HAL
open science

A task scheduler for ROS

Cédric Pradalier

▶ **To cite this version:**

Cédric Pradalier. A task scheduler for ROS. [Research Report] UMI 2958 GeorgiaTech-CNRS. 2017. hal-01435823v3

HAL Id: hal-01435823

https://hal.science/hal-01435823v3

Submitted on 22 Aug 2017 (v3), last revised 29 Dec 2020 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

A task scheduler for ROS

Cédric Pradalier
UMI 2958 GT-CNRS - GeorgiaTech Lorraine
Metz, France

August 22, 2017

Contents

1 Introduction **2**

2 Concepts **2**
2.1 What is a task? . . . 2
2.2 A basic example . . . 3
2.3 Purpose of the task system . . . 4
2.4 Task parameters and Dynamic Reconfigure . . . 4
2.5 Task Environment . . . 5
2.6 Templated Parents . . . 5
2.7 The Task Server . . . 6

3 Creating a new task framework **6**
3.1 Required files . . . 6
3.2 The Environment . . . 7
3.3 The task server . . . 8
3.4 Task Idle . . . 8
3.5 An example: Task GoTo . . . 8
3.6 Implementation Constraints . . . 10

4 Using the task framework **12**
4.1 Compilation and linking . . . 12
4.2 Launch file . . . 12
4.3 Console . . . 13
4.4 Simple missions . . . 13
4.5 Background tasks . . . 14
4.6 Interruptions . . . 14

5 Integration with other frameworks **15**
5.1 Integration with Smach . . . 15
5.2 Integration with the ActionLib . . . 16
5.3 Integration with MoveBase . . . 17
5.4 Tasks and multi-agent synchronization . . . 18

6 Acknowledgements **22**

1 Introduction

Developing a complete robotic system often requires combining multiple behaviours into a complex decision grid, with elements running in sequence or in parallel, eventually interrupting each others. To solve this "age-old" problem, ROS provides two main tools:

Actionlib: a client-server architecture that provides a way to specify results to be achieved. While the server works on these results, it should report progresses and ultimately report when the task is completed.

Smach: a python API to define complex state machines. It can interact with ROS services and actions to define a complex behaviour, including nesting, interruptions and concurrence.

Combining Smach and Actionlib, one could build arbitrarily complex systems. Hence, why would another task management system be necessary?

The main argument in favour of our task scheduler is the simplicity of its use, particularly in comparison with Actionlib. As usual, simplicity is a trade-off against expressiveness. Simplicity can be sacrificed by linking our task scheduler with Actionlib and/or Smach to exploit the best of both worlds.

This task scheduling framework is the culmination of about 10 years of experience developing robotic applications in the academic context. Most applications we designed could be handled using the task scheduling framework we will present in this document.

We provide the source code for this project at: https://github.com/cedricpradalier/ros_task_manager.

2 Concepts

2.1 What is a task?

In our framework a task is seen as a behaviour that can be started, run for some amount of time, possibly infinite, and then terminates either on completion or on interruption. A task requires some context such as system variables, ROS topics, and services, but it can also store its own internal variables and state.

Programmatically, a task is a C++ class that must implement an iterate function and may implement an initialize and terminate function. We distinguish two concepts:

Task Definition: the main description of a task such as its name, help text and parameter description, as well as some status information. It acts as a factory to instantiate the task for specific parameters and is used to communicate with the scheduler clients. For a given task name, a single task definition is possible.

Task Instance: a task instance is created from the task definition when the task needs to be run. This is the class that needs to implement the initialize, iterate, and terminate function as well as store any relevant variable. Multiple task instances with the same name and possibly different parameters may be launched from a single task definition.

During the life of a task instance, the task will report different status, defined in the ROS TaskStatus.msg file:

TASK_NEWBORN: The task has been created from the task definition.

TASK_CONFIGURED: The task has been configured by the task definition. This status is inherited from previous version and not used anymore.

TASK_INITIALISED: The task instance initialisation function has been called and returned successfully.

TASK_RUNNING: The task instance `iterate` function returns this when the task needs to keep running. This is only relevant for tasks whose loop is managed by the task scheduler.

TASK_COMPLETED: The task instance `iterate` function returns this when the task objective has been met. The `iterate` function will not be called anymore.

TASK_TERMINATED: Returned by the task instance `terminate` function. In addition, this status is added as a binary mask to any error status described below so it should be tested as `status & TASK_TERMINATED` in C.

Task instance will sometime fail or be interrupted before reporting completion. The status code is then a combination of `TASK_TERMINATED` and the following code:

TASK_INTERRUPTED: The task was interrupted by the task scheduler.

TASK_FAILED: The task reported failure in the `iterate` function.

TASK_TIMEOUT: The task instance execution timeout was passed and the task was stopped by the task scheduler.

TASK_CONFIGURATION_FAILED: The configuration stage failed (unused).

TASK_INITIALISATION_FAILED: Can be returned by the initialization function when it failed.

Should one of the task function return one of the status above, it will be terminated immediately. In addition, to the task status, a task can use the `setStatusString` function to document the cause of failure.

2.2 A basic example

For an initial example, we create a simple task for reaching a desired destination (let us ignore the template parameters for now).

```
class TaskFactoryGoTo : public TaskDefinition<TaskGoToConfig, TurtleSimEnv, TaskGoTo>
{

    public:
        TaskFactoryGoTo(TaskEnvironmentPtr env) :
            Parent("GoTo","Reach a desired destination",true,env) {}
        virtual ~TaskFactoryGoTo() {};
};
```

Using this task definition, we next create a specific task instance:

```
class TaskGoTo : public TaskInstance<TaskGoToConfig, TurtleSimEnv>
{
    public:
        TaskGoTo(TaskDefinitionPtr def, TaskEnvironmentPtr env) : Parent(def,env) {}
        virtual ~TaskGoTo() {};

        virtual TaskIndicator iterate();

        virtual TaskIndicator terminate();
};
```

The implementation of this particular task just need to focus on the specifics of the behaviour, taking advantage of some parameters available in the `cfg` variable.

```
TaskIndicator TaskGoTo::iterate()
{
    const turtlesim::Pose & tpose = env->getPose();
    // distance to target
    double r = hypot(cfg.goal_y-tpose.y,cfg.goal_x-tpose.x);
    // completion condition
    if (r < cfg.dist_threshold) {
      return TaskStatus::TASK_COMPLETED;
    }
    // angle to target
    double alpha = remainder(atan2((cfg.goal_y-tpose.y),cfg.goal_x-tpose.x)-tpose.theta,2*
        M_PI);
```

```
    // Saturated proportional control law.
    if (fabs(alpha) > M_PI/6) {
        double rot = ((alpha>0)?+1:-1)*M_PI/6;
        env->publishVelocity(0,rot);
    } else {
        double vel = cfg.k_v * r;
        double rot = cfg.k_alpha*alpha;
        if (vel > cfg.max_velocity) vel = cfg.max_velocity;
        env->publishVelocity(vel, rot);
    }
   return TaskStatus::TASK_RUNNING;
}
```

Finally, we implement the terminate function to ensure the last requested velocity is always zero.

```
TaskIndicator TaskGoTo::terminate()
{
    env->publishVelocity(0,0);
   return TaskStatus::TASK_TERMINATED;
}
```

2.3 Purpose of the task system

The purpose of the task system is to facilitate the development and testing of complex applications. In such an application, the system (i.e. the task server) will have a set of pre-implemented behaviours that can be instantiated by a client application. In the simplest case, the combination of behaviours is done in sequence: go to point A, take picture, go to point B, deliver coffee and go back to charging station. We define such a sequence as a mission. The simplest missions are statically defined but branching on some conditions could also be required: deliver coffee, if water is empty, go refill, go back to charging station. As more and more conditions are required, a proper state machine management could be implemented using Smach. In an academic setting though, most demonstrations can be implemented with simple, human-readable, quasi-linear missions.

Our missions are typically implemented in python and look like the following script:

```
#!/usr/bin/python
import roslib; roslib.load_manifest('task_manager_turtlesim')
import rospy
from task_manager_lib.TaskClient import *

rospy.init_node('task_client')
tc = TaskClient("/task_server",0.2)

while True:
    tc.Wait(duration=1.)
    tc.GoTo(goal_x=1.0,goal_y=1.0)
    tc.Wait(duration=2.)
    tc.GoTo(goal_x=5.0,goal_y=5.0)
```

In this listing, it is important to notice that the tasks are called as member functions of the TaskClient, using the name defined in the TaskDefinition class. These functions are dynamically generated from the information received from the task server. Because the server also provides the parameters that the function can accept, parameters are order-independent and explicitly named in the mission. This also allows checking and enforcing type compatibility before trying to execute a task.

2.4 Task parameters and Dynamic Reconfigure

To allow introspection, the task parameters are defined using the dynamic_reconfigure[1] framework. This allows specifying each parameter with a name, a help string, a type, a default value and a range where appropriate. The dynamic_reconfigure framework defines parameters in a config file similar to the following:

```
#! /usr/bin/env python
PACKAGE='task_manager_turtlesim'
import roslib; roslib.load_manifest(PACKAGE)
```

[1] http://wiki.ros.org/dynamic_reconfigure/Tutorials

```python

from dynamic_reconfigure.parameter_generator import *
from task_manager_lib.parameter_generator import *

gen = TaskParameterGenerator()
#        Name            Type            Description                     Default    Min    Max
gen.add("goal_x",        double_t, 0,    "X coordinate of destination",  0.0)
gen.add("goal_y",        double_t, 0,    "Y coordinate of destination",  0.0)
gen.add("k_v",           double_t, 0,    "Gain for velocity control",    1.0)
gen.add("k_alpha",       double_t, 0,    "Gain for angular control",     1.0)
gen.add("max_velocity",  double_t, 0,    "Max allowed velocity",         1.0)
gen.add("dist_threshold",double_t, 0,    "Distance at which a the target is considered reached",  0.1)

exit(gen.generate(PACKAGE, PACKAGE, "TaskGoTo"))
```

Note that the ParameterGenerator class has been overloaded with the TaskParameterGenerator to make sure default parameters common to all tasks are always present in the list.

A secondary benefit of using the dynamic_reconfigure framework is that it is possible to use the reconfigure gui to check the values of the parameters while a task is running and eventually change them. This is particularly useful to tune control parameters at run-time.

In practice dynamic_reconfigure generates a Python and a C++ class containing the parameter data as class members. This class is one of the template parameters of the TaskDefinition and TaskInstance classes. It is available in a variable named `cfg` in every instance, as can be observed in the `iterate` function above.

2.5 Task Environment

In most applications, a common set of functions, variables and topics of interest will be needed by most if not all classes. These could be robot dimensions, velocity commands, sensor measurements, etc... As mentioned above, it is possible for each task to subscribe to its own topics when started and unsubscribe when terminating. To simplify the development of functions shared between tasks the task server shares a common variable called the task environment with every task. There are no constraints regarding what can be included in the environment because the parent class only provides a common mutex:

```cpp
/**
 * Empty class, to be inherited for a specific application.
 * */
class TaskEnvironment {
    public:
        // This mutex will be locked in all the task instance function
        // (initialize, iterate, terminate) for periodic tasks. However,
        // for non-periodic tasks, it is not possible to lock the
        // environment forever, so the lock is taken only for initialize and
        // terminate, but the user needs to organise his/her own locking as
        // appropriate, e.g for a reader or a writer
        // boost::shared_lock<boost::shared_mutex> guard(env_gen->environment_mutex);
        // boost::unique_lock<boost::shared_mutex> guard(env_gen->environment_mutex);
        boost::shared_mutex environment_mutex;
    public:
        TaskEnvironment() {}
        virtual ~TaskEnvironment() {}
};
```

2.6 Templated Parents

To further simplify the implementation of new classes, common functions are written into a templated class, which is later inherited for specific applications.

TaskDefinition<Config,Environment,Instance> defines a new task factory, specialized on a specific task parameter of type `Config`, a specific environment, and specific instances.

TaskInstance<Config,Environment> specializes a task instance for an application.

The Environment class is used to create a member variable `env` pointing to the shared environment. The Config class is used to create a member variable `cfg`, which is filled with the task parameters when the task is instantiated and updated as appropriate by the dynamic_reconfigure framework.

2.7 The Task Server

The task server has multiple responsibilities in this framework. First, it loads the description of all tasks (TaskDefinition objects). It then provides a service to start or stop a given task, and keeps track of the status of all tasks currently running or recently terminated. It is also responsible for instantiating the task scheduler that manages the threads in which tasks actually run. Because none of this is application-specific, most of the task server can be made generic. As a result, the `main` function for a specific task server will often be as simple as the following listing:

```
#include "task_manager_lib/TaskServerDefault.h"
#include "task_manager_turtlesim/TurtleSimEnv.h"

using namespace task_manager_lib;

class TaskServer : public TaskServerBase {
    public:
        TaskServer(TaskEnvironmentPtr _env) : TaskServerBase(_env,true) {
            start();
        }
};

int main(int argc, char *argv[])
{
    ros::init(argc,argv,"turtlesim_tasks");
    ros::NodeHandle nh("~");
    TaskEnvironmentPtr env(new TurtleSimEnv(nh,1));
    TaskServer ts(env);
    ros::spin();
    return 0;
}
```

3 Creating a new task framework

To instantiate our framework through an example, we will start with the turtlesim simulator from the ROS tutorial. We will then create a minimal set of tasks to implement part of the LOGO functionalities[2].

The complete implementation is available in the example package `task_manager_turtlesim`.

3.1 Required files

Although the implementation-specific details will be given in the next sections, it is already possible to list what is required to create a new task framework:

- Define a specialized task environment subscribing to commonly required topics and providing common tools for all classes.
- Define a new task server `main` by mostly copy-pasting the above example.
- For each task, create a `.cfg` file to declare its parameters and define the task description (inheriting from TaskDefinition) and the task instance (inheriting from TaskInstance).

Once a proper CMakeLists.txt has been created, all the tasks will be compiled as a shared library and the task server will be able to dynamically load as many of them as it may find.

[2] LOGO is an old programming language used to programmatically draw curves on screen in the early days of computers: http://en.wikipedia.org/wiki/Logo_(programming_language)

3.2 The Environment

We set the environment up by adding function relevant to the application. For instance, it is a fairly safe guess that any task working with the turtlesim environment will need to read the turtle pose (`turtlesim/Pose`) and potentially publish velocity commands (`turtlesim/Velocity`). These functions must then be part of the environment. We also add a helper function that uses the turtlesim service to set the pen colour. This could arguably be subscribed to only in the task that needs it. The TurtleSimEnv class is declared in the listing (header file) below:

```
// File: task_manager_turtlesim/TurtleSimEnv.h
#ifndef TURTLE_SIM_ENV_H
#define TURTLE_SIM_ENV_H

#include "task_manager_lib/TaskDefinition.h"
#include "turtlesim/SetPen.h"
#include "turtlesim/Velocity.h"
#include "turtlesim/Pose.h"

namespace task_manager_turtlesim {
    class TurtleSimEnv: public task_manager_lib::TaskEnvironment
    {
        protected:
            ros::Subscriber poseSub;
            ros::Publisher velPub;
            ros::ServiceClient setPenClt;

            void poseCallback(const turtlesim::Pose::ConstPtr& msg) {
                tpose = *msg;
            }

            turtlesim::Pose tpose;

        public:
            TurtleSimEnv(ros::NodeHandle & nh);
            ~TurtleSimEnv() {};

            const turtlesim::Pose & getPose() const;

            void publishVelocity(double linear, double angular);

            void setPen(bool on, unsigned int r=0xFF, unsigned int g=0xFF,
                    unsigned int b=0xFF, unsigned int width=1);
    };
};

#endif // TURTLE_SIM_ENV_H
```

Additionally, the required functions are defined in the following implementation file:

```
// File: TurtleSimEnv.cpp
#include "task_manager_turtlesim/TurtleSimEnv.h"

using namespace task_manager_turtlesim;

TurtleSimEnv::TurtleSimEnv(ros::NodeHandle & n) : task_manager_lib::TaskEnvironment(n)
{
    setPenClt = nh.serviceClient<turtlesim::SetPen>("/turtle0/set_pen");
    poseSub = nh.subscribe("/turtle0/pose",1,&TurtleSimEnv::poseCallback,this);
    velPub = nh.advertise<turtlesim::Velocity>("/turtle0/command_velocity",1);
}

const turtlesim::Pose & TurtleSimEnv::getPose() const
{
    return tpose;
}

void TurtleSim::publishVelocity(double linear, double angular) {
    turtlesim::Velocity cmd;
    cmd.linear = linear;
    cmd.angular = angular;
    velPub.publish(cmd);
}

void TurtleSimEnv::setPen(bool on, unsigned int r, unsigned int g, unsigned int b, unsigned
     int width)
{
    turtlesim::SetPen setpen;
    setpen.request.r = r;
```

```
    setpen.request.g = g;
    setpen.request.b = b;
    setpen.request.off = !on;
    setpen.request.width = width;
    if (!setPenClt.call(setpen)) {
        ROS_ERROR("Failed␣to␣call␣service␣set_pen");
    }
}
```

3.3 The task server

As mentioned earlier, the task server is mostly a copy-paste from the example task server:

```
// File: task_server.cpp
#include "task_manager_lib/TaskServerDefault.h"
#include "task_manager_turtlesim/TurtleSimEnv.h"

using namespace task_manager_lib;

class TaskServer : public TaskServerBase {
    public:
        TaskServer(TaskEnvironmentPtr _env) : TaskServerBase(_env,true) {
            start();
        }
};

int main(int argc, char *argv[])
{
    ros::init(argc,argv,"turtlesim_tasks");
    ros::NodeHandle nh("~");
    TaskEnvironmentPtr env(new TurtleSimEnv(nh,1));
    TaskServer ts(env);
    ros::spin();
    return 0;
}
```

3.4 Task Idle

When no task is required to run, the task server must ensure a well-defined behaviour of the system. To this end, the task server is instantiated with an Idle task. The default one, which is instantiated in the above example, just does nothing forever. When writing the Idle class, one has to be careful that it may be run between two tasks: the server has no way to know that another task will be required when a specific task terminates. As an example, on an industrial vehicle, it could make sense for the Idle task to engage the parking brakes, or for an underwater vehicle to start coming back to the surface if no task is required. In both cases, it would probably be safer to wait for a small but relevant duration before acting. This would give time to the mission executive to request the execution of a new task.

3.5 An example: Task GoTo

The purpose of this task is mostly to have the simulated turtle reach a given destination. In addition to the goal parameters, it will have parameters for its control law (gains, saturation) and for its completion condition (distance from the goal).

The config file, header and source for this task have been used for the examples above. We nonetheless copy them here for completeness. First the config file that will define the different task parameters.

```
#! /usr/bin/env python
PACKAGE='task_manager_turtlesim'
import roslib; roslib.load_manifest(PACKAGE)

from dynamic_reconfigure.parameter_generator import *
from task_manager_lib.parameter_generator import *

gen = TaskParameterGenerator()
#        Name            Type            Description                     Default    Min
     Max
```

```
gen.add("goal_x",        double_t, 0,    "X coordinate of destination",  0.)
gen.add("goal_y",        double_t, 0,    "Y coordinate of destination",  0.)
gen.add("k_v",           double_t, 0,    "Gain for velocity control",  1.0)
gen.add("k_alpha",       double_t, 0,    "Gain for angular control",  1.0)
gen.add("max_velocity",  double_t, 0,    "Max allowed velocity",  1.0)
gen.add("dist_threshold",double_t, 0,    "Distance at which a the target is considered reached",  0.1)

exit(gen.generate(PACKAGE, "task_manager_turtlesim", "TaskGoTo"))
```

The following headers defines the task definition and instance.

```
#ifndef TASK_GOTO_H
#define TASK_GOTO_H

#include "task_manager_lib/TaskDefinition.h"
#include "task_manager_turtlesim/TurtleSimEnv.h"
#include "task_manager_turtlesim/TaskGoToConfig.h"

using namespace task_manager_lib;

namespace task_manager_turtlesim {
    class TaskGoTo : public TaskInstance<TaskGoToConfig, TurtleSimEnv>
    {
        public:
            TaskGoTo(TaskDefinitionPtr def, TaskEnvironmentPtr env) : Parent(def,env) {}
            virtual ~TaskGoTo() {};

            virtual TaskIndicator initialise() ;

            virtual TaskIndicator iterate();

            virtual TaskIndicator terminate();
    };
    class TaskFactoryGoTo : public TaskDefinition<TaskGoToConfig, TurtleSimEnv, TaskGoTo>
    {
        public:
            TaskFactoryGoTo(TaskEnvironmentPtr env) :
                Parent("GoTo","Reach a desired destination",true,env) {}
            virtual ~TaskFactoryGoTo() {};
    };
};
#endif // TASK_GOTO_H
```

An important point to notice is the Boolean `true` in the `TaskFactoryGoTo` constructor. This declares the class as periodic and instructs the scheduler to take care of calling the iterate function at an approximately constant rate. This rate is one of the default task parameters and a default value is provided to the task server on initialization.

The implementation of the task follows. Note that it can focus on the mathematics of the task and does not need to be aware of any infrastructure such as threading, mutexes, data storage and callbacks, etc...

Another important remark is that the initialise function is unnecessary in this case and could have been omitted in this code. It is only included here for illustration.

```
#include <math.h>
#include "TaskGoTo.h"
#include "task_manager_turtlesim/TaskGoToConfig.h"
using namespace task_manager_msgs;
using namespace task_manager_lib;
using namespace task_manager_turtlesim;

TaskIndicator TaskGoTo::initialise()  {
    ROS_INFO("TaskGoTo: Going to (%.2f,%.2f)",cfg.goal_x,cfg.goal_y);
    return TaskStatus::TASK_INITIALISED;
}

TaskIndicator TaskGoTo::iterate()
{
    const turtlesim::Pose & tpose = env->getPose();
    double r = hypot(cfg.goal_y-tpose.y,cfg.goal_x-tpose.x);
    if (r < cfg.dist_threshold) {
      return TaskStatus::TASK_COMPLETED;
    }
    double alpha = remainder(atan2((cfg.goal_y-tpose.y),cfg.goal_x-tpose.x)-tpose.theta,2*M_PI);
    if (fabs(alpha) > M_PI/6) {
```

```
        double rot = ((alpha>0)?+1:-1)*M_PI/6;
        env->publishVelocity(0,rot);
    } else {
        double vel = cfg.k_v * r;
        double rot = cfg.k_alpha*alpha;
        if (vel > cfg.max_velocity) vel = cfg.max_velocity;
        env->publishVelocity(vel, rot);
    }
   return TaskStatus::TASK_RUNNING;
}

TaskIndicator TaskGoTo::terminate()
{
    env->publishVelocity(0,0);
   return TaskStatus::TASK_TERMINATED;
}

// Declare the task so that it can be dynamically loaded from the shared library.
DYNAMIC_TASK(TaskFactoryGoTo);
```

3.6 Implementation Constraints

3.6.1 Constructors

For a given task, the task instance and task definition constructor must respect the profile used in the above example and initialize their parent class properly by transferring them the environment pointer. It is not possible to add other arguments to these constructors since they will always be called by the task scheduler which knows nothing about additional arguments.

The class inheriting from TaskDefinition (task factory) is the one responsible for naming the task and specifying its help string. The name must be specified and unique for a given application. The task scheduler will output a warning when trying to register twice a task with the same name. The help string could be ignored, but it comes in handy when running a task from the command line.

3.6.2 Initialization

Variable initialization should be implemented in the `initialise` function. In the context of a task class inheriting from the templated TaskInstance, the variable `cfg` has already been affected with the values read from the task parameters. The `cfg` variable is an instance of the Config class given as a template parameter. The `env` variable points towards the task environment and could be used, for instance, to recover a global ROS NodeHandle.

For classes inheriting directly from TaskInstanceBase (not recommended), the `parseParameters` function should be overloaded to recover the task parameters and process them as required.

The `initialise` function is the right place to create ROS publishers or subscribers, and to affect initial variable values. In particular, any task intending to implement a relative motion, e.g. rotate over 360 degrees, should use this function to record the current system state.

There are no constraints regarding the duration of the `initialise` function, it could include blocking calls or return immediately. However, the task scheduler does not publish the updated task status **while** a task is initializing.

The `initialise` function should return `TaskStatus::TASK_INITIALISED` on success, and `TaskStatus::TASK_INITIALISATION_FAILED` otherwise. Using the `setStatusString` function allows setting an error status that will then be published by the task scheduler. If a task does not report itself as initialised, the `iterate` and `terminate` functions are not executed.

If a task has no need for initialisation, it should just not overload the `initialise` function and use the default one that just returns `TaskStatus::TASK_INITIALISED`

3.6.3 Iterations

The `iterate` function is, by default, the place to implement the control-loop aspect of a task as well as its termination conditions. For classes inheriting from `TaskInstance`, the `cfg` variable contains the task parameters and the `env` variable points towards the task environment.

There are two types of tasks, and they condition the way the `iterate` function should behave.

Periodic tasks are used when the `iterate` function should be called at a constant frequency (up to the OS scheduler precision). In this case, the `iterate` function should be relatively short. If the task is deemed completed, the function returns `TaskStatus::TASK_COMPLETED`, otherwise, it returns `TaskStatus::TASK_RUN`. In the former case, the `iterate` function is not called anymore and the task transitions to termination. Example of this type of task include control loops and tasks waiting for specific events to happen.

Non-periodic tasks are called once and take an undefined time to complete. They update their status as they see fit and return `TaskStatus::TASK_COMPLETED` when done. Examples of this type of tasks include path planning functions, service calls, large pre-processing tasks, etc...

The selection between one type of task and the other is made in the `TaskDefinition` constructor, with its third argument (`is_periodic`). A value of `true` marks the task as periodic.

The `iterate` function **must** be overloaded for a task class to be valid (the parent function is pure virtual).

3.6.4 Termination

The `terminate` function is called once when the task is completed, whether it completes successfully or not. Its role is to leave the system in a consistent and safe state. For instance, on some platforms, it might be a good habit to set the vehicle speed to zero (assuming it is not flying) when completing any task.

There are no constraints on the duration of the `terminate` function. If a class does not require any specific code on termination, there is no need to overload the parent function.

The `terminate` function usually returns `TaskStatus::TASK_TERMINATED`. If it needs to report a failure, it can return `TaskStatus::TASK_FAILED`, but this is unusual at this stage.

The task instance object will be destroyed once a task has terminated, which will close any publisher or subscriber it owns. This destruction occurs a few seconds after the termination class, to ensure the task status is correctly updated and published. One should not rely on the instance destruction to implement system related clean-up. As a result, the task instance destructor is empty most of the time.

3.6.5 Dynamic Loading

It is assumed that tasks will be loaded dynamically by the task server. To this end, they have to be compiled as dynamic libraries (or plugins), and they have to define a common handle that the task scheduler will use to instantiate the task definition.

The `DYNAMIC_TASK` macro creates the required code and should be included in any task intended for dynamic loading (default behaviour).

3.6.6 Dynamic reconfiguration

When a task instance is created, it launches a ROS `dynamic_reconfigure` server, even before calling the `initialise` function. Dynamic reconfigure allows the modification of the task parameters at run-time using a practical graphical user interface. Besides introspection, this function is one of the reasons why the Config files are used to define task parameters.

This use of dynamic reconfigure has two caveats:

1. First, one should not assume that the value in the `cfg` variable will stay constant over the task life. For most parameters (control gains, set-points), this is not an issue. However, some parameters are used during initialisation to allocate memory or computational structures. In this case, it is recommended to store the initial value and ignore the `cfg` value later. Even better, issue a ROS_WARN if the value gets modified to prevent giving the impression that the reconfiguration server is not working.

2. Second, task parameters have to be single values: float, booleans, strings and integer. Dynamic reconfigure does not offer a way to encode arrays in parameters.

4 Using the task framework

4.1 Compilation and linking

The task framework is dependent on the ROS application building framework. It uses `catkin` (even though traces of rosbuild can still be found) and it is governed by a standard catkin `CMakeLists.txt` file.

The task framework requires the following elements of the CMakeLists.txt:

- Dependency on package `task_manager_lib` and `task_manager_msgs`.
- Processing of the config files for each task with the `generate_dynamic_reconfigure_options`. See the `dynamic_reconfigure` package documentation for details.
- Compile and link the task server and its environment (assumed to be in `src/Environment.cpp`):

```
ADD_EXECUTABLE( task_server src/task_server.cpp src/Environment.cpp)
TARGET_LINK_LIBRARIES(task_server ${catkin_LIBRARIES} dl )
```

- Making sure that the tasks plugins are generated in a well-defined place that we can link from the package launch files:

```
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
    ${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE_SHARE_DESTINATION}/tasks)
```

- Adding the tasks as shared library with the following code:

```
ADD_LIBRARY( TaskName SHARED tasks/TaskName.cpp)
TARGET_LINK_LIBRARIES(TaskName ${catkin_LIBRARIES} dl)
ADD_DEPENDENCIES(TaskName ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

  This assumes that a source file `TaskName.cpp` implements the task classes in the `tasks` directory. The ADD_DEPENDENCIES is important to inform CMake that the task actually cares about the generation of the config classes.

4.2 Launch file

The tasks and task server having been compiled by the previously listed CMakeLists.txt, the task server can be launched with the following command. Note that the `lib_path` parameter is absolutely necessary to let the task server know where to look for task plugins. With the task

```
<?xml version="1.0" encoding="UTF-8" ?>
<launch>
    <node name="task_server" pkg="my_task_package" type="task_server"  output="screen">
        <param name="lib_path" value="$(find␣my_task_package)/tasks"/>
    </node>
</launch>
```

In case a task is removed from the list of tasks, one will need to be careful to delete the corresponding dynamic library by hand. Otherwise, the task server will keep trying to load it. This should not have any detrimental effect so long as the old class is not instantiated. If it is, it might trigger a segmentation fault if the environment or configuration definition changed since it was compiled.

4.3 Console

The console application is a small layer over the Python client to the task server, instantiated through `ipython`. It implements a command line application with the following functionalities:

- List the existing tasks (`index()`).
- Display current task status (`status()`).
- Get the help string for the tasks and its parameters (`help(task name)`).
- Run a task. For instance, for the task GoTo described above, one can use console to run: `GoTo(goal_x=5.0,goal_y=5.0)`.

To launch the console, assuming the package `task_manager_lib` is in the current ROS workspace, one can use:

```
rosrun task_manager_lib console -s /server_node
```

where `server_node` is the name of the ros node instantiating the task server.

4.4 Simple missions

Simple missions also use the Python client to the task server and ressemble the following script:

```python
#!/usr/bin/python
import roslib; roslib.load_manifest('task_manager_turtlesim')
import rospy
from task_manager_lib.TaskClient import *

rospy.init_node('task_client')
tc = TaskClient("/task_server",0.2)

while True:
    tc.Wait(duration=1.)
    tc.GoTo(goal_x=1.0,goal_y=1.0)
    tc.Wait(duration=2.)
    tc.GoTo(goal_x=5.0,goal_y=5.0)
```

Up to the creation of the `tc` variable, missions are simple ROS nodes implemented in python. The task client class is instantiated in the variable `tc` and requires a first argument which is the name of the node implementing the task server and the default control loop frequency ($0.2s$ or $5Hz$ in this example). Based on this information the task client connects to the task server, gets the list of tasks, and the descriptions of their parameters. It then creates virtual member functions with the name of the tasks with parameters passed as dictionaries.

Because the task client gets the task list from the server, there is no need to specialize it when implementing a new task framework. There is also no need to change anything but the mission when a new task has been added to the server.

An important advantage of using Python to define missions is the possibility to use the normal Python control flow in the mission definition (here a `while` statement). Furthermore, because a mission is a normal ROS node, one can add subscribers to the mission script and possibly take mission decision based on variables received over ROS.

If a task fails while being executed (real failure, time out, ...), a `TaskException` will be generated. It can be caught within the mission script with a standard `try ... catch` statement and acted upon as appropriate.

```python
    try:
        tc.Wait(duration=0.2)
        tc.GoTo(goal=Andromeda)
    except TaskException, e:
        # This means a task failed. We need to react to it
        rospy.loginfo("Task fail with error status %d: %s" \
            %(e.status, str(e)))
```

4.5 Background tasks

In some situations, it may be useful to start a task in the background while another one is running. It may also be necessary to start two tasks simultaneously and wait for both of them to complete. Obviously, this only makes sense if the two tasks do not control the same actuators.

By default, tasks are started in the foreground. Only a single task can be run in the foreground, so when a foreground task starts, it first terminates any existing foreground task (typically the Idle task).

To start a task in the background, one just needs to add an argument `foreground=False` to its parameter list. In this case, the task will be run in its own thread and will not be killed by starting a foreground task concurrently. The function call then returns the task id:

```
id = tc.WaitForROI(foreground=False,roi_x=1.,roi_y=6.,roi_radius=1.0)
```

To wait for a background task to complete, the task client class provides several helper function:

- `tc.waitTask(id)` wait for the completion of a single task.
- `tc.waitAnyTasks([id1,id2,...])` wait for the completion of at least one task within the provided list.
- `tc.waitAllTasks([id1,id2,...])` wait for the completion of all the tasks within the provided list.

A background task can be terminated with `tc.stopTask(id)` and `tc.stopAllTasks()` terminates all tasks currently running.

4.6 Interruptions

There are often well-defined situations where a mission should be interrupted. This is particularly true in monitoring or service missions where low battery or an unexpected event require aborting the routine mission and starting a specific action.

The task framework defines these events as Condition variable. Such a Condition class must have a member function names `isVerified()`. Currently, the most useful condition is `ConditionIsCompleted`, which is True when the task it monitors is completed (or terminated). To add such a condition, the following syntax is available:

```
id = tc.WaitForROI(foreground=False,roi_x=1.,roi_y=6.,roi_radius=1.0)
tc.addCondition(ConditionIsCompleted("ROI␣detector",tc,id))
```

With such a condition, the mission can be written within a try-catch statement. On the condition being verified, a TaskConditionException is raised.

```
    try:
        tc.Wait(duration=0.2)
        tc.GoTo(goal_x=0.0,goal_y=1.0)
        # Clear the conditions if we reach this point
        tc.clearConditions()
        tc.stopTask(id)
    except TaskConditionException, e:
        # This means the conditions were triggered. We need to react to it
        # Conditions are cleared on trigger
        DoSomething()
```

Note that if the code reaches a point where a specific condition is not required anymore, it should use the `tc.clearConditions()` function to remove the current condition set. If the condition was waiting on the completion of a specific task, then this one must still be running and should probably be terminated before moving further.

5 Integration with other frameworks

5.1 Integration with Smach

Smach[3] is a Python framework within ROS to create complex state machines. In comparison with the missions written with the task manager, a Smach state machine lives completely within the Python scripts it is instantiated from. Each state is a Python class and Smach provides a nice framework to define potential transitions based on the outcome of a state. Smach also provides a graphical visualisation tool that displays the full state machine, its transitions and the state currently active. All this allows combining behaviours in a less linear and more complex way.

To take advantage of the strength of Smach while keeping our tasks as well defined behaviours, a wrapper class MissionStateMachine is provided in the TaskSmach module. This class provides three main functions to create containers (check Smach documentation for details on the type of containers):

- `createStateMachine()`: creates a state machine with parameters compatible with the task framework.
- `createSequence()`: create a container for states to be executed as a sequence, which is the most common setup for a mission.
- `createConcurrence(fg_task)`: create a container for at least two states (or nested state-machines) to be executed concurrently. The concurrence terminates when one of the states or nested state-machines terminates.

It also provides three functions to create tasks as state-machine states:

- `task(name, **params)`: create a generic state to be added to the state machine.
- `seq_task(name, ** params)`: create a state to be inserted in a sequence.
- `concurrent_task(name, ** params)`: create a state to be inserted in a concurrence.
- `epsilon_task(label,transitions)`: create a state that does nothing but can be used as a common branching points for real state.

Finally, the MissionStateMachine provides a run function that also instantiates the introspection server and handles ROS shutdown in a clean way.

An example of Smach based mission can be seen below:

```python
#!/usr/bin/python
# ROS specific imports
import roslib; roslib.load_manifest('task_manager_turtlesim')
import rospy
from math import *
from task_manager_lib.TaskSmach import *

rospy.init_node('task_client')
# Create a SMACH state machine
mi = MissionStateMachine()
sm = mi.createSequence()

# Add states to the container
with sm:
    # Create the initial state, and keep its id.
    init = mi.seq_task("Wait",duration=1.0)
    mi.seq_task("GoTo",goal_x=1.0,goal_y=1.0)

    # Create a concurrence state to handle background tasks
    sm_con = mi.createConcurrence('normal_seq')
    with sm_con:
        # First background task
        mi.concurrent_task("WaitForROI",foreground=False,roi_x=9.,roi_y=6.,roi_radius=1.0)
        # The second concurrent state is actually a sequence
        sm_sub = mi.createSequence()
```

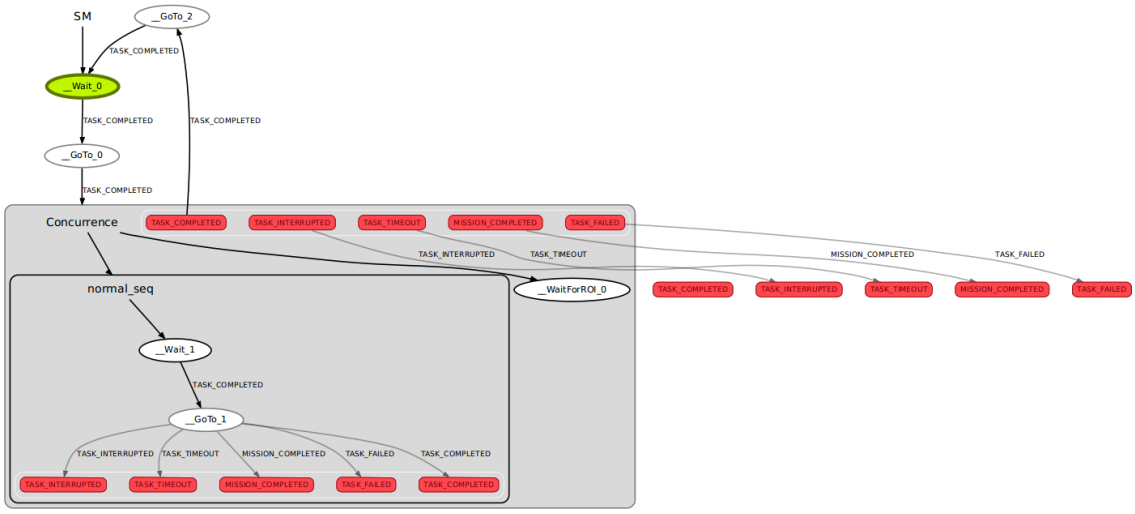
[3] http://wiki.ros.org/smach/Documentation

```
        with sm_sub:
            # Execute the three following task in sequence (assumes p is defined)
            mi.seq_task("Wait",duration=0.2)
            mi.seq_task("GoTo",goal_x=p[0],goal_y=p[1])
        # Add the sequence to the concurrence
        smach.Concurrence.add('normal_seq',sm_sub)

    smach.Sequence.add('Concurrence',sm_con)
    # Add the final task, and force it to transition to the initial state.
    mi.seq_task("GoTo",goal_x=5.0,goal_y=5.0,transitions={'TASK_COMPLETED':init})

mi.run(sm)
```

The resulting state machine can be visualized with `smach_viewer`:



5.2 Integration with the ActionLib

The ActionLib[4] is a ROS framework that provides a way to execute long actions, monitor their execution status and interrupt them. As such, its semantic integrates extremely well with the task framework presented here. It does not, however, provide a way to sequence different actions in an easy to read manner.

An action is described by a set of 3 messages, declared in an `.action` file: a goal, a feedback and a result. In the context of the task manager, we are mostly concerned with setting the action goal from the task arguments. The feedback is used to check if the corresponding task can report itself completed and the result is left to the responsibility of the user task.

The integration between the task manager and the ROS ActionLib is implemented in the `task_manager_action` package in the `TaskActionGeneric` templated class defined in the eponymous header. This class is templated on the action type, the task configuration and the task manager environment as any other task instance. For a specific implementation, the user will inherit from `TaskActionGeneric` and specify the following functions:

`getActionName()`: should provide the name of the action server (see the ActionLib documentation).

`buildActionGoal(Goal & goal)`: fill in the goal object based on the task configuration variable in `cfg`.

`handleResult(ResultConstPtr result)`: maybe overloaded if the task result need to be stored or acted upon in some way.

As an example, the MoveBase integration described below uses the ActionLib and integrates with it through the `TaskActionGeneric` class:

[4] http://wiki.ros.org/actionlib

```
template <class TaskEnvironment>
class TaskActionMoveBase : public TaskActionGeneric<move_base_msgs::MoveBaseAction,
    TaskActionMoveBaseConfig, TaskEnvironment>
{
    protected:
        // Create an alias to the Parent class, which must be explicitly used
        // in this templated inheritance.
        typedef TaskActionGeneric<move_base_msgs::MoveBaseAction,
                TaskActionMoveBaseConfig, TaskEnvironment> Parent;

        // In this case the name of the action server is stored in the task
        // cfg variable.
        const std::string & getActionName() const {
            return Parent::cfg.action_name;
        }

        // Convert the goal from the cfg variable to an ActionLib goal.
        void buildActionGoal(typename Parent::Goal & goal) const {
            const TaskActionMoveBaseConfig & cfg_ = Parent::cfg;
            goal.target_pose.header.frame_id = cfg_.frame_id;
            goal.target_pose.header.stamp = ros::Time::now();

            goal.target_pose.pose.position.x = cfg_.goal_x;
            goal.target_pose.pose.position.y = cfg_.goal_y;
            goal.target_pose.pose.position.z = cfg_.goal_z;
            goal.target_pose.pose.orientation =
                tf::createQuaternionMsgFromRollPitchYaw(cfg_.goal_roll,
                cfg_.goal_pitch,cfg_.goal_yaw);
        }

        // No result to handle here.
    public:
        TaskActionMoveBase(task_manager_lib::TaskDefinitionPtr def,
                task_manager_lib::TaskEnvironmentPtr env) : Parent(def,env) {}
        virtual ~TaskActionMoveBase() {};
        // Nothing more required, the initialize, iterate and terminate
        // functions are defined by TaskActionGeneric.
};
```

5.3 Integration with MoveBase

MoveBase[5] is a ROS framework to handle the navigation of a mobile robot to a goal destination defined as a ROS pose. It proposes two interfaces: a simple goal which is received on a ROS topic and is reached without providing explicit feedback and an ActionLib interface. The two interfaces are proposed in the tasks `TaskActionMoveBase` and `TaskActionMoveGoal` in the `task_manager_action` package. Check the `task_manager_turtlesim` package for implementation examples, even though MoveBase is not implemented for the turtlesim.

The `TaskActionMoveBase` uses the ActionLib as defined above. It proposes a generic config variable with a 6 DoF goal in a specified frame and an action name. Because the `TaskActionMoveBase` implements all the interaction with the action server, the instantiation of this task requires simply to pass the relevant TaskEnvironment. Here is an example from the `task_manager_turtlesim` package:

```
namespace task_manager_turtlesim {
    class TaskMoveBase : public TaskActionMoveBase<TurtleSimEnv>
    {
            TaskMoveBase(TaskDefinitionPtr def, TaskEnvironmentPtr env) :
                TaskActionMoveBase<TurtleSimEnv>(def,env) {}
            virtual ~TaskMoveBase() {};
    };

    class TaskFactoryMoveBase : public TaskFactoryActionMoveBase<TurtleSimEnv>
    {

        public:
            TaskFactoryMoveBase(TaskEnvironmentPtr env) :
                TaskFactoryActionMoveBase<TurtleSimEnv>(env) {}
            virtual ~TaskFactoryMoveBase() {};
    };
};
```

[5] http://wiki.ros.org/move_base

The `TaskActionMoveGoal` uses the simple goal of the MoveBase framework but requires a much more complex config structure to achieve the same result. In its simplest form, the task just send the goal once and report itself completed. If `wait_completion` is set, the task uses a transform listener (tf package) to monitor the current system configuration and reports itself complete when the error between the target goal and the current state comes below a threshold. Hence, the config variables contains the linear and angular thresholds, as well as the reference frame and vehicle frame to monitor. If necessary, the goal can also be resent regularly either to trigger a replanning or to handle unreliable connections. Despite the implementation difference, the task instantiation is very similar to the `TaskActionMoveBase` class. Here is an example from the `task_manager_turtlesim` package:

```
namespace task_manager_turtlesim {
    class TaskMoveGoal : public TaskActionMoveGoal<TurtleSimEnv>
    {
            TaskMoveGoal(TaskDefinitionPtr def, TaskEnvironmentPtr env) :
                TaskActionMoveGoal<TurtleSimEnv>(def,env) {}
            virtual ~TaskMoveGoal() {};
    };

    class TaskFactoryMoveGoal : public TaskFactoryActionMoveGoal<TurtleSimEnv>
    {

        public:
            TaskFactoryMoveGoal(TaskEnvironmentPtr env) :
                TaskFactoryActionMoveGoal<TurtleSimEnv>(env) {}
            virtual ~TaskFactoryMoveGoal() {};
    };
};
```

5.4 Tasks and multi-agent synchronization

In this section, we assume that multiple agents, each running their local roscore, are linked through the ros multimaster framework over a non-reliable communication medium, e.g. Wifi. More information about the multimaster concept can be found on http://wiki.ros.org/multimaster_fkie.

In this context, each agent may have it own task server and its own tasks and we may need to create synchronization checkpoints between their missions. This is specifically the objective of the `task_manager_sync` package. Note though that we are not intending to allocate tasks to a multi-agent system. An example of the kind of mission we are considering would consist of one loading robot and a delivery robot. The loading robot needs to wait for the delivery robot to be stopped at the loading dock before loading it, and the delivery truck needs to wait to be loaded before starting its delivery. Apart from these actions, both robot can run their missions independently.

An important assumption is that communication is not reliable and that, as a result, each robot is running his own roscore. The multimaster framework will synchronize topics when the two robots are connected on the same wifi, but they may not be connected all the time, and it cannot be assumed that standard ROS message will be delivered. This leads to a design based on hand-shaking, which unfortunately can be rather verbose. In such a system, any synchronization message needs to be repeatedly sent until acknowledged, without any guarantees that any of the messages will be delivered.

The `task_manager_sync` package adds two main elements to the normal task server and environment. Within the `TaskEnvironmentSync` parent class, every task servers is given a name (as a string) and publishes an integer state variable at a constant frequency while listening for the equivalent variables published by the other task servers on the other agents. Two tasks are also added automatically to the task server:

- `SetStatusSync(status)` can be used by one agent to change the value of its own state and set it to `status`.
- `WaitForStatusSync(partner,status)` waits for the partner identified by `partner` to reach state `status`.

Note that this design guarantees that the synchronization point can be reached if at least one message is delivered. However, it does not prevent deadlocks and it does not guarantee that a state will be "seen" by the synchronised agents if it is published for too short a time. As a result, it is left to the mission designer to make sure that state will only change when acknowledged. For instance, the delivery truck docks, set its state to "docked" and wait for the loading truck to switch to state "loading finished". At the same time, the loading truck wait for the delivery truck to reach the "docked state", loads it and then set its state to "loading finished". In this case, string state are used to simplify the semantic, but integers are used in practice. The mapping between integer and semantic is left to the mission designer as well.

5.4.1 Implementation: task server and environment

Similarly to the other section, an implementation example for the turtlesim is provided in the `task_manager_turtlesim_sync` package, which is designed to control three turtles in the simulation window.

Regarding the environment, the only required change is to inherit from the `TaskEnvironmentSync` class, and to provide a name to this instance of the task server:

```
// File: task_manager_turtlesim_sync/TurtleSimEnv.h
#ifndef TURTLE_SIM_ENV_H
#define TURTLE_SIM_ENV_H

#include "task_manager_sync/TaskEnvironmentSync.h"

namespace task_manager_turtlesim_sync {
    class TurtleSimEnv: public task_manager_sync::TaskEnvironmentSync
    {
        protected:
            // Same code as the original TurtleSimEnv

        public:
            TurtleSimEnv(ros::NodeHandle & nh, const std::string & name);
            ~TurtleSimEnv() {};

            // Same code as the original TurtleSimEnv
    };
    typedef boost::shared_ptr<TurtleSimEnv> TurtleSimEnvPtr;
    typedef boost::shared_ptr<TurtleSimEnv const> TurtleSimEnvConstPtr;
};

#endif // TURTLE_SIM_ENV_H
```

```
// File: TurtleSimEnv.cpp
#include "task_manager_turtlesim_sync/TurtleSimEnv.h"

using namespace task_manager_turtlesim_sync;

TurtleSimEnv::TurtleSimEnv(ros::NodeHandle & n, const std::string & name) :
        task_manager_sync::TaskEnvironmentSync(n,name,"sync") { // "sync" is the topic name
              for the state publisher
    ...
}
```

Using this environment, the task server can be instantiated easily:

```
// File: task_server.cpp
#include "task_manager_sync/TaskServerSync.h"
#include "task_manager_turtlesim_sync/TurtleSimEnv.h"

using namespace task_manager_turtlesim_sync;
using namespace task_manager_lib;

// Everything inherited from TaskServerSync
class TaskServer : public task_manager_sync::TaskServerSync {
    public:
        TaskServer(TaskEnvironmentPtr _env) : task_manager_sync::TaskServerSync(_env) {
            start();
        }

};

int main(int argc, char *argv[])
{
```

```
    ros::init(argc,argv,"turtlesim_tasks");//init ros
    ros::NodeHandle nh("~");
    std::string partner_name = "partner";
    nh.getParam("my_name",partner_name);

    // Create an environment for this instance and make sure the partner name
    // used.
    TurtleSimEnvPtr env(new TurtleSimEnv(nh,partner_name));
    // Add subscription to sync signals from other agents (which may be
    // received from the multimaster sync mechanism. The topic names "partner1"
    // or "partner2" are expected to be remapped in the launch file.
    env->addSyncSource("partner1");
    env->addSyncSource("partner2");
    // And we are done.
    TaskServer ts(env);
    ros::spin();
    return 0;
}
```

Finally, this task server is instantiated and with the following launch file (without the multi-master context)

```
<?xml version="1.0" encoding="UTF-8" ?>
<launch>
    <!-- Launching the turtlesim node and spawning two additional turtles -->
    <node name="turtlesim" pkg="turtlesim" type="turtlesim_node"  />
    <node pkg="rosservice" type="rosservice" name="spawn2" args="call␣--wait␣/spawn␣4␣4␣0␣
        turtle2"/>
    <node pkg="rosservice" type="rosservice" name="spawn3" args="call␣--wait␣/spawn␣7␣7␣0␣
        turtle3"/>

    <!-- Launching task server for turtle 1 and connecting it to the sync signals from
        turtle 2 and 3 -->
    <node name="turtlesim_tasks1" pkg="task_manager_turtlesim_sync" type="
        turtlesim_task_server_sync"  output="screen">
        <param name="lib_path" value="$(find␣task_manager_turtlesim_sync)/tasks"/>
        <param name="my_id" value="1"/>
        <param name="my_name" value="partner1"/>
        <remap from="~/partner1" to="/turtlesim_tasks2/sync"/>
        <remap from="~/partner2" to="/turtlesim_tasks3/sync"/>
    </node>

    <!-- Launching task server for turtle 2 and connecting it to the sync signals from
        turtle 1 and 3 -->
    <node name="turtlesim_tasks2" pkg="task_manager_turtlesim_sync" type="
        turtlesim_task_server_sync"  output="screen">
        <param name="lib_path" value="$(find␣task_manager_turtlesim_sync)/tasks"/>
        <param name="my_id" value="2"/>
        <param name="my_name" value="partner2"/>
        <remap from="~/partner1" to="/turtlesim_tasks1/sync"/>
        <remap from="~/partner2" to="/turtlesim_tasks3/sync"/>
    </node>

    <!-- Launching task server for turtle 3 and connecting it to the sync signals from
        turtle 1 and 2 -->
    <node name="turtlesim_tasks3" pkg="task_manager_turtlesim_sync" type="
        turtlesim_task_server_sync"  output="screen">
        <param name="lib_path" value="$(find␣task_manager_turtlesim_sync)/tasks"/>
        <param name="my_id" value="3"/>
        <param name="my_name" value="partner3"/>
        <remap from="~/partner1" to="/turtlesim_tasks1/sync"/>
        <remap from="~/partner2" to="/turtlesim_tasks2/sync"/>
    </node>
</launch>
```

5.4.2 Mission examples

The task set used in the `task_manager_turtlesim_sync` is the same as for the standard `task_manager_turtlesim`, with the addition of the two tasks mentioned above. The resulting sample missions are given below. They result in the 3 turtles doing their own missions but with sync point where they wait for each other.

- Turtle 1

```
#!/usr/bin/python
```

```
# File mission1.py: mission for turtle 1
import roslib; roslib.load_manifest('task_manager_turtlesim_sync')
import rospy
from task_manager_lib.TaskClient import *

rospy.init_node('task_client1')
# Make sure to select the specific server node
server_node = rospy.get_param("~server","/turtlesim_tasks1")
default_period = rospy.get_param("~period",0.2)
tc = TaskClient(server_node,default_period)

# reset status
tc.SetStatusSync(status=0);
# Move to start position and switch to state 1
tc.GoTo(goal_x=1.0,goal_y=1.0)
tc.SetStatusSync(status=1);
# Wait for the two other turtles to have completed their tasks and switched to status 1
tc.WaitForStatusSync(partner="partner2",status=1);
tc.WaitForStatusSync(partner="partner3",status=1);
# Go back to the center and switch back to state 1
tc.GoTo(goal_x=5.0,goal_y=5.0)
tc.SetStatusSync(status=0);

rospy.loginfo("Mission1 completed")
```

- Turtle 2

```
#!/usr/bin/python
# File mission2.py: mission for turtle 2
import roslib; roslib.load_manifest('task_manager_turtlesim_sync')
import rospy
from task_manager_lib.TaskClient import *

rospy.init_node('task_client2')
# Make sure to select the specific server node
server_node = rospy.get_param("~server","/turtlesim_tasks2")
default_period = rospy.get_param("~period",0.2)
tc = TaskClient(server_node,default_period)

# Reset state
tc.SetStatusSync(status=0);
# Wait for turtle 1 to switch to state 1
tc.WaitForStatusSync(partner="partner1",status=1);
# Move to specific position and switch this turtle's state to 1
tc.GoTo(goal_x=9.0,goal_y=1.0)
tc.SetStatusSync(status=1);
# Wait for turtle 3 to reach 1 as well, and for turtle 1 to come back to zero.
tc.WaitForStatusSync(partner="partner3",status=1);
tc.WaitForStatusSync(partner="partner1",status=0);
# Now it is time to go home and switch back to 0.
tc.GoTo(goal_x=9.0,goal_y=9.0)
tc.SetStatusSync(status=0);

rospy.loginfo("Mission2 completed")
```

- Turtle 3

```
#!/usr/bin/python
# File mission3.py: mission for turtle 3
import roslib; roslib.load_manifest('task_manager_turtlesim_sync')
import rospy
from task_manager_lib.TaskClient import *

rospy.init_node('task_client3')
# Make sure to select the specific server node
server_node = rospy.get_param("~server","/turtlesim_tasks3")
default_period = rospy.get_param("~period",0.2)
tc = TaskClient(server_node,default_period)

# Reset state
tc.SetStatusSync(status=0);
# Wait for turtle 2 to switch to state 1 (which will happen after turtle 1 does it)
tc.WaitForStatusSync(partner="partner2",status=1);
# Move to specific position and then switch this turtle's state to 1
tc.GoTo(goal_x=1.0,goal_y=9.0)
tc.SetStatusSync(status=1);
# Wait for the two other turtles to switch back to 0
tc.WaitForStatusSync(partner="partner1",status=0);
```

```
tc.WaitForStatusSync(partner="partner2",status=0);
# Finally go home and switch itself to 0.
tc.GoTo(goal_x=1.0,goal_y=1.0)
tc.SetStatusSync(status=0);

rospy.loginfo("Mission3 completed")
```

6 Acknowledgements

Some of the work presented in this document have been supported by EU Grant 644227 “Flourish”, http://flourish-project.eu/.