Títol: Supervisor of Real Time Components for a Humanoid Robot Platform
Autor: Jordán Palacios

Data: 02/11/2011

Director: Carles Lopez
Empresa del director: PAL Robotics

Ponent: Xavier Martorell
Departament del ponent: Departament d'Arquitectura de Computadors (DAC)

Titulació: Enginyeria Informàtica

Centre: Facultat d'Informàtica de Barcelona (FIB)
Universitat: Universitat Politècnica de Catalunya (UPC) BarcelonaTech

SUPERVISOR OF REAL TIME COMPONENTS FOR A HUMANOID ROBOT PLATFORM

JORDÁN PALACIOS

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API Application Programming Interface

CAN Controller Area Network

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

GDB GNU Project Debugger

IDE Integrated Development Environment

OS Operating System

RTOS Real-Time Operating System

STL Standard Template Library

UI User Interface

UML Unified Modeling Language

1 INTRODUCTION

PAL Robotics is a company dedicated to the Research & Development of humanoid robots and robotic components. The staff consists of people from various countries, mostly engineers in the fields of mechanics, electronics and software, with many years of experience in the robotic industry.

The near-term goals for PAL Robotics are to introduce global best practices and solutions that help our customers to adapt and succeed in today's ever changing business environment, enhance PAL Robotics' brand recognition and image, and promote sustained growth through partnerships with leading companies from various focus-markets. Ultimately, we want to create the ideal robot for everybody's use.

The number of applications running in a service humanoid robot is large and vary in functionality and priority. Perhaps the most distinctive trait of this kind of robots is that, in order to perform their duties, they need to interact with people without any sort of human supervision. Thus, from a safety point of view the most critical applications would be those in charge of manipulating the robot's motors and dictating its movements. The robot must not, under any circumstance, be a potential source of danger to any human. Additionally, any control failure could cause damage to its expensive and delicate machinery.

This document will describe the process of development of a supervisor application that will monitor the most critical system processes. Keep in mind the supervisor, as its name suggests, will see that the system is informed with the state of the monitored software at all times. Should it come across any potential misbehaviour, it will report it to higher layer applications in charge of the robot's state machine so the proper measures are taken.

1.1 MOTIVATION

Robotics has been an area of interest for me since the beginning of university studies. However, I must admit the area of service robots was totally unknown to me when I first joined PAL Robotics. Having the chance of working for a company mainly dedicated to research and development to a field new to me was very attractive. After a couple of months of adaptation, I was chosen to develop and maintain critical real-time components. I found the idea of working in a very delicate environment –where you must be aware and in control of every detail– rather challenging. Finally, I decided to make my project about something directly related to this area.

1.2 OBJECTIVES

As mentioned before our goal is to develop a supervisor module that can monitor critical system components and applications. The list of objectives for this project, not only the ones that involve its completeness, but also the ones that seek our personal professional development, is detailed next.

- The programming language chosen will be C++, the same the rest of PAL Robotics' applications are implemented with. I am no expert in this language, but I will try to hone my skills with it with the progressive advance of the project. For this feat, two great tools will be at our disposal: the Standard Template Library (STL)[20] and the Boost libraries[1].
- Get to know Qt Creator, the Integrated Development Environment (IDE) chosen for this project, and all the tools it provides to improve our programming skills.
- Investigate into the Operating System (OS) or Operating Systems (OSs) used in the robot, specially the one involved with the critical components.
- Research the rest of technologies used in conjunction with the robot applications.
- Specification, design and implementation of the supervisor application.
- Specification, design and implementation of the tools needed by the users in order to have their applications supervised.

1.3 PLANNING

Our contract with PAL Robotics started on December of 2010 and will end by September 14th of 2011. Our planning will try to make use of all the available time to fit the research, development of the project and writing of the required documentation. Around four weeks of holidays are also taken into account into the planning. Even though we will work eight hours a day, keep in mind that our arrangement with the company is to dedicate four hours to our project and the remaining time to PAL Robotics specific activities. Though sometimes both may be the related, we can not always guarantee this. As a consequence, the periods established for the different tasks are conservative in general.

The Gantt diagram detailing our planning can be seen in Fig. 1.

The first part of our project implies a research phase, where we will get to know all the elements related to us: the real-time application concepts, the Xenomai OS, the Orocos framework and the already

existing applications of PAL Robotics which can be seen as elements of the "Research" task.

The project coordinator encouraged us to follow the typical steps for developing any project: specification, design and implementation before going straight to coding. So, first, the goals for the project will be discussed and the specifications defined. After this, we can proceed with a first draft of the project's design starting from the most basic components first until the whole project is designed. See the "Design for the OroSupervisor project" task for more details.

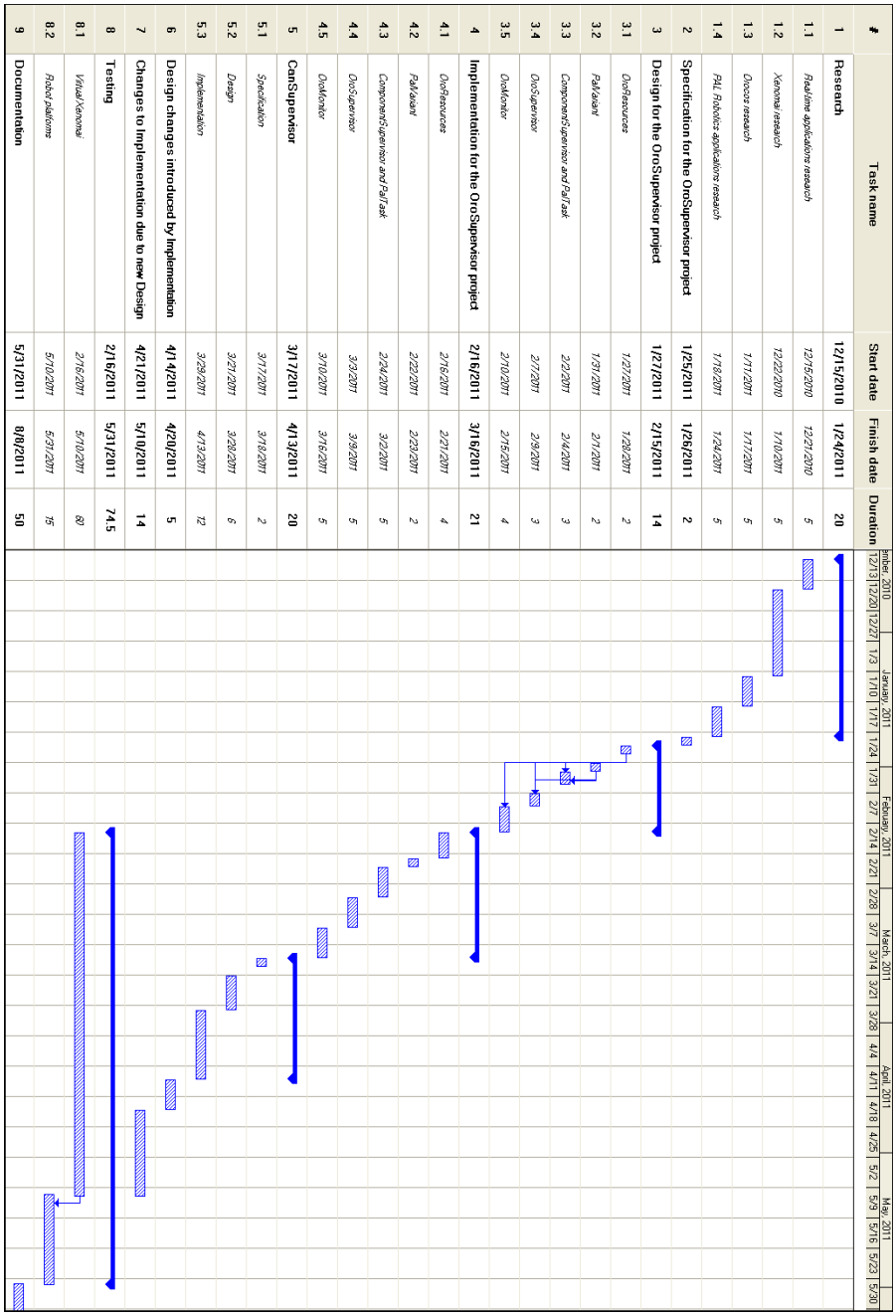


Figure 1: Planning

At this point, we can finally start implementing our design. For each module stated in the design we introduced the time needed for developing it. However, it is typical to find design errors while in the implementation phase. These will cause us to introduce design changes which will in turn cause more implementation changes.

Once the supervisor project has been finished we will develop a new specific supervisor for the CAN bus. It will be very similar to the first project and in fact will take advantage of most of it. The specification, design, implementation phases are contemplated in the "CanSupervisor" task.

The time that may be spent in this type redesigning is reflected in the "Design changes introduced by Implementation" and "Changes to Implementation due to new Design" tasks of the planning.

The "Testing" task reflects all the time that will be spent for testing the project in our local environment and the different robot platforms. Testing in the virtual robot environment will be performed in parallel while implementing the OroSupervisor and CanSupervisor and, running them on the actual robots, when both of their implementations are finished.

Finally, the remaining time until the end of our contract with PAL Robotics will be used for writing the project's documentation.

1.4 THESIS STRUCTURE

This document is structured in chapters, each dedicated to one specific topic.

The first chapter –which this section belongs to– is the Introduction. It opens by briefly explaining the roll of PAL Robotics in the robotics industry. After that the objectives for this project are enumerated and the planning for next few months detailed.

The second chapter lists the requirements for the project and the previous approximations in trying to fulfill them.

The third chapter covers the whole development process of the project. From the specification –explaining the technologies involved– to the implementation, passing through the design.

The fourth chapter explains the work environment and the tools at our hand. Then, the different testing platforms employed for checking the correctness of our code.

The fifth chapter is dedicated to an actual use case of the already implemented project, what shares with the original and the new introductions.

The sixth chapter makes the cost analysis of the whole project.

The final chapter concludes by analyzing the objectives completed and the future additions and improvements for this project.

2 PROBLEM

The applications that currently run in the robot can be divided in two large groups: applications that require real time and applications that do not. Our project will be mainly focused in the first group.

The real time applications tend to be more critical, such as the motors control components, and must be programmed in a special way or will not work as they are expected. In the case of motor control components this could be seen as the difference between a robot's arm smooth movement and other with rattling and sharp pauses. A more critical case would be a situation where the motor needs to stops in time to avoid a collision.

Knowing when one of these components is not behaving as expected has, thus, become a critical need for PAL Robotics. Moreover the OS employed by PAL Robotics does not offer any kind of monitoring or tracing tool.

However a typical instrumentation for monitoring will not work, as it could alter the end result of the application.

2.1 REQUIREMENTS

The following list contains the requirements that we set for this project, in accordance with PAL Robotics.

- A supervisor component that monitors the current state of other critical robot components and collects information and statistics about them in the least possible intrusive way.
- An interface the users can use to quickly instrument already existing components reducing code changes to a minimum.
- Using the collected information, the supervisor should be able to tell whether the component is behaving correctly or not and report its state.
- All information must ultimately be reported to the robot's Supervisor. See 2.2.1.
- Another supervisor with the specific task of for monitoring the components involved in managing the Controller Area Network (CAN) bus.
- PAL Robotics' software development must follow the continuous integration methodology .

2.2 PREVIOUS STUDIES

2.2.1 *The existing supervisor*

A supervisor for the system already exists in the robot. From now on, we will just call this the Supervisor, while ours will be the OroSupervisor since this is the internal name of the project anyway.

The Supervisor's main purpose is to report the current state of all devices and functionalities present in the system. Users can manually add tests to their programs. The environment is prepared in a manner that each application has a separate supervisor thread whose main goal is to run these tests. The tests must report the current state of the device or functionality they are related with, and the Supervisor must communicate the results to the user through a visual interface.

It is important to remark that the Supervisor is not real-time safe, i.e., performs tasks that can cause the component to lose possession of the Central Processing Unit (CPU). And this is expected, since applications developed without taking into account real-time safeness are most likely to be not real-time safe. This is the reason why the Supervisor is not suitable for monitoring the motor control components, or any other real-time component.

However, according to company protocols, all devices and functionalities must ultimately report to the Supervisor. The necessity to have something that acts as a bridge between real-time components and the Supervisor is clear then. This will be the main goal of our project: a specific supervisor, the OroSupervisor, for real-time components that reports their state to the Supervisor.

2.2.2 *CAN Statistics*

Since the beginning, the company has always required a detailed monitor of the CAN bus traffic. This is specially important for the developer of the components controlling the motors. For this reason, the CanStatistics element was created, even before having the need of a supervisor. CanStatistics is a resource for keeping track of CAN messages sent to and read from the CAN bus. The idea was to filter and classify all the gathered information and post it so that the correct behavior of CAN elements could be monitored.

Since CanStatistics was needed urgently, it was done in a rush and hard coded into the components that interact with the CAN bus itself. This resulted in a very difficult to extend code. Adding new statistics or filters demanded the user to add and modify variables in several places. Also, once the statistics were added, disabling them required commenting large chunks of code and recompiling. In conclusion, several bad alterations to a critical system component. Even worst, the

only way the user could access the collected data was by manually calling a method that printed it in the standard output channel.

As a consequence, when this project was devised, one of its goals was to move the whole CanStatistics out of the motors controller and made the necessary refactoring to make it able to report to the supervisor in a secure way.

3 SOLUTION

3.1 PROPOSITION

The project's proposal is a supervisor, the OroSupervisor, that gathers required information from the components. We will provide the users with means of instrumenting their code with the least possible changes so that the supervision occurs without them noticing. Besides, all system resources consumption should be kept to a minimum to avoid a large impact in the execution.

The OroSupervisor will pass all the collected information to a higher layer, where it will presented to the system administrator. The goal is, of course, that any user can tell whether a component is behaving correctly or not without having any knowledge of the OroSupervisor or the component itself.

3.2 SPECIFICATION

Before going into the OroSupervisor specification we need to define the environment in which it will run on. To begin with, the robot applications run under Xenomai, a Real-Time Operating Systems (RTOSs). Under this system, components form part of Orocos, a framework that provides robot control software. Finally, the information is published in the Supervisor explained in the The existing supervisor section. We will explain all of this elements next.

We have been talking a lot about real-time to this point, but have not really defined "real-time". Thus, lets first define a few basic concepts and then start enumerating the different elements that compose the system the OroSupervisor will form part.

3.2.1 *Basics concepts*

We will detail a few concepts now that will give us a better understanding of the kind of a environment the project will be develop in.

3.2.1.1 *Real-time applications*

Real-time applications have operational deadlines between some triggering event and the application's response to that event. To meet these operational deadlines, programmers use RTOSs on which the

maximum response time can be calculated or measured reliably for the given application and environment.

A typical RTOS uses priorities. The highest priority task wanting the CPU always gets the CPU within a fixed amount of time after the event waking the task has taken place. In a RTOS the latency of a task only depends on the rest of the tasks running at equal or higher priorities, all other tasks can be ignored. On a normal OS –such as normal Linux– the latencies depend of everything running in the system, which of course makes it much harder to be certain deadlines will be met every time on a reasonably complicated system. This is because preemption can be switched off for an unknown amount of time. The high priority task wanting to run can thus be delayed for an unknown amount of time by low priority tasks running with preemption switched off[13].

3.2.1.2 *Hard real-time and soft real-time*

An OS is said to be real-time if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed. Real-Time Operating Systems (RTOSs), as well as their deadlines, are classified by the consequence of missing a deadline.

- Hard: Missing a deadline is a total system failure.
- Soft: The usefulness of a result degrades after its deadline, thereby degrading the system's quality of service. Several deadline misses may lead ultimately to a system failure.

Thus, the goal of a hard real-time system is to ensure that all deadlines are met, but for soft real-time systems the goal becomes meeting a certain subset of deadlines in order to optimize some application specific criteria. The particular criteria optimized depends on the application, but some typical examples include maximizing the number of deadlines met, minimizing the lateness of tasks and maximizing the number of high priority tasks meeting their deadlines.

Hard real-time systems are used when it is imperative that an event is reacted to within a strict deadline. Such strong guarantees are required of systems for which not reacting in a certain interval of time would cause great loss in some manner, especially damaging the surroundings physically or threatening human lives. Hard real-time systems are typically found interacting at a low level with physical hardware, in embedded systems.

Soft real-time systems are typically used where there is some issue of concurrent access and the need to keep a number of connected systems up to date with changing situations[19].

Being the robot's motors controller a real-time application, our system is considered hard real-time since is imperative the motors synchronization is guaranteed at all times. Failures can cause motions to not be executed accordingly and goals to be lost.

3.2.1.3 *Typical real-time unsafe operations*

Any operation that can block the application by an unknown amount of time is not real-time safe, e.g., accessing a device, using the standard input/output channel and allocating/deallocating memory in the heap. In short, any type of instruction with an unbounded execution time or that results in a trap.

By default, we must suppose that any system call may be not real-time safe. This is normally the case for normal OSs. RTOSs provide safe system calls, but the specific documentation should be checked for each case. For example, taking a look at the Xenomai's Application Programming Interface (API)[22], a RTOS that we will cover later on, the act of consulting the information of a thread is always real-time safe while a sleep is always not real-time safe. In Xenomai, an operation that may trigger rescheduling is not real-time safe.

Typical programming language resources must also be used with care. For example, the vector container provided by the C++ STL is not real-time safe since its size is dynamic and the new system call used for allocating more memory is not real-time safe.

3.2.1.4 *Overrun*

When a real-time application surpasses its operational deadline it incurs in an overrun. The consequence of an overrun depends purely of the OS . Possible actions are throwing a signal or stopping the application altogether.

3.2.2 *Xenomai*

Xenomai is a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications, seamlessly integrated into the GNU/Linux environment[26].

Xenomai is based on an abstract RTOS core, usable for building any kind of real-time interfaces, over a nucleus which exports a set of generic RTOS services. Any number of RTOS personalities called "skins" can then be built over the nucleus, providing their own specific interface to the applications, by using the services of a single generic core to implement it.

Processes run in domains which contain some sort of entity capable of handling hardware or software interrupts. An interrupt is anything that can be trapped, including task switches, hooks, signals, etc.

As shown in Figure 2 on page 12, Xenomai has three domains:

- The primary domain which runs a real-time kernel.
- The secondary domain which runs Linux.
- An intermediate domain that is used as interrupt shield.

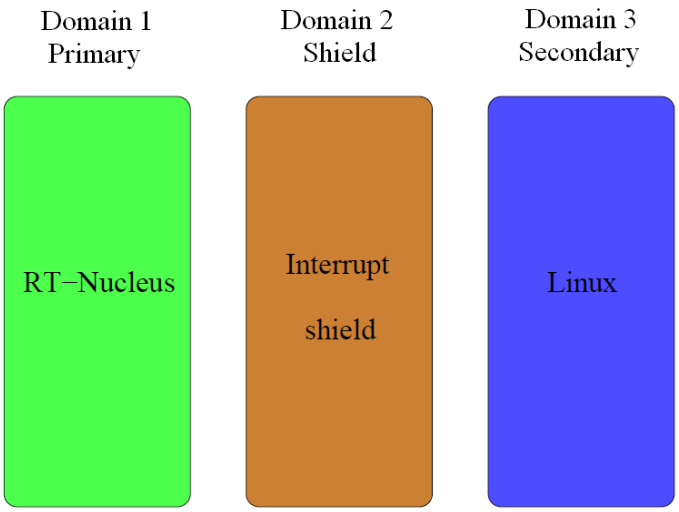


Figure 2: Xenomai Domains

Xenomai threads can run in either mode. Threads start in primary mode and stay there as long as they do not invoke any non real-time operation. If they do, then the thread switches to secondary mode. In Xenomai, an operation that is not deterministic is not real-time safe. They can be classified into two large groups:

- Those that can cause rescheduling, be it by a trap or an I/O to a device.
- Those with an unbounded execution time.

Each Xenomai thread has a priority between 0 and 99. Priority is used, among other things, to establish which thread will take possession of the CPU. The priorities used in the primary mode are compatible with those used in the secondary mode. Therefore, the thread's priority is kept even if it jumps from primary to secondary mode.

When in primary mode the thread is removed from the Linux scheduler and served by the real-time scheduler. Of course any thread in primary mode takes precedence over any other Linux process, even if the latter has a higher priority.

When a thread jumps to secondary mode, the real-time scheduler inserts it into the Linux queue and invokes the Linux scheduler. Xenomai threads in secondary mode still have more priority than normal Linux threads[27].

3.2.3 *Orocos*

Orocos stands for Open Robot Control Software and was an European funded project to write free control software for robots. It supports hard real-time RTAI/LXRT and Xenomai and normal GNU/Linux OSs [11].

The Orocos Real-Time Toolkit provides a C++ framework, or runtime, targeting the implementation of real-time and non real-time control systems. The Real-Time Toolkit library allows application designers to build configurable and interactive component-based real-time control applications.

The Real-Time toolkit allows components to run on real-time operating systems, Xenomai in our case, and offers component communication and distribution API and XML configuration.

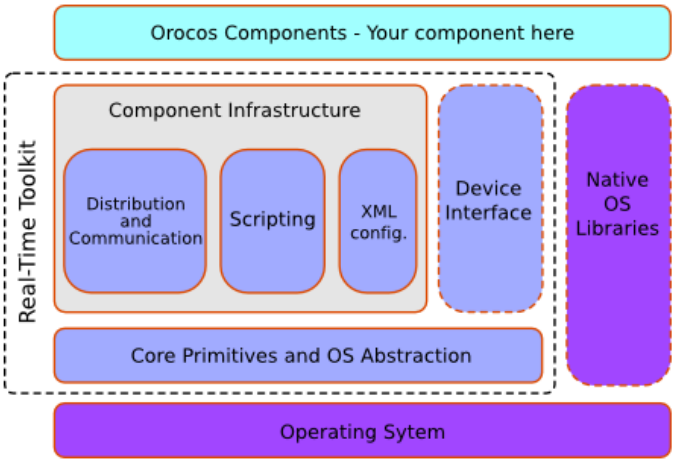


Figure 3: Real-Time Toolkit Application Stack

Each component is built using the TaskContext primitive: an active object which offers thread safe and efficient ports for lock-free data exchange. It can react to events, process commands, or execute Finite State Machines in hard real-time. It can be configured on-line through a property interface and XML files[10].

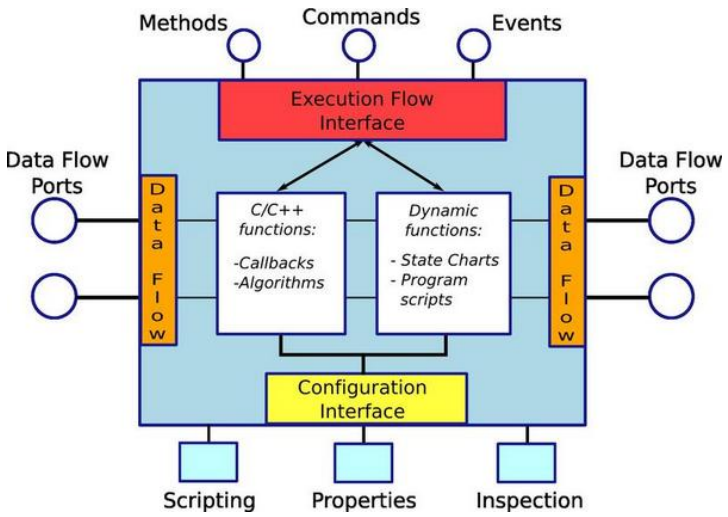


Figure 4: Schematic Overview of a TaskContext

The Execution Flow is formed by the processing of commands, methods and events, which call in turn user functions. The Data

Flow is the propagation of data from one task to another, where one producer can have multiple consumers.

When a TaskContext is running, it accepts commands or events using its Execution Engine. The Execution Engine will check periodically for new commands in its queue and execute programs which are running in the task. When a TaskContext is started, the ExecutionEngine is running. The complete state flow of a TaskContext is shown in Figure 5.

During creation, a component is in the Init state. When constructed, it enters the PreOperational or Stopped state. If it enters the PreOperational state after construction, it requires an additional configure() call before it can be start()'ed. The mentioned figure shows that for each API function, a user 'Hook' is available.

The user application code is filled in by inheriting from the TaskContext and implementing the 'Hook' functions. There are five such functions which are called when a TaskContext's state changes.

The user may insert his configuration-time setup/cleanup code in the configureHook() and cleanupHook().

The run-time application code belongs to the startHook(), updateHook() and stopHook() functions.

A TaskContext in the Stopped state may be start()'ed upon which startHook() is called once. If startHook() returns false, the start up sequence is aborted. If, by contrary, returns true, the TaskContext enters the Running state and updateHook() is called periodically by the ExecutionEngine. When the task is stop()'ed, stopHook() is called after the last updateHook() and the TaskContext enters the Stopped state again. Finally, by calling cleanup(), the cleanupHook() is called and the TaskContext enters the PreOperational state.

When a TaskContext is in Running state and a cycle of updateHook() surpasses the component's set period an overrun is reported. When a component reaches certain number of overruns Orocos stops it.

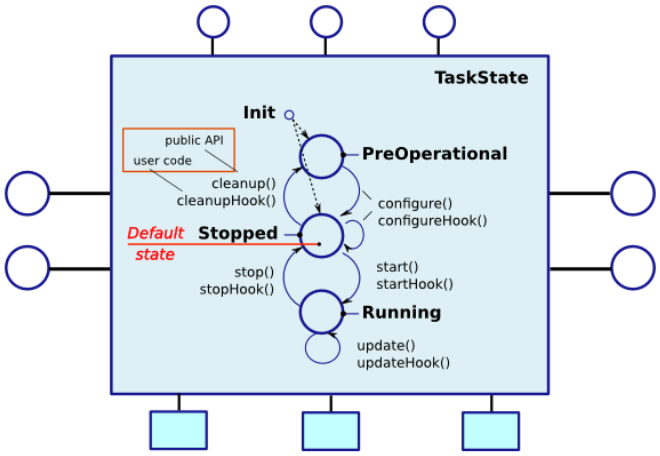


Figure 5: TaskContext State Diagram

The Data Flow is the stream of data between tasks. The data is passed buffered or unbuffered from one task to others. A Task can be woken up if data arrives at one or more ports or it can poll for new data on its ports.

Reading and writing data ports is always non blocking –therefor real-time safe– and thread-safe, on the condition that copying your data, i.e. your copy constructor, is as well.

The Orocos Data Flow is implemented with the Port-Connector software pattern. Each task defines its data exchange ports and inter-task connectors transmit data from one port to another. A Port is defined by a name and the data type it wants to exchange. Buffered exchange is done by BufferPorts and unbuffered exchange is done by DataPorts. Inter-process communication is done through the Common Object Request Broker Architecture (CORBA)[18].

Both can offer read-only, write-only or read-write access to the data. Also, you can opt that new data on selected ports wakes up your task by marking them as EventPort.

Orocos Methods resemble normal C or C++ functions, but they have the advantage to be usable in scripting or can be called over a network connection. They take arguments and may return a value. The return value can, in turn, be used as an argument for other Methods or stored in a variable. The Orocos Method template takes as argument the signature of the function to implement.

A task's Methods are intended to be called 'synchronously' by the caller, i.e. are directly executed like a function in the thread of the caller. They are used to calculate a result or change a parameter. Calling Methods is real-time but not thread-safe and should for a running component be guarded with an Orocos Mutex if it's functionality requires so[9].

Components need to be associated to an activity in order to be executed in the ExecutionEngine. These activities have a priority in a range from 0 to 99, just like Xenomai threads. They can run periodically or be triggered by an event –like the arrival of new information to a port of the component– or both.

Orocos also provides two schedulers in which activities must be located: a real-time and a non real-time one. Each activity runs in its own Xenomai thread and the Orocos scheduler is used to decide whether the thread will start running in primary mode, for the real-time scheduler; or secondary mode, for the non real-time scheduler.

A component with a real-time scheduler activity that suffers from a real-time loss will make the activity, the Xenomai thread, switch from primary to secondary mode and remain there until the current cycle finishes. A component that belongs to a real-time scheduler periodic activity will always be in primary mode at the beginning of each cycle, even if it had to switch to secondary mode in a previous iteration.

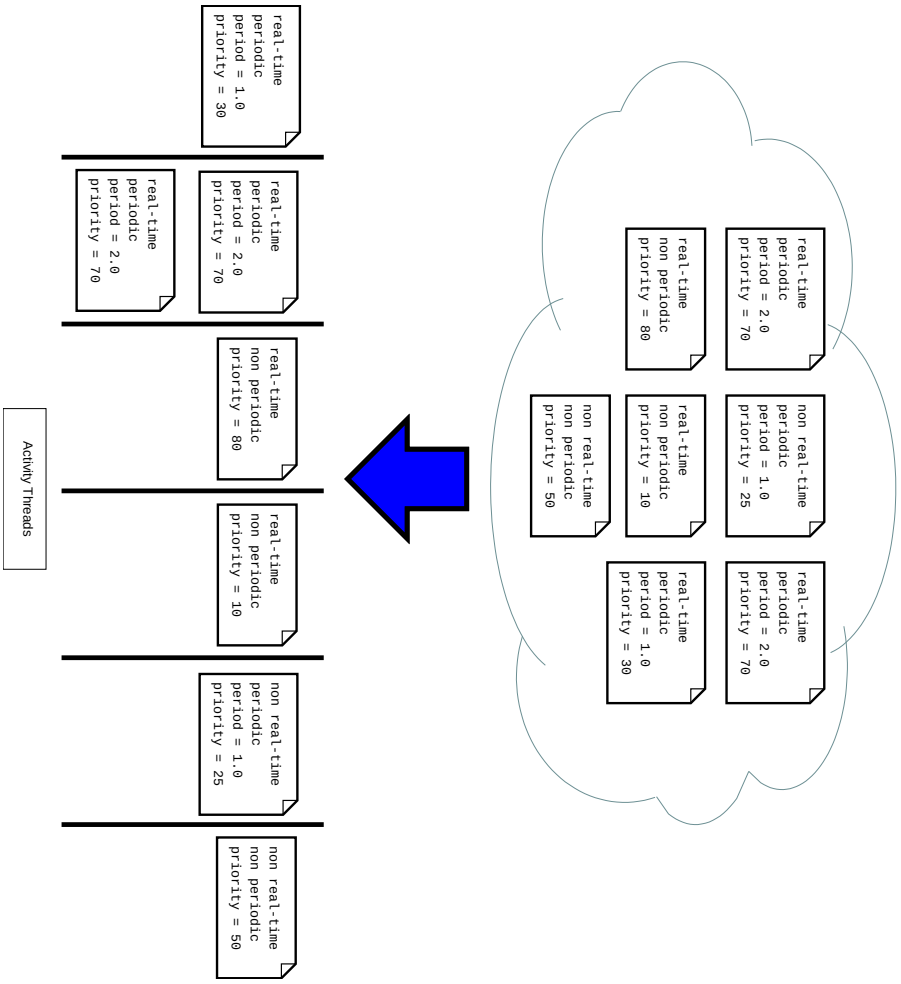


Figure 6: Orocos task scheduling example

An example of how several tasks are ordered in activities can be seen in Figure 6.

Orocos serialize different TaskContexts in the same thread if the activies associated to them have been parametrized with:

- Same scheduler, real-time or non real-time.
- Whether the activity is periodic or not.
- Same period, in the case of periodic activities.
- Same priority.

As explained then, periodic tasks with the same period, scheduler and priority are serialized in the same activity.

All the real-time safe applications developed by PAL Robotics are derived classes from the TaskContext primitive. This is important because here will be the main insertion point of the OroSupervisor to instrument said applications.

3.2.4 *OroSupervisor project*

It will consist of four parts. First, the instrumentation of the already existing components. This components will produce information that will be consumed by a new component, the OroSupervisor component. Then the information will be stored in a separate object, OroResources. Finally another new component, the OroMonitor component will read the information stored and pass it to the Supervisor.

3.2.4.1 *Instrumentation*

We will need to provide the users with tools they can easily use to instrument their already existent Orocos component. After some discussion with the project manager and the end users, it was decided that OroSupervisor instrumentation could be added in two different ways. See Figure 7.

- Automatic instrumentation: using a PalTask TaskContext as base class.
- Manual instrumentation: using a ComponentSupervisor object as class member.

The first method consists of a new class PalTask users can derive from instead of using the original Orocos TaskContext primitive. Since PalTask is nothing more than a TaskContext derived with the necessary infrastructure to add supervision, all the original methods provided by TaskContext are also present in PalTask.

The only changes users will have to incur, besides the class from which they derive from, are the original Orocos hooks. When using PalTask using a new series of hooks will be necessary. The need for this intermediate hooks will be explained in detail in the Implementation section. Even with this small issue, we consider that the use of PalTask is fairly simple and will bring the full supervision capabilities with small effort from the user side.

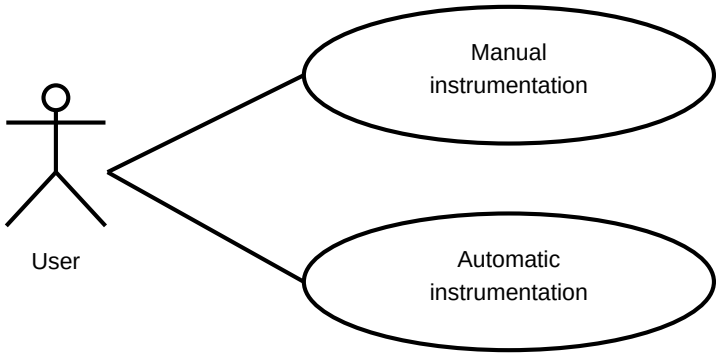


Figure 7: Instrumentation Use Case

The second method will consist of a new ComponentSupervisor object users will need to add the their already existing TaskContext components. This object will have a series of methods that will be need to be called inside the hooks in order to add supervision. However, we will also provide a wrapper object that will make the manual supervision a simpler task. The PalTask class will be implemented using the ComponentSupervisor.

Users that wish to customize which metrics and tests are done for their components will find the ComponentSupervisor fitting for their needs. If, on the other hand, they just want a quick way to add instrumentation to their code the use of PalTask is encouraged.

3.2.4.2 *OroSupervisor*

The new OroSupervisor component, that will also be a derived PalTask, is the component with the responsibility of periodically collecting all the information sent by supervised applications and store it so it can be publish later.

This component will run in the non real-time scheduler with a low priority, since it does not really need to be real-time safe to perform the specified tasks. Also, by this, we do not incur in the risk of taking away CPU time from more prioritary applications.

The OroSupervisor can receive information from real-time and non real-time components and save it without distinction. An effort in performing these tasks as quick as possible will be make so, again, the impact in the CPU usage is low. Remember that, even though OroSupervisor will run in a non real-time scheduler and not affect real-time components, there are other non real-time application whose normal behavior must not be affected. Specially if the priority of the latter is lower that the OroSupervisor.

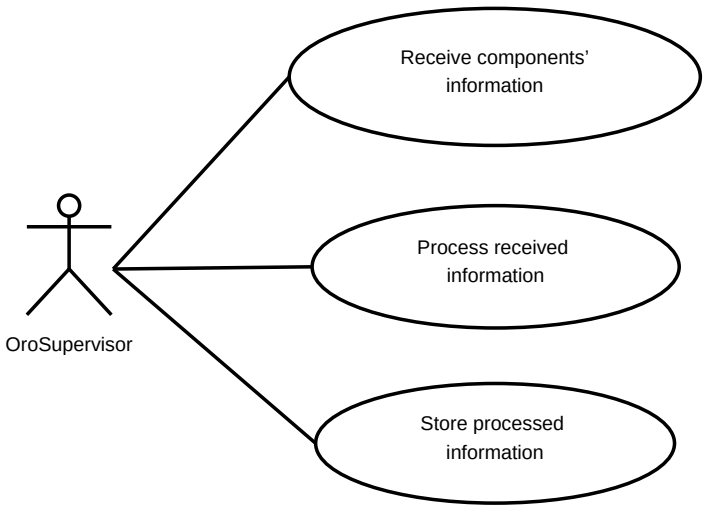


Figure 8: OroSupervisor Use Case

3.2.4.3 *OroResources*

All the information of each supervised component obtained by the OroSupervisor will be stored by the OroResources. It will be a database like object accessible from any part of the current process.

Given that there we will be at least one writer - the OroSupervisor - and one reader - the OroMonitor - concurrent safe access must be ensured.

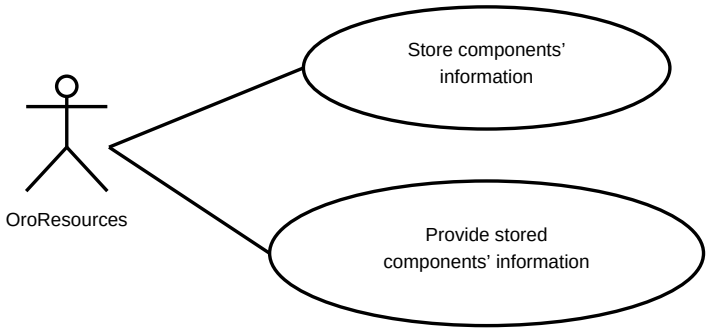


Figure 9: OroResources Use Case

3.2.4.4 *OroMonitor*

The OroMonitor will be another new Orocos component. It will prepare the necessary environment so the Supervisor can prompt all the supervisor components information to the user in a graphical interface.

Also it will report the current state of each component based on the collected information. For example:

- A component that is near to exceed its execution period will be reported as in a state of warning with a message that describes the problem. If no problem was found then the component will be in an OK state.
- A component that runs in the real-time scheduler and incurs in a real-time unsafe operation should be reported as in a WARN state so that the user may get a hold of the situation and takes the necessary measures.

The objective is that normal system administrators with no technical knowledge of the real-time and non real-time applications can report if a component is not behaving properly by just taking a look at the robot monitor.

For each component the following information will be reported:

- Task name
- Activity

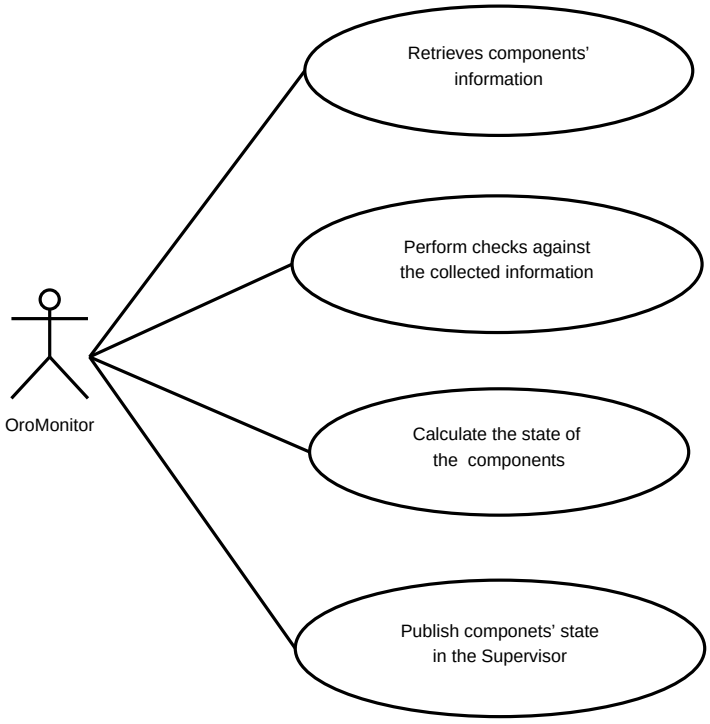


Figure 10: OroMonitor Use Case

- Priority
- Period
- Xenomai thread information
- Tasks sharing the thread
- Current machine state
- Current user execution time
- Current total execution time (user time + supervision time)
- Time between two consecutive cycles
- Overhead added by supervision
- Real-time losses
- Exceptions threw

For each component the following checks will be performed:

- Task initialized
- Overrun count.
- Task starvation, i.e., if the tasks is being executed at the set period.

- Exception threw
- Real-time lost

3.3 DESIGN

We will now proceed to explain the OroSupervisor project pipeline, i.e. the path the information follows, from the user's component to the Supervisor robot monitor passing through the OroSupervisor. The classes, structures and communication objects that will be need to be used will be also a part of this section. An Unified Modeling Language (UML) representation of the pipeline can be seen in Figure 11. Keep in mind, this is not the full class diagram of the project. To keep it simple, some details have been left out. Each scope of the project will be better described later.

First, users will need to instrument their Orocos components using one of the two methods explained in the Instrumentation subsection. The ComponentSupervisor object will create, in its initialization, an Orocos BufferPort for outgoing data, which we will call OutPort, and associate it to the component. After this it will register the component in the OroResources database in order to report its need for supervision. Figure 12 shows how the component will periodically push information about its state to the OutPort.

When the OroSupervisor object is created, it will access the OroResources database and for each registered component create a BufferPort for incoming data called InPort. Thus, a component will have one and only one OutPort - for as much as supervision is comprehend - while the OroSupervisor will have many InPort's. After this, it will proceed to connect all the component's OutPort to their respective InPort. The ports communication is managed automatically.

It was decided that for each supervised component only one port should be added in order to keep memory consumption low as possible. As was mentioned in subsection 3.2.3, Orocos ports are define by one and only one type. Thus, the type of information the BufferPort can carry must be generic so it covers all of our needs. In order to fulfill this goal a new generic type, the PalVariantType, is introduced. Our ports, be it InPort or OutPort, will transmit data of this type. It will be covered much more in detailed in the Implementation section, but for now we will just say that the PalVariantType is a "container" that can carry multiple types. This way we can have one port dealing with all the necessary transactions: strings, integers, structs or any other custom type.

One important limitation of Orocos BufferPort's is their fixed size, which makes sense; if they could change their size dynamically then they would not be real-time safe. This means that once the port is full additional arriving messages will be lost until some space is made in the buffer. Any component that executes more frequently than the

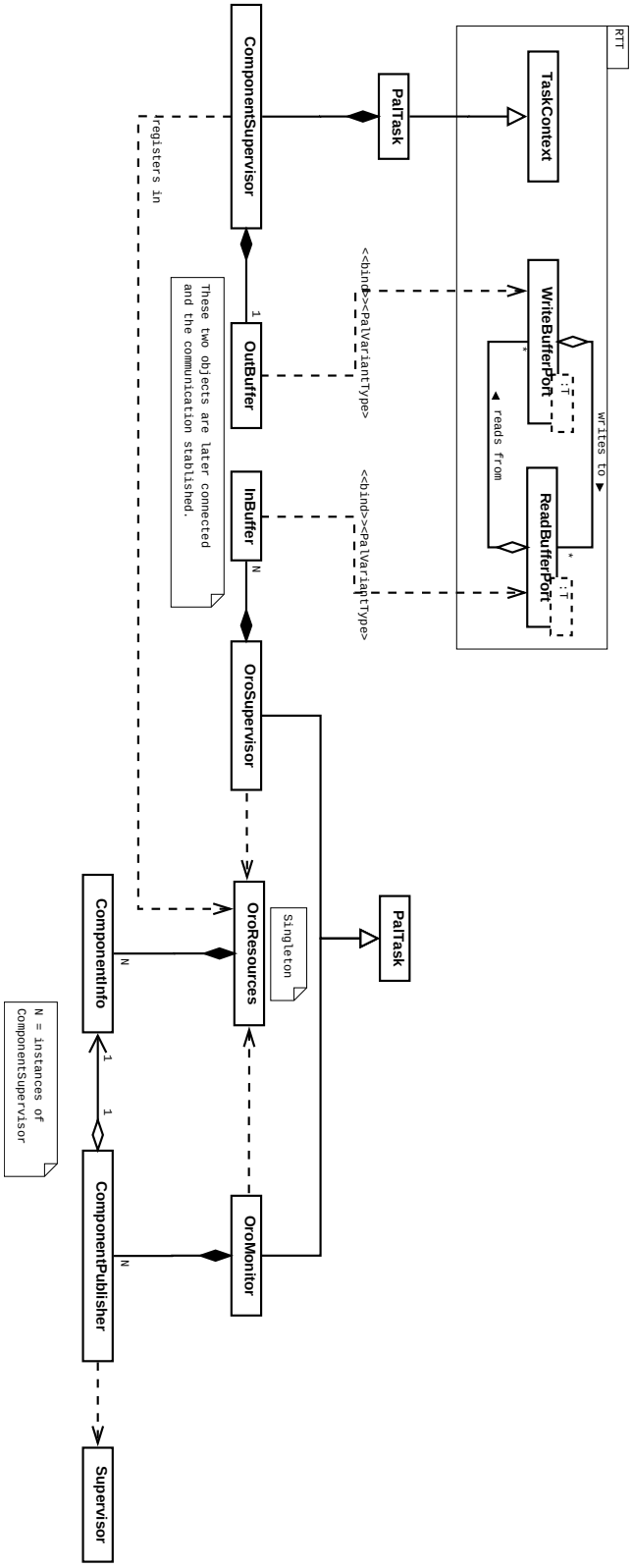


Figure 11: OroSupervisor Project Class Diagram

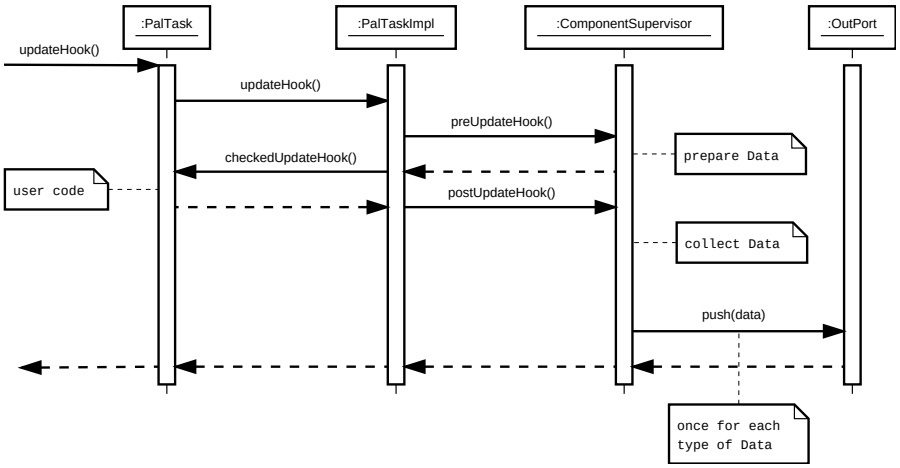


Figure 12: PalTask Sequence Diagram

supervisor is bound to fill its buffer sooner or later. However, the OroSupervisor empties all its InPorts in each cycle, so the next time the components cycle there will be room for pushing their data. In conclusion, some supervision information may be lost but some of it will always be reported and this is a trade-off we are willing to take.

After the component information arrives to the OroSupervisor in a PalVariantType message, the actual metric information is extracted from the container. In the following step, the OroSupervisor stores the metrics in the OroResources just like Figure 13 shows.

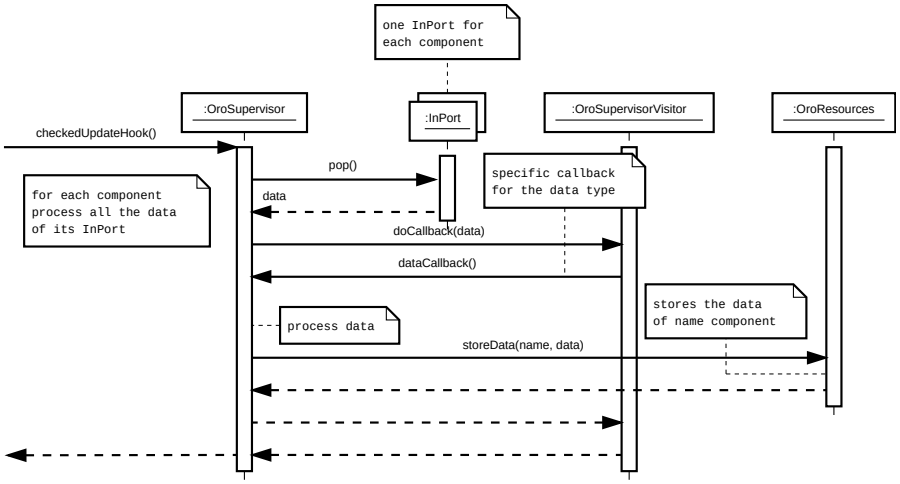


Figure 13: OroSupervisor Sequence Diagram

The OroResources object follows the Singleton programming pattern, which means one and only one instance of the object exists for each process[14]. While the aforementioned means access to the shared information is simplified, it also means we must ensure thread safety since concurrent access is a possibility. In our case, by OroSupervisor and OroMonitor.

Finally a separate Orocos component, the OroMonitor, transmits the information available in OroResources to the Supervisor. Internally OroMonitor creates a ComponentPublisher instance for each supervised component which in turn report their associated component data as shown in Figure 11. An example of this part of the pipeline can be seen in Figure 14.

The ComponentPublisher's have two main tasks. The first one is publishing the supervised components information, as already mentioned. The second is to report the current state of the component based on the collected information. For this the information of a component is passed through a series of Check's. Each one checks for the correct fulfillment of a defined rule based on the component's data. The final state is the result of all the Check's verifications.

An effort has been made so adding a new Check is a simple tasks. The base class is an interface which provides a series of methods the developer must implement. Then, the user just has to add the new Check to the list of checks performed by the ComponentPublisher class.

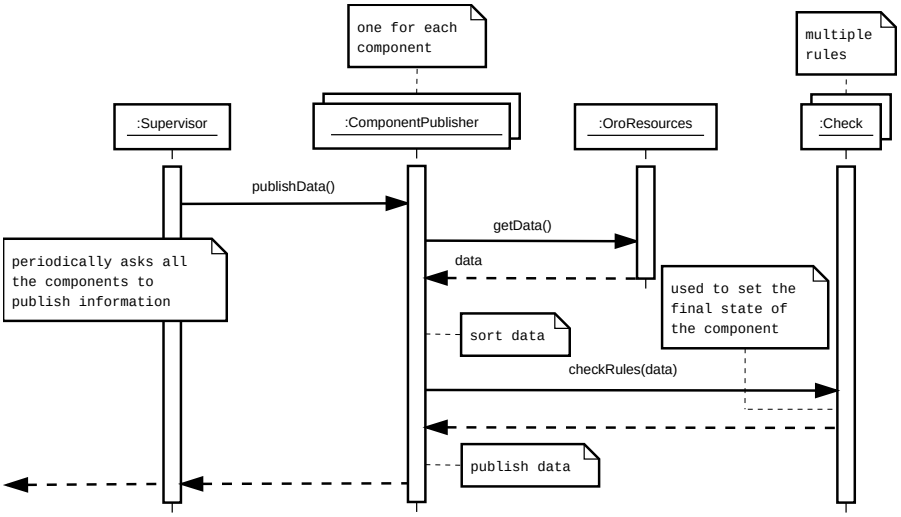


Figure 14: OroMonitor Sequence Diagram

3.4 IMPLEMENTATION

Since almost all of the code developed by PAL Robotics is programmed in C++, our project will naturally be too. Also, is the same language used in Orocos.

Besides all the custom tools developed by our company that are at our service, we will also use the Boost C++ library.

Boost is a set of free C++ source libraries. They work well with the C++ STL and are intended to be widely useful, and usable across a broad spectrum of applications. The final goal is that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already

included in the C++ Standards Committee's Library Technical Report (TR1) and will be in the new C++0x Standard now being finalized. C++0x will also include several more Boost libraries in addition to those from TR1. More Boost libraries are proposed for TR2[1].

3.4.1 *PalVariantType*

The first developed module for our project was the PalVariantType. As its name indicates is no more that our own implementation of the variant type present in Boost.

The Boost variant class template is a safe, generic, stack-based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner. Whereas standard containers such as std::vector may be thought of as "multi-value, single type", variant is "multi-type, single value"[2]. Using the Boost variant, we will able to generate a single type, the PalVariantType, that can contain all the types we want our Orocos InPorts/OutPorts to transfer. This way we reach our goal of having one port for all the transactions.

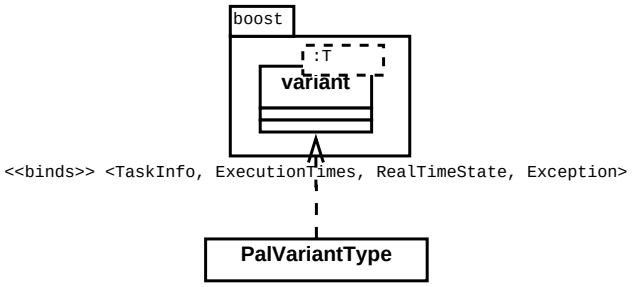


Figure 15: PalVariantType Class Diagram

At the time this documentation was written, four types were present in the PalVariantType:

- TaskInfo, stores information of the task such as the name of the Orocos activity or the period.
- ExecutionTimes, keeps track of the time taken to execute a cycle and the time between cycles.
- RealTimeState, just a counter of the number of time real-time has been lost by the component.
- Exception, details of any exception threw by the component.

Adding new types requires to just extend the PalVariantType definition, so, as a consequence we end up with a module that is easily scalable, a welcome characteristic in any code.

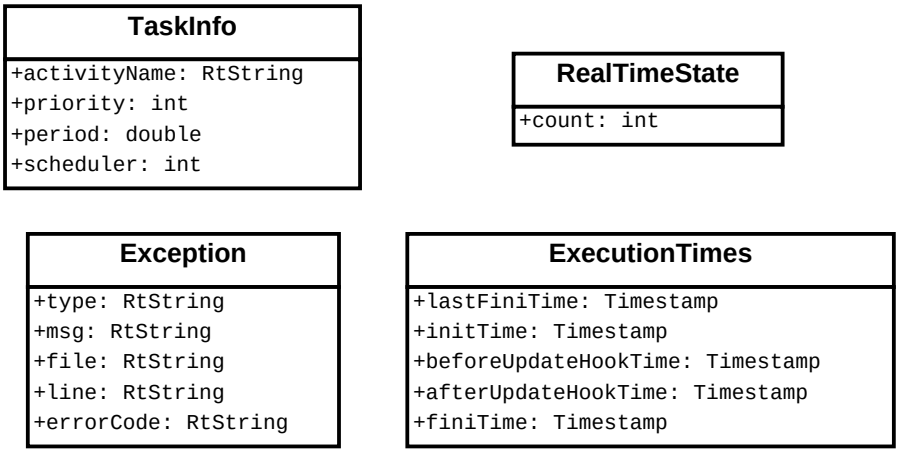


Figure 16: PalVariantType Types Class Diagrams

3.4.2 *ComponentSupervisor*

One of the two offered methods of instrumentation. The ComponentSupervisor is an object the user manually adds to its TaskContext as a member variable. The only parameter that must be passed on its construction is the pointer to the TaskContext itself.

As this will be an object the programmers use in their code and we do not know the nature of the component in advance, by default all that is ultimately executed inside the updateHook() must be real-time safe.

ComponentSupervisor has an Orocos BufferPort, the said OutPort, for transmitting data to the OroSupervisor. This object will be in charge of pushing the gathered data to the port.

In its construction, ComponentSupervisor registers the component it is associated with in OroResources. That way, the moment the application comes to form part of the system it can be supervised by OroSupervisor later on.

There are three methods the user must call in order to supervise their components.

First the init(), that accesses to all the information of the task and passes it to the OroSupervisor. The only prerequisite of this method is that it must be called from the thread of the component itself, i.e., in a configureHook() or startHook() of the TaskContext.

Then, in the updateHook() the user needs to use preUpdateHookSupervision() before any actions takes effect and postUpdateHookSupervision() at the end of hook. These two methods are in charge of sending the information needed in each cycle for tracking the component.

As it be can seen in Figure 17, the names of the methods clearly indicates where and when should be used. However, to make it even more simple, along with ComponentSupervisor another object is included, the SuperviseUpdateHook object. The purpose of this object is that the user only needs to instantiate it to gain supervision, it only needs

a pointer to the local ComponentSupervisor variable as parameter. Its behaviour is actually quite simple: SuperviseUpdateHook calls preUpdateHookSupervision() on construction and postUpdateHookSupervision() on destruction using the pointer to ComponentSupervisor. This follows a very common C++ idiom called "Resource Acquisition Is Initialization" (RAII)[12] which basically means that resources acquired in a function scope should be released before leaving the scope. The typical example of RAII is for ensuring the release of a lock in a Mutex. In our case, even though we are not actually acquiring any resources the principle is the same, a method should be called before leaving the updateHook() scope.

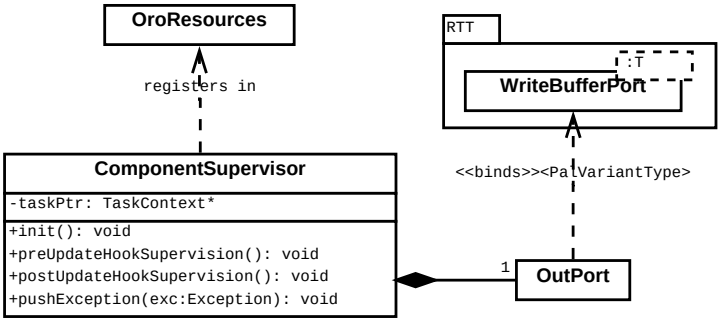


Figure 17: ComponentSupervisor Class Diagram

Of course, the user still needs to actually remember to implement the SuperviseUpdateHook in his configureHook(), but this is just a consequence of choosing manual instrumentation. Another side effect is that with ComponentSupervisor alone there will be not supervision of exceptions thrown by the task. The reason behind this will be explained in the next PalTask section.

All the necessary metrics that will be taken each cycle are prepared in the preUpdateHookSupervision(). The timestamp for the start of the cycle is taken and the detection for real-time losses is enabled. The latter is done with a tool developed by us for this project. It lets us reprogram a Xenomai signal to call a method when a switch from primary to secondary mode takes place. Whenever a switch happens, the counter for the affected thread will be increased. This way, if the switch happen while the component is being run we will know a real-time loss has happened. Later, in postUpdateHookSupervision() the signal is deactivated and the timestamps of the end of the cycle taken. Finally, all the collected information is pushed to the OutPort.

3.4.3 *PalTask*

Just like with TaskContext, PalTask is a class from where users can derive their components but with supervision already present. This way the user is spared of performing any of the steeps described in the previous ComponentSupervisor section.

Again, as described in ComponentSupervisor, all the code updateHook() will run must be real-time safe.

The only difference with PalTask is the names of the hooks the user overrides in his components. The new hooks follow the same checked[original 'Hook'] pattern. For example, instead of updateHook() now users need to implement checkedUpdateHook(). All the new hooks present in PalTask can be seen in the Figure 18.

The original hooks must not be overridden since they are used in PalTask for calling the necessary supervision methods and the customs checked hooks. To ensure users do not override the original hooks in PalTask some sort of constraint would be needed to be applied. However there is no simple way of protecting a function from being overridden in C++. Other programming languages do provide this kind of asset, like Java's final or C#'s sealed. It was decided that a stern warning in the documentation and the code would be enough. There are plans to find a better solution in the future though it does not fall in the scope of this project. See the Future work section for more information.

The actual implementation of the original hooks is not in PalTask as one may think but in a separate class called PalTaskImpl, following another common C++ idiom called "Pointer to Implementation" (Pimpl) or "Opaque pointer". Pimpl is a way to hide the implementation details of an interface from ordinary clients, so that the implementation may be changed without the need to recompile the modules using it. This benefits the programmer as well since a simple interface can be created, and most details can be hidden in another file[8], which is the case for PalTaskImpl. PalTask holds a pointer to its implementation as a local variable and its methods, the original hooks, call for their implementation in the PalTaskImpl object pointed by the pointer.

PalTaskImpl has two important members: a pointer to the PalTask it implements, necessary for calling back the users checked hooks; and an instance of ComponentSupervisor, for instrumenting the task. As the reader might have already guessed the steps detailed in the ComponentSupervisor section are implemented in PalTaskImpl so the users does not have to.

As was shown in Figure 12 in PalTaskImpl each original hook calls for its "preHook", then "checkedHook" and finally its "postHook". So, for example, updateHook() would call preUpdateHook() followed by the user's checkedUpdateHook() and finally the postUpdateHook(). The necessary code for supervision is introduced in the pre and post hooks, e.g., the calls to preUpdateHookSupervision() and postUpdateHookSupervision() are performed in the preUpdateHook() and postUpdateHook() respectively.

An additional advantage PalTask brings is the possibility to monitor the raise of exceptions. When a component throws an exception Orocos catches it, publishes an error message in a log and proceeds to put

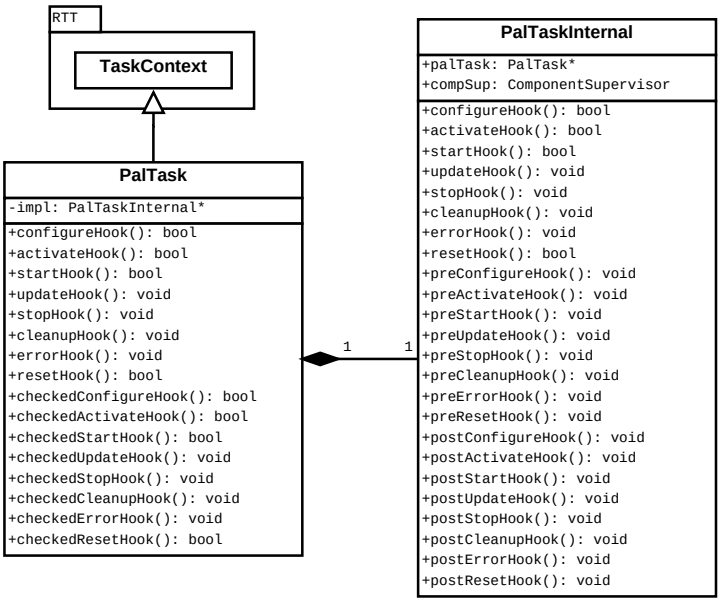


Figure 18: PalTask Class Diagram

the component in an error state. This behaviour is correct, otherwise one faulty component could bring down the whole system. Problem is sometimes founding this error message in a whole log can be tricky due to Orocos not publishing any information about the exception, not even which component caused it. PalTask tries to fix this or, at least, ease the experience for the users. In PalTaskImpl all hook scopes are surrounded by a try - catch clause. When the exception arises, first identifies it, then pushes the information to the OroSupervisor using the pushLogEntry() function provided by ComponentSupervisor and finally rethrows the exception. The latter is done so the system has the exact same behaviour as if the supervisor was not there.

3.4.4 *OroSupervisor*

The core of the project, OroSupervisor is a component that receives, process and publish all the information sent by the supervised applications.

OroSupervisor is, of course, another Orocos component. Even more, it derives from PalTask which means it can supervise itself.

When OroSupervisor is configured, through its checkedConfigureHook(), for each instrumented component creates an InPort and connects it to the said component's OutPort. The necessary data to create the connection, such as the components name, is retrieved from OroResources where components registered earlier on.

This leads us to our next topic, the order in which the all the system components are created, configured and started is important. If OroSupervisor is configured before any component exists, there will be no components to connect to. If components are started before Oro-

Supervisor is configured, the ports will be disconnected and the data pushed to them lost. For this project the order in which components are deployed in the system has been slightly changed. Before, all components were loaded, configured and started at the same time. Now, components are created at the same time. After this OroSupervisor is configured, then the rest of the components. Finally, OroSupervisor is started, then the rest of the components.

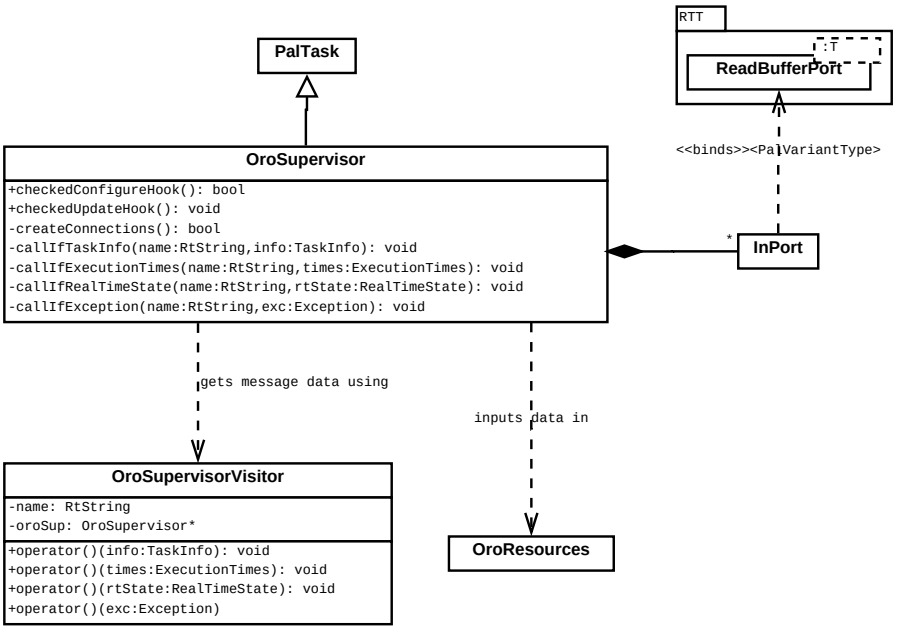


Figure 19: OroSupervisor Class Diagram

Once OroSupervisor is started, it will iterate over all the all the InPorts, pop their data and store it in OroResources in each cycle. Remember that the data arrives as a PalVariant message, so in order to get the original type we need to use the OroSupervisorVisitor, our implementation of the Boost visitor[2]. A clear example of how Boost variant can be used is shown in Figure 20.

OroSupervisorVisitor is an object that when called in conjunct with the incoming data, will call back one of OroSupervisor's methods specially designated to process that type of data. So, for example, if the type contained in the PalVariant message is a TaskInfo then OroSupervisorVisitor will call OroSupervisor's callIfTaskInfo() with the extracted data as parameter; callIfTaskInfo() in turn will take this information and put in OroResources.

As for the rest of the calls, callIfExecutionTimes() gathers all the timestamps taken each cycle and sets the minimum, maximum and average execution time values for each updateHook(), each checkedUpdateHook() and the time between consecutive updateHooks(). callIfRealTimeState() updates the counter of real-time losses and callIfException() the information of exceptions threw.

```
#include "boost/variant.hpp"
#include <iostream>

class my_visitor : public boost::static_visitor<int>
{
public:
    int operator()(int i) const
    {
        return i;
    }

    int operator()(const std::string & str) const
    {
        return str.length();
    }
};

int main()
{
    boost::variant< int, std::string > u("hello world");
    std::cout << u; // output: hello world

    int result = boost::apply_visitor( my_visitor(), u );
    std::cout << result; // output: 11 (i.e., length of "hello world")
}
```

Figure 20: Boost Variant Example

3.4.5 *OroResources*

OroResources can be think as the data base for all the collected information by OroSupervisor.

For each component registered, OroResources creates a ComponentInfo instance. ComponentInfo is an object to store all the information collected for said component: task information, counter for times real-time has been lost, the time the task takes to perform one cycle, etc.

As it was said in the Design section, OroResources access to one of its elements must be thread safe since more than one client may consult or modify it. Fortunately, Orocos provides thread safe lock free containers, the same that are used for implementing their ports. ComponentInfo will store all data component data in these type of containers.

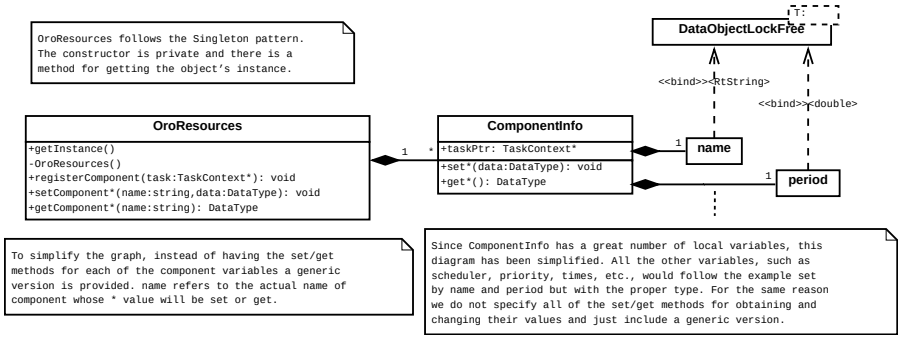


Figure 21: OroResources Class Diagram

3.4.6 *OroMonitor*

The final component of the OroSupervisor project pipeline, OroMonitor is an Orocos component, derived from PalTask, that offers the system Supervisor tests to publish all the information gathered with the OroSupervisor. An example of a real component's reported data is shown in Figure 22.

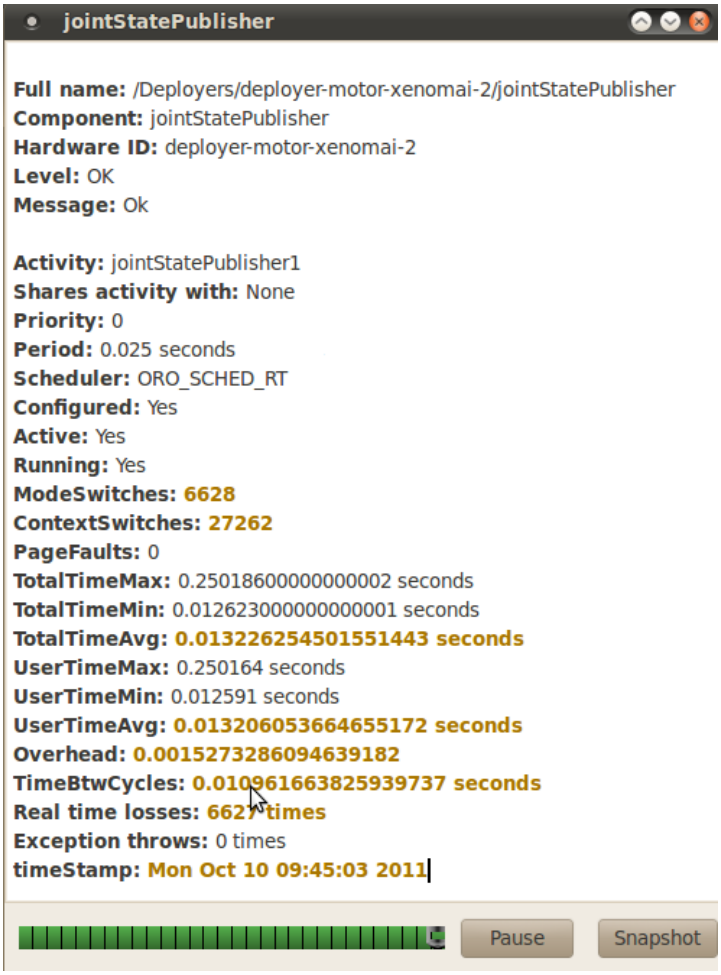


Figure 22: A component's reported data example

While in the OroSupervisor project clients, i.e. PalTask's, are responsible for reporting information, the Supervisor follows a polling model. That is, Supervisor will periodically call the tests the client has registered. Then the client implements this test, which will contain whatever information he wants to be published and also reports the result.

OroSupervisor will have one ComponentPublisher for each component that needs to report information to the Supervisor. When a ComponentPublisher is constructed it adds the local publishData() function as a new test for the Supervisor. At its destruction, the test is removed from Supervisor.

ComponentPublisher's publishData() will be the place then to write all the processed information to be published and the final status of the component. To input information the method add() is provided. It lets us input a key followed by a string with its value. For example, like Figure 22 shows, with add("Period", "0.025") we will see a "Period: 0.025" later when we check the information of the component in the robot's monitor application.

Apart from all the information coming from the component we also communicate other useful data to the user:

- Xenomai thread information. From the Orocos task we can get a pointer to the Xenomai thread in which it runs, and with it consult Xenomai for extra information of the thread. Mode and context switches are two examples of information directly provided by Xenomai.

- The current component's state machine information. It can be guessed with a pointer to the activity, which the component provides when registering to OroResources. So, we can publish if the component is configured, is activated and is running.

For setting the status of the test Supervisor provides three methods that are quite intuitive in what they do: setOk(), setWarn() and setError(). The status of the test showed in the robot monitor will depend on which of these methods were called during one cycle. Notice that once warning status is set it can not go back to OK and once error status is set it can not go back to either OK or warning for the rest of the actual Supervisor call. At first this may seem like a drawback, but it actually simplifies the casualistics when establishing the components status.

Once all the information has been published, a set of checks are run for the component. Each check is a derive object from the base Check class. Five checks are performed:

- InitializationCheck, in case the basic information of the component has not been set it is put in a warning state. This normally indicates that for some reason the component was not configured.

- ExceptionsCheck, sees if any exception has been thrown and in that case sets the component to error state.

- OverrunCheck, if the component's total execution time is coming near to its period the the possibility of an overrun is imminent and set the component to warning state.

- StarvationCheck, if the component is running and has not been called recently then is put in a warning state.

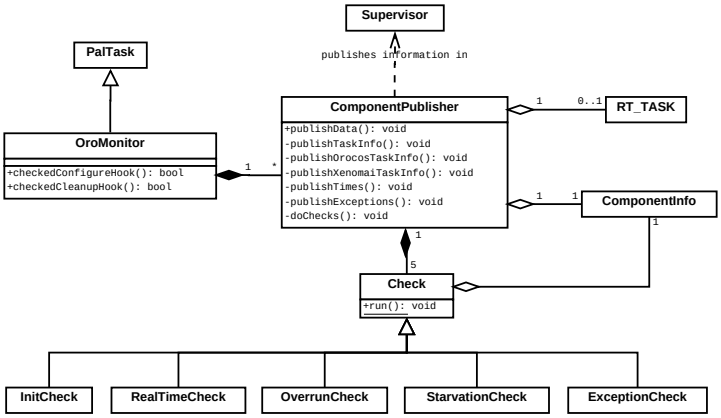


Figure 23: OroMonitor Class Diagram

- RealTimeCheck, if the component is running in the real-time scheduler and has incurred in real-time losses, then the component is set to warning state.

When setWarn() or setError() are used a message describing what happened is passed as parameter so it can later be visualized in the robot monitor.

If one check sets the component to error then no other check may overrun the said state. Even better, if several checks report the same state, the Supervisor will show all warnings and errors reported although the worst state will be kept. This is what we meant when we said that casualistics are simplified, no check need to worry of what the others do or report.

ComponentPublisher has a member vector with pointers to all of the checks. Since all of them derive from the same base class is simple to organize them in the same structure and even simpler to just iterate over this structure calling the run() method for each check.

4 TESTING AND EXPERIMENTATION

In this chapter we will explain first the development environment used for the OroSupervisor project and then the tools at our disposition. Finally we will describe our testing experience in different systems, such as a virtual Xenomai and all the different robot platforms.

4.1 WORK ENVIRONMENT

The OS running in our local machines is Ubuntu. Complementing it with a set of applications we will describe next we end up with a very complete environment for developing applications.

PAL Robotics does not impose any specific IDE to its staff, though recommends using Eclipse or Qt Creator, being the latter the most common choice among developers. After testing them both for some time, Qt Creator was our chosen IDE for this project. It does not really offer anything that Eclipse does not already have. However it feels "lighter" than the latter, specially when moving through the code. Even the simplest query takes a few seconds in Eclipse whereas results are displayed instantly with Qt Creator. In addition, it is fully integrated with our compiler and debugger making our developing experience much more pleasant. It also supports the use of our versioning software, though we decided to use this manually outside of the IDE.

What PAL Robotics does impose is the thorough testing of any code that is going to run in the robot. Even more, programmers are encouraged to develop unit tests in conjunction with their application. In software engineering, this methodology is called continuous integration, the continuous processes of applying quality control. Continuous integration aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development[17]. The CppUnit framework for unit testing is provided to fulfill this goal[6].

CppUnit lets us create independent tests for checking the correct behavior of a block of code, being it an object, method, etc. It also includes a complete set of macros to check the result of an operation. For example, we could assert against the Boolean result returned by a function, if it is true then the test continuous, if not the test stops and shows an error in the result of the unit test. When the unit test finishes running all tests a summary of the result is prompted at end, showing which tests failed and on exactly which assert macro. A complete unit test should try to cover all the possible cases of execution for an

object. This way, if we need to make changes but intend the behavior to remain the same we can run the unit test to make sure this is true. Even better, all the code is compiled during the night and all the unit tests run in the main server. If one of them fails a mail is sent to the developer responsible. So if an unintended change was introduced or a change was made and not tested at all we will eventually know.

Next is a list of tools that helped in our development.

Valgrind

Once our code is compiled running it is the next logical step. We must make sure that our application runs without issues and releases all the system resources when it completes its execution. Typical problems we will come across with are segmentation faults and memory leaks. Segmentation faults normally occur when trying to access a memory space we do not have permission to. Memory leaks are the result of not freeing resources we take. Unlike other programming languages, in C++ there is nothing like a garbage collector so we must release the resources ourselves. For solving these kind of problems one great tool is recommended: Valgrind[15].

When executing our application with Valgrind, it will instrument our code, run it and monitor it. When a segmentation fault arises it will stop the execution and show the user a back trace of the sequences of calls that produced the not permitted operation. This works also for uncatched exceptions.

It will also prompt errors when reading not initialized variables and this is the correct approach. A not initialized variable might not always cause a misbehavior, but there could be cases when it does. Even worst, when this cases happen finding the source of the errors is a time consuming task that requires the use of the debugger. Valgrind saves us from the trouble and points directly to a possible cause.

When execution ends and all object are destroyed, Valgrind will point those resources that lack a any kind of release. In normal execution we would not even notice this and leave the system with a chunk of memory allocated and not used at all. A common example of this is creating a new object and assigning it to a pointer. In destruction, the pointer would be freed but the actual object referenced by the pointer will continue to exist. To solve this, we should delete the object pointed manually or use any kind of smart pointer instead of a plain pointer, like the auto_ptr provided in the C++ STL.

As a good practice we have learnt that whenever is possible we should always run applications with Valgrind.

The only drawback of of Valgrind is that, at least for now, only runs under Linux. There is no support for Xenomai[25]. When developing an application for Xenomai, our procedure will involve compiling it for Linux and test it fully with Valgrind. We have the certainty

that, to a certain extend, a fixed applied for a Linux application with also work for a Xenomai application. So, we can assume that the application will run without errors on Xenomai, unless new errors arise or the behavior changes, which is of course a possibility since the OS is different. When this happens, we will not have other choice but resorting to the debugger.

We must take into account that our application will run significantly slower with Valgrind. This is an accepted disadvantage, Valgrind is made for finding execution errors not improving performance. For improving performance, we have Callgrind, which is used in conjunct with Valgrind and we will explain next.

Callgrind

As mentioned in the previous section, even though Callgrind is not exactly a debugging tool it does form part of Valgrind. And, since it will perform an active roll in our developing process we decided to add it to this chapter.

Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller-callee relationship between functions, and the numbers of such calls[5].

To use Callgrind we just need to run our application with Valgrind and enable some extra options. At program termination the profile data is written out to a file. After that we can have access to the presentation of the data and an interactive control of the profiling. The data file contains information about the calls made in the program among the functions executed, together with Instruction Read event counts.

Once our program is stable enough is time for trying to improve its performance. We will make use of the famous 80-20 rule. In a computer program this means that that 80% of the execution time is spent executing 20% of the code. In conclusion, we should focus most of our efforts on those parts of the code that are executed more often. On the profile presented by Callgrind we can visualize how many times a function is called during the execution, even more, it can present the number of times in relation to the rest of the calls. In short, by understanding the output offered by Callgrind we can know exactly which parts of the code to check first for possible performance improvements.

To understand the last concepts better we have prepared a small example extracted from the test phase of the project. The details of said phase will be explained later. The experiment consists of several dummy instrumented Orocos components running along with the OroSupervisor. The test is executed with Valgrind and the generation

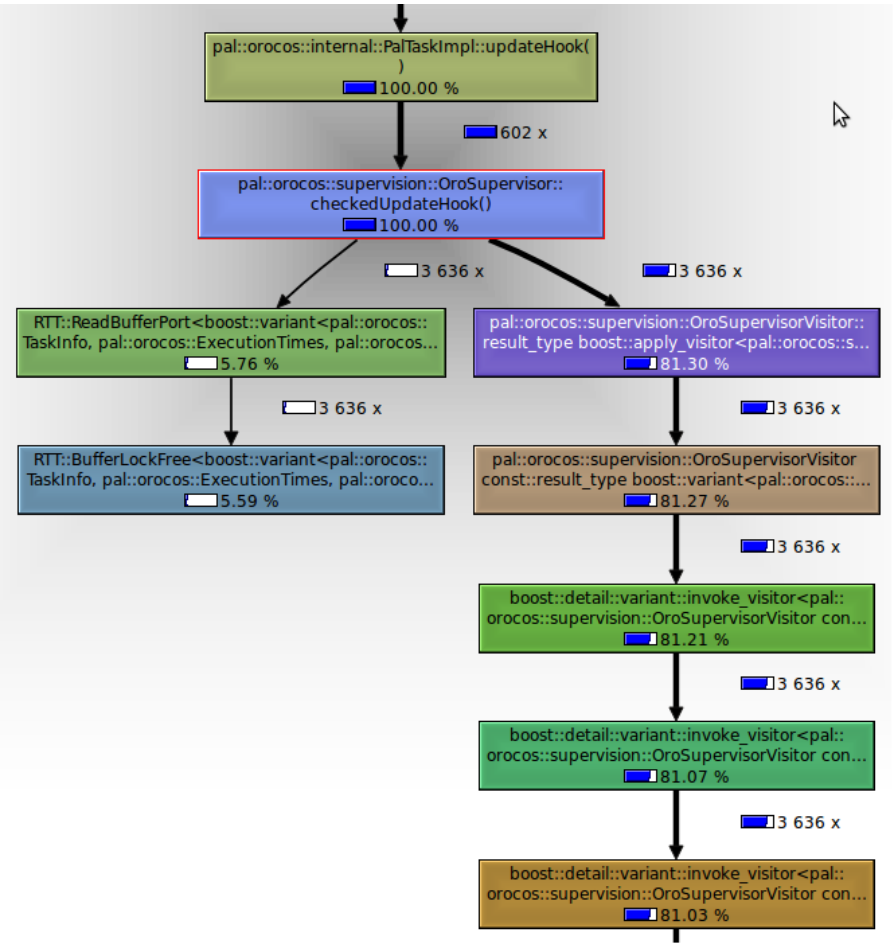


Figure 24: OroSupervisor Call-graph

of the profiling Callgrind file has been enabled. Finally, Figure 24 shows a part of the call-graph obtained for the OroSupervisor. The supervisor was programmed to be executed with a periodicity of 0.1 seconds and the total time of the test was one minute, resulting in 600 executions. This number is not far from the 602 calls showed in the graph. The error is understandable since Valgrind –like every instrumentation utility– does introduce some noise. Finally, we can appreciate that when executing the OroSupervisor's updateHook() most of the time is spent extracting the contents of a PalVariant message so, if our target is to improve the performance of this specific class trying to make this part more efficient would be the first step.

GDB

The GNU Project Debugger (GDB), is an application that allows the user to see the execution flow of another program or what it was doing at the moment of a crash[7].

It has lots of options for running the program, like the possibility of ignore any system interruption. We can execute the program in a step by step basis or run between breakpoints previously set . We can check the backtrace of execution and the values of the local variables at any moment. GDB can even attach to an already running program to check its state, perfect for situations where abnormal behavior arises during a common execution and we want to know what happened. We can navigate through all the running threads of a multi-thread application.

Qt Creator integrates GDB in its IDE, allowing us to write, compile, run and debug the programs from the same environment. It does not really implement all of the GDB capabilities, though enough of them are present so as to save us from the hassle of running the debugger from the standard Linux terminal.

It can run under any of our developing OSs, making GDB our main debugging tool for Xenomai applications.

4.2 TESTING

Through the duration of the whole project, several testing platforms were used. In our local machine besides Ubuntu we also had a virtual Xenomai. As for the robots, three different platforms were at our disposal.

4.2.1 *Testing in local environments*

As explained at the beginning of this chapter, all software must be tested first in a Linux environment, even if their final purpose is to be Xenomai applications. Once testing under Linux is complete, is time to move on to a platform that is closer to reality. We can achieve this goal by running a Xenomai virtual image, that is an exact copy of the image used in the robots, in a virtual machine.

Adding only few modifications we can compile and run our tests in Xenomai. Even better, we are left with a unit test code that does not need any more modifications to run either under Linux or Xenomai. Setting the correct compilation flag will produce the necessary binaries for any of these OSs. That means that now, we can add the "Test under Xenomai" step to our continuous integration cycle.

Xenomai provides two system files, see Figure 25 and 26, that contain updated data of the current running real-time threads[24, 23]. Useful information like:

- The number of switches to secondary mode incurred, i.e., possible real-time losses.
- Use of CPU and on which CPU is it running.
- Priority, period and timeout.

```
CPU  PID    MSW        CSW        PF    STAT       %CPU  NAME
  0  0      0          173991595  0     00500080   70.1  ROOT/0
  1  0      0          0          0     00500080   99.8  ROOT/1
  0  3851   897        2115       6     00300380    0.0  MainThread
  0  3997   1263979    14312845   0     00340184    5.0  TimerThreadInstance
  0  3999   1265243    16038258   0     00340184    5.0  TimerThreadInstance1
  0  4000   0          13601314   0     00300182    5.8  NonPeriodicActivity
  0  4002   0          5734798    0     00300182    1.6  NonPeriodicActivity1
  0  4005   0          4886633    0     00300184    0.3  TimerThreadInstance2
  0  4009   0          15843126   0     00300184    1.4  TimerThreadInstance3
  0  4010   1          12679194   0     00300184    0.3  motorThread21
  0  4011   1          12675998   0     00300184    0.3  motorThread22
```

Figure 25: Status Xenomai Information

```
CPU  PID    CLASS  PRI      TIMEOUT    TIMEBASE   STAT       NAME
  0  0      idle    -1      -          master     R          ROOT/0
  1  0      idle    -1      -          master     R          ROOT/1
  0  3851   rt      10      -          master     X          MainThread
  0  3997   rt      80      12ms452us  master      Dt         TimerThreadInstance
  0  3999   rt      80      1ms204us   master     Dt         TimerThreadInstance1
  0  4000   rt      97      -          master     W          NonPeriodicActivity
  0  4002   rt      96      -          master     W          NonPeriodicActivity1
  0  4005   rt      94      2ms419us   master     D          TimerThreadInstance2
  0  4009   rt      93      2ms893us   master     D          TimerThreadInstance3
  0  4010   rt      95      609us      master     D          motorThread21
  0  4011   rt      95      719us      master     D          motorThread22
  0  4012   rt      95      919us      master     D          motorThread23
```

Figure 26: Scheduling Xenomai Information

- Status: runnable, delayed, waiting, etc.

Some of the thread's information is also added in OroMonitor component's reports.

Thanks to all the information provided by this virtual environment we will be able to fine tune our software before proceeding to test is in the robots.

In Figure 25, MSW is the abbreviation for mode switches, i.e., switches to secondary mode performed by the thread. With that information we can check if our real-time loss detection is actually working fine. Components may have real-time losses during creation, configuration or starting so having a number of mode switches superior to zero is not necessarily a sign of trouble. Also, remember that more than one component can run in the same thread –as mentioned in the Orocos section– so MSW refers to all the switches incurred by the totality of components present in the thread. In contrast, the real-time losses value published by OroMonitor refers to mode switches that happen only in the updateHook() and only for a set component. In conclusion, the value of real-time losses reported by OroMonitor is more accurate than the system MSW indicator. As example, if we go back to the OroMonitor subsection and check Figure 22, we will notice a difference of one between the mode switches and the real-time losses. This variation is caused by a known loss that takes action in the configureHook().

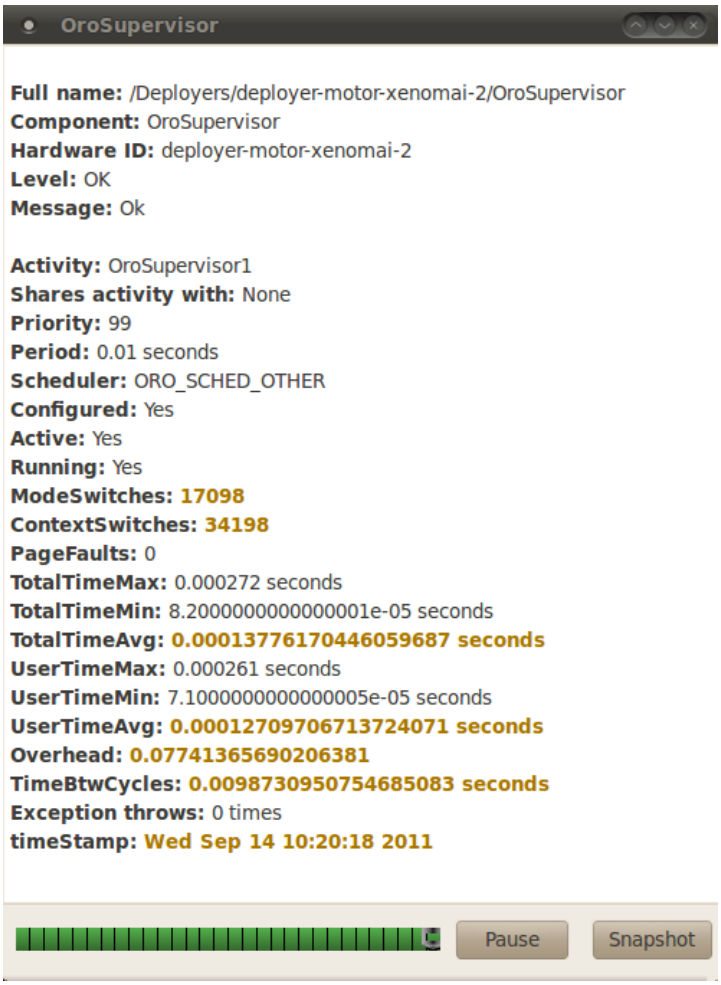


Figure 27: OroSupervisor Report

The status of a thread, STAT in Figure 26, gives us details of its health. For example, a permanent waiting status can be a sign of starvation in the case of periodic activities or that events are not rising for the case of non periodic activities. If a component is marked as in warning state for being starved, its thread is probably in a waiting status.

Once the virtual Xenomai environment is set, we can even run the robot monitor tool and check the output of our OroMonitor just like it in an actual robot. As example, in Figure 27 we can contemplate the output of OroSupervisor in an environment with several monitored dummy components.

The execution times of updateHook()s are reported and stored at the end of it, in the postUpdateHook(). One of these stored measures is the time taken to execute the whole updateHook(), i.e., the time taken to execute the custom checkedUpdateHook() implemented by the user plus the time taken to execute the pre and post 'hooks' of PalTask. The first approach was to just report the latest time registered. However this lead to several false negatives due to sporadic time peaks that

concede with the component being kicked out of the CPU by others with more priority.

We proceeded then, to store this information in a simple arithmetic mean that took into account all the reported execution times. In order to have an arithmetic mean that could be calculated efficiently we stored –in OroResources– for each component two values: the total number of measures taken and the sum of all execution times up to this point. That way, when a new measure of execution time for a set component arrived we simply added it to the accumulated time registered up to that point and increased the number of measures. Calculating the arithmetic mean is trivial then, we just need to divide this two values.

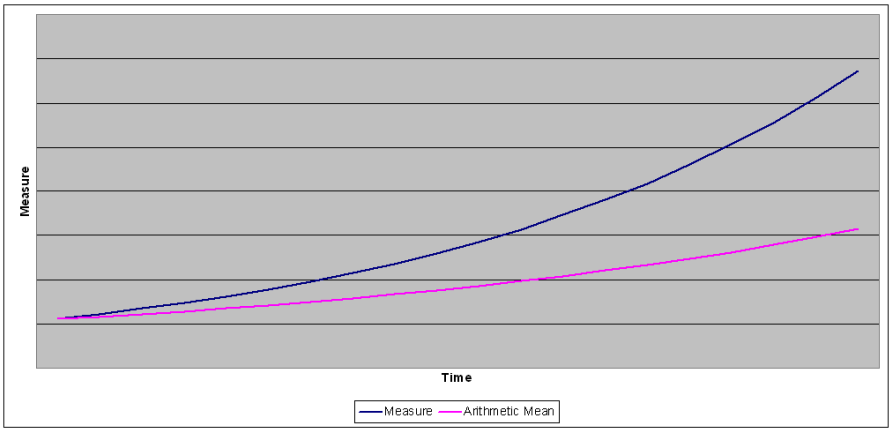
However when analyzing the execution times of the components we realized our previous implementation was flawed in some cases.

- If the execution time increases progressively, the arithmetic mean will never reach a value that mirrors reality. See Figure 28a.
- With the arithmetic mean there is no way to detect peaks that last only a few cycles. See Figure 28b.

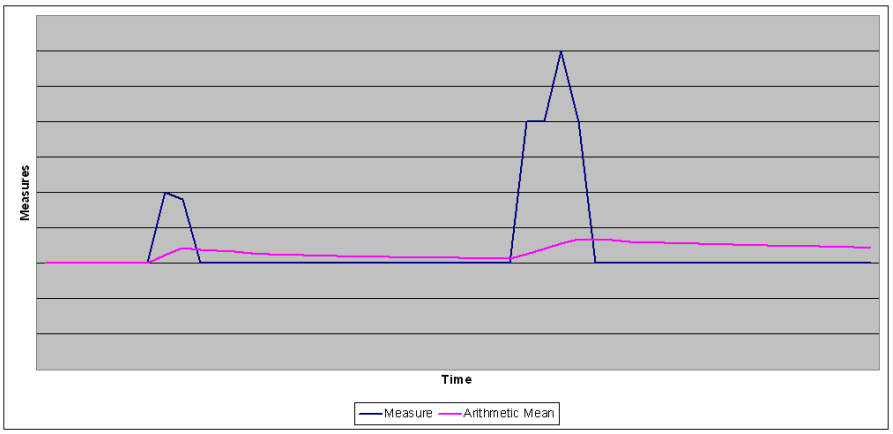
What we need, then, is a type of mean that gives more importance to recent values. That way, sudden increases can be detected and smoother so we do not report short peaks. Also our reported value will be closer to the real one. This type of mean does exist and it is called weighted mean[21]. Whereas in arithmetic mean all the measures have the same weight, in a weighted mean each measure has its own weight. Whether we want to give the most recent values more weight or not is our choice. For calculating the weighted mean we only need the previous weighted mean value, the new one and the weight we want to apply. Not having to store all the reported data makes this mean acceptable from an efficiency point of view.

As Figure 29a and 29b demonstrate, the weighted mean shows values much closer to the actual state of the component in both situations.

Finally we need to check if the actual overhead caused by the effect of supervision is significant for the total execution time of a cycle. The overhead is obtained by summing the time spent in preUpdateHook() and postUpdateHook() and dividing it by total execution time of the updateHook(). If we check the report of the previously seen components, see Figure 22 and 27, we will notice that the overhead represents 0.01% for the former and 7% for the latter. The actual amount of extra work introduced by supervision is constant so, the overhead will depend mostly on the period of the component. We believe the operations of supervision are as efficient as possible and should not vary the end state of component. Still, just in case, a Check for maximum overhead allowed was introduced to let us know if that is not the case.

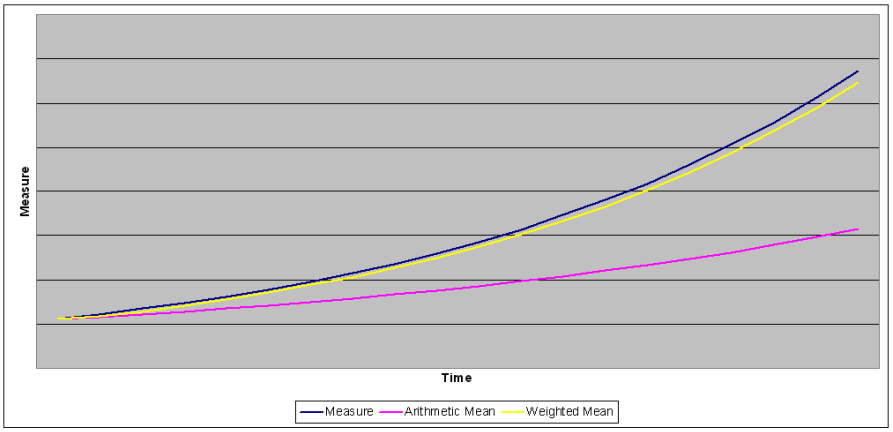


(a) Example 1

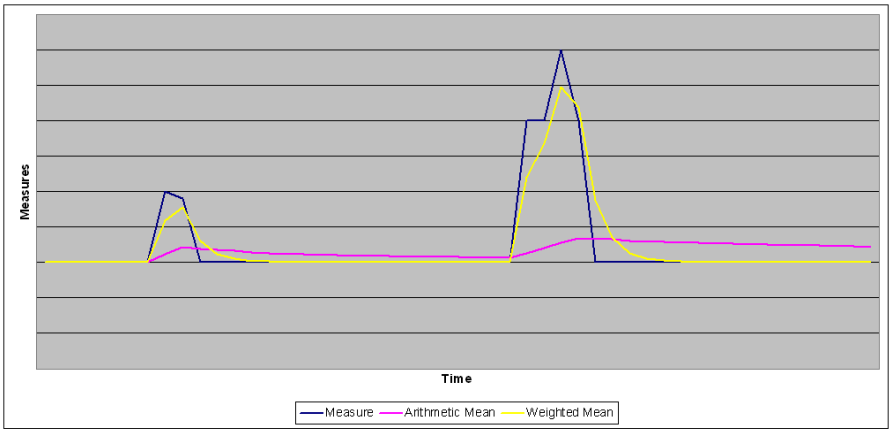


(b) Example 2

Figure 28: Arithmetic Mean



(a) Example 1



(b) Example 2

Figure 29: Weighted Mean Versus Arithmetic Mean

4.2.2 *Testing in robots*

The next and final step in the test process is run our project in the robot platforms. Currently PAL Robotics has three robot prototypes at our disposal:

- A mobile base that compromises a small portion of the totality of elements of the robot. The computer that runs Xenomai, two wheels and a few sensors. Because of this, some applications do not run in this platform, e.g. , the face tracking software is not run here; which is intended, without cameras there is really no sense in having it.
- REEM-H1 and REEM-H2. The actual humanoid robot platforms. They include all the necessary components to run the complete set of applications developed by PAL Robotics.

The first step will be, then, to try the OroSupervisor in the mobile base. This platform is like a middle ground between the actual robots and our local workstation. Perfect for a starting approach of running our software in a more real environment.

At this point we have included to the robot's deployment two new components: OroSupervisor and OroSupervisorMonitor. That means the are only two instrumented components for OroSupervisor to supervise. Even though we have demonstrated through testing that instrumenting a component does not actually have a great impact on its or the OroSupervisor's execution time by much, we would like to have at least one other user's component for this test. Preferably, a component that does run in all three platforms. After presenting our project to one of the developers of real-time applications of PAL Robotics, he agreed to have one of his components instrumented. Said component has already been shown in Figure 22 as part of the OroMonitor.

Testing should involve many different situations with all components running, with the robot idle or moving. For each of these cases we have a list of checks that must pass:

- The normal behavior of the running components remains unchanged.
- None of the components suffers from starvation.
- None of the components incurs in overruns.
- The supervised components report actual accurate information.
- The information reported by OroMonitor should match the one that can be obtained through the Xenomai system files.
- The CPU usage of our components should remain low.

If any of these fails then is time to go back to coding and fix the cause of the failure and again, repeating the whole development cycle: compile, test under Linux with Valgrind, test under virtual Xenomai and come back to the robot platform.

The first three checks may be hard to reproduce outside the robot, since is it not always possible to run them without it, and will require further testing in the platform. The last three checks can actually be successfully fixed while testing under Linux or Xenomai. For the CPU usage case, Callgrind will be specially useful.

The final step is to test the project in REEM-H1 and REEM-H2. We must take into account that both of these platforms are a scarce and very demanded resource inside the company. Thus our testing periods must be appointed and can not take too long, otherwise we would block the platform for the rest of the users. And that is why we have put so much emphasis in all the previous test steps, so when we arrive to these robots our program is as much free of bugs as possible and can run smoothly.

Of course this does not mean that we are exempt from any issues. The only difference between the REEM's and the mobile base are the number of elements present in the system, be them hardware or software, but just this can have a great impact in the outcome of the test. So, once again, the humanoid platforms will have to pass the list of checks described earlier.

5 CAN SUPERVISION

CAN is a message-based protocol, designed specifically for automotive applications but now also used in other areas such as industrial automation and medical equipment[16]. The communication between the robot's CAN nodes and the control computer is done through the CAN bus. In our case there is a motor in each CAN node, though there could be any other kind of CAN devices such as sensors, lasers, etc. As seen in the Objectives and Requirements sections, one of the goals of this project is to monitor this bus.

To realize this objective we will need a new kind of supervisor: the CAN supervisor, or CanSupervisor as we will refer to it from now on. The component supervised in this case forms part of the motors controller, one of the most critical systems in the robot, if not the most. For this reason our supervision should be applied with thoughtful care since it will be in the exact place of the CAN bus' I/O and a an approach different to that of the OroSupervisor needs to be taken.

In a nutshell the CanSupervisor will be what is commonly known as a "sniffer" though not directly connected to the CAN bus. It will check the incoming and outgoing CAN messages, without modifying their contents, and build statistics analyzing their contents. The final objective is to know the actual status of the motors, similarly to what is done in the OroSupervisor with the Orocos components. Additionally, the rest of components of the motors controller should stay unaffected. So, CanSupervisor is a specific supervisor for components managing the CAN bus. One important remark is that CanSupervisor is intended to be a developing and maintenance tool in contrast to the OroSupervisor that is a monitoring tool.

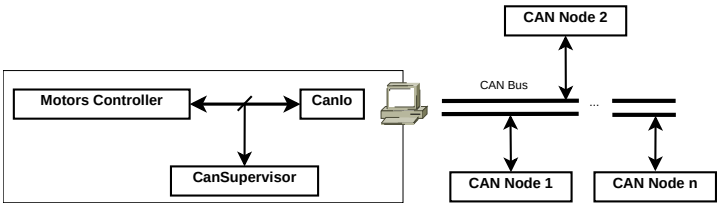


Figure 30: CanSupervisor Scheme

The CanSupervisor will take advantage of a significant part of the original OroSupervisor code:

- A new type CanVariantType will be used for communication instead of the PalVariantType. It will contain information of a CAN message.
- The InPort - OutPort connectivity will be the same.

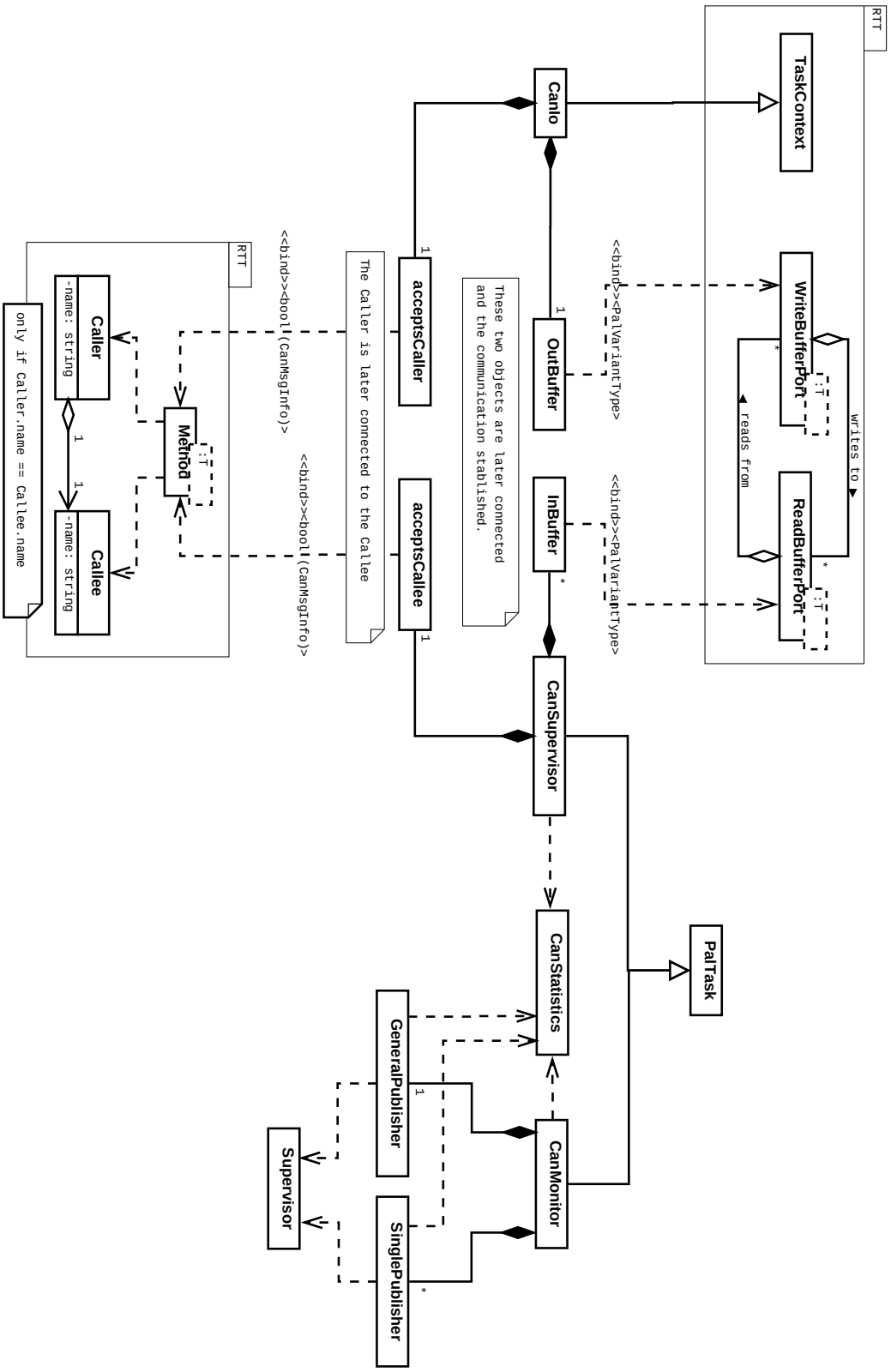


Figure 31: CanSupervisor Class Diagram

- For this new supervisor, we'll have only one supervised component: the CanIo, the component in charge of reading and writing on the CAN bus. Every time a new CAN message arrives or leaves the motors controller, a copy of it will be pushed to the OutPort for the CanSupervisor to analyze.
- Instead of a OroResources we will have a CanStatistics to store the collected data. Though the way in which they operate is the same.
- The CanSupervisor's behaviour is analogue to that of the OroSupervisor. It will classify all the information incoming from the CanIo and save it in the CanStatistics.
- Finally, the CanMonitor will be the only class that does not really resemble the original OroMonitor.

In fact, the final design of the whole CanSupervisor follows almost the same pattern as the OroSupervisor. Said design can be seen in Figure 31. There are a few changes that you may not understand yet and that we will explain later.

Even though the Boost variant was original designed to contain more than one type, CanSupervisor communication implements only one right now: the CanMsgInfo, which contains CAN message data. The diagram is shown in Figure 32. Of course, InPorts/OutPorts could use the CanIo type directly, but its left into a CanVariant in order to left it prepared for future additions.

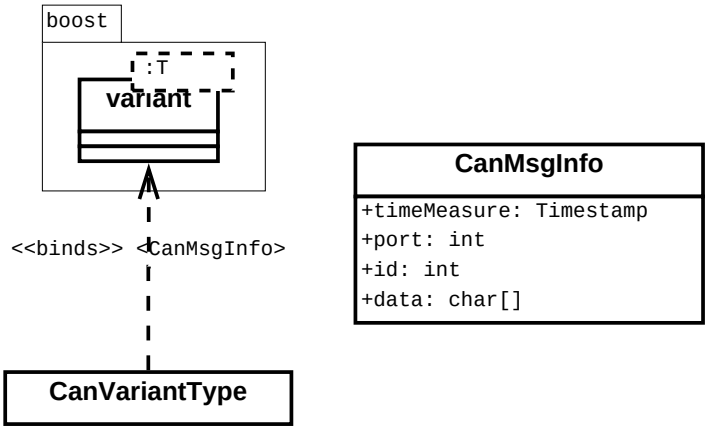


Figure 32: CanVariantType Class Diagram

As we said earlier, every time a new CAN message goes through the CanIo it will be pushed to the OutPort. This, however, results in a problem. The rate at what CAN messages income or outgo is far superior to the CanSupervisor's periodicity. As a consequence the buffers are bound to get full and CAN messages lost, just like with OroSupervisor and faster components data. CanSupervisor will still read the full contents of its buffer in each iteration, so there is always

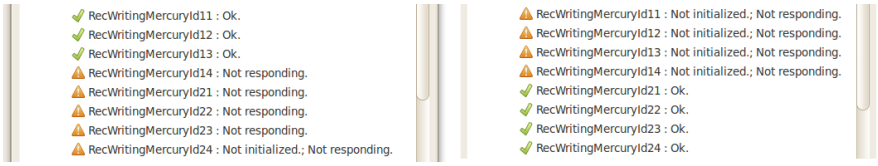


Figure 33: CanSupervisor Downsampling Example

fresh data in the next. However still remains the possibility of motors with more bandwidth usage leaving other devices unable to log their information.

An example of the previous statement is shown in Figure 33. The pictures –taken from the robot monitor application– display the outgoing data to a subset of the robot's motors, in this case corresponding to the arms. The figure on the left shows how CAN nodes with ID 11, 12 and 13 are working fine, while 14, 21, 22, 23 and 24 have the "Not responding" warning message which means they have not managed to report outgoing information in a while even though they should have. Whats more, CAN node 24 is not even initialized, i.e., not a single outgoing message has been received from it. This is a clear example of downsampling, and makes sense, since CanSupervisor has a periodicity inferior to that of the CAN bus. Also, components report in the order shown meaning the ones below are less likely to be able to make their information through. For the figure on the right a small experiment was made, the motors 11, 12, 13, and 14 were not enabled. As a consequence nodes 21, 22, 23 and 24 were able to report correctly now that the elements that were overshadowing them were not present anymore.

Though downsampling is a trade-off we have accepted, some additional effort has done in order to reduce its effects. Instead of normal pushes to the OutPort, CanIo will use filters provided by CanSupervisor with the objective of deciding which CAN messages should be pushed. With this we achieve two goals:

1. We reduce the possibility of motors unable to log their information.

2. We can focus our diagnostics in faulty elements instead of the full robot.

CanSupervisor will implement then an Orocos Method called acceptsCanMsg() that will take a CAN message as parameter and return a Boolean if it should be accepted or not. CanIo will connect to this Method and execute it for each CAN message. When the returned value is true CanIo will proceed to push the message information into the OutPort and skip the message otherwise. Looking back into Figure 31, we can now understand the changes between CanIo and CanSupervisor communication.

Each CAN message contains:

- The CAN node ID it is directed to or comes from.
- The CAN bus it is directed to or comes from, because there could be more than one.
- The actual instruction to the node. In case of the motors for example: move the motor to some position, report the actual current, stop the motor, temperature, etc.

So, with the right filter we could limit the supervision to the scope we are interested in. One filter could be, for example, only accept CAN messages of motors present in the arms and only those whose messages change the position of the motor. Right now only the default filter is actually implemented which accepts all messages. However all the filtering environment is ready and is easily extendable to create and use new ones. We decided to leave this task out of the scope of the project and move on.

The final CanSupervisor implementation is just like the one of OroSupervisor but with the new Orocos Method for communication with CanIo and the Filters used.

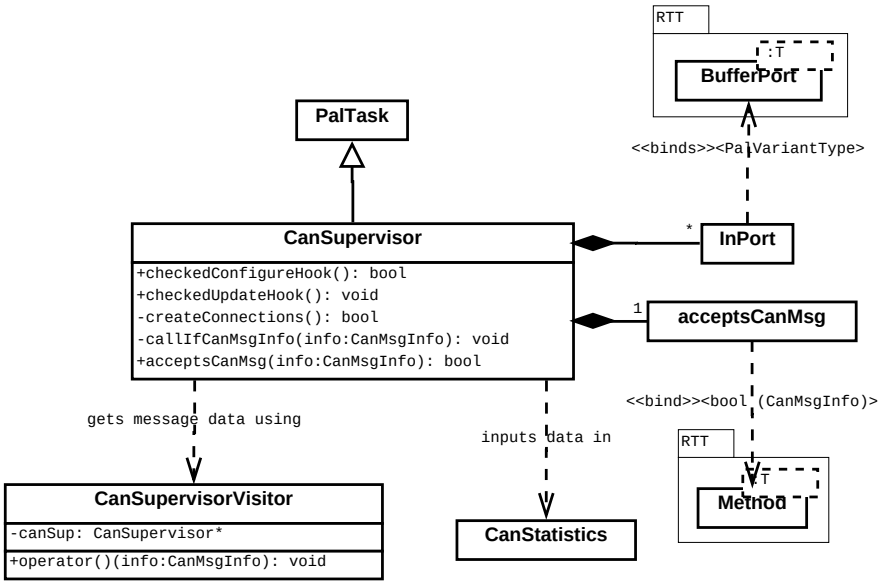


Figure 34: CanSupervisor Class Diagram

As mentioned in the CAN Statistics section of the previous studies, the original CanStatistics was used for collecting CAN messages information and was embedded directly into CanIo. In CanSupervisor the latter coupling has been eliminated by taking it out of CanIo and converting to an independent database like class, following the aforementioned Singleton pattern just like OroResources does. In the end, CanStatistics role in the CanSupervisor project is the same as OroResources in OroSupervisor. Also the class was improved to support safe concurrent access.

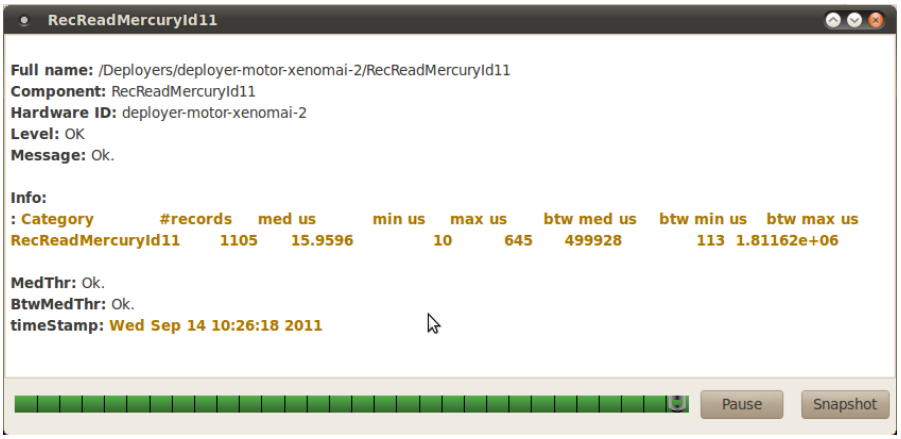


Figure 35: CanSupervisor Motor Information

While OroResources stores data divided by component, CanStatistics does it by Stats. Each Stat contains information from a specific node and its read or write data. There are also other Stats that group families of similar components, like all the motors from the same module or motors of the same brand. Currently, fourteen metrics are being track for each node, all of them related to timings:

- The number of incoming messages evaluated.
- The number of outgoing messages evaluated.
- The minimum, maximum and average time spent in sending a message to a node.
- The minimum, maximum and average time spent in reading a message to a node.
- The minimum, maximum and average time between two messages received from the node.
- The minimum, maximum and average time between two messages sent to the node.

This information is useful for checking the responsiveness of a node, its situation in the bus and how is it behaving in the context of the whole robot.

For example, in Figure 35 the current state of the CAN node 11 incoming information is shown. Again, the picture was taken from the robot's monitor application. The node's state OK since it has been reporting regularly and all of its statistics are under safe limits.

Finally, the component in charge of publishing information to the system Supervisor, the CanMonitor. Even though it performs the same roll as the OroMonitor in the system, the way in this is done is somewhat different. First of all, instead of ComponentPublishers reflecting component information, it will have StatPublishers which corresponds

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">

<properties>
      <simple name="MedThr" type="double"><value>1200.0</value></simple>
      <simple name="BtwMedThr" type="double"><value>600000.0</value></simple>
</properties>
```

Figure 36: CanSupervisor XML Configuration File

with the Stats of CanStatistics commented before. Each of these StatPublishers reports the information of the Stat and checks that the node is actually reporting or reading and the data timings are in safe thresholds. The thresholds can be set by the user using configuration XML files and can be even changed dynamically without the need to recompile or restart the process. Figure 36 shows and example of one of this XML files, this one corresponding to the incoming CAN node 11 information shown in Figure 35.

There is also a GeneralPublisher that shows all the CanStatistics data under the same window in the robot monitor application so the users can see all the information in one place. For specific node status the administrator needs to access the specific Stat element. An example of information shown in the GeneralPublisher is displayed in Figure 37.

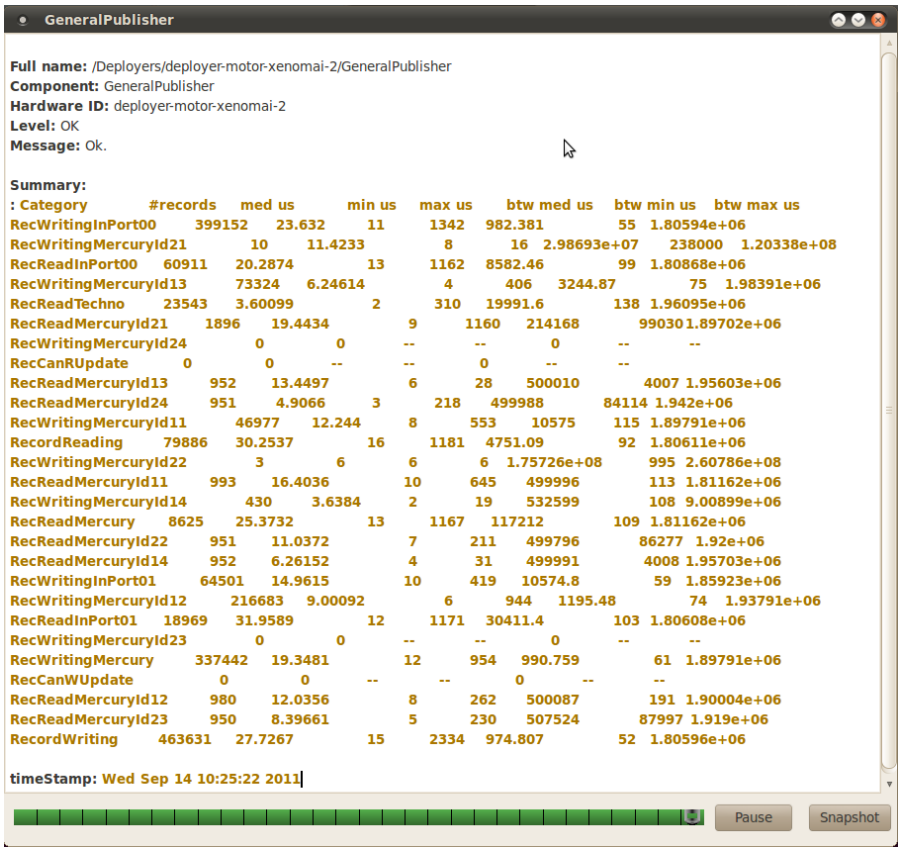


Figure 37: CanSupervisor General Publisher

Finally Figure 38, that shows the definitive implementation of CanStatistics and CanMonitor.

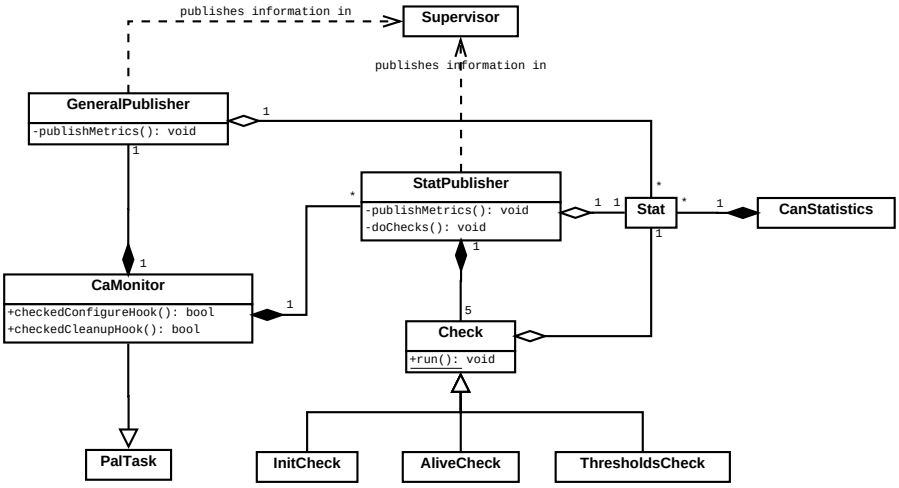


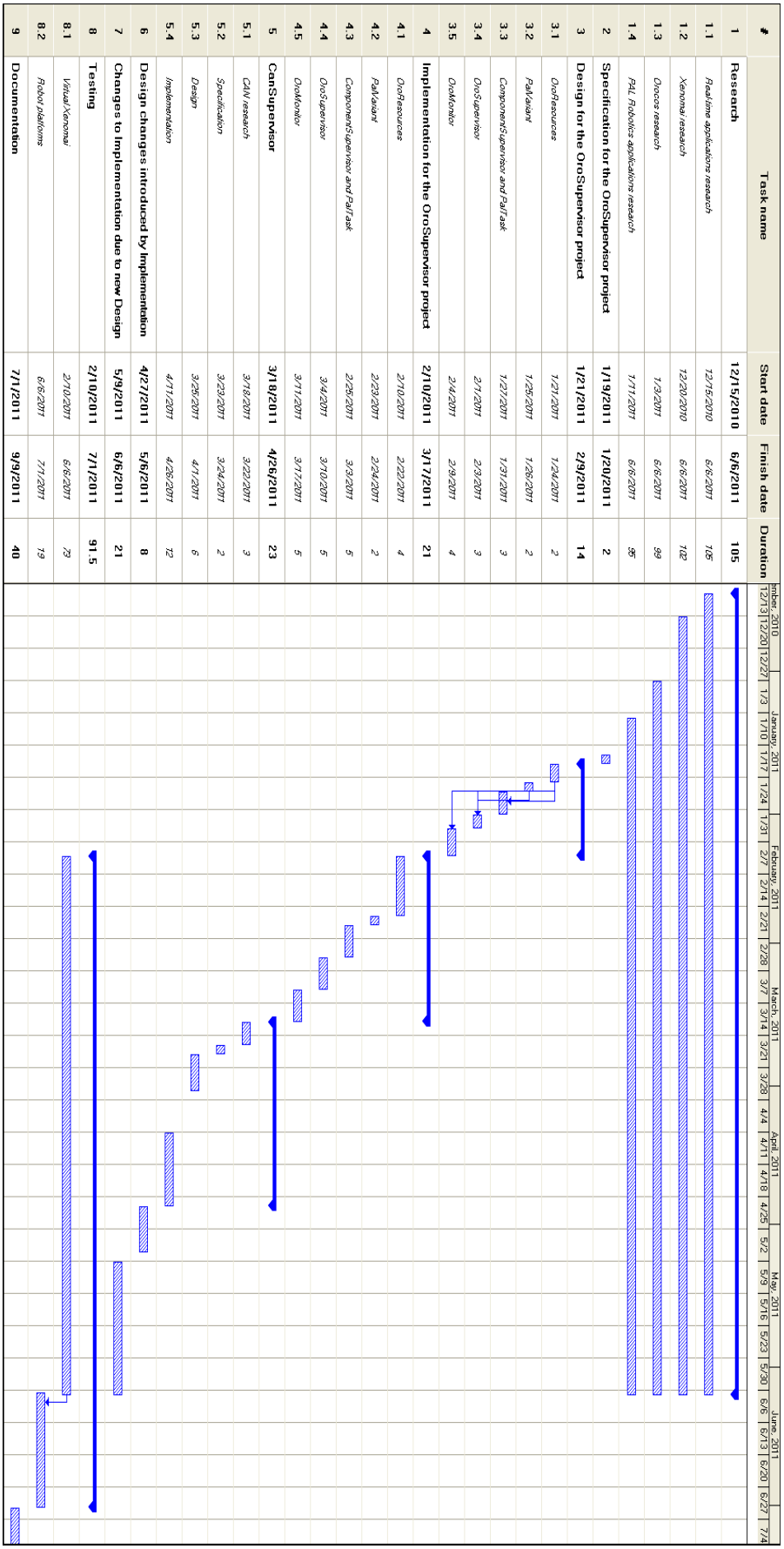
Figure 38: CanStatistics and CanMonitor Class Diagrams

6 FINAL PLANNING AND COST ANALYSIS

6.1 FINAL PLANNING

Comparing the original Figure 1 of the Planning section with the final planning shown in Figure 39 ahead we appreciate a final deviation of roughly thirty days in comparison to the original deadline. Though some tasks were actually finished ahead of schedule, due to some unseen factors or flaws in the original planning we ended up finishing the project later than expected. All the incidences are explained in detail next:

- Two weeks in which we could not make any advance in the development of the project due to a unexpected and urgent amount of work for the company. Said weeks are the 2/14 and 4/4.
- Overall research time spent in researching less than expected initially. However we did not foresee the necessity of researching during the development of the project. Though these were not actually very time consuming tasks, they are performed in parallel with the rest of activities of the project.
- A new task for researching CAN related topics.
- The tasks "Design changes introduced by Implementation" and "Changes to Implementation due to new Design" took longer than planned.
- Since robot platforms were available we decided to spend more time testing our project.
- PAL Robotics has two weeks of holidays in August. We spent one of these weeks working full time in the documentation of the project.



#	Task name	Start date	Finish date	Duration
1	**Research**	**12/15/2010**	**6/6/2011**	**105**
1.1	*Real-time applications research*	12/15/2010	6/6/2011	105
1.2	*Xenomai research*	12/20/2010	6/6/2011	102
1.3	*Orocos research*	1/3/2011	6/6/2011	99
1.4	*PAL Robotics applications research*	1/11/2011	6/6/2011	95
2	**Specification for the OroSupervisor project**	**1/19/2011**	**1/20/2011**	**2**
3	**Design for the OroSupervisor project**	**1/21/2011**	**2/9/2011**	**14**
3.1	*OroResources*	1/21/2011	1/24/2011	2
3.2	*PalValiant*	1/25/2011	1/26/2011	2
3.3	*ComponentSupervisor and PalTask*	1/27/2011	1/31/2011	3
3.4	*OroSupervisor*	2/1/2011	2/3/2011	3
3.5	*OroMonitor*	2/4/2011	2/9/2011	4
4	**Implementation for the OroSupervisor project**	**2/10/2011**	**3/17/2011**	**21**
4.1	*OroResources*	2/10/2011	2/22/2011	4
4.2	*PalValiant*	2/23/2011	2/24/2011	2
4.3	*ComponentSupervisor and PalTask*	2/25/2011	3/3/2011	5
4.4	*OroSupervisor*	3/4/2011	3/10/2011	5
4.5	*OroMonitor*	3/11/2011	3/17/2011	5
5	**CanSupervisor**	**3/18/2011**	**4/26/2011**	**23**
5.1	*CAN research*	3/18/2011	3/22/2011	3
5.2	*Specification*	3/23/2011	3/24/2011	2
5.3	*Design*	3/25/2011	4/1/2011	6
5.4	*Implementation*	4/11/2011	4/26/2011	12
6	**Design changes introduced by Implementation**	**4/27/2011**	**5/6/2011**	**8**
7	**Changes to Implementation due to new Design**	**5/9/2011**	**6/6/2011**	**21**
8	**Testing**	**2/10/2011**	**7/1/2011**	**91.5**
8.1	*Virtual Xenomai*	2/10/2011	6/6/2011	73
8.2	*Robot platforms*	6/6/2011	7/1/2011	19
9	**Documentation**	**7/1/2011**	**9/9/2011**	**40**

Figure 39: Final Planning

6.2 COST ANALYSIS

The economic cost of this project, i.e., the cost of the resources used in its development can be divided in three main categories: hardware, software and human resources.

For the calculations of each element we consider:

- A nominal cost, that represents the total cost of the resource.
- An effective cost, which is the proportional part of the nominal cost corresponding to the time employed by us in respect of the resource life span.

Hardware resources

The sum of our developing machine and the robots platforms conform this category and are summarized in the following Tables 1 and 2.

Element	Nominal Cost	Effective Cost
Developing machine	1,002.24 €	229.68 €

Table 1: Developing machine cost

Element	Cost per Hour	Hours	Effective Cost
Mobile base	8.00 €	32	256.00 €
REEM-H1	26.00 €	16	416.00 €
REEM-H2	34.00 €	28	952.00 €

Table 2: Robots usage cost

We consider a life expectancy of three years for the developing machine and we use it full time every day for the total duration of the project which amounts to approximately 10 months.

The actual cost of the robots is provided in the form of an hourly quota. According to the final planning seen Figure 1, nineteen days were dedicated to testing in robot platforms. Distributed in:

- Eight days spent in the Mobile base.
- Four days spent in the REEM-H1.
- Seven days spent in the REEM-H2.

Considering four hours per day, Table 2 contains the total effective cost of using these robots for the previously mentioned amounts of time.

Software resources

The next Table 3 lists all the software used for the developing of this project. Turns out all the utilities that conform our environment are open source or no paid license is required.

Application	Nominal Cost	Effective Cost
Ubuntu 10.04	0.00 €	0.00 €
Xenomai 2.5.4	0.00 €	0.00 €
VirtualBox 3.1.6	0.00 €	0.00 €
Orocos 1.8.2	0.00 €	0.00 €
QtCreator 2.0.1	0.00 €	0.00 €
GDB 7.1	0.00 €	0.00 €
GCC 4.4.3	0.00 €	0.00 €
CMake 2.8.0	0.00 €	0.00 €

Table 3: Software resources cost

Human resources

As we can conclude form Figure Final Planning the development of the software comprehends a total of 174 days with 4 hours of work per day, except the two weeks of holidays when I worked 8 hours per day. This amounts to a total effort of 736 hours. The full extension of the project has been done by a unique, with a cost for the company of €7.98/hour. The following Table 4 informs, then, the total cost of human resources.

Developer	Salary (€/hour)	Hours	Total
Student	7.98 €	736	5,873.28 €

Table 4: Human resources cost

Total cost

The next Table 5 details the total cost of the project.

Concept	**Amount**
Hardware	1853.68 €
Software	0.00 €
Human	5,873.28€
Total	**6,102.96 €**

Table 5: Projects total cost

7 CONCLUSION

The last chapter of this documentation explains the objectives covered at finalization of the project. Additionally, we also detailed more of the knowledge and experience acquired. Finally, we describe future changes or additions to the supervisor.

7.1 OBJECTIVES ACCOMPLISHED

We now proceed to enumerate the final state of the objectives listed in the Objectives section.

- Not only our skills with C++ language have improved, we have also acquired a much deeper understanding of the language. The lecture of some books, like Effective C++[28] and Effective STL[29], along with the experience acquired by interacting with our work colleagues has helped us to, in the end, be able to develop much more efficient, robust and clean software.
- We are now familiar with our IDE, moving with ease through the code and being able to use the developing tools with expertise. Specially the debugger which we can even operate outside of the Qt Creator environment. Also the introduction of Valgrind greatly increased our capacity to find and fix bugs.
- Research on real-time applications has been satisfactory, introducing us to a field that was completely unknown. Our expertise is of course not impressive yet, but we hope to keep improving in the future. The same can be said of the Xenomai OS. We can now use the specific tools it offers to get information and evaluate the real-time part of system. For the non real-time part we use the typical Linux tools, which we have come to known better too.
- Our knowledge of the Orocos framework is now extensive, being able to develop components, deployers and activities. Also, we can set up the necessary CORBA infrastructure for inter process component communication.
- We managed to write the list of specifications that clearly stated the client's, in this case our coordinator, desired goals for the project. The final version of the project succeed in covering all the specifications.

- The project's design ended up being clean and simple. We put special focus in dividing it in a series of modules with clearly distinctive functionalities that reduced the coupling and facilitated the test of each unit individually. Also we tried to make a design easy to scale taking into account the possibility of future changes or additions to the project. Our knowledge in UML has also been improved thanks to this project, learning how to include C++ specific features like templates.

- Given the fact that our starting knowledge of C++ language was basic, implementation was probably the time consuming part of the project. However, after much work, the final state of the implementation resulted satisfactory. The introduction of C++ idioms made for a professional implementation and solved problems we came across with. Also, specially useful was the advices given in the Effective books mentioned before, letting us develop a much more efficient code.

7.2 FUTURE WORK

Even though the project is finished, by the time this documentation was written there was already some new changes and additions that did not fit in the initial objectives. Also, there are plans for future features already. We proceed to enumerate them.

- The Orocos version used in PAL Robotics is the 1.8, but currently we are in the process of migrating to the last stable 1.12 version. Some changes to the code will be necessary due to slightly changes in the framework.

- A new feature for the OroSupervisor: a real-time safe logger. As previously said, logging information to any type of device is a non real-time safe operation. Thus, logging is normally forbidden inside real-time safe components loops. Taking advantage of the already existing communication infrastructure of the OroSupervisor with its supervised components, we can add a new type to be transmitted between them: log messages. After that, we develop a new set of macros that resemble those already being used in PAL Robotics. With this macros the user will be able to send log messages using the component's OutPort. Finally, the OroSupervisor will be responsible for log those messages.

- As mentioned in the PalTask implementation section, the users still have the possibility of overriding the original hooks methods. A good API should always be hard to use incorrectly, and for us this is not the case. Fixing this issue is one of our pending tasks. Even though we have researched during a considerable amount of time we have not been able to reach a satisfactory

```cpp
#include <stdio.h>

class A
{
public:
  virtual void foo() { return; }
};

template<typename ChildClass>
class B: public A
{
public:
  B(): A() {
    //casting to void* to check for the pointers
    if( reinterpret_cast<void*>(&B::foo) != reinterpret_cast<void*>(&ChildClass::foo) ) {
      ::printf("ERROR! foo() overridden!!!");
    }
    else {
      ::printf("OK! foo() not overridden.");
    }
  }

  void foo() { return; }
};

class C : public B<C>
{
public:
  void foo() { return; }
};

class D : public B<D>
{
public:
  //no overriding B::foo member function
};

int main(int argc, char *argv[])
{
  C c; //prints "ERROR! foo() overridden!!!"
  D d; //prints "OK! foo() not overridden."

  return 0;
}
```

Figure 40: Function override detection sample code

answer. An example of discussion related to this topic can be consulted in point 3 of the bibliography, though there are many other examples. It definitely seems the problem can not be solved in compilation time, since from a C++ design point of view if a function is not declared virtual users should never override them. Leaving such responsibility to the user is something we do not want. However, there is an apparent solution in runtime using reflection. In Figure 40 a code that successfully demonstrates this has been made. First we need to provide the base class with a pointer to the derived class. Said pointer can be passed to the base class using the Curiously Recurring Template Pattern (CRTP)[4] C++ idiom. With the CRTP a class X derives from a class template instantiation using X itself as template argument. Then, at the moment of the base class construction, we can analyze if a member function foo() was overridden or not by comparing the base class's pointer to foo() and the derived class's pointer to foo(). However a considerable refactoring to PalTask would be needed in order to grant the project this new feature.

- When serious problems with the motors are detected some kind of alarm should alert the user immediately. Right now, only a warning message is issued.
- When Orocos loads a set components from XML files does it in a random order. This is an important limitation, since there may be critical components that need to be deployed first. As a consequence, the motor control components are loaded separately from the rest and the process is, unfortunately, hardcoded. For the same reason, the OroSupervisor and CanSupervisor components are also hardcoded. However there is plan for the future to change this and have all the Orocos components loadable by XML files. The idea is to create a separate component that handles the deployment process. This way, supervision could be enabled or disabled without the need to recompile the whole code.
- As mentioned in Chapter 5, filters to reduce the amount of CAN messages –besides the default one– are not currently implemented. We are looking forward to do several types of these and offer some sort of User Interface (UI) for dynamically enabling/disabling them.
- More metrics, checks and rules to describe the current components and motors status.
- Offer the user the possibility to customize which metrics and rules should be collected and shown for his component in case the default behaviour does not cover his needs.

7.3 FINAL THOUGHTS

During the course of this project we have learnt the importance of a well thought planning, taking into account not only the time required for finishing the set tasks but for possible delays and setbacks.

When facing a problem, always explore all the possible solutions, gauge the advantages and drawbacks of each and select the one that suits the problem the most. Do not be afraid of committing mistakes, if the solution ends up not being optimal just re-evaluate the problem and repeat the aforementioned procedure.

Sharing your thoughts with your colleagues and hearing their opinions, being able to accept constructive criticism and learn from the experience of more educated people are all crucial for the personal development and completion of any project.

BIBLIOGRAPHY

[1] Boost C++ Libraries, . http://www.boost.org).

[2] Boost Variant, . http://www.boost.org/doc/libs/1_47_0/doc/html/variant.html.

[3] Discussion: C++ prevent a method from being overridden in sub classes. http://stackoverflow.com/questions/17483/c-anyway-to-prevent-a-method-from-being-over-ridden-in-sub-classes.

[4] Curiously Recurring Template Pattern. http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern.

[5] Callgrind. http://valgrind.org/docs/manual/cl-manual.html.

[6] Cpp Unit. http://sourceforge.net/projects/cppunit.

[7] GDB. http://www.gnu.org/s/gdb.

[8] Opaque Pointer. http://en.wikipedia.org/wiki/Opaque_pointer.

[9] The Orocos Component Builder's Manual, . http://www.orocos.org/stable/documentation/rtt/v1.12.x/doc-xml/orocos-components-manual.html.

[10] The Orocos Real-Time Toolkit, . http://www.orocos.org/rtt.

[11] Orocos RTT Frequently Asked Questions, . http://people.mech.kuleuven.be/~orocos/pub/stable/documentation/rtt/current/doc-xml/orocos-faq.html.

[12] Resource Acquisition Is Initialization. http://www.boost.org/doc/libs/1_47_0/doc/html/variant.html.

[13] Real-Time Linux Wiki. https://rt.wiki.kernel.org/index.php/Main_Page.

[14] Singleton Pattern. http://en.wikipedia.org/wiki/Singleton_pattern.

[15] Valgrind. http://valgrind.org.

[16] Controller Area Network, . http://en.wikipedia.org/wiki/Controller_area_network.

[17] Continuous Integration, . http://en.wikipedia.org/wiki/Continuous_integration.

[18] Common Object Request Broker Architecture, . http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture.

[19] Real-time Computing, . http://en.wikipedia.org/wiki/Real-time_computing.

[20] Standard Template Library, . http://en.wikipedia.org/wiki/Standard_Template_Library.

[21] Weighted Mean, . http://en.wikipedia.org/wiki/Weighted_mean.

[22] Xenomai: API, . http://www.xenomai.org/documentation/xenomai-2.0/html/api/group__task.html.

[23] Xenomai Scheduling Data, . http://www.xenomai.org/index.php/Proc/xenomai/sched.

[24] Xenomai Statistic Data, . http://www.xenomai.org/index.php//proc/xenomai/stat.

[25] Valgrind in Xenomai, . http://www.mail-archive.com/xenomai-help@gna.org/msg07725.htmll.

[26] Xenomai: Real-Time Framework for Linux, . http://www.xenomai.org/index.php/Main_Page.

[27] Giuseppe Lipari. Real-Time Linux and the Xenomai system, May 2008. http://retis.sssup.it/~lipari/courses/str07/xenomai-handout.pdf.

[28] Scott Meyers. *Effective* C++. Addison Wesley, .

[29] Scott Meyers. *Effective STL*. Addison Wesley, .