PlanetSim Tutorial

Behaviors in deep

© Marc Sànchez Artigas

Octubre 2006

```java
package planet.symphony.behaviours;

import planet.commonapi.Node;
import planet.commonapi.RouteMessage;
import planet.commonapi.behaviours.Behaviour;

/**
 * This behaviour drops whatever message.
 * @author Marc Sanchez
 */
public class DropBehaviour implements Behaviour {
```

1 Behaviours Specification

In order to provide a greatest degree of reusability, ***PlanetSim*** provides a mechanism to organize the actions taken at node level. This mechanism is based on the notion of *behaviour*. Strictly speaking, a behaviour is a class that let p2p nodes perform an action in response of an incoming message. For the developer's viewpoint, a *behaviour* is a piece of code that encapsulates an action that must be performed by the node when a suitable message arrives. By a suitable message we mean a message whose performative, that is, the type and the mode of the message matches the *behaviour* descriptor. The *behaviour* descriptor is an expression used to specify when a behaviour must be executed. In its most primary form, a behaviour descriptor can be a pair of literals corresponding to the type and a mode of a message. Nonetheless, *behaviour* descriptors can be more complex and accept several wildcards as we will see later in this chapter.

The core idea behind the use of behaviours is let the ***PlanetSim*** programmer hand-code different actions and use them as interchangeable pieces like a *Lego* artefact. For example, by binding a particular behaviour with a message, and later, swapping it by a new one without modifying the node's source code all the time. This let, via a configuration file, add and remove behaviours, that is, what a node must do without recompiling it again. Briefly, this configuration file has a new line for every behaviour entry. Every behaviour entry specifies the java class that encapsulates the behaviour and its descriptor that specifies when the behaviour must be executed. We will return to this topic in section 1.2.

1.1 Runtime Execution

Until now, we have introduced the notion of behaviour and its advantages but we don't have explained what happens at runtime when the simulator uses behaviours to model the p2p nodes involved in it.

Basically, a singleton object called behaviour's pool is loaded into the simulator. The behaviour's pool has the instances of the behaviours (an instance of each one for the whole simulator) and acts as a proxy executing the corresponding ones on the nodes that have new messages. In fact, the implementation of the behaviour's pool is not fixed and a programmer can customize a new one for its own interests. For that purpose, the simulator includes several interfaces (*BehavioursFactory, BehavioursPool, ...)* to let developers customize the runtime behaviour classes. Nevertheless, the way to do it is out of the scope of this tutorial.

Upon the behaviour's pool falls the core task of the behaviour's infrastructure, we will explain on follows a little example to see the whole process. By now, we consider that a structured p2p overlay like *Chord* wants to replicate the contents stored under a key when a *REPLICATE* message arrives. Until now, *PlanetSim* users would probably would make an implementation of such operation by modifying the *dispatcher* method inside the node. However, with this new approach in mind, a programmer would probably implement this new task in a new behaviour, called for instance, *ReplicateBehaviour*. Furthermore, we imagine that the latter one is what the

programmer decided to do. In that case, once the programmer had finished the implementation of the *ReplicateBehaviour,* we would edit the configuration file and would include a new behaviour entry specifying that when a *REPLICATE* message arrives the *ReplicateBehaviour* must be executed. Thereafter, the programmer would run a new simulation.

The simulation proceeds as follows. At the start up, the simulator instantiates the behaviour's pool and loads the behaviours included in the configuration file. Then, the behaviour's pool is ready to begin invoking behaviours. This is occurs by intercepting the incoming messages and checking them against the *behaviour* descriptors. An interesting feature of the current behaviour's pool implementation is that a single message can match more than one *behaviour* descriptor. For that reason, the behaviour's pool keeps a stack of behaviours for every possible message at node level. These stacks have the behaviour instances ordered from more specific to more generic. In fact, if the p2p protocol tends to perform common tasks for every new message arrival multiple behaviour invocations will incur frequently.

The behaviour's pool invokes a behaviour by passing a couple of arguments. These are the original *RouteMessage* and the reference to the node to whom the *RouteMessage* was addressed. This reference allows updating the node's internal state to reflect the last network transaction. Once the behaviour execution finishes, the *behaviour*'s pool returns the control to the node or either spawns a new behaviour depending on whether the stack of behaviours is over or not.

In the context of our example, when the behaviour's pool intercepts a *REPLICATE* message will dispatch the *ReplicateBehaviour* and finally, it will yield the control to the node.

1.2 Behaviours descriptors

A behaviour descriptor is the result of the union of several fields that attend to distinct reasons. These fields are described next:

- ***Type** of the message*: the type of message is used to represent a communicative act between to endpoints. A communicative act might involve a unique message, a query and response and even a complex interaction protocol. The message embedded in a communicative act should have in common the type.
- ***Mode** of the message*: the mode of message stems from the fact that any communicative act requires to identify in which state of the communication we are.
- ***Probability***: this property let *behaviour*'s pool add runtime uncertainty to the behaviour execution.
- ***Scope***: the scope property refers to the message's addressee. It can take three distinct literals. These are:
    - ***LOCAL***: tells the *behaviour*'s pool that the behaviour will execute if only if the message is for this node.

  - ***REMOTE***: tells the *behaviour*'s pool that the behaviour will execute if only if the message's recipient is a remote node actually distinct from this one.
  - ***ALWAYS***: tells the *behaviour*'s pool to ignore who is the message's addressee.
- ***Role***: tells the *behaviour*'s pool that the behaviour execution depends on the node's role. From a protocol's viewpoint, a node is ***GOOD*** when follows the protocol and ***BAD*** when exploits it for its own purposes. The ***NEUTRAL*** literal let the *behaviour*'s pool ignore the role of the *behaviour*. Note that this property provides a natural way to test an overlay protocol at node level. Developers can write a pair of disjoint set of behaviours, one for ***GOOD*** peers and the other for the ***BAD*** ones and analyse what happens when afraction of the nodes is faulty

The ***type*** and the ***mode*** properties are very tight to the kind of messages required by the overlay. They can take literal values like *DATA or REQUEST,* respectively. Hereinafter, we will treat the type and the mode properties as a pair: <*DATA*, *REQUEST*>. Nonetheless, they also can take a pair of powerful wildcards: the '***?***'(***Complementary wildcard***) and the '*******'(***Universal wildcard***). The universal wildcard let a behaviour execute despite the *type*, the *mode* or both the *type* and *mode* of the incoming message. The complementary wildcard let behaviour execute if there is not a most specific combination of the type and the mode for the incoming message. For example, imagine that it already exists a behaviour for the pair <*DATA*, *REQUEST*> and the overlay uses 3 distinct message modes: *REQUEST*, *REPLY* and *REFRESH*. Then, to assign the pair <*DATA*, ?> to a behaviour gives it the chance to execute for the incoming messages with performatives: <*DATA*, *REPLY*> and <*DATA*, *REFRESH*>.

As mentioned before the *behaviour*'s pool maintains a stack of behaviours ordered from more specific to more generic. The precedence is listed on following with the pairs ordered from the most specific to the most generic:

TYPE	MODE	
Literal	*Literal*	*+ spec.*
Literal	*?*	
Literal	*	
?	*Literal*	
*	*Literal*	
?	*	
*	*?*	
*	*	*- spec.*

On following, we have included a piece of a configuration file with the list of specified behaviours:

```
RoutingBehaviour    ?       *       1.0     REMOTE      NEUTRAL
JoinBehaviour       JOIN    REQUEST 1.0     LOCAL       NEUTRAL
Databehaviour       DATA    *       1.0     ALWAYS      NEUTRAL
```

1.3 Building a node

In this section, we will discuss what we must do to design and write a behaviour-based node implementation. The first step is telling the *behaviour*'s pool that our node requires behaviour's service. A good technique for doing so consists of calling the *GenericFactory* interface to get a reference of the *behaviour*'s pool. We recommend doing it at node's start up, that is, when *PlanetSim* calls the nodes constructor method. The following fragments of code have been extracted from the ***Trivialp2p*** overlay included on ***PlanetSim*** 3.0.

```
/** The behaviours pool to be used. */
private BehavioursPool behPool;

public TrivialNode() throws InitializationException {
      super();
      alive = true;
      behPool = GenericFactory.buildBehavioursPool();
```

Once we have a reference to the *behaviour*'s pool, the next step is using it to force the *behaviour*'s pool to filter the *RouteMessage* arrivals. A good practice that we recommend consists in writing a dispatcher method that communicates with the *behaviour*'s pool synchronously :

```
private void dispatchMessagesWithBehaviours() {
      while(hasMoreMessages()) {
            RouteMessage msg = nextMessage();
            try {
                  behPool.onMessage(msg, this);
                  GenericFactory.freeMessage(msg);
            } catch(NoSuchBehaviourException e) {
                  throw new Error("There is not a behaviour for
this type of message");
            } catch(NoBehaviourDispatchedException d) {
                  throw new Error("The behaviour's pool has not
dispatched any behaviour for this message");
            }
      }
      invokeByStepToAllAplications();
```

The ***dispatchMessagesWithBehaviours*** method searches the incoming queue of messages and for each one invokes the behaviour's pool via the ***onMessage*** method. Briefly, the ***onMessage*** method demands a pair of arguments: the incoming *RouteMessage* and the node's reference. Next, the *behaviour*'s pool looks over the type and the mode of the RouteMessage and determines which behaviours have to spawn. Then, it invokes them in order passing the *RouteMessage* and the node's reference to them. Note that the ***onMessage*** method throws a couple of exceptions:

- ***NoSuchBehaviourException***: this exception is thrown when a RouteMessage performative don't match any *behaviour* descriptor.
- ***NoBehaviourDispatchedException***: this exception is launched by the behaviour's pool when it does not dispatch any behaviour. This occurs

when the behaviour stack contains random behaviours, that is, behaviours with an execution probability inferior to one.

Finally, it solely remains to explain another two methods that are essential to let the *behaviour*'s pool properly work. These methods inherited from the ***Node*** interface are the following:

- **public boolean playsGoodRole():** this *callback* method indeed tells the behaviour's simulator about the role that plays the node, that is, the GOOD or BAD attributes.
- **public boolean isLocalMessage(RouteMessage msg)**: this callback method let the behaviour's pool determine when an incoming message goes to the input node or not, that is, the *LOCAL* or *REMOTE* attributes.

1.4 Writing Behaviours

The developer who wants to implement node-specific tasks should define one or more *Behaviour* classes and add them to the *behaviour*'s pool proxy. The *Behaviour* interface has a method, the ***onMessage*** method, which carries out the business logic of the behaviour. The ***onMessage*** method receives a pair of arguments, that is, the incoming *RouteMessage* and the reference to the invoker. The reference to the invoker is essential since ***behaviours*** are stateless, that is, they receive queries on the fly, process them and return the control to the behaviour's pool. Therefore, they don't store the execution contest for the entrant node. Consequently, the node is the responsible of maintaining itself its internal state, e.g., the routing tables etc. So by using the node's reference the behaviours can refresh the node's state all the time.

Because of the non preemptive model chosen for the behaviour's pool implementation, node programmers must avoid to use endless loops an even to perform long operations within the ***onMessage*** methods. Remember that when some behaviour's ***onMethod*** is running no other behaviour can go until the end of the method.

Following this idiom, we have included a behaviour example on following:

```
public class DataBehaviour implements Behaviour {
    /* Internal atributes. */
    private TrivialNode trivialNode = null;
    /**
     * @return The behaviour's pseudonym.
     * @see planet.commonapi.behaviours.Behaviour#getName()
     */
    public String getName() { return "DataBehaviour"; }

    public void onMessage(RouteMessage msg, Node node) {
      trivialNode = (TrivialNode)node;
      // new copy; the 'msg' is free at the end of behaviour
      trivialNode.dispatchDataMessage(
            trivialNode.buildMessage(msg),
            TrivialNode.REQUEST,
            TrivialNode.REFRESH
      );
    }
}
```

The ***DataBehaviour*** receives the incoming *RouteMessage* and after casting the node reference to the concrete node instance, that is, to the *TrivialNode*, it calls the ***dispatchDataMessage*** procedure. This procedure forwards the message to the application layer or retransmits the message again depending on whether the node is the manager or not of the requested data.

A good practice when we write a behaviour is to create a copy of the incoming message before retransmit it. This avoids that a more than one node simultaneously alters its contents through the *message*'s pool. In particular, the sentence ***GenericFactory.freeMessage(msg)*** let PlanetSim reuse ***RouteMessage*** instances to speed up simulations reducing the number of garbage collector irruptions at runtime.

Another clear example of behaviour is the ***DropBehaviour***. This behaviour is quiet simple so it does nothing; it is a black hole behaviour making disappear messages. This behaviour is a good tool to measure the robustness of an overlay when a group of bad nodes drop messages.

```
public class DropBehaviour implements Behaviour {
    public void onMessage(RouteMessage msg, Node node) { }
    public String getName() { return "DropBehaviour"; }
}
```

1.5 Configuration file

Finally, in this chapter we will discuss the set up properties related to the behaviour runtime execution. We will do it explaining a real configuration file with all of its properties. There it goes:

```
##########################################################################
# BEHAVIOURS GENERAL PROPERTIES
##########################################################################
```

- This class let ***PlanetSim*** create instances of the particular behaviour's pool implementation class etc. Is a constitutes a factory method design pattern:
    - `BEHAVIOURS_FACTORY=planet.generic.commonapi.behaviours.BehavioursFactoryImpl`
- This property tells to the ***behaviour***'s factory which is the ***behaviour***'s pool implementation chosen for this simulation:
    - `BEHAVIOURS_POOL=planet.generic.commonapi.behaviours.BehavioursPoolImpl`
- The ***BEHAVIOURS_ROLESELECTOR*** property let developers specify the class used to select role of the peers that form the overlay network

- **BEHAVIOURS_ROLESELECTOR**=planet.generic.commonapi.behaviours.BehavioursRoleSelectorImpl

- The ***BEHAVIOURS_INVOKER*** property provides a mechanism to let the behaviours pool's invoke a behaviour from the stack.

  - **BEHAVIOURS_INVOKER**=planet.generic.commonapi.behaviours.BehavioursInvokerImpl

- The ***behaviour***'s ***filter*** class is a helper class that executes before any behaviour and is used to filter dummy messages:

  - **BEHAVIOURS_FILTER**=planet.generic.commonapi.behaviours.BehaviorsIdleFilter

- This helper class encapsulates the pair ***<type, mode>*** of *behaviour descriptors*:

  - **BEHAVIOURS_PATTERN**=planet.generic.commonapi.behaviours.BehaviorsPatternImpl

- The behaviour class that parses this properties:

  - **BEHAVIOURS_PROPERTIES**=planet.generic.commonapi.behaviours.BehaviorsPropertiesImpl

- The default number of message types used in the current overlay. This property tells the *behaviour's* pool the expected number of distinct message types:

  - **BEHAVIOURS_NUMBEROFTYPES**=1

- The default number of message modes used in the current overlay. This property tells the *behaviour's* pool the expected number of distinct message modes:

  - **BEHAVIOURS_NUMBEROFMODES**=2

```
#########################################################################
# BEHAVIOURS SPECIFIC PROPERTIES
#########################################################################
```

- Specifies the fraction of faulty nodes that must be within the overlay network. This value is used by the ***behaviour***'s ***role selector*** class to select the faulty nodes in the overlay. Note that this value is a percentage so logically a legal value lays in the range [0..100]:

  - **BEHAVIOURS_PROPERTIES_FAULTY_NODES**=0

- This properties tells the ***behaviour***'s ***role selector*** how must distribute the *faulty* peers along the overlay: uniformly (***UNIFORM***) or forming a chain of consecutive malicious peers: (***CHAIN***):

  - **BEHAVIOURS_PROPERTIES_MALICIOUS_DISTRIBUTION**=CHAIN

- This property outputs ***debug*** information when it is set to true:
    - **BEHAVIOURS_PROPERTIES_DEBUG**=true
- Finally, we have the behaviour entries identified by an ***URI*** (***Universal Resource Identifier***) followed by a number in ascending order:

```
# COLUMN NAMES:
# URI =  BEHAVIOUR_CLASS, TYPE, MODE, PROBABILITY, LOCALITY, ROLE
#-----------------------------------------------------------------------
BEHAVIOURS_PROPERTIES_INSTANCE_1=planet.trivialp2p.behaviours.DataBeh
aviour, DATA, *, 1.0, ALWAYS, NEUTRAL
```