Distributed Cooperative Robot Systems

Stephan Opfer

September 11, 2019

Contents

1 Introduction

1.1 Purpose of this Document

Instead of giving the students a lot of Powerpoint slides that compress the complex topics of this course into a bunch of bullet points, we wanted the students to have a dedicated document for preparing their exam. Unfortunately, such a document is a lot more work and it needs several iterations of thorough reviewing. Therefore, we came up with the compromise that the students write this document theirselves and the lecturers only have to review and improve it.

Maybe you ask: "What is wrong with Powerpoint?"

A Powerpoint presentation should aid a presenter in presenting a certain message or information to his audience. The slides alone often miss critical parts (that couldn't easily condensed into bullet points) and are unclear without the corresponding vocal line of the presenter. Nevertheless, students end up alone with the slides trying to prepare their examinations.

Apart from that, try to google topics like "disadvantages of powerpoint" or "why we should ban Powerpoint" and you will find interesting articles that connect Powerpoint and the Accident of the Columbia Space Shuttle [1] [2].

1.2 How to Contribute to this Document?

Basically there are two aspects. At first you need to get the LaTeX source code of this document from our cnc-turtlebots GitHub Repository. It is located in the folder `doc/latex`. Anybody can download it, but for making changes to it you need to have an GitHub-Account which was given the corresponding privileges. That is typically arranged during the first session of this course.

This leaves us with the requirements, that you are able to write LaTeX and know how to use GIT in combination with GitHub. Regarding GIT and GitHub we recommend to read Section 3.1.

Maybe LaTeX is hard to master when it comes to tables, graphics and customising the layout of a document, but you don't have to do that for contributing to this document. You should only be able to write text referencing images and place useful links from time to time. Furthermore, you should be able to structure the content into chapters, sections, and subsections. Everything just mentioned is already done in this document, so you can start by copying commands from the corresponding sections.

For further reference and an in-depth study of LaTeX, we recommend the following sources:

LaTeX – eine Einführung und ein bisschen mehr: A little bit longer introduction to LaTeX basics.

LaTeX WikiBook: Nice for looking up simple stuff.

tex.stackexchange.com: Stack Overflow for LaTeX

TikZ: LaTeXpackage for designing advanced graphics directly in LaTeX.

The final issue is to decide which editor you choose for compiling this LaTeXdocument. We recommend TexMaker, because it is available for Ubuntu and Windows, although, there currently exists an issue with short cuts under Ubuntu 16.04 LTS. Consider this Ask Ubuntu Post for solving it.

With TexMaker you only have to open the `Software_Reference_Book.tex` file and click on `Quick Build`. If you get compile errors, you probably need to install the necessary LaTeXpackages. Under Ubuntu this is done by executing the following console command:

```
sudo apt-get install texlive-full
```

2 Foundations

In this chapter topics that are tackled during the lecture are introduced in a basic fashion. This includes agents, sensor, actuators, communication, etc. . .

2.1 Agents

Although not every agent is autonomous, our focus in this lecture is to design autonomous agents. Autonomous means, that the agent is not controlled from some external entity. It only perceives its environment and acts in order to change it. Why it tries to change its environment depends on the agent's properties. Agents can be reactive or proactive, they can communicate with other agents, remember the history of the environment, and predict the future of the environment according to some model. As a designer of the agents, you need to decide what kind of agents you need to achieve your goal. The famous book *Artificial Intelligence – A Modern Approach*, written by Russel and Norvic, elaborates properties of agents and environment in chapter two. The book is available in our library. **It is recommended to read chapter two for your exam preparations.**

2.1.1 Robot Control Architectures

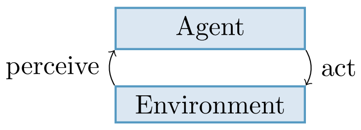


Figure 2.1: Simple Control Loop

Usually it is believed that the agent is interacting with its environment –whatever that environment is. This relation is often depicted as an agent-environment cycle (see Figure 2.1). As mentioned before, the details of an agent architecture do vary from use case to use case.

In Figure 2.2 an overview of the details of a state-of-the-art robot control system is shown. You should consider a robot as a physical agent, i.d., an agent with a physical representation. The Data Layers on the left hand side represent the utilised algorithms to gain information and knowledge from raw sensor data. During the data processing, the data is continuously filtered, condensed, and abstracted until symbols and relations can be integrated into a symbolic knowledge representation. This knowledge is required by the algorithms on the right hand side of Figure 2.2. The

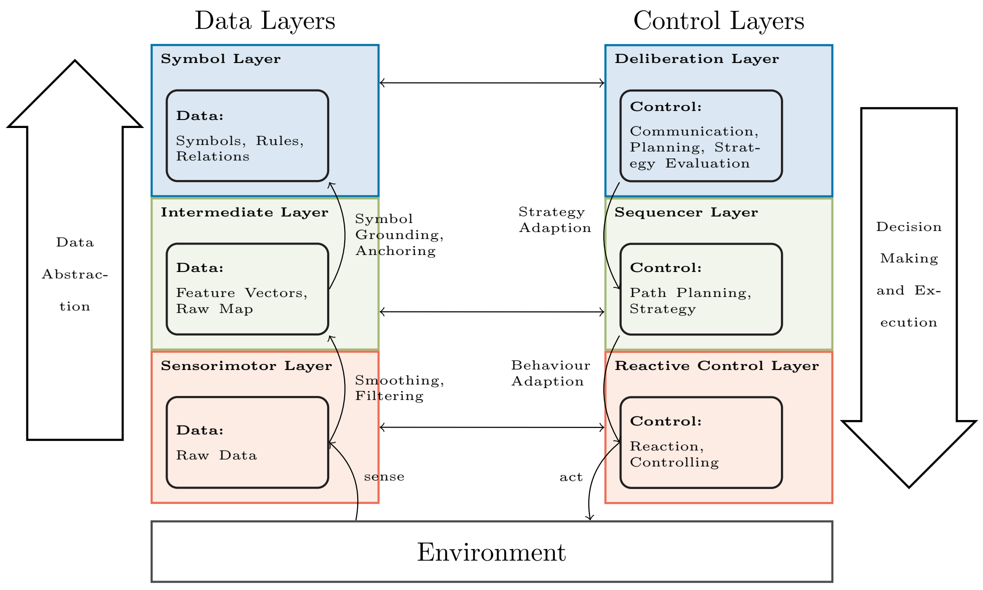


Figure 2.2: Control Loop

Deliberation Layer, including algorithms for planning, communication, and strategy evaluation, computes its decisions based on the knowledge of the symbolic knowledge representation. The results are processed by the Sequencer Layer and transformed into commands for controlling actuators of the robot.

The data abstraction and decision making cycle is not perfectly followed in all applications. Often the abstraction of data up to the level of symbols and relations is not necessary. Moreover, it isn't clever to make everything a decision on the top most available layers. Think of a robot control architecture as of a human brain. A lot of things in a human body are handled without having the brain actively thinking about it, e.g., breathing, heart beat, reflexes on your knee, sneezing, blinking, etc. For a robot the corresponding things could be obstacle avoidance, motor controlling, avoiding rough terrain such as stairs, etc. Things like that should also in a robot control architecture be handled on lower levels and not bothering algorithms for planning or communication.

However, don't take this easy. It can be quit difficult or even impossible to assign a certain task to a certain level. Obstacle avoidance, for example, is sometimes part of all three layers: While driving along a path, the robot in a office environment avoids humans or other robots by directly reacting to its 2D distance scans from its laser scanner (lower layers). The path itself is planned in order to avoid already known walls and finding the shortest path (intermediate layers). Planning this path is influenced by some knowledge about the environment, e.g., the robot knows that there is currently a meeting going on in some room and as the robots motion is relative noisy, it knows that it should avoid meetings like a virtual obstacle, in order to not disturb the meeting (top layers).

If you are in the position to decide on which layer something should be implemented, it sometimes helps to think about the following. The effort for abstracting data rises with each necessary processing step, therefore creating symbolic knowledge is often very costly. On the one hand, this argues for putting everything as on the lowest levels. On the other hand, the data on the lowest levels is often very large in terms of memory. You often have to ponder, whether it is more efficient to abstract the data and thereby reduce it to the smallest possible size or whether it is more efficient to directly operate on millions of raw camera pixel values without explicitly representing objects in the environment. Sometimes technical limitations also force you to a certain direction. If your robots, for example, need to communicate about objects it is probably impossible to send whole images of the objects to each other due to bandwidth limitations. Instead, the objects have to be abstracted to symbols or coordinates in order to match the bandwidth restrictions.

2.2 Sensors

Sensors are the only option for an agent to perceive its environment. This also means, that everything that helps an agent to perceive its environment can be considered as a sensor. Nevertheless, sensors are also used to measure internal values, not belonging to the environment. Before we are going to deep, let us start the sensor topic a little bit more general.

2.2.1 Sensor Categories

There are two sensor properties that help to categorise the plethora of sensors available nowadays: Active vs. passive sensors and external vs. internal sensors. Whether a sensor is active or passive determines the way it measures some value.

active: An active sensor emits something, e.g., energy, for measuring what ever it should measure. Typical examples are laser scanners or radars that emit electromagnetic waves of different wave length.

passive: Passive sensors directly measure energy from the environment. A camera for example perceives sunlight that is reflected from a surface, furthermore a compass measures the terrestrial magnetic field.

The difference between an internal or external sensor are the type of values the sensor measures, not how it measures it.

internal: Internal sensors measure values that are part of the system itself. Therefore it depends on the borders between the system and its environment. In case of a robot an internal sensor value could be its battery levels.

external: External sensors measure values from the outside of the system, i.d., values from its environment. Typical values are distances, light, and sound.

2.2.2 Sensor Properties

Subsection 2.2.1 is for categorising different types of sensors. If you need to choose a sensor for your application these categories are only a first orientation. In this subsection you will get a short overview of general things you have to consider in your decision for taking the right sensor for your application.

Size Does the sensor fit into your robot?

Weight Can the sensor be lifted or carried by your robot (especially relevant for drones)?

Energy Consumption Does the energy consumption of the sensor significantly reduce the runtime of your robot?

Sensor Range Sensors often have minimum and maximum values they can measure, e.g., distance starting from 0.4m up to 100m. Does the range fit your application?

Field of View In which direction can the sensor measure? Is it big enough? Can it be integrated into your robot without limiting the field of view?

Operating Conditions Some sensors are sensitive to temperature, vibration, light and so on. Does your application match their operating conditions?

Resolution What is the granularity of the measured values (directly influences properties from Subsection 2.2.3)?

Value Transformation Is the value received from the sensor in the unit you need? If not, do you know the transformation and how long does the transformation take?

Linearity Does a linear change in the value to be measured induce a linear change in the value received from the sensor? If not, what is the relation (see Value Transformation)?

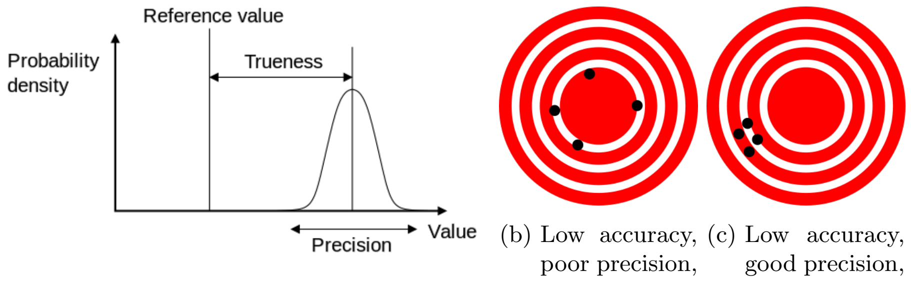
2.2.3 Sensor Value Properties

The sensor value properties of possible sensors are very important for making the right choice, too. In this subsection we collect these properties, explain their meaning, and explain why they are important.

The sketched definitions given below follow the ISO Standard 5725.

Precision Precision is about the reproducability or repeatability of measurements. The sensor has a high precision, if it always measures the same values under same conditions. Note that this measured values can be completely wrong!

Accuracy Accuracy is the average difference between the reference value (correct value) and the measured values. Please read this carefully, as it only slightly differs from trueness.



(a) Trueness vs. Precision

(b) Low accuracy, poor precision, good trueness

(c) Low accuracy, good precision, poor trueness

Figure 2.3: Examples for Precision, Accuracy and Trueness (Source: Wikipedia)

Trueness Trueness is the difference between the reference value (correct value) and the average of the measured values.

In Figure 2.3 several helpful pictures for understanding the above definitions are given. Especially the targets make the difference between accuracy and trueness comprehensible. In order to get a feeling for the relevance of precision, accuracy and trueness in practice let us exercise the utilisation of sensors with different properties. Imaging your task is to get to know what the correct value for a certain distance is.

The easiest case is when you have a highly accurate sensor, because this means that each measurement is relatively close to the correct value. It also means that the precision of the sensor must be relatively high, too. Here is why: If all measurements are close to a single static value (being accurate), the measurements cannot vary very much at the same time (low precision).

Now let us consider to use a sensor with low accuracy, but with high precision (like in Subfigure 2.3c). The measured values are relatively wrong, but the are reproduceable. Therefore, the sensor will measure under the same conditions the same wrong values. That gives you the chance to correct this relative constant offset or error by calibrating your sensor. For calibration you need a series of measurements for that you know the correct value. As you have a precise sensor, the difference between the measured values and the correct value should be more or less the same. In a naive calibration you could simply take this difference and correct the sensor's measurements by this difference at runtime. The problem is that precision means reproducability under the **same conditions**. These conditions often include the following parameters: the correct value, the state of the sensor (temperature, power supply, operation modes), and properties relevant for the measurement process (temperature, surfaces, light conditions). Therefore, calibration means to find the correct offset for all possible parameter combinations that influence the operating conditions of the sensor. Maybe this sounds like an almost unmanageable task, but here are some arguments against it:

- The sensor already can calibrate itself with a special operation mode that you only have to trigger from time to time.

- The most parameters are irrelevant for the measuring.
- Have a look at your application scenario, hopefully a lot of parameters are constant in you application (artificial lights, indoor temperatures).

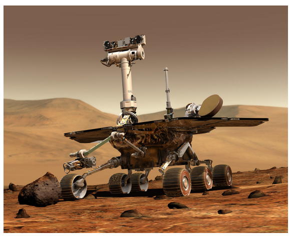
As a last relevant case, let us consider a sensor with low accuracy, but with high trueness (like in Subfigure 2.3b). Here the average of the measured values is relatively close the correct value. Therefore, you only need to take the average of some measurements, in order to know the correct value. The only question is: "How much measurements are necessary?" Simply make some experiments where you observe the change of the average relative to the increasing number of measurements. When the change of the average is low enough for your application you know the minimal number of measurements necessary to calculate a good average for determining the correct value. Think about your application then: Is it possible to make this number of measurements before the correct value changes, e.g., distance to moving obstacles. Obviously, you need more measurements if the standard deviation of the measurements is high, and less if its low. Note that a sensor with high trueness and a small standard deviation is actually an accurate sensor.

To summarise: Accurate sensors are fine, precise sensors need to be calibrated, and sensors with high trueness demand for calculating the average of some measurements.

2.3 Actuators

Actuators are the counterpart to sensors. While an agent perceives its environment with the help of sensors, it changes its environment with the help of actuators. Let's consider some examples and try to derive general categories from it (see Figure 2.4.

(a) Robot Arm (Source: Wikipedia)


(b) Robot Locomotion (Source: Wikipedia)

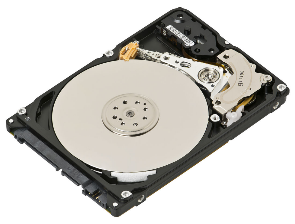

(c) Hard Disk Drive (Source: Wikipedia)

Figure 2.4: Examples for Actuators of Agents

Subfigure 2.4a shows a robto arm in a laboratory environment. This arm can manipulate the environment, e.g., by executing pick-and-place tasks, but it also represents the whole robot. In the context of this lecture we barely consider this arm as a physical agent for the following reasons: It is not autonomous, it has no external sensors for perceiving its environment, and it probably does the same movements over and over again. However, imagine that this arm is mounted on a mobile platform and controlled by an autonomous control architecture (see Subsection 2.1.1) that is also able to perceive its environment through dedicated sensors. Let us collect a list of rather blunt and naiv statements about this actuator.

- The arm is made for manipulating the environment, so it is not just an actuator, it is a manipulator.
- It has a certain flexibility which is often denoted as degrees-of-freedom (DOF).
- Reaching out from its mobile platform, its range is limited.
- Depending on its motors and energy sources, it can execute its movement with a limited force / speed.
- Depending on the mechanism at the end of the arm, it can only manipulate objects that suit this mechanism (denoted as Endeffector).

2.4 Communication

3 External Software and Developing Tools

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

3.1 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 an HTML documentation from it.

A list of GIT commands, that should be enough for the start, can be found on git-tower.com.

3.2 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.

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

After following this tutorial you should be able to explain a bunch of things:

Topics: How do they work?

Nodes: What is a ROS node?

roscore: What is its job?

package.xml: build_depend, run_depend, licences, ...

CMakeLists.txt: What are the critical ROS specific macros and how do they work? (not very well explained in the tutorial)

Console Commands: rosrun, rospack, roscd, rosls, roslaunch, rostopic, rosnode

catkin_make: How to compile a workspace/a package?

ROS-Workspace: What is its structure and why is it structured that way?

ROS-Services: How are they defined, compiled/generated, and how do they work?

ROS-Messages: How are they defined and compiled/generated?

roscpp API: How to create a publisher and a subscriber?

3.3 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.

4 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`.

4.1 Quickstart Guide

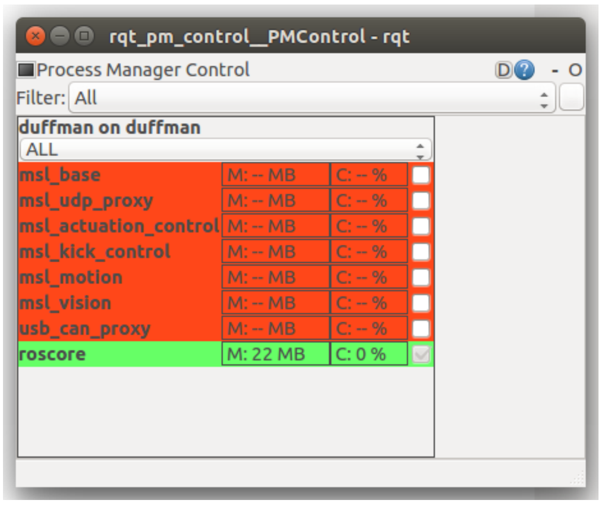
In order to bring up a single process manager and its remote control GUI, on the same PC, execute the following commands in the given order:

- `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 4.1).

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`



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 4.1: 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.

4.2 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.

4.3 General Information on the Remote Control GUI

Process GUI

The remote control GUI has a single process GUI element for each process (see Figure 4.2). 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 4.3).

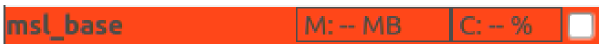


Figure 4.2: GUI Elements for one Process

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 4.4). 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.

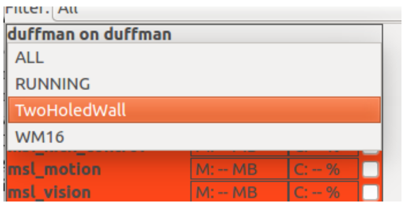


Figure 4.3: Bundle Selection by Drop Down Box

4.4 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]
```

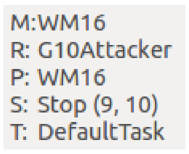
In this example, two bundles are specified: WM16 and TwoHoledWall. Each bundle consists of two key-value-pairs: processList and processParamsList. The processList specifies the list of processes, by listing the process ids. The processParamsList specifies the parameter sets for each process. It is important 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`).

4.5 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).

5 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 6). It integrates the same GUI component multiple times.



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

Figure 5.1: The ALICA Client GUI

In Figure 5.1 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.

6 Robot Control

TODO