GPars Guide To Remoting

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

Introduction . . . 2
  Remote Serialization . . . 2
Dataflows . . . 4
  DataflowVariable . . . 4
  DataflowBroadcast . . . 5
  DataflowQueue . . . 6
Actors . . . 7
  Remote Actor Names . . . 7
Agents . . . 9

Concepts like *Actors*, *Dataflows* and *Agents* are not restricted just to a single VM. They provide an abstraction layer for concurrent programming that allows us to separate logic from low level synchronization code. These concepts can be easly extended to multiple nodes in a network.

The following guide describes **Remoting** in **GPars**.

Remoting for **GPars** was a *Google Summer of Code 2014* project.

Introduction

To use *Actors*, *Dataflows* or *Agent* remotely, a new remote proxy object was introduced with the *Remote* prefix.

The proxy object usually has an identical interface to it’s local counterpart. This allows us to use it in place of local counterpart. Under the covers, a proxy object just sends messages over the wire to an original instance.

To transport messages across the network, the Netty library was used.

To create a proxy-object, the instance serialization mechanism was used (more in **remote-serialization** below).

The general approach to using remotes is as follows (details below):

At *host A*:

1. Create remoting context and start a server to handle incoming requests.
2. Publish an instance under a specified *name*

At *host B*:

1. Create remoting context
2. Ask for an instance with specified *name* from *hostA:port*. A promise object is returned.
3. Get a proxy object from the promise.

ℹ At this point, a new connection is created for each request

---

Remote Serialization

The following mechanism was used to create proxy objects:

object ←(serialization)→ *handle* ---- [network] ---- *handle* ←(serialization)→ *proxy-object*

One of the main advantages of this mechanism is that sending proxy-object references back is deserialized back to the original instance.

As all messages are seralized before sending over a wire, they must implement the *Serializable* interface.

This is a consequence of using build-in **Java8 serialization mechanism and *Netty**

ObjectDecoder/ObjectEncoder. On the other hand, it gives us the flexibility to send any custom object as a message to an **Actor** or to use **DataflowVariable**(s) of any type.

Dataflows

In order to use remoting for *Dataflows*, a context (*RemoteDataflows* class) has to be created. Within this context, *Dataflows* can be published and retrieved from remote hosts.

A Sample

```
def remoteDataflows = RemoteDataflows.create()
```

In all subsections we assume that a context has already been created as shown above.

After creating a context, if you want to allow other hosts to retrieve published *Dataflows*, you need to start a server. You need to provide an address and port to listen on (say, like: *localhost*:11222, or 10.0.0.123:11333).

Start A Sample Server

```
remoteDataflows.startServer HOST PORT
```

To stop the server, we have a *stopServer()* method. Note that both start and stop methods are asynchronous, and they don't block; the server is started/stopped in background.

Multiple execution of these methods or executing them in wrong order will result in an exception.

To only retrieve instances from remote hosts starting a server is not necessary.

DataflowVariable

The **DataflowVariable** is a core part of *Dataflows* subsystem that gains remoting abilities. Other structures(?) and subsystems depend on it.

Publishing a variable within context is done simply by:

Publishing a Context

```
def variable = new DataflowVariable()
remoteDataflows.publish variable "my-first-variable"
```

This registers the variable under a given name, so when a request for a variable with name *my-first-variable* arrives, the variable can be sent to the remote host.

It's important to remember that publishing another variable under the same name, will override the provious one and subsequent requests will send the newly published one.

Variable retrieval is done by:

Variable Retrieval

```
def remoteVariablePromise = remoteDataflows.getVariable HOST, PORT, "my-first-variable"
def remoteVariable = remoteVariablePromise.get()
```

The *getVariable* method is non-blocking and returns a promise object that will eventually hold a proxy object to that variable. This proxy has the same interface as a **DataflowVariable** and can be used seemlessly as a regular variable.

To explore a full example see: *groovyx.gpars.samples.remote.dataflow.variable*

DataflowBroadcast

It's possible to subscribe to a **DataflowBroadcast** on a remote host. To do this, we had to have published it first (assuming the context already exists):

A DataflowBroadcast Sample

```
def stream = new DataflowBroadcast()
remoteDataflows.publish stream "my-first-broadcast"
```

Then on other host it can be retrieved:

A Retrieval Sample

```
def readChannelPromise = remoteDataflows.getReadChannel HOST, PORT, "my-first-broadcast"
def readChannel = readChannelPromise.get()
```

The proxy object has the same interface as a **ReadChannel** and can be used in same fashion as a **ReadChannel** of a regular **DataflowBroadcast**.

To explore a full example, please see: *groovyx.gpars.samples.remote.dataflow.broadcast*

DataflowQueue

The **DataflowQueue** feature received similar functionality, and is published like this :

A Publish Sample

```
def queue = new DataflowQueue()
remoteDataflows.publish queue, "my-first-queue"
```

and in similar way, we can retrieved it on the remote host:

Retrieval from Remote Sources

```
def queuePromise = remoteDataflows.getQueue HOST, PORT, "my-first-queue"
def queue = queuePromise.get()
```

New items can be pushed into the queue of the remote proxy. Such elements are sent over a wire to the original instance and pushed into it.

Retrieval commands send a request for an element to the original instance.

Conceptually, the remote proxy is an interface - it just sends requests to an original instance.

To explore a full example see:

groovyx.gpars.samples.remote.dataflow.queue or *groovyx.gpars.samples.remote.dataflow.queuebalancer*

Actors

The `Remote Actors` subsystem is designed in similar way.

To start a *RemoteActors* class, a context has to be created. Then within this context, an *Actors* instance can be published or retrieved from a remote host.

Remote Creation

```
def remoteActors = RemoteActors.create()
```

Publishing :

```
def actor = ...
remoteActors.publish actor, "actor-name"
```

Retrieval :

```
def actorPromise = remoteActors.get HOST, PORT, "actor-name"
def remoteActor = actorPromise.get()
```

It’s possible to join a remote **Actor**, but this will block until the original **Actor** ends its work. Sending replies and the *sendAndWait* method are supported as well.

One can send any object as a message to an **Actor**, but keep in mind it has to be **Serializable**.

See example: *groovyx.gpars.samples.remote.actor*

Remote Actor Names

A *RemoteActors* class context may be identified by a name. To create one with a name use:

Create A Named Context

```
def remoteActors = RemoteActors.create "test-group-1"
```

Actors published within this context may be accessed by providing a special **Actor** URL.

For example: publishing an **actor** under the name of [blue]"actor" within this context makes it accessible under the URL "test-group-1/actor".

A Sample

```
def actor = remoteActors.get "test-group-1/actor"
```

The host and port of an instance holding this actor is determined automatically.

Invoking the *get* method will send a broadcast query to *255.255.255.255* with a search for an actor within a context with that specific name. A matching instance responds to that query with necessary information like host and port.

Allowed actor and context names

```
As the URL contains "/" (backslash) as a separator between context and actor name, we
cannot use backslashes in an actor's name, but a context name can contain any UTF
characters.
```

Agents

A `Remote Agents` system is designed in similar fashion.

To begin, a *RemoteAgents* class context has to be created. Within this context, *Agents* can be published or retrieved from remote hosts.

A Sample

```
def remoteAgents = RemoteAgents.create()
```

Publishing :

```
def agent = ...
remoteAgents.publish agent, "agent-name"
```

Retrieval :

```
def agentPromise = remoteAgents.get HOST, PORT, "agent-name"
def remoteAgent = agentPromise.get()
```

There are two ways to execute closures used to update the state of a remote *Agent* instance:

- *remote* - closure is serialized and sent to original instance and executed in that context
- *local* - current state is retrieved and closure is executed where the update originated, then updated value is sent to original instance. Concurrent changes to *Agent* wait until this process ends.

By default, remote *Agents* uses a *remote* execution policy. We can change it if necessary :

Changing Policy

```
def agentPromise = remoteAgents.get HOST, PORT, "agent"
def remoteAgent =  agentPromise.get()
remoteAgent.executionPolicy = AgentClosureExecutionPolicy.LOCAL
```