Jolie

A service-oriented programming language

Ivan Lanese

(Original slides from Fabrizio Montesi)

Dynamic binding

- In a SOA, a fundamental mechanism is that of *service discovery*.
- A service dynamically (at runtime) discovers the location and a protocol for communicating with another service.
- In JOLIE we obtain this by manipulating an output port as a variable.

```
outputPort Calculator {
Interfaces: CalculatorInterface
}
main
{
    Calculator.location = "socket://localhost:8000/";
    Calculator.protocol = "sodep";
    request.x = 2;
    request.y = 3;
    sum@Calculator( request )( result )
}
```

- Type for bindings is

```
type Binding:void {
        .location:string
        .protocol?:string { ? }
}
```

Multiple executions: processes

- The calculator works, but it terminates after executing once.
- We want it to keep going and accept other requests.
- We introduce **processes**.
- A process is an **execution instance** of a service **behaviour**.
- In JOLIE, processes can be executed **concurrently** or **sequentially**.
- Processes should either start with an input or be a choice among inputs.
- Initialization code can be inserted in an init block, run just once

```
execution { concurrent }
```

```
execution { sequential }
```

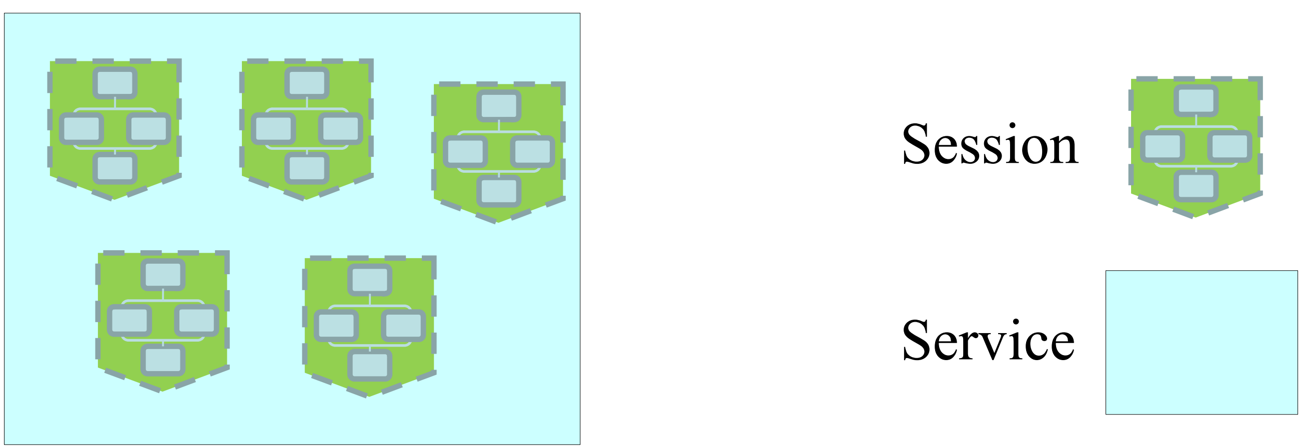
```
sum( request )( response ) {
    response = request.x + request.y
};
println@Console( message )()
```

```
sum( request )( response ) {
    response = request.x + request.y
};
println@Console( response )()
```

```
sum( request )( response ) {
    response = request.x + request.y
};
println@Console( response )()
```

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 (with the concurrent execution modality).
- Therefore, a service may have many parallel sessions running inside of it:



Shared state

- Each session has its private state

- Shared state can be defined using the keyword global
  - global.var is a global variable

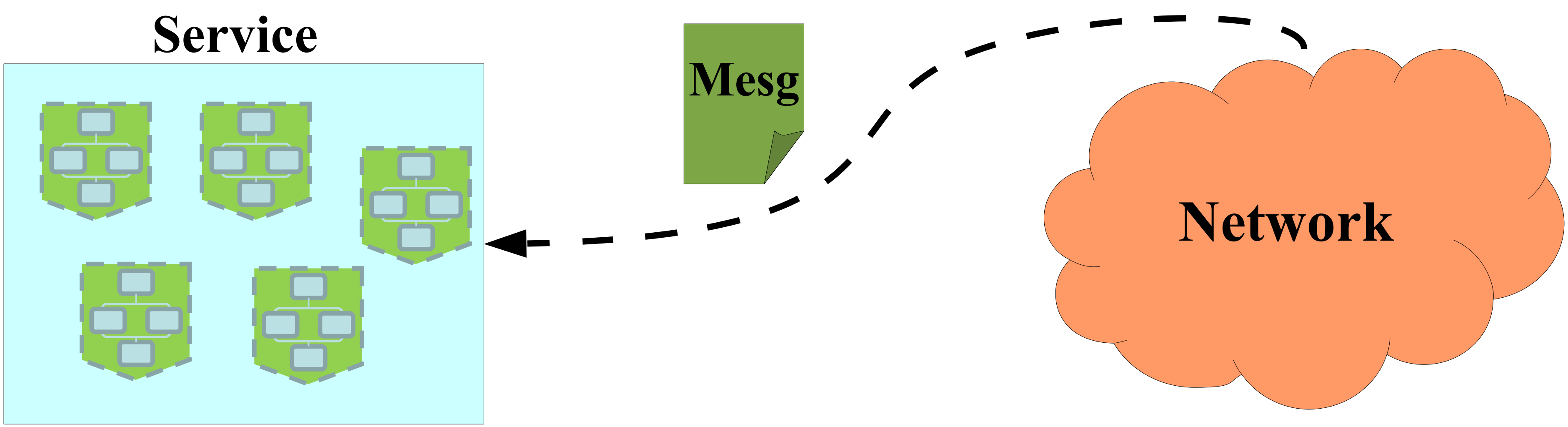
- Accesses to shared state should be synchronized

```
synchronized( id ){
    //code
}
```

- All the synchronized blocks with the same id (id is an identifier, i.e. a legal name) act as a unique critical section

Message routing

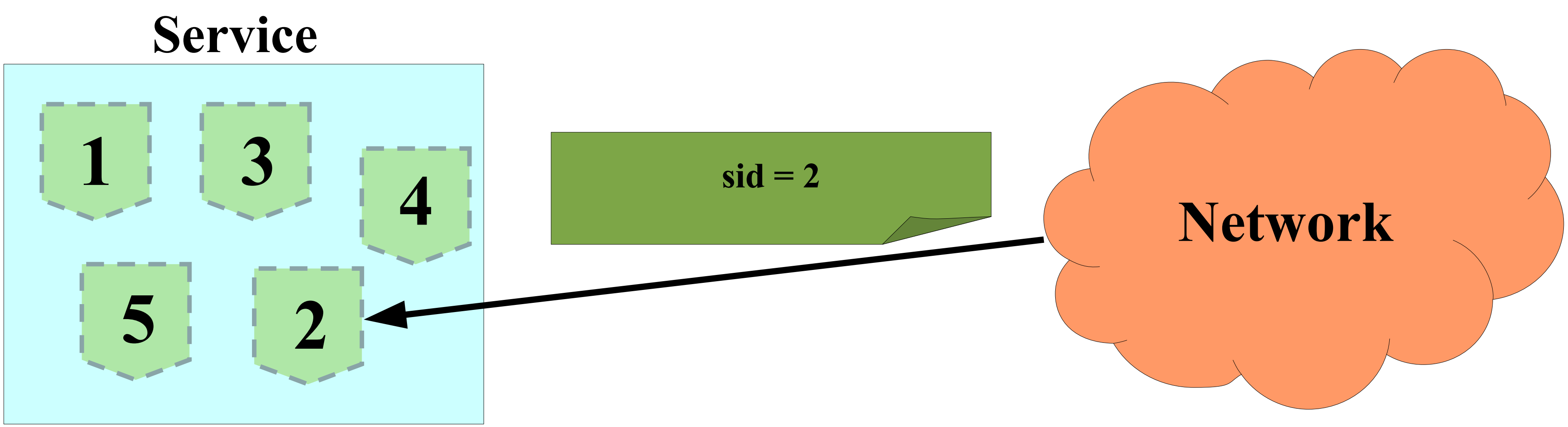
- What happens when a service receives a message from the network?
- We need to assign the message to a session!



- 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 a sid.
- The service gives the message to the session with the same sid.
- If no session has the required sid, a new session with the required sid is created

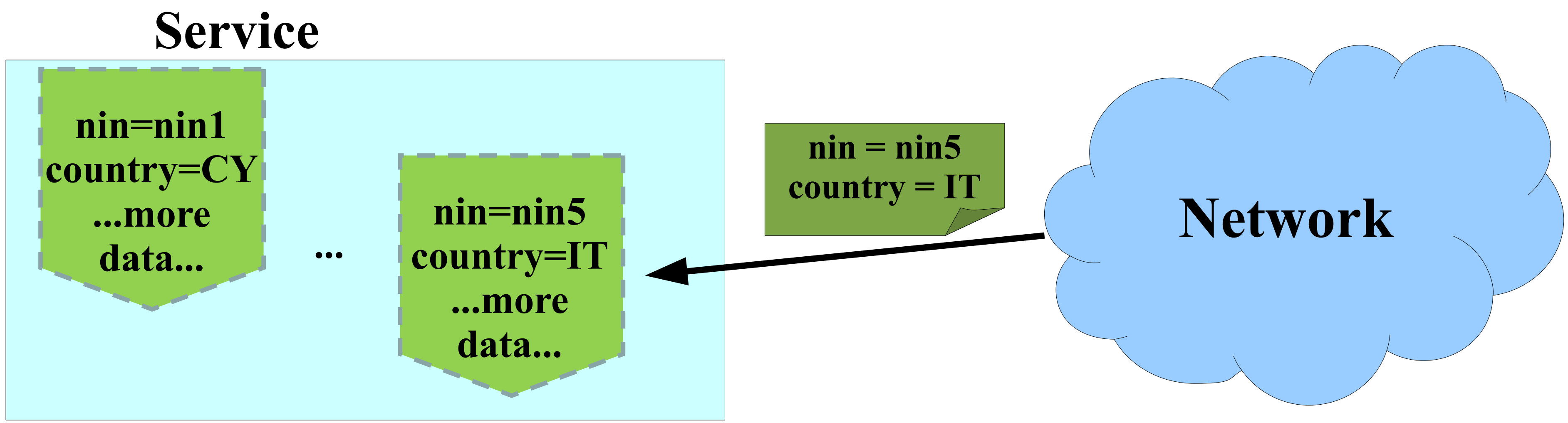


Correlation sets

- A *generalisation* of session identifiers.

- 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: easier to make mistakes.

Example: print server

```
type LoginRequest: void {
    .name: string
}

type OpMessage: void{
    .sid: string
    .message?: string
}

interface PrintInterface {
    RequestResponse: login(LoginRequest)(OpMessage)
    OneWay: print(OpMessage), logout(OpMessage)
}
```

```
cset {
    sid: OpMessage.sid
}

main
{
login( request )( response ){
   username = request.name;
   response.sid = csets.sid = new
   };
   // code
}
```

```
 main
{
request.name = "Pippo";
login@PrintService(request)(response);
    opMessage.sid = response.sid;
    // if user wants to print
        opMessage.message="my Message";
        print@PrintService( opMessage );
    // else he wants to logout
        logout@PrintService( opMessage )
}
```

Fresh value generator (of type string)

What happens when you use correlation sets

- When a message arrives, it is sent to the session it correlates with
- If it correlates with no session:
  - If it targets the initial operation, a new session is created
    - If the operation has a correlation set, the corresponding variables are initialised
  - Otherwise, a correlation error occurs
- The variable **`csets`** stores the values composing the correlation sets
- It can be initialized
  - By the starting operation
  - Directly by the code

More correlation sets

- A single correlation variable may link to many aliases in different message types
  - An alias shows where to find the value to be correlated in the incoming message
  - E.g., login.name

- A correlation set may be composed by different correlation variables
  - All of them have to be matched

- A service may have different correlation sets (using different cset constructs)
  - Each operation may refer to just one

```
cset {
    correlationVariable_1: alias_11 alias_12 …,
    correlationVariable_2: alias_21 alias_22 …
}
```

Exercise

- We design a SOA for handling bank accounts.
- An user can login.
- After login, in a private session, he can ask for withdrawal, deposit, or account report.
- At the end of the session he can log out.

- A bank supports many concurrent clients.