University of Glasgow | School of Computing Science

Real-time Robot Camera Control in Erlang

Andreea C. Lutac

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — Wednesday 7th October, 2015

Abstract

Abstract goes here.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

1.1 Aims

1.2 Background

1.3 Motivation

1.4 Report Content

Chapter 2

Literature Review

2.1 Robotics

2.1.1 History of Robotics

2.2 Robot Operating System

Within the last few years, an increasing number of endeavours in the field of robotics are making extensive use of a collection of software frameworks for robot development known as the Robot Operating System (ROS). Originally developed in 2007 in the Stanford Artificial Intelligence Laboratory and later ”incubated” at Willow Garage - a collaborative, cross-institution robotics research initiative - the project went on to become one of the most widely employed platforms for robot control, in both academic and industrial settings.

The main aim of ROS is to provide operating system-like functionalities for specialized robotic agents, under an open-source license. Thus, the services it compounds are similar to the ones found in most software middleware:

- Hardware abstraction, which enables the development of portable code which can interface with a large number of robot platforms
- Low-level device control, facilitating robotic control
- Inter-process communication via message passing
- Software package management, which ensures the framework is easily extendible

Of particular interest amongst these is the process management aspect. Indeed, the heavy reliance of ROS on its message-passing communication infrastructure is where most of its perceived benefits and drawbacks lie.

Thus, at the core of all the programs running on ROS is an anonymized form of the publisher-subscriber pattern [**?**]. An process is expected to first register with the ”master” ROS service as a ”node” [**?**] at the start of its routine. The master service is in itself a ROS node, responsible for providing naming an registration services to the rest of the nodes in the system. Its primary function is to act as node discovery hub, which means that after

node addresses have been relayed as needed, inter-node communication can be done peer-to-peer. Newly-created nodes are henceforth able to either publish messages on a public message log called a ”topic” [**?**] or subscribe to an already existing topic to receive data at a set frequency (Hz).

This type of message-passing architecture supports the easy integration of custom user code within the ROS ecosystem and unifies the different APIs that would normally be needed to access relevant system information (sensor data, actuator positions, etc.). However, scalability issues arise as the number of nodes grows. Given the fact that ROS adopts a graph model to store and manage its node network, a substantial increase in the number of edges (connections) or indeed the forming of a complete graph (in essence an all-to-all connection) is likely to affect performance. Adding to this problem is the fact that ROS bases its inter-process synchronisation mechanisms on time stamps, which can induce failure in certain processes if the time stamps are not received in order.

A further potential handicap exhibited by ROS is the lack of real-time OS functionalities. While this might not pose a problem in practical scenario where robots perform non-critical tasks, it is clear that high reactivity and low control latency are goals that all robotic systems should strive for. Having said this, ROS does support integration with external real-time toolkits such as Orocos, passing in sensor information and retrieving computation results.

2.3 Erlang

Erlang is a programming language centred around a functional paradigm and focused on ensuring effective concurrency and thus providing adequate support for distributed programming.

Developed by Ericsson engineers in 1986, Erlang was initially designed as a proprietary programming language within Ericsson. Following its release as open source in 1998, the language found use cases in numerous projects, both corporate and academic. Most frequently, Erlang's powerful concurrency model is employed in running chat, messaging and telephony services within companies such as Facebook, T-Mobile, Motorola and WhatsApp. In addition to these, Amazon makes use of Erlang in its implementation of the Amazon Web Services distributed database offering.

2.3.1 Models of Concurrency

Within the field of computing, two main concurrency models have distinguished themselves over time:

- Shared state concurrency
- Message passing concurrency

The first of these constitutes by far the most widely utilised pattern and is the standard approach to concurrency adopted by languages such as C, Java and C++. Shared state concurrency centres around the idea of mutable state, where memory-stored data structures are subject to modification by different processes or threads. Thus, in order to prevent race conditions resulting in inconsistent data, synchronisation mechanisms based on locks must be implemented.

However, complete failure of a process executing in the critical region - and thus actively holding the lock on a memory-stored location - can often result in the irremediable failure of all other processes waiting on the release of the lock.

To contrast with this model, message passing concurrency involves no shared memory regions, instead enabling inter-process communication through message exchanges, as the name suggests. And indeed this approach is the one espoused by Erlang, which favours parallelism over locking. This model presents some similarities to the publish-subscribe message passsing pattern built into ROS.

2.3.2 Concurrency in Erlang

Unlike the programming languages mentioned above, Erlang does not allow for mutability within its data structures. This feature removes the concern of processes making conflicting modifications to in-memory data structures, yet also imposes some constraints on the programmer in terms of how the code logic is developed.

In order to manage and monitor numerous tasks being executed simultaneously, Erlang leverages the actor model, one of the fundamental mathematical models for concurrent computation.

One of the foremost characteristics of Erlang...

2.4 Camera Control

Chapter 3

Requirements

3.1 Problem Analysis

Chapter 4

Design and Implementation

Chapter 5

Analysis

Chapter 6

Conclusion