sams: Core Manual

Nicolas Berthier
mail@nberth.space

November 3, 2016

sams stands for ”Synchronous Autonomic Management Software”; it is a pioneering prototype exemplifying the use of reactive design techniques for autonomic management. This manual describes the main concepts and internal structure of sams. It details its compilation and run-time dependencies, as well as the various steps required to actually build it.

1 Introducing sams

1.1 Purposes

sams is a proof-of-concept tool demonstrating the use of reactive design techniques in the context of autonomic management systems. For now, it is designed to handle single-layer applications: the *managed system* only involves sets of services (*e.g.,* processes running on remote virtual or physical machines). Yet, its architecture is extensible enough to allow the manipulation of other kinds of resources present at other virtualization levels, such as sets of *hypervisors* or sets of *blades*.

The internal logic encoding the management decisions of a particular deployment is encapsulated in a *tick*, and is programmed using *synchronous languages*.

1.2 sams' Distributed Architecture

First, lets fix some vocabulary before detailing the architecture of the software.

1.2.1 Preliminary Note on Terminology

In the following, we refer to several hardware and software components using the following terminology: *hosts* are virtual or physical machines running an operating system, and *nodes* refer to a triple gathering a host, a user, and a location in the file system of that host (yet, as remote file system locations are partly determined internally by sams, nodes will be referred to as couples $\langle \text{user}, \text{host} \rangle$); a whole *application* makes use of distinct *resources* (*e.g.,* in the n-tier application case, a *tier* per *service* it makes use of), each offered by *tasks* executing on nodes (*e.g.,* a task may gather zero or more *UNIX processes* altogether); as these tasks may be replicated on multiple hosts for performance or availability purposes, a resource may consist of one or more nodes. A *deployment* refers to a running application, plus its *management infrastructure* whose *decision logic* describes the behavior.

1.2.2 Agent-based Software, and A^3 Properties

An agent-based software is made up of a set of interacting *agents*. Agents are *reactive objects* communicating with each other by sending *notifications*. Upon their reception, agents *consume* them by executing their own `react()` method.

Reactions are *atomic*, meaning that they are either executed successfully or not executed at all (*i.e.,* side effects on the encapsulated data that constitute the state of the agent are not performed). If a reaction fails (*e.g.,* by throwing an exception), the notification that triggered it is not consumed and is then delivered again (hence it is never lost). For fault tolerance purposes, the state of agents (as well as all internal queues of notifications) may be made persistent, thereby survive hardware crashes not impacting the storage infrastructure.

Communications between agents are encapsulated in *transactions*, thus process failures do not lead to losses of notifications. Last but not least, *delivery* of notifications (hence their respective reactions) fulfills *causal ordering*:

- communications respect FIFO ordering between two agents; and
- if A, B and C are distinct agents, A sends a notification ab to B and ac to C, and B in turn reacts to ab by sending a notification bc to C, then ac is delivered to C **before** bc.

Agents are deployed and executed inside *agent servers*, *i.e.,* (isolated) portions of a JVM. In the sequel, agent servers may be referred to as *processes* as well (even though a single JVM may host several of them).

1.2.3 Overall Management Infrastructure

The management infrastructure for an application deployed with sams consists of:

- a *center process* centralizing the code taking all or most autonomic management decisions. This process bootstraps the deployment of the whole application, and features a basic console for monitoring purposes.

  This process incorporates an executable model of the system called the *tick*, and encoding the autonomic management decision logic for the whole managed system;
- one *remote process* per deployed task of the application (recall that a *task* is the name given to the set of computing units — *e.g.,* Unix processes — of a particular node, itself dedicated to a single resource). Remote processes are responsible for acting upon the application's set of computation resources (*e.g.,* starting and stopping processes, configuring), and monitoring the status of their respective nodes.

Note there could be several tasks assigned to a single host, hence several remote processes monitoring it, yet this peculiarity would need to be described in some way when specifying the decision logic (*i.e.,* programming the tick).

1.3 Implementation Aspects

More practically, sams essentially consists in a core set of scripts used internally, *e.g.,* to execute remote commands or transfer files remotely, plus Java classes that need to be specialized and instantiated in order to build a working management software. To do so, an *application-specific* part needs to be defined based on this *core*.

The Java classes constituting the core are organized in three packages:

`fr.lig.erods.sams.center` gathers all classes specific to the center process;

`fr.lig.erods.sams.depl` gathers classes needed by the remote processes only;

`fr.lig.erods.sams.common` gathers all classes common to the center an remote processes, and notably the types of objects that need to travel between them.

Before detailing the ways provided to specify a complete management infrastructure, some further concepts and utilities need to be presented.

1.3.1 Artifacts

Problem In sams, most communication patterns among agents involve sequences of high-level *instructions*. These instructions encode rather abstract operations that may involve multiple execution sites (agent servers), such as the startup of a remote agent server, the transfer of some files to a remote location, or the reconfiguration of a remote task. They are effectively performed by (sets of) methods and scripts, whose executions are triggered through the `react()` method of agents. (These agents can easily be seen as communicating automata whose transitions are triggered by the reception of notifications.)

Hence, in addition to the state of the current resource to manage (itself encapsulated into the internal data of the managing agent), each transition (a method executing a portion of instruction) must be given a *context* (some sort of *closure*). This context carries any data that needs to be transmitted between the successive calls to `react()` methods so as to perform high-level instructions about a particular entity (*e.g.,* resource, node, configuration data).

As a result, some local variables (the context) most often need to be kept between reactions. When programmed in any imperative language that do not provide any convenient support for the encoding of such contexts (most of them, actually), these code patterns lead to the *stack ripping* symptom.

Adopted Solution To circumvent this limitation, sams makes a rather peculiar use of A^3 notifications, by using "dynamic" records to store temporary data in the notifications themselves: called *artifacts*, each one of them carries a mapping from strings to any serializable value.

Once such a notification is instantiated, a mapping is also created that can then be filled with any named data needed to complete the sequence of instructions. These data may be identifiers, configuration parameters, or any other data that an agent needs to recall between successive calls to its `react()` method to execute an instruction.

Artifacts can be *chained* to convey a memory about the causes that lead to the execution a `react()` method.

Artifacts are instances of the `fr.lig.erods.sams.common.Not` class.

1.3.2 Acknowledgment Timeouts

The artifacts presented above also feature facilities to manage acknowledgments, including the ability to arm timeout notifications. Of course, this feature contrasts with the completely asynchronous nature of the A^3 middleware. However, this kind of mechanisms is required when one tries to detect hardware crashes that may lead to completely unresponsive remote agent servers.

See the source code of `fr.lig.erods.sams.common.Not` class for further details.

1.3.3 Identifier Sets

Concise and persistent (*i.e.,* non mutable) sets of integers are implemented in class `fr.lig.erods.sams.common.util.ISet`; sets of this kind are extensively used in sams, for gathering identifiers (*e.g.,* of remote processes) or allocation management.

1.3.4 Configuration Files Format & Environments

All configuration files involved in sams are essentially Java properties files. These files are basically used to build environments (mappings from *key* strings to *value* strings) representing internal data and configuration values for the resources.

Environments are built by loading property files successively and in a determined order, always overriding every already assigned keys. Some keys are also assigned internally, depending on the resource the environment is attached to.

Variable Expansion Environment values are subject to variable expansion *when they are effectively used*; values may contain references to other values in the form of `${k}` for a key `k`. Given an environment consisting in a mapping from keys to internal values, then the *effective value* of any key in this environment is the internal value associated to the key, with all its variable references recursively substituted with the effective values of the corresponding keys in the same environment[1].

[1]See the documentation on `http://commons.apache.org/proper/commons-lang/javadocs/api-release/org/apache/commons/lang3/text/StrSubstitutor.html` for more details.

Field Groups Some internal mechanisms and configurations in sams make use of *field groups* (that can be seen as some sort of *records*). A field group g in an environment is specified by a specification key g`.fields` whose value is a comma-separated list of associated fields F. The value of the field group g is then the sub-environment mapping all keys $g.f$ to strings, with $f \in F$.

An example field group named `example` and with three fields would be:

```
example.fields a, b, c
example.a value of a
example.b value of b
example.c value of c
```

These field groups notably serve to transfer configurable sets of keys from an environment to an artifact, and *vice versa*. The value of field group specification keys are subject to variable expansion, as well as the values of its fields.

See the source code of the class `fr.lig.erods.sams.common.Env` for further details.

1.4 Defining Application-specific Parts

An executable instance of sams is application-specific; *i.e.,* such a software gathers sams' core along with an *application-specific part*.

1.4.1 Operating Code Specification

The *operating code* is a combination of agents capable of performing the sequential operations that needed to monitor and change the state of the resources the application makes use of. These agents encapsulate internal memory for this purpose, and communicate by using chained artifacts carrying any data relevant to the instructions being executed.

At the moment, sams' core only provides some basic support to write operating code executing resource management instructions, among which the artifact and the environments described above. (Note however that a set of multi-tier-application-specific classes are provided with sams' core for the specifications of operating code in these cases; they are presented in the manual entitled "Multi-tier Application Management with sams by Example"[2]).

1.4.2 Decision Logic Specification

In sams, the decision logic is encoded in what is called a tick: it is a piece of sequential code, encapsulated in an object featuring at least the following methods:

`step(I)` updates the internal state of the object according to the input vector `I`; the results of its execution is a vector `O` of outputs.

The inputs of this method consist of:

[2] `http://sams.gforge.inria.fr/manual-ntier-by-example.pdf`

- up-to-date (at best) measures of values related to the managed resources (*e.g.,* CPU loads, number of new failure occurrences since the last execution of `step()`);
- impulses that notify events occurring on the resources (*e.g.,* new detected failure(s), end of management operations, commands from the user);

`reset()` sets the internal state of the object to the initial one.

1.4.3 Putting it All Together: The Model

This operation boils down to ”link”, or ”branch”, the tick encoding the decision logic, with the corresponding sequences of instructions in the operating code.

The *decision code* is then an agent whose state comprises a tick object, and whose reaction is in charge of calling its `step()` method with appropriate inputs and at *relevant instants*. It does so based on dedicated notifications it receives from the operating code. It also interprets the output vector resulting from the execution of the tick, and translates commands it encodes into artifacts sent to agents of the operating code.

The combination of the decision code with the associated operating code is called the *model*.

In Practice In sams, the set of agents constituting the operating code executing on the center, as well as its associated decision code, are directly specified and paired by using Java code; this is the role of classes implementing interface `fr.lig.erods.sams.center.ModelBuilder`. These classes must feature a `build()` method returning:

- a set of agents constituting the operating code (these agents must not have been deployed yet);
- an instance of a subclass of `fr.lig.erods.sams.center.DecisionCode`, *i.e.,* an agent associated to the operating code for resources to be managed that are modeled by the tick it encapsulates.

The types of tick objects implement interface `fr.lig.erods.sams.center.Tick`.

1.4.4 Task Specification

Every remote task of the application is paired with an agent server communicating with sams' center process (hence, there can theoretically be several agent servers hosted on one single host — even on a single JVM, actually). This agent server hosts an actuator, plus possibly some sensors:

- the *task agent* is an actuator that is in charge of executing commands remotely (and deploying and managing remote sensor agents);

- the *sensor agents* report on the status of the remote host and the task itself, by sending measures or impulses (possibly indirectly) to the decision code, either periodically (for timed monitors) or on demand.

In Practice Task agents are subclasses of `fr.lig.erods.sams.depl.opcode.Task` (that are basically able to receive notifications related to the management infrastructure). Timed monitors are subclasses of `fr.lig.erods.sams.depl.sensors.TimedMonitor`, that periodically build named measure notifications and send them to a specified destination agent (typically, a dedicated multiplexer on the center — see bellow); those measure notifications carry a name identifying their source sensor, and the value effectively measured. (No sensor responding on demand is available in sams yet.)

Measures and impulses are instances of `fr.lig.erods.sams.common.opcode.Measure` and `fr.lig.erods.sams.common.opcode.Impulse` respectively.

1.4.5 Multiplexing Measures and Heartbeats

The role of multiplexer agents are to gather specific types of measures and impulses, interpret them, and send notifications abstracting these input data. For instance, a multiplexer can compute and send the average measure of all the inputs it receives from distinct sources, whenever it receives a new value. Alternatively, it can account for the impulses it receives (heartbeats), and periodically report on missing ones. At last, multiplexers can perform both of these tasks by directly accounting for measure notifications.

In Practice Multiplexers are subclasses of `fr.lig.erods.sams.common.opcode.Muxer`; `fr.lig.erods.sams.common.opcode.HBMuxer`s are multiplexers accounting for measures and generating custom notifications when some are missing for a given time.

2 Building and Using sams' Core

2.1 Source Code Directory Structure for sams' Core

sams' source code is available for download at `https://gforge.inria.fr/projects/sams/`. The root of the source code directory contains a build file for Ant (`build.xml`), a properties file gathering some build-related configurations (`build.properties`), plus licensing information. Its sub-directories are organized as follows:

`etc/` Contains default configuration and miscellaneous files common to all deployments of sams:

  `etc/a3debug.cfg` Configuration of the logging system;

  `etc/build/` Extra Ant files related to the compilation of sams' core;

`doc/` Documentation directory for the project;

`src/` Contains the source code of the core part of sams:

- `src/java/` sams' core Java source code;
- `src/bin/` This directory contains the `sams-start` script, that launches sams' center agents on the machine where it is executed, plus a basic console; see bellow for usage information;
- `src/libexec/` This directory contains shell scripts that are used internally by sams to deploy remote tasks and data, as well as to start distant agents; these scripts mostly encapsulate remote tasks execution mechanics (by means of `ssh` and `scp` commands), and maintain a basic deployment database. They support concurrent executions;
- `src/etc/` Miscellaneous files used internally by sams' core, most notably:
  - `src/etc/sams.properties` Default values for internal variables of sams' center process. This file defines various values such as shell commands used internally for the deployment of applications (notably using scripts originating in `src/libexec/`), deployed files, and default port maps;
  - `src/etc/a3servers.xml` Initial configuration for the agent server of sams' center process;
  - `src/etc/build.properties` Configuration file related to the compilation of sams' core archives;
- `src/remote/` Contains two sub-directories:
  - `src/remote/bin/` Scripts that are deployed on remote nodes and executed internally;
  - `src/remote/etc/` Configuration files that are deployed on remote nodes and used internally;

`examples/` Directory gathering several examples illustrating the use of sams' core to manage multi-tier applications.

2.2 Building sams' Core

Dependencies and compilation instructions of sams' core are described in this Section; notice however that some additional dependencies are needed to compile the examples given in the distribution.

2.2.1 Dependencies

The dependencies needed to compile and execute sams' core are listed below. Note these programs and libraries should be available as packages in any good GNU /Linux distribution.

(Dependencies marked with * are run-time dependencies of the center process only, hence are solely necessary on the managing node; dependencies marked with ** are run-time dependencies of the remote processes as well, and should also be installed on remote nodes).

The following dependencies are needed to compile the core:

JDK Any Java development kit (version $\geq$ 1.5) should be able to compile the source code;

Ant The Ant tool is needed to build the project. The oldest version tested is 1.8;

JORAM A small portion of the Java Open Reliable Asynchronous Messaging (JORAM) message oriented middleware is used internally by sams: the A^3 distributed execution engine. Additionally, sams uses the Monolog library, which is included in JORAM distributions. JORAM is available at `http://joram.ow2.org/` under the LGPL license;

Apache-Commons (lang3) This library provides useful extensions for the Java standard library (see `http://commons.apache.org/proper/commons-lang/`). The corresponding package in GNU /Linux distributions is usually named `apache-commons-lang3`.

JLine The basic console and colored outputs of sams make use of the JLine library available at `http://jline.sourceforge.net/`;

lockfile-progs or procmail * As its name says, the lockfile-progs tool-set provides handy commands to lock files, such as `lockfile-create` and `lockfile-remove`. If those programs are unavailable, sams will try to use the similar `lockfile` program provided by procmail (if none are usable, it will report an error).

bash ** Some scripts use bash constructs, not only the POSIX subset.

Environment Setup Environment variables can be setup so that the build system can determine the location of required dependencies:

- `JORAM_HOME` must be setup so that `a3-common.jar` and `a3-rt.jar` can be found in `$JORAM_HOME/ship/bundle/`, and `monolog.jar` can be found in `$JORAM_HOME/ship/lib/`;
- If the file `commons-lang3.jar` is not located in `/usr/share/java/`, then the `APACHE_COMMONS_LANG3_JAR` environment variable is used to find it;
- Similarly to the previous case with `jline.jar` and `JLINE_JAR`.

2.2.2 Compilation

To compile sams' core distribution, run `'ant'` into the root directory of its source code (called `sams-core/` in the sequel). By default (Ant target `dist`), this operation should create an `output/` sub-directory containing both a distribution directory (`output/sams-core/`) and an archive of its contents (`output/sams-core.tgz`).

The directory structure of this distribution is the following:

`sams-core/bin/` A copy of the `src/bin/` directory of the source distribution;

`sams-core/libexec/` A copy of the `src/libexec/` directory of the source distribution;

`sams-core/jars/` All Java archives needed to execute sams' center process, plus some copies of system-wide archives (A^3, Monolog, Apache-Commons-lang3 and JLine — the two former are directly reused from JORAM and also included in its distribution);

`sams-core/etc/` Miscellaneous files for:

- compiling sams application-specific parts (`build/` sub-directory);
- executing sams' center process (`a3config.dtd` and `a3servers.xml`) — none of these files should be modified;
- configuring the Monolog logging library (`a3debug.cfg`);

`sams-core/remote/` All files that need to be transferred to nodes so as to start one remote agent server (they are directly referenced by internal environment values defined in `src/etc/sams.properties` in the source distribution).

Building Java Documentation Executing '`ant doc`' builds the Javadoc documentation for sams' core classes. These files are placed under the `output/doc/` directory.

Cleanup Executing '`ant clean`' cleans up the temporary compilation files (in the `_build/` sub-directory by default), and '`ant distclean`' further removes the `output/` directory. The `build.properties` file in the root directory of the source code allows to customize some properties of sams' core.

2.3 Executing sams

As stated in Section 1.3, sams' core is useless by itself. In order to use it, one needs to integrate it into application-specific codes and configurations; Section 3 bellow gives details on this point.

Assuming `<sams_root>` is the root directory of sams' core distribution (*e.g.,* `output/sams-core/`) and `app/` and `cfg/` are well-formed application-specific and configuration-specific directories respectively, executing the following command launches sams' center for this application:

```
<sams_root>/bin/sams-start app/ cfg/
```

Then, a rudimentary prompt ('`>`') indicates the successful startup of sams' center. At this point, typing the `<TAB>` key should list all available commands. Among them, `quit` is the only command that is defined by the core itself: it is used to radically stop the center process without any care about remote agents (so, be sure to have stopped any managed deployment before typing it, by using commands that should be defined by the application-specific parts).

3 Structuring Application-specific Parts

Application-specific parts for managing a particular application with sams requires specifying it with two directories:

- The *application-specific directory* gathers the *operating code* and the *decision logic* for this application. It also contains properties files used by sams' core to define internal entities and environments;
- The *application-specific configuration directory* specifies all configuration of the management software, that depend on a particular deployment target (a center node, plus, *e.g.,* a set of clusters).

3.1 Structure of Application-specific Directories

An application-specific directory `app/` **must** be structured as detailed bellow. Note this is a suggested organizational scheme used in the examples, and the location of the elements that are not marked with a `*` can be modified provided sams' core configuration files are updated accordingly.

`app/jars/ *` This directory contains all the Java archives needed to launch and execute the center management process (besides the ones present in `sams-core/jars/` of the core distribution);

`app/lib/ *` This directory may contain extra native libraries for the center process (it is "prepended" to its `LD_LIBRARY_PATH` environment variable);

`app/etc/` Application-specific configuration files:

> **`app/etc/env.properties *`** Configuration values common to the center agent server. This file **must** define the value of a `model-builder.class` field to the class name of the model builder for the application (see Section 1.4.3), as well as a value for `model-driver.class`, used to build an agent defining and relaying commands to the decision code (it is a subclass of `fr.lig.erods.sams.center.ModelDriver`); these classes should be defined in a Java archive located in the `app/jars/` directory;
>
> **`app/etc/tasks.properties *`** Tasks configuration file. For each task *t*, this file specifies a set of fields named `t.f`, defining, among other values, the files and archives to transfer to remote nodes and the class of the agent managing the task. This file may override default values (that can be seen in `src/etc/sams.properties`). To locate files located in (sub-)directories of the application-specific one, values may be defined based on the `app.dir` property (defined internally);

`app/remote/` This directory gathers application-specific files that need to be sent to remote nodes, and in particular:

`app/remote/jars/` Deployed Java archives containing tier-specific management code and agent definitions;

`app/remote/bin/` Any management scripts that need to be executed on the remote nodes;

`app/remote/*.properties` For each task *t*, the `app/remote/`*t*`.properties` file defines task-specific variables that constitute the environment of the remote management agent; for instance, it defines commands to execute to change the state of the task (*e.g.,* to start, stop or configure it) — they are typically calls to a script present in the `app/remote/bin/` directory and that is transferred to the remote nodes executing *t*.

3.2 Structure of Application-specific Configuration Directories

For its part, a configuration directory `cfg/` dedicated to an application specifies topological information about the deployment, mostly in terms of host names. Note that port allocations are based on host names, so the name of two physically or virtually identical hosts should be equal (more precisely, hosts sharing the IP layer must have matching names).

The configuration directory can at least contain the following files:

`cfg/center.properties` Extra environment of the center process; this file may override the value of the `local.host` key, defining the center host. If unspecified, then the internal value for designating the center node will be determined internally.

Additional configuration properties global to the center agent server may be specified in this file.

`cfg/a3debug.cfg` Optional configuration file for the Monolog logging library, overriding the default one provided in sams' core distribution (`sams-core/etc/a3debug.cfg`).