Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Linux as a Real-Time Operating System: a practical analysis in the mobile robotics domain

Relatore: Prof. Braione Pietro

Co-relatore: Prof. Ferretti Claudio

Relazione della prova finale di:
Jacopo Maltagliati
Matricola 830110

Anno Accademico 2019-2020

Sample Text
Sample Text

Contents

Chapter 1

Summary

In the past ten years, mobile robotics as a field has been gaining a lot of traction: as technology evolves, building a mobile platform has become more and more feasible for anyone, including small research teams and hobbyists. However, there is still a lack of easily approachable real-time solutions able to manage the workload, as many of the available frameworks are not mature enough to offer fine-grained real time application control: this is why we ultimately decided to push the Linux kernel to its limits, studying its real-time capabilities and ease of use.

1.1 The problem

Imagine driving on an empty road at night while listening to classical music: it's been a very long day, and your consciousness gradually drifts to a peaceful slumber. Suddenly, an animal crosses the road: are you going to notice? Will you be able to stop in time? Now, consider a self-driving car in the same situation: what would happen if the control computer "dozed off" even for a fraction of a second? The user would probably not be able to stomp on the brakes fast enough to prevent the car from crashing.

A possible solution to this problem is carefully designing a system built around existing technologies, such as real-time control, redundant hardware platforms, and failsafe mechanisms. However, creating a real-time system from the ground up tends to be very expensive and error-prone, as incorrect design or minor mistakes could lead to catastrophic failures such as the infamous Therac[1] incidents.

For this reason, many software houses have been working on various commercial solutions since the 1980s that enabled programmers and engineers to deploy their application on proven grounds, thus reducing the time and effort needed to create a real-time application. Arguably, the best aspect of those systems was their widespread availability: for example, DEC's VAXeln used lightly modified VAX computers, and Quantum Computer Inc. QNX ran on consumer-grade hardware, including x86 processors. While the VAXeln is now a historical platform, QNX is still commercially available from BlackBerry Inc. and major companies such as Apple use it for their products[2].

However, having always demonstrated an incredible flexibility, the ubiquitous Linux kernel is becoming a viable option for robotics and real-time usage, with both academic and commercial applications exploiting it like Tesla Motors, which maintains its own fork[3].

Of course, this flexibility comes at the cost of reduced effectiveness in each specific situation, so why do both academic and commercial researchers decide to use Linux? For example:

- Linux is free and comes at no cost

- Linux is open-source, so it might be tailored to specific needs with the right expertise
- Linux has a wide community of people constantly improving it, ensuring constant updates without any support contract

This is why, despite the shortcomings of the Linux kernel, we decided to adopt it for this project.

1.2 Current Implementation

At the moment, most of IRAIab's mobile platforms are controlled by a master node running Ubuntu Linux 16.04 LTS and the ROS framework. The ROS (Robot Operating System) framework is an open-source middleware, that is a set of software libraries and tools that help you build robot applications. ROS was created by Willow Garage as a rapid development platform for their PR2 robot and traces its roots into an even earlier effort by a team of researchers and students at Stanford University.

Despite its name, ROS needs a host operating system: at the time of writing, it can run on either Linux, Windows or macOS, but since the behavior of each platform has notable differences, we'll focus only on the ROS/Linux combination. Please also note that ROS and ROS2 will be treated as separate entities throughout this report.

1.2.1 Description

Currently, the navigation stacks uses mainly uses ROS for abstraction and rapid prototyping: through the use of *topics* and *messages*, ROS provides a facility to let the users write small programs, called nodes, and make them communicate with each other. A master node, provided by the framework, acts as a central hub, keeping tabs on the state of the *Computational Graph*. Moreover, ROS provides extensive timestamping and data logging facilities, letting the user easily track, monitor and simulate a mobile platform's behavior.

An in-depth analysis of the ROS functionalities and usage in an IRAIab mobile platform is outside the scope of this report, but it's received extensive coverage in works performed by Gerosa* and Di Lauro*.

1.2.2 Issues

The problem with ROS is, simply put, its clunkyness: while it's a very good rapid prototyping platform, there is no way to control the way that the whole thing is scheduled besides modifying the source code extensively, which would be an incredibly complex task. The authors of the software have acknowledged those problems themselves* and set out to release a newer, real-time friendly version of the platform.

1.3 ROS2

ROS version 2 is

1.4 Possible Solutions

To solve these problems we have identified a couple solutions

1.4.1 PREEMPT_RT

TODO
TODO

1.4.2 SCHED_EDF

TODO

Chapter 2

Solution

Our custom solution, dubbed `rt-app` (as in Real-Time Application) is a demonstration of a viable approach for building real-time applications on Linux.

Written from scratch, the application currently leverages several APIs provided by Linux such as sched, time and pthreads.

2.1 Design

`rt-app`'s design is based around some components which are regarded as standard in the mobile robotics field, such as:

- The global planner: a component that keeps track
- The local planner: a component responsible for tracking
- AMCL:
- The costmap:
- odometry: a component that constantly elaborates the data received from various sources, such as the move base's wheel encoders, to keep track of where the robot is going.

Moreover, the following components have been envisioned:

- The pose manager: this is a pivotal component that manages the lifecycle of a pose, and is able to merge and
- The dispatcher:

Most of these operations are to be performed on-line, that is in a real-time fashion, except for global planning and thread dispatching, both of which happen once, when the application sets the modules up.

2.1.1 Rationale

The application's design, per se, is not really important: our goal is demonstrating that replicating a ROS-like behavior is possible by only using facilities provided by the Linux kernel and the GNU stdlib.

2.1.2 Planning

Given that the work required to implement such a complex application can't be possibly condensed in a three-month, one person stage, we've settled for splitting the work in several phases, namely:

- Phase 1: High level design and survey of the possible solutions
- Phase 2: Implementation of the dispatcher, at least one of the modules and integration with an existing mobile platform.
- Phase 3: Maintenance and gradual addition of more modules, according to the programming guidelines and general architecture set by the work performed in phase 2.

Thus far, Phase 1 and a consistent portion of Phase 2 have been completed: the Odometry module has been implemented, on top of a simple dispatcher based on `pthreads` and `SCHED_EDF`. IRAlab's Otto* mobile platform has been selected to be the test candidate, thus the Odometry module has been interfaced with a microcontroller emulating the protocol and message format used by Otto. The nature of this emulation will be discussed in a later paragraph.

2.2 Implementation

2.2.1 Threads

2.2.2 Scheduling

2.2.3 Timing

2.2.4 Communications

Serial

2.2.5 Shared memory

2.3 Future Development

Chapter 3

Addendum

3.1 Patching and building the kernel

3.2 Emulating the mobile platform

3.3 Processor features that are detrimental to RT performance

3.4 Logging

3.5 Filesystem

Bibliography

[1] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[2] D. Amoth, "Apple CarPlay runs on BlackBerry QNX." `https://time.com/12452/gasp-apple-carplay-software-runs-on-blackberrys-qnx-platform/`, 2014.

[3] "Tesla Linux for Tegra." `https://github.com/teslamotors/linux`, 2020.