ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D6.1 (WP 6): Reliable Sim-Diasca Simulation

Due date of deliverable: Month 18 (March 2013)
Actual submission date: Month 22 (July 2013)

Start date of project: 1st October 2011 Duration: 36 Months
Lead Contractor: Heriot-Watt University Revision: 0.1

Purpose: The overall objective is to prepare the Sim-Diasca simulation engine for the use of the larger-scale computing infrastructures targeted by RELEASE; such massive number of cores requires that various reliability mechanisms be added. This includes notably the "k-crash resilience" feature described here.

Results: Now, since the 2.2.0 version of Sim-Diasca, the user is able to specify a value k corresponding to the maximum number of computing hosts whose failure in the course of simulation may be overcome.

Conclusion: Even if some restrictions still apply (some of which being scheduled for removal in subsequent Sim-Diasca versions), at the application-level the engine is ready to take advantage of massive resources with an increased reliability. If additionally taking into account the Erlang-level improvements brought by the other work-packages of RELEASE, we believe the whole solution will be soon fully able to tackle the future demanding larger-scale simulation cases we can currently foresee.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination level			
PU	Public		*
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Reliable Sim-Diasca Simulation

Author: Olivier Boudeville (olivier.boudeville@edf.fr)

Abstract

This document describes the k-crash resilience feature added to Sim-Diasca: why it is useful, and how it is designed, implemented and tested.

1 Introduction

The simulations of complex systems tend to be of a larger scale, and may last (at least in wall-clock time) very long.

Numerous computing hosts will then be involved in such simulations and, even if each of the cores of these hosts boasts a high MTBF, their number will imply that, if waiting for long enough, any proportion of them *will* fail. Not to mention that there *will* be as well network issues, transient or not.

As a result, without a specific mechanism for resilience, some demanding simulations would be (especially in future uses) unlikely to complete at all, limiting the actual maximum scalability that could be obtained for the engine.

This is why a *k-crash resilience* feature has been added to the Sim-Diasca engine, starting from its 2.2.0 version.

This support will be regarded as experimental for a few versions, and it will be enriched over time based on feedback and experience stemming from actual use.

2 Towards the use of SD-Erlang

If Sim-Diasca is more robust thanks to this k-crash resilience feature, hence more prepared for scalability, there will be further changes in that matter, notably as, in a next phase, we plan to integrate SD-Erlang in Sim-Diasca.

Indeed, once SD-Erlang documentation and implementation will be mature enough, the support for s-groups it will provide will remove the need for a fully-meshed connectivity of Erlang nodes, and thus will enable Sim-Diasca to take advantage of a larger number of cores (this is typically an expected prerequisite of a proper use of the upcoming Bluegene/Q port, as its purpose is to increase the resources Sim-Diasca can take advantage of).

For that, Sim-Diasca will have to properly integrate SD-Erlang, i.e. to define suitable s-groups to support its communication needs (message exchange patterns).

First studies on that topic have been started with the University of Kent, to determine how the simulation agents (notably the time manager hierarchy), the various sets of coupled model instances and result producers should rely on relevant s-groups to achieve a seamless, application-level transparent connectivity, even if the corresponding nodes have actually to rely on s-group routing underneath.

As this work is still in early stages, this document will concentrate on the application-specific changes that were needed to implement the k-crash resilience feature, knowing the upcoming SD-Erlang integration will certainly have an impact on these reliability and scalability topics.

3 A Tunable Resilience Service

Now, starting from this resilience-enabled version, at start-up the user of Sim-Diasca is able to specify, among the simulation settings (see, in `class_DeploymentManager.hrl`, the `crash_resilience` field of the `deployment_settings` record), what is the required level of resilience of the simulation, with regard to the loss of computing hosts in its course:

- either *none* is required (the default mode), in which case the simulation will crash as soon as a computing host is deemed lost (while, on the other hand, no resilience-induced overhead will exist in this case)
- or a positive integer **k** is specified, which designates the maximum number of simultaneous losses of computing hosts that the simulation will be able to overcome (a safety net which, of course, will imply some corresponding overhead)

For example, if `k=3`, then as long as up to 3 computing hosts fail at the same time during a simulation, this simulation will be nevertheless able to resist and continue after a short (bounded) automatic rollback in simulation-time.

This will include starting again from the last simulation snapshot (established automatically at the latest wall-clock simulation milestone, if any was met), converting back the states of simulation agents, actors and result producers (probes[1]) which had been then serialised, re-dispatching (load-balancing-wise), re-creating, updating and linking the corresponding processes, before making the simulation time progress again from that milestone.

There is no upper bound in the number of *total* losses of computing hosts in the simulation that can be overcome, provided that at any time the k threshold is not exceeded[2] and that the remaining hosts are collectively able to sustain the resulting load.

This last point implies that the resources of a fault-tolerant simulation *must* exceed the strict needs of a simulation offering no resilience. For example, if N homogeneous hosts are assigned to a k-resilient simulation, then the user must ensure that the simulation *can* indeed fit at the very least in N - k computing hosts, otherwise there is no point in requesting such a resilience.

Note that this resilience applies only to the random "workers", i.e. to the average computing hosts. For example, if the host of the root time manager is lost, the simulation will crash, regardless of the value of k. Depending on the optional services that are enabled (ex: performance tracker, data-logger, data-exchanger, etc.) other single points of failures may be introduced, as

[1] Their serialization support is currently only experimental.

[2] Currently, hosts that departed the simulation cannot join back. As a consequence, the remaining ones must be able to cope with the load. Therefore the simulation user ought to allocate more resources than strictly necessary initially, to compensate for the *sum* of all later losses, as they will not be redeemed. The extra hosts introduced in this case behave as spare ones, except that they do not remain idle until a host crashes: they participate to the simulation from its very start, to further smooth the computing load.

they tend to be deployed on different hosts (trade-off between load-balancing and resilience)[3]. Currently, depending on the aforementioned enabled services, very few single points of failure remain (typically up to three, compared to a total number of computing hosts that may exceed several hundreds, if not thousands in the future).

Similarly, should a long-enough network split happen, the k threshold may be immediately reached. If running on production mode, extended time-outs should provide a first level of safety.

Currently no support is offered for hosts that would join in the course of the simulation to compensate for previous losses (for example if being rebooted after a watchdog time-out): usually dynamic additions are not in line with the practice of cluster job managers (it is simpler and more efficient to directly use a larger number of nodes upfront).

Besides the resilience level (i.e., the number k), a (possibly user-defined) serialisation period will apply. It corresponds to the lower bound in terms of (wall-clock[4]) duration between two serialisations (default duration is two hours).

This serialisation activity will run even before the simulation is started, so that even simulations needing a long time to recreate their initial state benefit from some protection (typically such a serialisation will then happen at the very first evaluated diasca).

Finally, we ensured that this serialisation activity does not introduce a non-negligible latency (whether activated or not) - but, of course, once the regular serialisation is triggered, the whole simulation is bound to be stalled for some time (even if it is done in a quite parallel, distributed way). As a consequence, the resilience feature is only compatible with the batch mode of operation (i.e. not with the interactive one).

[3] Most, if not all, services could be made resilient; we simply started with the key ones. As we are able to store most of the reproducible simulation state and reconstruct the purely technical, transient information, a given simulation might even in the future survive the loss of *all* its computing nodes, and restart later from a blank state, just based on its serialisation data.

[4] Since hardware and software faults are ruled by wall-clock time; simulation time may flow in a very different manner.

4 Mode of Operation

4.1 Preparing for any later recovery

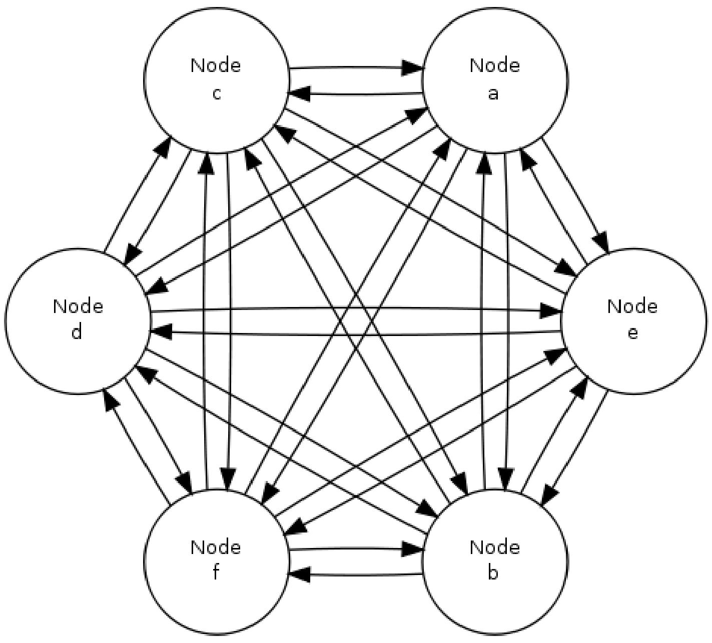
Let's suppose that N computing hosts are assigned to a simulation having to exhibit a k-crash resiliency.

This resilience service is implemented internally by first establishing a "k-map", which determines, for each of the N hosts, which of the k other hosts it backs-up and, reciprocally, which hosts back it up.

For example, if N=6, hosts may be `[a,b,c,d,e,f]`, and the k-map could then be, for a requested resilience level of k=5:

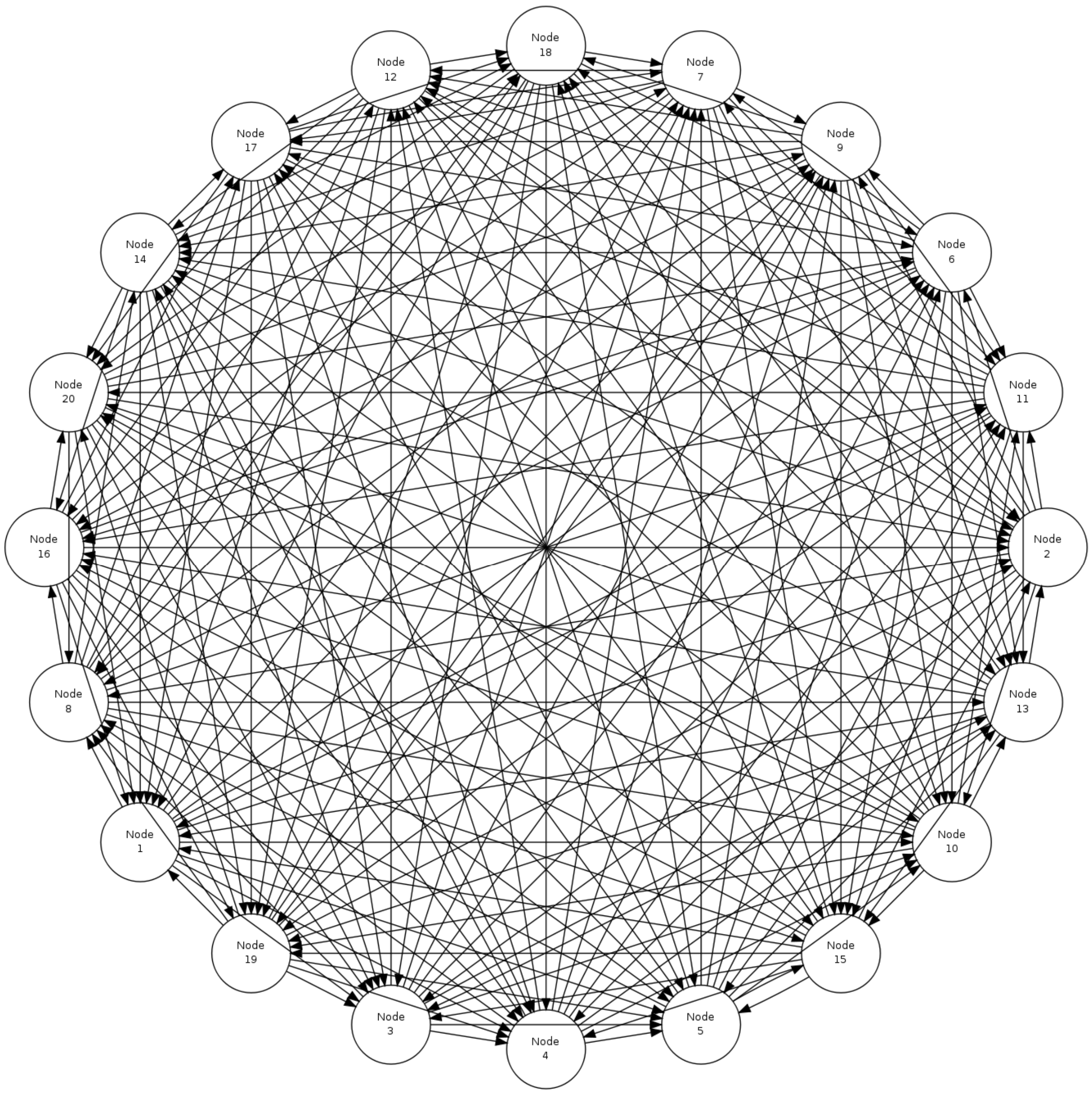
```
For a resilience level of 5, result is: k-map for 6 nodes:
+ for node a:
 - securing nodes [b,c,d,e,f]
 - being secured by nodes [f,e,d,c,b]

+ for node b:
 - securing nodes [c,d,e,f,a]
 - being secured by nodes [f,e,d,c,a]

+ for node c:
 - securing nodes [d,e,f,a,b]
 - being secured by nodes [f,e,d,b,a]

+ for node d:
 - securing nodes [e,f,a,b,c]
 - being secured by nodes [f,e,c,b,a]

+ for node e:
 - securing nodes [f,a,b,c,d]
 - being secured by nodes [f,d,c,b,a]

+ for node f:
 - securing nodes [a,b,c,d,e]
 - being secured by nodes [e,d,c,b,a]
```

This example corresponds to, graphically (see `class_Resilience_test.erl`):



Topological view of mesh 'Resilience 5-map for 6 nodes'

Of course this resilience feature is typically to be used with a far larger number of nodes; even with a slight increase, like in:



Topological view of mesh 'Resilience 10-map for 20 nodes'

one can see that any central point in the process would become very quickly a massive bottleneck.

This is why the actual work (both for serialisation and deserialisation tasks) is done in a purely distributed way, and exchanges are done in a peer-to-peer fashion, using the fastest available I/O for that[5], while the bulk of the data-intensive local work is mostly done in parallel (taking advantages of all local cores).

To ensure a balanced load, each computing host is in charge of exactly k other hosts, while reciprocally k other hosts are in charge of this host. After failures, the k-map is recomputed accordingly, and all relevant instances are restored, both in terms of state and connectivity (yet, in

[5] This includes tuned file writing and reading, operating on stripped-down binary compressed content, and relying on zero-copy `sendfile`-based network transfers.

the general case, on a different computing host), based on the serialisation work done during the last simulation milestone.

4.2 Actual Detailed Course of Action: from serialisation to rollback

Setting up the resilience service is a part of the deployment phase of the engine. Then the simulation is started and, whenever a serialisation wall-clock time milestone is reached, each computing host disables the simulation watchdog, collects and transforms the state of its simulation agents, actors and result producers (including their currently written data files), and creates a compressed, binary archive from that.

Typically, such an archive would be a `serialisation-2503-17-from-tesla.bin` file, for a host named `tesla`, for a serialisation happening at the end of tick offset 2503, diasca 17. It would be written in the resilience-snapshots sub-directory of the local temporary simulation (for example in the default `/tmp/sim-diasca-<CASE NAME>-<USER>-<TIMESTAMP>-<ID>/` directory).

This archive is then directly sent to the k other hosts (as specified by the current version of the k-map), while receiving reciprocally the same type of information from k other hosts. One should note that this operation, which is distributed by nature, is also intensely done in parallel (i.e. on all hosts, all cores are used to transform the state of local instances into a serialised form, and the two-way transfers themselves are made in parallel).

Then, as long as up to k hosts fail, the simulation can still rely on a snapshot for the last met milestone, and restart from it (provided the remaining hosts are powerful enough to support the whole simulation by themselves).

The states then collected require more than a mere serialisation, as some elements are technical information that must be specifically handled.

This is notably the case for the PIDs that are stored in the state of an instance (i.e. in the value of an attribute, just by itself or possibly as a part of an arbitrarily complex data-structure).

Either such a PID is the one of a lower layer (Common, WOOPER or Traces), or it is related directly to Sim-Diasca, corresponding typically to a simulation agent of a distributed service (ex: a local time manager, data exchanger or instance tracker), to a model instance (an actor) or to a result producer (a probe).

As PIDs are technical, contextual, non-reproducible identifiers (somewhat akin to pointers), they must be translated into a more abstract form prior to serialisation, to allow for a later proper deserialisation:

- Lower layers are special-cased (we have mostly to deal with the WOOPER class manager and the trace aggregator)

- Simulation agents are identified by `agent_ref` (specifying notably the service they implement)
- Model instances are identified by their `AAI` (*Abstract Actor Identifier*), a context-free actor identifier we already need to rely upon for reproducibility purposes, at the level of the message-reordering system
- Probes are identified based on their producer name (as a binary string); the data-logger service is currently not managed by the resilience mechanisms

In the case of the probes, beyond their internal states, the engine has to take care also of the data and command files they may have already written on disk.

The result of this full state conversion could be stored on the k nodes either in RAM (with an obvious constraint in terms of memory footprint), but storing these information instead in dedicated files offers more advantages (but then a two-way serialisation service is needed).

For that we defined a simple file format, based on a header (specifying the version of that format) and a series of serialised entries, each of them being made of a type information (i.e. serialisation for a model instance, a probe instance or an agent instance) and a content, whose actual format depends on that type. The full specification of the format is documented in `class_ResilienceAgent.erl` and in the sources of each serialised type.

Multiple steps of this procedure are instrumented thanks to WOOPER; notably:

- Once, with the help of the time manager, the resilience manager determined that a serialisation shall occur, it requests all its distributed resilience agents to take care of the node they are running on
- To do so, each of them retrieves references (PID) of all local actors (from the corresponding local time manager), local simulation agents and local probes (from the result manager); then each of these instances is requested to serialise itself
- Such a serialisation involves transforming its current state, notably replacing PID (that are transient) by higher-level, reproducible identifiers (the conversion being performed by a distributed instance tracking service); for that, the underlying data-structure of each attribute value (ex: nested records in lists of tuples containing in some positions PID) is discovered at runtime, and recursively traversed and translated with much help from nested higher-order functions and closures; it results finally into a compact, binary representation of the state of each instance
- On each node (thus, in a distributed way), these serialisations are driven by worker processes (i.e. in parallel, to take also advantage of all local cores), and the resulting serialised content is sent to a local writer process (in charge of writing the serialisation file), tailored not to be a bottleneck; reciprocally, the deserialisation is based on as

many parallel processes (for reading, recreating and relinking instances) as there are serialisation files to read locally

A few additional technical concerns had to be dealt with this resilience feature, like:

- The proper starting of Erlang VMs, so that the crash of a subset of them could be first detected, then overcome (initial versions crashed in turn; using now `run_erl/to_erl`)
- The redeployment of the engine services onto the surviving hosts; for example, the loss of nodes used to result in reducing accordingly the number of time managers, and thus in merging their serialised states; however this mode of operation has not been kept, as the random state of these managers cannot be merged satisfactorily in that case (to preserve reproducibility, model instances but also time managers need to rely on the same separate, independent random series as initially, notwithstanding the simulation rollbacks)
- Special cases must be accounted for, as crashes may happen while performing a serialisation snapshot or while being already in the course of recovering from previous crashes; much implicit information must be taken care of (ex: the random state of instances that may be stored in the process dictionary), either by storing them or by recreating them during a rollback

5 Testing

Most of the testing was done by specifying more than one computing host, and emulating first the simultaneous crashes of all other hosts at various steps of the simulation. This is to be done either by unplugging the Ethernet cable of the user host or, from a terminal on that host, running as root a simple command-line script like[6]:

```
$ while true ; do echo "Disabling network" ; ifconfig eth0 down ; \
  read ; echo "Enabling network..." ; dhclient eth0 &&           \
  echo "...enabled"; read ; done
```

(hitting Enter allows to toggle between a functional network interface and one with no connectivity)

For a better checking of this feature, we then relied on a set of 10 virtual machines (`HOSTS="host_1 host_2..."`) on which we simply:

- Updated the distribution with the right prerequisites: `apt-get update && apt-get install g++ make libncurses5-dev openssl libssl-dev libwxgtk2.8-dev libgl1-mesa-dev libglu1-mesa-dev libpng3 gnuplot`
- Created a non-privileged user: `adduser diasca-tester`
- Built Erlang on his account: `su diasca-tester` ; `cd /home/diasca-tester && ./install-erlang.sh -n`
- Recorded a public key on each of these 10 computing hosts:

  ```
  $ for m in $HOSTS ; do ssh diasca-tester@$m \
  'mkdir /home/diasca-tester/.ssh &&          \
  chmod 700 /home/diasca-tester/.ssh' ; scp   \
  /home/diasca-tester/.ssh/id_rsa.pub         \
  diasca-tester@$m:/home/diasca-tester/.ssh/authorized_keys;
  \
  done
  ```

- Ensured the right version of the Erlang VM is used:

  ```
  $ for m in $HOSTS ; do ssh diasca-tester@$m  \
  "echo 'export PATH=~/Software/Erlang/Erlang-current-
  install/bin:\$PATH' \
  ```

[6] Regarding the emulation of connections losses:

- `ifup` and `ifdown` are a lot less appropriate than `ifconfig` for that, notably as they apparently remove route definitions and DNS settings. Moreover even `ifdown --force eth0` may fail to stop a currently used interface (`SIOCDELRT: No such process`)

- the `dhclient` call here is not necessary for the current simulation to resume, but it is for the next launch, which will need DNS resolution

```
        | cat -  ~/.bashrc > /tmp/bash-erl &&        \
        /bin/mv -f /tmp/bash-erl ~/.bashrc"
```

This command is a tad complex, as some default `~/.bashrc` include:

```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

So the path must be specified at the *beginning* of the file, rather than later.

Simulations can then run on the user host and the 10 additional ones.

Then their failure can be simulated from the command-line, using tools provided by the vendor of the virtual infrastructure (ex: `VBoxManage controlvm` with VirtualBox, with VMWare vSphere command-line interface, etc.) or UNIX brutal kills through SSH.

Of course once the initial testing and troubleshooting has been done thanks to this setting, real-life situations (involving notably network links to be unplugged at random moments while a simulation is running) must be reproduced.

As sneaking into an HPC control room in order to perform selective sabotage on the relevant cables is not really an option, such a testing is better be done on a simple ad-hoc set of networked computers.

There may still be very specific corner cases which would prevent a proper rollback, but we believe we handled most of the recoverable failure cases that could happen in real use.

One should also note that this very young k-crash resilience feature (that we still consider as experimental) will be significantly impacted by the next feature we plan to integrate this year (WOOPER 2.0, as the change in the internal instance representation will require the serialisation/deserialisation to be updated accordingly).

6 Future Improvements

Many enhancements could be devised, including:

- Increasing the compactness of serialisation archives (alleviating in turn the network transfers)
- Tuning the resilience mechanisms thanks to larger-scale snapshots, to identify the remaining bottlenecks (profiling the whole serialisation process, meant to happen a lot more frequently than its counterpart deserialisation one)
- Enabling more simulation services to resist crashes: besides time management, we might in the future investigate data-logging, more complete result management, performance tracking, etc.
- Allowing for a "cold start", i.e. restarting from only serialisation files (while Sim-Diasca is not running), even though collecting them post-mortem on various computing hosts is not nearly as convenient as having the engine perform directly an automatic, live rollback (which might even remain unnoticed from an unwary user)
- Applying a second pass of load-balancing, onto the serialised actors (this would probably require implementing actor migration, which is almost already there), if the post-rollback computing and network load was found too uneven in some cases

More generally, additional resilience-related features, extra testing and maturation time will be needed until these brand new mechanisms can be safely applied to more production-minded matters.

Anyway, to the best of our knowledge, at least for civil applications, there are very few other discrete time massively parallel and distributed simulation engines, and we do not know any that implements resilience features akin to the one documented here, so we already benefit from a pretty advanced solution.

7 Change Log

0.1 First version.

- Detailed the links with SD-Erlang and the upcoming work regarding s-groups