ATLAS

Introduction to Orocos

Best Practices in integrating complex robotic systems

Gianni Borghesan
KU Leuven
Febraury 2020

Outline

1 Orocos Toolchain Overview

2 Deployment

3 OCL and other libraries

4 Basic Ros integration

Orocos Toolchain Definition:

The Toolchain allows setup, distribution and the building of real-time software components. It is sometimes referred to as 'middleware' because it sits between the application and the Operating System. It takes care of the real-time communication and execution of software components.

From the **Component Builders Manual**, [7]. Most of the figures are from the same source.

WHY! I

As a middle-ware

- When to you do not what to code *ad-hoc* solutions for communication composition, *etc.*
- When you do not what to reinvent-the-wheel about deployment, communication, scheduling...
- When you want to take advantage of some out-of-the-shelf component *e.g.* Ethercat Master.
- At some degree, it promotes re-usability of your own components.

WHY! II

As a RT middle-ware with ROS 1 integration

- It has realtime support — you can safely run control loops on it.
- Can take advantage of many ROS tools — expecially HMI and data visualization.

WHEN!

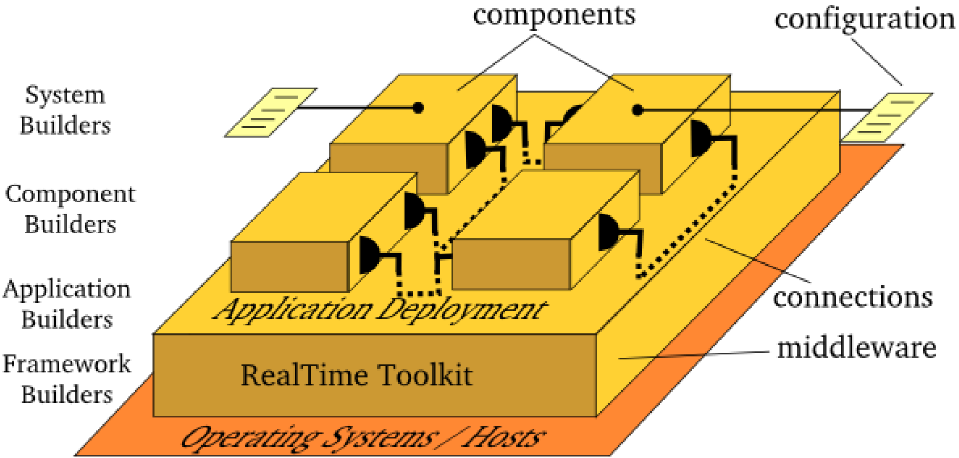
When

- When you need to design a complex system — especially you have to integrate several steps or behaviours.

When not

- When you need certification (too complex, too many dependencies) and developing a rigid application.

The Orocos structure



Developer Layers

The goal of the framework is to divide the development between

Application Developers

Take care of building an application by **deployment**, **configuration**, and **coordination**.

Components Developers

Creates components that are general purpose, *e.g.* hardware interface. Documentation and testing.

Developer Layers

The goal of the framework is to divide the development between

Application Developers

Take care of building an application by **deployment**, **configuration**, and **coordination**.

Components Developers

Creates components that are general purpose, *e.g.* hardware interface. Documentation and testing.

In practice, it is difficult to make components that are reusable.

Orocos Toolchain Tools I

The toolchain provides:

Tools to support the build system (IN ROS)

- Catkin (ROS).
- create-pkg: generate skeleton of an Orocos package
- code generation — generate new messages for transport

Orocos Toolchain Tools II

Real-Time Toolkit (RTT)

- Abstraction to access OS timer/scheduler, defining a number of activities:
  - Periodic activities.
  - Aperiodic activities, port-triggered.
  - Aperiodic activities, file descriptor-triggered.
  - Master/slave activities.
- Data and Data Flow abstraction

Orocos Toolchain Tools III

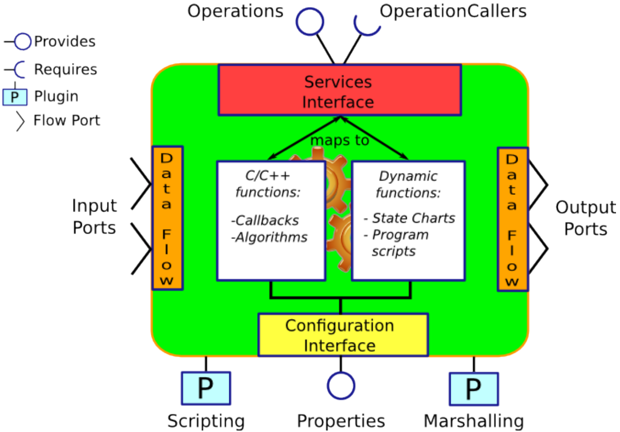
The Orocos Component Library (OCL)

- Deployer
- Task Browser (native and lua)
- Lua Component — also for deploying
- Logging
- Data Reporting (both cvs and binary)
- the base-class of all the components, the **Task Context**

The Orocos Component

The context exposes:

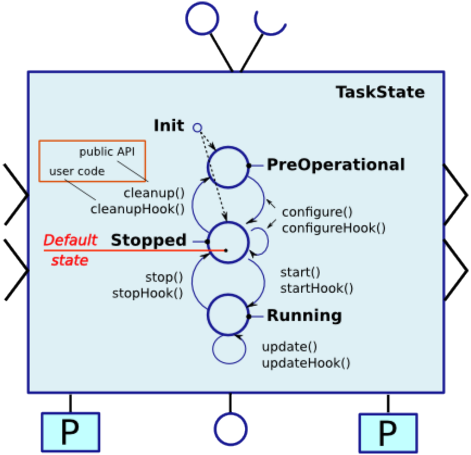
- Ports
- Operations
- Properties
- Services/Plugins



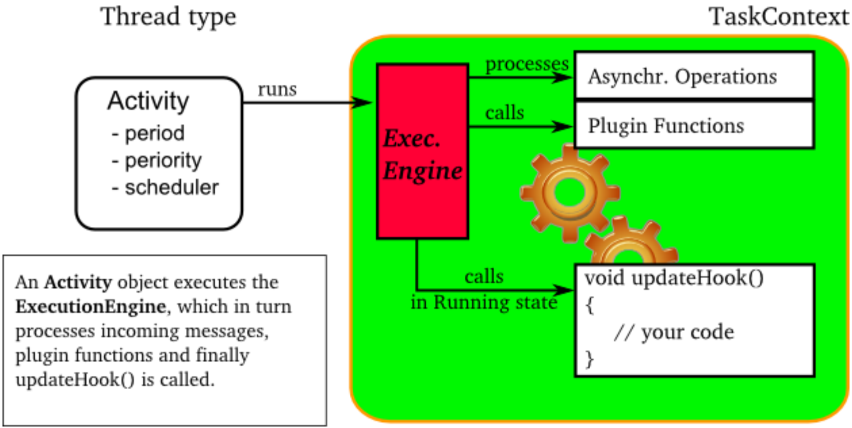
Component States — The Life-Cycle State-Machine

The life cycle state machine:

- Each transition calls a function.
- Self transitions are possible.
- More states (error, clean-up, . . .)



Component Execution



Activities

An activity registers a trigger in the scheduler, and creates a thread that execute the execution engine.

- **Periodic activities**
- **Aperiodic activities** — trigger on events, such as writing to a port or file descriptor.
- **Master/Slave activities**

Activities are parametrized in function of:

- Real Time or not real time,
- Priority, and
- Frequency (zero for aperiodic).

Parenthesis — Real Time

What a real time system is?

A (hard-)real time system, is a system that must provide a response to an external stimulus in a fixed amount of time.

- Over-run can be dangerous - robot becomes unstable.
- RT is about predictability, not speed!
- RT is associated with **Scheduling**.

How to get an RTOS-Orocos for desktop? I

Xenomai

- Is based on a micro-kernel (a RT kernel that runs also a linux kernel)— dificcult to maintain
- nice how-to xenomai-Orocos https://rtt-lwr.readthedocs.io/en/latest/index.html
- you need to recompile Orocos from sources, with correct flags . . .

How to get an RTOS-Orocos for desktop? II

PREEMPT_RT Patch

- is a patch for a normal kernel to make it (fully) pre-emptive
- No need to recompile, https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup
- you **do not** need to recompile Orocos from sources. . .
- PREEMPT_RT Patch **is becoming mainline** [1].

Component Interconnection — Ports

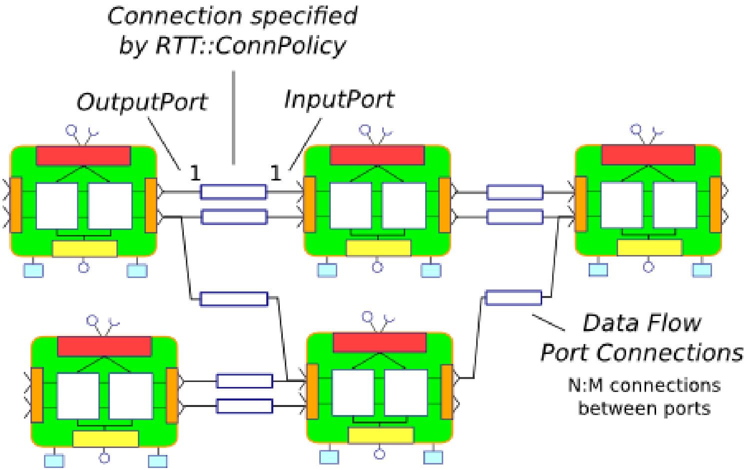
- Ports allow for data transfer.
- Data are stored in a buffer, that can hold more than the last value.
- There is a buffer for each connection.
- `Read` returns the last value, and if the buffer holds more than one data, consume it.
- `Read` returns also a status: `NoData`, `OldData` and `NewData`.
- input ports can be added as an `eventPort` — every time a new data is available for such port, the `updateHook` (**or** a callback) will be executed by the execution engine.

Component Interconnection — Connection Policy

Ports are connected by means of buffers or data. By default connections are locked (mutex).

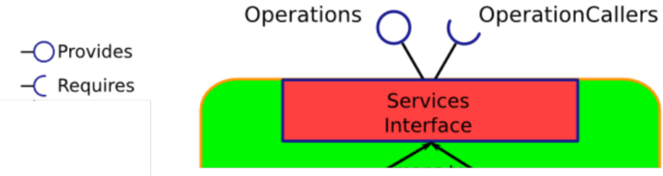
- **DATA** (default) only last data is maintained.
- **BUFFER** drops newer sample when full.
- **CIRCULAR_BUFFER** drops older samples when full.

Component Interconnection — Ports



KU LEUVEN ATLAS

Component Interconnection — Operations



- Operations allows to run functions.
- Operation can be executed by the owner **or** the caller: it influences at which priority the call is executed.
- **Components need to be peers**.

Component Interconnection — Operations

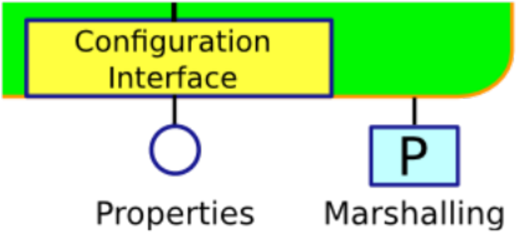
Execution Type	Requires locks in your component?	Executed at priority of	Examples
Client Thread	**Yes**. For any data shared between the ClientThread-tagged operation and updateHook() or other operations.	Caller thread	Stateless algorithms that get all data through parameters, Operations of real-time components that are not real-time
Own Thread	**No**. Every OwnThread-tagged operation and updateHook() is executed in the thread of the component.	Component thread.	Operations that do a lot of setup work in the component, Operations which are called from several places at the same time. *e.g.*:`moveToPosition(pos, time)`, `setParameter("name", value)`,..

Calling Vs Sending Operations

Operation can be called by other **peer** components

- directly: it blocks until the result is done,
- or send/collect: the operation is started in the given activity, and result must be collected via polling.

Properties



- Properties are data exposed in the interface.
- They can be changed by other components.
- They can be loaded and stored via the **Marshalling service**.

Services (and Plugins)

They are useful to group a number of operation under a name.
Services are

- provided: when added to the interface
- required: before called by other components

Services can also have properties.
They are very useful to write **plugins** [5]!
Plugins are factory-like libraries that extends the interface of a component (*e.g.* marshalling, soem hardware), normally with services.

Component Examples – Header

```
#include <rtt/RTT.hpp>
class MyComp : public RTT::TaskContext{
public:
        MyComp(std::string const& name);
        //bool configureHook();//no configure phase needed here.
        bool startHook();
        void updateHook();
        void stopHook();
        //operation
        void setCounter(int)
private:
        //ports
        RTT::OutputPort<int> outport_counter;
        //properties
        int init_value;
        //other variables
        int current_index;
};
```

Component Examples – CPP

```
#include "my_comp-component.hpp"
using namespace RTT;
MyComp::MyComp(std::string const& name) : TaskContext(name),
init_value(0){
        addPort("outport_counter",outport_counter);
        addProperty("init_value",init_value);
        addOperation("setCounter",&MyComp::setCounter,this,OwnThread)
}
bool MyComp::startHook(){
        if (init_value<0)
                return false;
        current_index=init_value;
        return true;//return false will stop the transition
}
void MyComp::updateHook(){
        current_index++
        outport_counter.write(current_index);
}
void MyComp::stopHook(){
        current_index=init_value;
}
void MyComp::setCounter(int new_value){
        current_index=new_value;
}
```

Event+Callback Example

```
//hpp
class base_interface : public RTT::TaskContext{
        ...
        std::vector<RTT::InputPort< std_msgs::Int32> > XYZ_Motor_encoder_inport;
}
//cpp
base_interface::base_interface(std::string const& name) : TaskContext(name,PreOperational)
, XYZ_Motor_encoder_inport(3)
...
{...
for (int i=0;i<3;i++){
        addEventPort("XYZ_Motor_encoder_"+ std::to_string(i+1),XYZ_Motor_encoder_inport[i],
                boost::bind(&base_interface::encoder_callback, this,_1,i))
                .doc("Encoder Value of Motor "+ std::to_string(i+1));

void  base_interface::encoder_callback(RTT::base::PortInterface* portInterface, int i){
        newDataArrived[i]=true;
        ...
}

void base_interface::updateHook(){
        if (!(newDataArrived[0]&&newDataArrived[1]&&newDataArrived[2]))
                return;
        ...
}
```

Event+Callback Example

```
//hpp
class base_interface : public RTT::TaskContext{
        ...
        std::vector<RTT::InputPort< std_msgs::Int32> > XYZ_Motor_encoder_inport;
}
//cpp
base_interface::base_interface(std::string const& name) : TaskContext(name,PreOperational)
, XYZ_Motor_encoder_inport(3)
...
{...
for (int i=0;i<3;i++){
        addEventPort("XYZ_Motor_encoder_"+ std::to_string(i+1),XYZ_Motor_encoder_inport[i],
                boost::bind(&base_interface::encoder_callback, this,_1,i))


void  base_inte
        newData
        ...
}

void base_interf
        if (!(n

        ...
}
```

Ports have not copy constructors!

Is possible to initialise **within the class inizialiser** list the vector, but no resize.
If you want to dynamically resize → pointers.

KU LEUVEN ATLAS

Calling an operation Example

Blocking call

```
TaskContext* a_task_ptr = getPeer("ATask");
OperationCaller<void(void)> my_reset_meth
= a_task_ptr->getOperation("reset"); // void reset(bool)
// Call 'reset' of a_task, blocking
bool ok=reset_meth();
```

Send and Collect

```
SendHandle<void(void)> handle = reset_meth.send();
bool ok;
if (handle.collectIfDone() == SendSuccess ){
 handle = reset_meth.send(ok);
 SendStatus ss = handle.collect();
 if (ss != SendSuccess) {
   cout << "Execution of reset failed." << endl;
 }
  cout << "Return value "<< ok << endl;
}
```

Logging

```
Logger::In in(this->getName());
RTT::log(Error)<<" failed, ...."<<RTT::endlog();
```

Timestamped Logging is saved automatically in `orocos.log` file. depending of minimum log level, is displayed in console.
The first line notifies the logger which component orignates the log.
there are six log levels, the most used are:

- Error
- Warning
- Info
- Debug

Logging breaks real-time

Last remarks on re-usability

- Components are a abstractions of threads - **not functions**
- Re-usability can be achieved at high granularity
- If you need to execute components in a chain, you can impose explicit scheduling using
  - Event Ports
  - Master slave-activities, or the *fbsched* component
- For big components, try to make libraries for the computational part, and glue-code to Orocos.

Last remarks on real-time code

you must write the code in such a way:

- Non-RT code is executed in the boot-strap phase (configureHook)
  - Memory allocation (including outputports, see `setDataSample`)
  - Variable-time algorithms
- RT is in the updateHook.
  - Do not log, nor cout.
  - Deterministic time algorithm

Outline

1 Orocos Toolchain Overview

2 Deployment

3 OCL and other libraries

4 Basic Ros integration

KU LEUVEN ATLAS

How to write a program

To build an application, we need to make a number of steps.
This can be done:

- directly writing a c++ main (`ORO_main`)
- using an executable that deploys the **Deployer component**, and interpret a deployment script (lua or native).

How to write a program

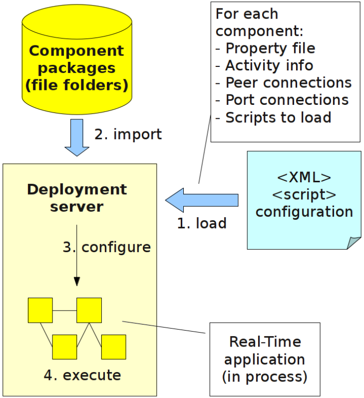
To build an application, we need to make a number of steps.
This can be done:

- directly writing a c++ main (`ORO_main`)
- using an executable that deploys the **Deployer component**, and interpret a deployment script (lua or native).

Compiling components

Components are normal child classes; they are compiled as **shared libraries** that can be dynamically linked (`ORO_main`) or dynamically loaded (deployer).

Deployer component activities



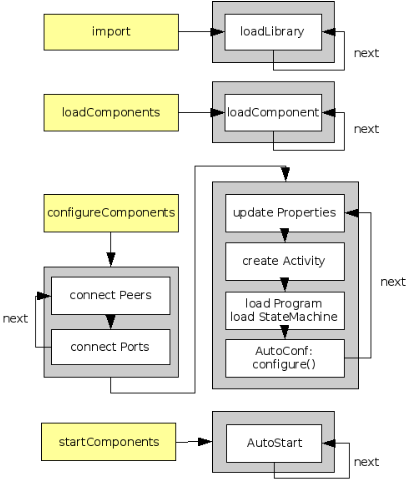
From the "*The Deployment Component*" webpage, [6].

Deployment Flow

The Flow for a simple activity is:

- import packages,
- load components,
- configure:
  - set properties
  - create activities
  - connect peers
  - call configure()
- start

From the "*The Deployment Component*" webpage, [6].



Deploying Components — OPS

Orocos has native scripting, the *Orocos Program Scripts* language. It can be used for coding a deployer file:

```
deployer -s myscript.ops
```

This command:

- opens a task browser (OPS shell),
- runs script, and
- return to an **interactive shell**.

Deploying Components — OPS

```
//this is a comment !
import("my_package")
loadComponent("producer", "Producer")
loadComponent("consumer", "Consumer")

setActivity("producer", 98, 0.01, ORO_SCHED_RT)
setActivity("consumer", 97, 0, ORO_SCHED_RT)

// connect ports
var ConnPolicy cp // non-buffered connection
connect("producer.outport", "consumer.inport",cp)
// configure the components
producer:configure()
consumer:configure()
// start components
consumer:start()
producer:start()
```

Deployer Commands

within the task-browser you can set a number of commands (in addition to the one in the previous slide).

Note: tab completion and backward research is available!

- `help`: print-out the help of the current component (including the Deployer)
- `ls`: list the component interface
- `.types`, `.services`: print known types and services
- cd: enter in a (peer) component
- `leave` and `enter`: change view of the task browser
- `var 'type' 'name'`: creates a variable
- call/send operations: like start, stop, and customs

Deploying Components — Lua

Xml and native scripting are now used less in favour of **Lua** scripting language [4].

A script is launched with the command rttlua:

```
rttlua -i myscript.lua
```

This command:

- opens a task browser (lua shell),
- loads two components, lua (OCL::LuaComponent) and Deployer, then
- the lua component executes the script (that uses the Deployer, being peers), and
- leaves the shell open to interactive mode (`-i` flag); otherwise, the process will close upon script competition.

Deploying Components — Lua

```
require "rttlib"
rttlib.color=true
tc=rtt.getTC()
depl = tc:getPeer("Deployer")
-- load package
depl:import("my_package")
-- create components
depl:loadComponent("producer", "Producer")
depl:loadComponent("consumer", "Consumer")
--... and get references to them
producer = depl:getPeer("producer")
consumer = depl:getPeer("consumer")
-- configure the components
producer:configure()
consumer:configure()
-- connect ports
depl:connect("producer.outport", "consumer.inport", rtt.Variable('ConnPolicy'))
depl:setActivity("producer", 98, 0.01, rtt.globals.ORO_SCHED_RT)
depl:setActivity("consumer", 97, 0, rtt.globals.ORO_SCHED_RT)
-- start components
consumer:start()
producer:start()
```

Outline

1 Orocos Toolchain Overview

2 Deployment

3 OCL and other libraries

4 Basic Ros integration

The Orocos component library

Orocos comes with a number of pre-built blocks

- ConsoleReporting
- **FileReporting**
- HMIConsoleOutput
- HelloWorld
- **LuaComponent**
- LuaTLSFComponent
- **NetcdfReporting**
- TcpReporting
- TimerComponent
- logging::Appender
- logging::GenerationalFileAppender
- logging::LoggingService
- logging::OstreamAppender
- logging::RollingFileAppender

The Reporter

OCL provides two types of reporter to write data to a file:

- FileReporting
- NetcdfReporting

They have the same interface:

- reportComponent(string const& Component)
- reportData(string const& Component, string const& Data)
- reportPort(string const& Component, string const& Port)

they can be used for periodic report (by setting a period) or in snapshot mode.

The Lua component

- It allows to write simple components directly in Lua.
- Operations in lua are not supported — but is possible to inherit the lua component and add in c++,
- Used a lot for loading the state machine based on lua (rFSM, [3])
- Please refer to the Lua CookBook [2].

The Lua component

- It allows to write simple components directly in Lua.
- Operations in lua are not supported — but is possible to inherit the lua component and add in c++,
- Used a lot for loading the state machine based on lua (rFSM, [3])
- Please refer to the Lua CookBook [2].

Lua is good for configuration and coordination, not for computation (easily wraps C and C++)!

The Lua component

To load a lua component (from lua deployment):

```
depl:loadComponent("event_echo", "OCL::LuaComponent")
event_echo = depl:getPeer("event_echo")
event_echo:exec_file(custom_folder .. "event_echo.lua")
```

The same can be done with OPS

KU LEUVEN ATLAS

The Lua component

The event_echo component (event_echo.lua)

```
require("rttlib")
tc=rtt.getTC()
local inport
local outport

function configureHook()
 inport = rtt.InputPort("std_msgs.String", "event_in")    -- global variable!
 outport = rtt.OutputPort("string", "event_out")    -- global variable!
 tc:addEventPort(inport)
 tc:addPort(outport)
 return true
end

function updateHook()
 local fs, ev_in = inport:read()
 outport:write(ev_in.data)
end

function cleanupHook()
 rttlib.tc_cleanup()
end
```

Components for various hardware

- Kuka iwa LWR (ROB)
- Kuka lwr 3 (ROB)
- Universal Robot https://github.com/gborghesan/URDriver
- SOEM master
  https://github.com/orocos/rtt_soem/tree/master/soem_master
  (Beckoff, Maxpos, Robotique hand, . . .)
- . . .

Outline

1 Orocos Toolchain Overview

2 Deployment

3 OCL and other libraries

4 Basic Ros integration

Packages (github.com/orocos/rtt_ros_integration)

- **rtt_ros** ROS package import plugin as well as wrapper scripts and launchfiles for using Orocos with ROS.
- **rtt_rosclock** Realtime-Safe NTP clock measurement and ROS Time structure construction as well as a simulation-clock-based periodic RTT activity.
- **rtt_rosnode** Plugin for ROS node instantiation inside an Orocos program.
- **rtt_rosparam** Plugin for synchronizing ROS parameters with Orocos component properties.
- **rtt_roscomm** ROS message typekit generation and Orocos plugin for publishing and subscribing to ROS topics as well as calling and responding to ROS services.
- **rtt_rosdeployment** An RTT service which advertises common DeploymentComponent operations as ROS services.
- **rtt_rospack** Plugin for locating ROS resources.
- **rtt_tf** RTT-Plugin which uses tf to allow RTT components to lookup and publish transforms.
- **rtt_actionlib** RTT-Enabled actionlib action server for providing actions from ROS-integrated RTT components.
- **rtt_dynamic_reconfigure** A service plugin that implements a dynamic_reconfigure server to update properties dynamically during runtime.
- **rtt_ros_msgs** ROS .msg and .srv types for use with these plugins.
- **rtt_ros_integration** Catkin metapackage for this repository.

Use of typekits from ROS messages

All the standard types of ros are already ready. To use them import the `rtt_*` version of the package:

```
Deployer [S]> import("rtt_std_msgs")
= true
Deployer [S]> .types
Available data types: std_msgs.Bool std_msgs.Bool[] std_msgs.Byte std_msgs.ByteMultiArray
   std_msgs.ByteMultiArray[] std_msgs.Byte[] .......
```

In case of custom messages you can generate custom message with command `create_rtt _msgs` of the `rtt_roscomm` package:

```
rosrun rtt_roscomm create_rtt_msgs my_custom_msgs
```

This generate also the headers, and they can used inside your components (also lua - see lua component example).

Use of typekits from ROS messages

All the standard types of ros are already ready. To use them import the `rtt_*` version of the package:

```
Deployer [S]> import("rtt_std_msgs")
= true
Deployer [S]> .types
Available data types:  std_msgs.Bool std_msgs.Bool[] std_msgs.Byte std_msgs.ByteMultiArray
std_msgs.ByteMultiArray[] std_msgs.Byte[] .......
```

In case of custom messages you can generate custom message with command `create_rtt _msgs` of the `rtt_roscomm` package:

```
rosrun rtt_roscomm create_rtt_msgs my_custom_msgs
```

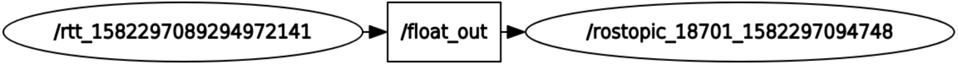
This generate also the headers, and they can used inside your components (also lua - see lua component example).

Set streams to/from ROS topics

```
import ("rtt_std_msgs")
import("rtt_rosnode")//makes orocos a ros node. will complain if roscore is not running
import("rtt_roscomm")//contains the topic services
stream("hello.my_ros_port", ros.topic("my_topic"))
```

Steps:

- load the typekit,
- load the rosnode package, it registers the orocos process as ros node (will fail if no `roscore` is running),
- connect the port.



import using ROS path, and ROS-Find equivalent

```
Deployer [S]> import("rtt_ros")
= true
Deployer [S]> ros.import ("hello1")
= true
```

rtt_ros allows to import taking advantage of ROS environmental path variables

```
Deployer [S]> import ("rtt_rospack")
= true
Deployer [S]> ros.find("hello1")
= /home/gborghesan/ws_kinetic/ws_my_projects/src/hello1
```

rtt_rospack allows to get directories - very useful for loading config files.

Other integration features

Launch files (https://github.com/jhu-lcsr/rtt_ros_examples)

```
<launch>
<arg name="LUA" default="true"/>

<include if="$(arg LUA)" file="$(find rtt_ros)/launch/ldeployer.launch">
<arg name="DEPLOYER_ARGS" value="-g -s $(find rtt_ros_integration_example)/example.lua"/>
<arg name="LOG_LEVEL" value="debug"/>
<arg name="DEBUG" value="false"/>
</include>

<include unless="$(arg LUA)" file="$(find rtt_ros)/launch/deployer.launch">
<arg name="DEPLOYER_ARGS" value="-s $(find rtt_ros_integration_example)/example.ops"/>
<arg name="LOG_LEVEL" value="debug"/>
<arg name="DEBUG" value="false"/>
</include>

<node name="update_monitor" pkg="rtt_ros_integration_example" type="update_monitor.py" output="screen"/>
</launch>
```

Other integration features

Services and ActionLib

- Examples here: https://github.com/jhu-lcsr/rtt_ros_examples
- These are very intrusive to use, especially ActionLib,
- they can be substituted with other mechanisms.
- if you use it, is better to have a component that acts as RPC server.

Bibliography I

[1] Lukas Bulwahn. *Real-Time Linux Continues Its Way to Mainline Development and Beyond*. https://www.linuxfoundation.org/blog/2018/09/real-time-linux-continues-its-way-to-mainline-development-and-beyond/. 2018.

[2] Markus Klotzbücher. *LuaCookbook*. https://www.orocos.org/wiki/orocos/toolchain/luacookbook. Lastly visited: Feb 2020.

[3] Markus Klotzbücher. *rFSM Statecharts*. https://github.com/kmarkus/rFSM. Lastly visited: Feb 2020.

Bibliography II

[4] Markus Klotzbücher, Peter Soetens, and Herman Bruyninckx. "OROCOS RTT-Lua: an Execution Environment for building Real-timeb Robotic Domain Specific Languages". In: *Proc. of the Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS*. 2010.

[5] Peter Soetens. *Extending the Real-Time Toolkit with Plugins*. https://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos-rtt-plugins.html. Lastly visited: Feb 2020. 2010.

[6] Peter Soetens. *The Deployment Component*. https://orocos.org/ocl/toolchain-2.9/xml/orocos-deployment.html. Lastly visited: Oct 2019. 2012.

Bibliography III

[7] Peter Soetens. *The Orocos Component Builder's Manual*. https://orocos.org/rtt/toolchain-2.9/xml/orocos-components-manual.pdf. Lastly visited: Oct 2019. 2014.