Towards Reliable and Scalable Robot Communication

Andreea Lutac, Natalia Chechina, Gerardo Aragon-Camarasa, and Phil Trinder
School of Computing Science, University of Glasgow, United Kingdom
Andreea.Lutac@gmail.com, {Natalia.Chechina, Gerardo.AragonCamarasa, Phil.Trinder}@glasgow.ac.uk

Abstract

The Robot Operating System (ROS) is the *de facto* standard platform for modern robots. However, communication between ROS nodes has scalability and reliability issues in practice. In this paper, we investigate whether Erlang's lightweight concurrency and reliability mechanisms have the potential to address these issues. The basis of the investigation is a pair of simple but typical robotic control applications, namely two face-trackers: one using ROS publish/subscribe messaging, and the other a bespoke Erlang communication framework.

We report experiments that compare five key aspects of the ROS and Erlang face trackers. We find that Erlang communication scales better, supporting at least 3.5 times more active processes (700 processes) than its ROS-based counterpart (200 nodes) while consuming half of the memory. However, while both face tracking prototypes exhibit similar detection accuracy and transmission latencies with 10 or fewer workers, Erlang exhibits a continuous increase in the total time taken to process a frame as more agents are added, and we identify the cause. A reliability study shows that while both ROS and Erlang restart failed computations, the Erlang processes restart 1000–1500 times faster than ROS nodes, reducing robot component downtime and mitigating the impact of the failures.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks

Keywords Robotics, ROS, Erlang, fault tolerance, scalability

1. Introduction

Modern robots are typically highly concurrent, complex systems that 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 reliable and concurrent communication between hardware, software and intelligence components. This interaction is commonly mediated by *the Robot Operating System (ROS)* [23], a state of the art robotic middleware. ROS supports concurrent synchronous and asynchronous processes communicating by message passing, and allows roboticists to compose systems from heterogeneous hardware and software components.

Most of the development work behind ROS comes through contributions from the inter-disciplinary robotics community, with strong academic representation. ROS has grown organically to support a large range of compatible robots. However, ROS exhibits a number of scalability, inter-process synchronization, and reliability limitations in practice. While these limitations are common knowledge among ROS users, they have not been widely explored or reported in the literature.

Erlang [3] offers the right combination of capabilities for robotics, namely real time capabilities, lightweight scalable concurrency, and world-leading reliability. In consequence there have been numerous academic and industrial projects investigating these synergies. These projects mainly use Erlang to control robots or robot components e.g. [4, 25], or teach Erlang using robots [12] (Section 2.2).

The approach presented here is novel in investigating whether Erlang has the potential to address the scalability and reliability issues with communication between ROS nodes. That is, we don't replace the entire Robot Operating System, only the communication layer. We investigate the hypothesis by comparing the scalability and reliability of two analogous middleware systems in a typical robotic control context. The context is real-time face tracking, and the main tasks are to extract a stream of video frames from a camera, perform face recognition with a number of classifier workers, and to redirect the camera to follow the face location. The ROS face tracker communicates using publish/subscribe on ROS topics, while the Erlang face tracker uses a bespoke Erlang framework.

We start with an overview of ROS and review prior work using Erlang in robotics (Section 2). We present the design and implementation of the ROS and Erlang-based face trackers (Section 3). We conduct five experiments that compare different aspects of the ROS and Erlang face trackers. Namely we compare their scalability (maximum number of concurrent workers: ROS nodes or Erlang processes); message latency; quality of real-time face recognition; reliability in the face of ROS node or Erlang process failures induced by a Chaos Monkey [27]; and the impact of failures on face match quality (Section 4).

2. Background

Since antiquity humans have been interested in automation, and there have been numerous attempts to construct autonomous mechanical systems. While the early days of robotics did not yield much in the way of full autonomy, since the second half of the 20th century, robotics has gradually expanded to provide essential tools in a range of domains like manufacturing, health-care, and space exploration.

Robot vision is a particularly intriguing facet, as it is the principal means through which intelligent systems 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. It is evident that human brains have evolved to be very efficient at

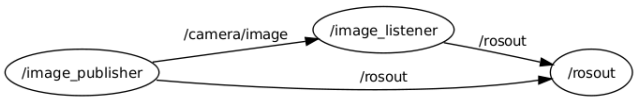


Figure 1. Example of a ROS Node Graph

visual data processing tasks such as feature extraction and object detection. However, these tasks remain challenging for computer systems, and solutions often stress physical limitations in memory, clock speed, data transfer rates, and the energy consumption of the hardware they utilise [16].

2.1 Robot Operating System

Modern robots are complex heterogeneous systems, and in recent years more and more robotic applications have started to make extensive use of the common platform provided by the Robot Operating System (ROS) [23]. Originally developed at Stanford Artificial Intelligence Laboratory, the project offers client libraries mainly in C++ (`roscpp` [24]), Python (`rospy` [5]) and LISP, but also experimentally in other popular languages such as Java, JavaScript, Haskell, and Ruby [21].

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

The main aim of ROS is to provide operating system-like functionalities, for instance:

- 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 extensible.

Of particular interest amongst these is the process management aspect. Indeed, the heavy reliance of ROS on its message-passing communication infrastructure [20] is where most of its perceived benefits and drawbacks lie. Processes are termed "nodes" in ROS, as they form part of an underlying graph (Figure 1) that keeps track of nodes 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 each computation to be delegated to a separate node, thus reducing the complexity of the control systems. At the core of all the programs running on ROS is an anonymized form of the publisher-subscriber pattern [7], i.e. ROS nodes communicate anonymously due to the fact that connections are not formed between named pairs of nodes, but rather channels themselves are named, and then any topic can publish or listen to the channels.

At the start of its execution, a node is expected to first register with the *master* ROS service, `roscore`. The master node is the core of a ROS system, responsible for providing naming and registration to nodes connected to it. Its primary functionality is to act as a 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 therefore able to communicate through unidirectional topics, request-reply RPC services, and a globally-viewable parameter server for static variables.

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, such as sensor data and actuator positions. However, as the number of nodes grows scalability issues arise. Given the fact that ROS adopts a graph model to store and manage its network of nodes, either a substantial increase in the number of connections or the forming of a complete graph (in essence an all-to-all connection) is likely to affect its performance. In addition, ROS' inter-process synchronisation mechanisms are based on time stamps, which can induce failure in certain processes if the time stamps are not received in order [6].

Furthermore, ROS provides limited mechanisms to detect failures and restart nodes. For example, ROS fault-tolerance consists of restarting a process only when it terminates abruptly, without considering network disconnections or runtime errors. The failed ROS node poisons the system, e.g. all other ROS nodes that expect data from the failed node can potentially stop working. The typical solution is manual failure detection and reboot. All this makes ROS not suitable for autonomous long-term operations. Industrial and academic roboticists are aware of, and concerned about, the lack of reliable and scalable communications in ROS, but these deficiencies have remained largely undocumented in the literature.

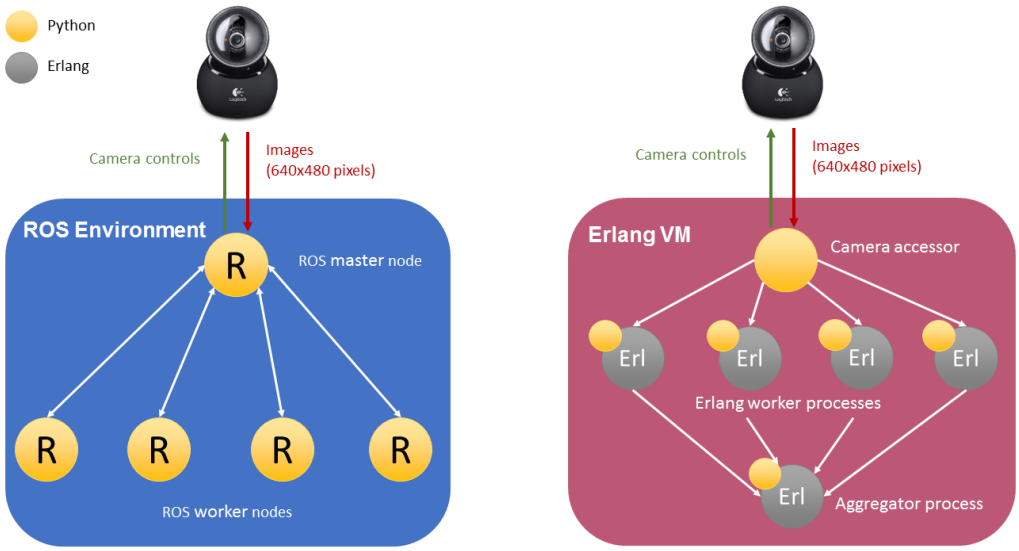
2.2 Erlang in Robotics

Erlang [3] clearly provides the right combination of capabilities for robot control, namely real time capabilities, lightweight scalable concurrency, and world-leading reliability mechanisms. Hence, there are a number of academic and industrial projects seeking to exploit its capabilities.

One of the first projects was conducted at the University of Catania in 2005 [25]. The goal of this work was to produce a complete robotic framework in Erlang for a set of autonomous robots participating in the Eurobot competition [18]. The project greatly emphasised system modularity, which meant that the framework was constructed using a layered architecture to separate each process concern, starting with the low-level control layers that govern physical movements and interface with robot hardware, to the higher-level interpretation layers, such as planning and reasoning functions, and control logic. Erlang's concurrent features were also utilised both in the AI aspects of the project and in the task of efficiently parallelizing the robot's control loops. While, traditionally, these modules would have been implemented in languages like LISP [19] and C/C++, the team asserts that Erlang can successfully merge both language's capabilities. The team later created ROSEN [26], a robotic framework and simulation engine developed in Erlang similarly designed to separate generic functionalities from the idiosyncrasies of specific robotic systems.

The RT-Erlang project was conducted in 2009–2010 at the National Institute of Advanced Industrial Science and Technology (AIST) in Japan [4]. The initial goal of this project was to develop an Erlang tool for the monitoring and orchestration of a network of OpenRTM-aist [2] components. Much like the ROS framework, 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.

Both Erlang Framework and RT-Erlang projects emphasised the importance of ensuring proper functioning of separate, yet dependent software components. In case of the former project, 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 in the RT-Erlang project, these came in the form of intercommunicating RT-components.



(a) ROS (b) Erlang

Figure 2. Architectures of the ROS and Erlang Face Tracking Programs

There have been industrial experiments using the Erlang family of languages for robot control, e.g. a US based software developing company Isotope11 developed drone demonstrators [1] in 2014 using Elixir [14]. This was a low cost side project that demonstrated how to use Erlang and Elixir on Android mobile platforms.

Where the foregoing projects mainly seek to implement robot control predominantly in Erlang, our approach is novel in preserving ROS as a common platform, and replacing only the communication layer between ROS nodes.

Erlang has also been taught through robotics, providing a robotic platform. The project was a collaboration between RWTH Aachen and the University of Kent [12]. 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 [11], the Kent Erlang Robotics Library. Here Erlang user level and middleware modules communicate with a robotic driver implemented in C++, which in turn makes use of Player [9], a cross-language robot device interface for a variety of control and sensor hardware, and Stage [29], a 2D simulation environment simulator.

The combination 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. In contrast we neither aim to build a robotic platform, nor use the technology as an Erlang teaching tool.

3. Design and Methodology

3.1 Common Design

The ROS and Erlang face tracking systems follow a similar client-server model, and for simplicity we describe a single architecture. The system comprises an image capturing device and pan-tilt actuators. The image capturing device (a pan-tilt enabled Logitech QuickCam® camera[1]) performs face detection on frames captured in real time and then relays the results to camera pan-tilt actuators to achieve face tracking. The decision of using a pan-tilt camera as an archetypal robotic component is based on two major factors. The first of these relates to the role that vision plays in robotics, which makes imaging sensors a relatively ubiquitous piece of hardware for autonomous robots. The second factor is due to the straightforward way in which image manipulation programs can be distributed and parallelized, thus facilitating the completion of scalability and reliability experiments.

Software modules common to both ROS and Erlang face tracking programs comprise the following [17].

- Off-the-shelf OpenCV face detection and localisation detectors. Here, we use in-built OpenCV Haar feature-based cascade classifiers [22].
- Software driver to control the Logitech QuickCam 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.

Figure 2 provides an overview of components and interactions of the face tracking program implemented in ROS and Erlang. To facilitate a common software architecture in both versions, we use Python (highlighted in yellow) that performs the following: image acquisition, face detection, frame display (to informally validate tracking results), and camera operation. By having a common and homogeneous software base, it allows us to focus on the communication capabilities of ROS and Erlang while keeping remaining functionality unchanged.

The key aspect examined in this design was distribution of the image frames to the worker nodes/processes in both implementations. We considered two alternatives:

1. *random* distribution of frames between workers (effectively ensuring frames are never processed more than once);
2. *broadcasting* every frame to all worker nodes and synchronise their detection results at the aggregation stage.

[1] `http://support.logitech.com/en_us/product/quickcam-sphere-af`

While the broadcast method incurs negligible detection accuracy penalties, and hence is a better fit for a system which performs a mean-based face detection, it also presents some downsides. That is, 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 the broadcast method, we employ random frame cast approach. Broadcast-based distribution is however used briefly in the face quality experiments (Section 4.3), where it enables to expose a flaw in ROS' message synchronisation protocol.

3.2 ROS interface

The ROS face tracking program has two types of ROS nodes (Figure 2(a)): main and worker. The main node acts as an interface to the camera and is responsible for acquiring images from the camera and distributing them to the worker nodes. The communication between the main node and a set of worker nodes is asynchronous. The worker nodes wait for frame inputs and subsequently detect faces on the given image frame. Detected faces from each worker node are sent back to the main node, where the face coordinates (represented as a rectangular box) are averaged to obtain a single face detection. The centre of the averaged rectangular box is then used to compute the physical adjustment required for the camera gaze to centre to the new face position.

The core of the ROS face tracking application is confined to a single Python script. This 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 (Listing 1). That is, the main node creates n topics, `camera_frames_i`, where $i \in [0; n)$ – 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 are 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 a function responsible for aggregating the faces and issuing camera instructions.

Listing 1. Initialization of ROS Main Node

```
 1  rospy.init_node('face_tracking_main')
 2  cap = cv.VideoCapture(device_ID)
 3  # Create frame topics
 4  publishers = []
 5  for i in xrange(expected_workers):
 6     topic = 'camera_frames_'+str(i)
 7     publishers += [rospy.Publisher(topic, Image,
          queue_size=1000)]
 8
 9  # Subscribe to face topics
10  subscribers = []
11  for i in xrange(len(publishers)):
12     topic = 'camera_frames_'+str(i)
13     subscribers += [rospy.Subscriber(topic,
          Int32Numpy, aggregate_faces, callback_args=[cap
          ])]
```

Notably, the face topics need to exist 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/received. Regarding the two image message types described above: `Image` and `Int32Numpy`. The `Image` is an in-built ROS message type that enables an efficient encoding of pixel matrices for publishing to a ROS topic and is, therefore, highly suited for wrapping frames retrieved from the camera. The `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 [10].

Having defined the above message passing protocols, the main node proceeds to open a continuous camera feed using OpenCV and randomly selects a frame topic 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` (where $i \in [0; n)$), 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 from 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 rectangles is composed of the following four Numpy integer elements, which aim to characterise the location of a face with respect to the image plane:

- `x_coord` is the top left x-coordinate point of a rectangle that the detector assumes circumscribes a face;
- `y_coord` is the top left y-coordinate point of the same rectangle;
- `width` is the width of the rectangle;
- `height` is 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, i.e. the largest rectangle, which is due to the nature of the cascade classification process that considers it to be the likeliest face candidate. The resulting face data is then sent back to the main node.

The function designated to receive faces at the main node is the aggregator, whose role is to manage the large influx of face readings and determine what movements the camera should execute. For this, we employ a finite-size dequeue (double-ended queue) data structure of variable capacity [13]. Declared as a global parameter, the dequeue 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 dequeue'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 dequeue 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 main and worker nodes of the application, as well as the `roscore` node with which all active nodes are registered. 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, and whether their alive status should be monitored. This last aspect is particularly significant when it comes to protecting the system from unexpected failures (Section 4.4).

3.3 Erlang interface

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 (Figure 2(b)). This added complexity is due to us using Erlang only for the implementation of

Table 1. ROS and Erlang Code Comparison

	ROS interface			Erlang interface		
	Processing	*Communication*	*Reliability*	*Processing*	*Communication*	*Reliability*
Modules	2	1*	1 (ROS 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 custom image message (Int32Numpy, Listing 1) was added to support `float32` images
** One Python module for processing and one Erlang interface module to start the application

the message-passing middleware components. In ROS, however, message-passing is provided though libraries as discussed in previous sections.

Mirroring the implementation of the ROS face tracking program (Section 3.2), the Erlang face tracking program is also structured as two primary modules: one corresponds to each instance of Python worker processes and the other encompasses the aggregation operation. However, these modules are exclusively tasked to perform message-passing. Diverging from ROS' free-form code design, the two Erlang modules follow standard Erlang `gen_server` behaviour that models the server side of the program's client-server communications. Accordingly, clients are represented by Python functions that deal with image acquisition and processing, as well as camera interaction. Note that the functional equivalence of the ROS and Erlang applications considerably facilitates code reuse between the two face tracking systems, meaning that the Python modules developed for use in ROS space requires minor re-structuring to be ported as Erlang clients. Since these implementations are already covered in the previous section, we do not go into further detail on the Python functions, instead we proceed to the Erlang modules.

The `face_server` module listens for incoming image frame messages from the Python camera driver 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 is used to resemble parallel ROS' topic-based interaction (Listing 2). For the Python interaction to take place, at the initialization the face server creates an instance of a Python interpreter via the ErlPort library [28]. A reference to this instance is then passed to each of the server's functions as a "state" parameter.

Listing 2. `gen_server` Functions for the `face_server`

```
1  % Asynchronous handler for incoming frames;
2  % frames are pattern matched to ensure correctness
3  handle_cast({frame, Frame}, PyInstance) ->
4      % Call the Python detector, supplying the frame
         and the Erlang Process ID
5      python:call(PyInstance,facetracking,detect_face,[
         Frame,self()]),
6      {noreply, PyInstance};
7
8  % Synchronous message handler;
9  % set to issue no replies
10 handle_call(_Message, _From, PyInstance) -> {noreply
       , PyInstance}.
```

Whenever a message is sent (or cast) to the face server, the server matches the message against the `{frame, Frame}` pattern. The `handle_cast` performs a call to the Python detector function via the Python interpreter, where the call process can pass to Python functions parameters of any type. 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. A similar multi-threaded behaviour is observed in ROS nodes.

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 (shared amongst the server's functions) and designed to store the faces before aggregation. While Erlang does not offer a built-in dequeue data structure in the same way Python does, its functioning is simulated through additional checks that probe the queue on fullness every time a new element is added, and remove the oldest element if the queue is full. That is, once a face message is cast to the aggregator, the server pushes it onto the queue and computes the same mean-based grouping operation as in the ROS framework. 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.

The reliability in the Erlang implementation is supported by a supervision tree (Figure 3). The tree consists of two main supervisors that monitor the execution of the face server processes and the aggregator server process, respectively, and one top-level supervisor. The top-level supervisor oversees the functioning of its direct children. All of the tree's vertices are configured as `permanent` children, implying they must always be restarted upon failure. 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 child processes are restarted.

3.4 ROS vs. Erlang Implementation Comparison

Through an empirical testing and subsequent extensive refinements we have determined that the two face tracking programs – ROS and Erlang – are functionally equivalent. In Table 1 we summarise the implementation details. It shows that the Erlang implementation requires more code facilities than its ROS analogue. While this may

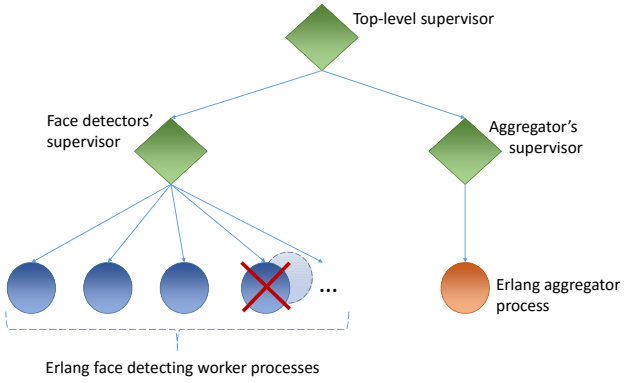


Figure 3. Supervision in the Erlang Face Tracking Application

Table 2. Experiment Parameters

Variable Name	Variable Description	Value
Variables common to both ROS and Erlang		
Device ID	The name under which the operating system identifies the Logitech QuickCam® camera port. The port labelling is the choice of the OS and thus this variable must be manually passed to the programs.	0 or 1
Cascade Classifier ID	The name of the Haar features-based cascade classifier used during the 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 where 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 Dequeue 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 (Section 4.3), since having more faces to aggregate causes the average face value to "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	The number of messages 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 (child processes that do not respond within 20 ms to an exit summons are automatically terminated), `type` = worker/supervisor
Supervisor Restart Strategy	Identifies child processes that are restarted in case of a failure. 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	The number of restarts allowed within a specific period of time before the entire program is deemed faulty and shuts down.	1000 restarts in 1 second

appear disadvantageous, the discrepancy is due to the fact that Erlang is only a development tool, rather than a fully-developed ecosystem such as ROS. Hence, communication and reliability facilities such as topic interaction and the `roslaunch` package[2] that are incorporated into ROS, had to be implemented in the corresponding Erlang program.

4. Experiments

To investigate Erlang potential to address the scalability and reliability issues with communication between ROS nodes we conduct the following experiments that compare the ROS and Erlang face trackers.

- Scalability measured as the maximum number of concurrent workers: ROS nodes or Erlang processes, and associated memory consumption (Section 4.1).
- Message latency (Section 4.2).
- Variations of face quality with a real-time aggregation (Section 4.3).
- Reliability in the face of ROS node or Erlang process failures induced by a Chaos Monkey [27] (Section 4.4).
- Variations of face quality with worker failures (Section 4.5).

The experiments are conducted on an Intel i7-4700HQ processor (4 cores, 2 threads per core), at 2.4 GHz, with 16GB DDR3 in RAM, running 64-bit Ubuntu 15.04, ROS Indigo, Erlang/OTP 18.0, and OpenCV 3.0. Table 2 summarises the parameters used in both programs. These help to eliminate bias in favour or against one of the programs and ensure results derived from each test are valid and meaningful. Below, we provide further justification for the assignment of particular values.

- `scaleFactor` defines percentage of up-scaling between two consecutive levels of the image scale pyramid. Here, `1.1` corresponds to a 10% difference in scale, which, for the purpose of the experiment, is an appropriate trade-off between performance (speed of classification) and thoroughness of detection.
- `minNeighbors` affects the quality of face detection. The higher the value, the better the quality (or accuracy) of the detected face, but the less results (faces) are obtained from an image. We set it to 6 as we only expect the dataset to have one face per image but

[2] `http://wiki.ros.org/roslaunch`

this face must be detected accurately to best assess the impact of failures on face quality.

- `minSize`. To keep the quality of detected faces high, they must be at least `50x50` pixels. Smaller faces have less sharp features (especially with the camera's `640x480` resolution). Thus smaller faces are harder to detect.
- *Publishing rate* of `5Hz` is slower than the rate commonly used (`10 Hz`) in ROS tutorials and example programs as we want to ensure higher data completeness, i.e. all frames arriving at their destination node.
- *Queue size* is kept small (`5`), since no message drops or any other performance impacts were observed. A smaller queue would presumably also use less memory.
- *Supervisor shutdown interval* of `20ms` allows for any non-responsive process to be considered as hanged and that needs a restarting protocol. We set this value to maintain real-time constrains.
- *Supervisor restart strategy* `one_for_one` mirrors ROS restarting behaviour, i.e. ROS only restarts the failed processes.
- *Supervisor maximum restart intensity*. The default values for the `max_restart` and `max_interval` values are `1` and `5`, respectively; which means a maximum of `1` restart within a span of `5` seconds. This proved unsuitable for the purpose of the reliability experiments, where the Chaos Monkey program would kill processes much faster than that. Thus, through experimentation, a `1000-in-1-second` value was chosen to be compatible with the Chaos Monkey failure rate.

Both ROS and Erlang face trackers are open source and can be downloaded from `https://github.com/AndreeaLutac/Erlang-Camera-Control-Project`.

4.1 Scalability

This experiment determines the scalability of the two systems by measuring the maximum number of face detection workers, i.e. the maximum number of ROS nodes or Erlang processes that the corresponding programs can support while functioning correctly, without a severe strain on the operating system and with no failures. The experiment proceeds by increasing the number of workers (ROS nodes or Erlang processes) until the programs start failing.

Figure 4 plots processing time (i.e. period between obtaining a frame from the camera and delivering the result to the corresponding face output) against the number of workers. This is weak scaling, i.e. more workers means that there is more work to be done. The figure shows that the ROS program scales up to 200 worker nodes, while

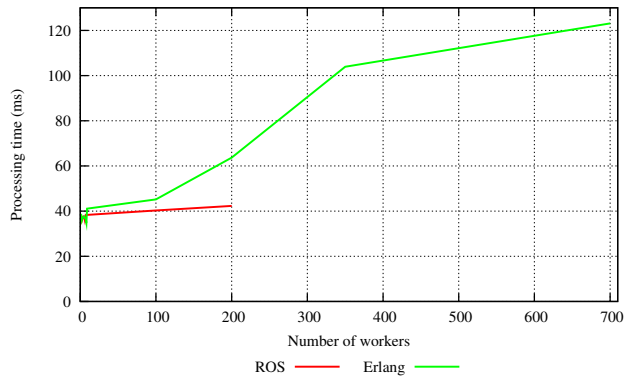


Figure 4. Scalability: Maximum Number of Workers

Table 3. Scalability: 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*

the Erlang program scales up to 700 worker processes. In the case of the ROS system, these failures are due to unspecified Xorg (a display server application for Ubuntu distribution) errors, while the primary cause of the failures in the Erlang application is reaching the OS' maximum open socket limit. The attempts to increase this limit have not resolved the issue, so we attribute the error to poor socket recycling on the part of the ErlPort library (we further discuss this in Section 5).

Memory may also limit the scalability of systems. To analyse this, we compare memory consumption of ROS and Erlang face recognition systems with 50 workers using measurements taken from Ubuntu's inbuilt task manager and the `top` command. Table 3 shows that both the main and the worker ROS nodes are relatively resource-intensive, consuming 65MB and 50MB per node respectively. 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 memory distribution, it can be extrapolated that the Erlang middleware consumes approximately 200MB per 50 processes = 4MB per process. If combined with the 22MB consumed by each Python instance, the total consumption of an Erlang processing unit amounts to *26MB*, approximately half of the corresponding ROS value.

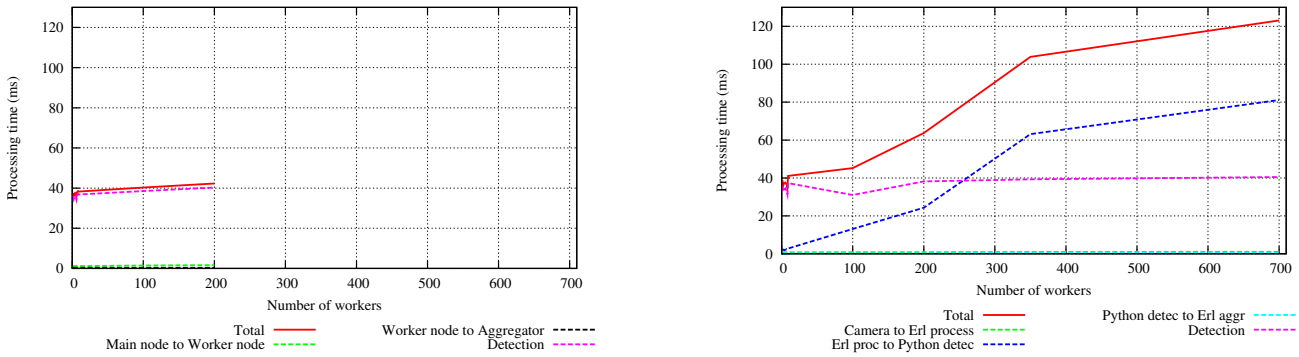
From this experiment we conclude that *Erlang communication provides at least 3.5 times larger scalability (200 ROS nodes against 700 Erlang processes) while consuming half of the memory as compared to a similar ROS implementation*.

4.2 Message Latency

The aim 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. 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 accessors 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 mean time taken for the OpenCV detection to execute is measured for each program. We then sequentially start instances of the ROS and Erlang programs running increasingly more workers and processes in each run. For each program, we obtain a message latency measure by taking the differences between the recorded time stamps, and then take the mean of the 700 latency measures.

Figure 5 shows the calculated latencies for ROS and Erlang programs. On a small scale of up to 9 nodes the total message latency is identical in both ROS and Erlang systems, and is *∼37 milliseconds* (red line in Figure 5). However, starting at 9 workers the time increases sharply for the Erlang system, while the ROS system demonstrates constant time until it reaches its maximum limit of 200 workers.

The delay induced by the OpenCV face detection operation (magenta line in Figure 5) is very similar in both ROS and Erlang



(a) ROS (b) Erlang

Figure 5. Face Tracking Processing Time

programs, thus it does not account for any performance differences. The detection functions perform similarly, with 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.

Further analysis indicates that the spike correlates to the function call from Erlang detector processes to the Python module (blue line in Figure 5(b)). While a possible explanation for this anomaly could relate to the size of the messages (large pixel arrays), the same behaviour is not observed in the camera to Erlang process relay, which deals with similar message sizes, and is also performed via I/O socket. Therefore, we attribute the discrepancy to faults in the behaviour of ErlPort's socket interfaces.

While ErlPort is a convenient method of linking Python and Erlang, its limited scalability significantly limits scalability of the Erlang middleware. From this we conclude that *Erlang is able to offer processing times which are at least as good as the ROS analogue, but great care should be taken in selecting the cross-language or cross-platform communication libraries*.

4.3 Face Quality Variations with Real-time Aggregation

In this experiment we assess the quality of face detection produced by multi-agent instances of the ROS and Erlang systems, investigating an impact of scaling the number of workers on the performance of the two programs.

For that we create a set of 100 (image, file) pairs, where the images are snapshots taken by the camera, 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. In addition, we 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. For each run, we record 100 face position results and, for each test image, we compare the accuracy of the face position against a manually annotated ground truth. The latter is performed by a metric derived from the Jaccard index [15], that computes the similarity of two sample sets. This variant metric, J_Q, replaces the sets with rectangle areas:

$$J_Q(A_{det}, A_{act}) = \frac{A_{ov}}{A_{det} + A_{act} - A_{ov}},\ J_Q \in [0,1], \quad (1)$$

where A_{det} is the area of the rectangle that describes the predicted face position, A_{act} is the area of rectangle that denotes the definitively established face location, and A_{ov} is the area of their overlap. We then classify each index by its corresponding frame, program version (ROS or Erlang), and worker node count. To establish the differences between the two implementations in the context of face quality, we average the indices and compare the resulting mean data. To distribute frames to detectors we use a broadcast approach discussed in Section 3.

From Table 4, we conclude that there is no significant difference between the quality indices resulting from ROS and Erlang programs. However, common to the systems *aggregator dequeue length* variable was observed to have a significant impact on face quality when faces are randomly distributed amongst detectors. This is due to the fact that the larger the dequeue size, the more out-dated faces get aggregated in the final face reading. The results also illustrate that if all output faces are considered individually (at dequeue 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 dequeue size.

Therefore, we conclude that *Erlang's ability as a middleware to support application functions is not hindered by growth in the number of worker processes*.

4.4 Reliability

We investigate the reliability of the ROS and Erlang face trackers by terminating face-detecting workers and aggregator, and then measure the latency between their shutdown and restart. Termination is done both by invoking the agents' exit procedures, through a standard termination signal, and also abruptly, by immediately ordering the kernel to terminate the agents.

To terminate "living" agents from their respective face tracking systems (ROS and Erlang) we have developed a pair of scripts in Python and Erlang. These scripts are modelled after the Chaos Monkey [27] reliability testing service introduced by Netflix. On both ROS and Erlang systems running 100 workers, we instruct the chaos monkey program to execute 200 random terminations every 1 second. We then compute latencies for each system based

Table 4. Mean Indices of Face Quality

	Dequeue Length		
	10	**5**	**1**
ROS	0.285	0.591	0.947
Erlang	0.302	0.608	0.922
Difference	-5.62%	-2.79%	2.71%

Table 5. Reliability: ROS vs. Erlang Recovery Time

Process type	Type of shutdown	
	SIGTERM	SIGKILL
ROS		
Main node	420ms	400ms
Worker node	369ms	291ms
Erlang		
Supervisor process	0.368ms	0.352ms
Worker and aggregator process	0.224ms	0.249ms

on differences between the recorded time stamps, and then take the average over 500 latency readings. We finally compare the resulting mean data to establish which of the two systems is faster to restart its failed agents.

The restart latency presented in Table 5 shows that ROS takes between 291ms and 420ms to restart a failed node, whereas Erlang restarts its processes in 0.224ms–0.368ms, which is approximately *0.06% to 0.08%* of the time taken by ROS. These findings demonstrate that *Erlang is significantly more efficient than ROS at restarting processes that fail randomly and unexpectedly (0.352ms and 0.249ms against 400ms and 291ms respectively)*. The significance of this analysis in the context of robotics is that system uptime for robots running an Erlang supervision mechanism has the potential to be significantly higher compared to ROS-based robots.

4.5 Face Quality Variations with Worker Failures

In the final experiment we analyse the impact that random partial failures have on the accuracy of the face detection process executed by the two face tracking systems. For that, we use a set of 100 images created for the experiment in Section 4.3 that contain pre-computed faces to resemble a simulated "real-time" image feed while running 100 workers. We then instruct the chaos monkey script created for the experiments in Section 4.4 to gradually terminate 10, 25, 50, 75 and 90 of the face-detecting workers, with no pause between the terminations. For each run and each test image, we compare the accuracy of the face positions against ground truth using the Jaccard similarity coefficient (1). We then compare the variations in face quality over time for each run of the two frameworks.

Similarly to the results in Section 4.4, we observed that the Erlang program is more reliable in terms of the face detection, even while experiencing sudden partial failures. Figures 6–10 show a percentage-based representation of quality variations, as they are observed during each phase of the testing (i.e. with 10, 25, 50, 75, and 90 failed workers). We refer to these phases as the *n-failure sub-tests*, where $n \in [10, 25, 50, 75, 90]$.

The data series represented by the green line is the quality baseline quantified by the pre-determined face coordinates for each of the 100 images. The red and blue lines represent experimental face quality registered after the point of mass agent termination (normal and abrupt, respectively). The point of termination is depicted as the purple dotted line, which we have tried to keep as consistent as possible within the context of each sub-test.

In the case of the 10-failure tests (Figure 6) and the 25-failure tests (Figure 7), the Erlang application indicates no face quality penalties, which is due to the worker processes being restarted before the camera has a chance to cast any frames to failed workers. The graph lines resulting from these observations thus coincides entirely with the "ground truth" line. This, however, does not hold for the ROS program, where even for a relatively small number of terminated nodes, the high latency of node restarts causes some frames to be published to topics that are no longer connected to their dedicated workers.

Unlike other log- and topic-based message passing frameworks, such as widely used in the area of big data Apache Kafka [8], ROS does not keep track of undelivered messages to topic subscribers, meaning that all messages that do not have a subscriber are permanently lost. This characteristic is observed in Figures 6–10 as a sudden and steep decrease in face detection accuracy immediately after the terminations occur, followed by fluctuations in the quality percentage. These latter fluctuations are believed 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.

As the sequence of frames relayed through the detectors nears its end, the red and blue series sink sharply to a 0% flat line For each of the n-failure 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. 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-failure sub-test, the Erlang program also exhibits drops in face quality, following similar patterns to the ones described above for the ROS detectors. However, the impact is relatively small since Erlang worker processes are largely able to recover before the end of the image sequence. Thus, by virtue of its sub-millisecond restart delays, the Erlang face tracking system is always able to finish its operation with all 100 processes alive.

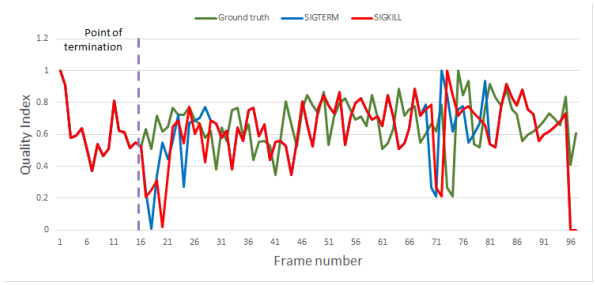
The last aspect to be addressed in the analysis of the experiment's findings relates to the sharp quality decline between frames 73 and 75 of most n-failure tests. The precipitous decrease is hypothesised to be caused by an anomaly in the OpenCV detection process, whereby for a short period the "real-time" detectors do not detect faces in an array of consecutive frames, even though the "static" (ground truth) detector does. As its causes have yet to be unequivocally determined, we may further examine the issue in the future.

From this experiment we conclude that *Erlang is capable not only to reduce robot component downtime, but also mitigate negative impact of these failures*. These findings become particularly compelling when considering the fact that many modern robots operate in safety-critical environments and any periods of unstable behaviour can result in hazardous functioning.

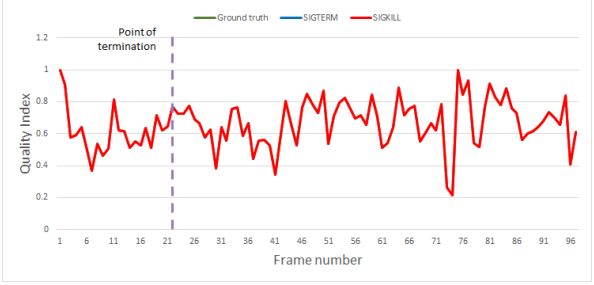
5. Conclusion

We investigate whether Erlang has the potential to address the scalability and reliability issues with communication between ROS nodes in robotic systems. The basis of the investigation is a pair of simple face-tracking applications: one using ROS message relay via topics, and the other a bespoke multi-process Erlang framework to transmit messages between functional units that reside 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.

We conduct five experiments to compare key aspects of the ROS and Erlang face trackers (Section 4). We find that Erlang communication scales better, supporting at least 3.5 times more active processes (700 processes) (Figure 4) than its ROS-based counterpart (200 nodes) while consuming half of the memory (Table 3). However, while both face tracking prototypes exhibit similar detection accuracy and transmission latencies when kept under 10 workers, Erlang exhibits a continuous increase in the total time taken to process a frame as more agents are added (Figure 5). Since the cause of this phenomenon is thought to lie with the external VM linking library, the implication is that a highly concurrent

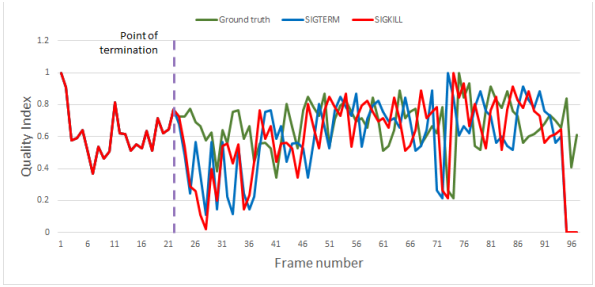


(a) ROS

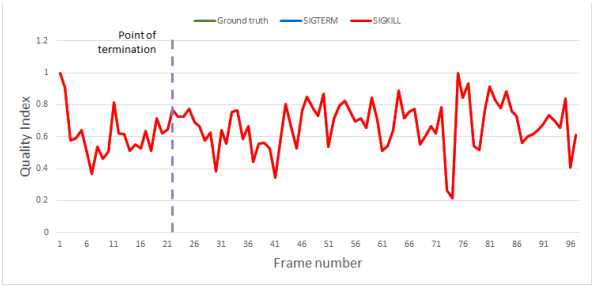


(b) Erlang

Figure 6. Quality Variation with 10% Failed Workers

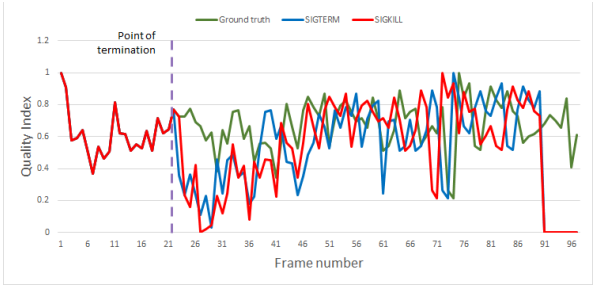


(a) ROS

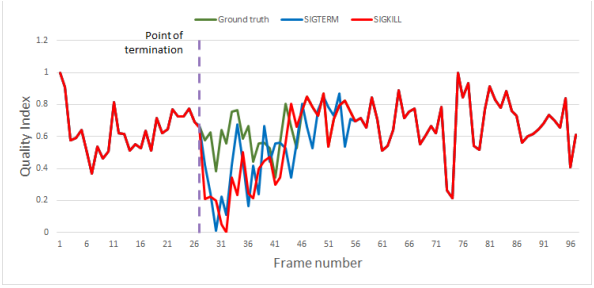


(b) Erlang

Figure 7. Quality Variation with 25% Failed Workers

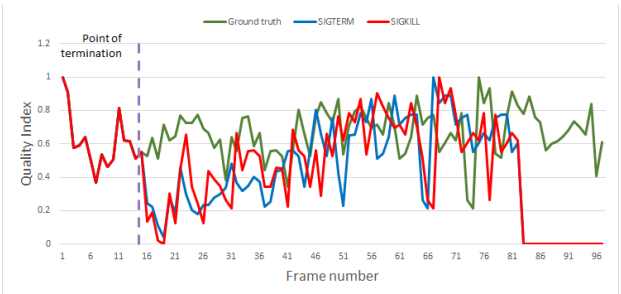


(a) ROS

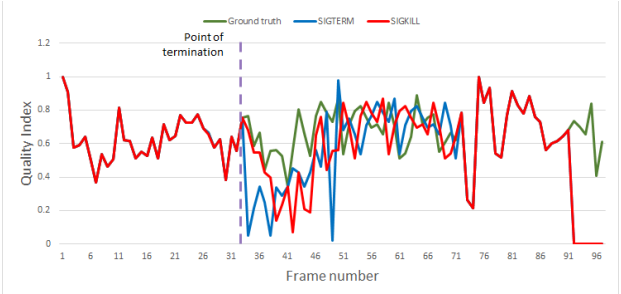


(b) Erlang

Figure 8. Quality Variation with 50% Failed Workers

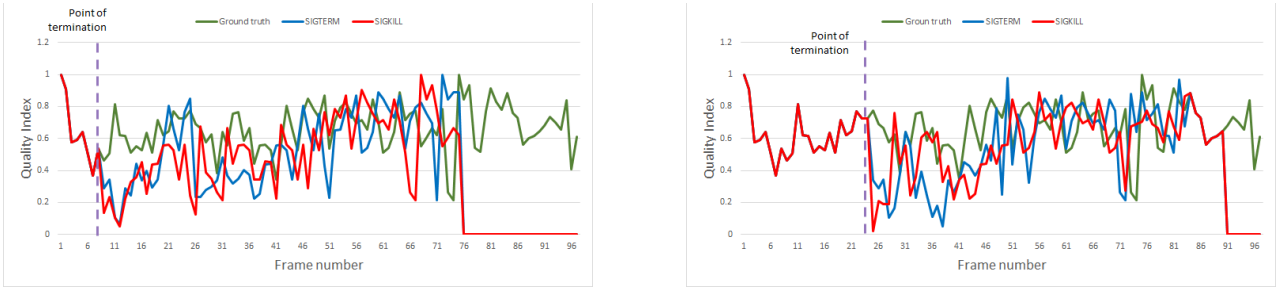


(a) ROS



(b) Erlang

Figure 9. Quality Variation with 75 Failed Workers



(a) ROS

(b) Erlang

Figure 10. Quality Variation with 90% Failed Workers

Erlang middleware would require and equally performant interface to efficiently interface with robotic components.

A study of reliability in the face of ROS node or Erlang process failures induced by a Chaos Monkey shows that both ROS and Erlang restart failed computations. However the Erlang processes restart 1,000–1,500 times faster than ROS nodes, e.g. it takes Erlang worker processes 0.352ms to restart while ROS worker nodes take 400ms). This enables Erlang not only to reduce robot component downtime, but also mitigate negative impact of these failures. For example, Figures 8 demonstrates that a successful completion of Erlang-based face tracking task is only mildly affected even when 50% workers fail.

As a direct improvement and continuation of the experiments presented in this paper, different variants of the face trackers could be developed and tested. For instance, they may benefit from writing files to a RAM disk or ROS' C++ API and Erlang-to-C/C++ natively implemented functions (NIFs) and port drivers. These may reduce some of the relay latency and remove the bias introduced by ErlPort (Figure 5(b)). It would be interesting to further investigate ROS and Erlang memory consumption providing detailed analysis of the data presented in Table 3. Our preliminary investigation would also benefit from other typical robot applications, like processing different types of sensor data and action coordination.

While the paper has focused solely on comparing ROS and Erlang as completely separate robotic middleware solutions, we believe that an integration of the two systems will 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 capabilities for concurrent communication and reliability, a more stable and better performing middleware can be developed. This would involve Erlang processes monitoring the execution of child processes represented by ROS nodes, and providing all inter-node communication in robots.

Acknowledgments

This work is partially funded by UK EPSRC grant AJITPar (EP/L000687/1).

References

[1] J. Adams. Distributed robots with Elixir. `http://www.erlang-factory.com/sfbay2014/josh-adams`, 2014. Online; retrieved July 29, 2016.

[2] N. Ando, T. Suehiro, and T. Kotoku. A software platform for component based RT-system development: OpenRTM-aist. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 87–98. Springer, 2008.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] G. Biggs. RT-Erlang. `https://staff.aist.go.jp/geoffrey.biggs/erlang.html`, 2010. Online; retrieved July 29, 2016.

[5] K. Conley. `rospy` package summary. `http://wiki.ros.org/rospy`, 2012. [Online; retrieved July 29, 2016].

[6] P. de Kok. Why ROS's timestamps are not enough. `http://pkok.github.io/2014/10/16/Why-ROS's-timestamps-are-not-enough/`, 2014. Online; retrieved July 29, 2016.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35 (2):114–131, 2003.

[8] N. Garg. *Apache Kafka*. Packt Publishing, 2013.

[9] B. P. Gerkey, R. T. Vaughan, K. Støy, A. Howard, G. S. Sukhatme, and M. J. Matarić. Most valuable Player: A robot device server for distributed control. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 1226–1231. IEEE, 2001.

[10] R. Gommers, T. Oliphant, R. Kern, C. Harris, and J. Millman. NumPy. `http://www.numpy.org/`, 2006. Online; retrieved July 29, 2016.

[11] S. Gruener and T. Lorentsen. KERL: Kent Erlang robotic library. `https://sourceforge.net/projects/kerl/`, 2013. Online; retrieved July 29, 2016.

[12] S. Grüner and T. Lorentsen. Teaching Erlang using robotics and player/stage. In *Proceedings of the 8th ACM SIGPLAN workshop on ERLANG*, pages 33–40. ACM, 2009.

[13] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 1968.

[14] S. S. Laurent and J. D. Eisenberg. *Introducing Elixir: Getting Started in Functional Programming*. O'Reilly Media, 2014.

[15] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234 (5323):34–35, 1971.

[16] S. Lloyd. Ultimate physical limits to computation. *Nature*, 406(6799): 1047–1054, 2000.

[17] A. C. Lutac. Real-time robot camera control in Erlang. Level 4 Dissertation, School of Computing Science, University of Glasgow, 2016.

[18] V. Nicosia and C. Santoro. Experiences from using Erlang for autonomous robots. In *In Proc. of 12th International Erlang User Conference (EUC)*, 2006.

[19] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP*. Morgan Kaufmann, 1992.

[20] *ROS Documentation*. Open Source Robotic Faundation, 2009. URL `http://wiki.ros.org/`. [Online; retrieved July 29, 2016].

[21] Open Source Robotic Foundation. ROS client libraries. `http://wiki.ros.org/Client%20Libraries`, 2009. [Online; retrieved July 29, 2016].

[22] OpenCV. Face detection using haar cascades. http://docs.opencv.org/3.1.0/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0, 2015. Online; retrieved July 29, 2016.

[23] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: An open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, page 5, 2009.

[24] M. Quigley, J. Faust, B. Gerkey, and T. Straszheim. roscpp package summary. http://wiki.ros.org/roscpp, 2012. [Online; retrieved July 29, 2016].

[25] C. Santoro. An Erlang framework for autonomous mobile robots. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG*, pages 85–92. ACM, 2007.

[26] C. Santoro and V. Nicosia. *ROSEN – The RObotic Simulation Erlang eNgine*, 2007. URL http://rosen.sourceforge.net/doc/index.html. [Online; retrieved July 29, 2016].

[27] A. Tseitlin. The antifragile organization. *Communications of the ACM*, 56(8):40–44, 2013.

[28] D. Vasiliev. ErlPort – connect Erlang to other languages. http://erlport.org/, 2013. Online; retrieved July 29, 2016.

[29] R. Vaughan. Massively multi-robot simulation in Stage. *Swarm Intelligence*, 2(2-4):189–208, 2008.