
GinFlow Documentation

Release 1.1.0

Javier Rojas Balderrama, Matthieu Simonin, Cédric Tedeschi

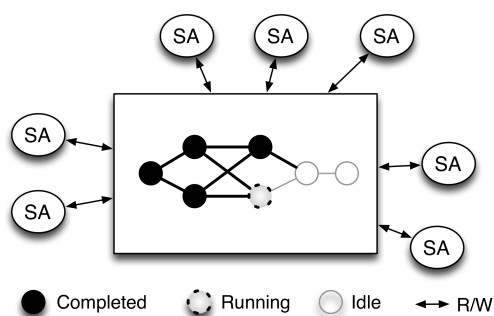
November 23, 2015

1	Background	1
1.1	What is GinFlow ?	1
1.2	Architecture	2
2	Quick Start	5
2.1	Your first workflow	5
2.2	Changing the executor to SSH	6
2.3	Using a configuration file	6
3	User Guide	9
3.1	Command line Interface	9
3.2	Configuration File	10
3.3	JSON workflow description	11
3.4	Adaptation Mecanisms	12
3.5	Java API	14
3.6	Deploying on Grid'5000	14
4	Developer Guide	17
4.1	Source code	17
4.2	Developping with Maven	17
	Bibliography	19

BACKGROUND

1.1 What is GinFlow ?

GinFlow is a decentralised workflow engine. As illustrated on the image below, the GinFlow workflow manager decentralizes the execution coordination of workflow-based applications by relying on an architecture where multiple workflow co-engines, *a.k.a.*, service agents (SAs) coordinate each others through the use of a shared space containing the description of the current status of the workflow execution.



At run time, the shared space gets fed with the description of the initial workflow description, namely, its tasks and the dependencies between them. Also, a set of agents, typically one per task of the workflow get started. These agents, concurrently, will pull a subportion of the space, more precisely the information of their assigned task, process it locally, and trigger the execution of their task if all its incoming dependencies are satisfied. Once the task is terminated, its results are collected by the agent and forwarded to their destination agents in a P2P fashion. The shared space is also rewritten by the agent so as to reflect this termination (and change in the state of the workflow).

The description of the workflow and its interpretation by the agents rely on an adaptive, chemistry-inspired rule-based programming language, namely the *Higher Order Chemical Language* (HOCL). This language relies on concurrent rewriting of a (multiset) space. You can find more information on HOCL here: [\[HOCL\]](#).

1.1.1 Executors

GinFlow exposes a high-level API (either JSON-based or Java code-based) to describe your workflow and lets you run it over various distributed environments. GinFlow supports the following workflow executors:

The *centralised executor* runs your workflow using a single agent containing an HOCL interpreter responsible for the whole coordination and invocation of commands realising the tasks.

The *SSH executor* deploys your workflow over a set of agents and fills the initial shared space. It relies on SSH to connect and start the agents on a set of worker nodes to be specified in the GinFlow YAML configuration file.

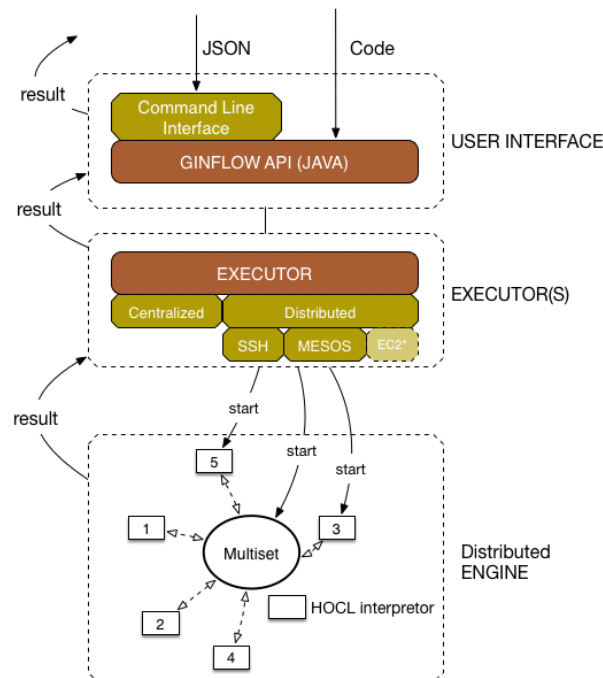
The *Mesos executor* submits your workflow to a dedicated Mesos¹ instance managing your worker nodes and uses Mesos primitives to control the agents.

1.1.2 Message brokers

The communication between agents and between an agent and the shared space relies on a persistent communication middleware. In practice, two message brokers are supported: ActiveMQ² (which is the default broker) and the Kafka³ / Zookeeper⁴ broker (which provides state recovery of an agent in case it fails). Your preferred broker can be chosen at set-up, in the YAML configuration file.

1.2 Architecture

The GinFlow architecture is composed of three layers, illustrated on the figure below :



The GinFlow architecture comprises three layers.

- The top layer is the user interface. It comes in two flavours allowing you to specify your workflow either using a simple JSON file or programmatically using the Java interface.
- The middle layer is the *executor*. It is the GinFlow layer which is responsible for the actual deployment of your workflow on your infrastructure. Three executors are currently proposed:
 - The *centralised executor* starts your workflow in a centralised mode where a single HOCL interpreter is started, fed with your workflow's description and will be responsible for the whole execution, the satisfaction of data dependencies and the actual calls to the commands realising the tasks.

¹ <http://mesos.apache.org/>

² <http://activemq.apache.org/>

³ <http://kafka.apache.org/>

⁴ <https://zookeeper.apache.org/>

- The *SSH executor* deploys your workflow on a set of worker nodes and fills the initial shared space with your workflow description. It relies on SSH connections to start and configure the agents. Once the agents are started and the shared space is initialised, the workflow execution is autonomous.
- The *Mesos executor* allows you to use your own Mesos instance managing your computing nodes. It is interfaced with the Mesos primitives to start and control the status of the agents.
- The bottom layer is the core of GinFlow and implements the workflow agents through a set of HOCL interpreters and a shared space containing an HOCL dynamic description of the workflow. The direct communication between the agents is assured by a persistent message broker. Two message brokers are currently supported: ActiveMQ and Kafka. The former is the default one. The latter provides a state recovery of agents in case of failure. More specifically, if some agent fails during execution, it can be restarted, without any further action from the user: kafka ensures that all the messages will be replayed so as to ensure the workflow will eventually recover. You will find more information on the bottom GinFlow layer in publication [\[FGCS\]](#).

QUICK START

This section will help you to go through the basics of running workflows with GinFlow and is organised in three steps:

- Writing and executing your first workflow
- Changing the centralised executor to the distributed, SSH-based executor
- Using a configuration file

2.1 Your first workflow

In this part you will learn how to create a simple workflow and run it on the central executor.

1. Download the latest jar file from <http://ginflow.inria.fr/download/>.
2. Create your first workflow file *wf.json*. The following example creates a sequence of two services : the first outputs "!" and the second will output "?" concatenated to the output of the first service.

```
{
  "name" : "test-workflow",
  "services": [{
    "name" : ["1"],
    "srv" : ["echo"],
    "in" : ["!"],
    "dst" : ["2"]
  }, {
    "name" : ["2"],
    "srv" : ["echo"],
    "in" : ["?"],
    "src" : ["1"]
  }]
}
```

- Launch it through the following command:

```
java -jar hocl-workflow-X.Y.Z.jar launch -w wf.json
```

- Check the result by having a look at the *wf.json.ginout* file generated at the end of the workflow execution. It contains the HOCL representation of the final state of the workflow. Without worrying too much about HOCL code, notice that it is comprised of the final state of both services in some arbitrary order. Notice in particular the "out":... which contains the output of the service. In case of the second service, it is:

```
.... "out":["! ?"] ...
```

Note finally that, "err":[...] and "exit":[...] artifacts similarly contain the exit code and potential errors collected at run time, respectively (from the execution of the echo command in this particular case.)

2.2 Changing the executor to SSH

You executed your first workflow using the central (by default) executor. In this section, you will learn how to use the SSH executor on your local machine.

- Configure passwordless connection to your local machine.
- Install and start ActiveMQ (typically using the `activemq start` command).
- Run the workflow using the SSH executor:

```
java -jar hocl-workflow-X.Y.Z.jar launch
-w wf.json
-e ssh
-o ssh.user=[your login]
-o ssh.private_key=[path to your private key]
-o ssh.machines=localhost
-o broker=activeMQ
-o activemq.broker.url=tcp://localhost:61616
-o distribution.home=[abs path to the directory containing the GinFlow jar]
-o distribution.name=hocl-workflow-X.Y.Z.jar
```

Let's explain the options :

```
ssh.user=[your login] # This login will be used to connect to the machines
                        specified by ssh.machines

ssh.private_key=[path to your private key] # this private key will be used to
                                           initiate the passwordless SSH
                                           connection to the machines
                                           specified by ssh.machines

ssh.machines=localhost # List of machines that will potentially run invoker
                        services. You can specified several machines in a comma
                        separated list: ssh.machines=machine1,machine2,machine3

activemq.broker.url=tcp://localhost:61616 # address of the activeMQ broker.
                                           This should be reachable by all
                                           the machines in the ssh.machines list.

distribution.home=[abs path to the directory containing the GinFlow binary]

distribution.name=hocl-workflow-X.Y.Z.jar # These two last options help the service
                                           invokers to find the GinFlow jar on
                                           the machines of ssh.machines list.
```

2.3 Using a configuration file

GinFlow can read all these options from a file called “ginflow.yaml” in the current directory :

```
ssh.user: [your login]
ssh.private_key: [path to your private key]
ssh.machines: localhost
broker: activeMQ
activemq.broker.url: tcp://localhost:61616
distribution.home: [abs path to the directory containing GinFlow binary]
distribution.name: hocl-workflow-X.Y.Z.jar
```

It simplifies the command line to the following (you only need to specify which executor you use by using the `-e` option):

```
java -jar hocl-workflow-X.Y.Z.jar launch -w wf.json -e ssh
```


3.1 Command line Interface

3.1.1 *launch*

We already met the *launch* subcommand. It allows you to launch a workflow. Let's list its options:

```
Usage: launch [options]
Options:
  -c, --config
        Config file
        Default: ginflow.yaml

  -d, --debug
        Default: 0

  -e, --executor
        Executor to use, possible values are : central, mesos, ssh
        Default: central

  -h, --help
        Default: false

  -o, --option
        Executor options
        Syntax: -o key=value
        Default: {}

* -w, --workflow
        Path to the json file containing the workflow definition
```

3.1.2 *clean*

The *clean* subcommand can be used to clean service invokers on remote machines. Usually the executor is responsible for cleaning the processes after the execution, but in some rare cases (e.g: connection error to some nodes during the deployment) you can use this subcommand to clean the remaining processes.

```
Usage: clean [options]
Options:
  -c, --config
        Config file
        Default: ginflow.yaml
```

```
-d, --debug
    Default: 0

-h, --help
    Default: false

-o, --option
    Executor options
    Syntax: -okey=value
    Default: {}

* -w, --workflow
    workflow id
```

3.2 Configuration File

By default the command line tries to load a file named *ginflow.yaml* in the current directory. An alternative file may be specified using the *-c* switch of the command line. Here is the full specification of the options you may find in the configuration file.

Notice first the Kafka part. Switching to a Kafka broker requires firstly having a Kafka installation running (including a ZooKeeper server), and secondly modifying the configuration file. The `broker` parameter must be changed for Kafka, and the Kafka and ZooKeeper servers fields need to be filled.

If you have a Mesos instance running on top of your resources, and you want GinFlow workers to be scheduled on your resources through it, you need to fill the Mesos part of the configuration file (the four `mesos.*` parameters).

```
## Distribution

distribution.home: # absolute path to the directory containing GinFlow JAR
distribution.name: # GinFlow binary e.g hocl-workflow-0.0.1-SNAPSHOT.jar

#
# Distributed parameters
#
### -e mesos
mesos.user      : # user to use to launch local invoker on mesos slaves
mesos.master    : # adress:port of the mesos master
mesos.resources.cpu: # cpu to use for running the local invoker e.g: "1"
mesos.resources.mem: # memory ammount to use for running the local invoker e.g: "128"

### -e ssh
ssh.machines    : # comma separated list of remote machines to use
ssh.user        : # user login used to connect to the remote machines
ssh.private_key: # path to the private key to use
ssh.sleep       : # time to wait after launching all the local invokers
                  and before launching the workflow (not used)

#
# Communication parameters
#
broker: # broker to use, possible values are activeMQ, Kafka

## ActiveMQ options
activemq.broker.url: # endpoint of the activeMQ broker
                    # e.g: "failover://(tcp://localhost:61616)"
```

```

## Kafka
kafka.bootstrap.servers: # comma separated list of bootstrap servers
                        # e.g: "localhost:9092"
kafka.zookeeper.connect: # comma separated list of zookeeper nodes
                        # e.g: "localhost:2181"

# hocl
hocl.debug: # debug level of HOCL engine, possible value are "0", ... "9"

# chaos
chaos.threshold: # probability that a service agent fails
                # e.g : "0.5"
chaos.sleep:     # time before failure (ms)
                # e.g : "10000"

```

3.3 JSON workflow description

As illustrated above, the simpler way to describe your workflow is to use the JSON format. A GinFlow workflow is basically a JSON list of services, preceded by a name:

```

{
  "name" : "workflow-name",
  "services": [
    ...
  ]
}

```

Each service's description comprises the following possible fields:

```

{
  "name" : ["..."],
  "srv"  : ["..."],
  "in"   : ["..."],
  "src"  : ["..."],
  "dst"  : ["..."],
  "src_control" : ["..."],
  "dst_control" : ["..."],
  "alt"  : <int>,
  "sup"  : <int>
}

```

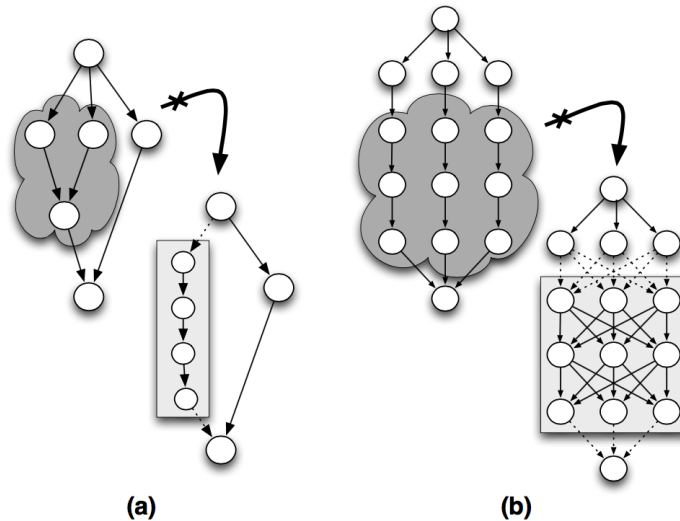
Let us briefly review these (all string) fields:

1. The `name` field is a name you give to the service. It will be used as an identifier for references from the other services. **MANDATORY**
2. The `srv` field gives the command line to be executed when triggering this service. **MANDATORY**
3. The `src` field is a comma separated list of service identifiers denoting the incoming data dependencies of the service. All output from this list of services will be used as input to this service. **OPTIONAL**, default = `[""]`
4. The `dst` field is a comma separated list of service identifiers to which send the output of this service. Note that the data dependencies are redundant: if there is a data dependency from *i* to *j*, *i* appears in the `src` field of service *j*, and *j* appears in the `dst` field of service *i*. **OPTIONAL**, default = `[""]`

5. The `src_control` and `dst_control` fields are similar to `src` and `dst` fields respectively, except that they express *control* dependencies instead of *data* dependencies. A *control* dependency expresses that some service has to wait for the completion of another one to start, but will not take its output as input. OPTIONAL, default = [""]
6. The `alt` and `sup` fields are exclusive and are related to the adaptation mechanism in play in GinFlow.
 - `sup` indicates that the service is under supervision. It means that if the service returns a non zero status code, the adaptation mechanism will occur for the group the service belongs to. The integer value identifies the group of the service.
 - `alt` indicates that the service is an alternative for the group identified by the given integer value.

3.4 Adaptation Mecanisms

GinFlow introduces an easy way to specify and execute alternative scenarios. Alternative sub-workflow get triggered on-the-fly by the engine in a transparent way for the user. Below you will find two examples of workflow whose shapes are changed at run-time.

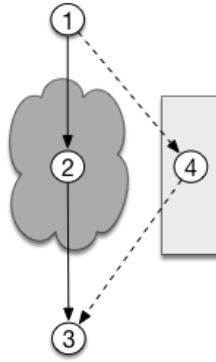


In both cases one group of services is changed in another group of services. The former group is called the supervised group and the latter the alternative group. GinFlow allows the definition of several supervised groups as long there exists corresponding alternative groups.

3.4.1 Use in the JSON workflow file

Let's consider a sequence of three services where the second service is supervised and replaced by an alternative in case of failure. The resulting workflow after the adaptation is also a sequence of three services. For this scenario we thus define :

- one supervised group with id 1 which contains the second service
- one alternative group with id 1 which contains the alternative service



```

{
  "name"      : "wf-1-1",
  "services": [
    {
      "name"      : ["2"],
      "srv"       : ["thiscommandisnotfound"],
      "in"       : ["2"],
      "sup"      : 1,
      "dst_control": ["3"],
      "src_control": ["1"]
    },
    {
      "name"      : ["1"],
      "srv"       : ["echo"],
      "in"       : ["1"],
      "src_control": [],
      "dst_control": ["2"]
    },
    {
      "name"      : ["3"],
      "srv"       : ["echo"],
      "in"       : ["3"],
      "src_control": ["2"],
      "dst_control": []
    },
    {
      "name"      : ["4"],
      "srv"       : ["echo"],
      "in"       : ["4"],
      "alt"      : 1,
      "dst_control": ["3"],
      "src_control": ["1"]
    }
  ]
}

```

The execution of the previous workflow leads to the following output :

```

- Initializing the workflow
- Workflow initialized with id 2061eea6-7b3d-40fc-8f11-4f70ce527a4c
- Starting the ssh executor
- Initial description to 4 services is being transmitted
- -> Service 1 done
- -> Service 1 done
- -> Service 4 done

```

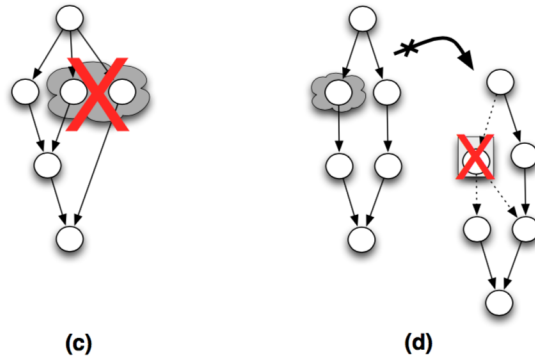
```

- -> Service 3 done
- -> Result received 1/1
- Workflow terminated
- Cleaning remaining processes
- Writing json output to test.json.ginout
- Exiting

```

3.4.2 Restrictions

- Two groups (of supervised services or alternative services) can not overlap.
- There must be only one destination for all final services of both (initial and replacement). Illustrations below.



3.5 Java API

Definition, execution of workflows can be done through the Java API of GinFlow. Please refer to the developer section to learn more about this feature.

3.6 Deploying on Grid'5000

The following steps will guide you through the basic use of GinFlow on the Grid'5000 platform¹.

1. Install one of the latest ActiveMQ releases (<http://activemq.apache.org/download-archives.html>)
2. Make a reservation :

```
screen oarsub -t allow_classic_ssh -l "nodes=N,walltime=XX:YY:ZZ" -I
```

3. Start ActiveMQ
4. Get the list of machines you reserved:

```
s=""; for i in $(uniq $OAR_FILE_NODES); do s=$s,$i; done; echo $s
```

5. In the config file ``ginflow.yaml``

- copy/paste the list of nodes in the ``ssh.machines`` field.
- set the other parameters accordingly to your installation.

¹ <http://www.grid5000.fr>

You are now ready to use the `'ssh executor'` with ActiveMQ.

Note : Depending on your needs Kafka and Zookeeper may replace ActiveMQ. You can refer to their respective documentations to install them.

4.1 Source code

The code base is hosted on Bitbucket : <https://bitbucket.org/ginflow/hocl-workflow>. Releases are tagged with the version number and can be found on the *release* branch.

4.2 Developing with Maven

4.2.1 Set up your project

You can add the GinFlow artifact as a maven dependency to your project. This can be done by adding the following lines to your *pom.xml*

```
<repository>
  <id>ginflow-snapshots-repository</id>
  <name>ginflow-snapshots-repository</name>
  <url>http://hocl-server.irisa.fr/downloads/maven/snapshots</url>
</repository>
<repository>
  <id>ginflow-releases-repository</id>
  <name>ginflow-releases-repository</name>
  <url>http://hocl-server.irisa.fr/downloads/maven/releases</url>
</repository>

<dependency>
  <groupId>fr.inria</groupId>
  <artifactId>hocl-workflow</artifactId>
  <version>X.Y.Z</version>
</dependency>
```

Fill the hocl-workflow version accordingly to your needs.

4.2.2 Using the Java API

This examples are extracted from the test suite. They aim at illustrating how to build and launch a workflow programmatically.

First we define a method to build the *i*-th service in a numbered sequence of services. Note that the first and the last service in a sequence need a special attention regarding their sources and destinations:

```
/**
 *
 * Construct a service in sequence.
 *
 * Ti-1 -> Ti -> Ti+1
 *
 * @param max      max number of service in the sequence
 * @param i        current id in the sequence
 * @param command  the command to run
 * @return The built service.
 */
protected Service Ti(int max, int i, String command) {
    Service service = Service.newService(String.valueOf(i)
        .setCommand(Command.newCommand(command)));
    Destination destination = Destination.newDestination();
    Source source = Source.newSource();
    if (i < max - 1) {
        destination = Destination.newDestination()
            .add(String.valueOf(i + 1));
    }
    if (i > 0) {
        source = Source.newSource()
            .add(String.valueOf(i - 1));
    }

    return service.setSource(source)
        .setDestination(destination);
}
```

Next we define the workflow containing all the services in sequences (in this case, all services are executing the *echo* command):

```
/**
 *
 * Build a sequence of max services, each running the same command.
 *
 * @param max      The number of services
 * @param command  The command
 * @return
 */
protected Workflow sequence(int max, String command) {
    Workflow workflow = Workflow.newWorkflow("sequence");
    for (int i = 0 ; i < max ; i++) {
        workflow.registerService(String.valueOf(i), Ti(max, i, command));
    }
    // set the last service has terminal.
    workflow.getTerminalServices().add(String.valueOf(max - 1));
    return workflow;
}
```

Then we can instantiate a workflow and an executor object to handle the execution of the workflow:

```
int max = 2;
Workflow workflow = sequence(max, "echo !");
GinflowExecutor executor = new CentralisedExecutor(workflow, new Options());
executor.decorate();
executor.execute();
```

To go further, please refer to the API documentation.

BIBLIOGRAPHY

- [HOCL] Jean-Pierre Banâtre, Pascal Fradet, Yann Radenac: Generalised multisets for chemical programming. *Mathematical Structures in Computer Science* 16(4): 557-580 (2006)
- [FGCS] Héctor Fernandez, Cédric Tedeschi, Thierry Priol: Rule-driven service coordination middleware for scientific applications. *Future Generation Comp. Syst.* 35: 1-13 (2014)