## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student Projects</td>
<td>3</td>
</tr>
<tr>
<td>TuCSoN &quot;as it is&quot;</td>
<td>3</td>
</tr>
<tr>
<td>TuCSoN &quot;as it will be&quot;</td>
<td>3</td>
</tr>
</tbody>
</table>
Student Projects

Here follows the list of ongoing student projects in the context of the class of Distributed Systems held by Prof. Omicini (2013/2014). Each entry on the list leads to a separate Wiki page students are encouraged to use at their advantage (write down ideas, track documents, attach code, etc.)

Projects have been divided into two different "categories", although they share a common goal (for you): learn what it means to model/design/implement/test a middleware. Why middlewares? Basically 3 reasons:

- being projects proposed in the context of the Distributed System class, it seems reasonable (and profitable, for your career) to start "play" with something distributed: middlewares are perfect for the purpose
- middlewares pose some non-trivial challenge both at the model, architectural and implementation level, mostly due to their typical distributed nature forcing programmers to drop many implicit assumptions made when programming a non-distributed setting
- a distributed middleware is already available, which is TuCSoN, so it seems reasonable not to start "from scratch". Furthermore, TuCSoN is a tuple-based coordinaion model and infrastructure, that is, a coordination artefact gaining increasing attention both from the academic world and from the industry

The two categories have been named "TuCSoN as it is" and "TuCSoN as it will be", which basically means the following:

- in "TuCSoN as it is" you will work on the ready-to-use TuCSoN distribution, improving/adding/fixing some details left behind during time
- in "TuCSoN as it will be" you will start redesigning ("from scratch") specific, well-delimited TuCSoN pieces based on its formal model as it is described in TuCSoN papers

What is the difference? The crucial point is that in the first case you deal with already existing code, thus will eventually deal with refactorings, dependency analysis, conflicts resolution; whereas in the latter case, you should face the issues of translating a model or, at best, an abstract architecture and an operational semantics, into code, carefully choosing which assumptions to make. Trust me: neither of the two is easier than the other.

TuCSoN "as it is"

!!! UPDATE !!!

Although GoogleCode still hosts TuCSoN source code, a new repository has been set up on Bitbucket: please clone TuCSoN code from this repository (default branch, which is 'master' branch, is fine) and work on a local branch named as follows ' $surname1- $surname2...' (if you don't know what I'm talking about, email me asap)

!!! UPDATE !!!

NB: For the projects below, usage of Eclipse, Checkstyle and FindBugs is strongly suggested

- Asynchronous Agents Behaviour in TuCSoN: pro-active notifications
- Fault-tolerant TuCSoN: persistency & recovery related aspects
- Naming: Agents, Tuple Centres and Beyond
- Toward Mobile-TuCSoN: minimizing TuCSoN memory footprint
- Toward Mobile-TuCSoN: network-awareness and virtual spatial-situatedness in TuCSoN
- TuCSoN Debug & Inspection: Properly implement logging facilities
- TuCSoN Debug & Inspection: inspecting the ReSpecT VM
- TuCSoN on Cloud: "Elastic" deployment of TuCSoN nodes in the Cloud
- TuCSoN on Cloud: User interface and accounting for TuCSoN nodes in the Cloud
- Exception Handling in Middlewares: the case of TuCSoN

TuCSoN "as it will be"
NB: For the projects below, usage of Eclipse, Maven, Checkstyle and FindBugs is mandatory

- TuCSoN coordination artefacts: tuple spaces type hierarchy
- TuCSoN nodes: identity and reference, naming and location services
- TuCSoN communication language: abstract tuples