Distributed Cooperative Robot Systems

Students of the Lecture

October 25, 2016

Contents

1 Introduction 3

2 External Software 4
2.1 GIT and GitHub 4
2.2 Robot Operating System (ROS) 4
2.3 Build Chain 4

3 Process Manager and Remote Control GUI 5
3.1 Quickstart Guide 5
3.2 General Information on the Process Manager 6
3.3 General Information on the Remote Control GUI 7
3.4 Configuration 9
3.5 Future Work 11

4 ALICA Client 12

5 TurtleBot Hardware 13

6 TurtleBot Software 14

7 TurtleBot Gazebo Simulation 15

1 Introduction

1.1 How to contribute to this document?

- LaTex
- TexMaker
- Git GitHUB

2 External Software

This chapter is intended to give a short introduction to external software packages or frameworks, we utilise for the Carpe Noctem Cassel software framework.

2.1 GIT and GitHub

GIT is one of the most advanced version control systems currently available. Nevertheless, during our daily work we only use 20% of its functionality. So for starters try to learn the stuff you need and ignore its advanced features like *rebase* and *cherry pick*.

The best reference and documentation about GIT can be found at `http://www.git-scm.com/docs`

Most of our software is published open source under the MIT License at our GitHub Repositories. Therefore, it is also interesting to read about the features of GitHub, like SSH-Key based authorization, groups, organisations, and MarkDown.

The README.md files in our repositories are written in MarkDown, because GitHub parses these MarkDown files and auto-generates and HTML documentation from it.

2.2 Robot Operating System (ROS)

ROS, as we use it, is a simple inter process communication middleware. Before you ask, yes it is not intended to be used for inter machine/robot communication. Therefore, we have developed a simple ROSUdpProxy, for our purposes at RoboCup.

Tutorials for ROS can be found here: `http://wiki.ros.org/ROS/Tutorials`

2.3 Build Chain

We utilise *catkin* from the ROS Universe as our build chain. Catkin is basically a workspace-oriented extension of CMake. Therefore, it heavily relies on CMake and in order to understand catkin it is recommended to understand CMake first.

CMake is open source and developed by KitWare. Basically CMake autogenerates Makefiles out of CMakeLists.txt files located in each software module. Therefore, our build chain can really be considered as a chain ☺:

Catkin $\xrightarrow{manages}$ CMake $\xrightarrow{auto-generates}$ Makefiles $\xrightarrow{commands}$ GCC $\xrightarrow{to\ compile}$ executables and libraries.

3 Process Manager and Remote Control GUI

The process manager is an executable for managing the processes running on a PC. Compared with the former C# framework, it replaces the process manager Care. The process manager can run on the robot for testing with real a robot, or on your local PC for testing with a simulator (add -sim as parameter). With the help of the *ROBOT* environment variable, it is possible to manage processes for multiple robots on a single PC, which is useful for multi-robot testing on a single PC. The name of the ROS-Package of the process manager is *process_manager* and can be found by using `roscd process_manager`.

The remote control GUI for the process manager is an RQT plugin (see `http://wiki.ros.org/rqt` for details). In the former C# framework this GUI was highly integrated into the LebtClient and mixed with robocup msl specific GUI elements. The idea of the new GUI is, to make it useable in other domains, too. The name of the ROS-Package of the remote control GUI is *pm_control* and can be found by using `roscd pm_control`.

3.1 Quickstart Guide

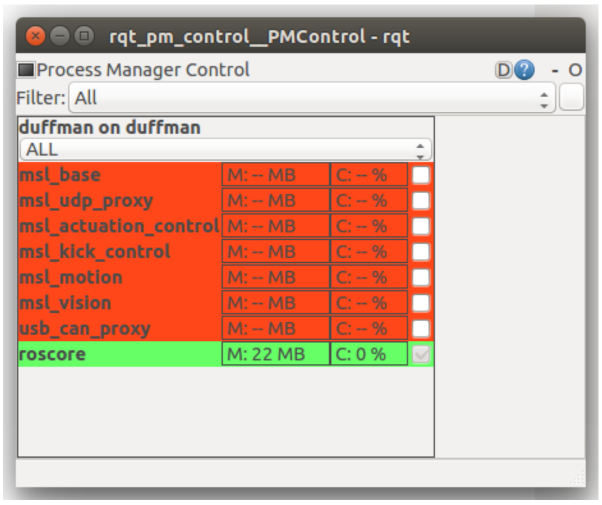
In order to bring up a single process manager and its remote control GUI, on the same PC, execute the following commands in the given order:

- `rosrun process_manager process_manager`
- `rosrun pm_control pm_control`

The first command starts the process manager, which will automatically start a roscore, if none is running. Please add the `-sim` parameter to the process manager, if you want to use it locally managing multiple robots (e.g. for simulation). The second command start the remote control GUI, which should display the received information of the process manager (see Figure 3.1).

If the process manager and the remote control GUI should run on different PCs, which are in the same network, you need to make sure, that on both machines a roscore and a UDP proxy is running. Therefore, you need to execute the following commands on the machine, where the process manager should run:

- `rosrun msl_udp_proxy msl_udp_proxy`
- `rosrun process_manager process_manager`


rqt_pm_control__PMControl - rqt

Process Manager Control

Filter: All

duffman on duffman

ALL

msl_base	M: -- MB	C: -- %	☐
msl_udp_proxy	M: -- MB	C: -- %	☐
msl_actuation_control	M: -- MB	C: -- %	☐
msl_kick_control	M: -- MB	C: -- %	☐
msl_motion	M: -- MB	C: -- %	☐
msl_vision	M: -- MB	C: -- %	☐
usb_can_proxy	M: -- MB	C: -- %	☐
roscore	M: 22 MB	C: 0 %	✓

Figure 3.1: The Remote Controle GUI

The msl-udp-proxy is an autogenerated proxy, which broadcasts MSL specific ROS messages from the local machine to a multicast address on the network. You can also use another UDP proxy (e.g. ttb-udp-proxy, or bbb-udp-proxy), as along as it forwards the *ProcessCommand* and *ProcessStats* ROS messages of the process manager.

On the machine, where the remote control GUI should run, execute:

- `roscore`
- `rosrun msl_udp_proxy msl_udp_proxy`
- `rosrun pm_control pm_control`

You need to start the roscore manually, as the remote control GUI does not start a roscore automatically.

3.2 General Information on the Process Manager

PROC Filesystem

The information about the managed processes are collected from the */proc* filesystem. This is a specific path, which is available on each linux-kernel-based operating system. Inside the /proc folder, each running process has a seperate subfolder, named after its process id (PID). Please note, that this PID is determined by the kernel and has nothing to do with the process ids of the ProcessManaging.conf file. The process manager uses the information available in the processes *cmdline* and *stat* file (e.g. /proc/2341/cmdline). These files are continuesly updated by the kernel. For further information about the proc filesystem, consider chapter 7 of the book *Advanced Linux Programming* (available at: `http://www.advancedlinuxprogramming.com/alp-folder/`).

The interesting thing about the proc filessystem approach is, that it enables the process manager to attach to processes it did not start in the first place! If you had some processes already up and running, just start the process manager and its remote control GUI to see the statistics about those processes.

Another fact about the process manager is, that it does not do anything to the processes, when it is closed. The launched processes are independent of the process manager and continue their execution without it. You accidentally stopped the process manager? No problem just restart the process manager and it will continue to monitor the started processes.

Communication

The process manager defines two ROS messages: *ProcessCommand* and *ProcessStats*. The ProcessCommand message is used to let him start and stop processes. The ProcessStats message is used by the process manager to report the statistics about its managed processes. The process manager is configured to scan the proc filesystem every two seconds, so that the ProcessStats is send roughly every two seconds.

The process manager subscribes on the ROS topic */process_manager/ProcessCommand* and publishes its commands to */process_manager/ProcessStats*.

Allowed Number of Processes

The process manager is written in a way, that it only allows to run a certain process only one time per robot. So if you want to run a process twice on the same machine, you need to modify the *ROBOT* environment variable for at least one of the two process instances. E.g.: `ROBOT=nase rosrun msl_base msl_base -m WM16` and `ROBOT=myo rosrun msl_base msl_base -m WM16`.

Another feature of the limitation on the number of allowed processes is, that if you did start too many processes, e.g. two or more image processing processes on one robot, you can simply start the process manager and it will clean up the mess. It will kill all but one process of each kind and start reporting statistics about the left processes.

3.3 General Information on the Remote Control GUI

Process GUI

The remote control GUI has a single process GUI element for each process (see Figure 3.2). From left to right it shows: the process name, its memory usage in MB, its cpu usage in percent (100% means one core), its check box for start and stop. The background color of the process GUI is either red (not running), green (running), gray (unknown). Note that the check box does not determine, whether a process is running or not, it is just for starting and stopping a process. Start-Commands for running processes are ignored by the process manager. The check box is disabled for the roscore, because stopping it would cut the communication to the process manager. Check boxes of other processes are disabled too, if they are running with parameters that differ from the currently selected bundle (see 3.3).

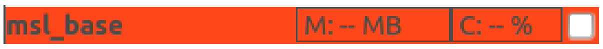


Figure 3.2: GUI Elements for one Process

If you have a running process (background is green), you can hover over the process GUI element, in order to show its ToolTip. The ToolTip shows the command, which was used to start the process. This way, you don't have to check the ProcessManaging.conf file, in order to know what parameters are used in a certain bundle.

Communication

The GUI elements are created at runtime, when a corresponding message from a process manager arrives. So if you don't receive any ProcessStats messages, your remote control GUI won't show anything. Furthermore, the GUI elements are deleted, if you don't receive messages for certain amount of time (roughly 3 seconds). The remote control GUI subscribes on the ROS topic */process_manager/ProcessStats* and publishes its commands to */process_manager/ProcessCommand*.

Bundle Selection

Selecting a bundle in the drop down box of the GUI, means that the listed processes will be started with the corresponding parameter set, as specified in the ProcessManaging.conf file (see Section 3.4). It also means, that you cannot stop a process which runs with a different parameter set, then specified in the selected bundle. In such cases, the check box of the process is disabled (grayed out). Nevertheless, there are two default bundles: ALL and RUNNING. If you select one of these two bundles, you can interact with all processes.

ALL lists all processes configured in the ProcessManaging.conf file. This is useful, if you want to start a set of processes, which is not specified as an explicit bundle. RUNNING lists all processes of the ProcessManaging.conf file, which are currently up and running. This is useful, for determining, whether there are unwanted processes running, which are not part of the bundle that you would like to use.

The *roscore* process is specially handled. If the roscore is stopped, the process manager cannot receive commands anymore. Therefore, it is not allowed to stop the roscore process within the remote control GUI.

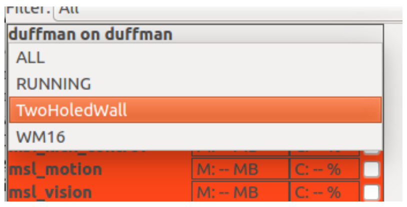


Figure 3.3: Bundle Selection by Drop Down Box

3.4 Configuration

In order to configure, which processes should be managed by the process manager, you need to edit the *ProcessManaging.conf* file. The file is usually located in the *etc* folder, determined by the *DOMAIN_CONFIG_FOLDER* environment variable. Inside the ProcessManaging.conf file several comments explain the config values itself. Nevertheless, lets explain the config for the msl_base process in detail:

```
[Base]
  id = 7
  execName = msl_base
  rosPackage = msl_base
  mode = none
  [paramSets]
    1 = -m, TwoHoledWallMaster
    2 = -m, ActuatorTestMaster
    3 = -m, WM16
    4 = -m, TestApproachBallMaster
    5 = -m, TestCheckGoalKicki
    6 = -m, WM16, -sim
  [!paramSets]
[!Base]
```

The **name** (Base) of the outmost section denotes the GUI-String representing the executable. The executable denoted by **execName** (msl_base) need to be located in the *PATH* environment variable, or should be found by executing
`catkin_find --libexec msl_base`
in order to work with the process manager. Here `msl_base` is the **rosPackage** name.

The **id** of the process must be unique in the ProcessManaging.conf file and is used to refer to this process in the bundles section (explained later) and in the messages send to and received from the process manager.

The **mode** decides how the process manager handles crashes of the process and some other things. At the moment, there are 3 different modes.

none Basically does nothing. It does not autostart the process, when the process manager is started with the *-autostart* parameter. It does not restart the process, when it did crash.

keepAlive Processes, configured with this mode, will be restarted by the process manager, when they crashed.

autostart This mode makes the process manager start this process, when the process manager is started with the *-autostart* parameter and restarts it after chrashes.

In the **paramSets** sections, it is possible to specify different sets of parameters which can be used to start the process. Each parameter set follow a simple key-value-pair convention, where the key must be a (for this process unique) positive integer, greater than 0. The value is a comma (,) seperated list of parameters. Please note, that the parameter `-m TwoHoledWallMaster` is actually two parameters: `-m` and `TwoHoledWallMaster`. The parameter set with the lowest key is considered to be the default parameter set, which means that this parameter set is used, if not specified otherwise (e.g. by choosing a bundle).

The **bundles** section of the ProcessManaging.conf file allows to specify a set of processes with a specific parameter set for each of it. Here is a small example:

```
[Bundles]
  [WM16]
    processList        = 0,1,2,3,4,5,6,7
    processParamsList  = 0,0,1,0,0,0,0,3
  [!WM16]

  [TwoHoledWall]
    processList        = 0,1,2,3,4,5,6,7
    processParamsList  = 0,0,1,0,0,0,0,1
  [!TwoHoledWall]
[!Bundles]
```

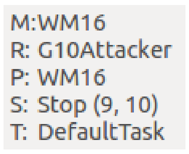
In this example, two bundles are specified: WM16 and TwoHoledWall. Each bundle consists of two key-value-pairs: processList and processParamsList. The processList specifies the list of processes, by listing the process ids. The processParamsList specifies the parameter sets for each process. It is important that the order of the processParamsList has to be the same as in the processList. For example: In the WM16 bundle the process with the id 7 (Base) is started with the parameter set id 3 (`-m, WM16`). In the TwoHoledWall bundle, it is started with the parameter set id 1 (`-m, TwoHoledWallMaster`). The parameter set id 0 is a special value, with the meaning, that there is no parameter set specified for this process, because it has no parameters. Another special value, which is only used in the messages send to and received from the process manager, is -1. It says that the parameters of a process are unkown, e.g., the msl_base is running with an unknown master plan (`-m FancyNewTestPlanMaster`).

3.5 Future Work

- Make the retry timeout for starting processes a parameter in the ProcessManaging.conf file.
- Make it possible to add the GUI for another virtual robot, in order to command a process manager to start processes for another robot. This is necessary for local testing with multiple robots.
- Document the implemented feature of starting launch scripts here. (For details see the ProcessManaging.conf file).

4 ALICA Client

The ALICA Client is a simple GUI for visualising AlicaEngineInfo messages. Currently it can be started for visualising the messages of a single engine by calling `rosrun alica_client alica_client`. If you have several robots running, the GUI will flicker between their incoming messages. For visualising the messages of multiple robots you should use the Robot Control GUI (see **??**). It integrates the same GUI component multiple times.



```
M:WM16
R: G10Attacker
P: WM16
S: Stop (9, 10)
T: DefaultTask
```

Figure 4.1: The ALICA Client GUI

In Figure 4.1 you can see the ALICA Client GUI. **M:** denotes the currently executed master plan (WM16). **R:** denotes the current role of the robot (G10Attacker). **P:** is the deepest plan in the alica plan hierarchy, the robot is currently executing. As it is WM16, it means, that it is inside the master plan, but not deeper. Inside this plan, the robot is currently one of two robots inside the Stop state (**S:**). In this case, it is either the robot with id 9 or 10. The name of the task associated with the active state is the DefaultTask (**T:**).

This tool is definitely usefull for debugging ALICA plans, but please note, that the AlicaEngineInfo messages are only send roughly every 100ms, but the plan state of an ALICA Engine typically changes much faster, than that. Therefore, you want recognize agents racing through the plans' state machines. Making this visible needs a litte bit more sophisticated GUI, which should be able to read logs of the ALICA engine itself. Please see the project description for some details, about this possible bachelor project.

5 TurtleBot Hardware

6 TurtleBot Software

7 TurtleBot Gazebo Simulation