GLU Agent

Introduction
The GLU agent is an active process that needs to run on every host where applications need to be deployed. Its main role is to run GLU scrips. It is the central piece of the deployment automation platform and is the only required piece of infrastructure. It exposes a REST api and a command line (which invokes the REST api under the cover).

Fabric & Agent name
An agent belongs to one and only one fabric (which is a group of agents) defined by its name. The agent needs to have a unique name within a fabric which by default is the canonical host name of the computer the agent is running on. You can change the agent name and the fabric when you start the agent (-n and -f options respectively).

GLU Script Engine
The agent is a GLU script engine: it knows how to install and execute GLU scripts.

Installing a GLU script:
1. Groovy API
```
    agent.installScript(mountPoint: '/geo/i001',
                        scriptLocation: 'http://host:port/glu/MyGluScript.groovy',
                        initParameters: [skeleton: 'ivy:/skeleton/jetty/1.0'])
```

2. Command Line
```
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -i http://host:port/glu/MyGluScript.groovy -a
"[skeleton:'ivy:/skeleton/jetty/1.0']"
```

3. REST API
```
PUT /mountPoint/geo/i001
{"args": {"scriptLocation": "http://host:port/glu/MyGluScript.groovy",
          "initParameters": {"skeleton": "ivy:/skeleton/jetty/1.0"} } }
```

- A GLU script gets installed on a given mount point which is the unique key which further commands will reference.
- The script location is a URI which points to the location of the script (this URI must obviously be accessible from the agent, so although you can use a URI of the form `file://`, it will work only if the file can be accessed (ex: local filesystem or nfs mounted file system)).
- `initParameters` is of type metadata and is a map that the agent will make available when executing the GLU script

Note: check the javadoc for more details on the API

Executing a GLU script action
1. Groovy API
```
    // non blocking call
    agent.executeAction(mountPoint: '/geo/i001', action: 'install')

    // blocking until timeout
    agent.waitForState(mountPoint: '/geo/i001', state: 'installed', timeout: '10s')
```

2. Command Line
```
# non blocking
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -e install

# blocking until timeout
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -e install -w installed -t 10s

# which can be run as 2 commands
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -e install
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -w installed -t 10s

# Shortcut for installscript + install + wait for state
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -I http://host:port/glu/MyGluScript.groovy -a
"[skeleton:'ivy:/skeleton/jetty/1.0']" -t 10s
```

3. REST API
```
// executeAction
POST /mountPoint/geo/i001
{"args": {"executeAction": {"action": "install"} } }

// wait for state
GET /mountPoint/geo/i001?state=installed&timeout=10s
```

You can execute any action on the script that you are allowed to execute (as defined by the state machine). Note that you use the same mount point used when installing the script. If you are not allowed then you will get an error: for example, using the default state machine you cannot run the start action until you run install and configure. The command line has a shortcut to do all this in one command:

4. Command Line shortcut

```
# Shortcut for installscript + install + wait for state + configure + wait for state  +
# start + wait for state
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -S http://host:port/glu/MyGluScript.groovy -a
"[skeleton:'ivy:/skeleton/jetty/1.0']"
```

You can also provide parameters to the action when you invoke it:

5. Groovy API (with action args)

```
    // non blocking call
    agent.executeAction(mountPoint: '/geo/i001', action: 'install' actionArgs: [p1: 'v1'])
```

6. Command Line (with action args)

```
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -e install -a "[p1:'v1']"
```

7. REST API (with action args)

```
// executeAction
POST /mountPoint/geo/i001
{"args": {"executeAction": {"action": "install", "actionArgs": {"p1": "v1"} } } }
```

They are then available through the normal groovy closure invocation functionality:

```
def install = { args ->
   if(args.p1 == 'v1')
   {
     // do something
   }
}
```

Uninstalling the script

1. Groovy API

```
    agent.uninstallScript(mountPoint: '/geo/i001')
```

2. Command Line

```
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -u"
```

3. REST API

```
DELETE /mountPoint/geo/i001
```

Simply uninstall the script. Note that you cannot uninstall the script unless the state machine allows you do to so. If you are in state 'running' you first need to run stop, unconfigure and uninstall. There is a way to force uninstall irrelevant of the state of the state machine:

4. Groovy API (force uninstall)

```
    agent.uninstallScript(mountPoint: '/geo/i001', force: true)
```

5. Command Line (force uninstall)

```
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -u -F"
```

6. REST API (force uninstall)

```
DELETE /mountPoint/geo/i001?force=true
```

The command line also has a shortcut to uninstall by properly running through all the phases of the state machine:

7. Command Line (shortcut)

```
agent-cli.sh -s https://localhost:12906/ -m /geo/i001 -U"
```

Capabilities

One of the main design goals in building the agent was the ability to write simple GLU scripts. This is achieved with the fact that the agent enhances the GLU scripts with capabilities that make it easier to write them. Most of the capabilities are made available to the GLU scripts by 'injecting' properties that the GLU scripts can simply reference (under the hood it uses groovy MOP capabilities).

log

The `log` property allows you to log any information in the agent log file. It is an instance of `org.slf4j.Logger`

```
def configure = {
  log.info "this is a message logged with info level"

  log.debug "this message will be logged only if the agent is started with debug messages on"
}
```

params

Every GLU script action has access to the initParameters provided at installation time through the `params` property:

```
def configure = {
```

```
  log.info "initParameters = ${params}"
}
```

mountPoint
The `mountPoint` on which the script was installed. In general, this property is used to install the application in a unique location (since the mountPoint is unique).

```
def install = {
  log.info "mountPoint = ${mountPoint}"
  def skeleton = shell.fetch(params.skeleton) // download a tarball
  shell.untar(skeleton, mountPoint) // will be unzipped/untarred in a unique location
}
```

stateManager
An instance of `org.linkedin.glu.agent.api.StateManager` which allows to access the state

```
def install = {
  log.info "current state is ${stateManager.state}"
}
```

state
Shortcut to `stateManager.state`

```
def install = {
  log.info "current state is ${state}"
}
```

shell
An instance of `org.linkedin.glu.agent.api.Shell` which gives access to a lot of shell like capabilities
* file system (see `org.linkedin.groovy.util.io.fs.FileSystem`) like ls, cp, mv, rm, tail…
* process (fork, exec…)
* fetch/untar to download and untar/unzip binaries (based on any URI). Note that the agent handles `zookeeper:/a/b/c` style URIs and can be configured to handle `ivy:/a/b/1.0` style URIs (TODO: add maven support).

```
def install = {
  def skeleton = shell.fetch(params.skeleton) // download a tarball
  shell.untar(skeleton, mountPoint) // unzip/untar (detect zip automatically)
  shell.rm(skeleton)
}
```

Accessing shell environment properties:
`shell.env` is a map which allows you to access all the configuration properties used when the agent booted including the ones stored in zookeeper. This allows for example to configure fabric dependent behavior. If you store the property:

```
my.company.binary.repo.url=http://mybinaryrepo:9000/root
```

in the configuration file (agent config) loaded in ZooKeeper for a given fabric then your scripts can use relative values:

```
shell.fetch("${shell.env['my.company.binary.repo.url]/${params.applicationRelativePath}"}
```

timers
An instance of `org.linkedin.glu.agent.api.Timers` which allows you to set/remove timers (for monitoring for example).

```
def timer1 = {
  log.info "hello world"
}

def install = {
  timers.scedule(timer: timer1, repeatFrequency: '1m')
}

def uninstall = {
  timers.cancel(timer: timer1)
}
```

OS level functionalities
The agent also offers some OS level functionalities
ps / kill

```
// groovy API
agent.ps()
agent.kill(12345, 9)

// command line
agent-cli.sh -s https://localhost:12906/ -p
```

```
agent-cli.sh -s https://localhost:12906/ -K 1234/9

// REST API
// ps
GET /process

// kill -9 1234
PUT /process/1234
{"args": {"signal": 9} }
```

tail / list directory content

```
// groovy API
agent.getFileContent(location: '/tmp') // directory content
agent.getFileContent(location: '/tmp/foo', maxLine: 500) // file content (tail -500)

// command line
agent-cli.sh -s https://localhost:12906/ -C /tmp
agent-cli.sh -s https://localhost:12906/ -C /tmp/foo -M 500

// REST API
GET /file/tmp
GET /file/tmp/foo?maxLine=500
```

ZooKeeper

By default the agent uses ZooKeeper to 'publish' its state in a central location as well as to read its configuration. Note that it is optional and ZooKeeper can be disabled in which case the whole configuration needs to be provided.

Auto Upgrade

The agent has the capability of being able to upgrade itself

Using the command line

```
agent-cli.sh -s https://localhost:12906/ -c org.linkedin.glu.agent.impl.script.AutoUpgradeScript -m /
upgrade -a "[newVersion:'2.0.0',agentTar:'file:/tmp/agent-server-upgrade-2.0.0.tgz']"
agent-cli.sh -s https://localhost:12906/ -m /upgrade -e install
agent-cli.sh -s https://localhost:12906/ -m /upgrade -e prepare
agent-cli.sh -s https://localhost:12906/ -m /upgrade -e commit
agent-cli.sh -s https://localhost:12906/ -m /upgrade -e uninstall
agent-cli.sh -s https://localhost:12906/ -m /upgrade -u
```

Using the console

Click on the Admin tab, then Upgrade agents.

Independent lifecycle

The agent can be started / stopped independently of the applications that it is managing: the agent stores its state locally (and in ZooKeeper if enabled) and knows how to restore itself properly (including restarting any timers that were scheduled by glu scripts!)

Requirements

The GLU agent requires java 1.6 to be installed on the host it is running on. As this stage only unix like hosts are supported (tested on Solaris and Mac OS X).

Agent boot sequence and configuration

TODO: add details about the boot sequence
ZooKeeper ephemeral node

Installation

The agent is bundled as a self contained full package (`org.linkedin.glu.agent-server-x.y.z.tar.gz`). Simply untar/unzip in a given location. The agent upgrade also comes as a package (`org.linkedin.glu.agent-server-upgrade-x.y.z.tar.gz`) and is normally installed using the auto upgrade capability described above.

Security

The agent offers a REST API over https, setup with client authentication. In this model, what is really important is for the agent to allow only the right set of clients to be able to call the API.

Key setup

The agent comes with a default set of keys. It is strongly suggested to generate your own set of keys. See key generation document to help on creating a new set of keys (currently under `glu/agent/org.linkedin.glu.agent-server/src/zk-config/keys/key_generation.txt`)
Coming soon: a wrapper to do this

Multiple agents on one host

You can run multiple agents on the same machine as long as you assign them different ports and different names although it is not recommended for production. This is usually used in development.