Extending the Real-Time Toolkit with Plugins

Revision History

Revision 2.0.0	21 Jun 2010 Initial plugin manual	ps
Revision 2.1.0	8 Oct 2010 More detail	ps

Abstract

This document is an introduction to using and creating plugins for the RTT. A plugin extends the RTT with new data types or new functions, which components can then use. Example plugins are scripting engines, XML readers and writers or remote communication libraries.

Typekits are plugins that are explained in the Typekit manual.

Table of Contents

1. Loading Existing Plugins

1.1. Paths and (not) default loading

Existing plugins are fairly easy to use. You only need to tell where they are located.

The most important thing to remember is to set the RTT_COMPONENT_PATH environment variable which is a list of colon or semi-colon seprated directories where to look for plugins. For example (in Linux):

```
export RTT_COMPONENT_PATH=.:/opt/rtt/lib:/usr/local/lib/orocos
```

Which tells plugins are in . (the current directory), /opt/rtt/lib and /usr/local/lib/orocos

When an Orocos application starts, it will look in these directories and load any RTT component or plugin found there and load it. The plugin directories are expected to be structured as such:

```
/usr/local/lib/orocos/       : component libraries
                /plugins : services and other plugins
                /types   : typekits and transports
```

The RTT will know that the plugins directory contains normal plugins, these are services useable by components if requested. The types directory contains typekits and transports, which are immediately process-wide available to every component.

The plugin::PluginLoader assumes that every directory listed in the RTT_COMPONENT_PATH has this structure, meaning, having a plugins or a types subdirectory. If neither exists, the directory is just ignored. The listed paths are *not recursed* !

Note

As a special provision for multi-target environments like Linux, if a directory plugins/*OROCOS_TARGET* or types/*OROCOS_TARGET* exists, this directory is searched too.

Only some plugins and components should be readily available in every application. In case you want to make them optional, you may place them in a subdirectory of one of your RTT_COMPONENT_PATH directories. For example:

```
/usr/local/lib/orocos/myrobot        : component libraries for 'myrobot'
                /myrobot/plugins : services and other plugins for 'myrobot'
                /myrobot/types   : typekits and transports for data used by 'myrobot'.
```

Since the /usr/local/lib/orocos directory will not be searched recursively, the myrobot directory is initially ignored. You can instruct the deployer to look for components by using the import statement, *in combination with the RTT_COMPONENT_PATH variable*. For example:

```
Deployer [S]> import("myrobot")
```

Which will look in each directory of the RTT_COMPONENT_PATH for a subdirectory "myrobot" in which it will locate and load components, plugins and typekits.

1.2. Using plugins in the Deployer

You can use the deployer application in order to list the found services, types and typekits using the **.services**, **.types** and **.typekits** commands:

```
Deployer [S]> .types
Available data types:  ConnPolicy FlowStatus PropertyBag SendHandle SendStatus TaskContext
 array bool char double float int rt_string string uint void
Deployer [S]> .services
Available Services:  scripting marshalling
Deployer [S]>
```

The marshalling and scripting plugins are delivered by default by the RTT.

Finally, you can load a new service into a component by using the **.provide** command:

```
Deployer [S]> .provide scripting
Trying to locate service 'scripting'...
2.250 [ Info   ][Logger] Loading Service scripting in TaskContext Deployer
Service 'scripting' loaded in Deployer
Deployer [S]>
```

This allows you to extend a running component with new services. Normally, you will already indicate that you want to use this service in your component when writing it in C++, as shown in the next section.

2. Using Plugins in C++

In order to use the functions that a plugin offers to a component, you need to 1. load the service, 2. call the functions ! Both steps can actually be combined, since the RTT loads services as you need them. For example, in your component's configureHook():

```
#include <rtt/scripting/Scripting.hpp>

// ... in your component:
  bool configureHook() {
    boost::shared_ptr<Scripting> scripting = this->getProvider<RTT::Scripting>("scripting"); // 1.
    if (scripting) {
      bool result = scripting->loadStateMachines("statemachines.osd"); // 2.
      //...
    }
  }
```

What happens here is this: In step 1., we use the TaskContext function getProvider to get an object to access the "scripting" service. What getProvider does is check if "scripting" is provided by this component, if not, it looks it up in the list of available plugins (see the **.services** command in the taskbrowser) and if found, loads it as a new service in the current component.

The Scripting class offers us access to the loaded service by exposing all its methods as Orocos RTT methods. This allows you to use the service synchronously ('call') or asynchronously ('send'), which is explained in the Component Builder's Manual.

In step 2., we use this Scripting object to access the loadStateMachines function of the "scripting" service (see the **scripting** command in the taskbrowser).

In case you only wanted to check if the "scripting" service was offered by this component, use:

```
bool result = this->provides()->hasService("scripting")
```

in your component code.

3. Writing your own plugins

There are very little requirements to make a library an Orocos plugin. The interface of an Orocos plugin is shown in the file rtt/plugin/Plugin.hpp. Take a look at that file for the full documentation of the Plugin API.

The most important function is the

```
extern "C"
bool loadRTTPlugin(RTT::TaskContext* c)
```

function. Note that it *must* have the **extern "C"** decoration in front of it, otherwise, this function will not be found. The safest way to achieve this correctly is to include the rtt/plugin/Plugin.hpp header in your plugin code.

The plugin::PluginLoader looks for this function in order to determine if the shared library is an RTT plugin. If found, it is called with a c = null argument to give the plugin a chance to load into the application. If it returns false or throws an exception, the library is unloaded again.

Typekits will use this 'null argument' call to register their types with the RTT type system, and refuse to do any more work if the user attempts to load the plugin in a specific component (non-null c).

Services will typically do very little during the 'null argument' call and do a lot more when a component (non-null c) is given. A Service will register a new instance of the service to the given component.

For both typekits and services, there are predefined macros that relieve you from writing the loadRTTPlugin function. See the rtt/plugin/ServicePlugin.hpp file and the rtt/types/TypekitPlugin.hpp file for the respective macros.

3.1. Creating a Service Plugin

A Service Plugin can be created by writing a subclass of Service:

```
#include <rtt/Service.hpp>
#include <rtt/plugin/ServicePlugin.hpp>
#include <iostream>

using namespace RTT;

/**
 * A service that offers the HelloWorld() operation.
 */
class MyService : public Service
{
public:
  MyService(RTT::TaskContext* c)
    : RTT::Service("myservice", c) {
    this->addOperation("HelloWorld", &HelloWorld, this);
  }

  void HelloWorld() {
     std::cout << "Hello World !" << std::endl;
  }
};

ORO_SERVICE_NAMED_PLUGIN(MyService, "myservice")
```

See the Component Builder's Manual for detailed explanations about creating and using services.

When you compile the above file into a library and put it in the plugins directory of your RTT_COMPONENT_PATH, it will be found by the RTT and you'll have access to a new service, called "myservice" and which offers one operation: "HelloWorld".

3.2. Creating a Typekit Plugin

See the Typekit Plugin manual for adding your own data types to the RTT.