TuCSoN examples with Prolog and JADE agents

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Giovanni Ciatto — giovanni.ciatto@unibo.it
Andrea Omicini — andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM – Università di Bologna a Cesena

Academic Year 2017/2018

What We Will Learn

1. Setting your environment up
2. Wetting your appetite
3. Prolog agents chatting over TuCSoN
4. Responder agent: the control loop
5. Master-workers agents in Prolog
6. TuCSoN integration with JADE

Setting your environment up I

1. Clone the provided git repository from Bitbucket into the `tucson-pl-jade` directory and move into the cloned directory:

```
$ mkdir tucson-pl-jade
$ cd tucson-ex
$ git clone https://bitbucket.org/gciatto_unibo/tucoson-example
$ cd tucson-example
```

Setting your environment up II

2. Run Gradle in order to automatically download the required dependencies and setup the project
   - `$ gradle build`
   - Consider using the lab PC instead of your own one if **any** of the following conditions are met:
     - JDK ≥ 8 is **not** installed on your PC
     - Your environment variables `JAVA_HOME` or `PATH` are **not** properly set
     - Your Wi-Fi connection is too slow to download the required dependencies
   - If Gradle is not installed on your PC, you can use the provided Gradle wrapper. This means you have to write `.\gradlew` on Windows or `./gradlew` on Posix, instead of `gradle`

Setting your environment up III

3. You can configure the project to be easily imported *as a Gradle project* by your favourite IDE by running the following commands
   - $ `gradle idea` (for IntelliJ Idea)
   - $ `gradle eclipse` (for Eclipse)

Setting your environment up IV

4. We provided you the following pre-configured Gradle tasks to be executed with the '`gradle <task>`' syntax:

   `runTucson`: runs a TuCSoN node on the default port (20504, unless another one is specified by means of the `-portno` argument). Note that the node will prevent you to re-use the same shell until its termination

   `runInspector`: runs a TuCSoN Inspector GUI enabling users to inspect the tuples contained within any tuple centre on a given TuCSoN node

   `runIDE`: runs a tuProlog IDE extended with TuCSoN interoperability facilities and ready to interact with any tuple centre.

Setting your environment up V

`runPlatform`: starts a JADE platform already configured to make agents executed on top of it interoperable with the TuCSoN middleware

`runPlatformWithAgents`: starts a JADE platform already configured to make agents executed on top of it interoperable with the TuCSoN middleware. The agents spawned on platform startup are specified within the `gradle.properties` file, as the value corresponding to the `agents` key

Wetting your appetite I

1. Execute the `runIDE` task: a very special tuProlog IDE should appear, showing the following code:

```
:-initialization((agent_name(N), node_address(A),
    node_port(P), acquire_acc(N, A, P))).

agent_name(agent_<user name>).
node_address('localhost').
node_port(20504).
```

2. If the `agent_name` is not a well formed Prolog atom, please fix it
   e.g. `agent_giovanni.ciatto2` → `agent_giovanni_ciatto`
3. Query the teacher for his/her TuCSoN node address & port and replace them accordingly within the aforementioned code
   - **You can keep using `localhost` if you are working at home**

Wetting your appetite II

4. Try write TuCSoN primitive invocations within the IDE's goal text box and then run the engine, like for instance:

```
?- out(something).
?- in(something).
?- inp(something_else).
?- no(something_else).
```

   - Why is this working without any initial configuration?
   - Where are you acquiring an ACC?

Prolog agents chatting over TuCSoN I

1. Copy&paste the `res/chat.pl` script (shown below) into the tuProlog IDE:

```
agent_name($agent_name). % TODO insert <surname>_<name> here
node_address('localhost').    node_port(20504).

send_message(Message, Receiver) :-
  agent_name(Me),
  out(message(Receiver, Message, Me)).

receive_message(Message, Sender) :-
  agent_name(Me),
  in(message(Me, Message, Sender)).
```

2. Fix the agent name (e.g. with your own <surname>_<name>) and the node address & port with the teacher's ones

Prolog agents chatting over TuCSoN II

3. Once everybody has properly configured their IDE, you can:
   - send a message to any colleague of yours...
     ```
     ?- send_message("Your message here", recipient_agent_id).
     ```
   - ... and wait for their answers
     ```
     ?- receive_message(MessageTemplate, Sender).
     ```

- Do you understand how does the interaction protocol works?
- What if the recipient's `receive_message/1` invocation occurs **after** the sender's `send_message/1` invocation?

Exercises – Expected time 5-10min

Polling messages

Implement a `poll_message(?MessageTemplate, ?Sender)` predicate enabling agents to check whether a message has been received or not, **without consuming it**

Public messages

Implement a `public_message(?Payload)` predicate enabling agents to publish a message which can be consumed by everyone

Responder agent: the control loop I

1. Copy&paste the `res/responder.pl` script (shown below) into the tuProlog IDE:

```
agent_name(responder).  % TODO insert responder_<surname> here
node_address('localhost').    node_port(20504).

agent_execution :-
  agent_name(MyName), node_address(Address), node_port(Port),
  acquire_acc(MyName, Address, Port),
  see(stdin), % <-- open standard input
  (agent_loop; true), % <-- PAY ATTENTION HERE
  seen(stdin), % <-- close standard input
  release_acc, !.

agent_loop :- !, agent_loop_step, agent_loop. % <-- recursion

agent_loop_step :-
  agent_name(MyName),
  in(message(MyName, Payload, Sender)),
  read(Response), % <-- read a dot-terminated term from standard input
  out(message(Sender, Response, MyName)).
```

2. Fix the agent name (e.g. with your own `responder_<surname>`) and the node address & port according to the teacher's ones

Responder agent: the control loop II

3. You just need to query `?- agent_execution.` to run the agent
4. Now, whenever someone sends a message to your responder agent, it will prompt the answer from the "Input" tab
   - you may need another tuProlog IDE to send messages to the responder (or any other TuCSoN agent)
   - the responder automatically takes care of setting up the recipient of your answer

- Notice the responder agent can interact with the agent from the previous exercise: the only thing they assume about interacting peers is their capability of producing/consuming tuples having the form `message(Recipient, Payload, Sender)`

Exercises – Expected time 5min

Exit command

Make your responder agent terminate when receiving a message having the `exit` atom as its payload.

Handling intensive computations

The master-workers pattern

- Some problems can be split in several quite trivial sub-problems – **tasks** – by an agent in charge of splitting problems—the **master**

  e.g. counting the amount of vowels within a single verse
- Tasks can be assigned to an arbitrary amount of **workers**, trying to *execute* them producing **partial results**
- Once all tasks have been executed, the master must simply **aggregate** all the partial results
- Such problems can be easily faced with TuCSoN tuple centres

The problem faced in the following

How many vowels does the (first poem of the) Divine Comedy have?

The master agent in Prolog

1. Copy&paste the `res/master.pl` script (shown below) into the tuProlog IDE:

```
agent_execution :- % ... agent_loop ...

agent_loop :-
  divine_comedy(Lines), % <-- get a list of verses
  partition(Lines, 3, Tasks), % <-- group verses into triplets
  schedule_tasks(Tasks),
  length(Tasks, N),
  aggregate_results(N, Result).

schedule_tasks([]).
schedule_tasks([T | Ts]) :-
  out(task(count_vowels, T)), % <-- publish the task
  schedule_tasks(Ts).

aggregate_results(N, R) :- aggregate_results(N, R, 0).
aggregate_results(0, R, R).
aggregate_results(N, R, Accumulator) :-
  in(result(count_vowels, Lines, Partial)), % <-- retrieve the partial result
  Accumulator1 is Accumulator + Partial, N1 is N - 1,
  aggregate_results(N1, R, Accumulator1).
```

2. Fix the node address & port according to the teacher's ones

The worker agent in Prolog

3. Copy&paste the `res/worker.pl` script (shown below) into the tuProlog IDE:

```
agent_execution :- % ... agent_loop ...

agent_loop :- !, agent_loop_step, agent_loop. % <-- recursion

agent_loop_step :-
  in(task(Name, Args)), % <-- retrieve a task
  handle_request(Name, Args).

% Handle the known task "count_vowels"
handle_request(count_vowels, Lines) :-
  count_vowels(Lines, R),
  out(result(count_vowels, Lines, R)). % <-- publish partial result

% Handle unknown tasks
handle_request(Name, Args) :- out(task(Name, Args)).
```

4. Fix the node address & port according to the teacher's ones

Wiring up master and worker(s)

5. Elect *a number of* colleagues in charge of running a worker agent
   - Run them now!
6. Elect *a single* colleague in charge of running the master agent
   - Run it now!
7. Watch your logs!

TuCSoN integration with JADE

Can JADE agents interact by means of a TuCSoN tuple centre?

Yes, in principle, but:

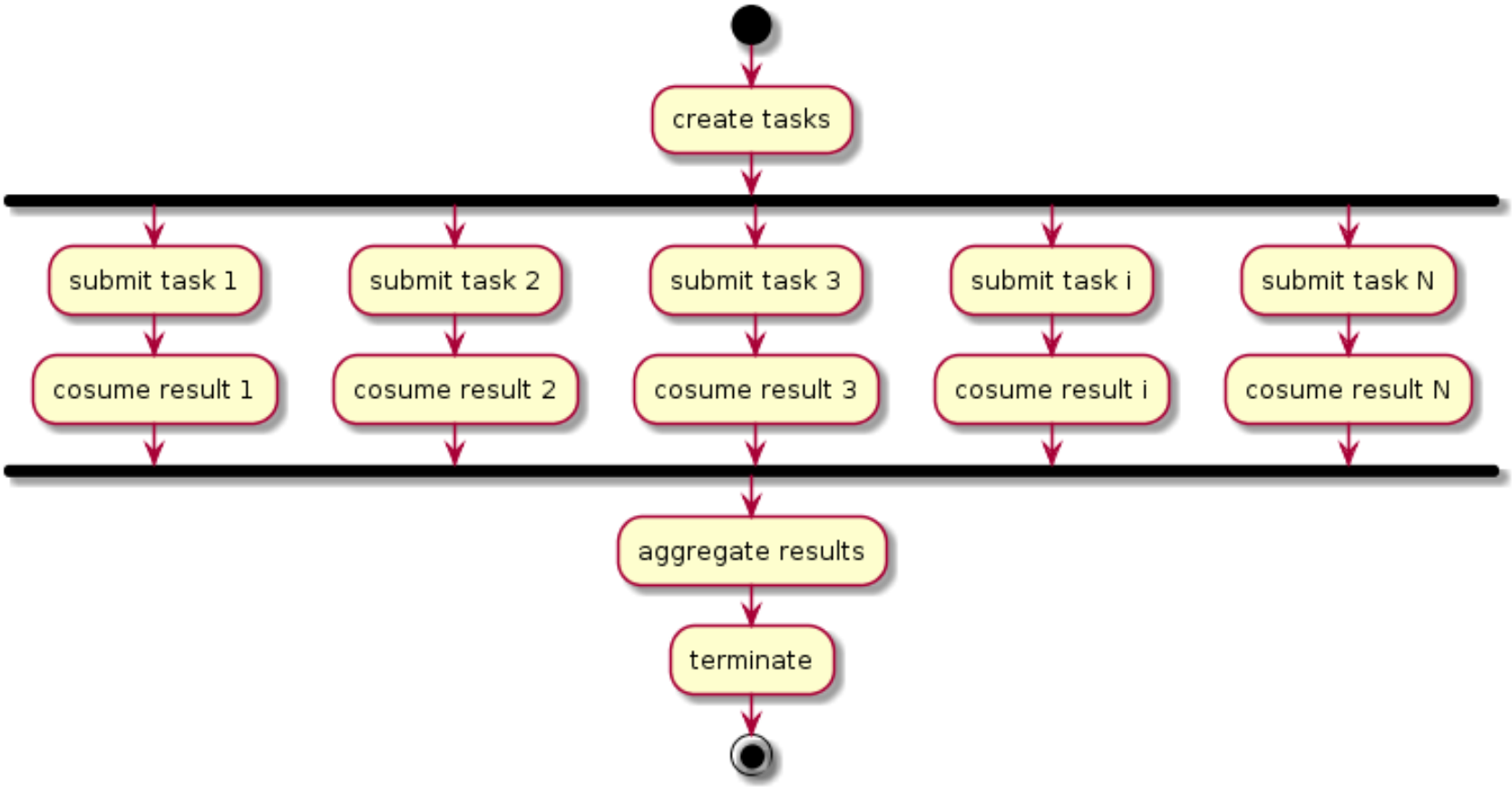
- primitives subject to LINDA's suspensive semantics actually block the thread invoking them, according to the current implementation
- this is indeed a problem in JADE where blocking the agent's thread implies blocking all its behaviors
- the TuCSoN4JADE project[a] faces and solves this problem harmonizing LINDA's suspensive semantics with JADE behaviors

[a] http://apice.unibo.it/xwiki/bin/view/TuCSoN/WebHome

TuCSoN primitives invocation in JADE

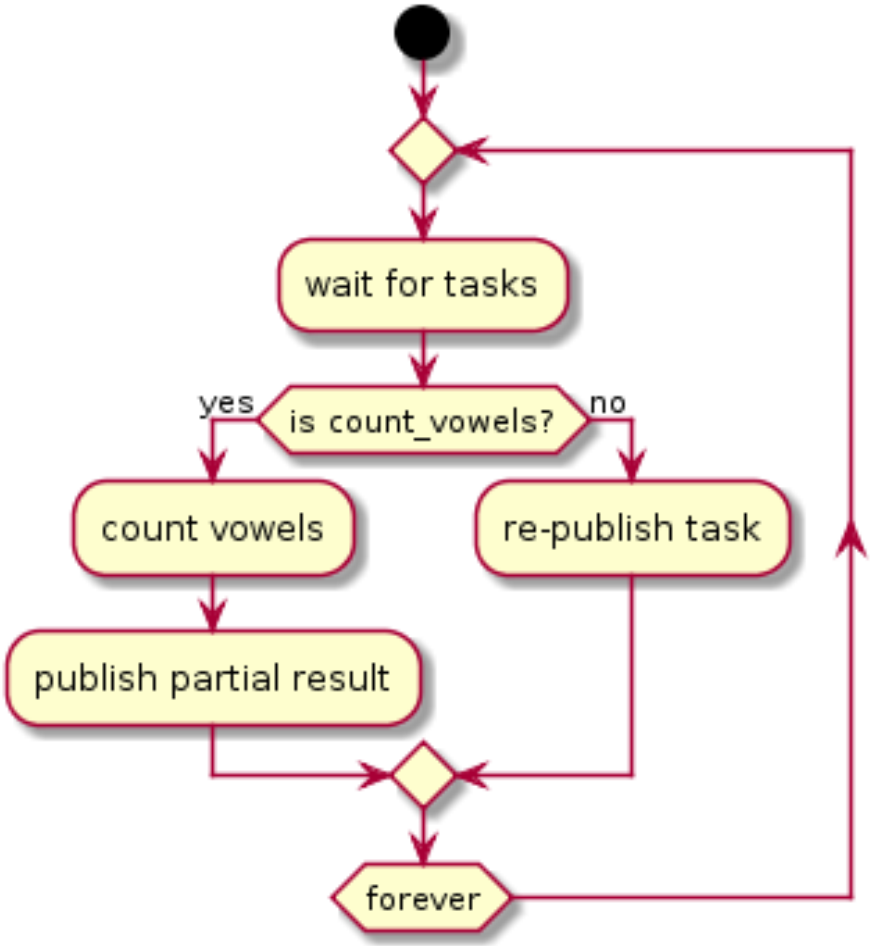
```
public class InvokeTucsonOperation extends Behaviour {

    private final BridgeToTucson bridge;
    private final AbstractTucsonOrdinaryAction operation;
    private TucsonOpCompletionEvent result;

    @Override
    public final void action() {
        result = bridge.synchronousInvocation(op, null, this);

        if (result != null) {
            handleResult(result);
        } else {
            block();
        }
    }

    @Override
    public final boolean done() {
        return result != null;
    }

}
```

The master agent in JADE



(Look at class `it.unibo.ds.jade.tucson4jade.Master`)

The worker agent in JADE



(Look at class `it.unibo.ds.jade.tucson4jade.Worker`)

Running the JADE master-workers examples

1. Execute the `runPlatformWithAgent` Gradle task enabling both the master and the worker agents from the `gradle.properties` file
2. Execute the `runPlatformWithAgent` Gradle task enabling the master agent and **a number of** worker agents from the `gradle.properties` file
3. Execute the `runPlatformWithAgent` Gradle task just enabling the master agent and then run a number of **Prolog worker** agents
   - are they interoperable?
4. Execute the `runPlatformWithAgent` Gradle task just enabling the worker agent and then the **Prolog master** agent
   - are they interoperable?

TuCSoN as a mean for interoperability

Loose coupling

- Agents can coordinate by means of TuCSoN without the need of knowing each other, sharing the same technology or space, being connected at the same time etc.
- TuCSoN only requires agents to be able to produce/read/consume logic tuples

$\Rightarrow$ TuCSoN makes interoperability easy

TuCSoN examples with Prolog and JADE agents

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Giovanni Ciatto — giovanni.ciatto@unibo.it
Andrea Omicini — andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM – Università di Bologna a Cesena

Academic Year 2017/2018