fjage Documentation

Release 1.12.0

Mandar Chitre

Sep 10, 2023

Contents

CHAPTER 1

Table of Contents

1.1 Introduction

fjåge provides a **lightweight** and **easy-to-learn** platform for agent-oriented software development in Java and Groovy.

1.1.1 Why fjåge?

Several frameworks exist for agent-oriented software development. For Java programmers, JADE provides a FIPA-compliant framework for multi-agent systems. The API for fjåge is largely based on the API available in JADE, and so any developer familiar with JADE should have very little difficulty learning to develop using fjåge. However, there are some significant differences between the philosophy between the two projects. The key advantages of fjåge are:

- fjåge is designed to be very **lightweight and fast**, and is suitable for Java-capable embedded systems.
- The API for fjåge is kept very simple with a view to making it **easy to learn**, and having very little scaffolding code. This enables a **quick agent development cycle**.
- fjåge can be run in realtime mode, or in a **discrete event simulation mode**. This makes it ideally suited for development of simulators, and allows rapid testing of production code in simulated environments.
- fjåge has excellent **Groovy support** that makes agent development easy to learn and enjoyable, and the resulting code very readable.
- fjåge provides an **interactive shell and scripting support**, making development, debugging and remote management easy.
- fjåge has APIs for access from Java, Groovy, Python, C and Javascript applications, and a well-defined JSON protocol that can be used to develop connectors from any other language.

On the flip side, although fjåge follows many of the ideas from FIPA, it is not fully FIPA-compliant and cannot directly interact with other FIPA-compliant multiagent systems.

1.1.2 Java and Groovy support

Although most of the functionality of the framework can be used in pure-Java projects, the adoption of Groovy in the project simplifies development immensely. In this guide, most of the code examples are in Groovy. Writing equivalent Java code is mostly trivial, though there are cases where the mapping may not be obvious. In such cases, we provide Java examples alongside the Groovy ones.

1.1.3 Key concepts

A multi-agent system developed in fjåge consists of several software *agents* that communicate with each other using *messages*. Each instance of an agent is identified by a unique *AgentID* that can be used to send messages to that agent. Agents have *behaviors* that are invoked on events, and allow the agents to take actions. The behaviors may be invoked at a specified time, at a specified rate, on reception of a message, occurance of some event, or even continously. The agents reside in one or more *containers* that provide agent management, directory and messaging services. Various containers may run on the same node, or on different nodes in a network. Each container runs on a *platform* that provides container management, timing and scheduling services.

There are two kinds of platforms available:

- **RealTimePlatform** – This platform provides timing and scheduling services such that the timing requirements of an application are met on a best-effort basis. For example, if an agent asks to have a behavior be activated every 500 ms, the platform tries its best (within the limits of what the operating system and hardware allows) to invoke the behavior every 500 ms. The time returned by the time-related functions is based on the operating system's real-time clock.
- **DiscreteEventSimulator** – This platform allows agents to be tested in a discrete event simulation mode. Essentially, all the time-related functions and scheduling use a *virtual time*. The passage of virtual time is simulated such that computation and processing does not take any virtual time, while scheduling requests are met accurately in virtual time. The virtual time advances in discrete steps such that the time when no agent is active is effectively skipped. This potentially allows for simulation of hours of virtual time within seconds.

To switch between the two platforms, the agent code does not require any changes as long as a couple of simple rules are followed while developing the agents:

- Agents must not use any system timing functions directly. Rather than use *System.currentTimeMillis()* or *System.nanoTime()*, the agents should use *Agent.currentTimeMillis()* and *Agent.nanoTime()*.
- Agents must not use any system scheduling functions directly. Rather than use *Thread.sleep()*, the agent should use *Agent.delay()*.

Following these rules guarantees that the agents transparently switch between the real-time or virtual time that the platform provides.

Agents may choose to provide one or more *services*. Rather than having to know the AgentID of an agent in advance, agents requiring the services of another agent may look for agents providing specific services via the *directory service*. Agents may choose to subscribe to and send messages to *topics*. All agents subscribing to a topic receive each message sent to that topic.

Agents may advertise *parameters*. These are generic key-value pairs that can be read and optionally written to using a *ParameterReq* message. Agents typically use parameters for configuration and status reporting.

That's pretty much it for the concepts that you need to understand to get started. If all of this seems a bit abstract at the moment, don't worry about it – things will become clear shortly as we go through some examples.

1.1.4 License

fjåge is released under the open source simplified (3-clause) BSD license.

1.1.5 Availability

fjåge is available as a binary release via Maven central:

```
<dependency>
  <groupId> com.github.org-arl </groupId>
  <artifactId> fjage </artifactId>
  <version> 1.12.0 </version>
</dependency>
```

Its source code is available via GitHub:

```
git clone git@github.com:org-arl/fjage.git
```

1.2 Getting Started

1.2.1 Quick start

We also assume a UNIX style platform (Linux, Mac OS X or Cygwin on Windows) with Java Development Kit (JDK) 1.8 or higher installed. If you don't have one, you'll need to install one. We also assume that you have network connectivity to download the relevant files automatically.

Create a new folder for your first fjåge project. Lets call it *MyFjageProject*. Open a terminal window in this folder and download the fjage_quickstart.sh script in that folder:

```
curl -O https://raw.githubusercontent.com/org-arl/fjage/master/src/sphinx/fjage_
↪quickstart.sh
```

To get your project ready, just run the script:

```
sh fjage_quickstart.sh
```

The script downloads all necessary libraries (jar files) and some template startup scripts to get you going. You may now delete off the *fjage_quickstart.sh* file, if you like:

```
rm fjage_quickstart.sh
```

Your directory structure should now look something like this:

```
fjage.sh
rconsole.sh
logs/
build/libs/
  fjage-1.12.0.jar
  groovy-all-2.4.4.jar
  commons-lang3-3.1.jar
  jline-3.90.jar
  gson-2.8.2.jar
etc/
  initrc.groovy
  initrc-rconsole.groovy
samples/
  01_hello.groovy
  02_ticker.groovy
  03_weatherRequest.groovy
  03_weatherStation.groovy
  04_weatherRequest.groovy
  04_weatherStation.groovy
  WeatherForecastReqMsg.groovy
```

Note: The *build/libs* folder contains all the necessary libraries. The *etc* folder contains startup files. The *samples* folder contains the example programs used in this documentation. *initrc.groovy* is the initialization script where you create your agents and configure them. *fjage.sh* is your startup script that simply sets up the classpath and boots up fjåge with the *initrc.groovy* script. The organization of the directory structure and names of the files are all customizable by editing *fjage.sh* and *initrc.groovy*.

To check that your fjåge installation is correctly working, type *./fjage.sh* (or fjage.bat in command prompt). That should simply give you an interactive fjåge Groovy shell with a > prompt. Type *ps* to see a list of running agents. There should be only one *shell* agent created by the default *initrc.groovy* script. Type *shutdown* or press control-D to terminate fjåge.

```
bash$ ./fjage.sh
> ps
shell
> shutdown

bash$
```

1.2.2 Hello world agent

As any good tutorial does, we start with the proverbial *hello world* agent. The agent isn't going to do much, other than print the words "Hello world!!!" in the logs.

Create a file called *hello.groovy* in your project folder and put the following contents in it:

```groovy
import org.arl.fjage.*

class HelloWorldAgent extends Agent {
  void init() {
    add new OneShotBehavior({
      println 'Hello world!!!'
    })
  }
}

container.add 'hello', new HelloWorldAgent()
```

This Groovy script creates an agent with *AgentID hello* of class *HelloWorldAgent*. The *init()* method of the agent is called once the agent is loaded. In this method, a one-shot behavior is added to the agent. One-shot behaviors are fired only once, as soon as possible; in our case, this is as soon as the agent is running. The one-shot behavior prints "Hello world!!!". The output of the agent is not directly displayed on the console, but instead sent to the log file, as we will see shortly.

To run the agent, start fjåge and run the script by typing *run 'hello'* or simply <*hello* (the *less than* sign '<' is a shortcut for *run* '. . . '). This will return you to the interactive shell prompt. To check that your agent is indeed running, type *ps*. You may then shutdown fjåge as before and check the log file for your output:

```
bash$ ./fjage.sh
> <hello
```

(continues on next page)

(continued from previous page)

```
> ps
hello
shell
> shutdown

bash$ cat logs/log-0.txt | grep HelloWorldAgent@
1377443280802|INFO|HelloWorldAgent@18:println|Hello world!!!
bash$
```

Tip: The code for the HelloWorldAgent is located in the samples directory. You can simply run it by typing:

<samples/01_hello.

The default fjåge log file format is pipe-separated, where the first column is the timestamp in milliseconds, the second column is the log level, the third column is the agent class name + threadID + method name, and the last column is the log message. You may change the format if you like by loading a custom logging configuration by specifying a *java.util.logging.config.file* system property while starting the JVM (see Java logging).

Congratulations!!! You have just developed your first Groovy fjåge agent!

Note: Stack traces for any exceptions caused by any agent will be dumped to the log file. This can be invaluable during debugging.

1.2.3 Packaging agents

The method shown above defined the agent class in a Groovy script that was executed from the interactive shell. If the Groovy script is modified, the agent can be reloaded by killing it and running the script again:

```
bash$ ./fjage.sh
> <hello
> ps
hello
shell
> container.kill agent('hello');
> ps
shell
> <hello
> ps
hello
shell
>
```

This is useful for testing. However, in a production system, you usually want to define agents in their own files, compile them and package them into a jar on the classpath. To do this, you would create a source file *HelloWorldAgent.groovy* with the class definition:

```groovy
import org.arl.fjage.*

class HelloWorldAgent extends Agent {
  void init() {
    add new OneShotBehavior({
      println 'Hello world!!!'
```

(continues on next page)

(continued from previous page)

```
    })
  }
}
```

or *HelloWorldAgent.java* with the class definition:

```
import org.arl.fjage.*;

public class HelloWorldAgent extends Agent {
  public void init() {
    add(new OneShotBehavior() {
      public void action() {
        println("Hello world!!!");
      }
    });
  }
}
```

You would then compile it into a *HelloWorldAgent.class* file using the *groovyc* compiler (or *javac* compiler) and perhaps package it into a jar file. You would then put this jar file or the class file on the classpath.

The *fjage.sh* startup script includes all jar files from the *build/libs* folder into the classpath. So you could simply copy your jar file into the *build/libs* folder and then run *fjage.sh*. You can then load the agent on the interactive shell:

```
bash$ ./fjage.sh
> ps
shell
> container.add 'hello', new HelloWorldAgent();
> ps
hello
shell
>
```

If you wanted the agent to be automatically loaded, you can put the *container.add 'hello', new HelloWorldAgent()* statement in the *initrc.groovy* startup script.

1.2.4 Typical bootup for Groovy applications

In order to fully understand how fjåge works, it is useful to look at a slightly simplified version of the bootup sequence of our hello world fjåge application. When we run *fjage.sh*, the shell script creates a CLASSPATH to include all jar files in the *build/libs* folder and then starts the JVM:

```
java -cp "$CLASSPATH" org.arl.fjage.shell.GroovyBoot etc/initrc.groovy
```

This command invokes the *main()* static method on the *org.arl.fjage.shell.GroovyBoot* class. The initialization script *etc/initrc.groovy* is passed as a command line argument to the *main()*.

Let us next take a look at a simplified code extract from the *org.arl.fjage.shell.GroovyBoot.main()* method:

```
public static void main(String[] args) throws Exception {
  GroovyExtensions.enable();
  engine = new GroovyScriptEngine();
  for (String a: args) {
    engine.exec(new File(a), null);
    engine.waitUntilCompletion();
```

(continues on next page)

(continued from previous page)

```
  }
  engine.shutdown();
}
```

Note: *GroovyBoot* also supports resource URLs of the form *res://path/to/package/script.groovy* and *cls://package.script* to execute initialization Groovy scripts loaded from Java resources (potentially inside jar files).

This code enables Groovy extensions in fjåge to add syntactic sugar for ease of writing Groovy agents, and then sequentially executes every initialization Groovy script given on the command line. In our case, this causes the *etc/initrc.groovy* to be executed:

```
import org.arl.fjage.*
import org.arl.fjage.shell.*

platform = new RealTimePlatform()
container = new Container(platform)
shell = new ShellAgent(new ConsoleShell(), new GroovyScriptEngine())
container.add 'shell', shell
// add other agents to the container here
platform.start()
```

The script imports the fjage packages. A real-time platform and a container is created, and a *shell* agent is configured and added to the container. The *shell* agent is set to provide the interactive shell on the console, and use Groovy for scripting. Finally, the platform is started. Now we have a fjåge container running with a single *shell* agent that provides an interactive shell on the console.

Any other agents that we may wish to start can be included in the *etc/initrc.groovy* script, just before starting the platform.

1.2.5 Bootup for Java applications

If you wanted a pure-Java project, you would forego the scripting ability (since that requires Groovy) and simply setup the platform and container directly from the *main()* program. For example:

```
import org.arl.fjage.*;

public class MyProject {
  public static void main(String[] args) throws Exception {
    Platform platform = new RealTimePlatform();
    Container container = new Container(platform);
    // add your agents to the container here
    // e.g. container.add("hello", new HelloWorldAgent());
    platform.start();
  }
}
```

As simple as that!

1.3 Agents and Behaviors

1.3.1 Agent lifecycle

When an Agent is added to a container, it starts in the INIT state. When the platform is running, agents in the INIT state are initialized by calling their *init()* method. An typical agent overrides this method and adds new behaviors to itself.

After initialization, the agent moves to a RUNNING state. In this state, active behaviors of the agent are scheduled to run. A typical agent is associated with one agent thread, and various behaviors are cooperatively scheduled on this thread. Due to the cooperative nature of the behaviors, a poorly written behavior may block execution of all other behaviors of that agent. Developers should avoid long-running or blocking code in behaviors; if such code is needed, it is best to create a separate thread to run that code.

A behavior that is not ready to run may be in a blocked state (e.g. a behavior that is to be executed at a specified later time). An agent may make a behavior as blocked, by explicitly calling the *block()* method on that behavior. If there are no active behaviors for an agent, the agent goes into an IDLE state. Behaviors may be activated due to timer events, message delivery events or by explicit *restart()* method calls. When one or more behaviors become active, the agent goes back into a RUNNING state.

When an agent is killed or the platform is shutdown, the agent is placed in a FINISHING state. Agents in this state are given a chance to cleanup via a call to their *shutdown()* method. An agent may override this method if a cleanup is required. After the cleanup, the agent is terminated and placed in a FINISHED state, and removed from the container.

An example skeleton agent is shown below:

```
class MyAgent extends Agent {

  void init() {
    // agent is in INIT state
    log.info 'Agent init'
    add new OneShotBehavior({
      // behavior will be executed after all agents are initialized
      // agent is in RUNNING state
      log.info 'Agent ready'
    })
  }

  void shutdown() {
    log.info 'Agent shutdown'
  }

}
```

Tip: *println()* from an agent is mapped to *log.info()*, and can be used interchangably. However, the *log* object provides more flexibility (e.g. *log.warning()*, *log.fine()*, etc).

The *traditional* behavior creation style may be used by Java and Groovy agents:

```
add(new OneShotBehavior() {
  public void action() {
    // do something
  }
});
```

The method to override depends on the behavior (e.g. *action()* for most behaviors, but *onTick()* for *TickerBehavior*, and *onWake()* for *WakerBehavior*).

Groovy agents support a simpler alternative syntax if the *GroovyExtensions* are enabled:

```
add new OneShotBehavior({
  // do something
})
```

Both variants are identical in function. With this syntax, the appropriate method is automatically overridden to call the defined closure. For the examples in the rest of this chapter, we will adopt the simpler Groovy syntax.

1.3.2 One-shot behavior

A OneShotBehavior is run only once at the earliest opportunity. After execution, the behavior is automatically removed. We have seen an example of the one-shot behavior above.

1.3.3 Cyclic behavior

A CyclicBehavior is run repeatedly as long as it is active. The behavior may be blocked and restarted as necessary.

```
class MyAgent extends Agent {
  int n = 0
  void init() {
    // a cyclic behavior that runs 5 times and then marks itself as blocked
    add new CyclicBehavior({
      agent.n++
      println "n = ${agent.n}"
      if (agent.n >= 5) block()
    })
  }
}
```

Tip: Although it may be possible in some cases to access agent methods and fields directly from a behavior method or closure, it is safer to always use an *agent.* qualifier to access them. Without the qualifier, the closure's delegation strategy causes the behavior methods and fields to be checked first; this can lead to bugs that are difficult to track.

Note: Since behaviors are cooperatively scheduled, they should not block. Hence *Behavior.block()* is not a blocking call; it simply marks the behavior as blocked and removes it from the list of active behaviors to be scheduled, and continues.

1.3.4 Waker behavior

A WakerBehavior is run after a specified delay in milliseconds.

```
add new WakerBehavior(1000, {
  // invoked 1 second later
  println '1000 ms have elapsed!'
})
```

1.3.5 Ticker behavior

A TickerBehavior is run repeated with a specified delay between invocations. The ticker behavior may be terminated by calling *stop()* at any time.

```
add new TickerBehavior(5000, {
  // called at intervals of 5 seconds
  println 'tick!'
})
```

1.3.6 Backoff behavior

A BackoffBehavior is similar to a waker behavior, but allows the wakeup time to be extended dynamically. This is typically useful to implement backoff or retry timeouts.

```
add new BackoffBehavior(5000, {      // first attempt after 5 seconds
  // make some request, and if it fails, try again after 3 seconds
  def rsp = request(req)
  if (rsp == null || rsp.performative == Performative.FAILURE) backoff(3000)
})
```

1.3.7 Poisson behavior

A PoissonBehavior is similar to a ticker behavior, but the interval between invocations is an exponentially distributed random variable. This simulates a Poisson arrival process.

```
add new PoissonBehavior(5000, {
  // called at an average rate of once every 5 seconds
  println 'arrival!'
})
```

1.3.8 Message behavior

A MessageBehavior is invoked when a message is received by the agent. A message behavior may specify what kind of message it is interested in. If multiple message behaviors admit a received message, any one of the behaviors may be invoked for that message.

A message behavior that accepts any message can be added as follows:

```
add new MessageBehavior(Message, { msg ->
  println "Incoming message from ${msg.sender}"
})
```

If we were only interested in messages of class *MyMessage*, we could set up a behavior accordingly:

```
add new MessageBehavior(MyMessage, { msg ->
  println "Incoming message of class ${msg.class} from ${msg.sender}"
})
```

Let us next consider a more complex case where we are interested in message of a specific class and from a specific sender:

```
def filter = { it instanceof MyMessage && it.sender.name == 'myFriend' } as␣
↪MessageFilter
add new MessageBehavior(filter, { msg ->
  println "Incoming message of class ${msg.class} from ${msg.sender}"
})
```

1.3.9 Finite state machine behavior

Finite state machines can easily be implemented using the FSMBehavior class. These machines are composed out of multiple states, each of which is like a *CyclicBehavior*. State transitions are managed using the *nextState* property.

For example, we can create a grandfather clock using a *FSMBehavior*:

```
def b = add new FSMBehavior()
b.add new FSMBehavior.State('tick') {
  void action() {
    println 'tick!'
    nextState = 'tock'
    fsm.block 1000
  }
}
b.add new FSMBehavior.State('tock') {
  void action() {
    println 'tock!'
    nextState = 'tick'
    fsm.block 1000
  }
}
```

1.3.10 Test behavior

The TestBehavior is a special behavior that helps with development of unit tests. Any *AssertionError* thrown in the behavior is stored and thrown when the test ends. A typical usage for a test case is shown below:

```
import org.arl.fjage.*

def platform = new RealTimePlatform()
def container = new Container(platform)
def agent = new Agent()
container.add agent
platform.start()

TestBehavior test = new TestBehavior({
  assert 1+1 == 2 : 'Simple math failed'
  def aid = agent.getAgentID()
  assert aid != null : 'AgentID undefined'
  assert agent.send(new Message(aid)) : 'Message could not be sent'
})
test.runOn(agent)

platform.shutdown()
```

1.3.11 Custom behaviors

Although the above behaviors meet most needs, there are times when you need a behavior that isn't already available. In such cases, you can simply extend the Behavior class to implement your own behavior. This typically involves overriding the *onStart()*, *action()*, *done()* and *onEnd()* methods.

An example two-shot behavior is shown below:

```
class TwoShotBehavior extends Behavior {
  int fired
  void onStart() {
    fired = 0
  }
  void action() {
    fired++
    log.info 'Bang!'
  }
  boolean done() {
    fired >= 2
  }
  void onEnd() {
    log.info 'You are dead!'
  }
}
```

1.4 Messaging

1.4.1 Sending and receiving messages

Agents interact with each other using messages. A Message is usally tagged with a Performative that defines the purpose of the message, and is uniquely identified by a message identifier. A message is also usually associated with a sender and a recipient *AgentID*. If a message is sent in reply to another message, the original message's message identifier is included as a *inReplyTo* property of the reply message. This allows the sender to associate the reply with the original request/query message.

Although the Message class provides all the basic attributes of a message, it does not provide any fields to hold the message content. Typical messages will extend the Message class and add relevant content fields.

Request message:

```
class WeatherForecastReq extends org.arl.fjage.Message {
  WeatherForecastReq() {
    super(Performative.REQUEST)
  }
  String city, country
}
```

Response message:

```
class WeatherForecast extends org.arl.fjage.Message {
  WeatherForecast() {
    super(Performative.INFORM)
  }
  WeatherForecast(Message req) {
    super(req, Performative.INFORM)    // create a response with inReplyTo = req
```

(continues on next page)

(continued from previous page)

```
    city = req.city
    country = req.country
  }
  String city, country
  float minTemp, maxTemp, probRain
}
```

A client agent may send a weather forecast request to another agent named "WeatherStation":

```
send new WeatherForecastReq(city: 'London', country: 'UK', recipient: agent(
↪'WeatherStation'))
```

The "WeatherStation" agent would receive the request and send back a reply. Although messages may be received using an agent's *receive()* method, the preferred way to process messages is using the *Message behavior*:

```
class MyWeatherStation extends org.arl.fjage.Agent {
  void init() {
    add new MessageBehavior(WeatherForecastReq, { req ->
      log.info "Weather forecast request for ${req.city}, ${req.country}"
      def rsp = new WeatherForecast(req)
      rsp.minTemp = 10
      rsp.maxTemp = 25
      rsp.probRain = 0.25
      send rsp
    })
  }
}
```

The client agent would then receive the message, either through a message behavior or by explicitly calling *receive()*. An easier alternative is to send a request and wait for the associated response via the *request()* method:

```
def req = new WeatherForecastReq(city: 'London', country: 'UK', recipient: agent(
↪'WeatherStation'))
def rsp = request req, 1000          // 1000 ms timeout for reply
println "The lowest temperature today is ${rsp?rsp.minTemp:'unknown'}"
```

1.4.2 Generic messages

Although it usually makes sense to create message classes for specific interactions, there are times when it can be useful to send a generic message with key-value pairs. This functionality is provided by the GenericMessage class, which provides a *java.util.Map* interface. In Groovy, this provides a nice syntax that allows the keys to work like dynamic attributes of the message. A weather forecast service implemented using generic messages is shown below.

Server code:

```
import org.arl.fjage.*

class MyWeatherStation extends Agent {
  void init() {
    add new MessageBehavior({ msg ->
      if (msg.performative == Performative.REQUEST && msg.type == 'WeatherForecast') {
        log.info "Weather forecast request for ${msg.city}, ${msg.country}"
        def rsp = new GenericMessage(msg, Performative.INFORM)
        rsp.minTemp = 10
        rsp.maxTemp = 25
```

(continues on next page)

(continued from previous page)

```
        rsp.probRain = 0.25
        send rsp
      }
    })
  }
}
```

Client code snippet:

```
def req = new GenericMessage(agent('WeatherStation'), Performative.REQUEST)
req.type = 'WeatherForecast'
req.city = 'London'
req.country = 'UK'
def rsp = request req, 1000         // 1000 ms timeout for reply
println "The lowest temperature today is ${rsp?rsp.minTemp:'unknown'}"
```

1.4.3 Alternate syntax

Let us assume we have an *AgentID* for the "WeatherStation":

```
def weatherStation = agent('WeatherStation')
```

It is sometimes nicer to be able to use a syntax like this:

```
weatherStation.send new WeatherForecastReq(city: 'London', country: 'UK')
```

or:

```
def rsp = weatherStation.request new WeatherForecastReq(city: 'London', country: 'UK')
```

or perhaps even:

```
def rsp = weatherStation << new WeatherForecastReq(city: 'London', country: 'UK')
```

This alternate syntax sometimes yields more readable code, and is supported by fjåge. It is important, however, to remember that the message is sent in the context of the client agent that provided us with the *AgentID*. Any *AgentID* returned by an agent (by methods such as *agent()*, *agentForService()*, etc) is associated with or *owned by* that agent. When this *AgentID* is used with the above syntax, the message is actually sent using the associated agent.

Note: If you create an *AgentID* explicitly as *new AgentID('WeatherStation')*, it does not have an owner, and therefore cannot be used with this alternate syntax. It can, however, be used with the original syntax as a recipient for a message.

1.4.4 Publishing and subscribing

So far we have sent messages to recipients whose *AgentID* we know. There are times when we may want to publish a message without explicitly knowing who the recipients are. All agents *subscribing* to the *topic* that we publish on would then receive the published message.

This is supported by fjåge using the messaging constructs we have already encountered. Messages can be sent to topics in the same way that messages are sent to other agents. A topic is simply a special *AgentID*:

```
def weatherChannel = topic('WeatherChannel')
```

Instead of using a *String* for the topic name, it is also possible (and usually recommended) to use Enums:

```
enum Topics {
  WEATHER_CHANNEL,
  TSUNAMI_WARNING_CHANNEL
}
```

and

```
def weatherChannel = topic(Topics.WEATHER_CHANNEL)
```

Agents can subscribe to the topic of interest, typically in their *init()* method:

```
subscribe weatherChannel
```

Messages can be sent to all agents subscribing to the topic:

```
def forecast = new WeatherForecast(city: 'London', country: 'UK', minTemp: 10,␣
↪maxTemp: 25, probRain: 0.25)
weatherChannel.send forecast
```

Agents that no longer wish to receive messages on a topic may also unsubscribe from the topic:

```
unsubscribe weatherChannel
```

1.4.5 Cloning messages

By default, a message delivered to another agent in the same container is the original object, and not a copy. This has some subtle but important implications. If an agent modifies a message after sending it, this can lead to unexpected behaviors.

Let's take an example:

```
def msg = new GenericMessage()
msg.text = 'Hello!'
agent('Susan').send msg
msg.text = 'Holla!'
agent('Lola').send msg
```

If the message is delivered to Susan before the agent modifies the message, Susan gets a "Hello!" message and then Lola gets a "Holla!" message. If the message is modified after delivery to Susan, but before she has had a chance to read it, both Susan and Lola get a "Holla!" message. If the message is modified and sent to Lola before it is delivered to Susan, the recipient of the message changes, and two copies of "Holla!" get delivered to Lola and nothing gets delivered to Susan. As you can see, the behavior is indeterminate and a debugging nightmare!

Fortunately, there are several simple ways around this:

1. Do not modify a message once it is sent. The code would then look like this:

```
def msg = new GenericMessage()
msg.text = 'Hello!'
agent('Susan').send msg
msg = new GenericMessage()         // create a new message, don't modify the old␣
↪one
```

(continues on next page)

(continued from previous page)

```
msg.text = 'Holla!'
agent('Lola').send msg
```

2. Send a copy of the message, rather than the original. You can then freely modify the original:

```
def msg = new GenericMessage()
msg.text = 'Hello!'
agent('Susan').send clone(msg)     // send a copy of the message
msg.text = 'Holla!'
agent('Lola').send msg
```

3. Ask the container to always send copies of messages rather than the original, and then you can use the original code without a problem:

```
container.autoClone = true
```

The cloning of the message is accomplished using the *org.apache.commons.lang3.SerializationUtils* class. This performs a deep clone (clones all objects contained in the message) by serializing the entire message, and then deserializing it. This is very portable (as long as your message is *Serializable*), but somewhat slow. A faster deep cloning implementation is available from *com.rits.cloning.Cloner*, but it is less portable (it seems to have trouble dealing with some Groovy messages). If you wish to try this implementation for your application, ensure that you have the following jars in your classpath:

- cloning-1.9.0.jar
- objenesis-1.2.jar

Then switch to using the fast cloner:

```
container.cloner = Container.FAST_CLONER
```

1.5 Directory Services

1.5.1 Advertising services

It is often undesirable to hardcode names of agents that we need to interact with. Directory services provide a simple mechanism to advertise services provided by an agent, and to find agents that provide specific services. A *service* is a contract between two agents, and usually defined by a set of messages and possibly behaviors.

Tip: A *service* is a logical concept, and not enforced by the framework. The messages and behaviors represented by a service are not described in the code. However, it is recommended that the documentation associated with the service clearly spell out the messages and behaviors that are expected of any agent claiming to provide the service.

An agent providing a service usually advertises the service during *init()*:

```
class MyServer extends org.arl.fjage.Agent {
  void init() {
    register 'WeatherForecastService'
  }
}
```

Rather than use a *String*, we may (and usually should) use an Enum to define the service:

```
enum Services {
  WEATHER_FORECAST_SERVICE,
  CLEANING_SERVICE,
  FOOD_DELIVERY_SERVICE
}
```

and then use it to advertise our services:

```
class MyServer extends org.arl.fjage.Agent {
  void init() {
    register Services.WEATHER_FORECAST_SERVICE
  }
}
```

1.5.2 Looking up service providers

A client interested in availing a specific service can look for an agent that provides the service:

```
def weatherStation = agentForService Services.WEATHER_FORECAST_SERVICE
def rsp = weatherStation << new WeatherForecastReq(city: 'London', country: 'UK')
```

If there are more than one agents providing the service, the *agentForService()* method returns any one of the service providers. If we wish to get a list of all service providers, we can use the *agentsForService()* method instead:

```
def providerList = agentsForService Services.WEATHER_FORECAST_SERVICE
```

1.5.3 Caching service providers

If your application uses a set of agents that are instantiated in the *initrc.groovy* and not terminated until the application terminates, it may be reasonable to lookup the service providers once and cache them once and for all. However, since the services are only advertised during agent initialization and the order of agent intialization may be indeterminate, the service lookups should be done after all agents are initialized (and not during *init()*). This can easily be accomplished using a one-shot behavior:

```
class MyClient extends org.arl.fjage.Agent {
  def weatherStation
  void init() {
    add new OneShotBehavior({
      weatherStation = agentForService Services.WEATHER_FORECAST_SERVICE
    })
  }
}
```

Note: As long as the agents are added to the container before starting the platform, fjåge guarantees that all agents are initialized before any agent behaviors are called.

1.6 Distributed Agents

1.6.1 Master and slave containers

Once we have developed our agents, it is easy to deploy them on multiple nodes as necessary. To do so, we require one MasterContainer in our application, and any number of SlaveContainer. The master container must be started first, and the slave containers connect to it.

To start a master container, we simply replace *Container* with *MasterContainer* in the *initrc.groovy*:

```
import org.arl.fjage.*
import org.arl.fjage.remote.*

platform = new RealTimePlatform()
container = new MasterContainer(platform, name)
println "Master container started on port ${container.port}"
// add agents to the container here
platform.start()
```

Specifying the *name* for the master container is optional, but recommended. Any *String* can be used as the container name. An additional parameter *port* may be specified while constructing the *MasterContainer*, if desired. In the absence of this parameter, the TCP port number is automatically chosen.

To start slave containers, we need to specify the hostname and TCP port of the master container:

```
import org.arl.fjage.*
import org.arl.fjage.remote.*

platform = new RealTimePlatform()
container = new SlaveContainer(platform, hostname, port)
// add agents to the container here
platform.start()
```

That's it! We can deploy agents on any of the containers in the system, and they can interact with agents from other containers transparently.

1.6.2 Remote console

It is often useful to connect a console shell to a running fjåge application to monitor, interrogate or modify it. To do this, we ensure that the application is running in a master container (and possible some slave containers). We then create a *rconsole.sh*:

```
#!/bin/sh

CLASSPATH=`find build/libs -name *.jar -exec /bin/echo -n :'{}' \;`
java -cp "$CLASSPATH" -Dhostname="$1" -Dport="$2" org.arl.fjage.shell.GroovyBoot etc/
↪initrc-rconsole.groovy
```

and *etc/initrc-rconsole.groovy*:

```
import org.arl.fjage.*
import org.arl.fjage.remote.*
import org.arl.fjage.shell.*

String hostname =  System.properties.getProperty('hostname')
```

(continues on next page)

(continued from previous page)

```
if (hostname == null || hostname.length() == 0) hostname = 'localhost'
int port
try {
  port =  Integer.parseInt(System.properties.getProperty('port'))
} catch (Exception ex) {
  port = 5081
}
println "Connecting to $hostname:$port..."
platform = new RealTimePlatform()
container = new SlaveContainer(platform, hostname, port)
shell = new ShellAgent(new ConsoleShell(), new GroovyScriptEngine())
container.add 'rshell', shell
platform.start()
```

The shell script passes the hostname and TCP port specified on the command line to the initialization Groovy script, that connects to the master container and offers a local console shell for the user to interact. Assuming you have a fjåge application running locally on port 5081, you can connect to it:

```
./rconsole.sh localhost 5081
```

1.6.3 Interacting with agents using a Gateway

Only agents may access messaging and related functionality provided by fjåge. For example, non-agent Java or Groovy threads cannot send messages to, or receive messages. To aid interaction of such threads with agents, fjåge provides a Gateway class. This class provides agent-like functionality to non-agent threads by creating a proxy agent in a slave container that has access to this functionality. Using the *Gateway* is fairly simple:

```
Gateway gw = Gateway(hostname, port)
def weatherStation = gw.agentForService Services.WEATHER_FORECAST_SERVICE
def rsp = gw.request new WeatherForecastReq(city: 'London', country: 'UK', recipient:␣
↪weatherStation)
println "The lowest temperature today is ${rsp?rsp.minTemp:'unknown'}"
gw.shutdown()
```

1.7 The Shell Agent

We have already used the console shell provided by fjåge many times. This shell is implemented by the *ShellAgent* class as we have seen before in the *initrc.groovy* scripts. Let's take a slightly deeper look at the shell agent in this chapter.

1.7.1 Shell commands

The default shell provided by fjåge is a Groovy shell, and can execute any valid Groovy code. A few standard commands, variables and closures are made available. Just typing *help* will provide a list of commands that are available:

```
bash$ ./fjage.sh
> help
help [topic] - provide help on a specified topic
ps - list all the agents
```

(continues on next page)

(continued from previous page)

```
services - lists all services provided by agents
who - display list of variables in workspace
run - run a Groovy script
println - display message on console
delay - delay execution by the specified number of milliseconds
shutdown - shutdown the local platform
logLevel - set loglevel (optionally for a named logger)
subscribe - subscribe to notifications from a named topic
unsubscribe - unsubscribe from notifications for a named topic
export - add specified package/classes to list of imports
agent - return an agent id for the named agent
agentForService - find an agent id providing the specified service
agentsForService - get a list of all agent ids providing the specified service
send - send the given message
request - send the given request and wait for a response
receive - wait for a message
>
```

Further help on an individual topic can be obtained by typing *help* followed by the topic name. You are encouraged to explore the help.

The commands in the shell are executed in the context of the *ShellAgent* (e.g. messages send are send using this agent). Any messages received by the *ShellAgent* are simply displayed.

Tip: If you wish to add your own closures or variables, you can do so by customizing initialization script. Initialization scripts can be added to the *ShellAgent* using the *addInitrc* method.

1.7.2 Remote shell over TCP/IP

If we wanted to provide a remote shell that users could *telnet* into, rather than a console shell, we would replace *ConsoleShell* with a *TcpShell* and specify a TCP/IP port number that is to provide the interactive shell. Here's what the resulting *initrc.groovy* would look like:

```groovy
import org.arl.fjage.*
import org.arl.fjage.shell.*

platform = new RealTimePlatform()
container = new Container(platform)
shell = new ShellAgent(new TcpShell(8001), new GroovyScriptEngine())
container.add 'shell', shell
// add other agents to the container here
platform.start()
```

We could then access the shell using *telnet*:

```
bash$ telnet localhost 8001
Trying localhost...
Connected to localhost
Escape character is '^]'.
> ps
shell
>
```

1.7.3 GUI shell using Java Swing

The *SwingShell* GUI has been deprecated and no longer available in fjåge 1.5 and above. Use the web-based shell instead.

1.7.4 Web-based shell

A web-based shell is available for users to access using a browser. An *initrc.groovy* enabling the web shell on port 8080 would look like this:

```
import org.arl.fjage.*
import org.arl.fjage.shell.*
import org.arl.fjage.connectors.*

platform = new RealTimePlatform()
container = new Container(platform)
WebServer.getInstance(8080).add("/", "/org/arl/fjage/web")
Connector conn = new WebSocketConnector(8080, "/shell/ws")
shell = new ShellAgent(new ConsoleShell(conn), new GroovyScriptEngine())
container.add 'shell', shell
// add other agents to the container here
platform.start()
```

The shell can be accessed by accessing http://localhost:8080 once fjåge is running.

Tip: The web-based shell uses the Jetty web server. For this to work, the Jetty classes need to be in the classpath. This is automatically done for you if you use the Maven repository to download fjåge and its dependencies. If you used the quickstart script to start using fjåge, you may have to manually download the Jetty web server jars into the *build/lib* folder.

1.7.5 Shell extensions

Shell extensions are classes that extend the *org.arl.fjage.shell.ShellExtension* interface, and can be executed in a shell using the agent's *addInitrc()* method or using *run()*. This interface is simply a tag, and does not contain any methods. All public static methods and attributes (except those that contain "__" in the name) of the extension class are imported into the shell as commands and constants.

If the extension has a *public static void __init__(ScriptEngine engine)* method, it is executed at startup. If the extension has a public static string attribute called *__doc__* , it is loaded into the documentation system. The documentation system interprets it's inputs as Markdown help snippets. A first level heading provides a top level description for the extension. Individual commands and attributes should be described in sections with second level headings.

An simple Groovy extension example is shown below:

```
class DemoShellExt implements org.arl.fjage.shell.ShellExtension {

static final public String __doc__ = '''\
# demo - demo shell extension

This shell extension imports all classes from the package
"my.special.package" into the shell. In addition, it adds
a command "hello", which is described below:

```

(continues on next page)

(continued from previous page)

```
## hello - say hello to the world

Usage:
  hello               // say hello
  hello()             // say hello

Example:
> hello
Hello world!!!
'''

    static void __init__(ScriptEngine engine) {
        engine.importClasses('my.special.package.*')
    }

    static String hello() {
        return 'Hello world!!!'
    }

}
```

1.8 Parameters

1.8.1 Simple parameters

Parameters are generic key-value pairs that can be read/written using *ParameterReq* messages. Agents often use parameters to provide status information, or to allow users to configure the agent. Parameters also have a special syntax in the shell, making it easy for users to interact with the agent.

In order for an agent to support parameters, it needs to: 1. Create an *enum* implementing the Parameter interface to list all the supported parameters. 2. Add a ParameterMessageBehavior to handle ParameterReq messages.

Let us explore this using an example. In order to expose parameters, we must first define an *enum* with the supported paramaters:

```
enum MyParams implements org.arl.fjage.param.Parameter {
  x,
  y
}
```

The above *enum MyParams* defines two parameters – *x* and *y*. An agent supporting these parameters defines them as properties with appropriate getters and setters using the JavaBean convention in Java:

```
public class MyAgentWithParams extends org.arl.fjage.Agent {

  private int x = 42;            // read-only parameter
  private String y = "hello";    // read-write parameter

  public int getX() {
    return x;
  }

  public String getY() {
    return y;
```

(continues on next page)

(continued from previous page)

```
  }

  public void setY(String s) {
    y = s;
  }

  public void init() {
    // add the behavior to deal with ParameterReq messages
    add(new org.arl.fjage.param.ParameterMessageBehavior(MyParams.class));
  }

}
```

Groovy automatically creates the getters and setters for us, and so the implementation in Groovy would look much simpler:

```
class MyAgentWithParams extends org.arl.fjage.Agent {

  final int x = 42                // final tells Groovy that this is read-only
  String y = "hello"              // read-write parameter

  void init() {
    // add the behavior to deal with ParameterReq messages
    add new org.arl.fjage.param.ParameterMessageBehavior(MyParams)
  }

}
```

That's it!

Let's try out our agent using the shell:

```
bash$ ./fjage.sh
> container.add 'a', new MyAgentWithParams()
a
> a
« A »

[MyParams]
  x   42
  y = hello

> a.x
42
> a.y
hello
> a.y = 'hi'
hi
> a
« A »

[MyParams]
  x   42
  y = hi

> a.x = 7
Parameter x could not be set
```

Since the *x* parameter is read-only (denoted by), we can only *get* its value, and not *set* it. The *y* parameter, on the other hand allows both getting/setting.

It is important to note that although the notation looks similar to reading/writing an attribute of the class, the actual interaction with the agent is via the ParameterReq and ParameterRsp messages. Unlike class attributes, this allows access to parameters in remote containers. We can directly sent the ParameterReq message to explicitly see this interaction:

```
> import org.arl.fjage.param.*
> a << new ParameterReq()
ParameterRsp[x*:42 y:hi]
> a << new ParameterReq().get(MyParams.y)
ParameterRsp[y:hi]
> a << new ParameterReq().set(MyParams.y, "howdy?")
ParameterRsp[y:howdy?]
```

We can also use the *AgentID get/set* convenience methods to access parameters:

```
> a.get(MyParams.x)
42
> a.set(MyParams.y, 'hiya!')
hiya!
```

Tip: Since there are so many ways to access parameters, which one should you use? The attribute notation *a.x* is simple and clear, and is best used in the shell. This notation is not available in Java or statically typed Groovy code, as it uses Groovy's dynamic features. We, therefore, recommend using the *AgentID.get()* and *AgentID.set()* methods instead from Java/Groovy code.

1.8.2 Dynamic parameters

We saw that setting and setting parameters stored as agent properties is simple. But what if we wanted to generate the values of the parameters dynamically? From the Java example in the previous section, the answer is obvious. You can generate the value of the parameter dynamically in the getter method. The same applies in Groovy agents. We show an example below:

```
class MyAgentWithParams extends org.arl.fjage.Agent {

  int x = 42

  int getY() {
    return x + 7
  }

  void init() {
    add new org.arl.fjage.param.ParameterMessageBehavior(MyParams)
  }

}
```

This creates a simple read-write parameter *x* and a dynamic read-only parameter *y*, with a value depending on *x*. Let us test it out:

```
bash$ ./fjage.sh
> container.add 'a', new MyAgentWithParams()
```

(continues on next page)

(continued from previous page)

```
a
> a
« A »

[MyParams]
  x = 42
  y   49

> a.x = 7
7
> a.y
14
```

1.8.3 Metadata paramters

fjåge defines a few *standard* meta-parameters that every agent with a ParameterMessageBehavior supports:

1. *name*: name of the agent
2. *type*: class of the agent
3. *title*: descriptive title for the agent (defaults to name, if not explicitly defined by agent)
4. *description*: description of the agent

To see how these parameters work, let us modify our agent to add a *title* and *description*:

```
class MyAgentWithParams extends org.arl.fjage.Agent {

  final static String title = "My agent with parameters"
  final static String description = "This is a sample agent to demonstrate the use of␣
↪parameters"

  int x = 42
  int y = 7

  void init() {
    add new org.arl.fjage.param.ParameterMessageBehavior(MyParams)
  }

}
```

Now, running the agent, we see the title and description when we lookup the agent parameters:

```
bash$ ./fjage.sh
> container.add 'a', new MyAgentWithParams()
a
> a
« My agent with parameters »

This is a sample agent to demonstrate the use of parameters

[MyParams]
  x = 42
  y = 7
```

1.8.4 Indexed parameters

Sometimes it is useful to have multiple parameters with the same name, but addressed by a numerical index. As an example, let us consider an agent that provides a telephone directory. It supports three indexed parameters: *firstname*, *lastname* and *phone*:

```
enum MyParams implements org.arl.fjage.param.Parameter {
  firstname,
  lastname,
  phone
}
```

Indexed parameters are defined using getters/setters with indexes. In addition, a *getParameterList* method needs to be overridden to provide a list of parameters for a given index (different indexes may provide different parameters, if desired):

```
class MyAgentWithParams extends org.arl.fjage.Agent {

  String getFirstname(int i) {
    if (i == 1) return "John"
    if (i == 2) return "Alice"
    return null
  }

  String getLastname(int i) {
    if (i == 1) return "Doe"
    if (i == 2) return "Wonderland"
    return null
  }

  String getPhone(int i) {
    if (i == 1) return "+123456789"
    if (i == 2) return "+987654321"
    return null
  }

  void init() {
    add new org.arl.fjage.param.ParameterMessageBehavior() {
      @Override
      List<? extends org.arl.fjage.param.Parameter> getParameterList(int i) {
        return allOf(MyParams)
      }
    }
  }

}
```

Now, we can test the indexed parameters:

```
bash$ ./fjage.sh
> container.add 'a', new MyAgentWithParams()
a
> a
« A »

> a[1]
« A »
```

(continues on next page)

(continued from previous page)

```
[MyParams]
  firstname  John
  lastname  Doe
  phone  +123456789

> a[2]
« A »

[MyParams]
  firstname  Alice
  lastname  Wonderland
  phone  +987654321

> a[2].firstname
Alice
> a[1].phone
+123456789
```

1.9 Simulation

1.9.1 Discrete Event Simulation

When used in the Discrete event simulation mode, fjåge allows agents to be tested rapidly through the notion of *virtual time*. The passage of virtual time is simulated such that computation and processing does not take any virtual time, while scheduling requests are met accurately in virtual time. The virtual time advances in discrete steps such that the time when no agent is active is effectively skipped. This potentially allows for simulation of hours of virtual time within seconds.

In order to use the discrete event simulation mode, a few conditions have to be met. The first two of these conditions are related to timing functions and were introduced in the "*Introduction*" chapter. The last condition is related to agents deployed in slave containers.

1. Agents must not use any system timing functions directly. Rather than use *System.currentTimeMillis()* or *System.nanoTime()*, the agents should use *Agent.currentTimeMillis()* and *Agent.nanoTime()*.
2. Agents must not use any system scheduling functions directly. Rather than use *Thread.sleep()*, the agent should use *Agent.delay()*.
3. All agents must be deployed in a single container for testing. Distributed containers (master or slave) are currently not supported by the discrete event simulator.

To run the agents in the discrete event simulation mode, the use of the *RealTimePlatform* in *etc/initrc.groovy* is replaced by *DiscreteEventSimulator*:

```
import org.arl.fjage.*

platform = new DiscreteEventSimulator()
container = new Container(platform)
// add agents to the container here
platform.start()
```

Now running *fjage.sh* should run the agents in the discrete event simulation mode. You can verify this by looking at the *logs/log-0.txt* file; the time entries in this file will start at 0, since all simulations start at time 0. When all agents become idle with no further events in the system, the discrete event simulator automatically terminates.

1.10 Python Gateway

1.10.1 Introduction

This Python package provides a *Gateway* class to interact with the fjåge agents. The fjåge agents reside in one or more containers that provide agent management, directory and messaging services. Various containers may run on the same node or on different nodes in a network. This Python *Gateway* class allows the external Python scripts to interact with fjåge agents using an interface implemented in Python. The Python APIs use the package *fjagepy*.

The first step is to install the *fjagepy* package using:

```
pip install fjagepy
```

Import all the necessary symbols from *fjagepy* package:

```
from fjagepy import Gateway, AgentID, Message, MessageClass, GenericMessage,␣
↪Performative
```

or just:

```
from fjagepy import *
```

1.10.2 Import message classes

Since the Java/Groovy message classes are not directly available in Python, we use the *MessageClass* utility to dynamically create specified message classes. An example of such is shown:

```
ShellExecReq = MessageClass('org.arl.fjage.shell.ShellExecReq')
```

The *ShellExecReq* class can now be used to instantiate new objects like:

```
msg = ShellExecReq()
```

The fully qualified class name as a string must be provided as an argument to this method. The fully qualified class names that are already supported by fjåge are documented here.

1.10.3 Open a connection

If a fjage server is running, we can create a connection using *Gateway* class:

```
gw = Gateway(hostname, port)
```

where *hostname* and *port* is the IP address and the port number of the device on which the fjåge server is running. The *gw* object is created which can be used to call the methods of *Gateway* class.

1.10.4 Send and receive messages

We have seen earlier that the agents interact with each other using messages. The python gateway can similarly send and receive messages to the agents running on containers running on diffeent machines. An example of request and response message are as shown below:

Request message:

```
msg = Message()
msg.recipient = abc
gw.send(msg)
```

where *abc* is the AgentID of the agent you are trying to send the message to.

Another alternative to send a message is following:

```
msg = Message(recipient = abc)
rsp = gw.request(msg, timeout)
```

In the above code snippet, a request method is used to send a message and receive the response back. Different responses that can be received are documented here.

msg is an instance of *Message* class and in the ablove example, the intended recipient is set to the AgentID *abc*. The constructed message *msg* is sent to the agents running on master container using *gw.send(msg)*.

A simple example of executing a shell command from remote connection opened using Gateway class is as shown below:

```
gw = Gateway(hostname, port)
ShellExecReq = MessageClass('org.arl.fjage.shell.ShellExecReq')
shell = gw.agentForService('org.arl.fjage.shell.Services.SHELL')
req = ShellExecReq(recipient=shell, cmd = 'ps')
rsp = gw.request(req, 1000)
print(rsp)
gw.close()
```

In the code above, we first open a connection to the fjåge server. Next, we import the *ShellExecReq* message that we will require later. We want to send this message to an agent which supports the *SHELL* service (honoring the *ShellExecReq* messages). The *agentForService* method of the *Gateway* class allows us to look up that agent. Next, we construct the *ShellExecReq* message to request execution of a shell command (in this case *ps*). The *request* method then sends the message and waits for a response, which we then print and close the connection.

1.10.5 Generic messages

As the use case of *GenericMessage* is already explained before, we will illustrate it's use using the Python gateway API:

```
gw = Gateway(hostname, port)
shell = gw.agentForService('org.arl.fjage.shell.Services.SHELL')
gmsg = GenericMessage(recipient=shell, text='hello', data=np.random.randint(0,9,
↪(100)))
gw.send(gmsg)
```

The shell agent running on the server side will receive this generic message sent through gateway:

```
rgmsg = receive(GenericMessage, 1000)
println rgmsg.text
println rgmsg.data
```

1.10.6 Publish and subscribe

We know that there are times when we may want to publish a message without explicitly knowing who the recipients are. All agents subscribing to the topic that we publish on would then receive the published message. For example:

```
gw.topic('abc')
```

returns an object representing the named topic. A user can subscribe to this topic using:

```
gw.subscribe(gw.topic('abc'))
```

But if we are interested in receiving all the messages sent from a particular agent whose *AgentID* we know (for example *shell*), then:

```
shell = gw.agentForService('org.arl.fjage.shell.Services.SHELL')
gw.subscribe(shell)
```

will allow to receive the published messages by *shell* agent.

1.10.7 Close a connection:

In order to close the connection to the fjåge server, we can call the *close* method provided by the *Gateway* class:

```
gw.close()
```

1.11 C Gateway

1.11.1 Introduction

This C package provides a Gateway interface to interact with the fjåge agents. The fjåge agents reside in one or more containers that provide agent management, directory and messaging services. Various containers may run on the same node or on different nodes in a network. This C gateway interface allows the external C programs to interact with fjåge agents.

The first step is to compile the library:

```
git clone git@github.com:org-arl/fjage.git
cd src/main/c
make
```

if all goes well, you should have a *libfjage.a* file in the folder after compilation. You'll need this and the *fjage.h* file to link your C program against.

In your C program:

```
#include "fjage.h"
```

1.11.2 Open a connection

If a fjage server is running, we can create a connection using *Gateway* class:

```
fjage_gw_t gw = fjage_tcp_open(hostname, port);
```

where *hostname* and *port* is the IP address and the port number of the device on which the fjåge server is running. This returns a gateway handle (or NULL on error) that is required by the rest of the API.

1.11.3 Send and receive messages

We have seen earlier that the agents interact with each other using messages. The C gateway can similarly send and receive messages to the agents running on containers running on diffeent machines. An example of request and response message are as shown below:

Request message:

```
fjage_msg_t msg = fjage_msg_create("org.arl.fjage.Message", FJAGE_REQUEST);
fjage_aid_t aid = fjage_aid_create("abc");
fjage_msg_set_recipient(msg, myaid);
fjage_send(gw, msg);
```

where *abc* is the name of the agent you are trying to send the message to. Once the message is sent, the message and the agentID needs to be freed:

```
fjage_aid_destroy(aid);
```

However, a successfully sent message should not be freed by the caller.

1.11.4 Close a connection:

In order to close the connection to the fjåge server:

```
fjage_close(gw);
```

1.11.5 Simple example

A simple example of executing a shell command from remote connection is shown below:

```
#include <stdio.h>
#include "fjage.h"

int main() {
   fjage_gw_t gw = fjage_tcp_open("localhost", 5081);
   if (gw == NULL) {
      printf("Connection failed\n");
      return 1;
   }
   fjage_aid_t aid = fjage_agent_for_service(gw, "org.arl.fjage.shell.Services.SHELL
↪");
   if (aid == NULL) {
      printf("Could not find SHELL agent\n");
      fjage_close(gw);
      return 1;
   }
   fjage_msg_t msg = fjage_msg_create("org.arl.fjage.shell.ShellExecReq", FJAGE_
↪REQUEST);
   fjage_msg_set_recipient(msg, aid);
   fjage_msg_add_string(msg, "cmd", "ps");
   fjage_msg_t rsp = fjage_request(gw, msg, 1000);
   if (rsp != NULL && fjage_msg_get_performative(rsp) == FJAGE_AGREE) printf(
↪"SUCCESS\n");
   else printf("FAILURE\n");
   if (rsp != NULL) fjage_msg_destroy(rsp);
```

(continues on next page)

(continued from previous page)

```
   fjage_aid_destroy(aid);
   fjage_close(gw);
   return 0;
}
```

This is compiled using *gcc -o demo.out demo.c libfjage.a* assuming that this file is saved as *demo.c*.

1.11.6 API documentation

This only scratches the surface of what can be done with the fjåge C gateway. For more, refer to the documentation in the C header file (*fjage.h* shown below) and examples in the test script (test_fjage.c).

1.12 Javascipt Gateway

1.12.1 Introduction

The Javascript Gateway allows web applications to access fjåge agents. While the Javascript API is very similar to the Python Gateway API, there are a few additional steps required to setup the web services needed for the Javascript API to work. Being limited by the single threaded browser model, the Javascript API uses promises and callbacks to deliver results from APIs that may incur latency.

1.12.2 Enable the web sockets connector

First the web sockets connector has to be enabled (let's say on port 8080), so that fjåge can be accessed over web sockets from a browser:

```
import org.arl.fjage.*
import org.arl.fjage.shell.*
import org.arl.fjage.connectors.*

platform = new RealTimePlatform()
container = new Container(platform)
def websvr = org.arl.fjage.connectors.WebServer.getInstance(8080)
websvr.add('/fjage', '/org/arl/fjage/web')
// add any other contexts needed to serve your application here
container.addConnector(new WebSocketConnector(8080, "/ws", true))
container.add 'shell', shell
// add other agents to the container here
platform.start()
```

1.12.3 Use the Javascript module

It is easiest to illustrate the use of the Javascript API though a simple code example:

```
import { Gateway, MessageClass, Performative } from '/fjage/fjage.js';

var gw = new Gateway();

MessageClass('org.arl.fjage.shell.ShellExecReq');
```

(continues on next page)

(continued from previous page)

```
gw.agentForService('org.arl.fjage.shell.Services.SHELL').then((aid) => {
    shell = aid;
    gw.subscribe(gw.topic(shell));
    makeRq(shell);
}).catch((ex) => {
    console.log('Could not find SHELL: '+ex);
});

gw.addMessageListener((msg) => {
    console.log(msg);
    return false;
});

function makeRq(shell) {
    let req = new ShellExecReq();
    req.recipient = shell;
    req.cmd = 'ps';
    gw.request(req).then((msg) => {
        console.log(msg);
    }).catch((ex) => {
        console.log('Could not execute command: '+ex);
    });
}
```

This code first opens a gateway through the web socket interface back to the web server that served this Javascript. It then imports the *org.arl.fjage.shell.ShellExecReq* message class, and looks for an agent providing the *org.arl.fjage.shell.Services.SHELL* service. If found, it subscribes to messages from that service and calls *makeRq()* to make a command execution request to the agent providing that service. The request is to execute a command *"ps"* and simply log the response to the browser's console.

The user should refer to the detailed API description for the Javascript API for more information.

1.13 JSON Protocol Specifications

1.13.1 Transport and framing

fjåge uses JSON for communication between remote containers. The JSON objects are exchanged over some low level networked connection like TCP. Rest of this section defines the protocol of the JSON objects exchanged between the containers.

fjåge containers are connected to each other over some form of networked transport. Typically this is a TCP connection, however, the JSON protocol does not assume any specific behavior of the underlying transport connection. Other transports like Serial-UART may also be used.

Over the networked transport, the containers communicate using line delimited JSON messages. These JSON objects are framed by a newline characters (\n or \r or \r\n). Each such frame contain a single JSON object which adheres to the JSON as defined in RFC7159, and does not support unescaped new-line characters inside a JSON object. The prettified JSON objects with new-lines are shown in this sections as examples to understand and should be "JSONified" before being used.

1.13.2 JSON object format

Basics

fjåge's JSON protocol objects are typically shallow JSON objects. The first level of attributes are typically used by the containers to hold metadata and perform housekeeping tasks such as agent directory service. The attribute message in the JSON object contains the actual message that is exchanged between agents residing in different containers. We describe the JSON message format below which when sent the task requested and respond with relevant notification JSON message, which the developer must look for and parse carefully. Next, we describe in detail the JSON message format which fjåge understands.

JSON message request/response attributes

A JsonMessage class is defined in fjåge which support a list of attributes. The attributes supported at the top level of the JSON object are listed below:

- *id* : **String** - A UUID assigned to to each object.
- *action* : **String** - Denotes the main action the object is supposed to perform. Valid JSON message actions supported are listed here:
  - *agents* - Request for a list of all agents running on the target container.
  - *containsAgent* - Request to check if a specific container has an agent with a given AgentID running.
  - *services* - Request for a list of all services running on the target container.
  - *agentForService* - Request for AgentID of an agent that is providing a specific service.
  - *agentsForService* - Request for AgentID of all agents that is providing a specific service.
  - *send* - Request to send a payload to the target container.
  - *shutdown* - Request to shutdown the target container.
- *inResponseTo* : **String** - This attribute contains the action to which this object is a response to. A response object will have the exact same id as the original action object.
- *agentID* : **String** - An AgentID. This attribute is populated in objects which are responses to objects requesting the ID of an agent providing a specific service *"action" : "agentForService"*. This field may also be used in objects with *"action" : "containsAgent"* to check if an agent with the given AgentID is running on a target container.
- *agentIDs*: **Array** - This attribute is populated in objects which are responses to objects requesting the IDs of agents providing a specific service with *"action" : "agentsForService"*, or objects which are responses to objects requesting a list of all agents running in a container.
- *agentTypes*: **Array** - This attribute is optionally populated in objects which are responses to objects requesting a list of all agents running in a container. If populated, it contains a list of agent types running in the container, with a one-to-one mapping to the agent IDs in the *"agentIDs"* attribute.
- *service* : **String** - Used in conjunction with *"action" : "agentForService"* and *"action" : "agentsForService"* to query for agent(s) providing this specific service.
- *services*: **Array** - This attribute is populated in objects which are responses to objects requesting the services available with *"action" : "services"*.
- *answer* : **Boolean** - This attribute is populated in objects which are responses to query objects with *"action" : "containsAgent"*.
- *relay* : **Boolean** - This attribute defines if the target container should relay (forward) the message to other containers it is connected to or not.

- *message* : **Object** - This holds two main attributes and is responsible for carrying the main payload. The first field is *clazz* and the second *data*. Note that the ordering of *clazz* and *data* fields is crucial. The developer must make sure that the *clazz* field comes ahead of *data* field. The structure and format of this object is discussed here:
  - *clazz* : **String** - A string identifier that identifies the type of the message. This is usually a fully qualified Java class name of the class of that type of message.
  - *data* : **Object** - The main payload containing data and message attributes. This holds the contents of the payload in objects with *"action" : "send"*. The structure and format of this object is discussed here:
    - *data* : **Object** - The main payload containing the data and type of data. **NOTE**: While sending a JSON message, the developer can choose to either follow this format as converting the data to Base64 and specifying the equivalent *clazz* or the data can be directly inserted as an array of numbers without specifying the *clazz* or *data* fields as explained later in the examples section. - *clazz* : **String** - This attribute contains the string to identify the type of data being carried by the JSON object. The types that are identified and supported are: *"[F"* - Float, *"[I"* - Integer, *"[D"* - Double, *"[J"* - Long, *"[B"* - Bytestring - *data* or *signal* : **Base64 String** - The data is encoded as a Base64 string and populated in this attribute. Either the *data* or *signal* attribute is used depending on the message that is being received or sent.
    - *msgID* : **String** - A UUID assigned to each message.
    - *perf* : **String** - Performative. Defines the purpose of the message. Valid performatives are:
      - *REQUEST* - Request an action to be performed.
      - *AGREE* - Agree to performing the requested action.
      - *REFUSE* - Refuse to perform the requested action.
      - *FAILURE* - Notification of failure to perform a requested or agreed action.
      - *INFORM* - Notification of an event.
      - *CONFIRM* - Confirm that the answer to a query is true.
      - *DISCONFIRM* - Confirm that the answer to a query is false.
      - *QUERY_IF* - Query if some statement is true or false.
      - *NOT_UNDERSTOOD* - Notification that a message was not understood.
      - *CFP* - Call for proposal.
      - *PROPOSE* - Response for CFP.
      - *CANCEL* - Cancel pending request.
    - *recipient* : **String** - An AgentID of the fjåge agent this message is being addressed to.
    - *sender* : **String** - An AgentID of the fjåge agent this message is being sent by.
    - *inReplyTo* : **String** - A UUID. Included in a reply to another object to indicate that this object is a reply to an object with this id.

Note that not all the above attributes need to be populated in a JSON message. The attributes depend on the task that needs to be executed by the agent running in the target container. Also, the message attribute may have additional attributes depending on the exact message that is being constructed.

Next, we describe some of the basic examples in order to let the developer understand what JSON messages to send and how to construct them for different use cases.

1.13.3 Examples

fjåge has very few built-in messages, since messages are usually introduced by the application built on top of fjåge. We therefore use examples from UnetStack, a specific application using fjåge, to show how the protocol works. To understand the fields in the messages, the reader is referred to UnetStack API documentation.

Simple JSON message to transmit a CONTROL frame:

The JSON message of a *TxFrameReq* message to transmit a *CONTROL* frame is as shown below. The *message* attribute contains the attributes specific to *TxFrameReq* message:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.phy.TxFrameReq",
    "data": {
      "type": 1,
      "data": [ 1, 2, 3],
      "msgID": "a2fbff38-a0fb-4e3a-bf22-ae6cf4642e6b",
      "perf": "REQUEST",
      "recipient": "phy",
      "sender": "MyCustomInterface"
    }
  }
}
```

A JSON message sent by UnetStack in response to the JSON message sent is as given below:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.phy.TxFrameNtf",
    "data": {
      "txTime": 3329986666,
      "type": 1,
      "msgID": "dc227a96-4d6e-4b64-9d55-bb108ea338b0",
      "perf": "INFORM",
      "recipient": "MyCustomInterface",
      "sender": "phy",
      "inReplyTo": "a2fbff38-a0fb-4e3a-bf22-ae6cf4642e6b"
    }
  },
  "relay": false
}
```

Note that there is a attribute *inReplyTo* populated in the response received which indicates that this JSON message was in reply to the JSON message with exact same *msgID*.

JSON message with byte array to transmit a Datagram:

Now, let us try to send a string *Hello World!* as a bytestring. This demonstrates the use of the *clazz* attribute in the *data* payload:

```
{
  "action": "send",
```

(continues on next page)

(continued from previous page)

```
  "message": {
    "clazz": "org.arl.unet.DatagramReq",
    "data": {
      "data": {
        "clazz": "[B",
        "data": "aGVsbG8gd29ybGQh"
      },
      "msgID": "8152310b-155d-4303-9621-c610e036b373",
      "perf": "REQUEST",
      "recipient": "phy",
      "sender": "MyCustomInterface"
    }
  }
}
```

JSON response sent by UnetStack:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.phy.TxFrameNtf",
    "data": {
      "txTime": 4550354666,
      "type": 1,
      "msgID": "fde91abf-68ac-4a93-b2ae-27d1cee01869",
      "perf": "INFORM",
      "recipient": "MyCustomInterface",
      "sender": "phy",
      "inReplyTo": "8152310b-155d-4303-9621-c610e036b373"
    }
  },
  "relay": false
}
```

JSON message with float array to record a baseband signal:

Another example of a data payload with a floating point array is shown in this section. In order to record a baseband signal with 100 baseband samples, we send the following JSON message:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.bb.RecordBasebandSignalReq",
    "data": {
      "recLen": 100,
      "msgID": "28db3bd4-ad14-4d86-b4a0-a2d8ebb3cb65",
      "perf": "REQUEST",
      "recipient": "phy",
      "sender": "MyCustomInterface"
    }
  }
}
```

The specific attribute such as *recLen* is message specific which in this case is *RecordBasebandSignalReq* and the relevant supported attributes can be found online at the UnetStack API documentation.

In response to the the JSON message to record baseband samples, the UnetStack sends a JSON message equivalent to the *RxBasebandSignalNtf* message containing the recorded data and is as shown here:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.bb.RxBasebandSignalNtf",
    "data": {
      "rxTime": 4905996833,
      "rssi": -43.190178,
      "adc": 1,
      "signal": {
        "clazz": "[F",
        "data": "vC2OBzxUeQs6EtF4O/ATCLyZgk27pg6hvHqGsTunBuC74ZmUOm/
↪YVrwfOrY7pBx+PEGvhLwbr4q7sk/
↪xvDaq4rtrNG87675VPDtOFjtaAPG6x2lQulmJIDqUZdG8PNRSu+uZGrx4Q4S7ngawu5e0Kju79bC7gT2iOpQ29LvDqvu71/
↪10vI/ndryL2rm8QPqau9ZmrzuRM7c7iNw0PALqJDwFH8+7jePovBiQYrwIpAY8CokaPFNJZDwp/
↪KM7Huoau9u1bDrIGEi6fkjSPAafjTsxGsc7mFnyO4J5D7t9dC66q6MYua+lXjwawm47EzIoufw6ibwQvGS73fHmu47QZTs0Ihs
↪eq5sOUAOs9hMjvMOI+8JT/VO7hmaTqB8lo8SCOZO/dsHzvVkYy7A1s/
↪OxyyTjqff5c7xVBZO3Jh6zsu1l07N1nAO6ljuTvKbu073bpNOlKv2LkAWaC7RBxSOs6XzDujzn06ySzaOwl4ervmgLY6xnA4uq
↪4+69s6MutZNiDtgdnC7H4Icu8RV0ruvX2y8HjuwuoGU3rvxp4+6iMx2Ou3fsDqvFyc78w5wOuQNwzsrQkM7KgS1OzyTCjuVmvy
↪DrEvQa6yUQGOq6sYjsBG0w7fkHyu5A0pTooWri7DpmKur+gyjqvUm+7aDZ3u5P/
↪5jr7IL26vDzSO5oyDDobQxS6tijpOn8iSLmH8OU6mg1KO7LFdLsvemo7KST5up0Kuru2gLe7IXBku+uJULv8Oxu7qNlpu0tVrL
↪OLM7duBXOsxH9DsaGWw75rZoO7rtxDtoJVE6ojTSuwUWuzrrdNw7G5+xPBYPAzq4FRk74HW1OvPaXDuWm3o6siqhOy1MMjqLVt
↪"
      },
      "fc": 12000,
      "fs": 12000,
      "channels": 1,
      "preamble": 0,
      "msgID": "7720595f-3512-4f12-8168-6b55da613766",
      "perf": "INFORM",
      "recipient": "MyCustomInterface",
      "sender": "phy",
      "inReplyTo": "28db3bd4-ad14-4d86-b4a0-a2d8ebb3cb65"
    }
  },
  "relay": false
}
```

Again it can be observed from the *inReplyTo* attribute that the above JSON message is in reply to the JSON message corresponding to the *RecordBasebandSignalReq*. Also note that this JSON message contains the data recorded as a base64 encoded string and the *clazz* attribute indicates that the actual values are floats. The developer/user can utilize this information and decode the recoded data from this *data* attribute to a usable format. The other *attributes* that are added to the JSON message in response can be found in the UnetStack online documentation for the *RxbasebandSignalNtf* message.

JSON message without base64 encoding to transmit a signal:

In general, arrays can be encoded with or without base64 encoding. The base64 encoding is default, as it compresses the message, but it's use is optional. In this section, we show an example without base64 encoding.

A signal can be transmitted using *TxbasebandSignalReq* message in one of the two ways:

- Without base64 encoding
- With base64 encoding

JSON message without base64 encoded signal:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.bb.TxBasebandSignalReq",
    "data": {
      "signal": [
        1,
        1,
        1
      ],
      "preamble": 1,
      "msgID": "24078a7f-0054-42c9-a578-99eb7f4c0c07",
      "perf": "REQUEST",
      "recipient": "phy",
      "sender": "MyCustomInterface"
    }
  },
  "relay": true
}
```

Equivalent JSON message with base64 encoded signal:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.bb.TxBasebandSignalReq",
    "data": {
      "signal": {
        "clazz": "[F",
        "data": "P4AAAD+AAAA/gAAA"
      },
      "preamble": 1,
      "msgID": "7774ae54-cb34-44c5-b5d0-4de12e2afcba",
      "perf": "REQUEST",
      "recipient": "phy",
      "sender": "MyCustomInterface"
    }
  }
}
```

A *TxFrameNtf* is sent in response by UnetStack, the equivalent JSON message of which is as shown below:

```
{
  "action": "send",
  "message": {
    "clazz": "org.arl.unet.phy.TxFrameNtf",
    "data": {
      "txTime": 6903128000,
      "type": 0,
      "msgID": "586fb281-8891-4308-8130-74563a8a7365",
      "perf": "INFORM",
      "recipient": "MyCustomInterface",
      "sender": "phy",
      "inReplyTo": "24078a7f-0054-42c9-a578-99eb7f4c0c07"
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "relay": false
}
```

The above response is shown when the signal is transmitted without the base64 encoding of the signal. The reader can compare the *msgID* and *inReplyTo* attributes of the corresponding message.

1.14 Frequently Asked Questions

1.14.1 Groovy syntax

Q Why does *add oneShotBehavior { … }* or *addOneShotBehavior { … }* (or other behaviors) not work even though I have *GroovyExtensions* enabled?

A These two synaxes supported by *GroovyExtensions* have been removed since fjåge v1.2. Instead use the new Groovy syntax *add new OneShotBehavior({ … })*. The similarity with the Java syntax avoids confusions caused due to previous Groovy syntax.

Q I just declared a variable in the shell using *def x = …*, but I get a *No such property* error when I try accessing it! Why?

A Variables declared with types or using *def* are available during execution of the command, but not exported to the variable binding of the shell. To declare a new variable in the binding, it should be declared without a type definition (e.g. *x = …*).

Q What is the difference between *import* and *export*?

A In a Groovy script, *import* is used in the same sense as Java or Groovy, to import a package or class. The imports are only active during the execution of the script. *export* is used to add an import to the shell, so that import is in force in the shell even after the script has terminated. At the shell prompt, *import* and *export* can be used interchangeably (with a slightly different syntax – see *help export* for more information).

1.14.2 Logging

Q How do I temporarily enable debug logging for fjåge applications without writing my own *logging.properties*?

A Debug logging (log level *ALL*) can be enabled by simply passing a *-debug* flag on the command line to *GroovyBoot*. To enable debug logging for only certain loggers, you can use a flag of the form *-debug:loggername*. Startup scripts (such as *fjage.sh*) pass all arguments to *GroovyBoot*, allowing this flag to be simply included on the command line while starting the application. An alternative solution is to use the command *logLevel* at the shell prompt to control the log level of a specific logger. For more information, try *help logLevel*.

1.14.3 Precompiled scripts

Q Why does my precompiled script not work correctly?

A Precompiled scripts should be derived from the *org.arl.fjage.shell.BaseGroovyScript* base class. To do this, ensure that you have the *@groovy.transform.BaseScript org.arl.fjage.shell.BaseGroovyScript fGroovyScript* annotation in the script.

CHAPTER 2

Useful Links

- fjåge GitHub home
- fjåge API documentation
- fjåge issue tracking