TuCSoN: **Tuple Centres Spread over the Network**

Basics

Andrea Omicini  
Stefano Mariani

{andrea.omicini, s.mariani}@unibo.it

**Alma Mater Studiorum**—Università di Bologna a Cesena

November 8th, 2012
These slides are adapted, rearranged, integrated starting from the official TuCSoN guide available at

http://www.slideshare.net/andreaomicini/
the-tucson-coordination-model-technology-a-guide

by the same authors.
1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4. Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5. Bibliography
Outline

1 Basic Model & Language
   • Basic Model
   • Naming
   • Basic Language
   • Basic Primitives

2 Basic Architecture
   • Nodes & Tuple Centres
   • Coordination Spaces

3 Basic Technology
   • Middleware
   • Tools
   • CLI Experiments

4 Basic Java APIs
   • Java Apps & Java TuCSoN Agents
   • Java Experiments

5 Bibliography
TuCSoN is a model for the coordination of distributed processes, as well as of autonomous agents [Omicini and Zambonelli, 1999]

References

Main Page http://tucson.apice.unibo.it/

FaceBook\(^a\) http://www.facebook.com/TuCSoNCoordinationTechnology

Google Code http://tucson.googlecode.com/

SourceForge http://sf.net/projects/tucson/

\(^a\)Obviously, you should now go to the page and like it ;)}
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 *reactive*, 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 through tuple centres.
- 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.
Basic Topology

- Agents and tuple centres are spread over the network.
- Agents could in principle move independently of the device where they run, whereas tuple centres are permanently associated to one device...
- ...which can also be a mobile device: hence tuple centres can be mobile if and only if the hosting device is mobile.

System view

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

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4. Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5. Bibliography
Nodes

- Each node within a TuCSoN system is *univocally identified* by the pair `<NetworkId, PortNo>`, where
  - *NetworkId* is either the IP number or the DNS entry of the device hosting the node
  - *PortNo* is the port number where the TuCSoN *coordination service* listens to the invocations for the execution of coordination operations

- Correspondingly, the abstract syntax\(^1\) for the identifier of a TuCSoN node hosted by a networked device *netid* on port *portno* is
  \[
  \text{netid : portno}
  \]

---

\(^1\)Actually, it’s 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*\(^2\) [Lloyd, 1984]

- Since each node contains at most one tuple centre for each admissible name, 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

\[
tname @ netid : portno
\]

\(^2\)“ground” basically means no variables in it
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)\(^3\).
- If an agent \texttt{aname} is assigned UUID \texttt{uuid}, its full name is \texttt{aname : uuid}.

\(^3\)http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html
Outline

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
     - Basic Primitives

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4. Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5. Bibliography
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, thus to synchronise on them.
- 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.
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\(^4\) logic Prolog atom is an admissible TuCSoN tuple...
- ...and an admissible TuCSoN tuple template

\(^4\)Be aware that atoms like \(X\) and thus coordination operations like \(\text{out}(X)\), are second-order logic atoms, hence are in principle not admissible.
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.

- We won’t really *see* them until talking about ReSpecT *reactions*. 
Abstract Syntax

- The abstract syntax of a coordination operation \( \text{op} \) invoked on a target tuple centre \( \text{tcid} \) is

\[
\text{tcid} \ ? \ \text{op}
\]

where \( \text{tcid} \) is the tuple centre *full name*

- Given the structure of the full name of a tuple centre, the *general abstract syntax*\(^5\) of a TuCSoN coordination operation is

\[
\text{tname} \ @ \ \text{netid} : \ \text{portno} \ ? \ \text{op}
\]

\(^5\)Actually, it’s also the concrete syntax used by TuCSoN to parse coordination operations, even inside ReSpecT reactions
Outline

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4. Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5. Bibliography
Coordination Primitives

The TuCSoN coordination language provides the following 9 basic\(^6\) coordination primitives to build coordination operations:

- `out`
- `rd, rdp`
- `in, inp`
- `no, nop`
- `get`
- `set`

\(^6\)We will see others in next lesson.
out($Tuple$) writes $Tuple$ in the target tuple space; after the operation is successfully executed, $Tuple$ is returned as a completion.

rd($TupleTemplate$) looks for a tuple matching $TupleTemplate$ in the target tuple space; if a matching $Tuple$ is found when the operation is served, the execution succeeds by returning $Tuple$; otherwise, the execution is suspended to be resumed and successfully completed when a matching $Tuple$ will be finally found in and returned from the target tuple space.

in($TupleTemplate$) looks for a tuple matching $TupleTemplate$ in the target tuple space; if a matching $Tuple$ is found when the operation is served, the execution succeeds by removing and returning $Tuple$; otherwise, the execution is suspended to be resumed and successfully completed when a matching $Tuple$ will be finally found in, removed and returned from the target tuple space.
Predicative Primitives

$\text{rdp}(\text{TupleTemplate})$ predicative (non-suspensive) version of $\text{rd}(\text{TupleTemplate})$; if a matching $\text{Tuple}$ is not found, the execution fails (operation outcome is FAILURE) and $\text{TupleTemplate}$ is returned;

$\text{inp}(\text{TupleTemplate})$ predicative version of $\text{in}(\text{TupleTemplate})$; if a matching $\text{Tuple}$ is not found, the execution fails, no tuple is removed from the target tuple space and $\text{TupleTemplate}$ is returned;
no\((\text{TupleTemplate})\) looks for a Tuple matching TupleTemplate in the target tuple space; if no matching tuple is found when the operation is served, the execution succeeds, and TupleTemplate is returned; otherwise, the execution is suspended to be resumed and successfully completed when no matching tuples can any longer be found in the target tuple space, then TupleTemplate is returned.

nop\((\text{TupleTemplate})\) predicative version of no\((\text{TupleTemplate})\); if a matching Tuple is found the execution fails and Tuple is returned.
get() reads all the Tuples in the target tuple space and returns them as a list; if no tuple occurs in the target tuple space at execution time, the empty list is returned and the execution succeeds anyway.

set(Tuples) overwrites the target tuple spaces with the list of Tuples; when the execution is completed, the list of Tuples is successfully returned.
Outline

1 Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2 Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3 Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4 Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5 Bibliography
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
  \[\text{tname} \@ \text{netid} ? \text{op}\]
  without specifying the node port number \text{portno}\n
- Any other port can be used for a TuCSoN node listening service (we will see how to change it in a few slides)

\[\text{7Meaning that the agent intends to invoke operation op on the tuple centre tname of the default node netid : 20504, hosted by the networked device netid.}\]
Tuple Centres

- Given an admissible tuple centre name $tname$, tuple centre $tname$ is an admissible tuple centre\(^8\)
- The *coordination space* of a TuCSoN node is defined as the collection of *all* the admissible tuple centres
- Any TuCSoN node provides agents with a *complete* coordination space, so that in principle any coordination operation can be invoked on any admissible tuple centre belonging to any TuCSoN node

---

\(^8\)It may seem awkward, but it’s not, so think well about it.
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.

The default tuple centre of any TuCSoN node is named default.

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

```plaintext
@ netid : portno ? op
```

without specifying the tuple centre name tname.
Outline

1 Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2 Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3 Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4 Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5 Bibliography
Global coordination space

- The TuCSoN global coordination space is defined at any time by the collection of all the tuple centres available on the network, identified by their full name.
- A TuCSoN agent running on any networked device has at any time the whole TuCSoN global coordination space available for its coordination operations, through invocations of the form
  \[ \text{tname} \circ \text{netid} : \text{portno} \ ? \text{op} \]
  which invokes operation \text{op} on the tuple centre \text{tname} provided by (hopefully, up & listening) node \text{netid} : \text{portno}
Local Coordination Space

- Given a networked device `netid` hosting one or more TuCSoN nodes, the TuCSoN local coordination space is defined at any time by the collection of all the tuple centres made available by all the TuCSoN nodes hosted by `netid`.
- An agent running on the same device `netid` can exploit the local coordination space to invoke operations of the form
  
  \[ \text{tname} : \text{portno} \ ? \ \text{op} \]

  which invokes operation `op` on the tuple centre `tname` locally provided\(^9\) by node `netid : portno`.

---

\(^9\text{Roughly speaking, netid can be substituted with localhost.}\)
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
Outline

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4. Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5. Bibliography
Technology Requirements

- **TuCSoN** is a **Java-based middleware** (Java 6 should be enough, but 7 is better and you are already up-to-date due to JADE right?)
- **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
- **TuCSoN** comes shipped with **tuProlog** already *inside*, so you don’t need anything else except a JVM and the TuCSoN jar

---

¹⁰ Last digits in TuCSoN version number (TuCSoN-1.10.2.0205) are for the tuProlog version, hence tuProlog version 2.5 now
Java & Prolog Agents

TuCSoN middleware provides

- **Java API for extending Java programs with TuCSoN coordination primitives**
  - package `alice.tucson.api.*`
- **Java classes for programming TuCSoN agents in Java**
  - `alice.tucson.api.TucsonAgent` provides a ready-to-use agent, whose `main()` method should be overridden by the user
- **Prolog libraries for extending tuProlog programs with TuCSoN coordination primitives**
  - `alice.tucson.api.Tucson2PLibrary` enables tuProlog agents to use TuCSoN primitives
  - use directive `:- load_library(path/to/Tucson2PLibrary)` to load the library
Service

- Given any networked device running a Java VM, a TuCSoN node can be booted to make it provide a TuCSoN service.

- A TuCSoN service can be started through the `alice.tucson.service` Java API, e.g.

  ```java
  java -cp TuCSoN-1.10.2.0205.jar alice.tucson.service.TucsonNodeService -port 20505
  ```

- The node service is in charge of
  - listening to incoming operation invocations on the associated port of the device
  - dispatching them to the target tuple centre
  - returning the operations completion to the source agent
Coordination Space

- Tuple centres in a node are either *actual* or *potential*: at any time in a given node
  - actual tuple centres are admissible tuple centres that already *do* have a reification as a run-time abstraction
  - potential tuple centres are admissible tuple centres that *do not* have a reification as a run-time abstraction, yet
- The node service is in charge of making *potential* tuple centres *actual* as soon as the first operation on them is received and served
Outline

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
     - CLI Experiments

4. Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5. Bibliography
Command Line Interface (CLI) I

- Shell interface for human agents / programmers, e.g.
  
  ```java
  java -cp TuCSoN-1.10.2.0205.jar
  alice.tucson.service.tools.CommandLineInterpreter
  -netid localhost -port 20505 -aid myCLI
  ```
## CLI Syntax

\[
\langle TucsonOp \rangle ::= \langle TcName \rangle \, @ \, \langle IpAddress \rangle \, : \, \langle PortNo \rangle \, ? \, \langle Op \rangle \\
\langle TcName \rangle ::= \text{Prolog ground term} \\
\langle IpAddress \rangle ::= \text{localhost} \mid \text{IP address} \\
\langle PortNo \rangle ::= \text{port number} \\
\langle Op \rangle ::= \text{out(T)} \mid \text{in(TT)} \mid \text{rd(TT)} \mid \text{no(TT)} \mid \text{inp(TT)} \mid \text{rdp(TT)} \mid \text{nop(TT)} \mid \text{get()} \mid \text{set([T1,...,Tn])} \\
\text{out\_all(TL)} \mid \text{in\_all(TT,TL)} \mid \text{rd\_all(TT,TL)} \mid \text{no\_all(TT,TL)} \\
\text{uin(TT)} \mid \text{ur(TT)} \mid \text{uno(TT)} \mid \text{uinp(TT)} \mid \text{urdp(TT)} \mid \text{unop(TT)} \\
\text{out\_s(E,G,R)} \mid \text{in\_s(ET,GT,RT)} \mid \text{rd\_s(ET,GT,RT)} \mid \text{no\_s(ET,GT,RT)} \\
\text{inp\_s(ET,GT,RT)} \mid \text{rdp\_s(ET,GT,RT)} \mid \text{nop\_s(ET,GT,RT)} \\
\text{get\_s()} \mid \text{set\_s([(E1,G1,R1),...,(En,Gn,Rn)])} \\
\]

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
Outline

1 Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2 Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3 Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4 Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5 Bibliography
TuCSoN Experiments I

1. Launch a local TuCSoN Node
   
   ```
   java -cp TuCSoN-1.10.2.0205.jar alice.tucson.service.TucsonNodeService
   ```

2. Launch the CLI on that node
   
   ```
   java -cp TuCSoN-1.10.2.0205.jar alice.tucson.service.tools.CommandLineInterpreter
   ```

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

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

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

1. Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2. Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3. Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4. Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5. Bibliography
To enable a Java application to use the TuCSoN technology, do the following:

1. **build a TucsonAgentId to be identified by the TuCSoN system**
   → TucsonAgentId aid = new TucsonAgentId("helloWorldMain");

2. **get a TuCSoN ACC to enable interaction with the TuCSoN system**
   → SynchACC acc = TucsonMetaACC.getContext(aid);

3. **define the tuplecentre target of your coordination operations**
   → TucsonTupleCentreId tid = new TucsonTupleCentreId("default", "localhost", "20504");

4. **build a tuple using the communication language**
   → LogicTuple tuple = new LogicTuple("hello", new Value("world"));

5. **perform the coordination operation using a coordination primitive**
   → ITucsonOperation op = acc.out(tid, tuple, null);

6. **check requested operation success**
   → if(op.isResultSuccess()) {...}

7. **get requested operation result**
   → LogicTuple res = op.getLogicTupleResult();
Extension APIs

To create a TuCSoN agent part of a TuCSoN system, do the following:

1. extend `alice.tucson.api.TucsonAgent` base class
2. choose one of the given constructors, e.g.
   ```java
   protected HelloWorldAgent(String aid) throws TucsonInvalidAgentIdException {
       super(aid);
   }
   ```
3. override `main()` method with your agent business logic
4. get your ACC from the super-class
   ```java
   SynchACC acc = getContext();
   ```
5. do what you want to do following steps 3 – 7 from previous slide
   ```java
   LogicTuple tuple = LogicTuple.parse("hello(world)" newX);
   LogicTuple template = LogicTuple.parse("hello(Who)" newX);
   ```
6. instantiate your agent and start its execution cycle (`main()`) by using method `go()`
   ```java
   new HelloWorldAgent(aid).go();
   ```
Outline

1 Basic Model & Language
   - Basic Model
   - Naming
   - Basic Language
   - Basic Primitives

2 Basic Architecture
   - Nodes & Tuple Centres
   - Coordination Spaces

3 Basic Technology
   - Middleware
   - Tools
   - CLI Experiments

4 Basic Java APIs
   - Java Apps & Java TuCSoN Agents
   - Java Experiments

5 Bibliography
TuCSoN Experiments II

1. Launch a local TuCSoN Node
   ```java
   java -cp TuCSoN-1.10.2.0205.jar alice.tucson.service.TucsonNodeService
   ```

2. Launch the HelloWorld Java program in `ds.lab.tucson.helloWorld` package

3. Launch the HelloWorldAgent TuCSoN agent in `ds.lab.tucson.helloWorld` package

4. If you feel confident enough, experiment with packages¹¹
   - `ds.lab.tucson.messagePassing`
   - `ds.lab.tucson.rpc`
   - `ds.lab.tucson.masterWorkers`

¹¹Read comments 'cause they’re there for you!


TuCSoN: Tuple Centres Spread over the Network
Basics

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

Alma Mater Studiorum—Università di Bologna a Cesena

November 8th, 2012