Tuple-based Coordination with TuCSoN

Distributed Systems / Technologies
Sistemi Distribuiti / Tecnologie

Stefano Mariani   Andrea Omicini
s.mariani@unibo.it   andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018

1. TuCSoN Basics
2. TuCSoN Advanced
3. TuCSoN Extensions

Disclaimer

- most of the following slides are adapted from the official TuCSoN guide
- the TuCSoN guide is available at
  `http://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide`

Next in Line. . .

Focus on. . .

Tuple Centres Spread over the Network (TuCSoN)

TuCSoN model [Omicini and Zambonelli, 1999]

TuCSoN is a model for the coordination of distributed processes, as well as of autonomous agents

References

- main page http://tucson.unibo.it/
- Bitbucket http://bitbucket.org/smariani/tucson/
- FaceBook http://www.facebook.com/TuCSoNCoordinationTechnology

Basic Entities

- TuCSoN agents are the *coordinables*
- ReSpecT tuple centres are the *coordination media* [Omicini and Denti, 2001]
- TuCSoN nodes represent the basic *topological abstraction*, which host the tuple centres
- agents, tuple centres, and nodes have *unique identities* within a TuCSoN system

System view

Roughly speaking, a TuCSoN system is a collection of agents and tuple centres *working together* in a (possibly) distributed set of nodes

Basic Interaction

- since agents are *pro-active* entities whereas tuple centres are (mostly) *reactive*, the coordinables need *coordination operations* in order to act over the coordination media
- such operations are built out of the TuCSoN coordination language, defined by the collection of TuCSoN coordination primitives that agents can use to interact — by exchanging tuples
- tuple centres provide the shared space for *tuple-based communication* (tuple space), along with the programmable behaviour space for *tuple-based coordination* (specification space)

System view

Roughly speaking, a TuCSoN system is a collection of agents and tuple centres *coordinating* in a (possibly) distributed set of nodes

Focus on. . .

Nodes

- each node within a TuCSoN system is *univocally identified* by the pair $< NetworkId, PortNo >$, where
  - *NetworkId* is the IP number of the device hosting the node
  - *PortNo* is the port number where the TuCSoN *coordination service* listens incoming requests
- correspondingly, the abstract syntax of TuCSoN nodes identifiers hosted by a networked device *netid* on port *portno* is

$$netid : portno$$

e.g. `localhost : 20504`

! *actually, this is also the concrete syntax used by* TuCSoN *to parse nodes ID*

Tuple Centres

- an admissible name for a tuple centre is *any* Prolog-like, first-order logic *ground term* [Lloyd, 1984]
- each tuple centre is uniquely identified by its admissible name associated to the node identifier
- hence the TuCSoN full name of a tuple centre *tname* on a node *netid : portno* is

$$tname\ @\ netid : portno$$

e.g. `default @ localhost : 20504`

Agents

- an admissible name for an agent is *any* Prolog-like, first-order logic ground term, too
- when it *enters* a TuCSoN system, an agent is assigned a *universally unique identifier* (UUID)

UUID http://docs.oracle.com/javase/8/docs/api/java/util/UUID.html

- if an agent *aname* is assigned UUID *uuid*, its full name is

aname : *uuid*

e.g. `stefano : 4baad505-ad2f-4ac4-b30b-bc3705a2c87a`

Focus on. . .

Coordination Language

- the TuCSoN coordination language allows agents to interact with tuple centres by executing *coordination operations*
- TuCSoN provides coordinables with coordination primitives, allowing agents to read, write, consume tuples in tuple spaces
- coordination operations are built out of coordination primitives and of the communication languages:
  - the tuple language
  - the tuple template language

! *in the following, whenever unspecified, we assume that `Tuple` belongs to the tuple language, and `TupleTemplate` belongs to the tuple template language*

Tuple & Tuple Template Languages

- given that the TuCSoN coordination medium is the *logic-based* ReSpecT tuple centre, both the tuple and the tuple template languages are logic-based, too
- more precisely
  - any first-order logic Prolog atom is an admissible TuCSoN tuple. . .
  - . . . and an admissible TuCSoN tuple template

Coordination Operations

- any TuCSoN *coordination operation* is invoked by a source agent on a target tuple centre, which is in charge of its execution
- any TuCSoN operation has two phases

  invocation — the *request* from the source agent to the target tuple centre, carrying all the information about the invocation

  completion — the *response* from the target tuple centre back to the source agent, including all the information about the operation execution by the tuple centre

Abstract Syntax

- the abstract syntax of a coordination operation op invoked on a target tuple centre *tcid* is

  tcid ? *op*

  where `tcid` is the tuple centre *full name*
- given the structure of the full name of a tuple centre, the *general abstract syntax* of a TuCSoN coordination operation is

  tname @ *netid* : *portno* ? *op*

e.g. `default @ localhost : 20504 ? out(t(hi))`

! *actually, this is also the concrete syntax used by* TuCSoN *to parse coordination operations, even inside* ReSpecT *reactions*

Focus on. . .

Coordination Primitives

The TuCSoN coordination language provides the following 9 basic coordination primitives to build coordination operations

- `out` to put a tuple in the target tuple centre
- `rd`, `rdp` to read a tuple matching a given tuple template in the target tuple centre
- `in`, `inp` to withdraw a tuple matching a given tuple template from the target tuple centre
- `no`, `nop` to check absence of tuples matching a given tuple template in the target tuple centre
- `get` to read all the tuples in the target tuple centre
- `set` to overwrite the set of tuples in the target tuple centre

Focus on...

Node

TuCSoN node

A TuCSoN node is characterised by the networked device hosting the service and by the network port where the TuCSoN service listens to incoming requests

Multiple nodes on a single device

Many TuCSoN nodes can run on the same networked device, as long as each one is listening on a different port

Default Node

Default port

The default port number of TuCSoN is 20504

- so an agent can invoke operations of the form

$$tname\ @\ netid\ ?\ op$$

  without specifying the node port number *portno*—if the agents intends to invoke operation *op* on the tuple centre *tname* of the default node *netid* : 20504, hosted by the networked device *netid*
- any other port can be used for a TuCSoN node listening service—we will see how to change it in a few slides

Default Tuple Centre

Default tuple centre

Every TuCSoN node defines a default tuple centre, which responds to any operation invocation received by the node that do not specify the target tuple centre

`default`

The *default tuple centre* of any TuCSoN node is named `default`

- as a result, agents can invoke operations of the form

  $$@\ netid : portno\ ?\ op$$

  without specifying the tuple centre name *tname*, thus meaning `default` as the tuple centre name

Defaults

- by combining the notions of default node and default tuple centre, the following invocations are also admissible for any TuCSoN agent running on a device `netid`:
  - *: portno ? op*
    invoking operation *op* on the `default` tuple centre of node *netid : portno*
  - *tname ? op*
    invoking operation *op* on the *tname* tuple centre of default node *netid* : 20504
  - *op*
    invoking operation *op* on the `default` tuple centre of default node *netid* : 20504

Focus on. . .

Technology Requirements

- TuCSoN is a Java-based middleware (Java 7 is enough)
- TuCSoN is also Prolog-based: it is based on the tuProlog Java-based technology for
  - first-order logic tuples
  - primitives & identifiers parsing
  - ReSpecT specification language & virtual machine

! *last digits in* TuCSoN *version number* (`TuCSoN-1.12.0.0301`) *are for the tuProlog version, hence tuProlog version* 3.0.1 *now*

Java & Prolog Agents

TuCSoN middleware provides

Java API for using TuCSoN coordination services from Java programs

- package `alice.tucson.api.*`

Prolog API for using TuCSoN coordination services from tuProlog programs

- `alice.tucson.api.Tucson2PLibrary` enables tuProlog agents to use TuCSoN primitives
- use directive `:-load_library('path-to-Tucson2PLibrary')` to load the library, where `path-to-Tucson2PLibrary` is a string atom representing the path to the Tucson2PLibrary

Service

- given any networked device running a Java VM, a TuCSoN node can be started to provide TuCSoN coordination services

```
java -cp libs/tucson.jar:libs/2p.jar alice.tucson.service.TucsonNodeService
    -portno 20505
```

- the node service is in charge of
  - listening to incoming operation invocations
  - dispatching them to the target tuple centre
  - returning the operations completion to the source agent

Let's try!

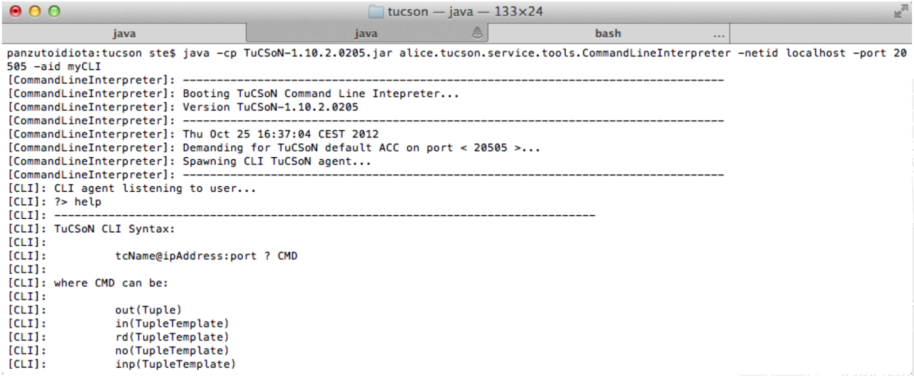
1. open a console, position yourself into the folder where `tucson` and `2p` jars are, then type the command above—on Windows, replace “:” with “;”
2. try to launch another TuCSoN node on a different *portno*

Focus on...

Command Line Interpreter (CLI) I

- shell interface for humans

```
java -cp libs/tucson.jar:libs/2p.jar
    alice.tucson.service.tools.CommandLineInterpreter
    -netid localhost -portno 20505 -aid myCLI
```



Command Line Interpreter (CLI) II

CLI Syntax

⟨*TucsonOp*⟩	::=	⟨*TcName*⟩ `@` ⟨*IpAddress*⟩ `:` ⟨*PortNo*⟩ `?` ⟨*Op*⟩
⟨*TcName*⟩	::=	Prolog ground term
⟨*IpAddress*⟩	::=	`localhost` | IP address
⟨*PortNo*⟩	::=	port number
⟨*Op*⟩	::=	`out(T)` | `in(TT)` | `rd(TT)` | `no(TT)` | `inp(TT)` | `rdp(TT)` | `nop(TT)` | `get()` | `set([T1,...,Tn])` | `out_all(TL)` | `in_all(TT,TL)` | `rd_all(TT,TL)` | `no_all(TT,TL)` | `uin(TT)` | `urd(TT)` | `uno(TT)` | `uinp(TT)` | `urdp(TT)` | `unop(TT)` | `out_s(E,G,R)` | `in_s(ET,GT,RT)` | `rd_s(ET,GT,RT)` | `no_s(ET,GT,RT)` | `inp_s(ET,GT,RT)` | `rdp_s(ET,GT,RT)` | `nop_s(ET,GT,RT)` | `get_s()` | `set_s([(E1,G1,R1),...,(En,Gn,Rn)])`
`T,T1,...,Tn`	::=	tuple (Prolog term)
`TT`	::=	tuple template (Prolog term)
`TL`	::=	list of tuples (Prolog list of terms)
`E,E1,...,En`	::=	ReSpecT event
`G,G1,...,Gn`	::=	ReSpecT guard predicate
`R,R1,...,Rn`	::=	ReSpecT reaction body
`ET`	::=	ReSpecT event template
`GT`	::=	ReSpecT guard template
`RT`	::=	ReSpecT reaction body template

TuCSoN CLI: Experiments

1. launch a local TuCSoN Node

   `java -cp libs/tucson.jar:libs/2p.jar alice.tucson.service.TucsonNodeService`

2. launch the CLI on that node

   `java -cp libs/tucson.jar:libs/2p.jar`

   `alice.tucson.service.tools.CommandLineInterpreter`

3. experiment with the semantics of basic TuCSoN primitives
   - `rd` vs. `in`
   - `rd`/`in` vs. `rdp`/`inp`
   - `rd` vs. `no`

4. experiment with LINDA-like coordination by working with multiple CLIs

5. experiment with TuCSoN distribution by working with multiple nodes (and possibly multiple CLIs)

Focus on. . .

External APIs

To enable a Java application to use the TuCSoN technology, do the following

1. build a `TucsonAgentId` to be identified by the TuCSoN system
2. get a TuCSoN ACC to enable interaction with the TuCSoN system
3. define the tuple centre target of your coordination operations
4. build a tuple using the communication language
5. perform the coordination operation using a coordination primitive
6. check requested operation success
7. get requested operation result

Let's try!

Launch Java class `HelloWorld` in package `ds.lab.tucson.helloWorld`

```
java -cp libs/tucson.jar:libs/2p.jar:bin/ ds.lab.tucson.helloWorld.HelloWorld
```

and check out code comments

Extension APIs

To create a TuCSoN agent, do the following

1. extend `alice.tucson.api.TucsonAgent` base class
2. choose one of the given constructors
3. override the `main()` method with your agent business logic
4. get your ACC from the super-class
5. do what you want to do following steps 3 – 7 from previous slide
6. instantiate your agent and start its execution cycle (`main()`) by using method `go()`

Let's try!

Launch Java class `HelloWorldAgent` in package `ds.lab.tucson.helloWorld`

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.helloWorld.HelloWorldAgent
```

and check out code comments

TuCSoN Experiments II I

Package `ds.lab.tucson.*`

1. launch a local TuCSoN node

```
java -cp libs/tucson.jar:libs/2p.jar alice.tucson.service.TucsonNodeService
```

2. `.helloWorld` package

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.helloWorld.HelloWorld
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.helloWorld.HelloWorldAgent
```

3. `.messagePassing` package

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.messagePassing.ReceiverAgent
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.messagePassing.SenderAgent
```

4. `.rpc` package

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.rpc.CalleeAgent
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.rpc.CallerAgent
```

TuCSoN Experiments II II

5. launch two local TuCSoN nodes on ports 20504 and 20505

6. `.masterWorkers` package

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.masterWorkers.MasterAgent
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.masterWorkers.WorkerAgent
```

Next in Line...

Focus on. . .

Bulk Primitives: The Idea

- bulk coordination primitives provide significant efficiency gains for that class of coordination problems involving the management of multiple tuples using a *single coordination operation* [Rowstron, 1996]
- briefly, instead of returning one single tuple, bulk operations return the *whole set of matching tuples*
- in case no matching tuples are found, they successfully complete anyway, returning an *empty list* of tuples (so, bulk primitives always succeed)

Bulk Primitives in TuCSoN

The TuCSoN coordination language provides the following 4 bulk coordination primitives:

`out_all(`*`Tuples`*`)` inserts in the target tuple space the given (Prolog) list of logic tuples

`rd_all(`*`Template`*`)` attempts to read from the target tuple space all the tuples matching the given template, returning them as a list (possibly empty)

`in_all(`*`Template`*`)` attempts to withdraw from the target tuple space all the tuples matching the given template, returning them as a list (possibly empty)

`no_all(`*`Template`*`)` tests the target tuple space for absence of any tuple matching the given template, returning the empty list in case of success and the whole set of matching tuples in case of failure

Bulk Primitives: CLI Experiments I

Try bulk primitives vs. corresponding LINDA primitives

- e.g., synchronise with M processes out of a pool of N (with $M < N$) in the most effective way;
- e.g., compute multiplicity of tuples or count how many tuples satisfy a given template;
- e.g., can any master-workers architecture benefit from these new primitives?

Bulk Primitives: CLI Experiments II

"Master-Workers" example: let's try!

- package `ds.lab.tucson.masterWorkers.bulk`
- launch two local TuCSoN nodes on ports 20504 and 20505

```
java -cp libs/tucson.jar:libs/2p.jar
    alice.tucson.service.TucsonNodeService -portno 20504
java -cp libs/tucson.jar:libs/2p.jar
    alice.tucson.service.TucsonNodeService -portno 20505
```

- `ds.lab.tucson.masterWorkers.bulk` package

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.masterWorkers.bulk.MasterAgent
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.masterWorkers.bulk.WorkerAgent
```

Focus on. . .

The spawn Primitive I

In order to enable TuCSoN agents to delegate complex computational activities related to coordination to the coordination medium itself, TuCSoN provides the `spawn` primitive—similar to Linda `eval`

Semantics

- `spawn` activates a *concurrent computational activity* – actually, either a Java thread or a tuProlog engine – to be carried out asynchronously w.r.t. the caller—either an agent or the tuple centre itself
- the execution of the `spawn` is local to the tuple space where it is invoked, and so are their results
  - correspondingly, the code (either Java or tuProlog) of the spawned computation must be local to the same node hosting the "spawning" tuple centre (no "code on demand")
  - also, the code can execute (a subset of) TuCSoN coordination primitives, but only on the same spawning tuple centre

The spawn Primitive II

General syntax

- spawn has basically two parameters

  activity — a ground Prolog atom containing either the tuProlog theory along with the goal to be solved – e.g., `solve('path/to/Prolog/Theory.pl', yourGoal)` – or the Java class to be executed—e.g., `exec('list.of.packages.YourClass.class')`

  tuple centre — a ground Prolog term identifying the target tuple centre that should execute the spawn

- from tuProlog, the two parameters are just the end of the story…

The `spawn` Primitive III

Java syntax

- ...a third parameter is instead necessary when *spawning* from TuCSoN Java agent (homogeneously with other TuCSoN primitives)
- it could be either

  listener — a listener `TucsonOperationCompletionListener` in case of an asynchronous call of `spawn`

  timeout — an integer value in milliseconds determining the maximum waiting time for the agent in case of a synchronous call of `spawn`—notice its execution is still a separate, concurrent computation

spawn primitive: CLI Experiments I

Try to spawn a Java program as a concurrent activity to be carried out by the coordination medium:

- e.g., coordinate 2 CLIs through the outcome of an expensive computation—or an expensive iteration over tuples in the space
- e.g., again, can any master-workers architecture benefit from this new primitives?

spawn primitive: CLI Experiments II

"Spawned Workers" example: let's try!

- package `ds.lab.tucson.masterWorkers.spawn`
- launch two local TuCSoN nodes on ports 20504 and 20505

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    alice.tucson.service.TucsonNodeService -portno 20504
java -cp libs/tucson.jar:libs/2p.jar
    alice.tucson.service.TucsonNodeService -portno 20505
```

- `ds.lab.tucson.masterWorkers.spawn` package

```
java -cp libs/tucson.jar:libs/2p.jar:bin/
    ds.lab.tucson.masterWorkers.spawn.MasterAgent
```

Focus on. . .

ACC

An Agent Coordination Context (ACC) [Omicini, 2002] is

- a *runtime* and *stateful* interface released to an agent to execute operations on the tuple centres of a specific organisation
- a sort of *interface* provided to an agent by the infrastructure both to *enable and constraint* it admissible interactions with the system—thus other agents and the coordination medium itself

Ordinary ACCs

`OrdinarySynchACC` enables interaction with the *ordinary* tuple space and enacts a synchronous behaviour from the agent's perspective: whichever the coordination operation invoked (either suspensive or predicative), the agent *blocks* waiting for its completion

`OrdinaryAsynchACC` enables interaction with the *ordinary* tuple space and enacts an asynchronous behaviour from the agent's perspective: whichever the coordination operation invoked (either suspensive or predicative), the agent *does not block*, but is instead *asynchronously notified* upon completion

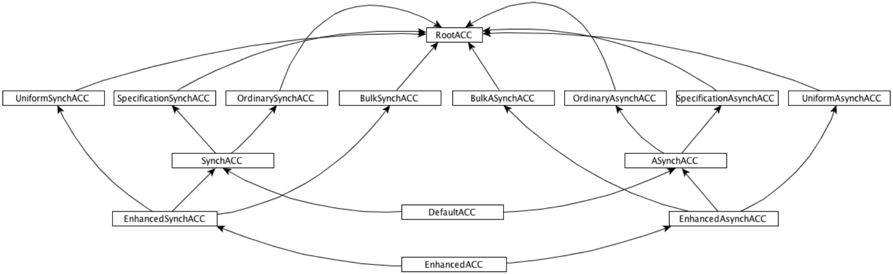
Bulk ACCs

`BulkSynchACC` enables bulk interaction with the *ordinary* tuple space and enacts a synchronous behaviour from the agent's perspective: whichever the bulk coordination operation invoked, the agent *blocks* waiting for its completion

`BulkAsynchACC` enables bulk interaction with the *ordinary* tuple space and enacts an asynchronous behaviour from the agent's perspective: whichever the bulk coordination operation invoked, the agent *does not block*, but is instead *asynchronously notified* of its completion

Overall View over TuCSoN ACCs

Other ACCs exist: some enabling access to the ReSpecT specification space and others being a convenient combination of previous ones



Focus on. . .

TuCSoN Inspector I

A GUI tool to monitor the TuCSoN coordination space & ReSpecT VM

- to launch the Inspector tool

```
java -cp libs/tucson.jar:libs/2p.jar
    alice.tucson.introspection.tools.InspectorGUI
```

- available options are

  `-aid` — the name of the Inspector Agent

  `-netid` — the IP address of the device hosting the TuCSoN Node to be inspected...

  `-portno` — ...its listening port...

  `-tcname` — ...and the name of the tuple centre to monitor

TuCSoN Inspector II

What to inspect

In the *Sets* tab[a] you can choose whether to inspect

- **Tuple Space** — the *ordinary* tuples space state
- **Specification Space** — the (ReSpecT) *specification* tuples space state
- **Pending Ops** — the *pending* TuCSoN operations set, that is the set of the currently suspended issued operations (waiting for completion)
- **ReSpecT Reactions** — the *triggered* (ReSpecT) reactions set, that is the set of specification tuples (recursively) triggered by the issued TuCSoN operations

[a] The *StepMode* tab is for debugging of ReSpecT reactions.

Next in Line. . .

Focus on. . .

Jade

- Jade is one of the oldest and nowadays most widely used agent development frameworks [Bellifemine et al., 2007]
- Jade can be downloaded freely from `http://jade.tilab.com`
- integrating TuCSoN with Jade essentially means to make coordination via tuple centres generally available to agent programmers

TuCSoN4JADE

- TuCSoN4JADE integrate TuCSoN and JADE by implementing TuCSoN as a JADE *service* [Omicini et al., 2004]
- an example of how to use TuCSoN from JADE is reported in the TuCSoN main site at

  http://apice.unibo.it/xwiki/bin/download/TuCSoN/Documents/tucson4jadequickguidepdf.pdf

Synchronous vs. Asynchronous Invocation

- the `BridgeToTucson` class is the component mediating all the interactions between JADE and TuCSoN
- in particular, it offers two methods for invoking coordination operations, one for each *invocation semantics* JADE agents may choose [Mariani et al., 2014]:

`synchronousInvocation()` — lets agents invoke TuCSoN coordination operations *synchronously w.r.t. the caller behaviour*. This means the caller behaviour *only* is (possibly) suspended – and automatically resumed – as soon as the requested operation completes, not the agent as a whole—as in [Omicini et al., 2004].

`asynchronousInvocation()` — lets clients *asynchronously* invoke TuCSoN coordination operations. Regardless of whether the coordination operation suspends, the agent does not, thus the caller behaviour continues [Mariani et al., 2014].

References I

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007).
Developing Multi-Agent Systems with JADE.
Wiley.

Lloyd, J. W. (1984).
Foundations of Logic Programming.
Springer, 1st edition.

Mariani, S., Omicini, A., and Sangiorgi, L. (2014).
Models of autonomy and coordination: Integrating subjective & objective approaches in agent development frameworks.
In Braubach, L., Camacho, D., and Venticinque, S., editors, *8th International Symposium on Intelligent Distributed Computing (IDC 2014)*, Madrid, Spain.

Omicini, A. (2002).
Towards a notion of agent coordination context.
In Marinescu, D. C. and Lee, C., editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA.

Omicini, A. and Denti, E. (2001).
From tuple spaces to tuple centres.
Science of Computer Programming, 41(3):277–294.

References II

Omicini, A., Ricci, A., Viroli, M., Cioffi, M., and Rimassa, G. (2004).
Multi-agent infrastructures for objective and subjective coordination.
Applied Artificial Intelligence: An International Journal, 18(9–10):815–831.
Special Issue: Best papers from EUMAS 2003: The 1st European Workshop on Multi-agent Systems.

Omicini, A. and Zambonelli, F. (1999).
Coordination for Internet application development.
Autonomous Agents and Multi-Agent Systems, 2(3):251–269.
Special Issue: Coordination Mechanisms for Web Agents.

Rowstron, A. I. T. (1996).
Bulk Primitives in Linda Run-Time Systems.
PhD thesis, The University of York.

Tuple-based Coordination with TuCSoN

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Stefano Mariani   Andrea Omicini

s.mariani@unibo.it   andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018