Distribute Plugin – User Guide

Michael Simon

June 18, 2016

Contents

List of Figures

Chapter 1

Setup

1.1 Running Multiple Connected Renew Processes

First, the distribute registry has to be started. For listening on the default port 1099 execute

```
> sh startRegistry.sh
```

Alternatively, to define a specific port execute

```
> java -Dde.renew.distribute.createRegistry=<port> -jar loader.jar \
    keepalive on
```

This starts a new Renew process which provides a distribute registry to other Renew processes and is kept alive, but does nothing else.[1] The given port is used to listen for connection attempts from the other Renew processes. If `=<port>` is omitted, the default port 1099 is used.[2] This Renew process must be kept running while other processes are connected. A `&` may be appended to the line to run the process in the background.

Additional Renew processes may be started locally or remotely and connected to the distribute registry by executing

```
> java -Dde.renew.distribute.registryHost=<hostname>:<port> -jar \
    loader.jar gui
```

`:<port>` can be omitted, if the default port 1099 is used and the whole `registryHost` argument can be omitted, if additionally the distribute registry runs on the same host.

Make sure that the Renew processes can access each other through TCP. Any firewalls should be configured accordingly. This can be quite difficult, e.g. the default firewall configuration for active VPN connections of the Macintosh Operating System prevents connections even to the local host. The inability of a Renew process to connect to the distribute registry may result in an exceptionally long waiting time at the startup before the connection attempt times out.

Some common errors in the RMI interaction of the Renew processes can be fixed by adding the argument `-Djava.rmi.server.hostname=<address>` before the `-jar` argument, where `<address>` is the network address by which the other Renew processes can reach the local host.

[1] See [2, Subsection 2.4.1] for details on starting Renew.

[2] A RMI registry is created internally to connect the additional Renew processes to the distribute registry. 1099 is the default port for a RMI registry.

1.2 (De)activation

When the DISTRIBUTE plugin is loaded, it automatically sets the active formalism to *Distribute Java Compiler*. For all practical purposes, the DISTRIBUTE plugin can be disabled for the simulation by selecting another formalism.

Chapter 2

Modeling

The main functionality of RENEW is provided by its Java Reference Net formalism. It allows for Petri Nets with Java as an inscription language. Net instances may also contain other net instances and interact with them through synchronous channels. The Java Reference Net formalism is described in [2, Chapter 3] and the theoretical and implementation details are given in [1] (in German). The implementation is also described in [3, Chapter 3].

The DISTRIBUTE Plugin provides another formalism which extends the Java Reference Net formalism conservatively. For the initial exchange of net instance references it provides simple Java classes and interfaces.

2.1 Exchanging Remote Net Instances

The DISTRIBUTE Plugin provides the `de.renew.distribute.DistributeNetInstance` interface to reference a specific net instance in a remote RENEW process. An initial exchange of net instance references can be facilitated by a `de.renew.distribute.DistributeRegistry` object. RENEW processes that were set up according to Section 1.1 share the same distribute registry and thus can exchange references to net instances among themselves. In each RENEW process the same distribute registry is returned by the static method `de.renew.distribute.DistributePlugin.getRegistry()`.[1] Remote net instances can be registered at the distribute registry with the method `registerNetInstance(Serializable, DistributeNetInstance)` and retrieved with the method `getNetInstance(Serializable)`. In both cases the first argument is an arbitrary key.

The functionality described above is not provided by the formalism, but by normal Java methods. Because they have side effects (changing the registered net instances), they should only be called inside a net from within *action* inscriptions. The receiver example net in Figure 2.1 registers itself at a distribute registry. The sender example net in Figure 2.2 retrieves this remote net reference.

2.2 Send Channels

The Java Reference Net formalism provides synchronous channels to synchronously fire multiple transitions. Parameters can be defined to exchange data between the transitions. A synchronous channel is called by a *downlink* inscription that specifies a net instance, a channel name and the parameters, e.g. `net:ch(a,b,c)`. In the specified net instance a corresponding *uplink* transition is sought. It specifies the channel name and the parameters at that transitions side, e.g. `:ch(x,y,z)`. The parameters of both sides get unified and a binding of the variables of both transitions is searched and determines the mode they are fired in. Synchronous channels

[1] A single distribute registry is made remotely accessible by RMI.

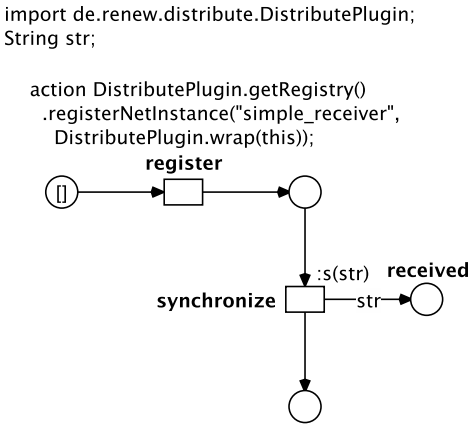

Figure 2.1: A receiver net with a channel uplink.

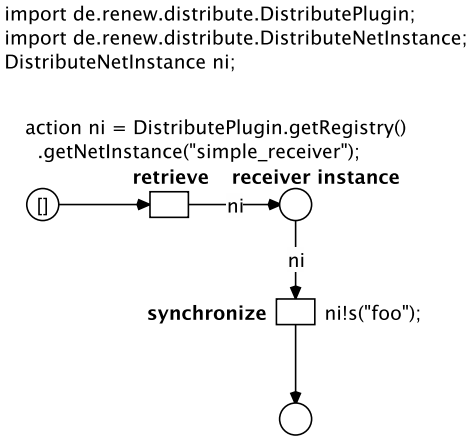

Figure 2.2: A sender net with a send channel downlink.

allow for data flow in any direction.[2] They are introduced in [2, Section 3.7].

Normal downlinks can only be called on net instances in the same RENEW process. The DISTRIBUTE plugin adds another downlink inscription that can be called on remote net instance references. It connects to the same uplinks. However, all parameters must be fully evaluated at the transition with the downlink before the connection is made. If this is not possible, no binding will be found and the transitions will not fire. This means that the data can only flow to the called uplink transition. The new downlink inscription uses a *!* in the place of the *:* of the original downlink inscription.[3] Parameters of send channels are send by RMI and thus need to be of special types. They need to be either serializable, or RMI remote objects.

The **synchronize** transition of the sender in Figure 2.2 calls the **synchronize** transition of the receiver in Figure 2.1. The string *foo* is fully evaluated by the sender, before it is send. In the receiver it is unified with *str* variable and put out to the **received** place.

2.3 Return Expressions

Even though the send channels provided by the DISTRIBUTE plugin do not offer the same flexibility in data flow as the synchronous channels, they can be used for simple bidirectional data flow. Downlinks and uplinks can be extended by a return expression analogous to a return value in a Java function. By using a tuple multiple values can be returned. A return expression can only be added to send channel downlinks (written with *!* instead of *:*). On the other side any uplink may have a return expression (uplinks are not separated). The expression is appended to send channel downlinks with an -> arrow to indicate that the data flows *out* of the channel call. It is appended to uplinks with an <- arrow to indicate that the data flows *back* to the origin of the channel call.

The uplink transitions return expression has to be fully evaluated at that transition, because the result is send back to the downlink transition. This is analogous to how the parameters have to be fully evaluated at the downlink transition, but the result is send in the opposite direction.

The receiver example net in Figure 2.3 extends the one in Figure 2.1 by sending back the string *bar* in addition to storing the received string. The sender example net in Figure 2.4 extends the one in Figure 2.2 by storing the returned string in addition to sending the string *foo*.

[2] The same search algorithm is used to search for bindings of a single transition and for bindings of multiple transitions. A transition with the uplink is merely added to the search after unifying the channel parameters at both sides.

[3] Based on popular convention, the ! indicates that the data flows *into* the call.

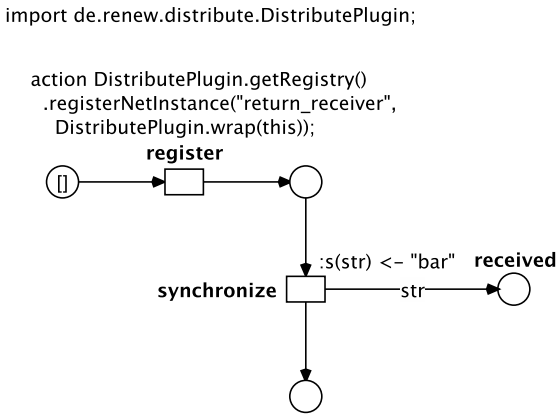

Figure 2.3: A receiver net with a channel uplink with a return expression.

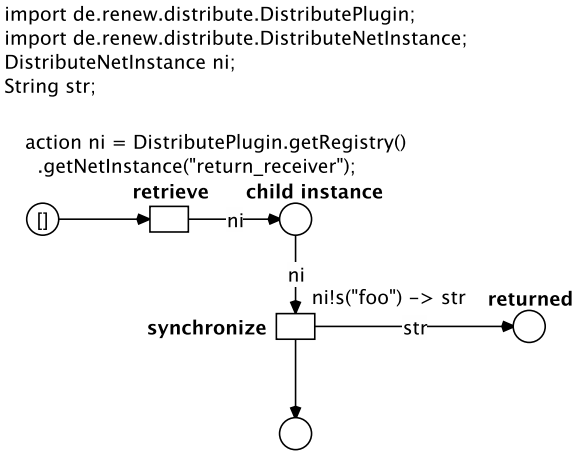

Figure 2.4: A sender net with a send channel downlink with a return expression.

Bibliography

[1] Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.

[2] Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. *Renew – User Guide (Release 2.4.3)*. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, September 2015.

[3] Michael Simon. Concept and implementation of distributed simulations in RENEW. Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, March 2014.