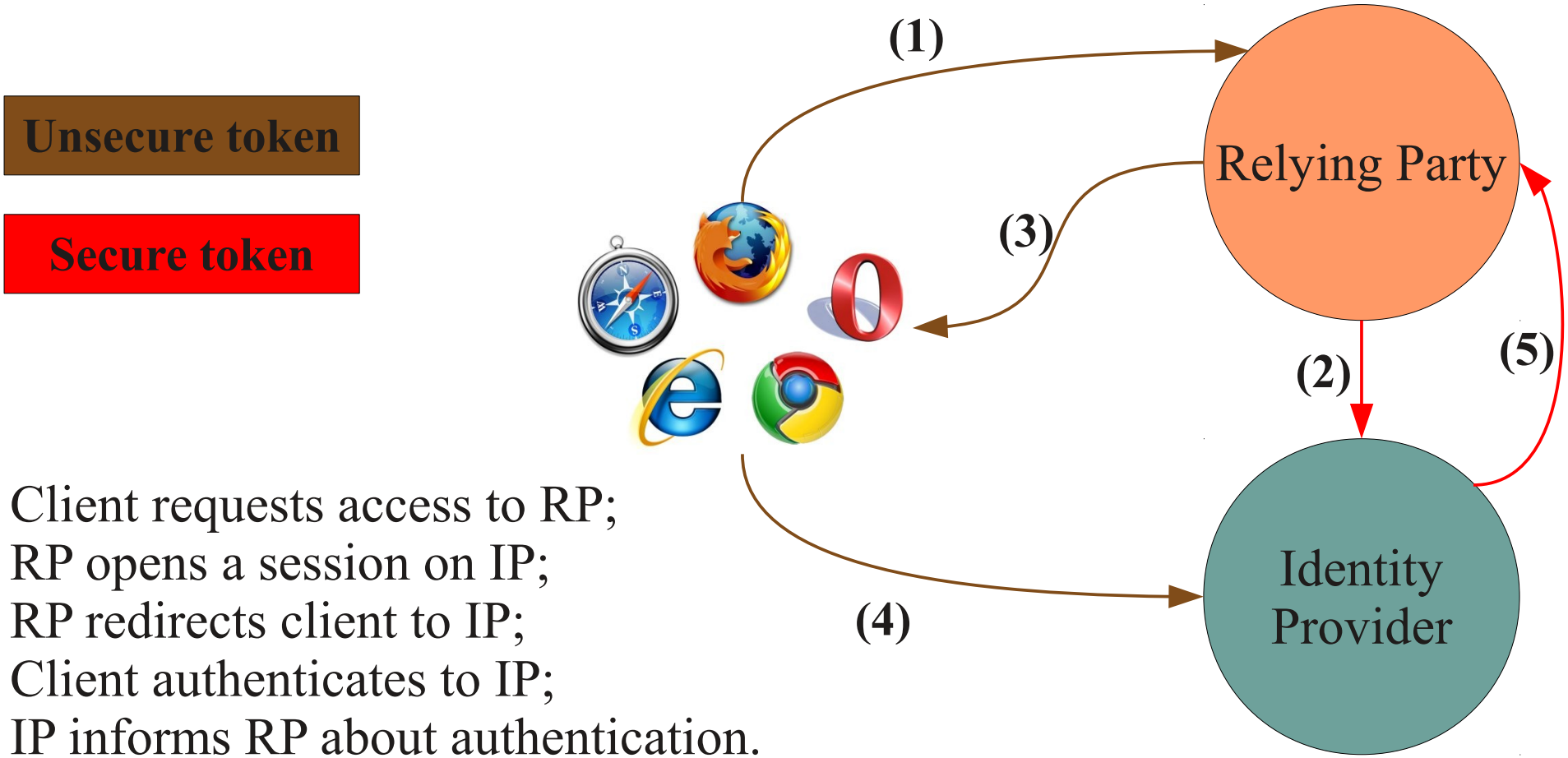
Our solution

Demo

Conclusions

Multiparty session coordination

- A distributed authentication protocol, inspired by OpenID
- A service (called Relying Party) delegates authentication to another service (called Identity Provider).



(1) Client requests access to RP;
(2) RP opens a session on IP;
(3) RP redirects client to IP;
(4) Client authenticates to IP;
(5) IP informs RP about authentication.

Demo

Introduction

Our solution

Demo

Conclusions

Conclusions

- Correlation sets are a powerful mechanism for programming multiparty sessions.

- We showed how their programming should be disciplined.

- Future work:
  - behavioural types (session types);
  - security (e.g., *compromised* administration token);
  - use correlation for more complex routing, e.g., publish/subscribe.

Questions?