TuCSoN4JADE quick guide

In this quick guide, you will learn how to get TuCSoN4JADE (**t4j** for short), build it and run a first example showcasing its features. You will also learn the main API available to JADE developers and how to use them in your code, for your JADE agents, as well as a bit of "behind the scenes" on how t4j works.

Assumptions are you are familiar with Java (compilation), JADE (usage), Git (cloning repositories), optionally, ANT (buildfiles), JADE and TuCSoN.

Prior to reading this how-to is *highly recommended* to **read the reference paper** on integrating TuCSoN and JADE:

- Mariani, S., Omicini, A., Sangiorgi, L.: "*Models of Autonomy and Coordination: Integrating Subjective & Objective Approaches in Agent Development Frameworks*". Published in 8th International Symposium on Intelligent Distributed Computing (IDC 2014), 3-5 September 2014

available from here.

---

1. Getting Started with t4j

1.1 Downloading

If you want the *ready-to-use* distribution of t4j, download **t4j.jar** archive from the "Downloads" section, here > http://bitbucket.org/smariani/tucson4jade/downloads.

If you want the *source code* of t4j, clone the t4j **Git repository** hosted here on Bitbucket, at > http://smariani@bitbucket.org/smariani/tucson4jade.git (e.g., from a command prompt type `$> git clone https://smariani@bitbucket.org/smariani/tucson4jade.git`) [1].

In the former case, skip to the "Running" section below. In the latter case, keep reading.

1.2 Compiling

By cloning t4j you have downloaded a folder named `tucson4jade/`, with the following *directory structure*:

```
tucson4jade/
|__...
|__t4j/
   |__...
   |__ant-scripts/
      |__build.xml
      |__environment.properties
   |__eclipse-config/
   |__how-to/
   |__license-info/
   |__src/
```

t4j depends on 3 other Java libraries to function properly [2]:

- **JADE**, downloadable from JADE "Download" page, here > http://jade.tilab.com/download/jade/ (**jade.jar**)
- **TuCSoN**, downloadable from TuCSoN "Downloads" section, here > http://bitbucket.org/smariani/tucson/downloads (**tucson.jar**)
- **tuProlog**, downloadable from tuProlog "Download" page, here > http://apice.unibo.it/xwiki/bin/view/Tuprolog/Download (**2p.jar**)

Once you got the above libraries, you are ready to compile t4j source code.

The easiest way to do so is by exploiting the ANT script named `build.xml` within folder `ant-scripts/`, which takes care of the whole building process for you, from compilation to deployment (covered in next section). To do so, you need to have ANT installed on your machine [3]. If you don't want to use ANT, build t4j jar archive using the tools you prefer, then skip to the "Running" section below.

To compile t4j using ANT:

1. Edit the `environment.properties` file according to your system configuration:

   1.1 Tell ANT where your JDK and your `java` tool are

   1.2 Tell ANT which libraries are needed to compile t4j (those you just downloaded, that is JADE, TuCSoN and tuProlog)

   1.3 Tell ANT where you put such libraries (e.g. if you put them into `t4j/libs/` you are already set)

   [1.4 Tell ANT your Bitbucket username (for automatic syncing with t4j repository, ***not supported at the moment****)]*

2. Launch the ANT script using target `compile` (e.g., from a command prompt position yourself into the `ant-scripts/` folder then type `$> ant compile`) [4]. This will create folder `classes/` within folder `t4j/` and therein store Java `.class` files.

Other ANT targets are available through the `build.xml` file: to learn which, launch the ANT script

using target `help`.

1.3 Deploying

Deploying t4j is as simple as giving a different build target to the ANT script `build.xml`:

- if you only want the **t4j jar** archive, ready to be included in your JADE project, launch the script using target `lib`. This will compile t4j source code into binaries (put into `t4j/classes/` folder) then package them to **t4j.jar** into `t4j/lib/` folder [5].
- if you want a **ready-to-release distribution** of t4j, including also documentation and support libraries, launch the script using target `dist`. This will:
    - compile t4j source code into binaries, put into `t4j/classes/` folder
    - package them to t4j.jar, put into `t4j/lib/` folder
    - generate Javadoc information, put into `t4j/doc/` folder
    - create folder `jade/add-ons/TuCSoN4JADE-${version}` including:
        - folder `docs/` including the generated Javadoc information as well as this "how-to"
        - folder `libs/` including JADE, TuCSoN and tuProlog libraries used to build t4j
        - folder `rel/` including t4j jar archives

The complete directory structure obtained by launching `ant dist` build process should look like the following (assuming you put JADE, TuCSoN and tuProlog libraries in folder `t4j/libs/`):

```
tucson4jade/
|__...
|__t4j/
   |__...
   |__ant-scripts/
      |__build.xml
      |__environment.properties
   |__classes/
   |__doc/
   |__eclipse-config/
   |__how-to/
   |__jade/
      |__add-ons/
         |__TuCSoN4JADE-${version}/
            |__docs/
               |__how-to/
               |__javadoc/
            |__libs/
            |__rel/
         |__...
   |__lib/
   |__libs/
   |__license-info/
   |__src/
```

Other ANT targets are available through the `build.xml` file: to learn which, launch the ANT script

using target `help`.

1.4 Running

To run t4j, you need:

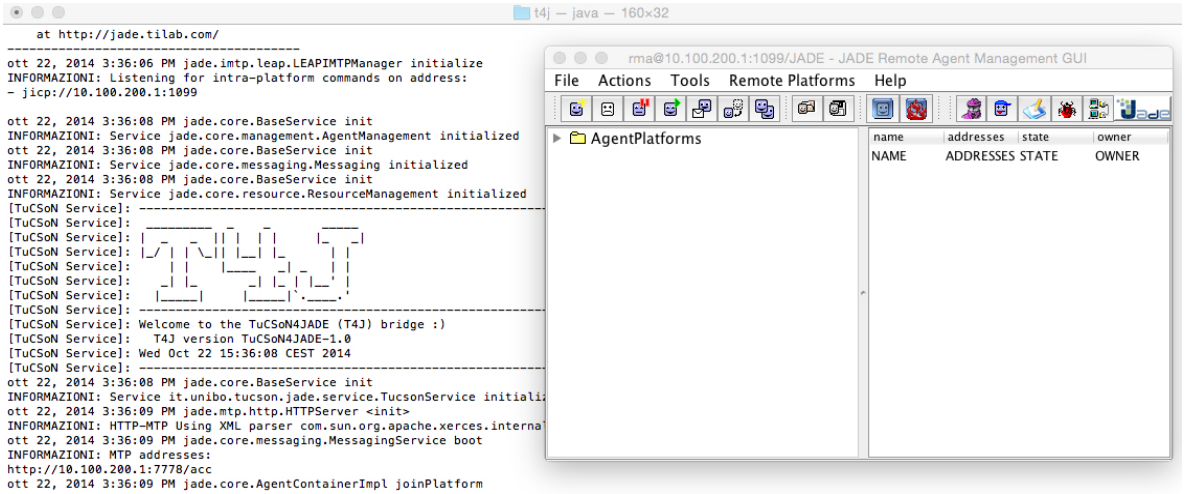
- t4j jar, **t4j.jar**
- JADE jar, **jade.jar**
- TuCSoN jar, **tucson.jar**
- tuProlog jar, **2p.jar**

Supposing you built t4j using the provided ANT script [6] and that you are comfortable with using a command prompt to launch Java applications [7]:

1. open a command prompt and position yourself into either `t4j/lib/` or `t4j/jade/add-ons/TuCSoN4JADE-${version}/rel/` folder
2. launch the JADE platform specifying you want to exploit TuCSoN services provided by t4j, e.g. as follows [8]:

```
 java -cp t4j.jar:../libs/tucson.jar:../libs/2p.jar:../libs/jade.jar jade.Boot
-gui -services it.unibo.tucson.jade.service.TucsonService
```

JADE GUI should appear as well as the t4j ASCII logo on the command prompt, as depicted below.



As long as no JADE agents start exploiting TuCSoN coordination services wrapped by t4j, nothing happens. Thus, here follows instructions on how to launch the example application shipped within t4j.jar, showcasing its features: the "Book Trading" MAS (package `it.unibo.tucson.jade.examples.bookTrading`).

Supposing you successfully launched the JADE platform as described above, to launch the "Book Trading" example:

1. operate on JADE gui to launch one "seller agent" (`it.unibo.tucson.jade.examples.bookTrading.BookSellerAgent`), whose name should adhere to TuCSoN agents naming rules (roughly, lowercase letters [9])

2. operate on JADE gui to launch at least one "buyer agent" (`it.unibo.tucson.jade.examples.bookTrading.BookBuyerAgent`), whose name should adhere to TuCSoN agents naming rules

You should see many prints on the command prompt, tracking what happens in the MAS [10].

NB: To showcase TuCSoN4JADE features, the example is designed to start & stop a *default* TuCSoN node (from the seller agent), thus:

- launching more seller agents will likely raise exceptions
- starting a buyer agent first will likely raise exceptions
- shutting the seller agent prior to the buyer agent will likely raise exceptions

If you want to try a setting with more sellers, comment out node starting/stopping code in seller agent's source code and start the TuCSoN node separately [11].

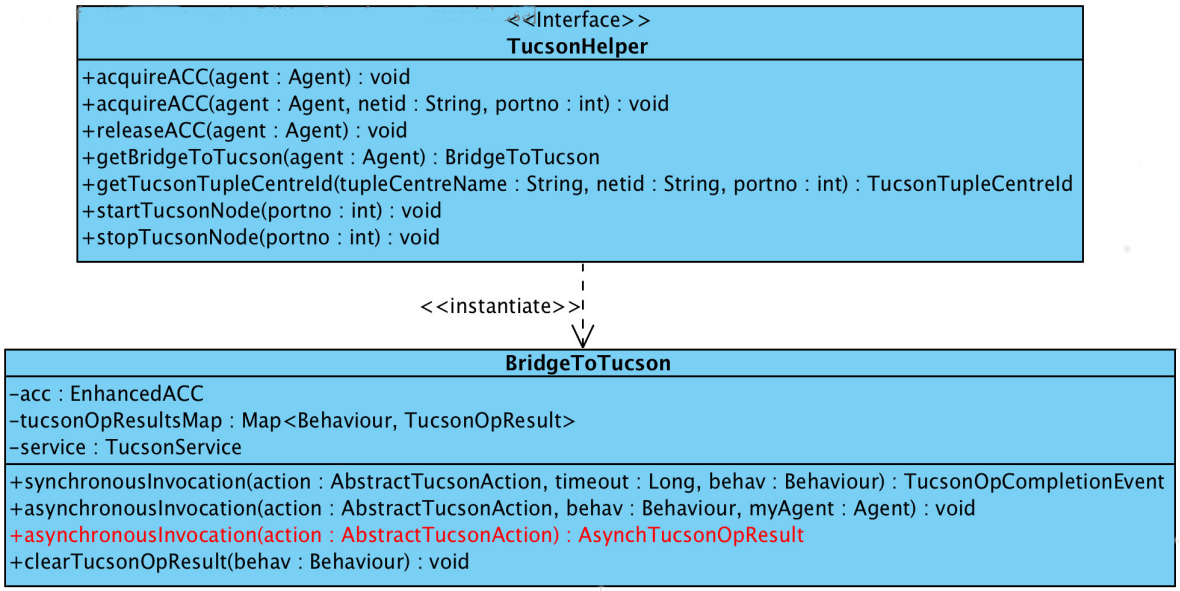
---

2. Using t4j

2.1 API Overview

The first step in integrating TuCSoN and JADE has been implementing TuCSoN *as a* JADE *service*.

This means JADE `BaseService` class has been extended with the `TucsonService` class, representing TuCSoN service entry point. This class **has to be used to get an helper class**, extending JADE `ServiceHelper` interface, working as the actual mediator between clients and the TuCSoN service: in t4j, the helper role is played by the `TucsonHelper` interface---whose implementation class is hidden to clients. Its methods are quite self-explanatory if you know TuCSoN terminology, and are listed in the picture below.



The only "unusual" method is `getBridgeToTucson()`: `BridgeToTucson` is the class which `TucsonHelper` **delegates TuCSoN coordination operations invocation** to.

`BridgeToTucson` exposes the following API:

- `synchronousInvocation()` — lets clients **synchronously** invoke TuCSoN coordination operations
- `asynchronousInvocation()` — lets clients **asynchronously** invoke TuCSoN coordination operations

Synchronous Invocation. Given a coordination operation to perform (`AbstractTucsonAction` subtypes, see t4j Javadoc), a maximum waiting time to be possibly suspended for (`timeout`), and a reference to the caller Jade behaviour, the chosen coordination operation is requested to the active TuCSoN service **synchronously w.r.t. the caller behaviour**. This means *the caller behaviour only is (possibly) suspended and automatically resumed by t4j as soon as the requested operation completes*—returning the completion event reified by `TucsonOpCompletionEvent` object.

Such mechanism encourages JADE programmers using t4j to adopt the same programming style suggested by the JADE Programmers Guide (available here) regarding message reception:

1. the communication method – `synchronousInvocation()` in t4j, `receive()` in JADE – is first called
2. the result is checked, and *(i)* handled, if available, *(ii)* otherwise method `block()` is called

```
@Override
public void action () {
    // field 'mt' stores the ACL message template
    final ACLMessage msg = myAgent.receive(mt);
    if (msg != null) { // message received: process it
        ...
    } else { // message not received yet: wait
        block();
    }
}

@Override
public void action () {
    // field 'tuple' stores the TuCSoN tuple template
    final Rd op = new Rd(tcid, tuple);
    final TucsonOpCompletionEvent
            res = bridge.synchronousInvocation(op, Long.MAX_VALUE, this);
    if (res != null) { // tuple found: process it
        ...
    } else { // tuple not found yet: wait
        block();
    }
}
```

This allows JADE runtime – through the behaviours scheduler – to *keep on scheduling others behaviours belonging to the caller agent while the invoking behaviour remains suspended* (within JADE "waiting queue").

Asynchronous Invocation: "Interrupt Mode". Lets clients *asynchronously* invoke TuCSoN coordination operations, handling results "by interrupt".

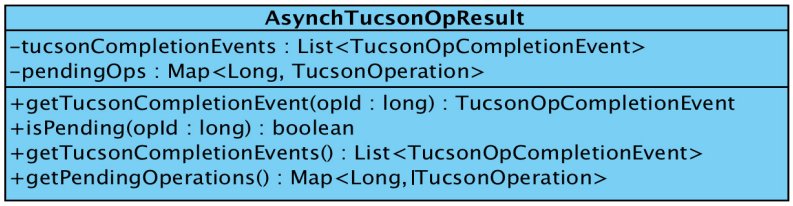
In particular, when the requested operation completes, the JADE behaviour given as actual parameter is activated to handle the operation result-that is, put in the ready queue, ready to be

scheduled.

The "result-handling" behaviour written by JADE programmers should implement t4j `IAsynchCompletionBehaviour` interface: the `setTucsonOpCompletionEvent()` method is the "hook" for t4j to share completion events between the caller and the "result handler" behaviour, transparently to JADE programmers.

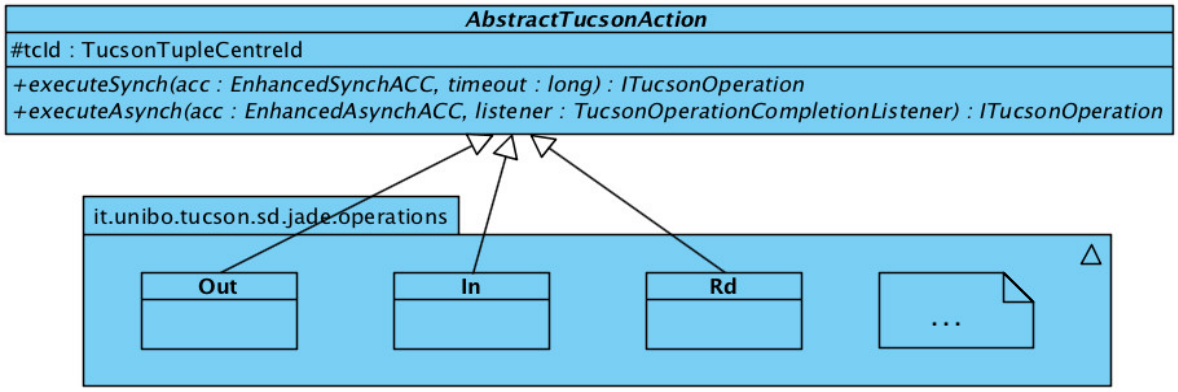
Asynchronous Invocation: "Polling Mode". Lets clients *asynchronously* invoke TuCSoN coordination operations, handling results "by polling".

In particular, the caller agent gets a data structure (`AsynchTucsonOpResult`, depicted below) representing the operation result, which it may query to check completion and (eventually) retrieve the actual result.



In both cases, regardless of whether the coordination operation suspends or not, the agent does not, thus the caller behaviour keeps on executing.

As a last note, the type hierarchy representing TuCSoN coordination operations is in package `it.unibo.tucson.sd.jade.operations` as depicted in figure below.



2.1 "Hands-on" step-by-step tour

NB: What follows is based on the "Book Trading" example scenario shipped within t4j distribution (see t4j "Getting Started", here).

If you want to work with t4j, **remember to instruct the JADE platform to boot the TuCSoN service**, as explained in t4j "Getting Started", here.

Regardless of how you are willing to exploit TuCSoN services, you need to:

1. get the service helper class from TuCSoN service instance

```
ITucsonHelper helper = (TucsonHelper) this.getHelper(TucsonService.NAME);
```

2. *[OPTIONAL]* [12] start the TuCSoN node you wish to operate on

```
if (!this.helper.isActive(20504)) {
    this.helper.startTucsonNode(20504);
}
```

3. get an ACC (which is *actually associated to the* `BridgeToTucson` *object* you'll get from the helper in step 4)

```
this.helper.acquireACC(this);
```

4. get the bridge object *through which all your TuCSoN operations will go*

```
BridgeToTucson bridge = this.helper.getBridgeToTucson(this);
```

5. *[OPTIONAL]* [12] stop the TuCSoN node when its services are no longer needed

```
if (this.helper.isActive(20504)) {
    this.helper.stopTucsonNode(20504);
}
```

Now, what to do next obviously depends on what your program logic needs. Anyway, you will likely perform some of the following operations:

- build the identifier of the tuple centre you wish to use (e.g., putting "hello" tule on default TuCSoN node)

```
TucsonTupleCentreId tcid = this.helper.buildTucsonTupleCentreId(
                "default", "localhost", 20504);
```

- build the tuples you need (using usual TuCSoN facilities)

```
LogicTuple adv = LogicTuple.parse(
                "advertise(provider("
                    + this.getAID().getName()
                    + "), service('book-trading')))");
```

- *build an* "*action*", representing TuCSoN (meta-)coordination operations, choosing from all the type hierarchy rooted in `it.unibo.sd.jade.operations.AbstractTucsonAction`

```
Out out = new Out(this.tcid, this.adv);
```

- perform the action according to your preferred *invocation semantics* (e.g. asynchronous,

"polling" mode)

```
AsynchTucsonOpResult res = this.bridge.asynchronousInvocation(out);
```

(e.g. asynchronous, "interrupt" mode)

```
this.bridge.asynchronousInvocation(out,
            new AdvertisingCompletedBehaviour(this.adv), this);

private class AdvertisingCompletedBehaviour extends OneShotBehaviour
    implements IAsynchCompletionBehaviour {...}
```

(e.g. synchronous mode)

```
TucsonOpCompletionEvent result =
        BookSellerAgent.this.bridge.synchronousInvocation(in, null, this);
```

remembering, if needed, to exploit JADE's usual programming pattern

```
if (result != null) { // tuple found: process it
    ...
} else { // tuple not found: wait
    block();
}
```

Contact Information

Author of this "how-to":

- *Stefano Mariani*, DISI - Università di Bologna (s.mariani@unibo.it)

Authors of the add-on:

- *Stefano Mariani*, DISI - Università di Bologna (s.mariani@unibo.it)
- Luca Sangiorgi, Università di Bologna
- Prof. Andrea Omicini, DISI - Università di Bologna

---

[1] Git standalone clients are available for any platform (e.g., SourceTree for Mac OS and Windows). Also, if you are using Eclipse IDE for developing in JADE, the EGit plugin is included in the Java Developers version of the IDE.

[2] Recommended JADE version is **4.3.2**. Regarding TuCSoN and tuProlog, recommended version is **1.11.0.0209** and **2.9.1**, respectively. Others (both newer and older) may work properly, but they have not been tested.

[3] Binaries available here, installation instructions covering Linux, MacOS X, Windows and Unix

systems here.

[4] If you are using Eclipse IDE for developing in JADE, ANT is included: click "Window > Show View > Ant" then click "Add buildfiles" from the ANT view and select file `build.xml` within `ant-scripts/` folder. Now expand the "TuCSoN4JADE build file" from the ANT view and finally double click on target `compile` to start the build process.

[5] Actually, also a **t4j-noexamples.jar** is built. It is the same as **t4j.jar** except for the explanatory example in package `it.unibo.tucson.jade.examples.bookTrading`, which is excluded.

[6] If you directly downloaded t4j jar or if you built it from sources without using the provided ANT script, simply adjust the given command to suit your configuration.

[7] If you do not want to use the command prompt to launch Java applications, adjust the given command to suit your configuration, e.g., if your are using Eclipse IDE: right-click on "jade.jar > Run As > Run Configurations..." then double-click on "Java Application", select "Boot - jade" as the main class, finally in the arguments tab put `-gui -services it.unibo.tucson.jade.service.TucsonService` as program arguments (`-cp t4j.jar:../libs/tucson.jar:../libs/2p.jar:../libs/jade.jar` is automatically added by Eclipse according to project's build path settings).

[8] Separator `:` works on Mac & Linux only, use `;` on Windows.

[9] Actually, a TuCSoN agent identifier can be any valid tuProlog *ground term*. See tuProlog documentation, here.

[10] Because the seller agent starts the TuCSoN node, the whole MAS runs in the same JVM process, thus sharing the same standard output. If you want to better distinguish what the agents do from what the TuCSoN node does, comment out node starting/stopping code in seller agent's source code and start the TuCSoN node separately [11].

[11] E.g., in a command prompt type `java -cp tucson.jar:2p.jar alice.tucson.service.TucsonNodeService`. More on TuCSoN in its documentation, here.

[12] You can also start/stop a TuCSoN node separately, from another Java class, bash script, or even by hand. See TuCSoN "Getting Started" available here for more info.
