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 — Tuesday 22nd March, 2016

## Abstract

This dissertation investigates the applicability of Erlang as an alternative robot control system software architecture that may improve upon the scalability and reliability limitations of more traditional approaches based on the Robot Operating System (ROS). Literature examining Erlang in the context of robotics has identified its potential within the field of robotic control, but has so far failed to employ concrete metrics to quantify the non-functional advantages of Erlang. The project bridges this research gap by performing a comparative analysis of the scalability and reliability of two typical robotic middlewares relying on ROS and Erlang, respectively.

Using a controllable camera as a typical robotic component, two face tracking systems were developed that apply ROS and Erlang-based asynchronous message passing models to achieve communication between a camera device and associated face detection worker processes. To verify the hypothesis of Erlang offering improved performance, a suite of five experiments was carried out on both applications. The first three of these trials targeted scalability and their findings indicated that Erlang is capable of supporting 3.5 times more active processes than ROS, at a generally similar communication latency of approximately 37 ms for a camera-process-camera roundtrip and no decline in face quality. In addition, Erlang exhibited 2.2 times lower per-process memory costs when compared to the ROS system. The latter two experiments tackled the reliability of the face tracking architectures and illustrated that, while both systems are able to successfully recover from partial crashes, Erlang is able to restart failed processed approximately 1000 to 1500 times faster than the ROS services and therefore better mitigate the impact on the quality of tracking.

The implications of these results provide credence to the original arguments and highlight the favourable prospects of Erlang in robotics. The project concludes that, if applied appropriately and accessed through optimised linking libraries, an Erlang-based middleware possesses the potential to greatly improve the flexibility and availability of large-scale robotic systems.

## 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: ____________________

## Acknowledgements

I would like to thank **Professor Phil Trinder**, my project supervisor, for his support, patient guidance and useful critiques of this research work. The assistance provided by my co-supervisor **Dr. Gerardo Aragon-Camarasa** on the features of ROS and metrics for assessing face quality were also invaluable to the successful completion of the project. My grateful thanks are also extended to my co-supervisor **Dr. Natalia Chechina** for offering her expertise in Erlang, as well as great advice on keeping my progress on schedule.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Context

Modern autonomous robots are large-scale, complex systems which require intensive interaction between various hardware and software components. Many of the services crucial to robot operation, such as sensor monitoring, motor control or gripper actuation, rely on the fast and reliable bridging between the physical layer and the program logic. For these purposes, a robust robotic middleware is required to manage the complexity involved in connecting heterogeneous systems of components.

A common abstraction layer used in advanced robotics projects is the Robot Operating System (ROS) [45], an emergent de-facto standard that supports concurrent synchronous and asynchronous processes communicating by means of a message passing protocol. Most of the development work behind ROS comes through contributions from the robotics community (mainly academic groups) and thus the middleware has grown organically to support a large range of compatible robots. However, in spite of its widespread adoption, ROS exhibits a number of scalability and inter-process synchronization limitations in practice, which have not been commonly explored and reported.

This project thus sets out to investigate the use of Erlang [13] as an alternative to ROS in the field of control system software architecture. As a programming language specifically designed for high-volume telecommunications, Erlang is well known for its strong concurrency model, its ability to handle a large number of processes, and its reliability mechanisms. All of these features imply that Erlang has the potential to offer a solid and fault tolerant underlying architecture on which to build a system that is functionally equivalent to ROS, yet provides improvements on its inter-communication capabilities.

## 1.2 Aims and Objectives

In light of these observations, the aim of this research is to perform an experimental comparison of the scalability and reliability of two analogous middleware systems in a robotic control context. The first of these harnesses the message passing capabilities of ROS, while the second uses Erlang to model similar component-bridging software.

The function of these systems is to model a control layer for an image capture device, performing face detection on frames captured in real time and then relaying the results to camera pan-tilt-zoom (PTZ) actuators to achieve face tracking. The decision of using a PTZ camera as an archetypal robotic component is based on two major factors. The first of these relates to the core sensory role vision plays in robotics, which makes cameras a relatively ubiquitous piece of hardware in autonomous robots. The second factor is represented by the straightforward way in which image manipulation programs can be distributed and parallelised, thus facilitating the completion of scalability and reliability experiments.

Consequently, the project seeks to meet a number of high-level objectives, listed below:

- Create two functionally-equivalent face detection systems in ROS and Erlang, whose execution can be accurately benchmarked in terms of scalability and reliability
- Establish experiment parameters and document the assumptions made and the limitations of the non-functional tests
- Conduct these comparative benchmarks with the goal of gaining concrete metrics of how ROS and Erlang perform as a message-passing medium
- Analyse the experiment results and draw conclusions about the suitability of Erlang in a robotic context

## 1.3 Achievements

Throughout its timeline, the project built upon the above objectives and accomplished a number of goals.

### Review of Literature and Software Technologies

The first of these entailed the careful study of ROS features, followed by an analysis of the strengths and shortcomings of its message-passing model. A similar review was done on the intricacies of the Erlang programming language. Here the most significant challenge lay in the process of familiarization with the functional programming paradigm and the differences between its syntax and imperative programming patterns.

The last stage of the literature review looked into previous research efforts that sought to employ Erlang in robotics. This step was particularly beneficial in cementing a clear understanding of how these past endeavours mirrored the project and in which aspects they deviated.

### Development

Before any concrete implementation could be created, the architectural models of two envisioned Linux-based ROS and Erlang face-tracking systems were carefully planned and iteratively optimised so as to ensure the programs' functional equivalence. Their common goals are thus:

- obtaining images from a Logitech Orbit Sphere [36] camera device
- performing distributed face detection on the images

- aggregating the results and controlling camera movement actuators

Hence, a two-module Python program was developed within the ROS environment, acting as a typical image processing and camera control robotics module. This 200-line application receives image frames from a camera source in real-time and performs face detection operations on these images using an OpenCV Haar cascade classifier [17]. The operations are distributed across multiple concurrent processes and their results aggregated for output to a camera control service.

Parallel to this, an implementation of a comparable Erlang face tracking solution was carried out. This application makes use of Erlang as the message-passing layer between the camera source and an equivalent set of face detection processes running the same Python-based cascade classifier as used in ROS. The program encompasses approximately 300 lines of functional code spread amongst three Erlang modules and two Python scripts, with three additional Erlang modules guaranteeing reliable operation via supervision trees.

## Analysis

After establishing appropriate parameters for each of the two implementations that would ensure comparable functional performance, five experiments examining two non-functional aspects of the programs were designed, conducted and analysed as a basis for comparing the capabilities of the ROS and Erlang technologies.

Three *scalability* experiments were run, that took into account the variation in computation and camera response time when the number of face detection processes is gradually increased. In order of execution, these are:

- **Concurrent Worker Limits**, an experiment that determined the systems' limitations with regard to the maximum number of face-detecting workers that can be simultaneously active. Worker numbers were gradually increased for both the ROS and the Erlang application up to the point of unstable operation or program crashes. Thus, the Erlang middleware was discovered to be highly scalable, supporting more than three times the number of workers ROS tolerates.

- **Message Latency**, an experiment that timed the latency of both systems' from when the frame is collected from the image feed to the moment the face position is ready to be sent to the actuator. Detailed breakdowns of the delays incurred in each step on the communication process were recorded and compared for each of the two systems. This led to the conclusion that both programs have initially similar latencies, with Erlang incurring a heavy cross-language message relay penalty once the number of worker processes increases.

- **Face Quality Variations with Real-time Aggregation**, an experiment that assessed the accuracy of the detection process as frame and face data is relayed through the two middlewares, using increasingly more worker agents. This experiment replaced the camera source with a simulated "real-time" sequence of 100 images already annotated with face positions and gauged how similar the face results of the ROS and Erlang systems were to the ground truth reading. Using a metric derived from the Jaccard similarity index [64], the initial hypothesis that both programs exhibit comparable face quality was confirmed.

*Reliability* experiments tested the two systems' resilience when faced with partial failures (e.g. processes hanging or terminating abruptly) and how these failures affect the quality of face detection. In order of execution, these are:

- **System Resilience**, an experiment that compared the behaviour of the two systems when processes are terminated unexpectedly. As part of this assessment, ”chaos monkey” scripts were developed in order to simulate random disruptions in main and worker node/process activity, both through ”safe” terminations (with the signal SIGTERM) and abrupt kills (with the signal SIGKILL). The period between these terminations and the respective agents resuming operation was timed for both systems, showing that Erlang processes restart approximately 1000-1500 times faster than ROS nodes.

- **Face Quality Variations with Worker Failure,** an experiment that assessed the accuracy variations in face detection when the systems are faced with abrupt partial worker failures. In order to carry out this experiment, the same simulated ”real-time” image source used in the third experiment was linked to the detectors and an increasing proportion of agents were terminated each time. This served to illustrate the fact that the Erlang middleware suffers less from drops in face quality, owing to its swift process restart capability.

## 1.4 Outline

The subsequent chapters of this dissertation discuss each of the mentioned achievements in detail, starting with the research phase, outlined in the Literature Review. The development of the face tracking programs is described in the Design and Implementation chapter, while the experiment structure, execution and findings are explained in the Evaluation chapter. The dissertation closes with a discussion of the conclusions drawn from the experimental findings and a review of the project's limitations and how these could be improved upon in future refinements.

# Chapter 2

# Literature Review

This chapter is intended to provide background information on the technologies and concepts utilized in the project, as well as an overview of other relevant works on Erlang and robotics.

## 2.1 Robotics and Computer Vision

Ever since classical antiquity, humans have manifested a marked interest in automation, translated into numerous attempts at the construction of autonomous mechanical systems. While the early forays into robotics did not yield much in the way of full autonomy, starting with the second half of the 20th century, robotics has gradually expanded to become one of the most innovative fields of research [37].

Modern robotics spans a multitude of branches, including electrical, electronic engineering and software-focused computing science, which are then integrated with various aspects characteristic of natural sciences. This is especially evident in robots that are built to emulate human perception or behaviour, where inherently human traits are sought to be replicated programmatically with as much accuracy as possible.

Robot vision is a particularly intriguing facet, as it forms the principal means through which intelligent creatures analyse their surroundings. In the case of robotic systems, the capture and interpretation of visual stimuli is done via artificial sensors such as digital cameras. While it is evident that human brains have evolved to be very efficient at visual data processing tasks such as feature extraction and object detection, computer systems attempting to do the same are bound by the physical limitations in memory, clock speed and data transfer rates of the hardware they utilise [65]. This is the reason why optimisations in the data flow between a system's components — be they software modules or physical robotic actuators — are just as important in guaranteeing good performance as efficient algorithms.

### 2.1.1 OpenCV

A widely used image processing library within the robotics community is OpenCV (Open Computer Vision) [27]. Built in C++, yet offering a Python wrapper as well, it is designed to provide a suite of optimised machine learning algorithms trained in the tracking and detection of faces, as well as in the separation of facial depictions into primary edges. This project employs in-built OpenCV face detection modules to identify human faces within an image, based on a method using Haar feature-based cascade classifiers [17]. Cascade classification uses a supervised learning approach, building a collection of common characteristics identified in human faces from a training data set and matching these traits against unseen image samples.

A feature of this algorithm (and most computer vision algorithms in general) is that it operates on a pixel-by-pixel basis, thus resulting in increased complexity and slow running time on single core machines [62]. However, image processing algorithms are also often suitable for parallelization, as their discrete and separable operations [69] can be effectively mapped to a network of agents and later reduced or compounded to obtain correct results in a fraction of the time taken for sequential execution. The challenge lies in streamlining such interactions between network nodes, and is addressed by technologies like the ROS framework and the Erlang programming language, described in the following sections.

## 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) [45]. Originally developed in the Stanford Artificial Intelligence Laboratory and later ”incubated” at Willow Garage - a collaborative, cross-institution robotics research initiative - the project offers client libraries mainly in C++ (`roscpp` [41]), Python (`rospy` [46]) and LISP, but also experimentally in other popular languages such as Java, JavaScript, Haskell and Ruby [3].

Since its conception in 2007, it has become one of the most widely employed platforms for robot control in both academic and industrial settings, as more systems began to take advantage of its modularity, inter-platform communication abilities [47] and in-built support for a wide range of hardware components [38].

The main aim of ROS is to provide operating system-like functionalities for specialized robotic agents, under an open-source license [39]. Thus, the services it offers 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 [44] 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 [35]. Processes are termed ”nodes” [25] in ROS, as they form part of an underlying graph (Figure 2.1) that keeps track of nodes (via the graph resource name [24], a unique identifier) and all the communications between them at a fine-grained scale. Since most robotic systems are assumed to comprise numerous components, all requiring control or I/O services, the ROS node architecture expects that each computation be delegated to a separate node, thus reducing the complexity involved in developing monolithic control systems.

At the start of its execution, a node is expected to first register with the ”master” ROS service, `roscore` [20]. 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 communicate in one of three ways: [40]

- Topics
- RPC Services
- Parameter Server

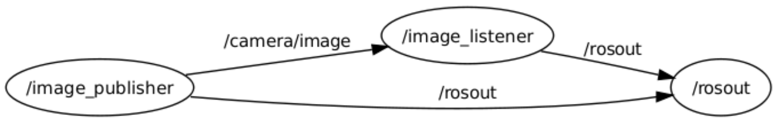

Figure 2.1: Example of a ROS node graph [56]

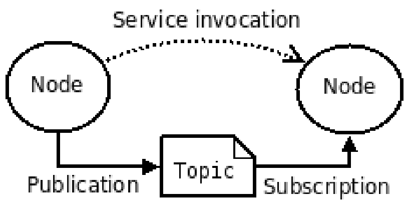

Figure 2.2: ROS topic interaction [40]

The most common communication mechanism involves transmitting and receiving messages asynchronously on a strongly-typed named bus or log called a ”topic” [51] and is the method explored in the project’s prototypical ROS face tracking application. Nodes can create, publish or subscribe to a topic to send and receive data in the form of messages of a predefined type, at a set frequency measured in Hz (Figure 2.2). Built for unidirectional streaming communication, topics also benefit from anonymous publish/subscribe semantics, thus removing communicating nodes’ awareness of each other.

However, in cases where the many-to-many transport provided by topics is not appropriate, ROS offers a form of node interaction based on the request/reply paradigm of remote procedure calls [48]. These services are defined by a pair of messages, of which one is the request sent to a node and the other is the reply.

The parameter server [31] takes the form of a globally-viewable dictionary of typically static, non-binary data points. Designed primarily for the storage of shared configuration parameters, the server does not uphold any premise of high performance and thus should not be seen as global variable store.

The ROS message-passing architecture supports the easy integration of custom user code within its 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 [23], which can induce failure in certain processes if the time stamps are not received in order [59].

## 2.3 Erlang

Erlang [13] is a programming language centred around a functional paradigm and focused on ensuring effective concurrency and providing adequate support and reliability for distributed programming. Developed by Ericsson engineers in 1986, Erlang was initially designed as a proprietary programming language within Ericsson [54]. Following its release as open source in 1998 [53], 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 [63] and WhatsApp [66]. In addition to these, Amazon makes use of Erlang in its implementation of the Amazon Web Services distributed database offering [70].

Designed from the very beginning for a highly distributed world, Erlang's primary strong point lies in its concurrency model, which deviates from the approaches historically employed by other programming languages, as detailed below.

### 2.3.1 Models of Concurrency

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

- Shared state concurrency [49]
- Message passing concurrency [22][21]

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++ [5][19]. Shared state concurrency centres around the idea of mutable state, where memory-stored data structures can be accessed and subjected to modification by different processes or threads. Thus, in order to prevent race conditions resulting in inconsistent data, synchronisation mechanisms based on locks are usually implemented, restricting processes' access to a shared memory location while it is being modified by another process. However, failure of a process executing in the critical region — and therefore 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 [68].

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. This approach is the one endorsed by Erlang, which favours parallelism over locking [57]. It can be noted here that this model presents some similarities to the publish-subscribe message passing 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 [55]. 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 [6], one of the fundamental mathematical models for concurrent computation (Figure 2.3).

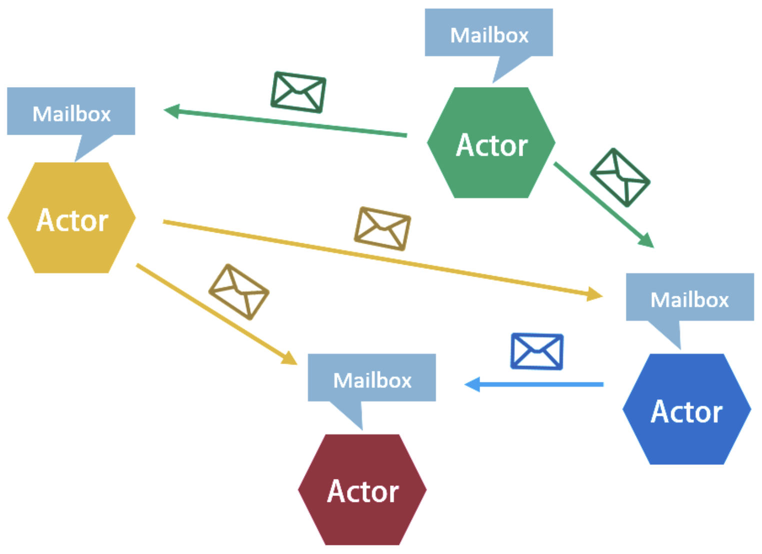

Figure 2.3: Actor model

Designed to avoid some of the problems posed by threading and locking (such as deadlocks and race conditions), the actor model presupposes that each object in the system is an actor and that all communications between actors are achieved via asynchronous message exchanges. The implication of this is that actors can function fairly independently of one another, without being required to wait or ”block” until their messages have been received by the other parties. Furthermore, asynchronism presupposes guarantees regarding at-least-once delivery [1], but offers no method of enforcing message order.

While seemingly quite an abstract concept, actors are in fact an intrinsic language feature in Erlang, implemented as spawnable processes and scheduled in a pre-emptive fashion [55]. This scheduling is performed in a concurrency-specialised form within the Erlang Virtual Machine (VM) [72], rather than being delegated to the operating system. This is due to the fact that most operating systems exhibit strong variations in scheduler performance, owed primarily to their design that is oriented towards general-purpose computing.

The Erlang VM spawns scheduler thread on every core, each possessing a run queue that defines the order in which Erlang processes get to execute in their time slices. The VM ensures that no core is overloaded by performing rebalancing operations, thus avoiding processing bottlenecks. When coupled with the lightweight nature of Erlang processes, which makes spawning, context switching and de-spawning fast, this feature of the Erlang VM means computation is highly scalable [57]. Within the hardware limitations of the machine or cluster of machines it runs on, Erlang can achieve near-linear performance scaling, meaning that given an adequately parallelised program, one can expect a smooth increase in performance, as computation power (e.g. processor cores) also increases.

Fault-tolerance is also accounted for in Erlang's design, embodied best by the "let-it-crash" coding philosophy [71]. By discouraging programmers from developing defensively, Erlang shifts the burden of error handling to the VM, where termination and subsequent restart of a malformed process is much preferred over salvaging it.

Nevertheless, Erlang endeavours to isolate these errors firstly through the avoidance of shared memory, as mentioned previously, and secondly through a well-constructed process supervision model [50]. This model allows for Erlang processes to be designated as supervisors [10], keeping track of the behaviour of worker processes under their watch and ensuring they can be shut down or restarted accordingly if they hang or die unexpectedly. A supervisor process can oversee multiple worker processes and can in turn be supervised by another process.

A further attribute of Erlang that exhibits some similarities to supervisors comes in the form of monitor processes [15]. Unlike supervisors, they are not responsible for the operation of any other extrinsic processes, but nevertheless require notifications in case these independent actors encounter failures.

Erlang's capabilities for reliable distributed computation thus mean that it could find successful applications within the field of robotics. Its intrinsic reliability and efficiency of scale has the potential to facilitate both inter-component communication within a robotic system, as well as external control of the actuators from remote locations.

However, it is important to note here that not all systems in need of parallel distributed computation would benefit from an implementation in Erlang. Indeed, owing to its preference for small, lightweight processing units, Erlang does not perform well for problems involving costly numerical algorithms such as brute-force searches over a solution space [61] (forward and inverse kinematics [58], in the case of robots) or complex, GPU-intensive image processing operations. Therefore, Erlang implementations in this field should not be regarded as complete replacements for robotic frameworks, but rather as complementary communication modules.

### 2.3.3 ErlPort

For the purposes of communicating with processes running outside of the Erlang VM that may be written in other programming languages, a library called ErlPort [14] is used as part of the project. It makes use of the Erlang VM's port protocol to allow cross-language function calls to be made from Erlang to Python or Ruby, and vice-versa. In spite of its incipient state, it strives to provide adequate support for the sharing of custom data types and message passing between Erlang and Python/Ruby.

## 2.4 Prior Work

While the task of integrating ROS and Erlang is a novel approach, the merits of Erlang's powerful concurrency-oriented design in the context of robotics have been recognised in several previous academic endeavours.

The first of these took the form of a project undertaken by Corrado Santoro et al at the University of Catania in 2007 [67]. The goal of his work was to produce a complete robotic framework in Erlang for the control of an autonomous robot participating in the 2007 Eurobot competition [16]. The project greatly emphasised system modularity, which meant that the framework was constructed using a layered architecture to separate each concern, starting with the low-level control layers, governing physical movement (interfacing with native robot code) and sensor/actuator driving to the higher-level interpretation layers: sensor/actuator functionalities, control logic and planning/reasoning functions.

Erlang's concurrent features were thus harnessed on both fronts, being utilised both in the AI aspects of the project and in the task of efficiently parallelising the robot's control loops. While, traditionally, these modules would have been implemented in languages like LISP (common in artificial intelligence research [30]) and C/C++, respectively, Santoro's team asserts that Erlang can successfully merge both language's capabilities. While the source material makes no explicit mention any performance testing being conducted, the robot is shown to have delivered satisfactorily in the actual competition.

Santoro and his team later on went to create ROSEN [11], a robotic framework and simulation engine developed in Erlang similarly designed to separate generic functionalities from the idiosyncrasies of specific robotic systems.

A further project that set out to link Erlang with the field of robotics emerged in 2009 from the collaboration of Sten Gruner and Thomas Lorentsen at RWTH Aachen and the University of Kent [60]. Intended as a response to the waning popularity of computing science degrees in British universities, the project's goal was primarily educational and focused on building an interactive framework to teach Erlang via robotic simulation.

The software artefact produced as a result of the project was KERL [12], the Kent Erlang Robotics Libary. Here Erlang user level and middleware modules communicated with a robotic driver implemented in C++, which in turn made use of Player [33], a cross-language robot device interface for a variety of control and sensor hardware and Stage [34], a 2D simulation environment simulator. The pairing of all of these technologies created a framework which, in spite of being a research project with less conventional applications, was nevertheless successful in providing a starting point for robotics in Erlang.

The last major research endeavour in this direction — and the one project that most resembles the system described in this dissertation — is the 2009-2010 work of Geoffrey Biggs at the National Institute of Advanced Industrial Science and Technology (AIST) in Japan [52]. The initial goal of this project was to develop an Erlang tool for the monitoring and orchestration of a network of OpenRTM-aist [29] components. Much like the ROS framework mentioned previously, OpenRTM-aist is an open source robotic middleware which provides communication services to sensor and control programs designated as RT-components.

Following the completion of this tool, a much more in-depth investigation of Erlang's use in low-level robotic control was launched, culminating with the development of a complete re-implementation of OpenRTM-aist in Erlang.

### 2.4.1 Discussion

As shown by the paragraphs to follow, process of studying the development lifecycle of these prior undertakings offered a helpful perspective on the broader context of this project and helped cement a clearer understanding of the use cases of Erlang within the field of robotics. The salient feature that is common to all of the works mentioned above is the acknowledgement of Erlang as a suitable candidate for robotic research.

Both Santoro and Biggs' projects emphasised the importance of ensuring proper functioning of separate, yet dependent software components. In Santoro's case these were represented by the various robot control loops that passed data from one to the other (e.g. wheel control and speed sensing), while for Biggs these came in the form of intercommunicating RT-components. Similarly, this project produces an Erlang framework with high reliability compared to analogous ROS networks, with the potential of being extending to form an Erlang-to-ROS supervision system similar to Biggs' Erlang-to-OpenRTM-aist concept. As Biggs states in his project description [52], an ideal scenario would have the developer selecting appropriate responses to failure events, including the "re-wiring" of the network to exclude failing nodes, as well as the "hot-swapping" of faulty nodes.

While Gruner and Lorentsen's approach diverges from this project's aims, it nevertheless provides a solid foundation for furthering research into Erlang for robotics. The implied intuitiveness of KERL and its ease in interfacing with control frameworks for a diverse range of robots highlights the flexibility of the language and its as of yet not fully tapped potential as a robotics language.

However, one important aspect to note about all of the described projects is that they are all marked by a lack of explicit non-functional testing experiments. While it is implicitly presumed that each piece of software provided an adequate solution to the initial hypotheses or possessed certain advantages over the tools they were meant to emulate or replace (in the case of Biggs), these are never clearly described. Whether this omission is deliberate or simply a result of lacking documentation, it represents a clear difference from the goals of this real-time camera control project. In this work, pomparing and contrasting the performances of the two systems (ROS and Erlang) is paramount, as the analysis provides concrete evidence of the potential advantages of Erlang.

# Chapter 3

# Design and Implementation

This chapter describes of the design and development of the ROS and Erlang-based face tracking programs and how the project's implementation goals evolved to produce the final design. Additionally, it establishes the salient features of these two pieces of software and draws parallels between them, where appropriate.

## 3.1 Initial Design

The first step in the design process involved establishing a concrete set of initial functional requirements for the ROS and Erlang programs, including the specific tasks they were expected to complete:

- Acquisition of images from a Logitech QuickCam Sphere camera [36], the PTZ device used as an archetypal robotic component in the context of this project; this was envisioned to be achieved either through dedicated Logitech APIs or through third-party software.
- Face detection and localisation using pre-trained OpenCV face detectors
- Tracking control to drive the pan-tilt gaze direction of the camera based on the output from the face detector modules.
- User interface to report the state of the system and present a view of the images currently being captured, annotated with any detected faces and tracking data.

In their original formulation, these requirements presupposed the use of C++ as an implementation language for the ROS-reliant program, a notion that was also supported by the comprehensive documentation offered by the C++ releases of both ROS [41] and OpenCV [28]. While this initial conception helped outline discrete steps in the development process, some refinements were ultimately required in order for a viable implementation to be produced within the constraints of time and technology.

## 3.2 Requirements Refinement and Final Design

The first identified issue with the requirements above concerns the use of programming language. Although it is undeniable that C++ offers performance improvements over many other languages [4], its object-oriented paradigm was in the end deemed unsuitable for the purposes of this project, as the two developed systems are not intended to function as deployment-ready applications. The code produced as part of this research endeavour is strictly prototypical in nature and thus needs to be flexible and easily adjustable for each experiment to be conducted on it. For this reason, Python was favoured over C++ as the language of choice when implementing functions in the ROS space. Moreover, Python was utilized in the development of Erlang-space program modules responsible for specialised or resource-intensive operations (i.e. image acquisition, face detection, frame display and camera operation) both for consistency and since these types of computation are not Erlang's strong point.

One other aspect that is readily observable from the requirements is the emphasis placed on the systems' tracking component, despite this feature not forming a crucial part of the research. After careful consideration, it was decided that defining the Logitech camera's exact tracking behaviour in the context of concurrent access from numerous processes would require locking restrictions which would not only be incompatible with the asynchronicity of the message passing model, but also lie outside of the project's scope. Moreover, as mentioned previously, the camera is only intended to serve as a generic robotic component and thus allocating too much time to the development of very specific code facilities would have negatively impacted the number of experiments able to be carried out and detracted from the generality of the findings.

Therefore, for the purposes of simplifying the interaction with the camera device, a decision was made to aggregate the results produced by each face detection node or process via a dedicated function or process. This module is responsible for aggregating the faces received from the detectors into a mean face reading and issuing appropriate instructions to the camera. The classifier running on each detector is expected to only identify one or no faces in a frame, for similar simplicity reasons.

Taking into account the matters addressed above, the design plan was modified and expanded to incorporate a more detailed breakdown of the capabilities of the face tracking systems. The final design is illustrated in Figure 3.1.

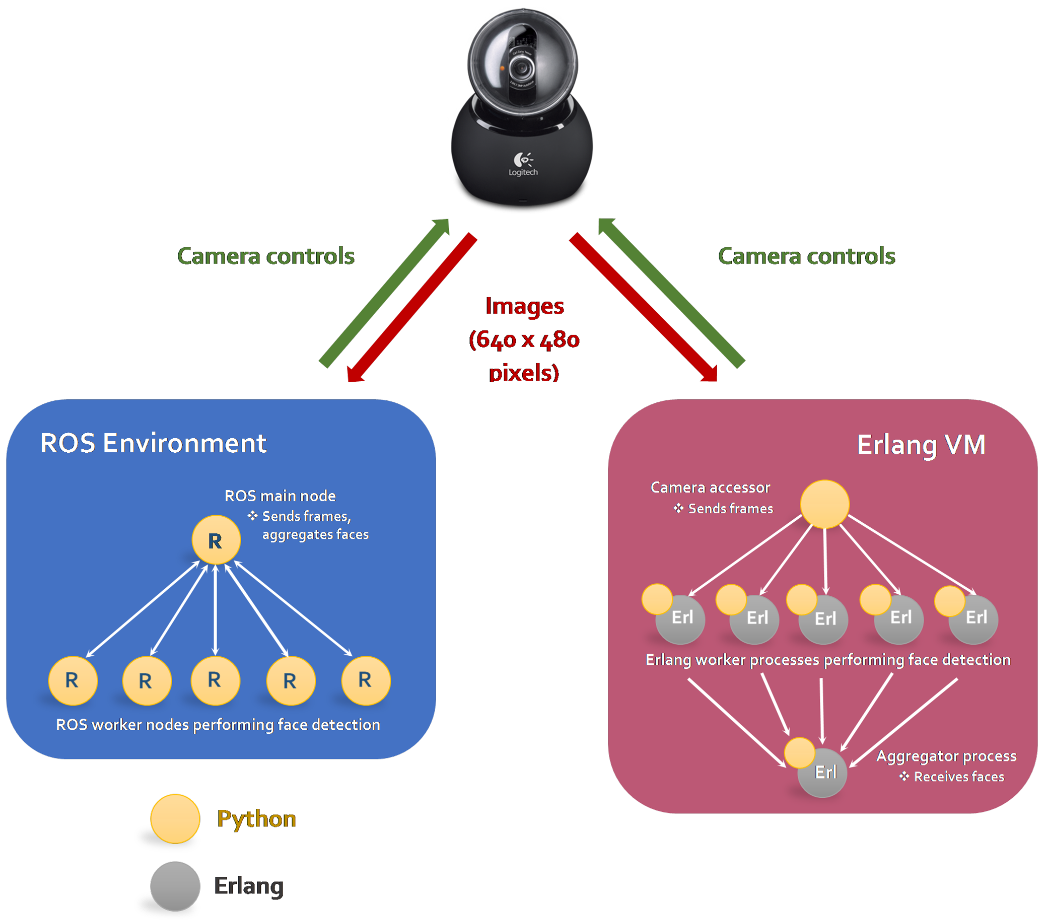
Camera controls
Camera controls
Images (640 x 480 pixels)
ROS Environment
ROS main node
Sends frames, aggregates faces
R
ROS worker nodes performing face detection
Erlang VM
Camera accessor
Sends frames
Erl
Erlang worker processes performing face detection
Aggregator process
Receives faces
Python
Erlang

Figure 3.1: ROS and Erlang face tracking architecture

The **ROS** face tracking program follows the client-server architectural model [32], with a main ROS node communicating asynchronously with a set of numerous worker nodes. The main node acts as an interface with the camera and is thus responsible for retrieving real-time images from the Logitech device and distributing them to the worker nodes. These nodes wait for frame inputs to be provided to them and subsequently perform OpenCV-powered face detection on the frames, resulting in a set of coordinates that denote the position of a face with regard to the frame dimensions. During the last step of the pipeline, information about the detected faces from each worker node is sent back to the main node, where the face coordinates are averaged and compounded into a single data point. The latter is then used to compute the physical adjustment required for the camera gaze to centre on the new face position and appropriate pan-tilt controls to be issued.

The **Erlang** face tracking program is conceptually similar in its design to the ROS-based architecture, yet requires more modules written in both Python and Erlang. This added complexity is justified, as the project aims to use Erlang only for the implementation of the message-passing middleware components that are automatically handled by ROS in its analogous system.
Hence, given this structuring, the code responsible for interfacing with the camera and performing the OpenCV face detection computations resides entirely in Python functions. However, rather than directly interacting, these functions are called from within the Erlang environment and all messages (i.e. frames, faces) are routed through Erlang processes. Unlike the ROS system, the Erlang program does not require the presence of a main actor to

capture and distribute frames to worker processes. It was reasoned that such a feature would only serve to add run time overhead by having the frame go through an additional, redundant "stop" before reaching the workers. Nevertheless, after using their corresponding Python functions to detect a face, the Erlang workers require a module that can aggregate these results and control the camera. This function is thus accomplished by an aggregator process linked to a Python function that issues camera commands. The aggregation behaviour is identical to that of its ROS-based counterpart function.

A key aspect examined in this design was the decision of how to distribute the frames to the worker nodes/processes. Two alternatives were considered in this regard: randomly decide which worker should receive a specific frame (effectively ensuring frames are never processed more than once), or broadcast every frame to all worker nodes and synchronise their detection results in the aggregation step. While the latter method incurs negligible detection accuracy penalties and is thus a better fit for a system which performs a mean-based face grouping operation, it also presents some downsides. Apart from increasing the volume of messages by a factor of $n$ (where $n$ is the number of workers), such replication also requires additional code that can determine when all of the results for a particular frame have been received at the aggregator before actuating the camera. Due to the complexity and added latency of this method, the former random frame cast practice was preferred. Broadcast-based distribution is however featured briefly in the face quality experiments, where it serves to expose a flaw in ROS' message synchronisation protocol.

With this functional structure in mind, the ROS and Erlang systems were developed as shown in the following section.

## 3.3 Implementation

### 3.3.1 ROS Application

The core of the ROS face tracking application is represented by the main node. Contained within a single Python script, the node initialises its operations by registering itself with the ROS master node under a unique node name, `face_tracking_main` . This is followed by the node establishing the parameters of its topic interactions (Figure 3.2). Thus, the main node creates $n$ topics, `camera_frames_i`, $0 \leq i < n$ — one for each worker node it expects to engage — and reserves them for the publishing of `Image`-type messages. Similarly, the node registers as a subscriber for $n$ face topics carrying `Int32Numpy`-type messages. The most important parameter of the subscription process is the callback function name, which specifies which of the main node's functions is to be invoked when a message is published to any of its subscribed topics. In this particular case, the callback function for all face topics is set to one single function responsible for aggregating the faces and issuing camera instructions.

Notably, the face topics need not exist in order for the subscription process to succeed, as the callback system employed by ROS ensures that subscriber-topic interaction only occurs when a message is published. As regards the two message types mentioned, `Image` is an in-built ROS message type meant to efficiently encode pixel matrices for publishing to a ROS topic and is thus highly suited for wrapping frames retrieved from the camera; `Int32Numpy`, on the other hand, is a custom message type created solely for the purposes of this application, which wraps a set of four integers of type `int32`, a numerical type specific to the Numpy Python library [26]. The creation of this latter type was found necessary in order to convey the structure of face location information, discussed further in the description of worker nodes.

```
rospy.init_node('face_tracking_main')

cap = cv.VideoCapture(device_ID)
# Create frame topics
publishers = []
for i in xrange(expected_workers):
        publishers += [ rospy.Publisher('camera_frames_'+str(i),
                        Image,
                        queue_size=1000) ]

# Subscribe to face topics
subscribers = []
for i in xrange(len(publishers)):
        subscribers += [rospy.Subscriber('camera_faces_'+str(i),
                         Int32Numpy,
                         aggregate_faces,
                         callback_args=[cap])]
```

Figure 3.2: Initialization of ROS main node

Having defined these message passing protocols, the main node proceeds to open a continuous camera feed using OpenCV and randomly selects a frame topic to which to publish each retrieved frame.

The second major component of the ROS application resides in the code used by all worker node instances. This module features an initialization process similar to the main node, registering itself under a unique name, `face_tracking_i`, $0 \leq i < n$, defining the topic to which it intends to publish the results of face detection and finally subscribing to its dedicated frame topic. The name of the callback function for the latter operation is in fact the name of the worker module's only function, `aggregate_faces`, that carries out face detection.

Once a frame is published to the frame topic, the face detection function begins to process it, calling upon the `detectMultiscale()` function in the OpenCV library. The function applies a classifier to a supplied image and returns a list of rectangles denoting deduced face positions. Each of these rectangle is composed of four Numpy integer elements, `[x_coord, y_coord, width, height]`, which aim to characterise the location of a face with respect to the image plane, as follows:

- `x_coord` — the x-coordinate corresponding to the top left point of a rectangle that the detector assumes circumscribes a face
- `y_coord` — the y-coordinate corresponding to the top left point of the same rectangle
- `width` — the width of the rectangle
- `height` — the height of the rectangle

To ensure that only one face is returned as a detection result, the list of rectangles is condensed to a single value, basing off the assumption that due to the nature of the cascade classification process, the largest rectangles are the likeliest face candidates.
After this step concludes, the resulting face data can be sent back to the main node by publishing directly to the worker's dedicated face topic. Since the four face components correspond exactly to the elements of the `Int32Numpy` message type accepted by the topic, no conversion is needed.

The function designated to receive faces on the main node front is the aggregator, whose role is to manage the large influx of face readings and determine what movements the Logitech device should execute. In order to fulfil this task, a deque data structure of variable capacity is utilised. Declared as a global parameter, the deque is gradually filled with each incoming face until it reaches its maximum number of elements, at which point the oldest element is discarded and the deque's "tail" is shuffled backwards to make space for a new face. This sliding window approach to face storage is then paired with an aggregation operation, whereby the face coordinates currently in the deque are averaged to produce a single, "best" face reading. Consequently, these features help circumvent the issue of conflicting concurrent camera access and ensure that the resulting face is a reasonable reflection of the real face position at the time of aggregation.

Lastly, a ROS `.launch` file is used to start both the main and the worker nodes of the application, as well as the sole `roscore` node with which all active nodes must register. This XML-formatted file serves as a configuration specification for the program, informing ROS of where the nodes are located (i.e. which package and which host/machine), what user-defined parameters they require, notably, whether their alive status should be monitored. This last aspect is particularly significant when it comes to protecting the system from unexpected failures, as shown in the evaluation section on reliability experiments.4

### 3.3.2 Erlang Application

Mirroring the previous description of the ROS system, the Erlang-based program is also structured as two primary modules: one to correspond to each instance of a Python worker processes and one to encompass the aggregation operation. However, unlike the ROS scripts, these modules are exclusively tasked with performing message-passing. Diverging from ROS' free-form code design, the two Erlang modules follow a pre-defined template called a "behaviour" [9]. Behaviours are the Erlang equivalents of interfaces in object-oriented languages and are thus employed in order to specify what types of interactions and potential failure modes the compiler can expect from an actor.

For the purposes of this application, a standard `gen_server` [8] (generic server) behaviour was used to model the server sides of the program's client-server relations. Accordingly, the clients of the system are represented by Python functions that deal with image acquisition and processing, as well as camera interaction. Here it is valuable to note that the functional equivalence of the ROS and Erlang applications considerably facilitated code reuse between the two face tracking systems, meaning that the Python modules developed for use in ROS space required only some re-structuring in order to fit as Erlang clients. Since these implementations were already covered in the previous section, the dissertation does not go into further detail on the Python functions, instead shifting the focus to the Erlang `gen_servers`.

The first of these modules, `face_server`, listens for incoming frame messages from the Python camera accessor and relays them to a Python face detector. While the generic server specification provides code facilities for both synchronous and asynchronous messaging — through the `handle_call` and `handle_cast` functions, respectively — only the latter aspect is used, to better parallel ROS' topic-based interaction (Figure 3.3). In order for the Python interaction to take place, upon initialization the face server creates an instance of a Python interpreter via the ErlPort library. A reference to this instance is then passed to each of the server's functions as a "state" parameter.

```
% Asynchronous handler for incoming frames;
% frames are pattern matched to ensure correctness
handle_cast({frame, Frame}, PyInstance) ->
        % Call the Python detector, supplying the frame and the Erlang Process ID
        python:call(PyInstance, facetracking, detect_face, [Frame,self()]),
        {noreply, PyInstance};

% Synchronous message handler;
% set to issue no replies
handle_call(_Message, _From, PyInstance) -> {noreply, PyInstance}.
```

Figure 3.3: Implementations of standard `gen_server` functions for the `face_server`

Whenever a message is sent (or cast) to the face server, the module matches the message against the {`frame, Frame`} pattern, where `frame` is an immutable named constant (called an Atom [57] in Erlang) and `Frame` is a variable expected to hold the list of pixel values. The role of this operation is both to verify that the message was sent by the camera accessor and that it contains the expected image data. Following this step, `handle_cast` performs a call to the Python detector function via the Python interpreter. The call process admits the passing of parameters of virtually any type to Python functions and is performed in a blocking manner, such that the Erlang caller only resumes operation once the Python function returns. Each face server has its own associated Python interpreter, thus the server actors can easily run independently from one another.

Likewise, the aggregator module, `aggregator_server` makes use of its defined `gen_server` behaviour and a server state parameter to communicate with Python functions outside of the VM, with the exception that in this case the server pattern matches {`face, Face`} messages received from all of the Python detectors and sends the aggregated face position to a Python tracking control function. A prominent feature of the aggregator server's state is that it is a list data type, in which the first element is its associated Python instance and the second is a queue structure, globally shared amongst the server's functions and designed to store the faces before aggregation. While Erlang does not offer a built-in deque data structure in the same way Python does, its functioning is simulated through manual updates performed on a simple queue. Hence, once a face message is cast to the aggregator, the server pushes it onto the queue and computes the same mean-based clustering operation as in the case of ROS system. The last step of the process involves the relay and translation of these final face coordinates into usable camera control instructions, via a call to a Python function.

Before the Erlang system can be started through a command-line interface, some consideration needs to be given to providing the same reliability guarantees as found in the ROS environment. This assurance is given by the creation of a supervision tree around the system's modules, composed of processes that implement the `supervisor` behaviour [10]. As seen in the diagram below (Figure 3.4), the tree consists of two main supervisors that monitor the execution of the numerous face server processes and the aggregator server process, respectively, and one top-level supervisor. The latter supervisor oversees the functioning of its direct children and is a simplified equivalent ROS' `roscore` node. All of the tree's vertices are configured as `permanent` children, implying they must always be restarted upon failure. Additionally, the supervisor modules offer more customisability than the ROS `.launch` script, allowing for different restart strategies to be employed. This implementation makes use of the strategy called `one_for_one`, whereby, in the case of failures occurring in a multi-child supervisor, only the failed children are restarted. The functioning of these reliability mechanisms is validated in the second stage of the evaluation.

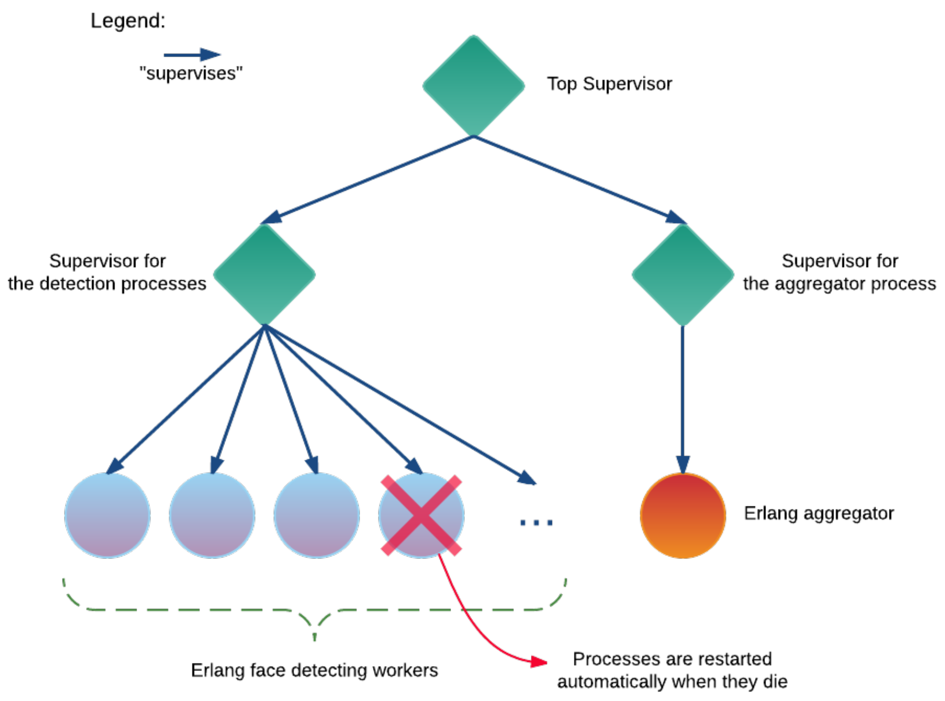
Legend:
"supervises"
Top Supervisor
Supervisor for the detection processes
Supervisor for the aggregator process
...
Erlang aggregator
Erlang face detecting workers
Processes are restarted automatically when they die

Figure 3.4: Supervision in the Erlang face tracking application

### 3.3.3 Implementation Comparison

Through empirical testing and subsequent extensive refinements, the two face tracking applications were determined to be functionally equivalent, relaying frame and face messages across their respective node and process networks. Table 3.1 summarises the implementation details of the ROS and Erlang systems, evidencing that the Erlang implementation requires more code facilities than its ROS analogue. While this may appear disadvantageous, the discrepancy is justified by the fact that Erlang is only a development tool, rather than a fully-developed ecosystem like ROS. Hence, communication and reliability facilities such as topic interaction and the `roslaunch` service that are incorporated into ROS, had to be manually implemented in the corresponding Erlang program.

| | **ROS system** | | | **Erlang system** | | |
|---|---|---|---|---|---|---|
| | Processing | Communication | Reliability | Processing | Communication | Reliability |
| **Modules** | 2 | 1[*] | 1 launch script | 2[**] | 2 | 3 supervisors |
| **Lines of code** | 194 | 1 | 22+ | 104 | 139 | 81 |
| **Total** | 200+ lines of code | | | 300+ lines of code | | |
| [*]Communication is handled implicitly by ROS, but the `Image32Numpy` message type was custom added. | | | | | | |
| [**]One Python module for processing and one Erlang interface module to start the application. | | | | | | |

Table 3.1: ROS and Erlang code comparison

# Chapter 4

# Evaluation

This chapter presents and discusses the scalability and reliability experiments carried out on the ROS and Erlang face tracking systems, establishing the specifications of the testing environment and the parameters used in each test. It also ventures into an analysis of the findings that resulted and examines what impact they may have in the context of Erlang middleware usage in a real robotic setting.

## 4.1 Hardware Context and Variables

All of the five experiments were performed on a machine powered by an Intel i7-4700HQ (4 cores, 2 threads per core) processor, at 2.4 GHz, with 2 x 8GB DDR3 memory chips at 1.6 GHz, running 64-bit Ubuntu 15.04. These specifications form the basic hardware constraints of the system and impose performance limitations on the face tracking programs, particularly with regard to scalability. On the software side, the ROS Indigo Ingloo dsitribution was used, alongside Erlang/OTP 18.0 and OpenCV 3.0.

Prior to the formulation of the experiments, a list of relevant of controlled variables and parameters present within the code of the ROS and Erlang applications was compiled and appropriately equivalent values were set for each variable. This was done in an effort to eliminate all bias in favour or against one of the programs and thus ensure that the results derived from each test are valid and meaningful. This list of variables is given in Figure 4.1.

Simultaneously, there exist some factors influencing the testing process which were not accounted for. Mainly related to the intrinsic attributes of the software technologies used, these aspects nevertheless introduce restrictions on the conclusions that can be drawn from each experiment. For instance, the use of ErlPort — a library still in its incipient development stages — as a coupling layer between Erlang and Python modules is thought to both introduce latency in the Erlang program, as well as alter its behaviour when processes are started. Additionally, in ROS space, the considerable size of the images captured by the camera (640x480 pixels) and the accordingly sizeable message published to the topic limits the publishing rate to 5 Hz.

| Variable Name | Variable Description | Value |
|---|---|---|
| **Variables common to both ROS and Erlang** | | |
| **Device ID** | The name under which the operating system identifies the Logitech Orbit Sphere camera port. The port labelling is the choice of the OS and thus this variable must be manually set. | 0 or 1 |
| **Cascade Classifier ID** | The name of the Haar features-based cascade classifier used by both programs during their OpenCV face detection stage. | haarcascade_frontalface_1 |
| **OpenCV Detector Arguments** | The list of arguments passed to the detection function: `scaleFactor` (how much the image size is reduced at each step), `minNeighbors` (the confidence of face detection (higher values mean a more rigorous selection process), `minSize` (faces smaller than this minimum rectangle size are ignored). | `scaleFactor` = 1.1, `minNeighbors` = 6, `minSize` = (50,50) |
| **Aggregator Deque Length** | The size of the sliding window the aggregator should take into account. This parameter was experimentally found to yield the best results at low values, since having more faces to aggregate means the average face value will "lag" behind the real-time position. | 5 |
| **ROS-specific Variables** | | |
| **Publishing Rate** | The rate at which messages are published to a topic | 5 Hz, as imposed by `rospy` message transmission latency. |
| **Topic Queue Size** | How many messages are temporarily stored by rospy if it cannot immediately publish all of them to the topic. Generally recommended to be the same as the publishing rate. | 5 |
| **Erlang-specific Variables** | | |
| **Supervisor Child Specification** | Child configuration parameters: `restart`, `shutdown`, `type` | `restart` = permanent (children are always restarted upon failure), `shutdown` = 20 ms (children that do not respond within 20 ms to an exit summons will be automatically terminated), `type` = worker/supervisor |
| **Supervisor Restart Strategy** | Which children are restarted in the instance of failure in the system. Quantifies the importance of partial failures. | `one_for_one` (if one child process terminates and should be restarted, only that child process is affected) |
| **Maximum Restart Intensity** | How many restarts are allowed within a specific period of time before the entire program is deemed faulty and shut down. | 1000 restarts in 1 second |

Table 4.1: Variable list

## 4.2 Experiments

The process of designing the experiments ran parallel to the development of the two systems and was similarly iterative, undergoing changes as the programs evolved. Each experiment addresses a specific hypothesis and is formulated as a sequence of concrete steps that aim to define the circumstances of the testing and what external action is required.

### 4.2.1 Concurrent Worker Limits Experiment

This experiment aims to determine how many face detection worker nodes and processes, respectively, the ROS and Erlang applications can feasibly support. In this context, ”feasible” denotes a system capable of functioning correctly, without severe strain on the operating system and without spontaneous crashes.

#### Hypothesis

Given that Erlang is specifically engineering to support a large number of concurrent processes, the expectation is that the Erlang face tracking program is able to support a much larger number of detection processes before encountering impaired performance and partial or complete failures, possibly 2-3 times more than the equivalent ROS version.

#### Method

The methods for undertaking the experiment were as follows:

1. Start an instance of the ROS program running 1 worker node and verify that it operates without failure.
2. Sequentially start instances of the ROS program running increasingly more worker nodes. Worker numbers should increase in increments of 50.
3. Execute the second step until the system experiences failures occur and note how many workers it was running at the time.
4. Follow the same procedure (1. - 3.) for the Erlang application and its face detection processes.
5. Compare the resulting data to establish the maximum supported worker count for each system.

#### Findings and Conclusion

At the point of its conclusion, the experiment yielded a promising set of results, which served to confirm the hypothesis expressed earlier that Erlang outperforms ROS when it comes to the maximum simultaneously active workers. This is shown in Figure 4.2.

Additionally, the Erlang system exceed the initial expectations by exhibiting stable operation at up to 700 detection processes. By contrast, the ROS-based application could only scale to 200 worker nodes — 3.5 times less than Erlang — until persistent failures began hindering its execution. In the case of the latter system, these

| Maximum number of active agents | |
|---|---|
| **ROS system** | **Erlang system** |
| ˜200 worker nodes | ˜700 worker processes |

Table 4.2: Concurrent worker limits

| Memory consumption at 50 agents | |
|---|---|
| **ROS system** | **Erlang system** |
| Python main node: ˜64 MB x 1 instance | Erlang VM: ˜200 MB x 1 instance |
| Python worker nodes: ˜50 MB x 50 instances | Python interpreters: ˜22 MB x 50 instances |
| Total: 2,564 MB | Total: 1,300 MB |

Table 4.3: Memory analysis

failures assumed the form of unspecified Xorg (a display server application for Ubuntu distribution) errors, while the primary cause of failure for the Erlang application was the reaching of the OS' maximum open socket limit. Although measures were taken to drastically increase this limit, the persistent resurgence of the error was ultimately attributed to poor socket recycling on the part of the ErlPort library.

In order to further investigate the factors determining these limits, memory profiling of the two systems running 50 agents was done, by examining how much memory each of the systems' agents consume throughout their runtime. The analysis, illustrated in Figure 4.3, shows that both the main and the worker ROS nodes are fairly resource-intensive. On the other hand, the exact resource consumption of each individual Erlang process could not be accurately determined, due to abstraction resulting from the self-contained nature of the Erlang Virtual Machine. However, assuming equal distribution it can be extrapolated that the Erlang middleware consumes approximately 200MB$/$50 processes = 4MB of memory per process.

The results in Tables 4.2 and 4.3 indicate that the Erlang application scales better than RO and uses less memory, thus rendering Erlang in a very favourable light as far as scalability matters are concerned. Hence, the general conclusion that can be derived is that Erlang may be successfully employed both in large-scale robotic systems and also in modular robots whose number of components is envisioned to grow over time.

### 4.2.2 Message Latency Experiment

The intent of this experiment is to quantify the systems' performance as the number of nodes and processes increases. This is done by measuring the time each message spends in transit from one stage of the processing pipeline to another. Thus, in the case of ROS, the verified latencies are the ones incurred by message passing between the camera and each face detection node and between the face detection node and the aggregator function. In the case of the Erlang application, measurements are made of the delays between the Python camera accessor and the Erlang face detection processes, between the Erlang face detection modules and the Python detector and between the Python detector and the Erlang aggregator process. Additionally, the average time taken for the OpenCV detection to execute is measured for each system.

#### Hypothesis

The expectation behind this experiment is that Erlang system will exhibit lower overall latencies (from initial frame capture to final face position) on account of Erlang being explicitly engineered for streamlined message passing.

### Method

The steps involved in running the experiment were as follows:

1. Amend the ROS face tracking modules with facilities for logging round-trip message time, ensuring that time stamps are issued:
    1.1. ...before the image is published to the intended frame topic
    1.2. ...when the image is received at the worker node
    1.3. ...when the OpenCV detection process starts
    1.4. ...when the OpenCV detection process ends
    1.5. ...before the detected face is published to the worker's face topic
    1.6. ...when the face is received at the aggregator function
2. Similarly, create means for the Erlang face tracking application to time the processing pipeline, by issuing time stamps at these points:
    2.1. ...before the image is cast from Python to the intended Erlang worker process
    2.2. ...when the image arrives at the worker process
    2.3. ...when the image leaves the worker process via a function call to the Python detector
    2.4. ...when the image is received at the Python detector function
    2.5. ...when the OpenCV detection process starts
    2.6. ...when the OpenCV detection process ends
    2.7. ...when the detected face is cast to the Erlang aggregator process
    2.8. ...when the face is received at the aggregator process
3. Sequentially start instances of the ROS and Erlang programs running increasingly more workers and process at least 700 real-time faces in each run. Worker numbers should increase in the following increments: 1, 2, 3, 4, 5, 6, 7, 8, 9, 100, 200 and 700 nodes/processes.
4. For each system, compute latencies based on differences between the recorded time stamps.
5. For each system, average the first 700 latency numbers.
6. Compare the resulting data to establish where Erlang outperforms ROS and vice-versa.

### Findings and Conclusions

This experiment resulted in surprising findings, as shown in the set of Figures 4.1 - 4.8, plotting the calculated latencies for each system. Contrary to the hypothesis, the overall relay time of messages passing through the Erlang system was approximately identical to the overall latencies experienced by the ROS system, in all instances running workers numbers within the 1-9 range. What's more, starting at the 9-workers mark, the relay time increased sharply for the Erlang system, while the ROS delays continued to stay consistent until its maximum limit of 200 workers was reached.

As can be seen in Figures 4.5, 4.6 and 4.7, the factor with the highest impact on the overall latency of both systems appears to be the detection time, with most other communication delays registering consistently as 0 - 1 milliseconds. However, the second Erlang system graph (Figure 4.8) shows a marked spike in the time taken for the function call from Erlang detector processes to the Python module to be made, beginning at the 100-process mark. Thus, at the 200-process mark — the limit of the ROS system — Erlang is 1.5 times slower at relaying messages across its process network. While a possible explanation for this anomaly could relate to the size of the message being communicated (large pixel arrays), the same phenomenon is not observed in the camera to Erlang process relay, which deals with similar message sizes and is also performed via socket I/O. Thus, the discrepancy was ultimately attributed to faults in the behaviour of ErlPort's socket interfaces.

As illustrated in Figure **??**, the delay induced by the OpenCV face detection operation is identical (within a reasonable error margin) for both the ROS and the Erlang systems and thus does not account for any performance differences. Both detection functions perform similarly, with the slightly higher ROS values being attributed to an additional step the detector must perform to convert the frame from an `Image`-type topic message to a usable pixel matrix.

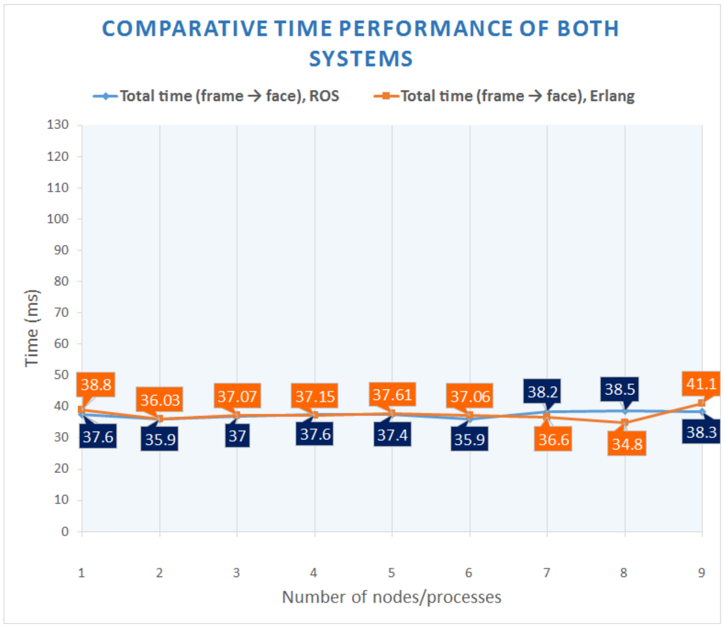

Figure 4.1: Comparative system timing

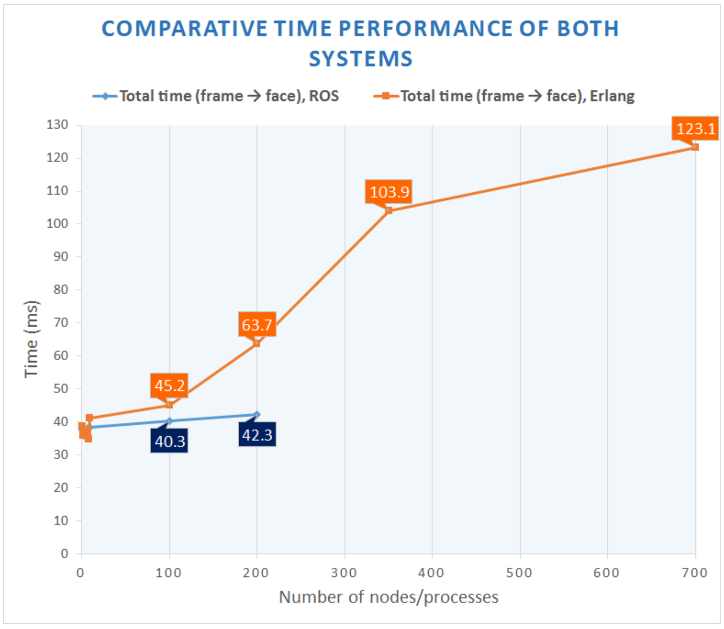

Figure 4.2: Comparative system timing (cont.)

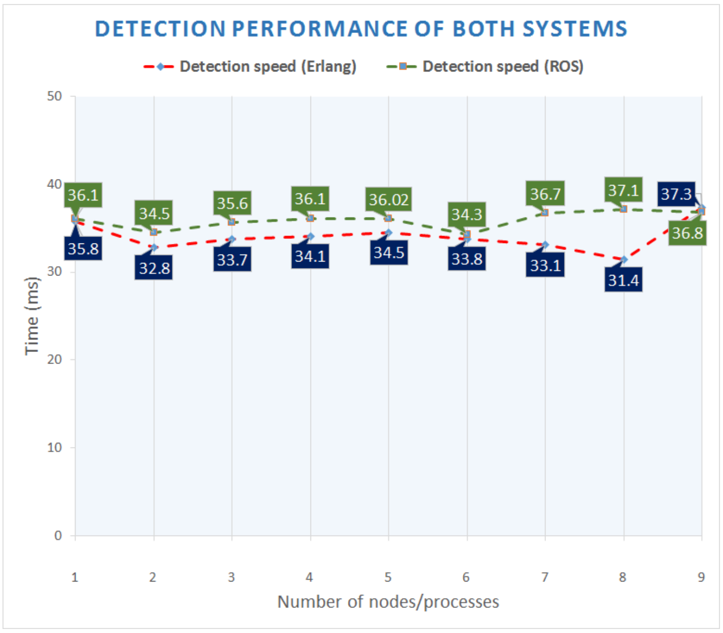

Figure 4.3: Comparative detection timing

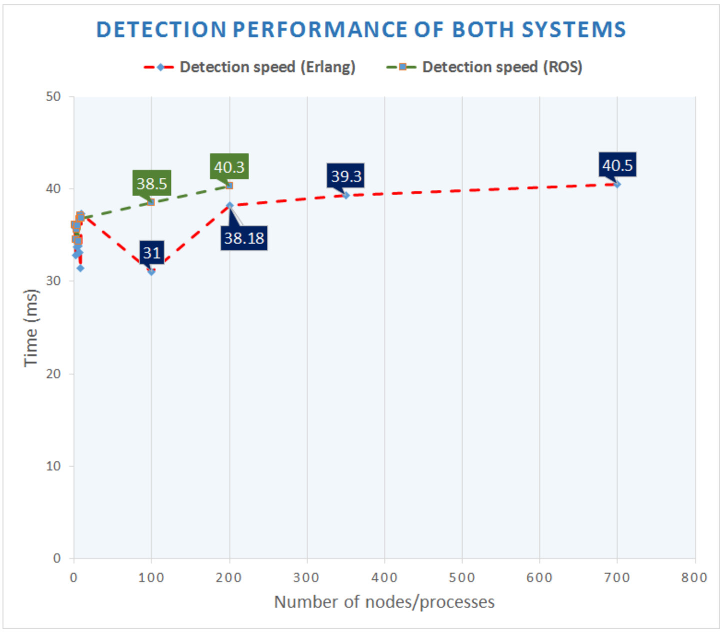

Figure 4.4: Comparative detection timing (cont.)

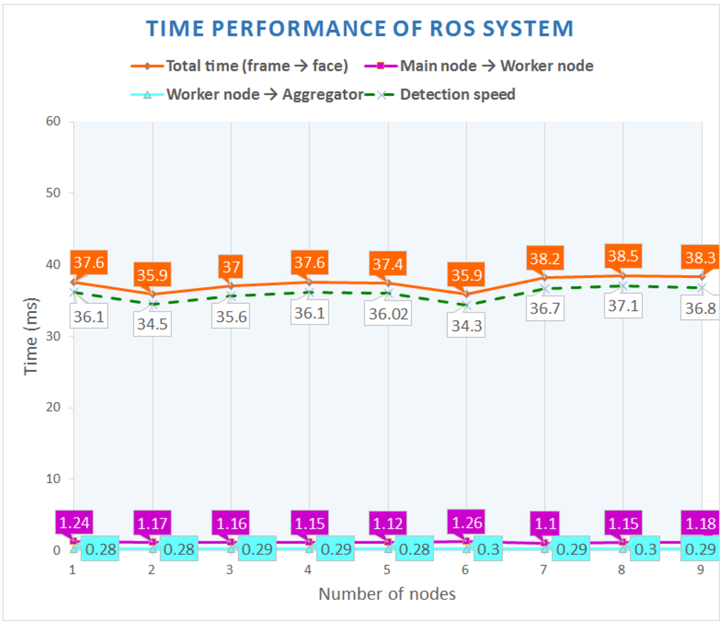

Figure 4.5: ROS system timing

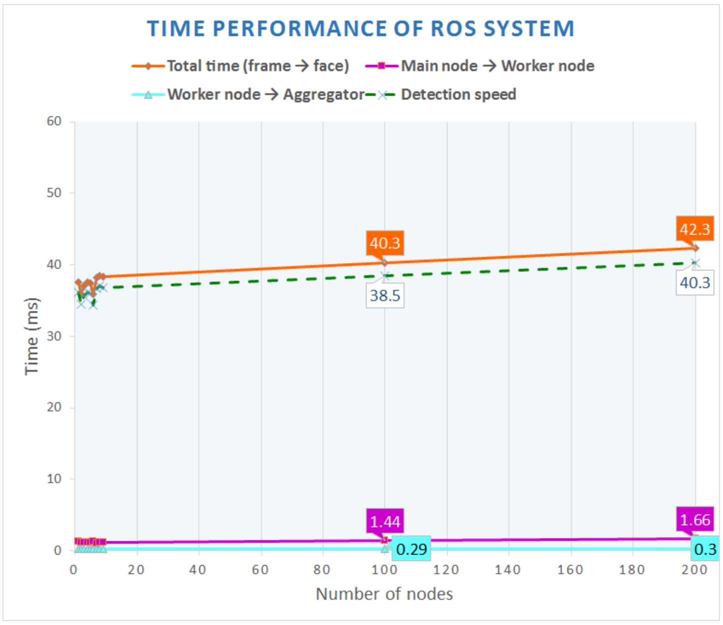

Figure 4.6: ROS system timing (cont.)

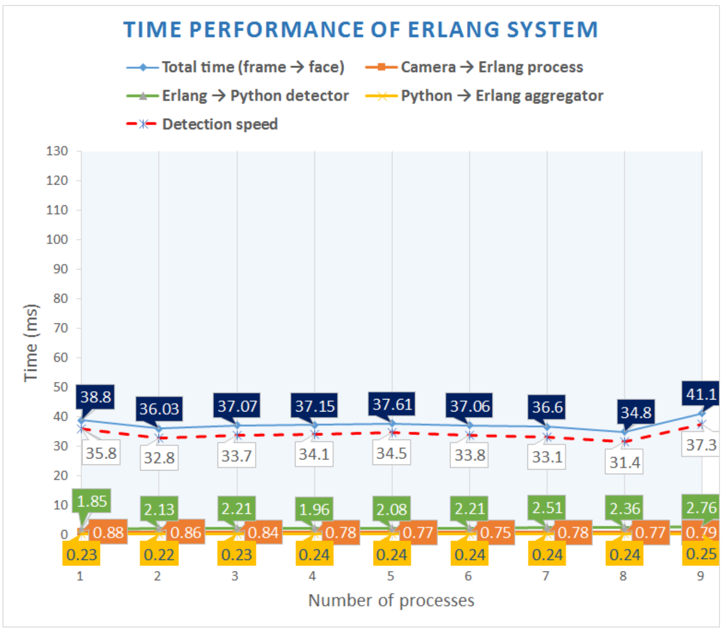

Figure 4.7: Erlang system timing

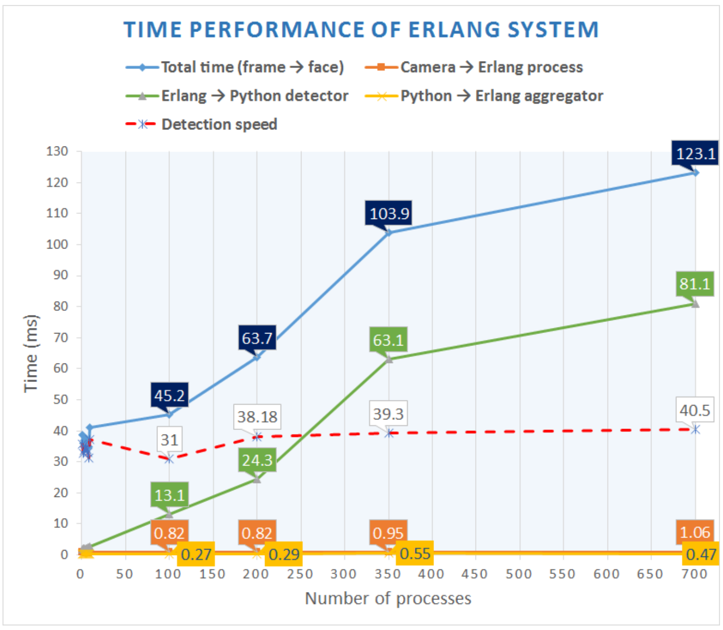

Figure 4.8: Erlang system timing (cont.)

While ErlPort represents a convenient method of linking Python and Erlang, its immaturity as a VM interaction library appears to strongly reflect on the scalability of Erlang middleware implementations. Considering the partial incongruity of the Erlang system performance with the anticipated outcome, the most apt conclusion drawn from this experiment is that Erlang is able to offer performance that is at least as good as the ROS one, but great care should be taken in selecting the cross-language or cross-platform communication library. As exchanges within the VM are virtually costless, the communication bottleneck of a robotic system would thus reside in the interface to external environments such as sensors, actuators or computation units.

### 4.2.3 Face Quality Variations with Real-time Aggregation Experiment

The third experiment assesses the quality of the faces produced by multi-agent instances of the ROS and Erlang systems, investigating whether worker scaling affects this quality metric and to what extent these impacts differ between the two programs.

#### Hypothesis

This evaluation is the only one featured in this project that bases partially off a null hypothesis. Namely, it predicts that no variations should be evident between the two systems' performance, even as detection workers increase in number. The reason for this anticipated equivalence lies partially in the behaviour of the OpenCV face classifier, which — while not completely deterministic in its execution — is reasonably consistent in terms of results and partially in the randomised way in which frames are distributed to worker nodes. Thus, taking both of these factors into account, the programs should perform comparably, barring any implementation bugs.

Additionally, as mentioned in the design section, variants of the two programs that ensured all captures frames were received by every detector were also trialled, in an effort to ascertain whether detection performance scales linearly with the number of agents. The assumption in this case was that, with increasingly more face results being aggregated for each frame, both the ROS and the Erlang-based applications should experience a steady climb in the quality of detection.

### Method

The steps involved in running the experiment were as follows:

1. Create a set of 100 (image, file) pairs, where the images are snapshots taken by the Logitech camera (Figure 4.9) and the files contain the coordinates of the primary face present in the image, determined through OpenCV detection with the same `haarcascade_frontalface_1` classifier.
2. Replace the real-time camera feed of the ROS and Erlang programs with the prepared set of 100 images and run instances of each program with agent numbers increasing as follows: 1, 100 , 200 nodes/processes.
3. For each run of each program, record the 100 face positions that are output.
4. For each test image, compare the accuracy the face position issued by the ROS application against the ground truth reading established in step 1. This is done via a numerical quality metric derived from the Jaccard index [64], that computes the similarity of two sample sets. This variant metric, $J_Q$, replaces the sets with rectangle areas, as per the formula:

   $$J_Q(A_{detected}, A_{actual}) = \frac{A_{overlap}}{A_{detected} + A_{actual} - A_{overlap}},\ J_Q \in [0, 1],$$

   where $A_{detected}$ is the area of the rectangle describing the ROS-predicted face position, $A_{actual}$ is the area of rectangle denoting the definitively established face location and $A_{overlap}$ is the area of their overlap (Figure 4.10).
5. Repeat step 4. for the Erlang program.
6. Classify each index by its corresponding frame, program version (ROS/Erlang) and worker node count and average the indices.
7. Compare the resulting averaged data to establish if and by how much the two implementations differ in the context of face quality.
8. Attempt steps 1. - 7. for a versions of the two face tracking applications that used a broadcast approach to distributing frames to each detector.

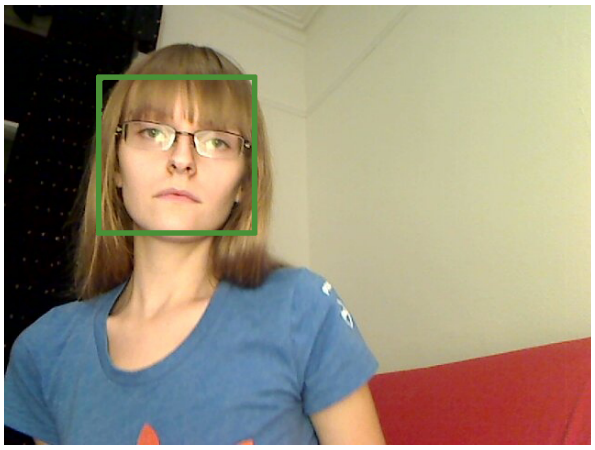
Figure 4.9: Example of an image from the test set, with the ground truth detection evidenced in green

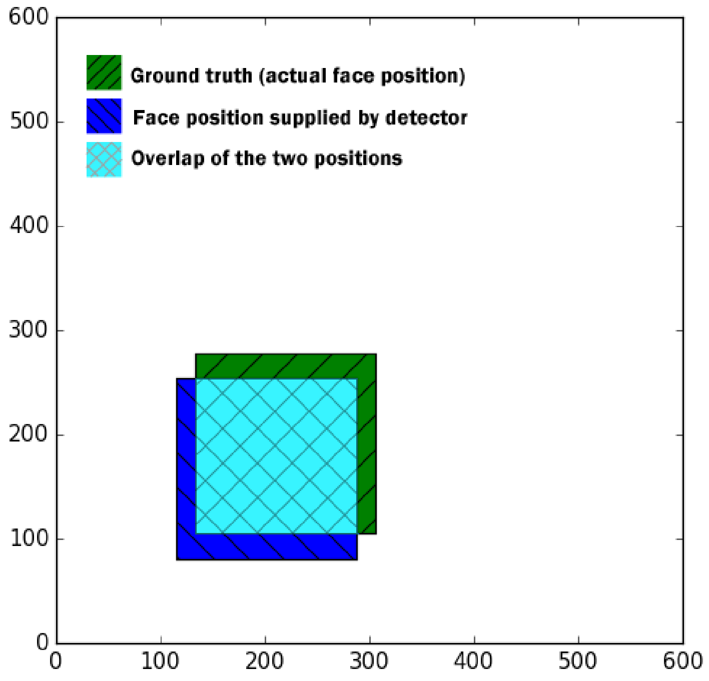

Figure 4.10: Areas used in computing the quality index for the face in Figure 4.9

**Findings and Conclusions**

The information gathered at the end of this testing phase was found to agree with the initial prediction that no major differences are evident between the quality indices resulting from the two detection processes (Table 4.4). However, a variable common to the systems, the Aggregator Deque Length, was observed to have a significant impact on face quality when faces were randomly circulated amongst detectors. This finding stems from the fact that the larger the deque size, the more ”past” faces get aggregated in the final face reading. Table 4.4 illustrates that if all output faces are considered individually (at deque size 1 and hence with no aggregation) the resulting $J_Q$ is very close to the expected ground truth index $J_{Q_{actual}} = 1$, while if aggregation is performed, the values of $J_Q$ drops with each increase in deque size.

To remedy this behaviour and verify the second part of the experimental hypothesis, an attempt was made to compile similar results from the broadcast-based runs of the two face tracking systems. This implementation variant simplified the ROS process of publishing faces by no longer requiring a dedicated topic for each worker node and instead placing all frames on a common frame topic all detectors could subscribe to. Notwithstanding, the evaluation of this method was faced with problems when it came to interpreting the data gathered from the ROS detector nodes. As described in the design section, the ROS main node subscribes to each of the worker topics via a callback mechanism, with the name of the single aggregator function supplied as the callback argument. While ROS ensures that the aggregator will be invoked every time a new message appears on either of the face topics, it provides no guarantees on the order in which these these calls are performed. In the context of this experiment, this was found to lead to disordered arrivals of the faces at aggregator (e.g. the face detected in frame #28 by worker #1 arrived after the face identified in frame #12 by another worker). This peculiarity made the task of pairing each face with its associated frame quite difficult in practice and postponed any further investigation, due to time constraints.

| Average indices of face quality | | | |
|---|---|---|---|
| System | Deque length = 10 | Deque length = 5 | Deque length = 1 |
| **ROS system** | 0.285 | 0.591 | 0.947 |
| **Erlang system** | 0.302 | 0.608 | 0.922 |
| **Difference (%)** | -5.62 | -2.79 | 2.71 |

Table 4.4: Indices $J_Q$ quantifying the quality of detection, averaged over 100 faces

Nevertheless, the experiment did succeed in establishing that Erlang's ability as a middleware to support application functions is not hindered by growth in the number of worker processes. This conclusion carries the implication that using Erlang in an robot control environment should not in theory pose any risks of unexpected and potentially hazardous failures or anomalous behaviours appearing in the message passing platform as more robotic components are integrated.

### 4.2.4 System Resilience Experiment

The first of the reliability experiments puts the system uptime guarantees of each of the two face tracking programs under scrutiny, by gauging how long the systems take to recover from random partial failures. This is done by programmatically targeting face-detecting worker agents and aggregation agents and measuring the latency between their shutdown and restart. Termination is done both by invoking the agents' standard exit procedures, through a standard termination signal, and also abruptly, by immediately ordering the kernel to terminate the agents.

#### Hypothesis

Initially, the expectation was that the ROS face tracking system would never restart any of its failed nodes, since ROS offers no reliability mechanisms when nodes are started individually, via the `rosrun` command. However, at the end of the literature review phase, it was discovered that nodes can be automatically monitored by the ROS platform, provided they are initialised via an associated launch script [43]. Consequently, the hypothesis of the experiment was revised to issue the prediction that, while both applications are capable of restarting their failed agents, Erlang should be able to do so faster, thanks to the lightweight nature of its processes and its intrinsic focus on reliability.

#### Method

The steps involved in running the experiment are as follows:

1. Develop a pair of scripts in Python and Erlang dedicated to terminating ”living” agents from their respective face tracking systems (ROS/Erlang). These programs are modelled after the Chaos Monkey [18] reliability testing service employed by Netflix for its computer systems.
2. Create the necessary code facilities to record and store initialization and shutdown times for each of the two face tracking systems.
3. On a ROS system running 100 worker nodes, instruct the Python chaos monkey program to execute 200 random node terminations at 1 second intervals.
4. Repeat step 3. for the Erlang program and its associated chaos monkey module.

5. For each system, compute latencies based on differences between the recorded time stamps.
6. For each system, average the 500 latency numbers.
7. Compare the resulting averaged averaged data to establish which of the two systems is faster at restarted its failed agents.

### Findings and Conclusions

As predicted by the experimental hypothesis, the results of this trial ruled greatly in favour of Erlang. By comparing the accumulated restart latency data shown in the logarithmic scale graph in Figure 4.11, it was discovered that Erlang restarts its terminated processes processes (both terminated normally and killed via a kernel invocation) in approximately 0.06% to 0.08% of the time taken by ROS for the equivalent action. Both the Erlang aggregator process and the worker processes are resumed with comparably small delays and were thus combined in the same average. The ROS system, on the other hand, features a discrepancy of around 50 - 100 milliseconds between the restart latencies of the main node and the worker nodes, respectively.

In the case of the Erlang face tracking application, the restart latencies of each of the supervisor processes were further examined, mainly to determine how these correlated with the delays incurred by the message passing processes. Hence it was observed that supervisors take approximately 0.100 - 0.140 milliseconds more to restart compared to their child processes, presumably on account of the VM's internal initialization of processes with `supervisor` behaviour. Here it is important to note that, upon termination, a supervisor's children are also shut down and must be restarted when the supervisor resumes operation. However, the latency figures cited in Figure 4.11 do not include the time taken for each child to be restarted. Much like the `roscore` node in the ROS environment, the operation of the topmost Erlang supervisor is not monitored by any process and thus failures in this supervisor are unrecoverable.

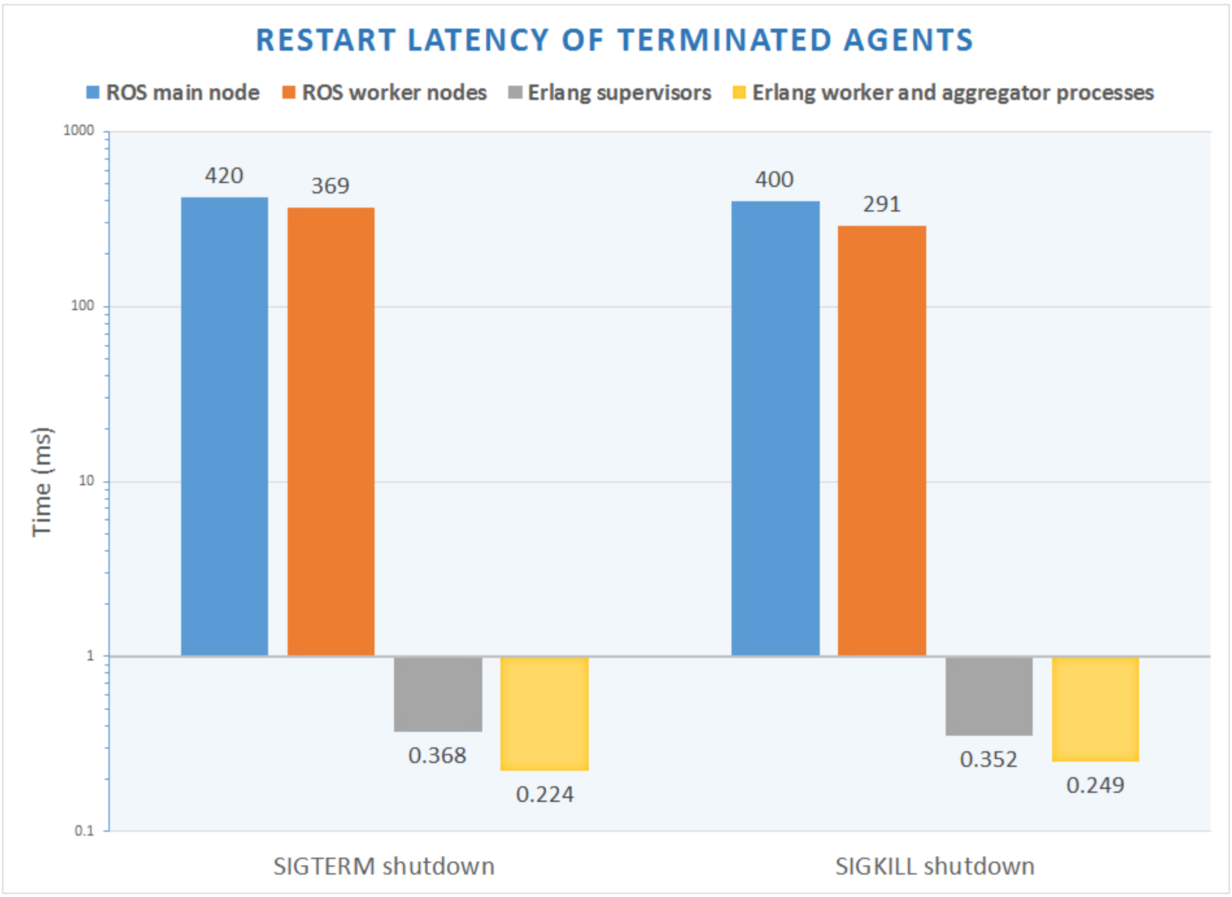

Figure 4.11: Comparison of ROS and Erlang recovery times

The significance of these findings in the context of robotics is that system uptime for robots running an Erlang middleware has the potential to be drastically higher compared to ROS-based machines. An illustrative example would be a robot with 100 components, where each component is likely to suffer 50 failures a year. Running under ROS, such a system would suffer circa 30 minutes of downtime a year, corresponding to an availability of 99.995%. By contrast, if this system were to make use of an Erlang-based framework, the downtime could fall to as little as 1.12 seconds per year, implying a theoretical availability of around 99.999995%.

### 4.2.5 Face Quality Variations with Worker Failure Experiment

The final experiment analyses the impact random partial failures have on the accuracy of the detection process, as executed by the two face tracking applications. For this purpose, the same set of 100 images used in the third experiment and the chaos monkey scripts created for the fourth experiment were brought together to asses whether spontaneous agent terminations cause significant drops in the quality of the detected faces, when compared to the ground truth face positions.

#### Hypothesis

In accordance with the hypothesis formulated in the previous experiment, the conjecture in this circumstance is that Erlang's lower restart latency results in markedly reduced disruptions to the detection process, with less pronounced quality variations than in the case of the ROS application.

## Method

The steps involved in running the experiment are as follows:

1. Making use of the existing test set of 100 images containing pre-computed faces, establish a simulated ”real-time” image feed to two instance of the face tracking programs, running 100 nodes and processes, respectively.
2. Instruct the ROS-targeted chaos monkey script to gradually terminate 10, 25, 50, 75 and 90 of the face-detecting worker nodes, with no intervals of pause between the terminations.
3. After each increase in the number of nodes terminated, record the 100 (or less) face positions that are output and restart the ROS application.
4. For each run and each test image, compare the accuracy the face positions issued by the ROS application against the ground truth readings, via the Jaccard similarity coefficient.
5. Perform steps 2. - 4. for the Erlang-based application and its associated chaos monkey module.
6. Compare the variations in face quality over time for each run of the two programs (i.e. the run in which 10 nodes were terminated, etc.)

## Findings and Conclusions

Similar to the results of the previous evaluation, this test showed that the Erlang program was more adept at stabilising the detection results being output, even while running in an environment experiencing sudden partial failures. The graphs presented in Figures 4.12 - 4.19 showcase a percentage-based representation of quality variations, as they were observed during each phase of the testing (i.e. with 10 failed nodes, 25, 50, 75 and 90). The forthcoming paragraphs shall henceforth refer to these phases as the **n-failures sub-tests**, for $n \in [10, 25, 50, 75, 90]$. The data series coloured in green represents the quality baseline quantified by the pre-determined face coordinates for each of the 100 images and is thus maintained at a constant 100% value. On the other hand, the data points residing on the red and the blue series symbolise all of the drops in face quality that were registered after the point of mass agent termination (normal and abrupt, respectively). The latter event is depicted as the purple dotted line labelled ”Point of termination” and was kept as consistent as possible within the context of each sub-test.

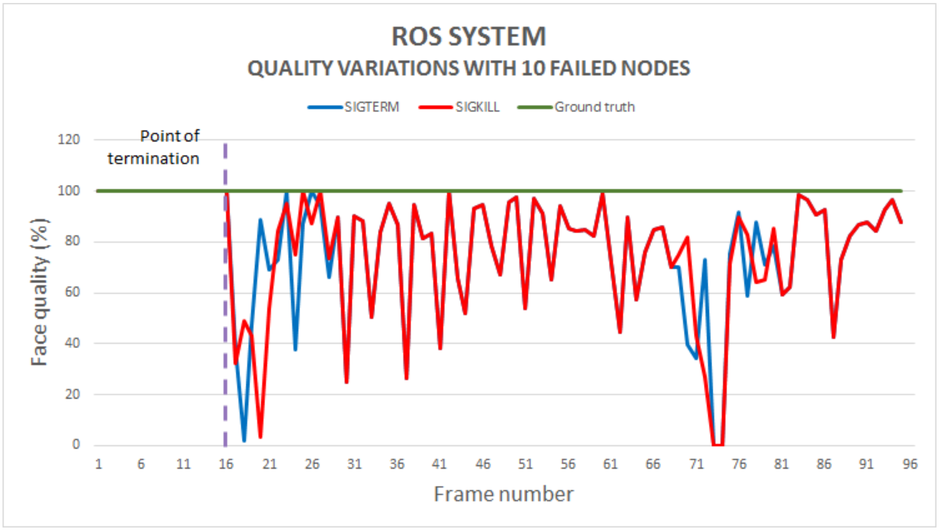

Figure 4.12: 10-node ROS sub-test

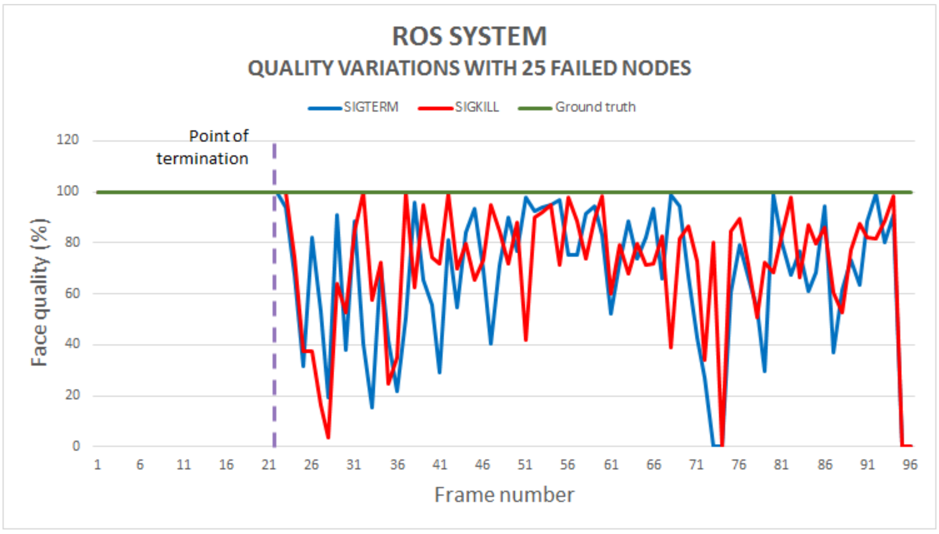

Figure 4.13: 25-node ROS sub-test

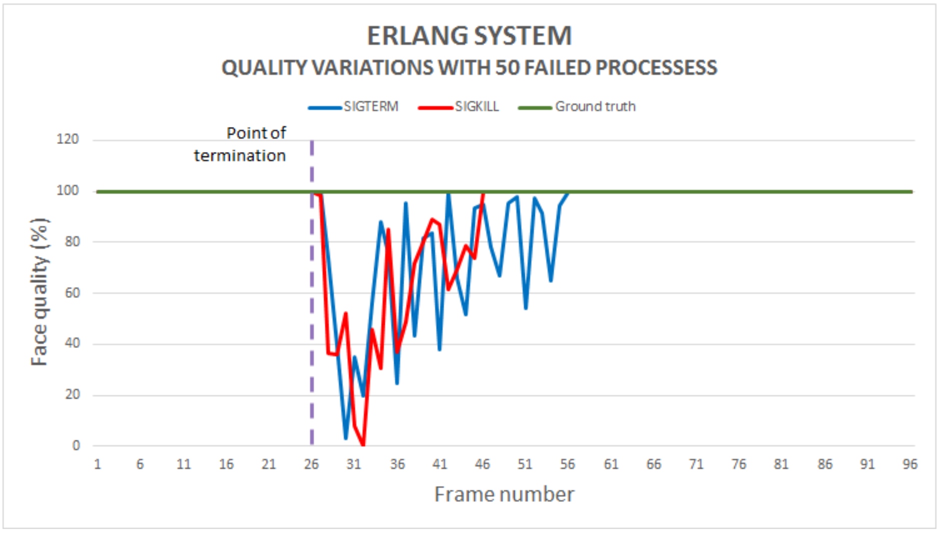

Figure 4.14: 50-node Erlang sub-test

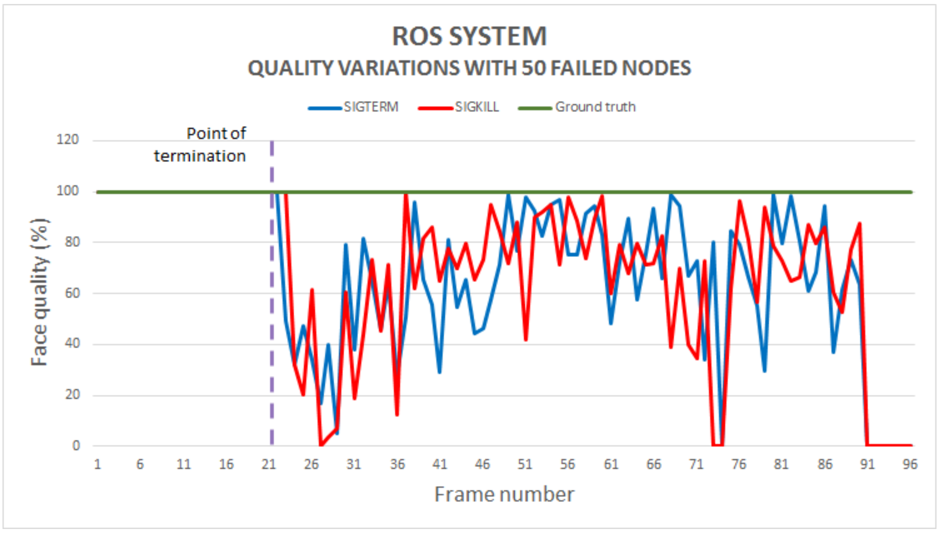

Figure 4.15: 50-node ROS sub-test

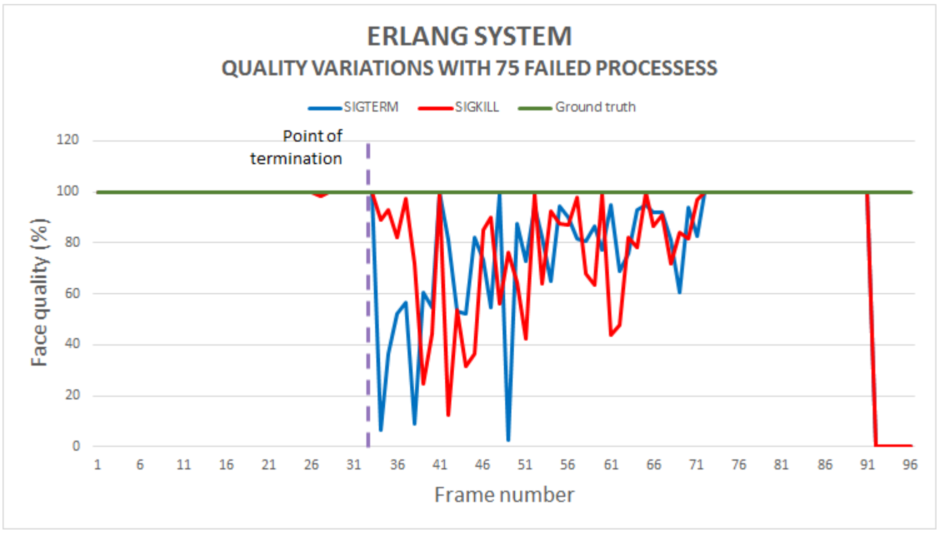

Figure 4.16: 75-node Erlang sub-test

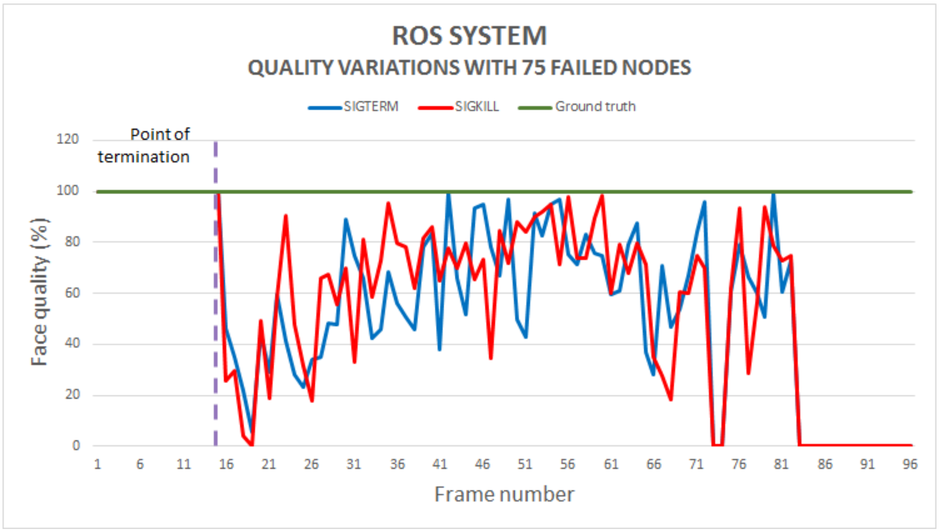

Figure 4.17: 75-node ROS sub-test

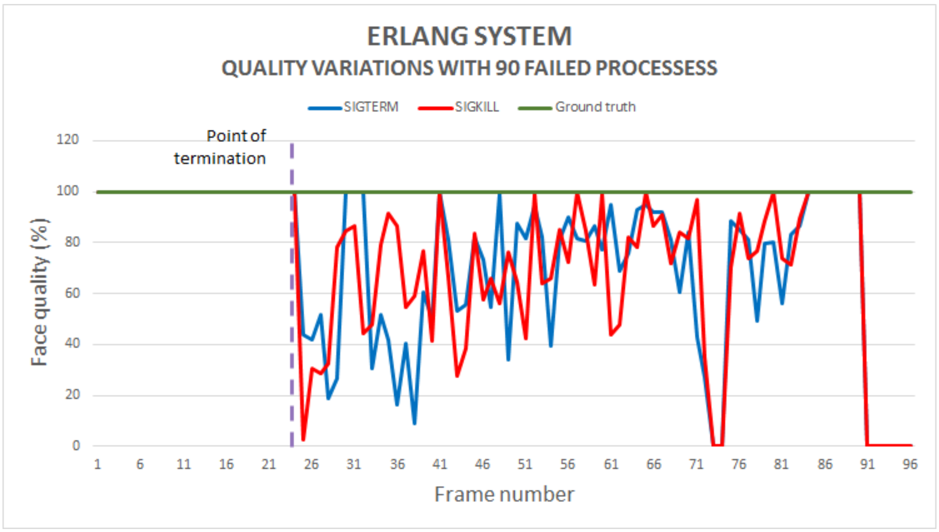

Figure 4.18: 90-node Erlang sub-test

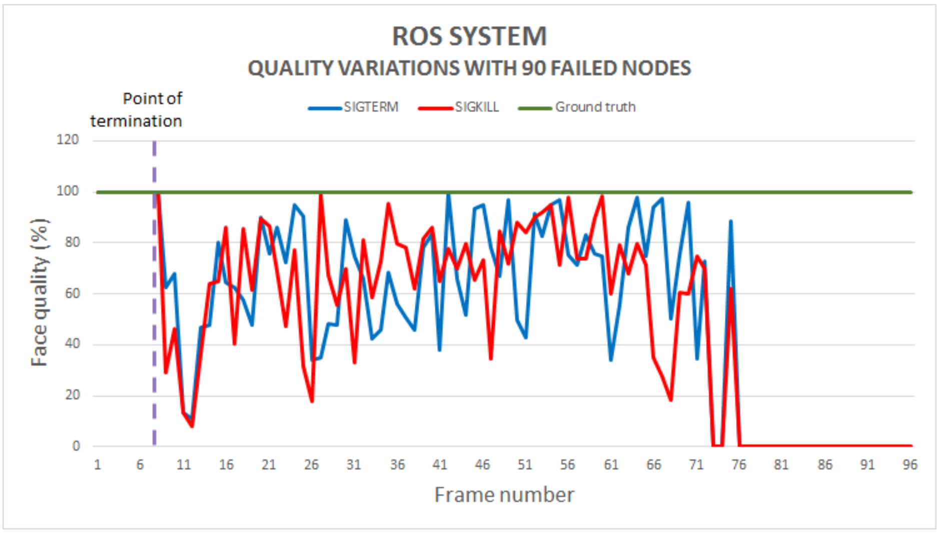

Figure 4.19: 90-node ROS sub-test

In the case of the 10-failures and the 25-failures tests, the Erlang application suffered no face quality penalties, as the worker processes were restarted before the camera had a chance to cast any frames to a dead worker. The graph lines resulting from these observations thus coincided entirely with the 100% ground truth reading and are not shown. The same situation does not, however, hold true for the analogous ROS program, where even for a relatively small number of terminated nodes, the high latency of node restarts cause some frames to be published to topics that are no longer connected to their dedicated worker.

Unlike other log/topic-based message passing frameworks (e.g. Apache Kafka [2], used in the field of Big Data), ROS does not keep track of which messages a topic subscriber has received, meaning that all messages on subscriberless topics are permanently lost. This characteristic reflects on the result graphs as a sudden, steep decrease in face detection accuracy immediately after the terminations occur, that is then followed by ”zig-zags” in the quality percentage. These latter fluctuations were reasoned to be caused primarily by the aggregator receiving an unexpected face reading for a frame $f_i$ and aggregating it with readings associated with ”older” frames, $f_{i-k}, f_{i-k-1}, etc.$, thus skipping over the $k$ lost frames and resulting in a final aggregation product that ”lags” behind the ground truth reading. The lagging phenomenon is more clearly observable in the sample quality graphs included in the appendix, where the establish ground truth value is also shown as a fluctuating series, rather than a 100% line.

As the sequence of frames relayed through the detectors nears its end, the red and blue series sink sharply to a 0% flatline. For each of the n-failures tests, this drop occurs increasingly sooner into the image feed and is considered to be indicative of the number of faces lost as a result of progressively more nodes being shut down. What’s more, throughout the execution of the tests, the ROS system was observed to routinely be unable to restart all of its terminated workers before the last image is transmitted.

Starting with the 50-failures sub-test, the Erlang-based middleware also begins to manifest drops in face quality, following similar patterns to the ones described above for the ROS detectors. In contrast with the ROS situation, however, the impact is small enough such that Erlang worker processes are largely able to recover the few experienced failures before the end of the image sequence. Thus, by virtue of its sub-millisecond restart delays, the the Erlang face tracking application is always able to finish its operation with all 100 processes alive.

The last aspect to be addressed in the analysis of the experiment's findings relates to the sharp quality decline evident between frames 73 and 75 of most n-failures tests. The precipitous decrease is hypothesised to be cause by an anomaly in the OpenCV detection process, whereby a short array of consecutive frames tests negative for the presence of faces, despite a reading for them existing in the ground truth series. As its causes have yet to be unequivocally determined, this matter would be addressed in future continuations of this research.

The conclusions that can be derived from this last reliability evaluation closely mirror the ones drawn in the previous experiment and serve to show that Erlang may possess the capacity to not only reduce robot component downtime, but also mitigate the negative effects of these failures. These findings become particularly compelling when considering the fact that many modern robots operate in safety-critical environments and any periods on unstable behaviour could result in business losses or even hazardous functioning.

# Chapter 5

# Conclusion

This chapter provides a synopsis of the ideas presented within this dissertation, reviewing the research premise, the methodologies used to examine it and the derived conclusions.

## 5.1 Summary

This research set out to investigate the feasibility of a robotic control framework written in Erlang that could achieve fast, reliable and extensible communication between robot components. The project acknowledged that as autonomous systems continue to grow in complexity, so too do their needs for fast concurrent operation and high fault tolerance and used this argument as its primary research motivation.

The Robot Operating System (ROS) is a widely-used robotic middleware that offers well-documented solutions for sensor-to-actuator information flow, but nevertheless manifests a number of scalability and reliability-related shortcomings that limit its performance in large-scale systems. Conversely, the Erlang programming language puts high-volume concurrency and reliability at the core of its paradigm and has thus been demonstrated to be capable of supporting large distributed operations in a business context. Building on this fact and on the correspondence between both ROS' and Erlang's facilities for asynchronous message passing, the project hypothesised that Erlang has the potential to offer a solid and fault tolerant underlying architecture on which to build ROS-like processes and inter-communication mechanisms.

To this end, a prototypical face-tracking application was developed and run using two middleware variants. The first one utilises ROS-powered message relay via topics, while the second employed a specially-built multi-process Erlang framework to transmit messages between functional units residing outside of the Erlang VM. Both applications interface with an external PTZ camera to capture frames in real time and rely on Python-based Haar cascade classification to perform distributed human face detection on these images.

Diverging from previous research on incorporating Erlang into robotics research , the project proceeded to study the non-functional similarities and differences between the two face tracking systems. This was achieved through a series of five experiments that compared and contrasted the scalability limits of the systems, the performance impact of process scaling and the programs' recovery protocols in the face of unforeseen partial failures.

Hence, it was found that Erlang is capable of scaling to 3.5 times more active processes than its ROS-based counterpart, supporting a maximum of 700 lightweight operational agents. This finding suggests that an Erlang robotic middleware could find fruitful applications in large-scale robotic settings such as manufacturing industries or modular systems with a flexible number of components. However, while both face tracking prototypes were discovered to exhibit similar detection accuracy and transmission latencies when kept under 10 worker agents, Erlang exhibited a continuous increase in the total time taken to process a frame as more agents were added . Since the cause of this phenomenon is thought to lie with the external VM linking library used, the implication is that a highly concurrent Erlang middleware would require and equally performant interface to efficiently bridge robotic components.

In terms of reliability, Erlang processes that encounter sudden terminations restart 1000-1500 times faster than ROS nodes and thus the successful completion of their face tracking tasks is only lightly affected. From an operational perspective, this result carries great significance for robots using Erlang as a middleware, since it means that Erlang can offer an impressively high availability of nearly 99.999995%.

## 5.2 Future Work

While the research done as part of this project has uncovered many valuable results and represents the first documented attempt at quantifying the non-functional performance of Erlang in a robotic context, further work would still be required in order to explore its full potential.

Firstly, as a direct improvement and continuation of the experiments carried out by the project, different variants of the two described applications could be developed and tested. For instance, the research may benefit from a re-implementation of the prototypes using ROS' C++ API and Erlang-to-C/C++ natively implemented functions and port drivers. Such a cross language port could theoretically reduce some of the relay latency and — if tested and optimised accordingly — remove the bias introduced in previous experiments by ErlPort. Alternatively, the systems' distributive model could be redesigned to have frames fragmented into $n$ parts and sent to $n$ workers that carry out separable detection operations on their respective fragments, thus probing Erlang's support for concurrent processing.

From a concurrency perspective, it is also important to recognise that the asynchronous messaging model is not a good fit for all robotic systems and that some agents must necessarily communicate synchronously (e.g. sensors on the same robot or on separate machines that collaborate in performing a multi-part task). Given this observation and the fact that both ROS and Erlang offer support for synchronicity, a foray could be made into comparatively evaluating two applications making use of remote procedure calls to wait for data and issue appropriate replies.

Lastly, while the project focused solely on comparing ROS and Erlang as completely separate robotic middleware solutions, it is thought that an integration of the two systems would serve to create a more versatile communication platform. By harmonising both ROS' rich and well-documented library ecosystem — brought about by its popularity in the robotic community — and Erlang's demonstrated aptitude for concurrent communication and reliability, a more stable and better performing middleware could be developed. This proposal would involve Erlang supervisor processes monitoring the execution of children represented by ROS nodes, as shown in Figure 5.1. Thus, children could share some information between each other using ROS topic mechanics and also maintain a connection with other services running within the Erlang VM.

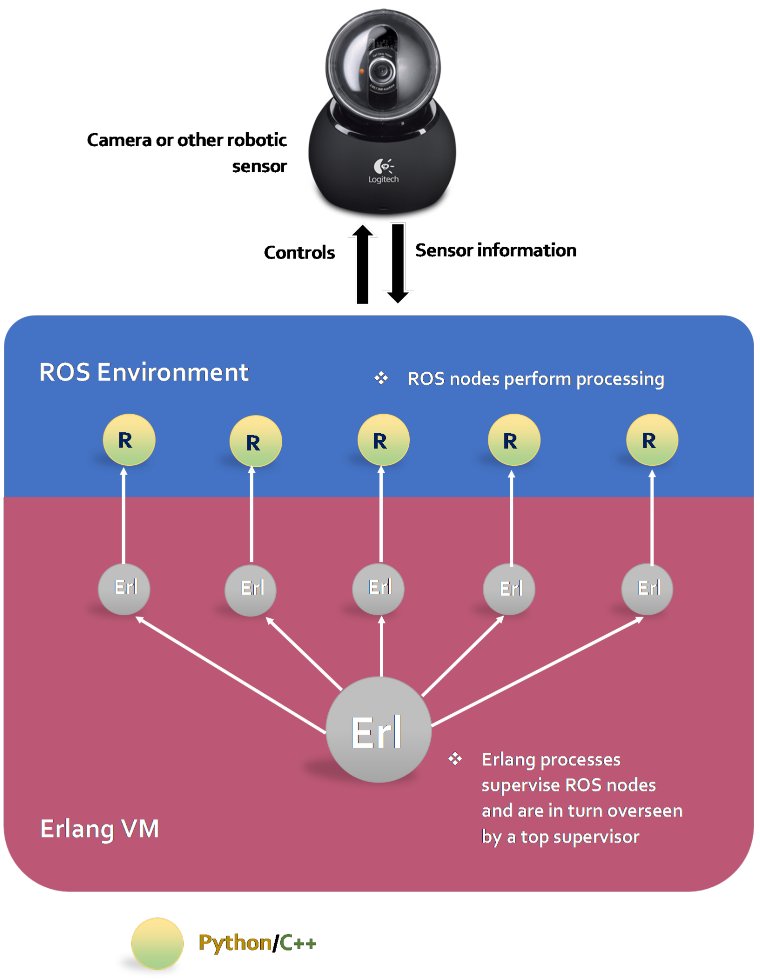
Camera or other robotic sensor
Controls
Sensor information
ROS Environment
ROS nodes perform processing
R
R
R
R
R
Erl
Erl
Erl
Erl
Erl
Erl
Erlang processes supervise ROS nodes and are in turn overseen by a top supervisor
Erlang VM
Python/C++

Figure 5.1: Architecture of potential project extension

# Chapter 6

# Bibliography

[1] Actors Model. http://c2.com/cgi/wiki?ActorsModel. Online; retrieved Tuesday 22nd March, 2016.

[2] Apache Kafka. http://kafka.apache.org/. Online; retrieved Tuesday 22nd March, 2016.

[3] Client Libraries - ROS Wiki. http://wiki.ros.org/Client%20Libraries. Online; retrieved Tuesday 22nd March, 2016.

[4] Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.html. Online; retrieved Tuesday 22nd March, 2016.

[5] Concurrency in C++11. https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/labs/lab6/. Online; retrieved Tuesday 22nd March, 2016.

[6] Concurrency in Erlang & Scala: The Actor Model. https://rocketeer.be/articles/concurrency-in-erlang-scala/. Online; retrieved Tuesday 22nd March, 2016.

[7] Erlang – Data Types. http://erlang.org/doc/reference_manual/data_types.html. Online; retrieved Tuesday 22nd March, 2016.

[8] Erlang – gen_server Behaviour. http://erlang.org/doc/design_principles/gen_server_concepts.html. Online; retrieved Tuesday 22nd March, 2016.

[9] Erlang – Overview. http://erlang.org/doc/design_principles/des_princ.html#id59018. Online; retrieved Tuesday 22nd March, 2016.

[10] Erlang – Supervisor Behaviour. http://www.erlang.org/doc/design_principles/sup_princ.html. Online; retrieved Tuesday 22nd March, 2016.

[11] Erlang Central — Erlang & Robotics Workshop. https://erlangcentral.org/erlang-robotics-workshop/#.Vii88Cv1-6E. Online; retrieved Tuesday 22nd March, 2016.

[12] Erlang Central — Kent Erlang Robotic Library. https://erlangcentral.org/erlang-projects/details/164. Online; retrieved Tuesday 22nd March, 2016.

[13] Erlang Programming Language. http://www.erlang.org/. Online; retrieved Tuesday 22nd March, 2016.

[14] ErlPort - connect Erlang to other languages. http://erlport.org/. Online; retrieved Tuesday 22nd March, 2016.

[15] Errors and Processes - learn you some Erlang for great good! http://learnyousomeerlang.com/errors-and-processes. Online; retrieved Tuesday 22nd March, 2016.

[16] Eurobot. http://www.eurobot.org/. Online; retrieved Tuesday 22nd March, 2016.

[17] Face Detection using Haar Cascades. http://docs.opencv.org/3.1.0/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0. Online; retrieved Tuesday 22nd March, 2016.

[18] GitHub - SimianArmy Wiki. https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey. Online; retrieved Tuesday 22nd March, 2016.

[19] Lesson: Concurrency, Java Documentation. https://docs.oracle.com/javase/tutorial/essential/concurrency/. Online; retrieved Tuesday 22nd March, 2016.

[20] Master - ROS Wiki. http://wiki.ros.org/Master. Online; retrieved Tuesday 22nd March, 2016.

[21] Message Passing. http://c2.com/cgi/wiki?MessagePassing. Online; retrieved Tuesday 22nd March, 2016.

[22] Message Passing Concurrency. http://c2.com/cgi/wiki?MessagePassingConcurrency. Online; retrieved Tuesday 22nd March, 2016.

[23] Messages - ROS Wiki. http://wiki.ros.org/Messages. Online; retrieved Tuesday 22nd March, 2016.

[24] Names - ROS Wiki. http://wiki.ros.org/Names. Online; retrieved Tuesday 22nd March, 2016.

[25] Nodes - ROS Wiki. http://wiki.ros.org/Nodes. Online; retrieved Tuesday 22nd March, 2016.

[26] NumPy — numpy. http://www.numpy.org/. Online; retrieved Tuesday 22nd March, 2016.

[27] OpenCV. http://opencv.org/. Online; retrieved Tuesday 22nd March, 2016.

[28] OpenCV API Reference. http://docs.opencv.org/2.4/modules/refman.html. Online; retrieved Tuesday 22nd March, 2016.

[29] OpenRTM-aist. http://www.openrtm.org/openrtm/en. Online; retrieved Tuesday 22nd March, 2016.

[30] Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. http://norvig.com/paip.html. Online; retrieved Tuesday 22nd March, 2016.

[31] Parameter Server - ROS Wiki. http://wiki.ros.org/Parameter%20Server. Online; retrieved Tuesday 22nd March, 2016.

[32] Patterns of Client-Server Architectures. http://c2.com/cgi/wiki?PatternsOfClientServerArchitectures. Online; retrieved Tuesday 22nd March, 2016.

[33] The Player Project - Player. http://playerstage.sourceforge.net/index.php?src=player. Online; retrieved Tuesday 22nd March, 2016.

[34] The Player Project - Stage. http://playerstage.sourceforge.net/index.php?src=stage. Online; retrieved Tuesday 22nd March, 2016.

[35] Publish Subscribe Model. http://c2.com/cgi/wiki?PublishSubscribeModel. Online; retrieved Tuesday 22nd March, 2016.

[36] Quickcam Orbit AF. http://support.logitech.com/en_us/product/quickcam-sphere-af. Online; retrieved Tuesday 22nd March, 2016.

[37] Robots - 2015 News and Articles - Robotic Technology. http://www.livescience.com/topics/robots/. Online; retrieved Tuesday 22nd March, 2016.

[38] Robots - ROS Wiki. http://wiki.ros.org/Robots. Online; retrieved Tuesday 22nd March, 2016.

[39] ROS Ecosystem. http://www.ros.org/browse/list.php. Online; retrieved Tuesday 22nd March, 2016.

[40] ROS/Concepts - ROS Wiki. http://wiki.ros.org/ROS/Concepts#ROS_Computation_Graph_Level. Online; retrieved Tuesday 22nd March, 2016.

[41] roscpp - ROS Wiki. http://wiki.ros.org/roscpp. Online; retrieved Tuesday 22nd March, 2016.

[42] ROS/Introduction - ROS Wiki. http://wiki.ros.org/ROS/Introduction. Online; retrieved Tuesday 22nd March, 2016.

[43] roslaunch - ROS Wiki. http://wiki.ros.org/roslaunch. Online; retrieved Tuesday 22nd March, 2016.

[44] ROS.org — Core Components. http://www.ros.org/core-components/. Online; retrieved Tuesday 22nd March, 2016.

[45] ROS.org — powering the world's robots. http://www.ros.org/. Online; retrieved Tuesday 22nd March, 2016.

[46] rospy - ROS Wiki. http://wiki.ros.org/rospy. Online; retrieved Tuesday 22nd March, 2016.

[47] ROS/Tutorials/UnderstandingNodes - ROS Wiki. http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes. Online; retrieved Tuesday 22nd March, 2016.

[48] Services - ROS Wiki. http://wiki.ros.org/Services. Online; retrieved Tuesday 22nd March, 2016.

[49] Shared State Concurrency. http://c2.com/cgi/wiki?SharedStateConcurrency. Online; retrieved Tuesday 22nd March, 2016.

[50] Supervision Principles - Erlang. http://www.erlang.org/documentation/doc-4.9.1/doc/design_principles/sup_princ.html. Online; retrieved Tuesday 22nd March, 2016.

[51] Topics - ROS Wiki. http://wiki.ros.org/Topics. Online; retrieved Tuesday 22nd March, 2016.

[52] G. Biggs. https://staff.aist.go.jp/geoffrey.biggs/erlang.html, 2009-2010. Online; retrieved Tuesday 22nd March, 2016.

[53] J. Armstrong. Erlang: Software for a concurrent world. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07. Springer-Verlag, 2007.

[54] J. Armstrong. A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III. ACM, 2007.

[55] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[56] R. Barraquand and A. Negre. The Robotic Operating Ssystem At a Glance. http://barraq.github.io/fOSSa2012/slides.html, 2012. Online; retrieved Tuesday 22nd March, 2016.

[57] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.

[58] S. Cubero. *Industrial Robotics: Theory, Modelling and Control*. Pro-Literatur-Verlag, 2007.

[59] P. de Kok. Why ROS's Timestamps Are Not Enough. http://pkok.github.io/2014/10/16/Why-ROS's-timestamps-are-not-enough/. Online; retrieved Tuesday 22nd March, 2016.

[60] Sten Grüner and Thomas Lorentsen. Teaching erlang using robotics and player/stage. In *Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG*, ERLANG '09. ACM, 2009.

[61] F. Hébert and J. Armstrong. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, 2014.

[62] A. Kika and S. Greca. Multithreading image processing in single-core and multicore cpu using java. *International Journal of advanced computer science and applications*, 2013.

[63] E. Letuchy. Erlang at Facebook. http://www.erlang-factory.com/upload/presentations/31/EugeneLetuchy-ErlangatFacebook.pdf, 2009. Online; retrieved Tuesday 22nd March, 2016.

[64] M. Levandowski and D. Winter. Distance between Sets. *Nature 234*, 1971.

[65] S. Lloyd. Ultimate physical limits to computation. *Nature 406*, 2000.

[66] R. Reed. Scaling to Millions of Simultaneous Connections. http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf, 2012. Online; retrieved Tuesday 22nd March, 2016.

[67] C. Santoro. An erlang framework for autonomous mobile robots. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*, ERLANG '07. ACM, 2007.

[68] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts, 9th Edition*. Wiley Global Education, 2012.

[69] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer London, 2010.

[70] N. Torkington. Amazon SimpleDB is built on Erlang. http://radar.oreilly.com/2007/12/amazon-simpledb-is-built-on-er.html, 2007. Online; retrieved Tuesday 22nd March, 2016.

[71] M. Verraes. Let It Crash. http://verraes.net/2014/12/erlang-let-it-crash/, 2014. Online; retrieved Tuesday 22nd March, 2016.

[72] R. Virding. Hitchhikers Tour of the BEAM. http://www.erlang-factory.com/upload/presentations/708/HitchhikersTouroftheBEAM.pdf, 2012. Online; retrieved Tuesday 22nd March, 2016.

# Chapter 7

# Appendix

## Appendix A: Project Timeline

| | | |
|---|---|---|
| **Preliminaries** | **Week 1** (21st – 27th September 2015) | ❖ Initial meetings<br>❖ Start of Literature Review<br>❖ Erlang tutorials |
| | **Week 2** (28th – 4th October 2015) | ❖ Obtained camera<br>❖ Investigated ROS implementation<br>❖ Erlang tutorials (cont.) |
| | **Week 3** (5th – 11th October 2015) | ❖ Investigated and resolved Linux compatibility with camera<br>❖ Documented Erlang fault tolerance |
| **ROS Development** | **Week 4** (12th – 18th October 2015) | ❖ Developed image processing script for ROS nodes and integrate with camera |
| | **Week 5** (19th – 25th October 2015) | ❖ Finalised initial development of camera to ROS interfaces<br>❖ Documented progress made in dissertation<br>❖ Finalised Literature Review |
| **Erlang Development** | **Week 6** (26th – 1st November 2015) | ❖ Investigated camera control through Erlang |
| | **Week 7** (2nd – 8th November 2015) | ❖ Commenced development of camera control nodes in Erlang |
| | **Week 8** (9th – 15th November 2015) | ❖ Erlang development work |
| | **Week 9** (16th – 22nd November 2015) | |
| | **Week 10** (23rd – 29th November 2015) | |
| | **Week 11** (30th – 6th December 2015) | |

| | | |
|---|---|---|
| **Erlang Development** | **Week 12**<br>(7th – 13th December 2015) | ❖ Commenced Erlang supervision tree |
| | **Week 13**<br>(14th – 20th December 2015) | ❖ Developed Erlang supervision tree |
| | **Week 14**<br>(21st – 27th December 2015) | |
| | **Week 15**<br>(28th – 3rd January 2016) | |
| **ROS vs. Erlang Analysis** | **Week 16**<br>(4th – 10th January 2016) | ❖ Prepared to start analysis of the two programs (ROS/Erlang) |
| | **Week 17**<br>(11th – 17th January 2016) | ❖ Performed scalability and reliability experiments on both platforms<br>❖ Devised programmatic tests<br>❖ Fixed last minute bugs preventing testing |
| | **Week 18**<br>(18th – 24th January 2016) | |
| | **Week 19**<br>(25th – 31st January 2016) | |
| | **Week 20**<br>(1st – 7th February 2016) | |
| | **Week 21**<br>(8th – 14th February 2016) | |
| **Finalisation** | **Week 22**<br>(15th – 21th February 2016) | ❖ Review and testing<br>❖ Documented progress made in dissertation<br>❖ Started considering wider goals of project |
| | **Week 23**<br>(22nd – 28th February 2016) | |
| **Documentation** | **Week 24**<br>(29th – 6th March 2016) | ❖ Dissertation work |
| | **Week 25**<br>(7th – 13th March 2016) | |
| | **Week 26**<br>(14th – 20th March 2016) | |
| | **Week 27**<br>(21st – 27th March 2016) | |

## Appendix B: Code and Experiment Data

The project code and the resources used in the experiments (images, code, data) can be found in the project's GitHub repository.

## Appendix C: Face Quality Variations with Worker Failure Experiment - Extra Figures

The following figures are variations of Figures 4.12 - 4.19 where the y-axis denotes the quality index. The ground truth detection is also shown as a fluctuating line, to better evidence the delays incurred by random worker failures.

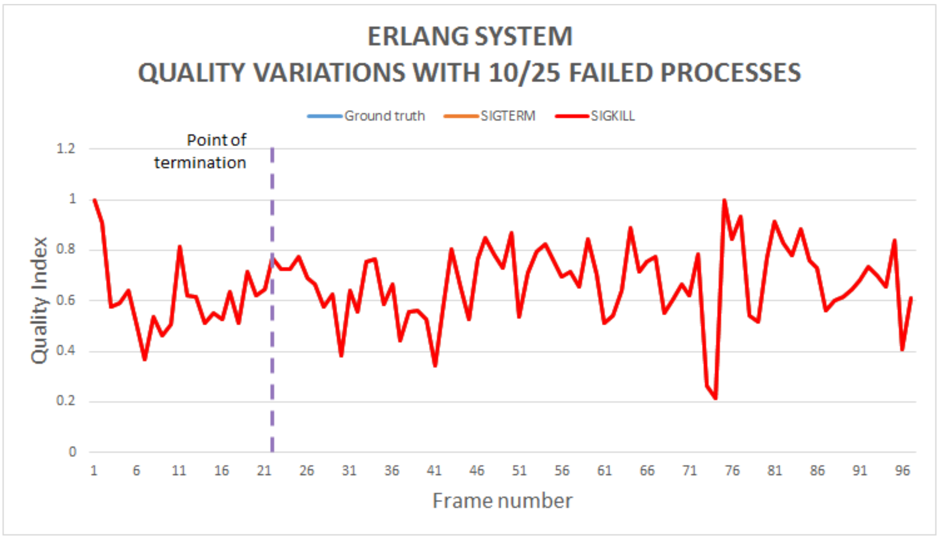

Figure 7.1: 10/25-node Erlang sub-test

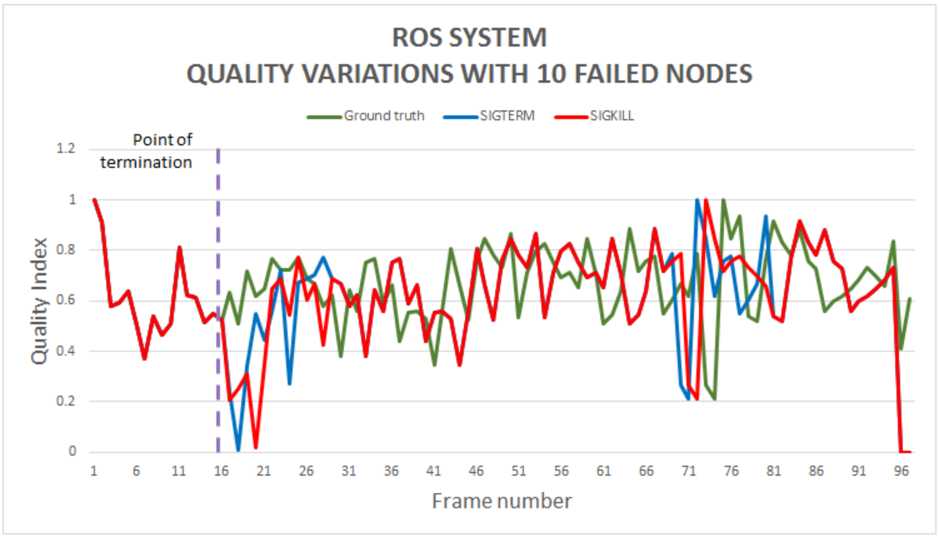

Figure 7.2: 10-node ROS sub-test

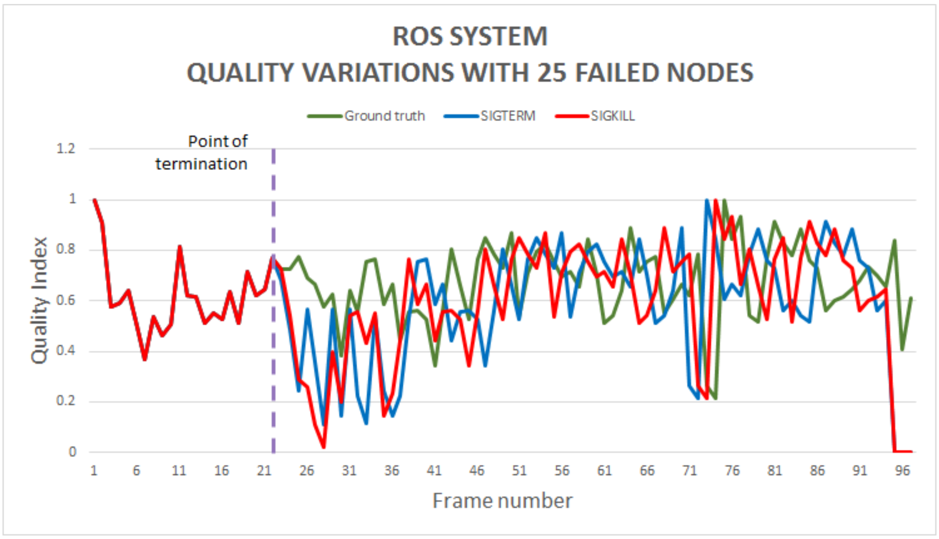

Figure 7.3: 25-node ROS sub-test

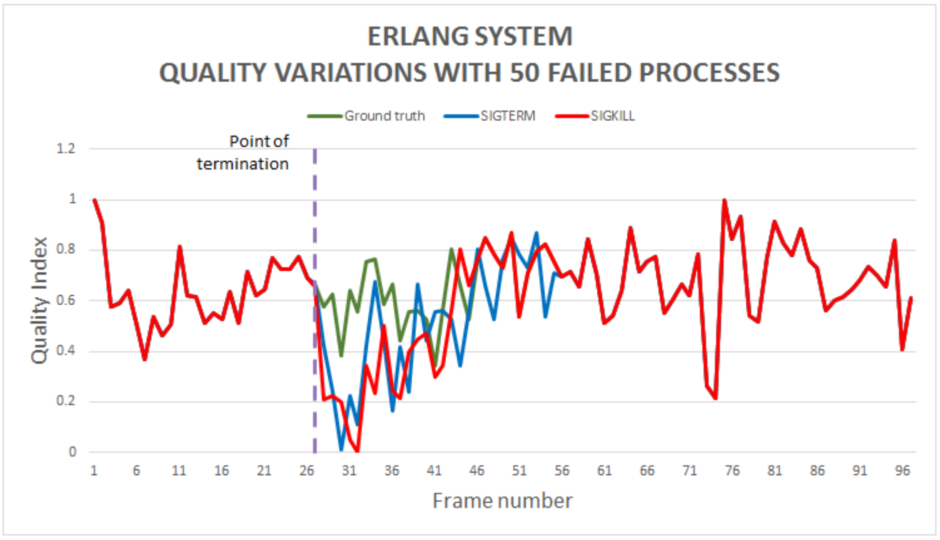

Figure 7.4: 50-node Erlang sub-test

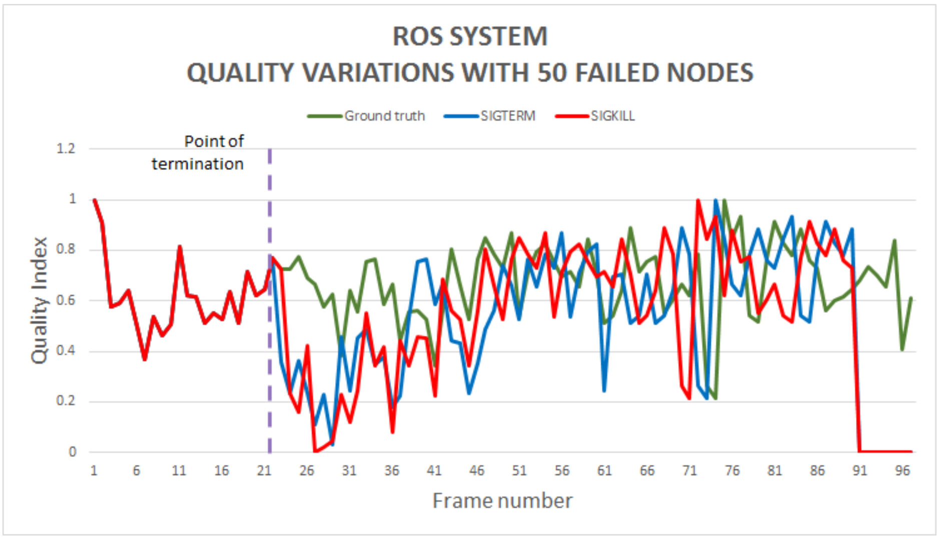

Figure 7.5: 50-node ROS sub-test

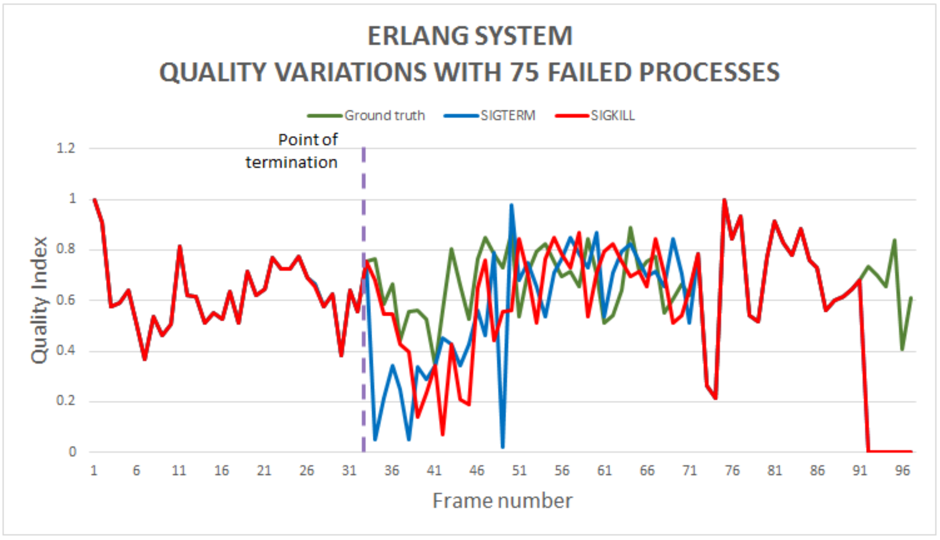

Figure 7.6: 75-node Erlang sub-test

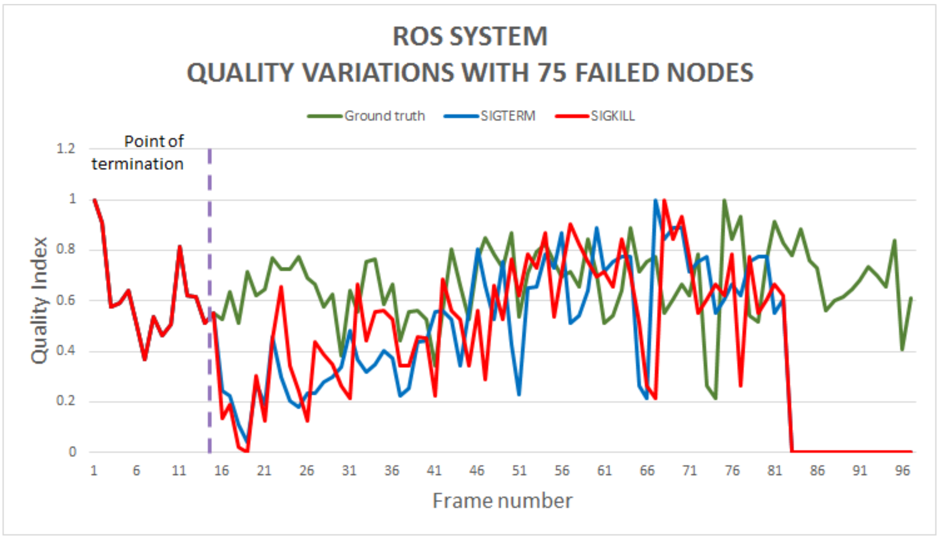

Figure 7.7: 75-node ROS sub-test

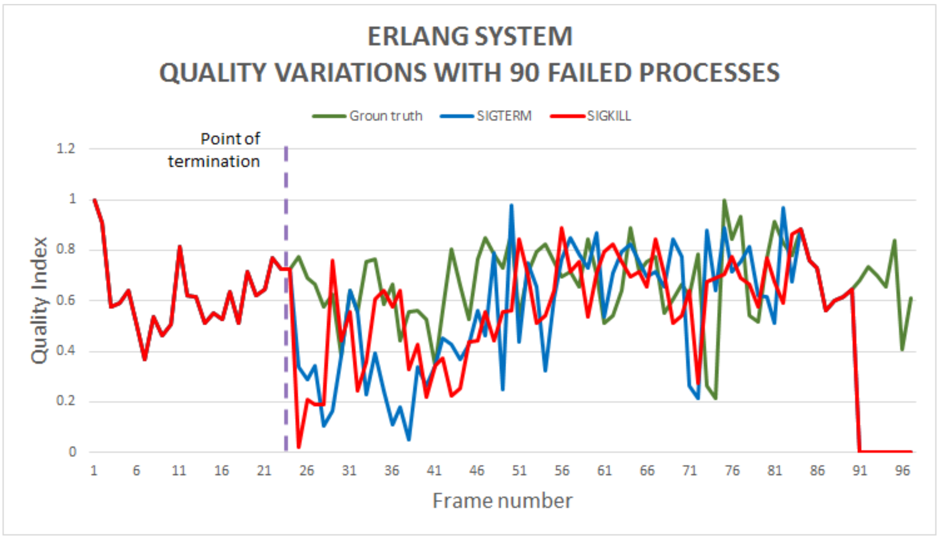

Figure 7.8: 90-node Erlang sub-test

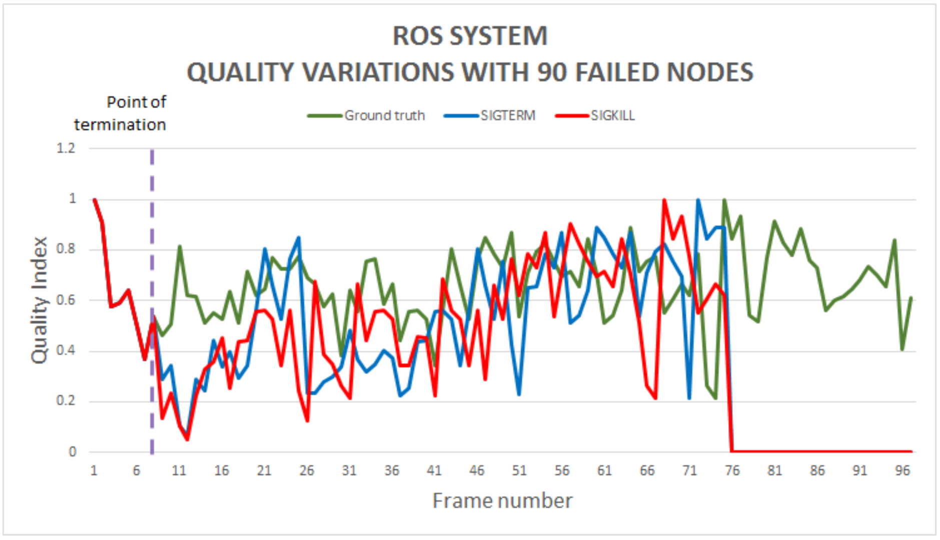

Figure 7.9: 90-node ROS sub-test