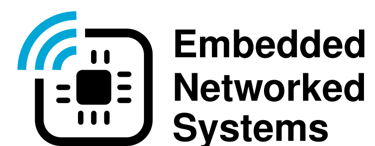
Delft University of Technology
Master of Science Thesis in Embedded Systems

Improving the Predictability of Event Chains in ROS 2

Charles Randolph
Supervisor: Dr. Mitra Nasri



TUDelft Delft University of Technology

Improving the Predictability of Event Chains in ROS 2

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Charles Randolph
15-03-2021

Author
Charles Randolph

Title
Improving the Predictability of Event Chains in ROS 2

MSc Presentation Date

31-03-2021

Graduation Committee

Koen Langendoen	Delft University of Technology
Mitra Nasri	Eindhoven University of Technology
Stefanie Roos	Delft University of Technology

Abstract

ROS 2 is a well-known software framework for the development of robotic applications. The open-source project enjoys active interest and participation from both academia and industry in applications spanning from autonomous vehicles to industrial robotics. ROS 2's success is underpinned by its disposition for collaborative development as well as its rich third-party package system. This has enabled developers to rapidly design, apply, and share their work while abstracting away the tedium of reimplementing common software constructs. Since the incarnation of ROS 1, ROS has shifted its focus away from medium-power workstations towards embedded systems. These efforts have born out ROS 2, a revision of the original ROS concept with a goal of supporting real-time applications.

Unfortunately, ROS 2 suffers from several major drawbacks that diminish its feasibility for applications with real-time requirements. Its idomatic C++ library contains an executor with a noncomformant callback scheduling model, which is neither priority aware or capable of preemptive callback execution. This makes ROS 2 considerably less time-predictable, as high-priority callbacks may suffer from priority inversions, and common real-time scheduling policies cannot be applied. Furthermore, ROS 2 is built upon a distributed messaging system, with system functions split across callbacks linked through a publish-subscribe communication mechanism. This frustrates the development of solutions for ROS, as it still requires developers cope with distributed chains of execution that must contend with precedence relations, synchronisation devices, and message passing overhead.

In response to these deficiencies, this thesis presents five major contributions: (1) two new executor implementations, with support for both callback prioritisation and preemptive callback execution. (2) a new priority-synthesis algorithm, which uses a graph model to map a set of callback-chain priorities to callbacks within ROS applications. (3) A simple feasibility test for ROS 2 applications based on callback chains. (4) A schedulability test for ROS 2 applications with harmonic chains. (5) A novel testing framework, which integrates the priority-synthesis algorithm, and enables the easy and rapid generation of test applications for development purposes.

The evaluation reveals that the new executors deliver superior performance in metrics such as worst-case response time, deadline misses, and jitter. On applications with up to 60% utilisation under rate-monotonic scheduling, the new executor implementations deliver no deadline misses, as opposed to the standard executor encountering them at just 10%. Finally, on high priority chains, the new executors can deliver up to **1300% times less** jitter over the standard.

"The solution of problems is the most characteristic and peculiar sort of voluntary thinking" – William James

Preface

In the world of collaborative robotics, no software framework is more popular than ROS. Be it automated sanding, self-driving vehicles, or drone autopilots - ROS has it all. The convergence between ROS and the real-time community, therefore, was always inevitable. Unfortunately, ROS has proven to fall short in terms of providing adequate facilities for supporting real-time applications. The body of this work dives into the reasons why these discrepancies exist, and proposes new and improved solutions.

Writing a thesis is often characterised as a solitary undertaking, but the reality is far from that. The work produced in this thesis could not have been accomplished without the input and advice of my direct supervisor, fellow students, and smattering of anonymous online personalities. I extend my deepest gratitudes to my supervisor, Dr. Mitra Nasri, for her unbounded enthusiasm, patience, and dedication to my work. Over the course of almost a year, she has consistently engaged, discussed, and pushed me to explore new angles and solutions - irrespective of her own personal circumstance or inconvenience. To all my friends and family, who have stood behind me with an unconditional mix of enthusiasm and bemusement, I am also extremely grateful. Finally, I wish to extend my thanks to the unsung communities of online contributors who have aided me in any capacity, be it applied or theoretical.

Charles Randolph

Delft, The Netherlands
31st March 2021

Contents

Preface ii

Glossary vi

1 Introduction 1
- 1.1 Problem . . . 2
  - 1.1.1 The ROS application model . . . 2
  - 1.1.2 Deficiencies . . . 3
  - 1.1.3 Verdict . . . 3
- 1.2 State of the art . . . 4
- 1.3 Research . . . 4
  - 1.3.1 Predictability . . . 4
  - 1.3.2 Objectives . . . 5
- 1.4 Solution . . . 5
  - 1.4.1 Contributions . . . 5
- 1.5 Approach . . . 5
- 1.6 Organization . . . 6

2 Background 7
- 2.1 The ROS software stack . . . 7
- 2.2 Communication patterns and callback chains . . . 8
  - 2.2.1 Communication patterns . . . 8
  - 2.2.2 Callback chains and their characteristics . . . 9
- 2.3 Scheduling in RCLCPP . . . 10
  - 2.3.1 Multi-threaded executor . . . 12

3 Motivation and problem formulation 13
- 3.1 Challenges . . . 13
- 3.2 Motivation . . . 14
- 3.3 Problem formulation . . . 15
  - 3.3.1 Callback chains . . . 15
  - 3.3.2 Predictable ROS scheduling . . . 16
  - 3.3.3 Priority synthesis . . . 16
  - 3.3.4 Assumptions . . . 16
  - 3.3.5 Experimental setup . . . 17

4 Related work 18
- 4.1 Predictability in ROS . . . 18
  - 4.1.1 Predictability through reimplementation . . . 18
  - 4.1.2 Predictability through analysis . . . 18
  - 4.1.3 Predictability through optimisation . . . 19
- 4.2 Scheduling theory and ROS . . . 19
  - 4.2.1 The job shop problem (JSP) . . . 20
  - 4.2.2 Example . . . 20
  - 4.2.3 JSP and ROS scheduling . . . 21

5 Solution 23
5.1 Preemptive, priority-aware executor . . . 23
5.1.1 Executor design considerations . . . 23
5.1.2 Assigning priorities to callbacks . . . 24
5.1.3 Thread dispatch model . . . 24
5.1.4 Producer-consumer model . . . 25
5.2 Callback priority synthesis . . . 27
5.2.1 Synthesis model . . . 27
5.2.2 Synthesis algorithm . . . 28
5.2.3 Example . . . 29
5.3 Feasibility test based on individual chains . . . 31
5.3.1 Feasibility test algorithm . . . 31
5.4 Schedulability test for harmonic chains . . . 32
5.4.1 Worst-case response-time analysis . . . 32
5.4.2 Converting ROS 2 application graphs into FPPS task-sets . . . 33
5.4.3 Example . . . 33

6 Evaluation Framework 36
6.1 Overview . . . 36
6.2 ROSGEN . . . 38
6.2.1 Generation rules . . . 38
6.2.2 Generating callback chains . . . 39
6.2.3 Random merges and synchronisations . . . 40
6.2.4 Utilisation and period assignment . . . 41
6.2.5 Benchmark assignment . . . 42
6.2.6 Priority assignment . . . 42
6.2.7 Feasibility check . . . 43
6.2.8 Application generation . . . 43
6.3 Tracing and logging . . . 44
6.4 Analysis . . . 44
6.5 Message data . . . 44

7 Evaluation 46
7.1 Experimental Foundation . . . 46
7.1.1 Performance . . . 46
7.1.2 Investigation . . . 47
7.2 Experimental setup . . . 48
7.2.1 Environment . . . 48
7.2.2 Core allocation . . . 49
7.3 Experiments . . . 49
7.3.1 Overhead of executor implementations . . . 49
7.3.2 Effect of utilisation on executors . . . 53
7.3.3 Effect of chain length on executors . . . 56
7.3.4 Evaluating executor choices for high-priority time-sensitive callback chains . . . 59
7.3.5 Experimental hypothesis . . . 60
7.3.6 Effect of priority-synthesis algorithm . . . 62
7.4 Findings . . . 64
7.4.1 Summary . . . 64

8 Conclusion 66
8.0.1 Summary . . . 66
8.0.2 Contributions . . . 66
8.1 Discussions . . . 67
8.2 Future Work . . . 68

A APPENDIX TITLE **71**
A.1 Executor Implementations . . . 71
A.1.1 Single-threaded executor . . . 71
A.1.2 Thread-dispatch executor . . . 72
A.1.3 Producer-consumer executor . . . 72
A.2 Evaluation framework . . . 74
A.2.1 Merge/synchronisation algorithm . . . 74
A.2.2 Generated assets . . . 75

Glossary

BCRT Best-case response time

DAG A graph with directed edges in which there are no cycles

DDS Machine-to-machine standard for data exchange

NWCRT Normalised worst-case response time

RCL ROS client library which enforces the ROS application model

RCLCPP A C++ wrapper interface for the ROS client library

RCLPY A Python wrapper interface for the ROS client library

RMW ROS middleware interface providing minimal and essential ROS functionalities

WCET Worst-case execution time

WCRT Worst-case response time

Chapter 1

Introduction

Robot operating system, best known by its acronym ROS, is a free and open-source software framework for the development of robot applications. ROS provides developers with a suite of software tools that enable the rapid creation of complex software systems. A key strength of ROS is its disposition for collaborative software. This has enabled ROS to become one of the most popular frameworks for both development and research purposes in the robotics field. Consequently, it has amassed a significant community following, with hundreds of actively maintained packages, thousands of users, and a yearly conference called ROSCon. [5]

Core features of the ROS framework include a uniform application model, modular communication backend, and support for a wide array of third party software packages. ROS exists in two variants: A legacy version, often referred to as just ROS or ROS 1, is the original open-source project. A modernised version, called ROS 2, was designed to accommodate a host of new use cases and technologies for robotics that was too disruptive for the legacy framework. A key motivation for the redesign was the desire to expand ROS to small embedded platforms, with a focus on real-time applications. [11]

However, despite the express goal of bringing real-time capabilities to ROS, the current state of ROS 2 still suffers from several shortcomings. ROS applications are not only inhibited by the absence of a well-defined scheduling model, but by architectural decisions that underpin the framework. These shortcomings include but are not not-limited to a non-standard execution model, lack of callback prioritisation, and lack of real-time scheduling policies (see section 1.1.2). These altogether limit the breadth of applications suitable for ROS, and underscore a need for further changes to ROS that can increase the predictability of the framework.

The importance of real-time systems for robotics is underscored by an increasing demand for robots that can make intelligent decisions in real-time, and act in a dependable fashion [24]. This interest is reflected across both industry and academics, and manifests itself in areas such as autonomous driving technology, automated aircraft flight software, and manufacturing automation. As robotic applications perform increasingly important tasks, they also inherit a proportional amount of responsibility for ensuring they operate both safely and reliably. Real-time systems are indispensable in providing application developers with the means to produce provably safe systems, and are therefore widely used in time-critical applications.

There is thus considerable interest and value in bringing real-time capabilities to ROS [1]. Consequently, various efforts have been undertaken in order to improve ROS's real-time capabilities. This has ranged from extensions, to modifications, to the complete reworking of ROS 1 into ROS 2 (see section 1.2 for a survey). Unfortunately, many third party efforts are either insufficient or developed for ROS 1 (for which there is not forwards compatibility).

[1] In this work, all references to ROS refer to ROS 2. Legacy ROS will be explicitly denoted as ROS 1

And the redesign to ROS 2 still suffers from the aforementioned flaws (e.g. the absence of a well-defined scheduling model).

1.1 Problem

To accurately describe the problems that hinder ROS in the domain of real-time applications, we provide a brief overview of the ROS application model. We then end by enumerating the various deficiencies identified with the model.

1.1.1 The ROS application model

While ROS reduces the difficulty involved in prototyping robot applications, it applies abstractions that come at a cost transparency and control of the scheduling behaviour. ROS encourages applications by design to adhere to an application model. This application model, seen in Figure 1.1, organizes executable code into callbacks. Callbacks are then organised into nodes (ostensibly object-oriented programming (OOP) classes) which share access to common resources. Finally, nodes are organised into executable entities, forming a set of independent executables that (when executed concurrently) represents the complete application run-time.

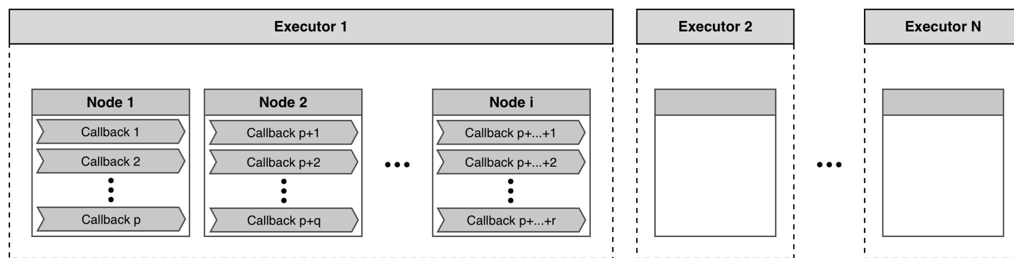


Figure 1.1: **ROS application organisation (RCLCPP)**

This hierarchical and distributed structure gives ROS modularity, and is suitable paired with a *publish-subscribe* communication mechanism. The publish-subscribe mechanism allows callbacks to be invoked when messages are published to *topics* they are subscribed to. These callbacks themselves may also publish to topics, invoking other callbacks in a one-to-many relationship (see Figure 1.2). Timers provide a means for the periodic activation of callbacks. Communication between entities within ROS is handled out on the backend by an implementation of the data distribution service (DDS) specification. This implementation is designed to be swappable, allowing ROS application developers to switch vendors with minimal effort.

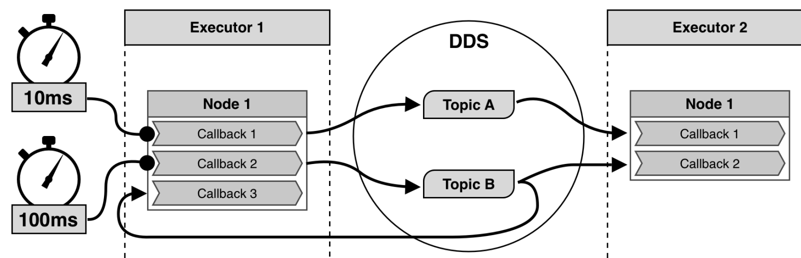


Figure 1.2: **ROS publish-subscribe mechanism**

Although this application model standardises the organisation and structure of applications, *it does not describe "how" and "when" callbacks respond to topics they are subscribed*

to, and the course of action that decides which (of possibly many) ready callbacks should be run. Effectively, no guidelines exists for scheduling callbacks, and so ROS scheduling behavior is both ambiguous, and as we discuss later, antithetical to the needs of real-time applications in practice. This indistinctness makes ROS unsuitable for applications with hard real-time requirements, as it (1) results in schedulers with complex and nonconformant behaviour, (2) allows priority inversions to arise, and (3) introduces delays in time-sensitive callback chains.

1.1.2 Deficiencies

The shortcomings outlined in the application model of section 1.1.1 are restated below, along with other identified drawbacks.

- **Complex execution model.** The absence of a specification for how callback execution is conducted results in the execution model being determined by the creators of the ROS front-end language-specific libraries. As these libraries are open to implementation for different programming languages, the execution behavior can vary considerably (see section 2.1). Furthermore, library creators may not always design their implementations with real-time scheduling policies in mind, forcing application developers to contend with counterintuitive and sometimes inferior schedulers. The consequence is that ROS applications may suffer from priority inverions, and developers are forced to verify their applications empirically through repeated testing.
- **Lack of explicit callback prioritisation.** Callbacks have no explicit priorities in ROS. They inherit a priority from (1) the underlying process that executes them and (2) the scheduling decisions ingrained in the language-specific front end library. These scheduling decisions may be based upon arbitrary characteristics of callbacks, such as registration order (see section 2.3). The consequence is that developers cannot directly prioritise important callbacks.
- **Lack of preemptive callback execution.** Although the execution model of ROS is determined by the front-end client-facing language-specific library, its most idiomatic front-end library does not support preemptive callback execution on the same executing ROS process. The lack of preemptive execution limits the ability of developers to reduce the response-time of time-sensitive chains of callbacks in ROS, and causes priority inversions by allowing less important callbacks to delay the completion of more important ones. A more in depth explanation on ROS execution behaviour is provided later in section 2.3.
- **Lack of real-time scheduling policies.** As ROS provides no mechanisms for controlling the scheduling policy that governs callback execution, common real-time scheduling policies are not present. This also hinders application developers from implementing their own schedulers, or customising the existing one. This is particularly needed when an application's quality of service depends on its response time.
- **Lack of adequate tooling for synthesising predictable applications.** There exists a gap in both tooling and analysis for application designers. Although response-time analyses are just now emerging for ROS (see section 1.2), practical tooling for assisting developers in designing applications does not exist. Such tooling might help developers select callback priorities such that the end-to-end response time of time-sensitive chains is minimised, or help rapidly prototype application designs. Of course, this tooling presupposes the presence of a priority-aware scheduler, which does not yet exist in ROS.

1.1.3 Verdict

The deficiencies identified and discussed can be summarized as follows:

1. The absence of predictable execution due to a non-standard scheduling model

2. A lack of adequate tooling for synthesising predictable applications (given the hypothetical existence of a well-defined scheduling model)

1.2 State of the art

The state of the art in solutions for the problems raised in section 1.1 are addressed independently. To the best of our knowledge, there exists no work which has addressed *both* the lack of predictable execution in ROS *and* the need for tooling which can synthesize more predictable ROS applications. Therefore, we only survey the state of the art in efforts to improve the predictability of ROS applications. We visit the subject of synthesis later in related work (see Chapter 4).

There are three categories of approaches when it comes to improving the predictability of ROS applications:

1. **Predictability through reimplementation**: This approach involves the creation of a separate, deterministic version of the ROS scheduling model. The Micro-ROS project [18] extends ROS concepts to microcontrollers running real-time operating systems. They provide a two-layer executor solution with static order scheduling policies and callback group level preemptions. However, developers must provide schedules for the executor in the ROS client library (RCL), and the RCL layer changes are a departure from the core ROS project.

2. **Predictability through analysis**: This approach subjects ROS to special operating conditions, such that a response-time analysis can be produced for applications. Casini et al. [6] are the first to propose a response-time analysis for ROS 2 applications for predicting the end-to-end latency of time-critical processing chains. However, their solution simply identifies and models ROS 2 scheduling behavior, and no changes are proposed for an improved executor. Tang et al. [28] build upon this work and propose an improved response-time analysis and prioritisation scheme. However, their priority scheme does not account for more complex ROS constructs such as message synchronisation mechanisms, depends on gaming the inherent hierarchy present, and relies on model assumptions that they later found to not hold under emprical evaluation.

3. **Predictability through extensions**: This approach involves optimising or extending portions of ROS and validating improvements in performance through empirical testing. Gutiérrez et al. [12] measure the effectiveness of DDS QoS profiles and priority thread assignments to minimise round-trip latency tests in ROS 2. They provide insight into optimal configurations for ROS 2 applications, but don't address real-time defects. Saito et al. [23] provide a real-time scheduling framework for ROS 1 applications. However, their scheduler cannot contend with callback chains that share callbacks, and is not interoperable with ROS 2. Similarly, Takase et al. [26] provide a custom platform for running ROS nodes on low-power devices using a real-time system. However, the work is also for ROS 1, and does not address real-time execution properties of ROS. Finally, Dehnavi et al. [8] provide an implementation of a resource-constrained variant of DDS called XRCE-DDS for a time-predictable system-on-a-chip platform called *CompSOC*. They then use scenario aware dataflow models (SADF) with markov chains to compute long-term throughput in ROS applications. Their work primarily concerns DDS, however, and therefore does not address issues of predictability in the ROS execution model.

1.3 Research

1.3.1 Predictability

Before outlining the research objectives, it is necessary to define what is meant by *predictability*. In a real-time context, a system is said to be *time predictable* if the response-time

jitter of its tasks is small (i.e. the difference between the worst-case response-time (WCRT) and best-case response-time (BCRT) is minimal). When framed as a real-time system, we may say that ROS 2 exhibits poor timing predictability. This is because its disposition for causing priority inversions and non-preemptive callback execution result in highly variable response-times. Throughout the remainder of this work, references to the term *predictability* should be understood as referring to *timing predictability* as defined above.

1.3.2 Objectives

The intentions of this work is two fold:

1. To identify and implement the necessary changes to the ROS language-specific client-facing libraries that improve the real-time capabilities of the ROS scheduling model. This entails enabling support for (1) the prioritisation of callbacks and (2) preemptive callback execution.

2. To explore how an improved scheduling model can be leveraged to create more predictable ROS applications by synthesising callback priorities for applications. The objective being that the synthesis scheme reduces the end-to-end response-time of time-sensitive callback chains.

1.4 Solution

The solution is composed of a series of distinct steps. First, sources of priority inversion and non-preemptive execution are identified in the existing ROS scheduling model. Next, an alternative scheduling model is proposed which addresses these problems by providing developers with a priority-aware executor with support for preemptive fixed priority scheduling. Finally, a tool is designed and implemented which synthesizes the necessary priority assignment schemes for minimizing end-to-end latency of priority chains in user applications.

1.4.1 Contributions

The primary contributions of this work are thus:

1. A ROS 2 executor implementations with support for preemptive fixed-priority scheduling (in two variants).

2. An algorithm that synthesises callback priority assignments for ROS 2 applications

3. A feasibility test for ROS 2 applications

4. A schedulability test for ROS 2 applications with harmonic chains

5. A toolchain that enables automatic generating and testing of ROS 2 applications

1.5 Approach

In order to keep the scope of the work manageable, both the solution and evaluation are conducted on a standard PC running Ubuntu 18.04.5 LTS (Bionic Beaver), with ROS 2 (Foxy Fitzroy). Additionally, all work concerning the ROS code base deals strictly with the ROS C++ client facing library (RCLCPP). The decisions governing these choices are two-fold. The first is that the C++ implementation is the most true-to-life choice for the development of real-time applications, with minimal overhead and few abstractions. The second is that the C++ client facing is the idiomatic ROS interface [3], which increases the potential practical value and reach of the work. Finally, a special emphasis is placed on maximising compatibility with standard ROS by introducing as few disruptive measures as possible in the pursuit of our goal.

Schedule synthesis is approached as a scheduling theory problem, and an attempt is made to relate it to existing problems in Chapter 4. A solution for schedule synthesis should be an algorithm that proposes a priority assignment scheme for a ROS 2 application that reduces the end-to-end response time of specific chains (or sub-chains) of callbacks. It will also not attempt to modify the input application.

1.6 Organization

The thesis is structured as follows: Chapter 2 the necessary contextual material needed to better understand the problem. Chapter 3 introduces the motivations behind the work, and formally defines the problem. Chapter 4 discusses related work. Chapter 5 outlines the solution, and Chapter 6 introduces the custom testing framework. Chapter 7 presents the evaluation. Finally, Chapter 8 presents closing discussions and possible avenues for future work.

Chapter 2

Background

The aim of this chapter is to provide the requisite information to understand *why* and *how* ROS is unsuitable for real-time applications. This background is provided by introducing the ROS software stack, communication patterns, and RCLCPP scheduling.

2.1 The ROS software stack

ROS 2 has been carefully designed to maximize its modularity and flexibility, an approach reemphasized in the development of ROS 2. Figure 2.1 provides a breakdown of the software layers that compose a typical ROS 2 application. From an abstract point of view, the user application is built on top of the ROS framework, and runs as a standard program in user space, subject to the operating-system scheduler. The various layers of the framework enumerated in the image are elaborated upon below.

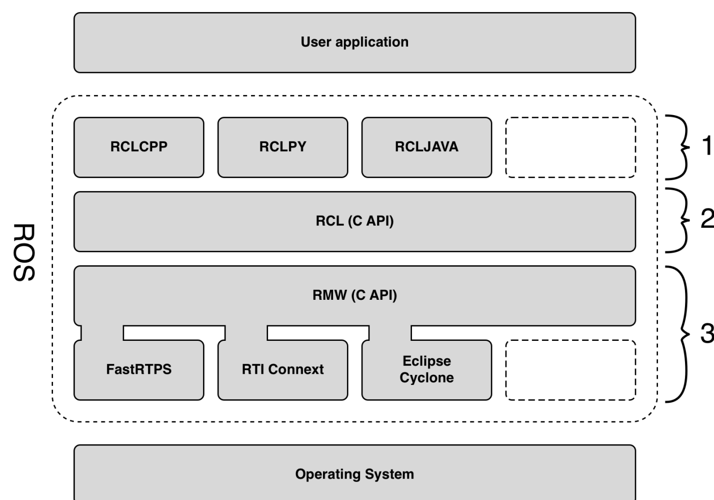


Figure 2.1: **ROS 2 software stack** [9]

1. Language-specific client library interface

The language-specific client library (figure 2.1, brace 1) is a wrapper for the underlying ROS client library. This layer interfaces directly with the user application and provides access to ROS constructs and concepts though a particular programming language. Programming languages may espouse different paradigms and runtime behavior (e.g. employing parallelism or subroutines) [6]. Therefore, this library also determines how executable units are scheduled, and may vary between implementations. The most popular language-specific

client libraries are RCLCPP, for the C++ programming language, and RCLPY, a Python based implementation.

2. ROS client library

The ROS client library (figure 2.1, brace 2) manages access to the ROS graph by providing a language agnostic, common implementation. This layer is written in C, and interfaces with both the language specific client library and middleware layer. This layer also enforces the application model. Topics, services, and actions, are examples of middleware concepts provided [30].

3. DDS and ROS middleware interface

The ROS middleware layer (figure 2.1, brace 3) provides the most minimal and essential functionalities of ROS to the underlying RCL layer. It is effectively just an interface and left open to implementation. Implementations of this interface are provided by various DDS vendors, and are designed to interact with their specific DDS implementation. This is indicated with the connecting segments between the various DDS vendor implementations and ROS middleware interface (RMW) in figure 2.1. The role of DDS is to provide an configurable and networked model supporting a publish-subscribe communication pattern, and abstracting away the job of node discovery and message serialisation/deserialisation.

2.2 Communication patterns and callback chains

2.2.1 Communication patterns

RCLCPP provides several communication pattern in addition to the publish-subscribe mechanism. It also enables developers to trivially design more sophisticated constructs such as synchronisation mechanisms. This sub-section visits both the template communication patterns provided by ROS, shared callbacks, and message synchronisation techniques.

Template communication patterns

RCLCPP provides several communication patterns for developers out of the box. These are designed to abstract common patterns of communication between nodes. They are illustrated in figure 2.2.

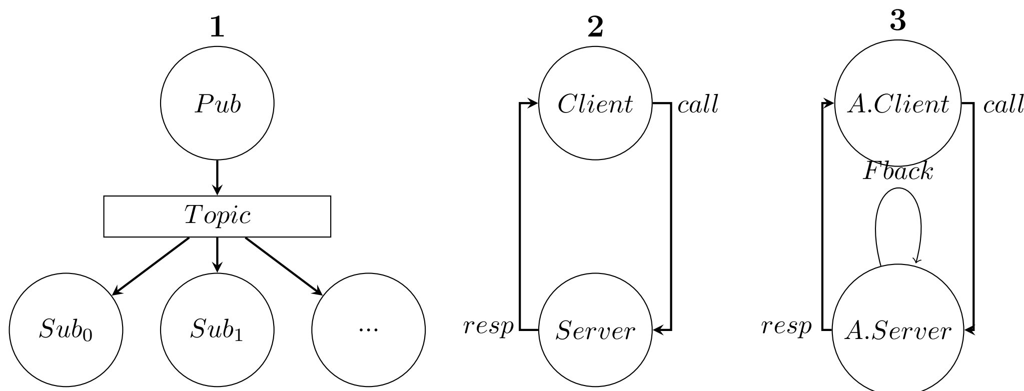


Figure 2.2: **ROS 2 communication patterns**

1. **Publish-Subscribe.** As the staple communication pattern of ROS, publish-subscribe (figure 2.2, 1) is a straightforward communication mechanism that distributes a message published to a topic to one or more subscribers (one-to-many).

2. **Services.** A service (figure 2.2, 2) enables a caller (called a client) to perform a remote procedure call (carried out by a server) and receive a response in return (call-response).

3. **Actions.** Actions (2.2, 3) provide an abstraction for long running tasks. An action client creates a goal, which an action server attempts to fulfill. As the task is completed, steady feedback is provided. Unlike publish-subscribe and services, actions are not truly their own patterns, but built upon services and publish-subscribe. Actions may also be cancelled by the client.

Message synchronisation mechanisms and shared callbacks

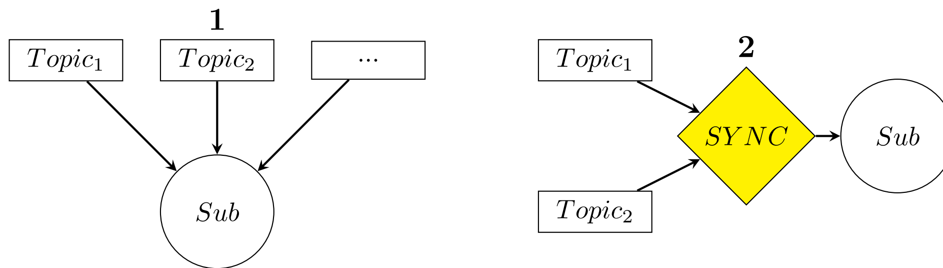


Figure 2.3: **ROS 2 shared callbacks and message synchronisation mechanisms**

1. **Shared callbacks.** In figure 2.2, 1, we see that a publisher may publish to multiple subscribers. Likewise, a subscriber may be subscribed to multiple publishers. The very same callback may be used as long as the message type expected by the callback is used by all the topics. This means that callbacks can be *shared* between different chains of callbacks. We revisit the relationship between shared callbacks and callback chains later in sub-section 2.2.2.

2. **Message synchronisations.** Applications combining data from multiple sensors often require synchronised messages to maximise the utility of the information. Synchronisations in ROS can be implemented either by the developer themselves or through the use of packages [13]. We consider a simplistic synchronisation model, which can be seen in figure 2.3, 2. The yellow diamond represents the synchronisation mechanism, which we define to have a binary input. The synchroniser produces an output message whenever it receives a message from *both* of the input topics. Just as with shared callbacks, message synchronisation mechanisms impose interesting constraints on the design of a solution. We discuss these later with respect to callback chains in sub-section 2.2.2.

2.2.2 Callback chains and their characteristics

In section 1.1, a mention was made of so-called *callback chains*. Put succinctly, the communication patterns of ROS result in functionality being distributed across chains of callbacks. A callback chain may be triggered either periodically by a timer, or via an external event. This sub-section begins by defining a callback-chain, and then focuses on various types of chain interactions. Understanding these interactions is necessary for deriving functional synthesis algorithms later, where time-sensitive callback chains (or sub-chains) may have various dependencies.

Callback chains

ROS applications as directed graphs. A ROS application is composed of callbacks that may be triggered in various ways. A callback may be invoked periodically by a timer or sporadically by an environmental event. A callback may also be invoked by a message arriving on a topic that it is subscribed to. These relations may be generally modelled as

a *directed acyclic graph* (DAG) (we assume there are no call cycles). Callbacks form nodes within the graph, and edges denote call relations between callbacks over certain topics.

From DAGs to callback chains. To make sense of callback behavior, the directed graph of callbacks that composes a ROS application may be split into ordered sequences of callbacks. These are called *callback chains*. A callback chain is simply a sequence of publish-subscribe linked callbacks triggered by a single source. A callback chain may share callbacks with other chains, but must contain at least one callback not found in any other chain. The purpose of defining callback-chains is to *capture aspects of functionality* in a ROS application (e.g. the sequence of callbacks from a sensor reading to a decision). As such, a callback chain has a value, or *priority*. We will more rigorously define callback chains later in section 3.3.1 when we revisit the topic.

The impact of communication patterns on callback chains

We previously described how ROS applications may contain shared callbacks, as well as message synchronisation mechanisms. These impose some interesting constraints on callback chains. We summarise them below:

Impacts of shared callbacks.

1. Callbacks that are shared between a high priority chain and a low priority chain creates a conflict of priority.
2. Chains sharing callbacks must be reentrant if preemptive behavior is present, and therefore the shared callbacks should not depend on a state (i.e. data local to a node). Shared callbacks that access persistent data should then protect accesses using mutual exclusion. However, this topic is not the focus of this work.

Impacts of message synchronisation mechanisms.

1. Synchronised chains place an implicit priority on all predecessors leading up to the synchronisation point within a chain. This causes a dependency between the inputs, which may result in a high priority chain depending on a low priority one.

2.3 Scheduling in RCLCPP

In order to manage the execution of callbacks in ROS, the RCLCPP library relies upon entities called *executors*. Executors are available in both a single-threaded and multi-threaded variant. They multiplex callbacks for timers, subscriptions, and services within ROS. As such, executors assume the role of a de-facto scheduler. The first thorough documentation of ROS scheduling behavior is provided by Casini et al.[6], and targets the single-threaded ROS executor. They define a set C to contain all callbacks assigned to a particular executor, with subsets C^{tmr}, C^{sub}, C^{srv}, and C^{clt} comprising all callbacks for timers, subscribers, services, and clients, respectively. New messages to be processed are made available to the executor by interfacing with the middleware layer through RCL. However, newly ready messages are not immediately considered. Instead, the executor queries the ready set in intermittent, irregular intervals. Between queries, it processes all callbacks in its copy of the set. The executor scheduling algorithm is reproduced below in figure 2.4:

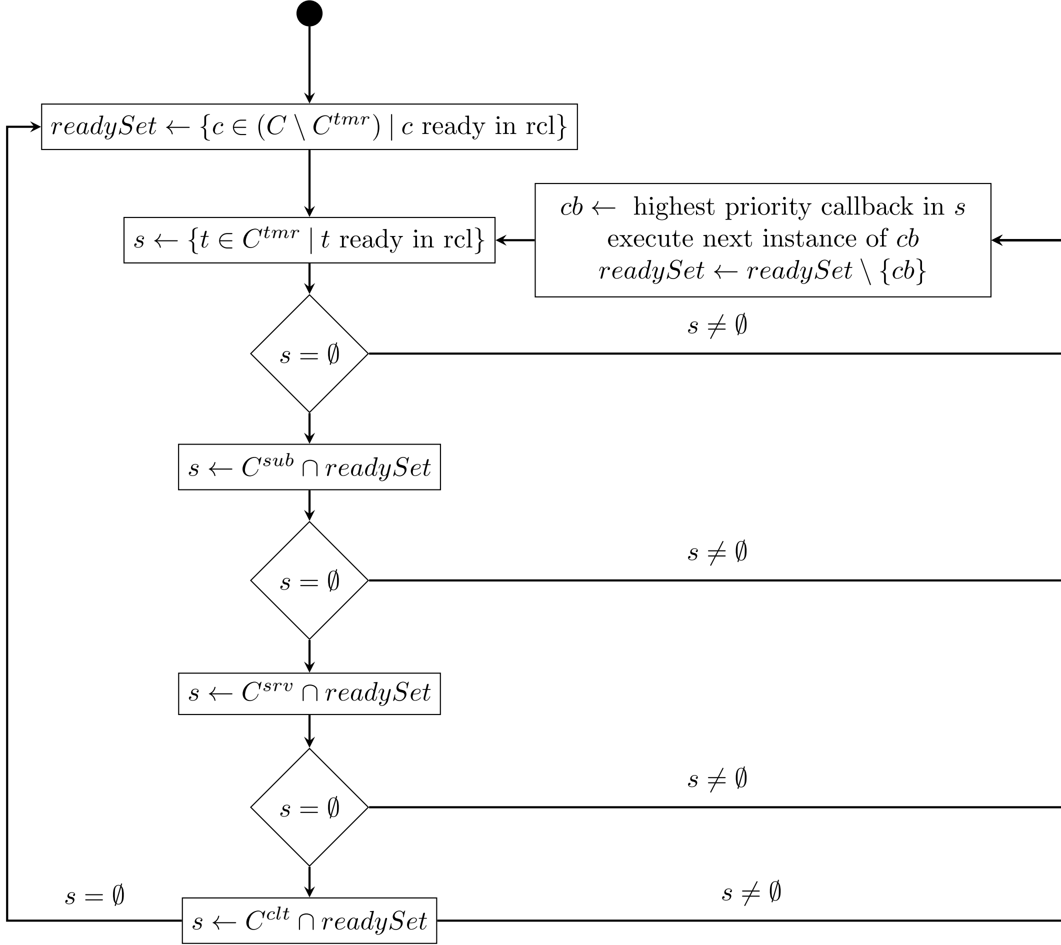


Figure 2.4: **(Reproduction) ROS Scheduling**[6]

Casini et al. outline the following peculiarities of the algorithm shown in figure 2.4:

1. An implicit priority hierarchy exists between callback types. This is because all timers are always processed first, followed by all subscriptions, then all services, and finally all clients.

2. An implicit priority hierarchy exists between callback instances of a type within the callback queue. This is because ready callback instances are evaluated in the order they registered.

3. Non-timer callbacks are taken from a ready set, and not a ready list. This means the algorithm is unable to consider multiple ready instances of a particular non-timer callback.

4. The set *readySet* is only updated after all callbacks within it have been processed. The time at which the set is updated is called a *polling point*, and the interval between two consecutive polling points a *processing window*. This may cause priority inversions to occur, since lower priority callbacks within the *readySet* extend the processing window, delaying higher priority callbacks from being considered until all callbacks have been completed at the next polling point.

5. Once a callback is being executed, it cannot be preempted. The *non-preemptive* nature of the executor is amplified by the presence of the processing window and polling point.

2.3.1 Multi-threaded executor

Although this work is principally concerned with the single-threaded executor, it is necessary to also clarify the role of the multi-threaded executor. In its current implementation, the multi-threaded executor executes the same algorithm as the single-threaded executor. However, the pool of ready callbacks is simply drawn from by multiple threads implementing the single-threaded executor algorithm.

Chapter 3

Motivation and problem formulation

This chapter describes the challenges (section 3.1) associated with the research problem, leveraging the background information introduced in chapter 2. These challenges are further supported by an example in the motivation sub-section (section 3.2). Finally, the goals and expectations of the solution are rigorously established in the problem formulation (section 3.3).

3.1 Challenges

ROS presents a number of challenges for improving the predictability of its run-time behavior, along with any accompanying synthesis tools. These are enumerated below:

1. **Distributed architecture**: ROS is fundamentally a distributed system, with applications consisting of one or more independent cooperating processes, all linked through a one-to-many publish-subscribe mechanism. This complicates predictability and analysis, as the run-time behavior may be dependent on multiple concurrent processes, possibly spread across physically separate machines.

2. **Undefined scheduling model**: ROS provides an application model which dictates how ROS applications are to be structured, but no standard for how to conduct execution. Rather, these are left as implementation decisions to developers of the language-specific client interfaces that interact with the common ROS client library (RCL) [6]. Consequently, while ROS is successful in abstracting its application model from any particular language, the scheduling model is inconsistent and differs between language specific implementations.

3. **Heterogeneous platforms**: ROS runs as a userspace application, meaning that it has to compete for CPU time with other running applications. Its execution behavior is dictated by the scheduler of the operating system on which it runs. Furthermore, ROS is supported by most common operating systems, and has even been ported to run on microcontrollers [18]. Consequently, timing predictability on even one platform, let alone a mix of heterogeneous ones, is entirely reliant on how the host system schedules processes and assigns priorities. This can vary widely between systems, severely complicating the production of reliable, reproducible behavior across platforms.

4. **Event activation types**: ROS is designed to accommodate both time-driven (i.e. triggered by a periodic timer) and event-driven (i.e. triggered by an environmental cue) activities. The interplay of these activities complicates analysis, as existing analyses are designed for purely periodic systems. [6]

3.2 Motivation

Example. The necessity of a time-predictable executor for ROS 2 applications is demonstrated using an example. Figure 3.1 presents a ROS application inspired from an autonomous vehicle platform by Kato et al [14]. The platform makes use of lidar sensors that produce 3D point-cloud data, along with omnidirectional camera sensors for detecting moving objects. The images produced are passed to callbacks that perform pattern analysis using algorithms such as Deformable Part Models (DPM) (see *Obj identify* in Figure 3.1). This 3D point-cloud data maps distances to coordinates in the images, which when fused provide the distance to detected objects (see *Scan 2 image* and *fusion* in Figure 3.1). This distance information may be used to track and act upon classified objects and perform local navigation (see *Local plan*, and *Obstacle detect* in Figure 3.1). Finally, the information pipeline results in actuations which adjust the vehicles velocity and direction (*actuate* in 3.1).

Hard, soft, and firm deadlines. Given that autonomous vehicles should prioritise safety, functions of the system should be prioritised in a manner that reflects this principle (i.e. ranked by their importance in providing safe operation). We call functions of a system *hard* when the consequence of missing their deadlines results in a total system failure. Using the example, an autonomous vehicle that fails to detect a pedestrian or blocking object may cause harm to human life and/or destruction of the vehicle. Therefore, the deadline for obstacle detection is hard. When deadline misses are tolerable for system functions, two classifications may apply: A *firm* deadline is one for which results after the deadline are useless, and a *soft* deadline is one in which those results may still have value (albeit diminished).

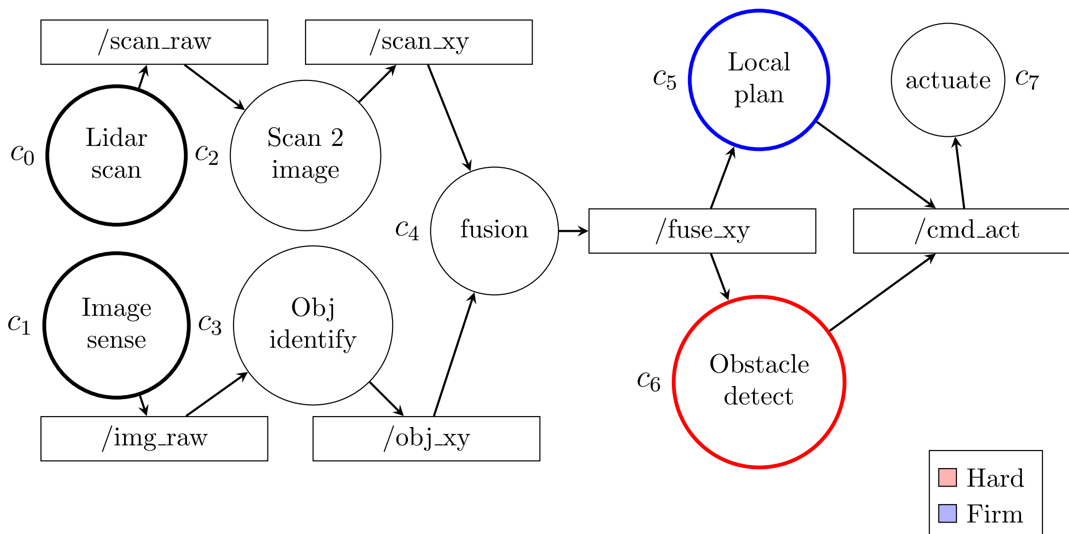


Figure 3.1: **Example of a ROS application with varying priority functions. Callbacks are indicated with circles, and timer callbacks have bolded borders. Boxes contain topics, and the arrows denote publish-subscribe relations. The subscription c_i identifies a callback**

Priorities for time-sensitive paths. For the system in Figure 3.1, *hard* and *firm* system functions are indicated with colored borders around their callbacks. We can see that obstacle detection is hard, as being able to detect and brake when sensing an obstacle is necessary for safety. Local planning is firm, as missing fused sensor data is not helpful for navigation, but not immediately dangerous either. A ROS application should therefore prioritise the hard callback over the firm callback. Unfortunately, the standard single-threaded executor provided with RCLCPP provides no ability to do so. The scheduling peculiarities mean that callbacks will be prioritised by type and registration order (see section 2.3). Even carefully choosing the types and registration order does not remove the risk of priority inversions due to the polling period and lack of global priority among multiple executor

instances. Reducing the end-to-end response time of priority callback chains is all but completely infeasible.

Synthesising priorities. Although the ability to ascribe priorities to callbacks (that are respected by a ROS scheduler) would go a long way in improving the timing predictability of the application, the task of doing so may become burdensome if the application is more complex with more interactions between chains. Therefore, the ability to synthesise priorities by specifying chains (or sub-chains) of critical callbacks and having an algorithm determine the best assignments has considerable value. For example, it would be productive to automatically synthesise a high priority to the actuation callback in the chain as well, as it is as critical to safe operation as obstacle detection. This would reduce the amount of time needed by developers to extract smaller end-to-end response times for time-sensitive chains.

3.3 Problem formulation

In this section, the problem is clearly and concisely formulated. As the problem is derived from the research question, we restate the goals of section 1.3:

1. To improve the predictability and real-time capabilities of the ROS scheduling model, enabling support for (1) the prioritisation of callback chains and (2) preemptive execution

2. To explore how the improved scheduling model can be leveraged to create more predictable ROS applications by synthesising callback priorities for applications that reduce the end-to-end response-time of time-sensitive callback chains

3.3.1 Callback chains

In this sub-section, we provide a formal model for ROS as a system of callbacks. We begin by defining a callback, and then publish-subscribe relations as a directed acyclic graph (DAG). Finally, we describe callback chains as paths within the DAG, along with some additional definitions.

- **Callback.** A callback is a two-tuple $c_i = (\nu_i, \theta_i)$ where $\nu_i \in \{sync, tmr, sub\}$ denotes the type of the callback, and $\theta_i \in \mathbb{R}$ the worst-case-execution-time (WCET) of the callback. There are three types of callbacks: Type *sync* identifies the callback as being triggered by a synchronisation event, type *tmr* by a periodic timer, and type *sub* by a message arriving on a subscribed topic.

- **Publish-subscribe relations.** Relationships between callbacks are modelled using a DAG $G = (C, E)$. The set $C = \{c_1, ..., c_m\}$ contains all callbacks within the graph. The set $E \subseteq C \times C$ captures publish-subscribe relations between callbacks, with the base of a directed edge being a publisher, and the end a subscriber. Several additional constraints govern how callbacks are connected in the graph. All callbacks without incident edges must be of type *tmr*, while all *sub* callbacks must have one or more incident edges, and all *sync* callbacks exactly two incident edges.

- **Callback chain.** A callback chain is a path in the publish-subscribe DAG. A path is an ordered sequence of callbacks, that always begins with a timer callback. Each path must contain at least one unique callback not found in any other path. Not all paths within a graph are necessarily callback chains. We focus on callback chains that have been identified by a user. These callback chains are noted as $\tau_i = \{c_p, ..., c_q\}$. A ROS application has a set of callback chains, which is defined as $\mathbb{T} = \{\tau_1, ..., \tau_n\}$. As mentioned in section 3.3.1, callback chains have an ascribed priority. The function $\mathbb{X} : \mathbb{T} \rightarrow \mathbb{Z}^+$ maps a callback chain to a non-negative priority value.

- **Additional definitions.** We also define some additional helper functions and constructs that come in use later in chapter 5. The function $reverse : \mathbb{T} \rightarrow \mathbb{T}$ returns

the given chain with all callbacks in the *reverse* order. Additionally, the function $\lambda : C \rightarrow \mathbb{Z}^+$ maps a callback c_i to a positive integer priority value, and the function $\Delta : \mathbb{T} \rightarrow \mathbb{Z}^+$ maps a chain to a period represented as a positive integer number of time units. Next, we define $\alpha = \{c_p, ..., c_q\}$ to be a path of callbacks (but not a chain) in the set of all possible paths A within the DAG, with a $reverse : A \rightarrow A$ function that returns the path in reverse order, and function $pred : \mathbb{T} \times C \rightarrow A$ which returns the ordered sequence of *predecessor* callbacks within a given chain up to, but not including, the given input callback.

3.3.2 Predictable ROS scheduling

Designing a more predictable scheduler requires creating an executor for RCLCPP that does not suffer from the drawbacks identified in section 2.3. It should therefore:

1. Place no implicit priority on callback type
2. Place no implicit priority on the order in which callbacks have registered
3. Consider multiple ready instances of a callback (as opposed to only handling one instance per polling period)
4. Not cause priority inversions (e.g. via irregular polling intervals).

It should additionally have the characteristics outlined in the research question:

Preemptive callback execution: Although non-preemptive schedulers have a place in real-time applications, the requirement for preemptive execution expands the number of applicable scheduling algorithms. It also would enable a scheduler to avoid priority inversions caused by forcing newly ready high priority callbacks to wait until a busy low priority callback completes.

Explicit priority assignment: All callbacks should be explicitly assigned priorities which the executor respects. This means that a priority value should be assignable to a callback by either the developer or a synthesis tool. The presence of explicit priority assignment will directly enable developers and tools to capitalise on the preemptive capabilities of the executor, and satisfies the accessibility requirement in our predictability definition.

3.3.3 Priority synthesis

A system that can synthesize priorities for a hypothetical preemptive executor with callback prioritisation will also need to account for characteristics of callback chains in ROS applications. Therefore, an ideal priority synthesizer:

1. Operates on a model of callback chains as described in section 3.3.1.
2. Accounts for shared callbacks between chains
3. Accounts for dependencies such as message synchronisation mechanisms.
4. Take advantage of the new scheduler capabilities (preemption, explicit priority assignment).

3.3.4 Assumptions

To bound the scope and feasibility of this work, the solutions are designed under the following assumptions (some are restated from section 1.5):

1. All callbacks are run on a single core within the system
2. All work exclusively concerns RCLCPP, the C++ client library
3. Only the publish-subscribe communication pattern is considered

4. Only periodic ROS applications are considered (all chains start with timers)
5. Callbacks under execution are assumed to *not* be members of callback-groups. These are ROS constructs which require members of the group to be executed sequentially through a mutually exclusive access scheme.

3.3.5 Experimental setup

1. All work is developed exclusively for ROS 2 Foxy Fitzroy (June 2020), on Ubuntu 18.04.5 LTS (Bionic Beaver), with Linux kernel 5.4.0-65-generic.
2. The DDS vendor used in testing is ADLINK Cyclone DDS

Chapter 4

Related work

The purpose of this chapter is two-fold. First, we revisit some of the state of the art attempts to improve the predictability of ROS that were first introduced in section 1.2, and discuss their work in more depth. Second, we visit existing research related to the problem of schedule synthesis, namely in relation to the classical job-shop scheduling problem. The chapter is organised as follows: section 4.1 addresses related-work that concerns improving the predictability of ROS applications, and section 4.2 schedule synthesis.

4.1 Predictability in ROS

When discussing work related to improving the predictability of ROS applications, we classify it as done in section 1.2: (1) Predictability via the creation of a separate, deterministic version of the ROS scheduling model, (2) predictability by subjecting ROS to special operating conditions such that a response-time analysis can be produced, and (3) predictability that involves optimising portions of ROS (e.g. the DDS configuration) and validing performance improvements through empirical testing.

4.1.1 Predictability through reimplementation

- **MicroROS**: Micro-ROS [18] extends ROS concepts to microcontrollers running real-time operating systems. It provides a two-layer executor solution, with one RCL (middleware API) layer executor enforcing a deterministic, static order scheduling policy with logical execution time (LET) semantics, and a second RCLCPP layer executor providing priority assignment and preemption for callback groups. However, the RCL layer executor requires developers provide a pre-defined callback order, and no known priority synthesis scheme is available. The RCL layer executor is also a departure from the core ROS project, as it modifies the ROS client library in addition to the language-specific interfaces.

4.1.2 Predictability through analysis

- **Response-Time Analysis of ROS 2 Processing Chains under Reservation-Based Scheduling**: Casini et al. [6] propose a response-time analysis for ROS 2 applications, for the purpose of predicting the end-to-end latency of time-critical processing chains. They consider systems under which ROS 2 executors are placed in reservation servers. Although the response-time analysis is the first of its kind for ROS, the solution simply identifies and models implicit priority hierarchies and inversions present within ROS. No changes or alternatives are proposed for an improved executor.

- **Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors**: Tang et al. [28] build upon the seminal work of [6] and propose an improved response-time analysis and prioritisation scheme. Their response-time

analysis accounts for cases of optimism and pessimism found in the the model by [6], and their prioritisation scheme describes how end-to-end response times can be somewhat adjusted by careful manipulation of the final callback of a chain. However, their priority scheme does not account for more complex ROS constructs such as message synchronisation mechanisms, depends entirely on gaming the inherent priority hierarchy present, and relies on model assumptions that they later find to not hold under empirical evaluation. For instance, their model assuemd that messages produced during a processing window are always processed at the next polling point. This did not turn out to be true under observation.

4.1.3 Predictability through optimisation

- **Towards a distributed and real-time framework for robots**: Exploring the suitability of ROS 2 for real-time applications, Gutiérrez et al. [12] measure the effectiveness of DDS QoS profiles and priority thread assignment to minimize round-trip latency tests between a microcontroller and PC. Their evaluation includes varying network traffic and system load, along with message sizes. Although valuable insights are gleaned with respect to the tuning and configuration of ROS 2 applications, the subject of timing predictability with respect to the ROS 2 executor is not discussed or addressed.

- **ROSCH**: Saito et al. [23] provide a real-time scheduling framework for ROS 1 applications. They extend ROS 1 with a real-time library housing an HLBS (heterogeneous laxity-based scheduling) scheduler. However, their scheduler appears unable to content with callback chains that share callbacks, and is not interoperable with ROS 2.

- **mROS**: Takase et al. [26] provide a custom platform for running ROS 1 nodes on low-power embedded devices as part of a larger ROS network, with an aim of enhancing the real-time performance of robot systems. They accomplish this by implementing a custom API on their TOPPERS/ASP real-time system kernel, and validate their inter-node communication stack with a case study. However, the work is for ROS 1, and also does not address real-time execution properties within ROS itself.

- **Modeling, implementation, and analysis of XRCE-DDS applications in distributed multi-processor real-time embedded systems**: Dehnavi et al. [8] address the lack of adequate performance analysis tools for DDS implementations made for low-powered embedded systems. Specifically, they target the XRCE-DDS standard, which is designed for use in resource-constrained environments. Their work involves deploying an implementation of the protocol on the CompSOC platform, a multi-processor SoC. They partition XRCE-DDS such that the agent and client interfaces leverage different hardware. XRCE-DDS agents, which interface with FastRTPS, are placed on an ARM Cortex A9 running Ubuntu Linux, and XRCE-DDS clients (which need only interface with the XRCE protocol) on the MPSoC side, with each client receiving its own RISC-V tile. They use a composition of Scenario Aware Dataflow Model (SADF) to represent both publishers and subscribers in a system. A markov chain is used to compute the long-term throughput of the system, for which probabilities have been derived via experimentation for the transition between different publish/subscribe outcome scenarios. They find that experimental implementations exhibit similiar long-term throughput results to those predicted by the model. Like much of the other work concerning DDS, the subject is not directly concerned with the predictability of RCLCPP executors, and therefore does not address how the peculiarities of the RCLCPP execution model affect throughput.

4.2 Scheduling theory and ROS

Synthesising schedules can be categorised as a scheduling problem [21]. This category of problems is both well known and well studied, with over fifty years of research from both industry and academia. In order to apply solutions to scheduling theory problems to ROS,

it is necessary that (1) a scheduling model be defined for ROS, and (2) the scheduling model be reduced to a known one from scheduling theory. The following subsections introduce the job shop problem as a candidate, and discuss it in relation to ROS.

4.2.1 The job shop problem (JSP)

Description. Job shop scheduling (JSP) is a classic and broadly applicable scheduling theory problem. Its most basic formulation consists of a set of p jobs $J = \{j_1, ..., j_p\}$ and q machines $M = \{m_1, ..., m_q\}$, with the aim of scheduling each job on every machine. An operation is used to denote a instance of a particular job being scheduled on a particular machine. Therefore, a job is comprised of a sequence of operations, where operations are drawn from the set $O = \{o_1, ..., o_r\}$. Machines processing an operation for a job may not be preempted or interrupted. This means that operations competing for the same resource (machine) may not be performed concurrently. Finally, jobs are assumed to have an independent flow pattern from one another. This means that jobs may perform their operations in any order, as opposed to the same order (called a *flow shop*).

Goal. Although various measures of performance may be defined for the JSP, by far the most popular goal is to find an optimal ordering of all operations on all machines such that a minimal *makespan* is acheived (the difference in time between the end of the final operation of the final job, and the start of the initial operation of the initial job). As machines may process jobs in any order, the job shop problem has been proven to be NP-complete (see [10] for a proof).

Complexity. The complexity of the JSP has born out a slew of solutions. Pseudo-polynomial and polynomial time solutions exist for specific cases of the JSP (notably Akers (1956) for the $2 \times M$ case, and Jackson (1956) for the $N \times 2$) [19]. However approximation techniques are the most popular contemporary approach, given that optimal search procedures and mathematical formulations are generally unable to quickly derive solutions.

Representation. A practical representation of the JSP is as a disjunctive graph. This was first proposed by Roy, Bernard, and Sussman (1964)[22]. The graph is defined as $G = (O, A \cup E)$, where O is the set of operations (vertices) extended with two extra vertices $\{o_0, o_{n+1}\}$. These represent a source and sink vertex, each with a processing time of 0. The set A consists of conjunctive arcs reflecting precedence constraints between operations within jobs. The set E contains undirected disjunctive edges, connecting mutually unordered tasks that require the same resource (i.e machine). Every arc is given a weight equivalent to p_i, the processing time associated with the base of the arc. An example of a disjunctive graph is given in figure 4.1.

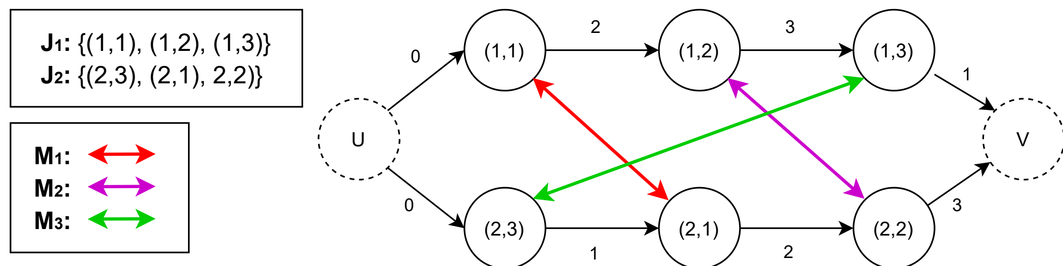


Figure 4.1: **A disjunctive graph representing an instance of the JSP with two jobs and three machines. Operations form nodes, with the notation (i, j). i refers to the job, and j to the machine on which the job is to be executed. Disjunctive arrows indicate all jobs competing for the same machine**

4.2.2 Example

The disjunctive graph model of the JSP may be used to derive the *makespan* for a particular set of jobs and machines. To obtain the makespan, the graph first undergoes an *acyclic*

orientation. This ensures that the execution order of all tasks competing for the same resource is resolved. The chosen orientation **must be acyclic** to be feasible. Many possible orientations are possible. An acyclic orientation of figure 4.1 demonstrated in figure 4.2.

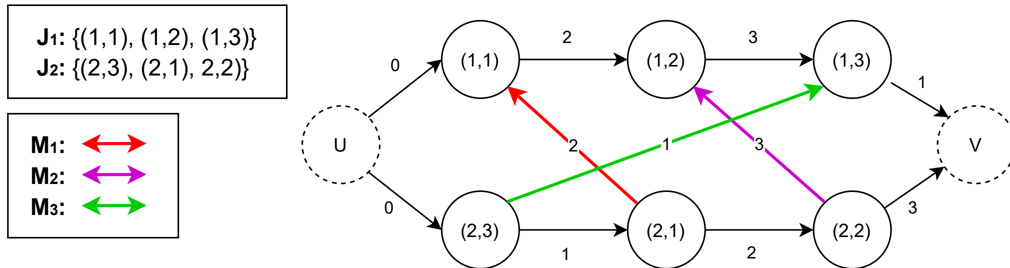
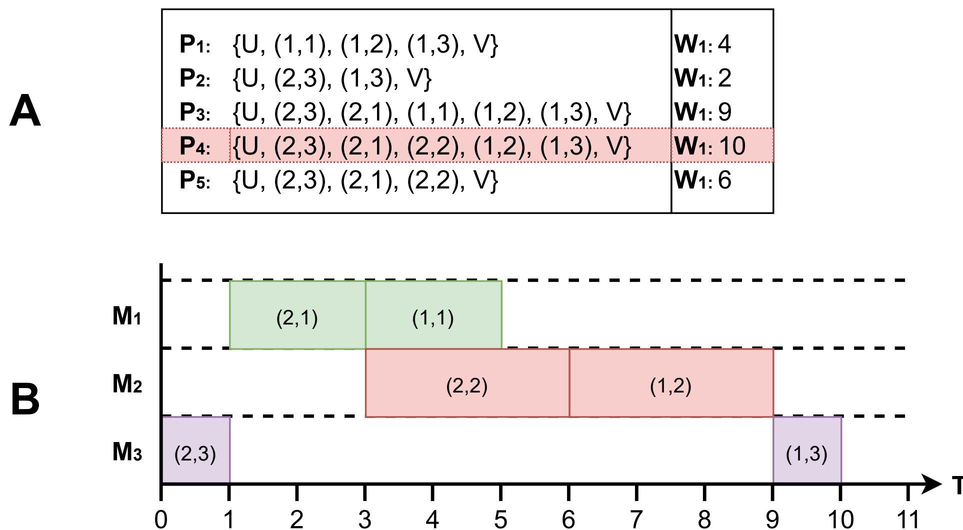


Figure 4.2: **A disjunctive graph representing an instance of the JSP with two jobs and three machines. An acyclic orientation has been performed on the graph, and all disjunctive edges transformed into conjunctive edges. The graph has no cycles.**

For any orientation selected, the makespan of the graph is computed by *summing the weights* (processing times) of the edges along the *longest* path within the graph. We demonstrate how the sum of the weights corresponds with the true schedule in the form of a GANTT chart. This can be seen in figure 4.3.



P1:	{U, (1,1), (1,2), (1,3), V}	W1: 4
P2:	{U, (2,3), (1,3), V}	W1: 2
P3:	{U, (2,3), (2,1), (1,1), (1,2), (1,3), V}	W1: 9
P4:	{U, (2,3), (2,1), (2,2), (1,2), (1,3), V}	W1: 10
P5:	{U, (2,3), (2,1), (2,2), V}	W1: 6

Figure 4.3: **(A) tabulates all paths within figure 4.2. (B) contains a GANTT chart which visualises the resulting schedule of the given graph orientation**

4.2.3 JSP and ROS scheduling

Similarities. In many ways, synthesizing a schedule for ROS has many parallels with the JSP. Jobs can be represented with callback chains, and operations with callbacks. Callbacks also have precedence constraints identical to that of operations. Machines exist in both models, as ROS may distribute work across executors located on physically separate hosts. Although some versions of the JSP assumes each job runs on every machine, more general models permit machine repetition and absence, lending the necessary flexibility to represent realistic callback chains. Finally, properties of the machine, such as non-preemption, do fit

the existing executor model in ROS. However, this is where the similarities end.

Differences. Unlike the JSP, synthesizing a schedule for a ROS application is subject to far more constraints and conditions. For one, ROS applications require certain callbacks be run on certain machines, reducing the search space. Next, callback-chains are periodic, meaning that they release work at defined intervals as opposed to a single large batch. This can be accounted for, however, by considering all operations up to the hyperperiod of the callback-chains, with additional release-time constraints on operations. Finally, and perhaps most importantly, the scheduling model present in ROS is non-preemptive, with an unusual implicit priority scheme and scheduling behavior (see section 2.3). This behavior makes the JSP not directly applicable to ROS as it exists. Furthermore, when we consider the problem formulation in section 3.3, the proposed improvements include a *preemptive* scheduler, which is normally assumed to not be possible in traditional JSP formulations.

Verdict. A final difficulty in applying the JSP to ROS would be the difference in goal or problem being solved. The JSP attempts to minimize some performance metric globally, typically the makespan of a set of jobs. This isn't necessarily the same for ROS. In our work, we are **also** interested in enabling developers to minimize the end-to-end latency of callback chains or sub-chains, even at the expense of global system performance. This difference in objectives makes JSP solutions harder to apply to our problem of synthesizing ROS schedules.

Chapter 5

Solution

This section discusses the solutions developed for both a priority-aware, preemptive executor model, and a schedule synthesis scheme.

5.1 Preemptive, priority-aware executor

The development of an executor that can meet the requirements outlined in section 3.3 yields more than just one possible implementation. In this section, we begin by outlining how we approach designing a new executor. Next we explain how priorities are assigned to callbacks in ROS. Finally, we present two executor implementations that take advantage of priorities and preemption, along with their respective advantages and disadvantages.

5.1.1 Executor design considerations

In section 3.3.2, a number of requirements were outlined for an improved executor design. For convenience, we reiterate them here. Then, we discuss how to approach creating an executor.

Requirements

A design for a more predictable scheduler requires creating an executor for RCLCPP that:

1. Places no implicit priority on the type of callback
2. Places no implicit priority on the order in which callbacks have registered
3. Consider multiple ready instances of a callback (as opposed to only handling one instance per polling period)
4. Does not cause priority inversions (e.g. via irregular polling intervals)

Such an executor should also support *preemptive callback execution*, and allow callbacks to be assigned an *explicit priority value*.

Approach

Many of the requirements for an improved executor concern priority. More specifically, eliminating implicit sources of priority and enabling explicit priority assignment that an executor will respect. Therefore, a solution should begin by ensuring there exists a foundation for prioritising callbacks in ROS. Because callbacks are tied to higher level constructs such as timers, subscriptions, and message-filters, these should first be extended or modified in order to accomodate a priority value.

Next, an executor design will need to eliminate sources of priority inversions, and enable callbacks to be executed preemptively. To remove priority inversions present in the standard executor, two modifications will need to be made:

1. The scheduler of the executor will need to consider *all* ready callbacks when choosing what to execute next (as opposed to simply selecting the next ready callback by type, or by registration order). This can be done by maintaining a hierarchy of ready callbacks ranked by priority, and always selecting the highest priority callback at each invocation of the scheduler.

2. The scheduler will need to respond to newly arrived messages while executing callbacks. This means it cannot wait between irregularly sized polling points in order to process them. This can be accomplished by having a timer regularly invoke the scheduler to check for newly arrived work, or having the scheduler block as a higher priority thread when no work is present, but wake up (and preempt) any busy work thread when messages arrive.

Enabling preemptive callback execution is therefore necessary for removing priority inversions. The executor should be able to suspend the execution of one callback and execute another higher priority callback if it becomes ready during the execution of the former. Preemption should also be harnessed by the scheduler of the executor itself in order to let the scheduler respond rapidly to new messages.

5.1.2 Assigning priorities to callbacks

In order to accomodate the need for priorities in ROS, several key RCLCPP classes and interfaces are modified and extended to support a priority value assignment. The *timer* and *subscription* RCLCPP classes are first given an integer priority value field. This priority value is then associated with the callback that is attached to the timer (which is called periodically) and subscription (which is called when a message is ready on the subscribed topic). Next, the classes themselves have their constructors (initialiser functions) extended to incorporate an optional priority value assignment (with a default in case none is provided). As the scope of the work is limited to the publish-subscribe communication patterns in ROS, the *service* classes are left unchanged.

5.1.3 Thread dispatch model

The first executor implementation that is considered for solving the problem of priority inversions and non-preemptive execution is named the *thread dispatch* model, and is a multi-threaded design. The thread dispatch model executes callbacks in dedicated, one-off work threads (see figure 5.1, B). It has a scheduler which is set to run at the highest priority in the system, and blocks while waiting for messages to arrive from the middleware layer (which interacts with the DDS backend). When any messages do arrive, the scheduler creates a new thread for each ready callback with a priority value proportional [1] to the one assigned to the callback (bottom block of figure 5.1, A). The thread is then detached, and the scheduler blocks to wait for messages to arrive. Meanwhile, the dispatched thread runs the callback to completion and is then destroyed.

[1] The term proportional is used to denote the mapping from the priority range provided to the scheduler to that realised by the kernel. Real-time linux scheduling supports a priority range from 0 to 99. Executors with N different priorities always place the scheduler at priority 99, and map the N priorities from $[(99 - N), 99)$

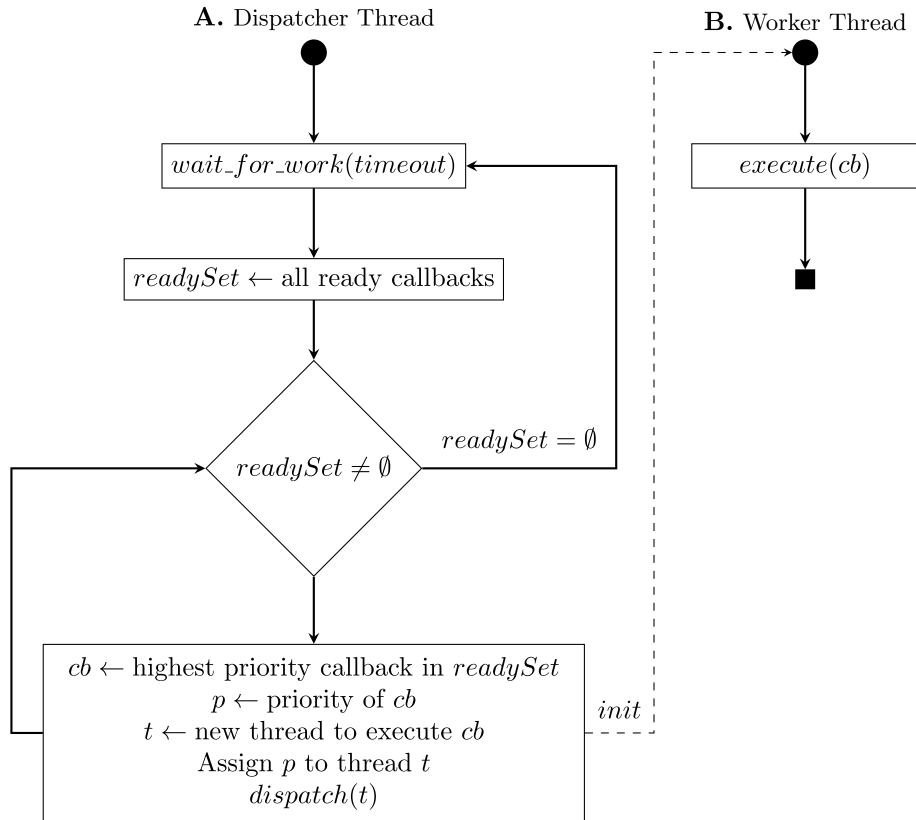


Figure 5.1: **Thread-dispatch executor model**

The complete thread dispatch executor model is visualised in figure 5.1. Here, we see how the dispatcher thread launches work threads, which then terminate (indicated with a black square) after completion. This design enables preemption by relying on properties of the Linux real-time `SCHED_FIFO` scheduling policy. This policy runs all ready threads of the highest priority in a first-in, first-out fashion, before moving to those of lower priority [15]. As the priority value hierarchy of callbacks is mapped to that of dispatched threads, callbacks with higher priority values can naturally preempt those with lower ones. Although the scheduler may ultimately preempt everything, it only activates when new work is present, and therefore allows callback threads to run in the quiet time. This fulfills the preemptive requirement and respects explicit callback priority assignment.

Advantages

The thread-dispatch model launches a dedicated thread for each callback it executes. Because of this, the design is expected to scale well on multicore systems. This is because the presence of multiple cores allows work threads to be easily distributed across them, enabling several higher priority threads to be executed concurrently and automatically preempt lower priority ones.

Disadvantages

A cost is incurred when the scheduler launches a thread, as the creation requires action by the operating system kernel. This cost may be compounded when multiple callbacks are scheduled in succession, leading to a deterioration in performance.

5.1.4 Producer-consumer model

The second executor implementation that provides both preemptive callback execution and prioritisation is called the *producer-consumer model*. This implementation is based on the

classic producer-consumer synchronisation problem, with some minor difference we revisit later. In this model, the scheduler plays the role of a producer while running in a dedicated thread at the highest priority in the system. When there is no new work to process, the producer blocks and waits for new work to arrive. When work arrives, it wakes up and sorts the ready callbacks into a series of one or more consumer queues, before returning to sleep. Each consumer queue is attended to by a dedicated consumer thread, which is awoken when work is placed in their queue (see *notify* in figure 5.2). After consuming all the work in its queue (i.e. executing all the callbacks), a consumer thread returns to sleep once again. The complete operation of both the producer and consumer threads is illustrated in figure 5.2. In this figure, a dashed line is used to represent thread interactions. The difference between the classic producer-consumer problem and this implementation is that the consumer does not wake up the producer when it finds there is no more work to do. Here, only the producer wakes the consumer [25] [27].

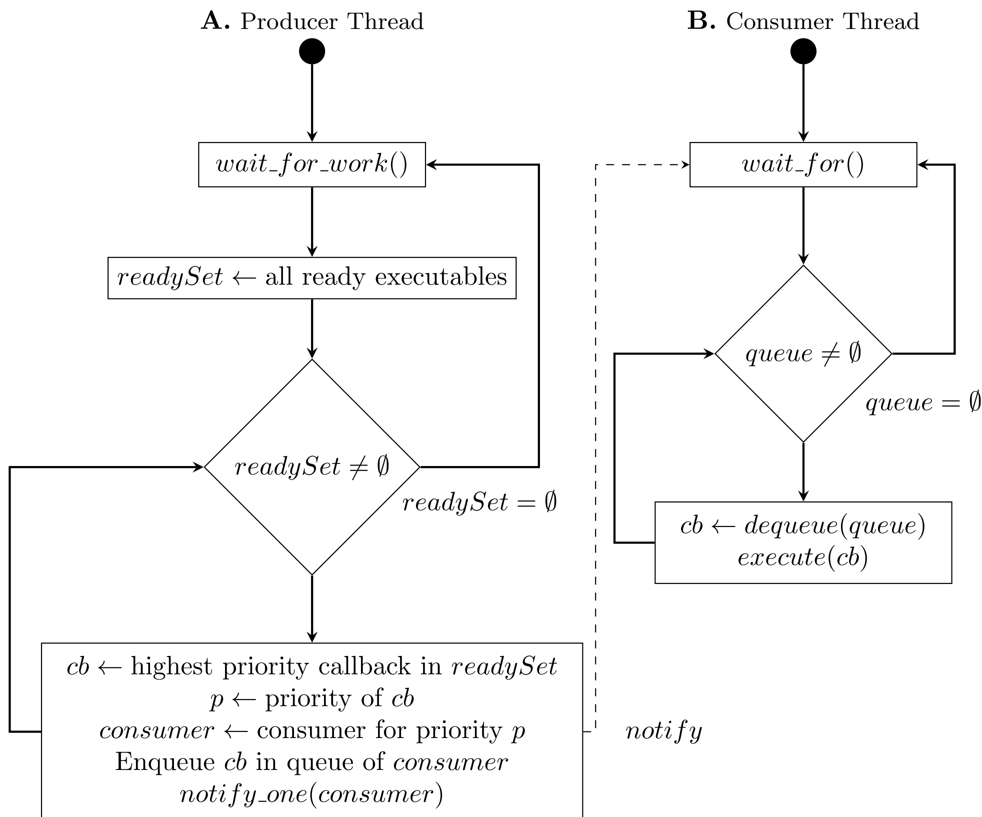


Figure 5.2: **Producer-consumer executor model**

This design enables preemption to occur by assigning each consumer thread a unique priority. For a ROS application with N unique priorities, N consumer-queue pairs are created. The producer selects which queue to place a callback in by indexing the ordered set of consumers by the priority value of the callback. Each consumer thread is running at a priority proportional to that of the index it is located at. Through the properties of the Linux real-time `SCHED_FIFO` scheduling policy, this enables higher-priority consumer threads to preempt lower-priority ones, thus fulfilling the requirement for both callback preemption and priority assignment.

Advantages

As the number of threads are fixed, and no new threads created over the lifetime of the ROS application, the producer-consumer model enjoys minimal overhead from the kernel, and thus more stable and consistent behavior than the thread dispatch model.

Disadvantages

Unlike the thread-dispatch model, the producer consumer model is less suited for spreading work across multiple cores. This is because it pairs a single consumer thread to a queue of a unique priority. Therefore, the presence of more cores simply distributes dedicated consumer threads across them. The effect is that multiple high priority callbacks may be ready in a high priority queue, but they will be executed sequentially by the dedicated consumer thread. In the meantime, lower-priority callbacks will be executed on other cores by their consumer threads, effectively causing a priority inversion. This problem can be mitigated by enabling more than one consumer thread to consume from a queue. Ideally, each queue associated with a priority level would be assigned as many consumers as there are cores.

5.2 Callback priority synthesis

The new executor designs introduced in section 5.1 effectively implement a *preemptive, fixed-priority* scheduling policy. When callbacks of the same priority are ready to be scheduled, the schedulers execute them in a FIFO order. The ability to explicitly assign priorities and have that respected by the executor already lends a degree of control to the developer unparalleled by that of the standard RCLCPP single-threaded executor. However, this work also aims to explore how a priority synthesis algorithm can leverage these new capabilities to produce more time-predictable ROS applications. The following subsections address this question by first introducing a model used for priority synthesis, and then the algorithm itself.

5.2.1 Synthesis model

The concept of callback chains is once again leveraged to represent ROS applications. A formal definition of callback chains was provided in section 3.3.1. In summary, a ROS application has a set of callback chains, which is defined as $\mathbb{T} = \{\tau_1, ..., \tau_n\}$. Each callback chain is a nonzero ordered set of callbacks $\tau_i = \{c_p, ..., c_q\}$. A callback is a two-tuple $c_i = (\nu_i, \theta_i)$ where $\nu_i \in \{sync, tmr, sub\}$ denotes the type of the callback, and $\theta_i \in \mathbb{R}$ the worst-case-execution-time (WCET) of the callback. The function $\mathbb{X} : \mathbb{T} \rightarrow \mathbb{Z}^+$ maps a callback chain to a non-negative priority value, which is assumed to be given.

Directed acyclic graph

Just as was mentioned in section 3.3.1, a directed acyclic graph (DAG) may be used to represent the synthesis model. A DAG is defined as $G = (C, E)$. The set $C = \{c_1, ..., c_m\}$ contains all callbacks within the graph. The set $E \subseteq C \times C$ captures publish-subscribe relations between callbacks, with the base of a directed edge being a publisher, and the end a subscriber. An example of such a DAG is provided in figure 5.3, where timer callbacks are indicated with a bold circle, standard callbacks with a thin circle and synchronisation callbacks with a yellow diamond. Edges are defned as $e_i \in E$.

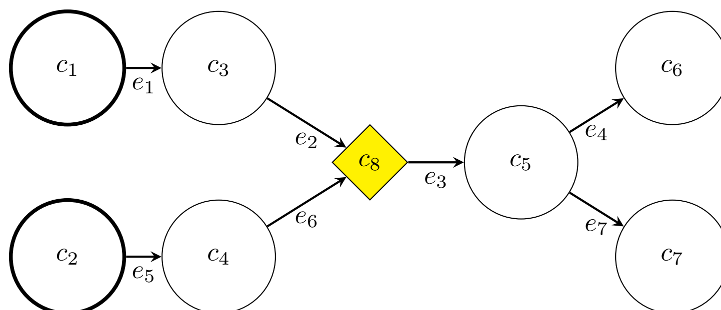


Figure 5.3: **Directed acyclic graph of a ROS 2 application with two callback chains synchronising on callback c_8 and sharing callback c_5**

5.2.2 Synthesis algorithm

The research question outlines that priority synthesis should aim to produce more time predictable ROS applications, partly by reducing the end-to-end response-time of time-sensitive callback chains. Examples of time-sensitive callback chains might be the sequence of callbacks between a sensor being read and a brake applied, or an input triggering an event that requires an urgent message be sent.

Intuitive description

Consider the DAG represented in figure 5.3, which is composed of two chains:

$$\tau_0 = \{c_1, c_3, c_8, c_5, c_6\}$$

$$\tau_1 = \{c_2, c_4, c_8, c_5, c_7\}$$

A developer has a *goal* of reducing the end-to-end response-time of both chains, while prioritising one chain over another. For example, the developer may want to prioritise τ_0 over τ_1. The simplest approach is to assign callbacks within τ_0 a priority greater than τ_1. However, two considerations arise:

- What priority should be given to callbacks *shared* between τ_0 and τ_1, such as c_5?
- What priority should be given to *synchronisation* callbacks, such as c_8?

Any decisions taken should reduce the end-to-end response-time of chains by order of priority. The following discussion points outline how shared and synchronisation callback are addressed:

1. **Shared callbacks**: Callbacks that are *shared* should always adopt a priority equal to that of the *highest priority* chain that uses it. This form of priority inheritance enables high priority chains to enjoy an advantage over lower-priority chains, but it also allows lower-priority chains that contain the shared callback to run it at a higher priority.
2. **Synchronisation callbacks**: A synchronisation callback requires a binary input in order to produce an output. This creates a dependency between the two input chains. Therefore, it is necessary that all paths leading up to the synchronisation mechanism be adjusted such that they run at the priority of the higher branch. To not do so would allow a lower priority input to delay the output of the synchronisation point (i.e. a priority inversion). As an example, we may consider the chain in figure 5.4. Here, we see that chain τ_1 has the highest priority. If all three chains are activated at the same time by a timer, τ_1 will run c_1 first. However, it must then wait for the input from τ_2 to arrive. Since τ_2 is of lower priority than τ_3, chain τ_1 must wait for τ_3 to complete first. This causes a priority inversion.

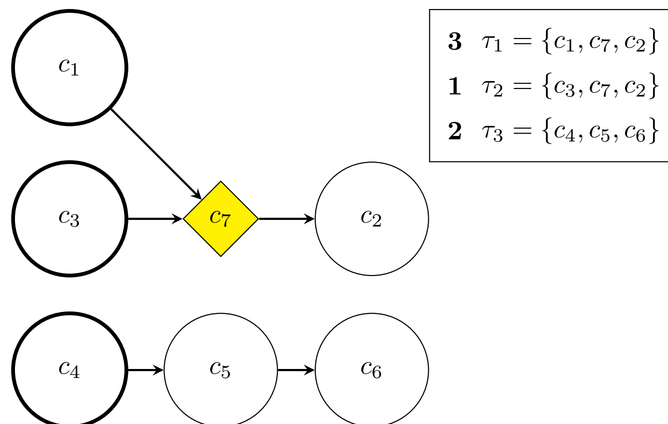


Figure 5.4: **A directed acyclic graph (DAG) with three chains and one synchronisation. The legend indicates the priority of the chains in bold**

Therefore, synthesising priorities should be composed of three steps:

1. **Initial priority assignment**: All callback chains are assigned a priority value by the developer. This is assumed to be given with the chains of the ROS application.
2. **Priority resolution**: Callbacks automatically adopt the highest priority value for all chains of which it is a member
3. **Priority propagation**: Synchronisation mechanism callbacks have their highest priority automatically propagated backwards down both of their input chains

Algorithm

Algorithm 1 Priority synthesis

```
1: procedure PrioritySynthesis(𝕋, 𝕏)    ▷ Takes as input a
   set of chains 𝕋, and function 𝕏 : 𝕋 → ℤ⁺ mapping chains to a priority value. Returns
   a priority mapping for callbacks within the chains λ : C → ℤ⁺

2:     ▷ Set initial callback priorities
3:     for each callback chain τ_i ∈ 𝕋 do
4:         p ← 𝕏(τ_i)
5:         for each callback c_j ∈ τ_i do
6:             λ(c_j) ← max{λ(c_j), p}
7:         end for
8:     end for

9:     ▷ Propagate callback priorities until stable
10:    repeat ← TRUE
11:    while repeat = TRUE do
12:        repeat ← FALSE
13:        for each callback chain τ_i ∈ 𝕋 do
14:            p ← 𝕏(τ_i)
15:            ▷ Retrieve property of callback j in τ_i
16:            for each callback (ν_j, θ_j) ∈ reverse(τ_i) do
17:                if ν_j = sync then
18:                    p ← max{p, λ((ν_j, θ_j))}
19:                end if
20:                if p > λ((ν_j, θ_j)) then
21:                    repeat ← TRUE
22:                end if
23:                λ((ν_j, θ_j)) ← max{p, λ(c(ν_j, θ_j)}
24:            end for
25:        end for
26:    end while

27: end procedure
```

5.2.3 Example

To better understand how the priority synthesis algorithm works, various steps of the algorithm are laid out in figure 5.5 for an example ROS 2 application with three chains.

$$
\begin{array}{ll}
\mathbb{X}(\tau_1) = 0 & \tau_1 = \{c_1, c_{11}, c_2, c_3, c_7\} \\
\mathbb{X}(\tau_2) = 1 & \tau_2 = \{c_4, c_{11}, c_5, c_{12}, c_6, c_7\} \\
\mathbb{X}(\tau_3) = 2 & \tau_3 = \{c_8, c_9, c_{12}, c_{10}, c_7\}
\end{array}
$$

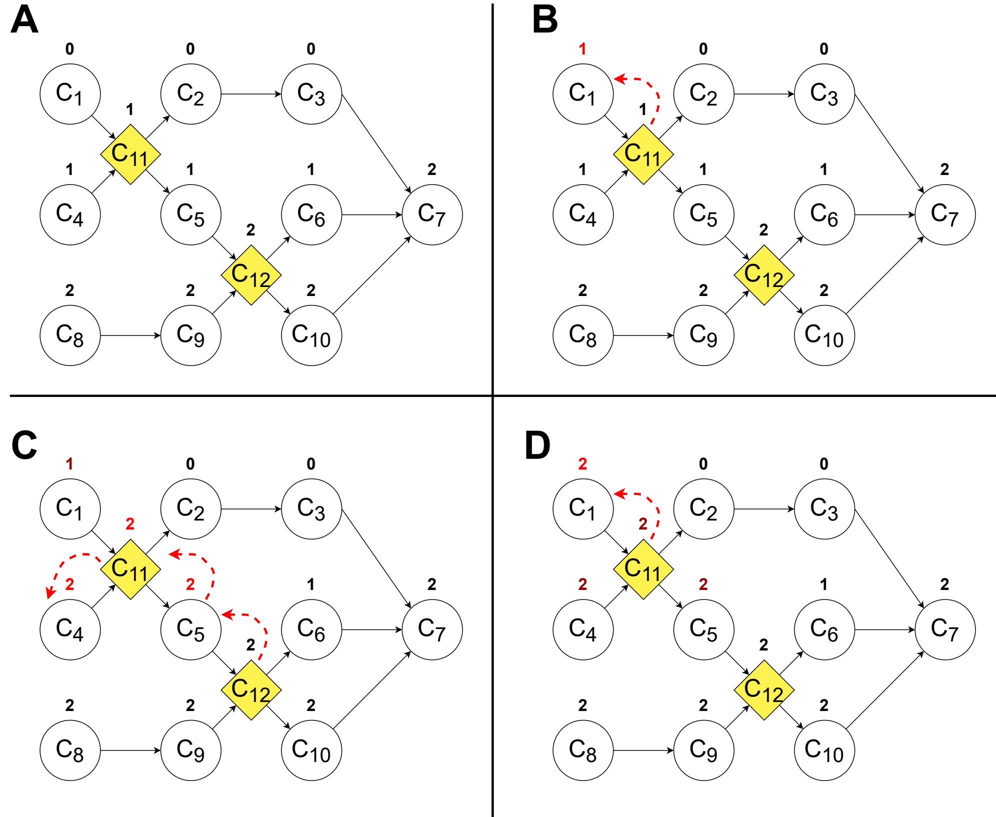


Figure 5.5: **An example of the priority synthesis algorithm applied to a ROS 2 application with three chains of differing priority**

We discuss each step of the algorithm below:

(a) The first phase of the algorithm iterates down all chains, and sets the priority of the callback to the maximum between the priority of the chain, and that currently set on the callback. We can see that the callback c_7 is shared by all chains, and thus settles with the priority of τ_3 (the highest priority chain). Likewise, the synchronisation callbacks c_{11} and c_{12} inherit the larger priority value of their respective inputs.

(b) The second phase of the algorithm sweeps through each chain from *back to front*, and continues to do so until no callbacks in any chain undergoes a change in priority. The algorithm begins by setting a *minimum* priority value to that of the chain it is sweeping through. Then, it goes backwards through the callbacks in the chain, and updates the priority value of each callback if it is less than the minimum. At each synchronisation point, the minimum value is updated to that of the synchronisation node if the synchronisation node has a higher priority than the minimum. We can see that the first chain, τ_1 inherits the priority value 1 from τ_2, because the synchronisation node c_{11} is shared. That value is then propagated back to c_1.

(c) The third phase of the algorithm sweeps through τ_2, and propagates the priority value of τ_3 backwards from the synchronisation callback c_{12}. The algorithm then does the same for chain τ_3. However, nothing needs to be propagated. Nevertheless, there *have*

been some changes made to the priority of callbacks in the other chains, so the check loop must continue

(d) Finally, we see that the second sweep of τ_1 propagates the new priority value 2 backwards into c_1. This marks the final changes made by the algorithm. It sweeps through τ_2 and τ_3, but no changes are made. One more pass is made after this (due to the change in τ_1), and then the algorithm terminates.

The final result, shown in quadrant D of figure 5.5, demonstrates how all the dependent chains for the highest priority chain are also prioritised from the synchronisation points.

5.3 Feasibility test based on individual chains

In addition to the synthesis algorithm, a feasibility test can be created for ROS 2 applications using the model developed in section 3.3. The feasibility test can determine if a particular application is schedulable by traversing callback chains and computing a new, *actual workload*, for each chain based on the dependencies of the chain (i.e. other chains to which it is synchronised). If any chains are present in the application which contain a workload larger than their deadline, then the application is deemed unfeasible.

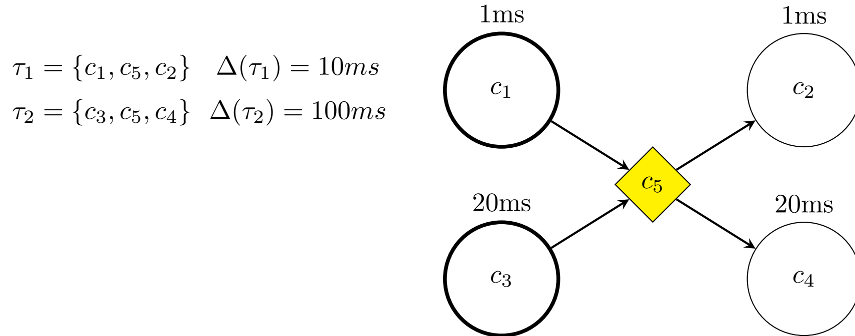


Figure 5.6: **Example of an unfeasible application. Callbacks are indicated with circles, and synchronisation points with yellow diamonds. Labels above callbacks denote the computational time needed to process them**

Consider the graph in figure 5.6. Suppose a developer has just inserted the synchronisation point c_5 between chain τ_1 and τ_2. If we compute the workloads of τ_1 and τ_2 naively by simply summing the required computational time along the chain, we find that $\theta_1 = 1ms + 1ms = 2ms$, and $\theta_2 = 20ms + 20ms = 40ms$. When compared to their deadlines, the workload appears reasonable. τ_1 may complete $2ms$ of work within $10ms$, and τ_2 can complete $40ms$ of work in $100ms$. However, the synchronisation point forces τ_1 to wait, *at minimum*, 20ms, before it proceeds to c_2. This is because it has to wait for τ_2 to complete c_3. This changes the workload of τ_1 to $\theta_1 = 22ms$, which now makes the chain impossible to complete in $10ms$, and hence the system is unfeasible.

5.3.1 Feasibility test algorithm

We define the feasibility algorithm as follows:

Algorithm 2 Feasibility test based on individual chains

```
1: procedure ACTUALCTIME(τ_i, α_j)   ▷ Returns the actual computational time required
   by the chain τ_i given path α_j
2:     sum ← 0                                              ▷ Sum of actual CPU time
3:     if no callbacks in path α_j then return sum
4:     end if
5:     c_k ← next callback in α_j
6:     if ν_k = sync then                                   ▷ ν_k is the type of c_k
7:         τ_m ← Other chain input to synchronisation callback c_k
8:         sum ← sum + ActualCTime(τ_m, reverse(pred(τ_m, c_k)))
9:     else
10:        sum ← sum + θ_k                         ▷ θ_k is the WCET of callback c_k
11:    end if
   return sum + ActualCTime(τ_i, pred(τ_i, c_k))
12: end procedure
1: procedure FEASIBLE(𝕋)                           ▷ Returns TRUE if 𝕋 is feasible
2:     for each callback chain τ_i ∈ 𝕋 do
3:         d ← Δ(τ_i)                                       ▷ Deadline of chain τ_i
4:         R ← ActualCTime(τ_i, {c_j | c_j ∈ τ_i})
5:         if R > d then return FALSE
6:         end if
7:     end for
   return TRUE
8: end procedure
```

5.4 Schedulability test for harmonic chains

We propose a schedulability test for ROS 2 applications that are running on top of the new executor implementations. The test depends on the following schedulability conditions:

1. The scheduling policy is fixed-priority preemptive scheduling
2. For every chain in the system, its deadline is equal to its period
3. Every chain that is connected by a synchronisation mechanism has the *same period*

In the following subsection, we describe the worst-case response-time analysis computation for tasks under fixed-priority preemptive scheduling. Next, we show how we can convert a ROS 2 application graph (under the assumptions above) into a comparable task-set to which the analysis can be applied.

5.4.1 Worst-case response-time analysis

The schedulability test is based on the worst-case response-time analysis for **fixed-priority preemptive scheduling** task-sets, proposed by Audsley et al. [2]. In their schedulability analysis, they consider the worst-case response-time analysis for a task to occur at the *critical instant*. This is the instant at which the task under consideration is released along with all higher priority tasks. For a given task τ_i, we say that the worst-case response time (R_i) is defined by the following fixed point iteration [2]:

$$R_i^0 = \theta_i + \sum_{1 \leq j \leq i} \theta_j \tag{5.1}$$

$$R_i^k = \theta_i + \sum_{1 \leq j \leq i} \left\lceil \frac{R_i^{k-1}}{T_j} \right\rceil \theta_j \tag{5.2}$$

In equation 5.1 and 5.2, θ_i refers to the worst-case execution time of task τ_i, and T_i to the period of task τ_i. The equations assume that tasks are *prioritised* in descending order

by their subscription (e.g. $\mathbb{X}(\tau_0) > \mathbb{X}(\tau_1)$), so that τ_0 is always the highest priority task. The worst-case response time computation stops when either (1) the same value is computed across two consecutive iterations or (2) the deadline of task τ_i is exceeded.

5.4.2 Converting ROS 2 application graphs into FPPS task-sets

In order to map a ROS 2 application to task-sets under fixed-priority preemptive scheduling, we need to disentangle chains linked through synchronisation devices. This can be done using the following steps:

1. Apply the priority-synthesis algorithm to the application graph
2. For each chain, a workload value is computed by summing the workload of each individual callback within the system **that has a priority value equal to that of the chain itself**.
3. Apply the worst-case response-time analysis described above

We illustrate how this works with a supporting example.

5.4.3 Example

Consider the application in figure 5.7, A. The graph contains three chains: τ_1, τ_2, and τ_3. Observe that in conformance with the schedulability conditions, the synchronised tasks τ_2 and τ_3 have the same period.

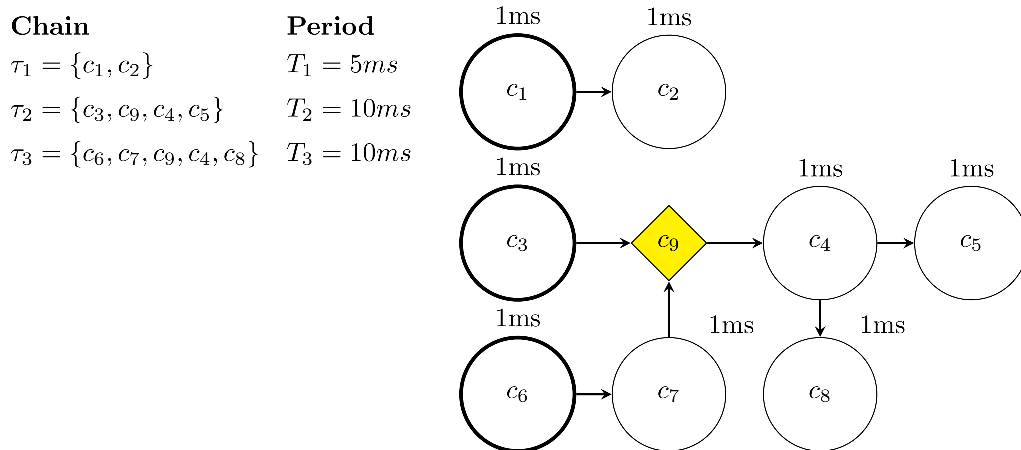


Figure 5.7: **Example ROS 2 application with three chains and harmonic chain periods. The WCET of each callback is labelled above them in millisecond units**

We begin converting chains into tasks. First, we apply the synthesis algorithm, which propagates the priorites of each chain throughout the graph. The resulting callback priorities may be seen in figure 5.8.

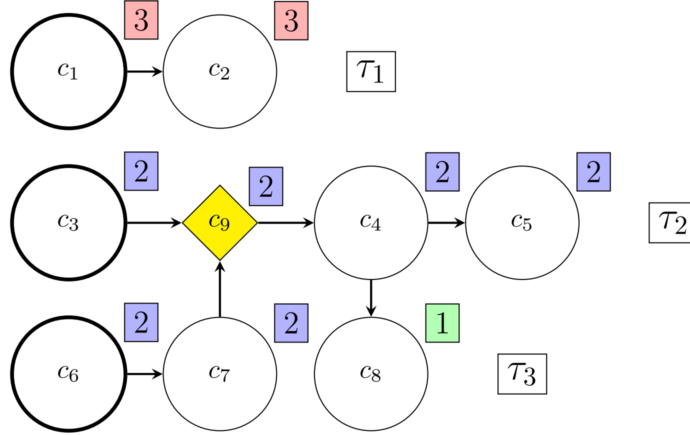


Figure 5.8: **Example ROS 2 application with priority synthesis applied. Each callback takes 1ms to complete (see figure 5.7)**

Next, we convert the chains one by one into a set of tasks:

1. τ_1: The chain τ_1 has *no dependencies*. Therefore, this chain is directly analagous to a task with period $T_1 = 10ms$, and $\theta_1 = 2ms$. We compute θ_1 trivially by seeing that two callbacks compose chain τ_1.

2. τ_2: The chain τ_2 has a synchronisation point at node c_9. We therefore see that in order to continue, it must wait for the work completed in callbacks c_6, and c_7. We add that to the workload of τ_2, along with the callbacks that remain in the chain. Therefore, we arrive at $\theta_2 = 5$.

3. τ_3: The chain τ_3 is *lower priority* than τ_2. Therefore, we see that after synchronisation point c_9, it has to wait for the remaining callbacks of τ_2 to complete before it can continue. In terms of a schedulable entity with a distinct priority, only c_8 counts as the workload of task τ_3 (so $\theta_3 = 1$).

   This works precisely because the chains τ_2 and τ_3 share the *exact same release times*. This ensures that the remaining workload of τ_3 always appears exactly after that of τ_2 due to the fact that (1) both are released at once and (2) callbacks required for τ_2 always run at a higher priority.

The shortcut for computing the workload values of each chain is simply to iterate down the list of callbacks that compose the chain, and sum up the workload of each callback that has the original priority of that chain. Callbacks that have a higher priority (this is the only possibility) technically do not belong to the chain, as we are attempting to group workload into tasks with different priorities. The new derived taskset is defined in table 5.1.

Table 5.1: **Task set for the ROS application in figure 5.7**

Task	**T**	**C**
τ_1	5ms	2ms
τ_2	10ms	5ms
τ_3	10ms	1ms

The critical instant for the tasks is shown in figure 5.9. the task set is demonstratably analogous to a fixed-priority preemptive schedule.

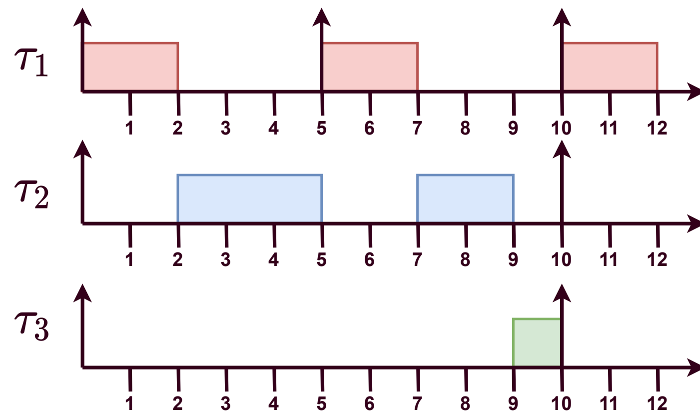


Figure 5.9: **Callback execution timeline for the chains of graph 5.7**

The schedulability test of Audsley et al. [2] can be applied to the task set to come to a similar conclusion (the task-set is schedulable under fixed-priority preemptive scheduling).

Chapter 6

Evaluation Framework

Evaluating the new executors and priority synthesis algorithm developed for ROS immediately poses a few challenges. Firstly, no large scale benchmarking framework exists for evaluating ROS executors. Secondly, the priority synthesis algorithm developed expects ROS applications to have certain properties such as message synchronisation mechanisms. The solution is therefore to develop a testing framework for ROS, with the aim of generating tunable, traceable, and realistic ROS applications that operate within the given assumptions established in section 3.3.4. This chapter provides an overview of the testing framework and all related considerations.

6.1 Overview

The testing framework is composed of a set of interacting programs and supporting packages designed to automate the end-to-end process of:

1. Specifying properties and constraints for a ROS application.
2. Automatically generating said application, with some randomised features.
3. Automatically running and analysing said application.

It is written in the Go programming language, and makes extensive use of Go templates to easily generate source files, makefiles, and diagrams. A visualisation of the programs and their dependent packages are shown in figure 6.1.

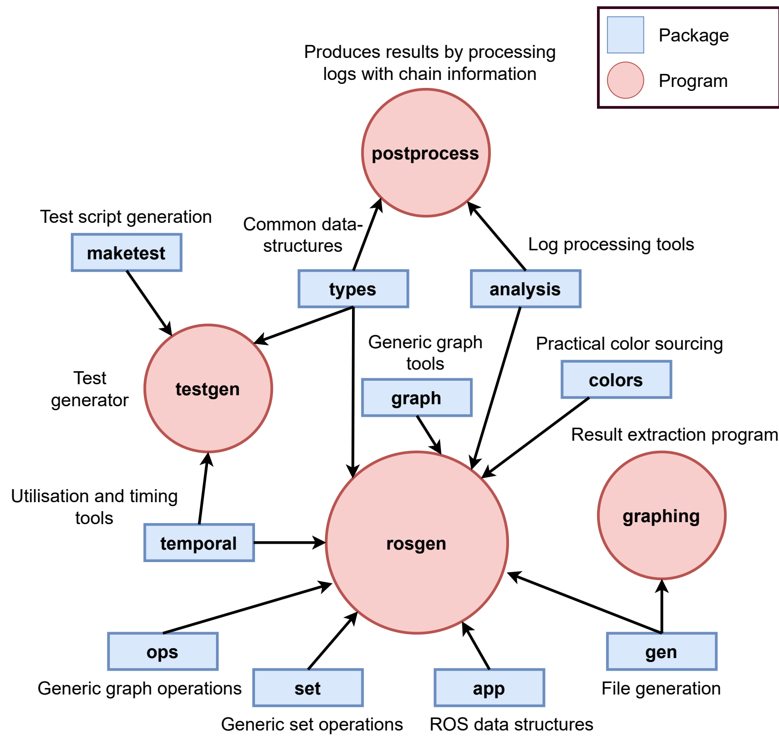


Figure 6.1: **Dependency graph for the evaluation framework. Circle nodes correspond to executable programs, and boxes to imported packages**

The central program of the testing framework is called *rosgen*, with the others playing a more peripheral role. *rosgen* fulfills the task of converting a specification of a ROS application, called a *rules* file, into a compilable ROS package. The program itself is composed of a series of six stages, split into two sections. The stages are illustrated in figure 6.2.

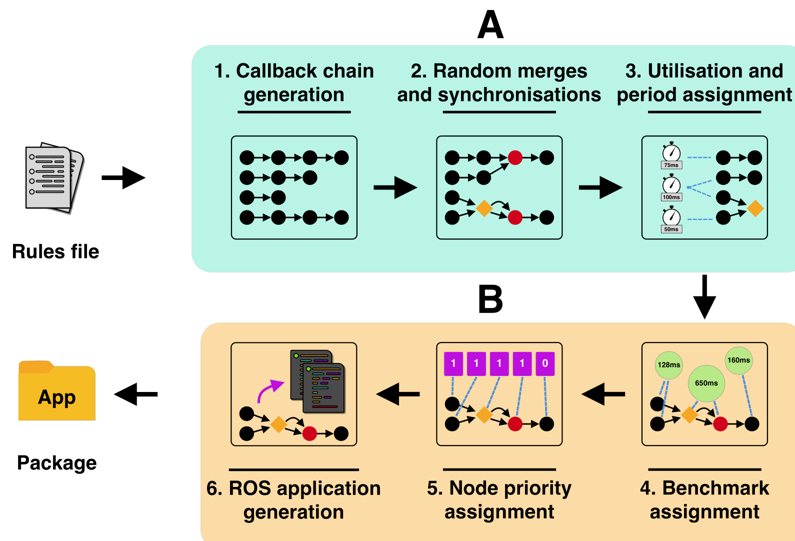


Figure 6.2: **Steps involved in converting a rules file to a ROS application package. Section A corresponds to the generation of a random program, and section B to the conversion of a program specification into a ROS application**

The following sections will break down the steps of the testing framework in more detail. First, the stages of the *rosgen* program pipeline will be broken down and discussed. Then,

an overview of how tracing and logging is performed is provided, ending with the runtime-properties and analysis techniques used.

6.2 ROSGEN

6.2.1 Generation rules

In order to evaluate the solutions developed in section 5, ROS applications with various properties need to be generated. For instance, the effectiveness of a synthesis algorithm can be more meaningfully evaluated if applied to applications with adjustable amounts of synchronisations and shared callbacks. Likewise, an executor can be better evaluated against ROS applications where the utilisation can be varied. Therefore, the automatic generating tool should allow the developer to specify what properties of the application they want when generating it. The current generation tool, called *rosgen*, supports the parameters listed below in table 6.1:

Table 6.1: **Adjustable properties for generated ROS applications**

Name	Description
Average chain length	Length of chains in the system if averaged
Merge probability	Probability of a merge between any two callbacks
Synchron--isation probability	Probability of a synchronisation between any two callbacks
Chain length variance	How much a chain's length can vary as a proportion of the average length
Total utilisation	Ratio of the CPU time required by all chains against the period
Minimum period (micro--seconds)	Smallest period allowed for a chain
Maximum period (micro--seconds)	Largest period allowed for a chain
Chain count	Number of chains to use in the system

Name	Description
Hyper--period count	Number of hyperperiods to run application (optional)
Maximum duration (micro--seconds)	Cutoff time for the executors (optional)
Executor type	What kind of executor to use: (0) Standard ROS (1) Producer-consumer (3) Thread-dispatch
Executor count	Number of executors to use in the system
Random seed	Pseudo-random generator seed (reproducibility)
Logging mode	Log types to collect: (0) None (1) by callback (2) by chain
Period step (micro--seconds)	Step size for periods
Use priority synthesis	Whether to use priority synthesis

Limitations on properties. Evidently, the properties in table 6.1 are tailored for maximising the control of callback chains. Important ones would be the number of chains in the system, along with the average length of the chains and the likelihood that callbacks might be synchronised or merged. It should be noted that not all configurations are possible, however. For instance, a ROS application model might only support so many synchronisa-

tions. And it is also not possible to merge all callbacks without violating properties of the chain model (e.g. at least one independent callback). The program therefore delivers a best-effort approximation of the desired properties, while ensuring that constraints are respected.

Input format. All of the aforementioned properties are specified in a *rules* file in **JSON** format. This file may be fed to *rosgen* as input. *rosgen* also supports a number of other parameters and arguments, such as an ability to override the derived timing information for chains.

6.2.2 Generating callback chains

Generating chain lengths. *rosgen* generates callback chains using the number of chains, average chain length, and chain variance values provided by the rules file. It attempts to ensure that the average chain length of the resulting set of chains matches the input value, while still allowing chains to vary in length in accordance with the variance. To do this, the program uses the average chain length as a mean, and computes a standard-deviation from the given variance float. It then samples a normal distribution based on the mean and standard-deviation to obtain the random chain lengths.

Representation. The graph itself is represented using an *adjacency* matrix, a type of topological model for networks. Once the lengths of the chains are established, the total number of callbacks in the system are known. Given a system of N callbacks, an $N \times N$ adjacency matrix is created. An example of an adjacency matrix for a simple graph is shown in figure 6.3. Because there are multiple callback chains in a ROS application, there is a need to distinguish to what chain an edge belongs to in the adjacency matrix. This can be accomplished by either maintaining a separate adjacency matrix for every chain in the system, or using tokens to identify edges in a single adjacency matrix. The approach chosen is to do the latter. Each element of the matrix is a set. Within the set, tokens may be placed which identify what chain has an edge from the callback identified by the given row to the callback identified by the given colummn.

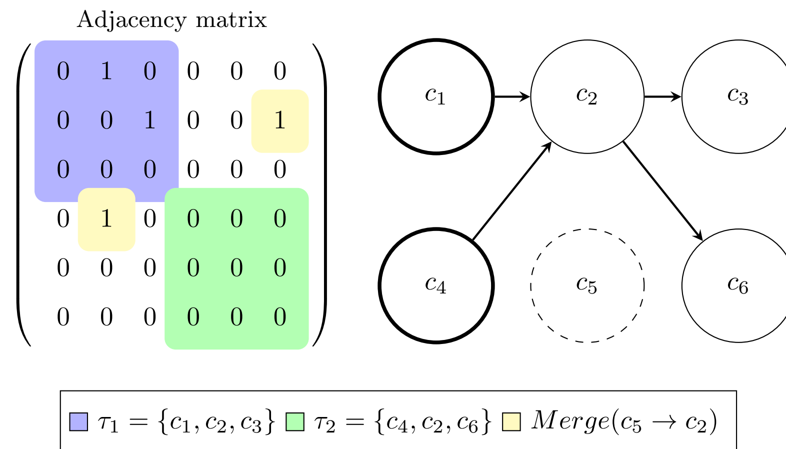


Figure 6.3: **An adjacency matrix representing two chains with one merged callback**

Once a graph has been initialised, the chains are written to the graph one after another. The first chain τ_1 in figure 6.3 of length three is assigned rows zero through two. The next chain of length three (τ_2) is assigned row three until five, and so on. A row corresponds to a particular callback, and the columns along the row are slots. If a token is placed into the slot, it means that an edge departs from the callback at the given row, to the callback on the given column (for the chain with which the token is associated). It can be seen in figure 6.3 that a merge operation has adjusted τ_2 such that c_5 is merged into c_2, hence row five and column five are empty in the adjacency matrix. In the following sections, references to

nodes in a graph will be made. In order to not confuse these with ROS nodes, these nodes will be referred to as *graph nodes*.

6.2.3 Random merges and synchronisations

Random merges in the graph are specified using a merge probability value. This value dictates the likelihood that any two random graph nodes may be merged. In order to compute how many merges should occur, the tool calculates a number of *chances*, which is dependent on the number of available graph nodes in the model. In short, the number of chances corresponds to a number of dice rolls. For a system of N graph nodes, there are $\frac{(N(N-1))}{2}$ interactions. The algorithm for calculating the merges to perform is 3, and may be found in the appendix (section A.2.1). In short, the algorithm effectively attempts to generate merges for as many times as it has chances. Once a merge is achieved, the number of remaining graph nodes has decreased by one. Therefore, the number of chances is recomputed and it continues to attempt to merge. If it happens to not merge any of the graph nodes in a full iteration through the number of chances, it exits.

Merge constraints. After a set of possible merge operations has been computed, the program attempts to apply them to the graph. However, as mentioned previously, not all merges are possible. Therefore, only those that do not violate graph constraints are permitted. The rules that determine the legality of a merge are as follows:

- If a merge would create a cycle in any path (chain), it is forbidden
- If a merge would cause a path (chain) to be indistinguishable from any other, it is forbidden
- It is forbidden to merge graph nodes of differing types (e.g. timers with subscription nodes, or subscriptions with sync nodes)
- It is forbidden to merge adjacent graph nodes from the same chain (as it would shorten the chain length)
- It is forbidden to disconnect a graph node (i.e. callback) from the graph (merges do not count)

Consequently, graphs usually appear to have fewer merges than input might imply.

Synchronisations. Synchronisations are performed similarly to merges, although the constraints are slightly different. In order for a synchronisation to occur, the program searches for graph nodes with two or more incident edges or outgoing edges. This method does impose a certain limitation on the variability of graphs, as it does **not** consider synchronising independent nodes each with one *outgoing* edge. An example is demonstrated using figure 6.3. Here, a synchronisation is applied between c_1 and c_4, going to c_2. The effect on the matrix representation may be seen in figure 6.4. The synchronisation operation is represented by an *extension* of the matrix in order to accomodate the new synchronisation node. The necessary inputs and outputs are then rewired.

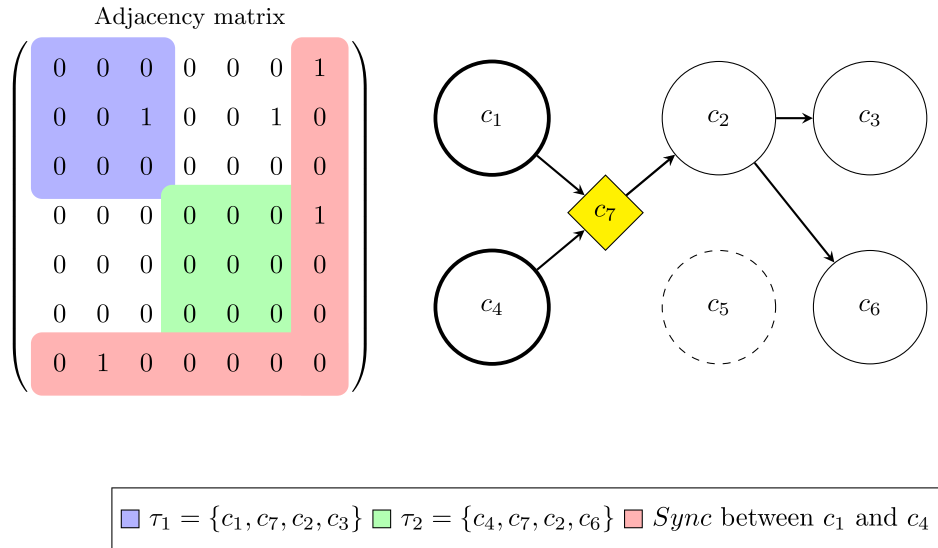


Figure 6.4: **Graph with two chains, one merge, and one synchronisation**

Deadlock detection. Generated ROS applications with arbitrary synchronisations may cause deadlocks to occur. This is because we assume message synchronisation mechanisms hold and wait for all inputs to arrive before releasing an output. The generation of circular wait relations thus produces deadlocks which are neither realistic or useful to evaluate. A deadlock detection check is thus performed before each synchronisation is added. The algorithm creates a deep copy of the existing adjacency matrix, and applies the synchronisation to it. It then searches backwards down the inputs of each synchronisation mechanism until it (1) hits a dead-end (2) arrives at another synchronisation graph node. Each synchronisation graph node is thus assigned a list of dependencies. By recursively searching the dependencies of each graph-node for itself, deadlocks may be detected. The presence of a deadlock then causes the proposed synchronisation to be discarded, leaving the original adjacency matrix deadlock-free. This deadlock check is also known as a *wait-for* check.

Synchronisation limitations. As synchronisations are only applied to multiple inputs and outputs of graph nodes, the parameters are effectively not independent. Although it would be possible to allow arbitrary synchronisations between graph nodes with only one outgoing edge, this approach was avoided for the reason that synchronising arbitrary graph-nodes has the potential to cause deadlocks not detectable by the detection algorithm (e.g. a chain synchronising two nodes within its own chain). However, the limitation is acknowledged, and the development of more thorough deadlock detection left for future work.

6.2.4 Utilisation and period assignment

Utilisation. The *rules* file permits the user to specify a *total utilisation*, along with a *minimum period* and *maximum period* for a ROS application. The utilisation value is a real number between 0.0 and 1.0, and represents the ratio between the CPU time used by the application (C) and the total available CPU time (T), as $U = \frac{C}{T}$. In order to divide utilisation up among N chains in an efficient manner, the *UUnifast* algorithm for uniprocessor tasksets by Bini and Buzzato [4] is used, and the total sum set to the desired overall utilisation such that:

$$U = \sum_{i=0}^{N} u_i$$

Period. Once a utilisation value has been generated for each callback chain in the system, the period of each chain's timer is randomly chosen within a range bounded by

the desired minimum and maximum value, and as a multiple of the step value, and the computation time derived in accordance with the utilisation. As chains may have merged timer callbacks, a follow up step picks a common timer for affected chains, and updates their CPU times in order to respect their prescribed utilisation.

6.2.5 Benchmark assignment

The need for benchmarks. Just as utilisation assigns CPU time to callback-chains, there must also be a way to simulate it. Using the built-in thread *sleep* utilities is considered inadequate. This is because they effectively block the thread, enabling other threads to run, and therefore don't simulate real work. Ideally, task-sets (or callback chains) should be performing work representative of what is commonly performed on real-time systems. Thus, a task-set generator based on the TACLeBench benchmark suite is adapted in order to meet the need for implementation-based analysis (as opposed to simulation and formal proof) in ROS [7].

TACLeBench . The TACLeBench benchmark suite contains a number of publically available real-time systems benchmarking programs for the purpose of evaluating scheduling algorithms. For the system under test, the various programs that compose the suite are compiled and executed for a fixed number of iterations. Each iteration is timed, and an average execution time derived. The results of all benchmarks are stored in a file. When *rosgen* is executed, it draws upon this timing file in order to assign benchmarks to different callbacks. The benchmark may be executed one or more times in order to simulate the exact amount of CPU time needed by a callback. *rosgen* performs the following steps in benchmark assignment:

1. **Mapping CPU time to callbacks**: Each callback in the system is assigned an empty set. For each chain, the total CPU time of the chain is divided by the number of callbacks that compose it. This fractional CPU time is then distributed across all callbacks, and added to their sets.

2. **Resolving shared callbacks**: Callbacks that are shared must pick only one CPU time. Therefore, the smaller of the CPU times is always kept, and the extra time needed by the affected chain redistributed across its remaining callbacks. As all chains must have one unique callback, this residual execution time can always be alloted safely.

3. **Selecting a benchmark**: Benchmarks are assigned on a callback basis. In order to determine which benchmark to use for a callback, the required CPU time is divided by the averaged execution time of each benchmark. The benchmark that is the *best divisor* (smallest difference as an integer multiple) is selected, and the name and number of iterations (the multiple) recorded.

Packaging benchmarks. In order to make the TACLeBench benchmarks accessible to ROS programs, the open source project is transformed into a static archive (library), and pruned of benchmarks that no longer work (for various reasons). This static archive is copied into every generated ROS program, along with a common header. Note that the creation of the library and pruning is done manually, but only needs to be done once.

6.2.6 Priority assignment

Priority assignment in the toolchain is often adjusted within the source code itself for ease of development. This is for two reasons:

1. Priority schemes are far easier to realise with direct access to the data-structures

2. As *rosgen* generates applications with a random element to their graph structure, the amount of information available beforehand (i.e. what callbacks exist) is very limited.

Two priority modes may be chosen within the source code. They are:

1. **Rate monotonic**: Priorities are assigned in descending order by ascending chain period (i.e the chain with the smallest period is assigned the highest priority)

2. **Priority chain 0 / RM priority chain 0**: A higher priority is assigned to the first chain (ID = 0), and all other chains are left at an identical lower priority. Alternatively, rate monotonic priority assignments are given to the chains, but the highest priority chain is always given ID = 0. This assignment scheme is used for evaluation.

Priority synthesis

If the *rules* file provided as input has indicated that synthesis is to be used, the synthesis algorithm from section 5.2.2 is applied once all the chains have been given a priority value.

6.2.7 Feasibility check

Following the assignment of priorities, ROSGEN applies the feasibility test defined in section 5.3 to the graph model. If the generated application is not feasible, it halts the build and informs the user.

6.2.8 Application generation

Generation steps. Generating the actual ROS package requires accounting for benchmark libraries, tracing utilities, logging modes, and more. Extra source files (and their headers) must be included in the source directories of the package, and also in the build scripts that accompany it. Executables must be both generated and listed in a launch file such that they all can be initialised simultaneously. The steps of application generation are enumerated below:

1. **Metadata assembly**: ROS packages, header files, and libraries are prepared into a application data structure. The duration from the *rules* file is also included, and the priority range (for the producer-consumer executor) is calculated so the generated program knows how many priority levels to use (if using that executor).

2. **Graph to application model**: The adjacency matrix representation is transformed into a structure resembling the ROS application model. Publishers and subscribers are created, and callbacks are assigned topics to publish on or respond to. Callbacks then undergo the following organisations:

   (a) **Executor distribution**: Callbacks must be distributed across the number of executors specified in the *rules* file. A number of buckets are created, and callbacks distributed across the buckets in a random order.

   (b) **Node grouping**: As described in section 1.1.1, ROS organises callbacks into nodes within executors. The policy chosen for allocating callbacks to nodes is simple. Callbacks belonging to the same chain are likely to share data in real-life applications. Therefore, they are placed into the same node. One node is created for each chain in the system on every executor. This policy does not apply to message filter (synchronisation) nodes.

   (c) **Synchronisation mechanism allocation**: Message synchronisation mechanisms are placed in the same node as their output callback.

3. **Package generation**: The necessary directories for the package are first created, with the top-level directory named after that specified in the *rules* file. Then, the executor source files are built from templates, using the data structures created in the previous step. The supporting source files, headers, and archives are also copied in. Finally, the *makefile*, package descriptor file (*package.xml*), and launch file are created and placed into the directories.

4. **Assets**: To help developers assess generated applications, a *graph* structure image and *application* structure image are automatically generated using *graphviz*. These are placed in an assets directory within the package. An example the automatically generated graphs may be seen in the appendix (see section A.2.2 for figures A.1 and A.2). The system displayed there has four executors and three chains of average length three (merging probability of 0.1, sync probability of 0.1).

6.3 Tracing and logging

It is necessary that all generated applications support some form of logging for analysis. Besides disabling logging outright, *rosgen* enables *chain-level* logging and *callback-level* logging. We first describe in more detail how logging is performed, and then enumerate over the logging modes available.

Logging a ROS application is more complex than a single executable. As an application may consist of several concurrent executables at run-time, it is not possible to log to a shared file easily without possible race conditions, and, assuming those were safeguarded against, extra overhead from mutexes or other thread-safe devices. Therefore, the path of least resistance is taken, and a native Linux logging utility (`syslog`) used. The utility is socket based, relying on UDP datagrams. This nullifies the need for any access-control and is well suited for use with concurrent executables. Two logging modes are available to developers:

1. **Chain-level**: Chain-level logs instruct *rosgen* to insert logging calls in the very *last* callback of each callback-chain. These log the start and duration of the chain using a timestamp in the forwarded message. This allows the developer to get the end-to-end response-time of a chain instance directly in the logs. It performs just a single log per chain instance, but callback-level introspection is not available.

2. **Callback-level**: Callback-level logs instruct *rosgen* to insert logging calls at the beginning and end of each callback. These contain information about which chain the invocation belonged to, and the start and duration of the callback. It performs more logging than chain-level does, and also provides more detailed information about how callback execution occurred. However, it requires more complex log analysis in order to reconstruct an end-to-end response time, and adds more overhead.

6.4 Analysis

The analysis program in ROS, named *postprocess*, is a log processor for use with automated testing scripts. Depending on the logging mode, *postprocess* reads in a logfile along with a *chain* file (*rosgen* creates this file with each package, and it describes the properties of the generated system to analyse). It then aggregates end-to-end response times for callback-chains in the logs, and outputs a worst-case, best-case, and average-case response time. It pairs the response-time with other information about the chain that it gets from the *chain* file. Its output is thus a single row per chain in the analysed system, with a standard layout to the columns. This allows logs from several different tests to be concatenated together, and the results of the test parsed out of the file by filtering on column keys (e.g. all tests with an average chain length of 5).

6.5 Message data

One aspect of ROS applications not touched upon by the evaluation framework is the data transported in messages. All messages relayed in the generated programs consist of a single eight byte value, which always holds a timestamp created at the timer (so that the duration of the chain may be measured at the end). This is evidently not a realistic reproduction of real-life applications, which may send messages of varying sizes. The expansion of the evaluation framework to account for this is left as future work. Nevertheless, interesting

research on the effect of messages sizes on performance in ROS 2 may be found in work by Maruyuma et al. [17]. They study the effect of varying the size of messages in combination with various quality of service (QoS) profiles for DDS (the ROS 2 backend).

Chapter 7

Evaluation

Evaluating the principle contributions of this work requires answering the following question: To what extent do the preemptive-priority executor implementations and priority synthesis scheme improve predictability of ROS 2 applications over standard ROS? Answering this, however, requires not only that we define metrics of predictability suitable for ROS applications, but that the conditions under which they are evaluated remain true to those of the real-world.

Therefore, the evaluation must be prefaced by both defining how we measure *predictability*, and conducting a brief investigation into a number of ROS applications. These investigations will help establish reasonable estimates for the properties used when generating synthetic tests. The estimates can then be used in conjunction with the testing framework described in chapter 6. The result is that the test outcomes are more or less based on realistic configurations.

The structure of the evaluation chapter is thus as follows. First, an experimental foundation will be setup in which we define performance measurements and establish estimates for ROS 2 application properties. Next, we compare the overhead of the new executor implementations to the standard RCLCPP executor. Following this, we evaluate how the new executors and synthesis algorithm perform relative to the standard ROS executor under a number of scheduling policies and test configurations. Finally, we summarise our findings.

7.1 Experimental Foundation

7.1.1 Performance

The preemptive-priority executors and the accompanying priority synthesis scheme are designed to remove priority inversions and synthesise callback priorities respectively. Two potential applications exist for these contributions. The first is that it can be used to improve the general schedulability of ROS applications by enabling the use of contemporary scheduling policies. The second is it can be used to provided targeted reductions in the end-to-end response time of priority callback chains. As automatically generated applications contain differing numbers of callback chains and utilisations, a normalised metric is required in order to meaningfully compare outcomes. The predictability of a ROS application is therefore qualified using the following metrics:

1. **Normalised response-time**: The normalised response-time of a chain is computed by dividing the response-time of the chain with the period of the chain. We assume an implicit deadline equal to the period ($D_i = T_i$). The ratio gives an indication of how well a chain performs. A ratio less than or equal to 1.0 means that the response-time meets the deadline. Values larger than 1.0 indicate an a deadline miss.
2. **Mean normalised jitter**: We compute the normalised jitter of callback chain to be the difference between its (normalised) worst and best case response times. Over a

series of trials, the *mean jitter* is the average of the observed jitter values for each chain in a particular instance of a test application. Jitter is an important metric in assessing the stability of a system.

3. **Mean minimum deadline misses**: When ROS 2 applications are evaluated, a post-processing stage of the evaluation framework outputs a maximum, minimum, and average response-time metric for *each chain* in the system for a thirty second evaluation period. The occurrence of a normalised worst-case response-time greater than 1.0 means at *least* one deadline miss was observed (although there may have been more misses of a smaller magnitude) for a chain over that time. We define the *minimum deadline misses* over a test application to be the number of chains that certainly missed one deadline. Over a series of trials, we define the *mean minimum deadline misses* to be the average number of minimum deadline misses over the tests in the series.

7.1.2 Investigation

To establish a foundation for experiments that reflect real-world conditions, we investigate several existing ROS projects. The number of executors, length of callback-chains, and ROS constructs used in each project are tabulated (see table 7.1). As ROS projects may consist of multiple packages, dozens of source files, and sometimes require special hardware to be run, we inspect them by code review only, and not through the use of online-tools such as `ros_graph`. Manual inspection is also necessary for locating and counting executors, as that information is not available through the aforementioned tools.

- **Collaborative-Robot-Sanding**: The Southwest Research Institute (SwRI) sanding robot is a product of an internal research project into force and speed control software, with an aim of providing a more general approach to industrial robotic applications. The robot performs the bulk of repetitive sanding tasks used in aerospace applications.

  Inspection of the open-source project reveals a neat isolation of functionality into servers. We count approximately **seven** executors (four `MultiThreadedExecutors`, two `SingleThreadedExecutors`, and a custom modified variant of the existing executor implementation), fulfilling tasks such as motion-planning, perception, comms, area-selection, etc. We also find **six** timers across the system, typically publishing at **2Hz**. The timers are linked to callbacks which publish on topics related to attributes like the system state, and various markers. Our inspection of the callback-chains determines them to have a length of at least **three**, due to presence of ROS action constructs. Finally, we find that the system makes extensive use of ROS services, although we don't consider them due to the lack of support for services in our solution.

- **OSRF-Drone-Demo**: The Open Source Robotics Foundation (OSRF), a member of the Open Robotics group, also produces open-source software and hardware platforms in addition to supporting ROS itself. One such project is known simply as ng drones, or the drone demo. This project, which is still ongoing, bridges PX4 autopilot software and Gazebo together through ROS, providing an entry-point for general UAV software development. [1] [29]

  The two items of focus in the inspection are the ROS components of the SITL Launcher and drone node itself. The odometry broadcast node appears to run in its own `SingleThreadedExecutor`, and publishes odometry information. The drone node also seems to runs in a `SingleThreadedExecutor`, and handles flight mode goals as well as converting PX4 vehicle messages to another message format called REP 147. We find a timer that publishes the flight mode at **10Hz**. Odometry, battery status, attitude, and other information is passed to RVIZ (for visualization), to which control input is returned in a loop via a command pose. In summary, we count at least **two** executors involved with the drone, with a callback-chain length of at least **three** in length, and at least **two** timers involved. As with the previous project, we find that ROS action constructs are also used to carry out flight goals. We do not find any services in the project.

- **Vanderbilt F1Tenth**: The University of Vanderbilt F1Tenth team produced a candidate vehicle for the autonomous racing competition F1Tenth. Originally an initiative by the University of Pennsylvania, F1Tenth serves as a means to foster interest and development in autonomous systems research. Analysis of this project is not ideal, as the project is based on ROS 1, and not ROS 2. Furthermore, the code is produced for a Python implementation of the ROS client facing libraries, and not C++. Finally, the project includes third party packages such as Hector SLAM, which obfuscates details [20].

  The results of the analysis expose what appear to be at least **five** independent processes of execution, completing tasks such as a driver for the IMU, two handlers for keyboard and joystick input, and servers for odometry correction as well as control. Throttle and steering commands are driven by a single timer, and published at a rate of **20Hz**. The vehicle also appears to receive camera and LIDAR data, which is processed using the Hector SLAM package. Although we don't inspect the contents of the Hector SLAM package, callback-chains are expected to be at least **four** in length.

Conclusions. The following verdicts are reached as a result of the investigation (which are also handily available in table 7.1): ROS projects typically make use of **between two to seven executors**. Third party packages, however, typically run their own executors, which suggests the *actual number may be even higher* for some of the projects covered. Next, between **one and six timers** are discovered across the projects. This variation is present due to both the size and scope of projects, and the design philosophy (time-driven vs event-driven) used. Finally, **callback-chains are determined to span between three to four in length**. Again, this could only be done for chains which could definitely be traced between callbacks, and messages to packages may well go through several processing steps before returning, suggesting the processing chains are most likely longer.

Table 7.1: **Estimates for properties of various investigated ROS 2 applications**

Project	Executors	Chain length	Timers	Timer Periods
Collaborative-Robot-Sanding	7	3	6	˜2Hz
OSRF-Drone-Demo	2	3	2	˜10Hz
Vanderbilt F1Tenth	5	4	1	˜20Hz

7.2 Experimental setup

Evaluating the new executor implementations and algorithm involves various tests in which different properties of ROS 2 applications are controlled for. However, some conditions, such as the environment in which the tests are run, and how CPU resources are allocated, are common to all the tests conducted. The following subsections discuss these matters in detail. In all coming sections, the following terminology is used:

- **Standard**: Refers to the standard RCLCPP `SingleThreadedExecutor`, introduced in section 2.3.
- **New (PC)**: Refers to the new *producer-consumer* executor implementation, introduced in section 5.1.4.
- **New (TD)**: Refers to the new *thread-dispatch* executor implementation, introduced in section 5.1.3.

7.2.1 Environment

All tests are run on Ubuntu 18.04.5 LTS, with a modified implementation of the RCLCPP library (containing the new executors) for ROS 2 Foxy Fitzroy. All executors are run under the Linux real-time scheduling policy `SCHED_FIFO`. With the exception of the work threads on the new executor implementations, all executors are run at real-time priority 99 (the highest allowable priority under the policy). Prior to any test, the host system is rebooted

with networking disabled, and the trials run under root privileges as the only user-initiated process on the system.

7.2.2 Core allocation

For both the New (PC) and New (TD) executors, two cores are allocated. One core is dedicated for the scheduler thread of the executor, and all of the remaining work threads are pinned to the other core. The standard executor is not multi-threaded, and thus runs only on one core. Irrespective of the scheduler overhead, all callbacks are **executed on a single core** for every executor.

7.3 Experiments

We conduct the following experiments in order to assess both the new executor implementations and the priority synthesis algorithm:

1. **Overhead of executor implementations.** (section 7.3.1) This test compares the overhead of the new executor implementations (New (PC), New (TD)) against the Standard executor by outfitting the executors with profiling code and measuring the time needed for each executor to process a message and launch a callback.

2. **Effect of utilisation on executors.** (section 7.3.2) This test looks at how applications with varying levels of utilisation (10% - 90% with increments of 10%) compare under each executor under a rate-monotonic priority assignment policy. The results are discussed in terms of normalised worst-case response time, mean minimum deadline misses, and mean normalised jitter.

3. **Effect of chain length on executors.** (section 7.3.3) This test looks at how applications with varying chain length (length 2 - 10, with increments of 2) and proportionally scaled computational work compare under each executor under a rate-monotonic priority assignment policy. The results are discussed in terms of normalised worst-case response time, mean minimum deadline misses, and mean normalised jitter.

4. **Evaluating executor choices for high-priority time-sensitive callback chains.** (section 7.3.4) In this test, the difference in the predictability of high-priority chains on applications is compared between each executor. The results are discussed in terms of normalised worst-case response time, mean minimum deadline misses, and mean normalised jitter.

5. **Effect of priority synthesis algorithm.** (section 7.3.6) Testing the priority synthesis algorithm involves applying it to test-applications with varying levels of synchronisation probability. In this test, we vary the average chain length of test-applications between 3 and 9 with a step of 3. This is performed twice. Once for a synchronisation probability of 15%, and once for 30%. The Standard algorithm is used to run each test-application, and is compared to the New (TD) executor and New (TD) executor *with synthesis*.

A summary of the results is provided in section 7.4.1.

7.3.1 Overhead of executor implementations

Prior to assessing the performance of ROS applications on the new executors, their overhead is evaluated with respect to the standard. To accomplish this, each executor is outfitted with profiling code that measures the time it takes for the scheduler to process one ready callback. This time measured begins once work is retreived from middleware (RMW) via the client library (RCL), and ends right before a callback is executed. Profiling code may be seen in appendix A.1, and is enabled with a pre-processor macro.

Experimental setup

We evaluate the overhead of the new executor implementations by using a single test application. The parameters of the application may be seen in table 7.2.

Table 7.2: **ROS application test parameters for evaluating the overhead of executors**

Application configuration(s)

Property	Value
Average chain length	4
Number of chains	5
Synchronisation probability	0%
Total utilisation	60%
Use priority synthesis	No
Priority assignment	RM
Period (min, max, step)	1s
Executor type(s)	Standard, New (PC), New (TD)

Test settings

Property	Value
Test duration	30s
Number of trials	50

Parameters. The period of all timers is set to exactly one second (so they fire in unison), and the utilisation at 60%. This gives every one of the five chains 120ms of work. The period selection is in line with the findings of section 7.1.2, in which all timer periods (2Hz, 10Hz, 20Hz) fell into a band of roughly 50ms to 1000ms. For each executor evaluated, fifty trials are performed, in which the application is executed for a duration of 30s. Priority is assigned according to a rate-monotonic policy. As all chains have the same period, the resulting priorities are simply in descending order by chain ID.

Data collection. The scheduler profiling times are recorded in a pre-allocated buffer within the executor during the trial. Once the executors are signalled to finish, they dump the traces to a log file. This ensures minimal overhead is incurred from the profiling method.

Experimental Hypothesis

The intent of the test is to ensure multiple callbacks arrive simultaneously, as this creates burst of callbacks for the executor to handle. This should incur slightly more overhead on the producer-consumer executor (New (PC)) than on the standard (Standard), because the producer-consumer executor sorts the arrived callbacks into priority queues, which takes more time than simply grabbing and executing the next available callback handle. It is expected that in general, the new executor designs will suffer from a larger median overhead than the single-threaded RCLCPP executor. This is because of their more complex scheduling algorithms, and the risk of preemption overhead which is not present in the standard executor.

Results

The results are plotted in figure 7.1. The data for each executor was trimmed of outliers, keeping 2% - 98% of the distribution of results.

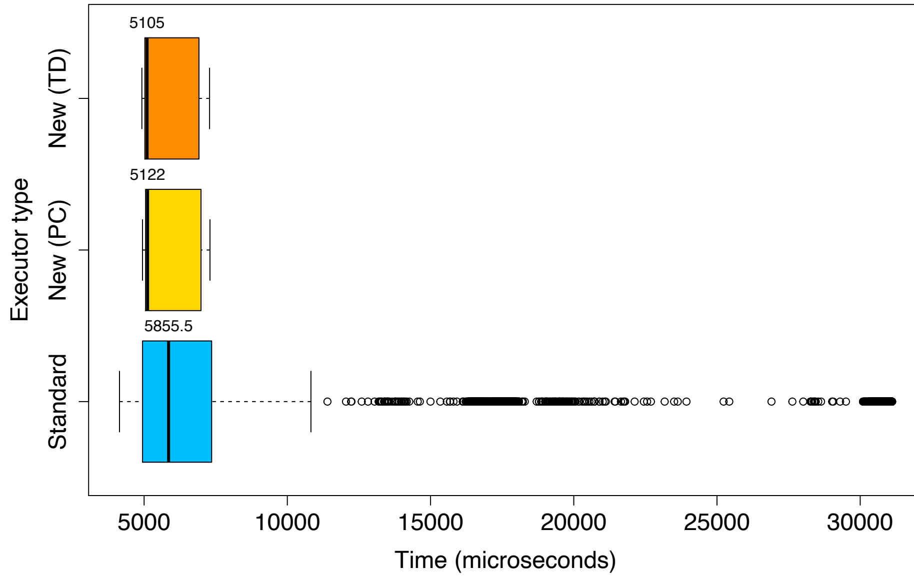


Figure 7.1: **Scheduler overhead (Chains: 5, Length: 4, Utilisation: 60%)**

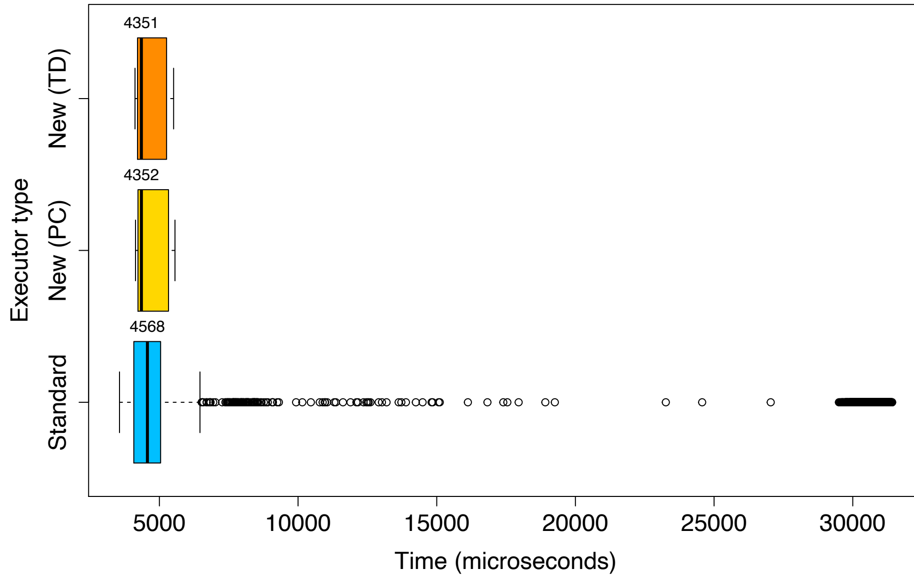
Initial results. The results show that the New (PC) and New (TD) executors experience less median overhead than Standard, with a *much* tigher distribution. The presence of a large amount of outlier overhead times for Standard runs counter to the hypothesis. This raises questions about whether the test conditions are the source of the anomaly. These are investigated by a series of subsequent tests that control for various possible causes of high overhead times.

Refined test. A refined test requires a hypothesis as to the cause of the high overhead for the Standard executor. Given the simplicity of the test-application, we derive two possibilities:

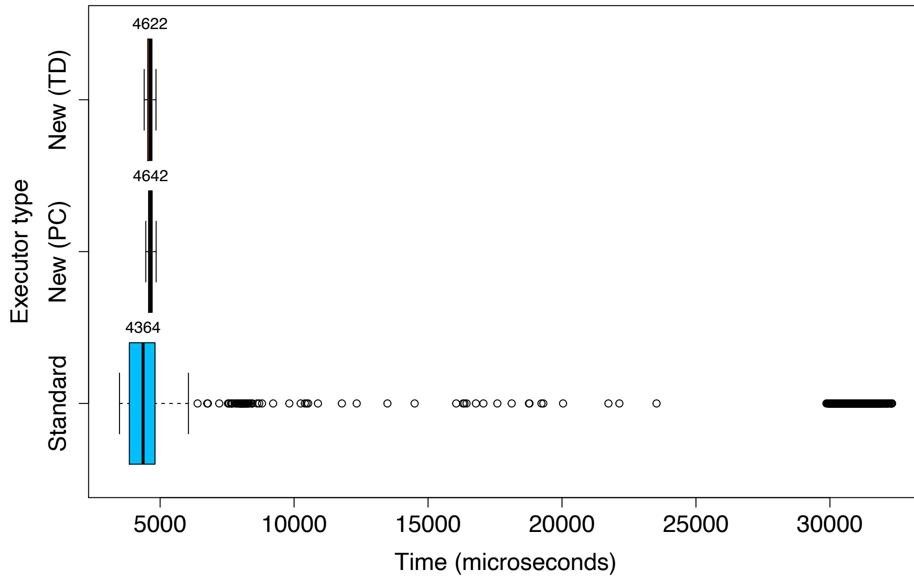
1. **Publish-subscribe message overhead**: The overhead incurred while waiting for messages to propagate down the chains may be sufficiently large that chains are missing their deadlines - in turn causing interference on the Standard executor when subsequent bursts of timer callbacks arrive.

2. **High utilisation**: The utilisation may be too high for the Standard executor to handle. By lowering the utilisation significantly and re-running the test, this factor can be mitigated.

Two subsequent tests are performed. The results are visible in figure 7.2:



(a) **Chain length: 1, Utilisation: 60%**



(b) **Chain length: 1, Utilisation: 15%**

Figure 7.2: **Refined executor overhead tests**

Figure 7.2a displays the results for the test in which all five chains in the test application were reduced to a length of one. We see that although a change in all median overheads is observed (despite no change to the utilisation), the Standard executor still performs worse than the New (PC) and New (TD) executors on both its median overhead, and the distribution of its results. The figure also clearly displays that a large amount of outlier times are still present.

Figure 7.2b illustrates the results for the test in which the chain length of all chains is reduced to one, *and* the utilisation is reduced from 60% to 15%. We see that in these

conditions, the Standard executor actually exhibits a superior median overhead time, but still suffers from the presence of many high outlier overhead times, which give it a wider result distribution.

Discussion

The results of the three tests indicate that the Standard RCLCPP executor exhibits large outlier overhead times irrespective of the (1) amount of publish-subscribe activity and (2) utilisation of the application. This behavior may cause problems for time-sensitive applications, as the high outlier overhead times may significantly affect the response-time of short callback-chains in the range of 20ms. By constrast, the two new executor implementations (New (PC), New (TD)) exhibit a far tigher band of overhead times. Perplexingly, neither test conducted allows for a conclusion as to the source of the overhead in the Standard executor. A follow-up hypothesis is that the high overhead is caused by the buildup of excessive bookkeeping work in the middleware layers. This would occur because the Standard executor is single-threaded, and cannot attend to newly arrived messages between polling points. By constrast, the new executors can process arriving messages in real-time.

7.3.2 Effect of utilisation on executors

A key item of interest in evaluating ROS 2 applications is how the new executor implementations deal with different utilisation. The following test investigates this question by subjecting all three executors to a series of applications in which the utilisation is swept between 10% and 90%.

Experimental setup

The experimental setup for the utilisation test is listed in table 7.3.

Table 7.3: **ROS application test parameters for evaluating the effects of varying utilisation across all executors**

Application configuration(s)

Property	Value
Average chain length	5
Number of chains	5
Number of executors	4
Synchronisation probability	0%
Total utilisation	10%-90%
Use priority synthesis	No
Priority assignment	RM
Period (min, max, step)	(20ms, 1000ms, 50ms)
Executor type(s)	Standard, New (PC), New (TD)

Test settings

Property	Value
Test duration	30s
Number of trials	50

In this test, each executor runs automatically generated ROS 2 applications with the level of utilisation being varied from 10% to 90% with a step of 10%. Fifty trials are performed for each level of utilisation, and for each executor. The generated applications sample the initial periods for chains from a range with a lower bound of 20ms, in order to ensure that the overhead of the executors measured in section 7.3.1 does not affect response-times too much. The maximum period is set to 1s.

Next, we choose only a single executor instance to be run during each test. This is to ensure that no interference occurs between processes, which would significantly penalise the Standard executor. It also gives a clearer picture as to how varying workloads are handled

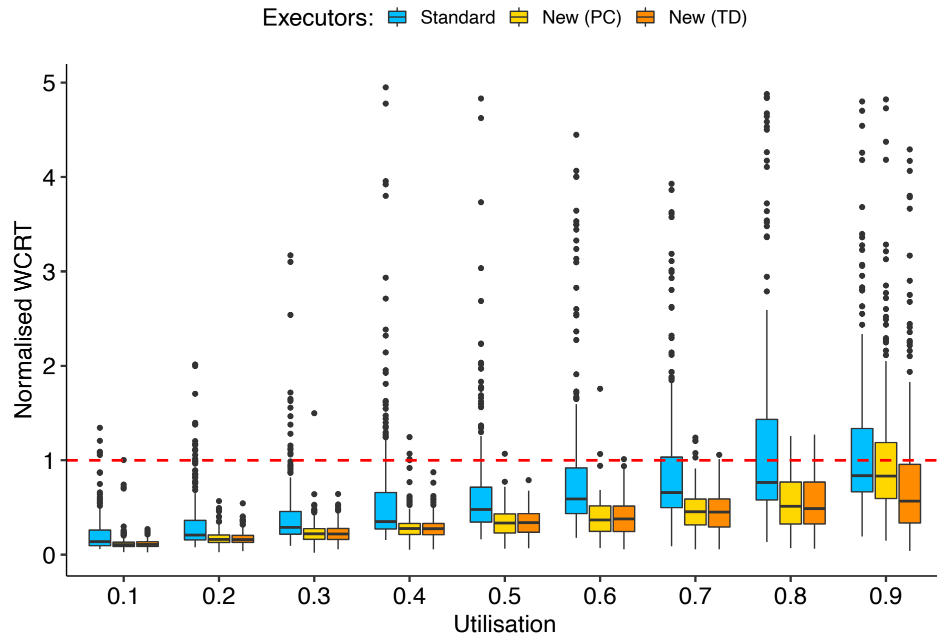
by the executor implementations. For the executors that can support prioritisation (New (PC), New (TD)), priority is assigned according to a rate-monotonic policy.

Finally, synchronisations and merges between executors are disabled, and the priority synthesis algorithm disabled. This is because synchronisations alter the utilisation of callback chains (by making them dependent on each other to complete), which the testing framework does not account for.

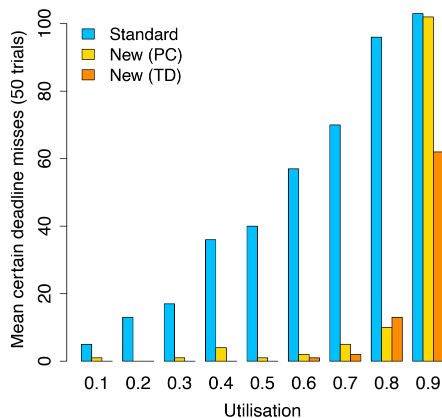
Experimental hypothesis

It is expected that applications run under the Standard executor will begin to exhibit deadline misses earlier than New (PC) and New (TD). This is due to the fact that the new executors are able to run the given applications using a rate-monotonic fixed-priority policy. Since we assume that for every chain, the deadline equals the period ($D_i = T_i$), then it is an optimal priority assignment method. By contrast, the Standard executor is non-preemptive, and thus has no optimal policy. According to the Liu and Layland rate-monotonic utilisation based schedulability test, a generated taskset is schedulable if it meets the following requirement: $U \leq n \times (2^{\frac{1}{n}} - 1)$ under ideal conditions [16]. Although conditions are far from ideal, we can extrapolate using the test that we should certainly be seeing deadline misses occuring at or before 74% for the New (PC) and New (TD) executors assuming we take $n = 5$ for five chains. We also expect to see deadline misses much earlier than that for the Standard executor.

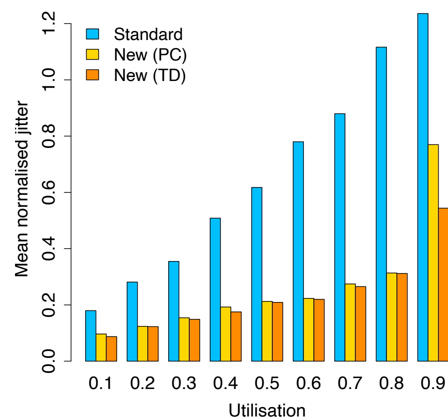
Results



(a) **Normalised worst-case response times vs utilisation. The red dashed line indicates the deadline miss threshold. Outliers over 5.0 are not shown**



(b) **Mean minimum deadline misses by utilisation level**



(c) **Mean normalised jitter by utilisation level**

Figure 7.3: **Response times, mean minimum deadlines, and mean normalised jitter by utilisation level**

Discussion

The results, which are shown in figure 7.3, indicate that deadline misses occur as early as 10% for the Standard executor and New (PC) executor, but not until 60% for the New (TD) executor implementation (see Table 7.4). The deadline miss threshold is indicated with a red dashed horizontal line at the 1.0 mark. We also see that the mean jitter is substantially smaller for all new executor implementations when compared to the standard (see figure 7.3c). At a utilisation level of 60%, the Standard executor exhibits a mean jitter that is **72%** larger than that of applications run under the New (TD) executor!

These outcomes fall in line with the hypothesis, although the presence of deadline misses (albeit small) in the lower utilisation range for the New (PC) executor is a bit surprising. The

Table 7.4: **Number of deadline misses by utilisation level**

Utilisation	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Standard	5	13	17	36	40	57	70	96	103
New (PC)	1	0	1	4	1	2	5	10	102
New (TD)	0	0	0	0	0	1	2	13	62

Standard executor behaved as expected, with deadline misses occuring with more frequency than either of the new executor implementations.

7.3.3 Effect of chain length on executors

While the effects of utilisation on the different executor implementations are now known, the experiment was conducted using synthesised ROS 2 applications for which the average chain length was fixed at 5. Furthermore, the test was only conducted on one executor. In practice, applications may have callback chains of varying lengths, and may be spread across more than one executor. Therefore, there is an interest in understanding how more realistic configurations of varying chain lengths affect response-times on the different executor implementations.

Experimental setup

The test parameters for the generated test applications may be seen in table 7.5.

Table 7.5: **ROS application test parameters for evaluating the effects of varying chain length across all executors**

Application configuration(s)

Property	**Value**
Average chain length	2-10
Number of chains	5
Number of executors	4
Synchronisation probability	0%
Total utilisation	60%
Use priority synthesis	No
Priority assignment	RM
Period (min, max, step)	(20ms, 1000ms, 50ms)*
Executor type(s)	Standard, New (PC), New (TD)

Test settings

Property	**Value**
Test duration	30s
Number of trials	50

In the generated applications, the chain length is varied from 2 to 10, with a step size of 2. The number of executors is also increased to 4 (a value selected between 2 and 7, which are the estimated number of executors found from the investigation of section 7.1.2). The utilisation is increased to 60%. From the results of the utilisation test in section 7.3.2, we see this is the highest accepted utilisation level for the New (TD) executor. Finally, merges and synchronisations are disabled just like in the utilisation test. This is done to ensure that deadline misses are not caused by synchronisations between chains, which alter their utilisation.

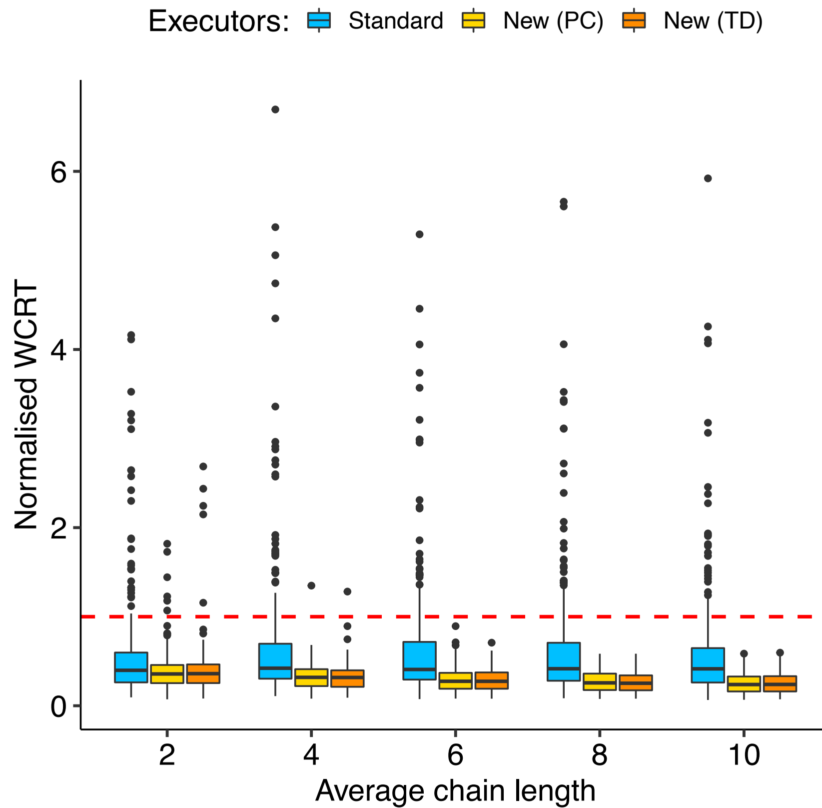
Custom timing. In this experiment, the computation time required by each chain scales proportionally with the length of the chain. To maintain the required utilisation, the period is accordingly adjusted. This is done to make sure that longer chains don't divide the same amount of computational work across an increasing amount of callbacks. This would amplify the effect of the executor and publish-subscribe overhead. This has the side effect

of raising the maximum period beyond the bounds indicated in 7.5, but is deemed necessary for the test.

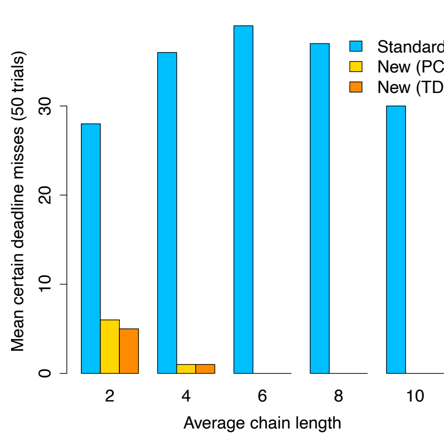
Experimental hypothesis

Just as with the utilisation test, we expect that the rate-monotonic priority assignment policy will ensure that both new executors exhibit lower worst-case response-times and fewer deadline misses than the Standard across all chain lengths. Next, we expect that as the chain length increases, we will see less of an effect of scheduler overhead across all executors. This will happen because the computational work per chain scales with chain length, which drives up the period, and drives down the significance of the scheduler overhead in comparison. This should also ensure that the effect of publish-subscribe overhead remains small, as the increase in the number of publish-subscribe events is countered by the increase in the period of the chains.

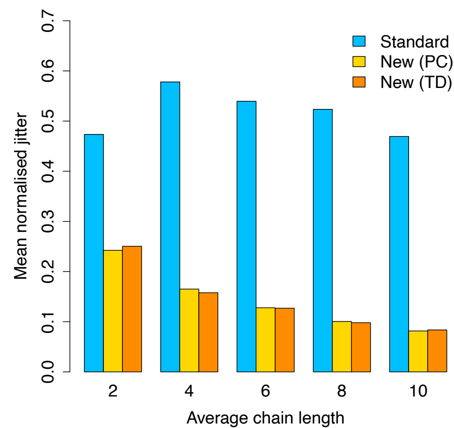
Results



(a) **Normalised worst-case response-times vs chain length. The red dashed line indicates the deadline miss threshold**



(b) **Mean minimum deadline misses by average chain length**



(c) **Mean normalised jitter by average chain length**

Figure 7.4: **Response times, mean minimum deadlines, and mean normalised jitter by average chain length**

Discussion

The results, shown in figure 7.4, fall mostly in line with the experimental hypothesis. The increase in chain length, which is paired with a proportional increase in the period of the chains, lowers the median response times of the New (PC) and New (TD) executors due to

the scheduler overhead becoming less pronounced (see table 7.6). Perplexingly, the Standard executor does not follow this trend, with larger median response-times despite the increase in period at larger chain lengths. This indicates that the overhead of the scheduler is likely not the cause. Overall, the normalised worst-case response-times remain lower for New (PC) and New (TD) executors with respect to the Standard. This outcome was expected, given the advantage enjoyed by the rate-monotonic priority assignment policy applied under the new executors.

Table 7.6: **Median normalised worst-case response-times for each executor type by average chain length**

Average chain length	2	4	6	8	10
Standard	0.398	0.422	0.408	0.416	0.415
New (PC)	0.357	0.319	0.275	0.257	0.238
New (TD)	0.360	0.317	0.274	0.253	0.239

In keeping with the trend set by figure 7.4a, we see that for the new executor implementations, the mean minimum deadline misses and mean normalised jitter also decrease with increasing chain length (and hence period). This can also be explained the same way. Namely, that the larger periods and computation times are offsetting the impact of the scheduler and publish-subscribe overhead on the response-times. Just as before, the Standard executor does not follow this trend. We see a slight increase in the jitter when moving a chain length of 2 to 4, and mean minimum deadline misses increasing until an average chain length of 8. A hypothesis for these increases is that the Standard executor is suffering from the increase in publish-subscribe message overhead, which is not offset until the periods of the chains are sufficiently large.

7.3.4 Evaluating executor choices for high-priority time-sensitive callback chains

Thus far, we have seen that the new executors provide superior performance (in terms of normalised worst-case response-times and deadline misses) for synthetic applications with both varying levels of utilisation and chain length. All of these applications have assigned priority according to a rate-monotonic policy. However, a key interest of the work is also to see how the new executor implementations can be leveraged to provide lower response time for specific time-sensitive processing chains. Therefore, this test aims to demonstrate how a prioritised chain performs relative to the rest of a ROS 2 application for both the new and Standard executors.

Experimental setup

The test parameters for the generated test applications may be seen in table 7.7.

Table 7.7: **ROS application test parameters for evaluating the effects of priority path chains across all executors**

Application configuration(s)

Property	**Value**
Average chain length	2-10
Number of chains	5
Number of executors	4
Synchronisation probability	0%
Total utilisation	60%
Use priority synthesis	No
Priority assignment	Chain 0: 1, All others: 0
Period (min, max, step)	(20ms, 1000ms, 50ms)*
Executor type(s)	Standard, New (PC), New (TD)

Test settings

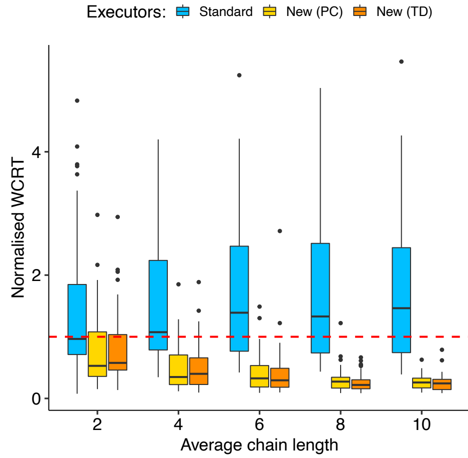
Property	**Value**
Test duration	30s
Number of trials	50

The generated applications are produced under identical conditions to that of the the chain-length test in section 7.3.3. This is because the varying number of chains and selected number of executors are a good approximation for a range of ROS 2 applications. However, the priority assignment scheme is adjusted in this test. Instead of a rate monotonic policy, the following scheme is applied: The chain with the smallest period is selected to become the priority chain. This chain is adjusted such that it is always the chain with ID 0. It is given a priority value one higher than the default (1), and the others left to the default priority (0). This makes it the only prioritised chain in the system.

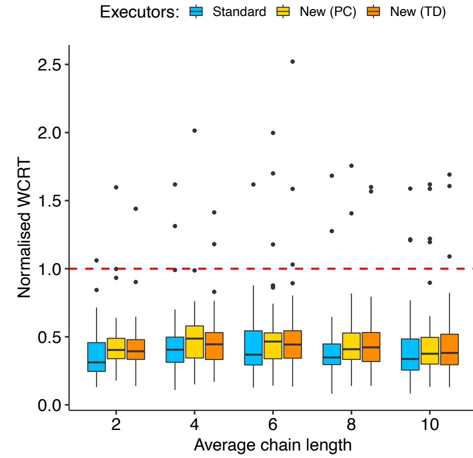
Custom timing. As in the previous experiment, the computation time required by each chain scales proportionally with the length of the chain. To maintain the required utilisation, the period is accordingly adjusted.

7.3.5 Experimental hypothesis

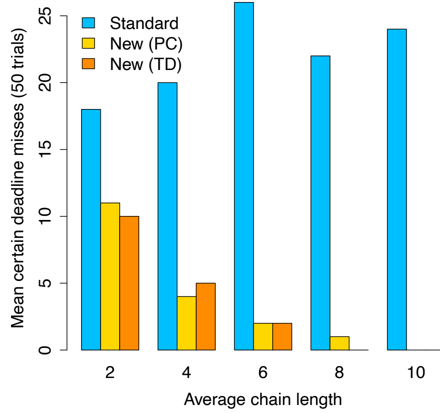
In this test, we expect to see that the prioritised chain exhibits much improved response-times across all chain lengths under the new executors. The non-prioritised chains are expected to perform worse than under the Standard, however. This is because they will not benefit from a rate-monotonic priority assignment policy, and will be subject to increased interference from the higher priority chain.



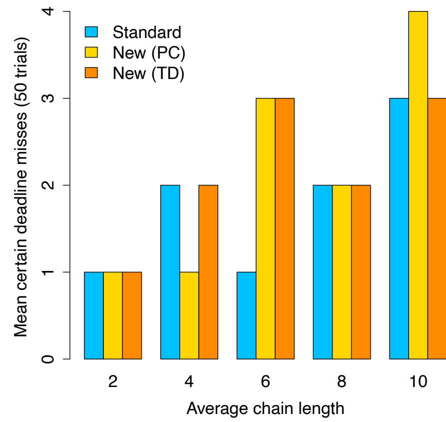
(a) **Normalised worst-case response-times for the high priority chain (chain 0). The red dashed line indicates the deadline miss threshold**



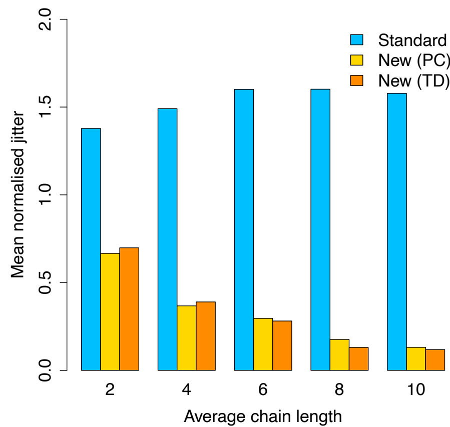
(b) **Normalised worst-case response-times for an arbitrary low priority chain (chain 2). The red dashed line indicates the deadline miss threshold**



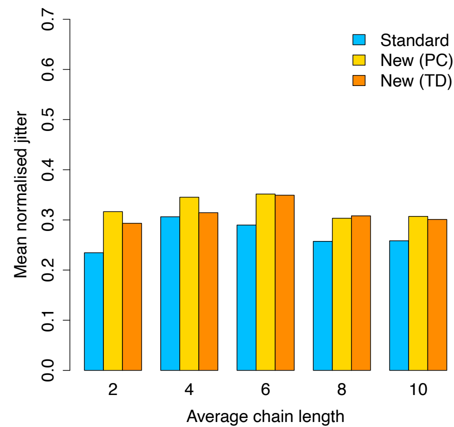
(c) **Mean minimum deadline misses for high priority chain (Chain 0)**



(d) **Mean minimum deadline misses for low priority chain (Chain 2)**



(e) **Mean normalised jitter for high priority chain (Chain 0)**



(f) **Mean normalised jitter for low priority chain (Chain 2)**

Figure 7.5: **Response times, mean minimum deadlines, and mean normalised jitter between a high priority chain (Chain 0) and an arbitrary low priority chain (Chain 2)**

Discussion

Response-time and deadlines. The results, shown in figure 7.5, show that the prioritised chain (chain 0) exhibits smaller normalised worst-case response-times under all chain lengths than the non-prioritised chain (see figure 7.5a, and figure 7.5b). The non-prioritised chain was selected at random from the four non-priority chains that are a part of each test-application. The prioritised chain also exhibits a significantly smaller number of mean minimum deadline misses than the non-prioritised chain (see figure 7.5c and figure 7.5d respectively). At an average length of 10, we see that the priority chain exhibits zero deadline misses under New (TD), as opposed to **24** from the Standard executor.

Jitter. Finally, we see that the normalised jitter for the priority and non-priority chain (see figures 7.5e and 7.5f respectively) is drastically smaller across all chain lengths for the priority chain. Most interestingly, we see that while the non-priority chain does exhibit poorer jitter under both new executor implementations, the relative difference is not as significant. For example, the largest mean jitter discrepancy between the Standard and new executor implementations on the non-priority chain occurs between New (PC) and Standard for an average chain length of 2, at 135% that of Standard. By contrast, Standard has a mean jitter that is 1300% that of New (TD) for a mean length of 10 as its largest discrepancy in the priority chain.

As hypothesised, the penalty across non-prioritised chains is consistently worse, though not egregious. This demonstrates that time-sensitive chains can enjoy superior end-to-end response times under simplistic priority assignment schemes at a relatively modest cost to that of non-prioritised chains.

7.3.6 Effect of priority-synthesis algorithm

As of this point, we have performed three evaluations. The first evaluation has examined how the executors respond to test applications with increasing levels of utilisation, under a rate-monotonic scheduling policy. The second evaluation has examined how the executors handle task sets of varying chain length under the same rate monotonic policy. The third evaluation looked specifically at time-sensitive priority chains, and how their worst-case response-times compare to the standard in test applications in which they are the highest priority chain. Finally, we assess the performance of the priority synthesis algorithm.

Until now, no evaluation has been performed on test-applications that include synchronisation mechanisms. This is because synchronisations alter the utilisation of applications by placing a dependency between chains with different periods. Given this is a metric that we wish to control in earlier tests, the probability of generating such constructs was always set to zero. However, realistic ROS 2 applications may contain synchronisations. Furthermore, the priority synthesis algorithm is designed to operate on them.

Now, we examine how the priority synthesis algorithm can improve performance when compared against (1) the Standard executor and (2) the New (TD) executor (without priority synthesis). Because the New (TD) executor is designed to operate with priorities, callbacks are by default assigned the highest level priority of any chain that crosses them (as a naive developer might do). The New (TD) executor with synthesis test furthers the priority assignment by using the priority-synthesis algorithm to propagate the priority down paths related the inputs of synchronisation nodes.

Experimental setup

The test parameters for the generated test applications may be seen in table 7.7.

Table 7.8: **ROS application test parameters for evaluating the priority synthesis algorithm under rate-monotonic scheduling policy**

Application configuration(s)

Property	Value
Average chain length	3, 6, 9
Number of chains	5
Number of executors	4
Synchronisation probability	15%, 30%
Total utilisation	30%
Use priority synthesis	Yes
Priority assignment	RM
Period (min, max, step)	(100ms, 1000ms, 100ms)*
Executor type(s)	Standard, New (TD)

Test settings

Property	Value
Test duration	30s
Number of trials	20

In table 7.8, there are a couple of notable differences in the parameter choices when compared to the other tests. The first is that the utilisation has been dropped to 30%. This utilisation value has been selected in order to buffer for the increase in utilisation that will occur when chains synchronise with one another. Next, we choose two synchronisation probabilities to apply at each step in chain length. These are 15% and 30%. Finally, the executors chosen have been the Standard executor (as the status-quo) and the thread-dispatch executor (New (TD)). This executor was selected over New (PC) as it has proven to be more tolerant to applications with higher levels of utilisation than the producer-consumer implementation (see the results in section 7.3.1).

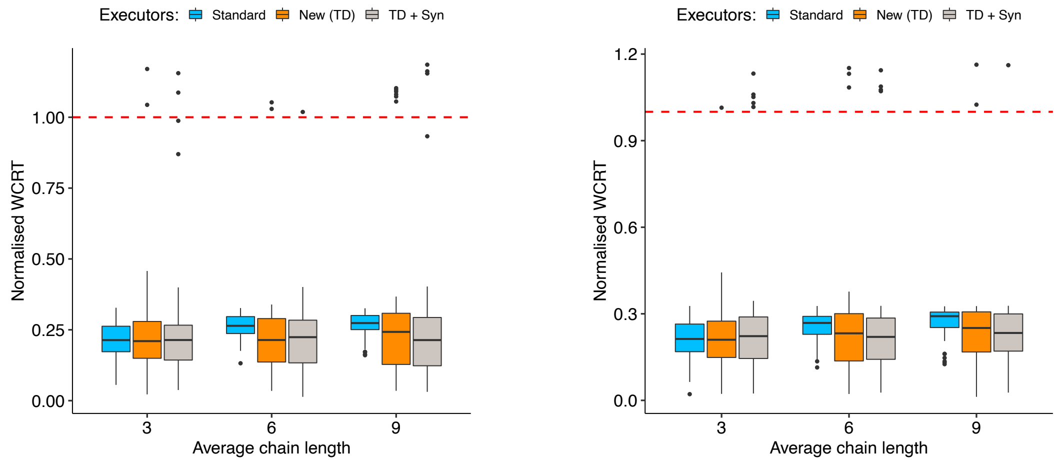
Custom timing. As in both previous experiments, the computation time required by each chain scales proportionally with the length of the chain. To maintain the required utilisation, the period is accordingly adjusted. In addition, the generated tests have the periods of their chains automatically adjusted in order to meet the schedulability requirements outlined in section 5.4 (namely, that chains synchronised together have the same period).

Synchronisation mechanism. The synchronisation mechanism chosen for this test is the ROS 2 `message_filters` package, using the approximate synchronisation policy. This policy attempts to best match input messages to a filter within an adjustable duration. The duration was set very high for this test (300s) to ensure that expiring messages would not be a cause of faults.

Experimental hypothesis

In this test, we expect to see lower response-times by applications running the synthesis algorithm. This is because the synthesis algorithm should propagate priorities down the branches of synchronisation nodes (message-filters) for higher priority chains. This should enable some high-priority chains that depend on lower priority chains to achieve smaller worst-case response times, as the slower dependent chain will be given priority as well. Both tests of New (TD) executor with and without synthesis are expected to perform better than the Standard.

Results



(a) **Normalised worst-case response-times (for test-applications with 15% synchronisation chance)**

(b) **Normalised worst-case response-times (for test-applications with 30% synchronisation chance)**

Figure 7.6: **Response for test-applications scheduled under all executors with 15% and 30% chance synchronisations**

Discussion

The results, shown in figure 7.6, demonstrate that under rate-monotonic priority-assignment, the priority synthesis algorithm delivers marginally improved time-predictability applications over simply running them on New (TD) at larger application chain lengths. When looking at the normalised worst-case response-times (see figure 7.6a and figure 7.6b), we observe that up until a chain length of around 6, the synthesis-algorithm appears to perform similar or worse to New (TD). At length 9, however, the median response times improve (see table 7.9)

Table 7.9: **Median normalised worst-case response-times for each executor setup by average chain length (30% synchronisation chance)**

Average chain length	3	6	9
Standard	0.213	0.268	0.292
New (TD)	0.210	0.232	0.251
New (TD) + Syn	0.223	0.220	0.234

The results demonstrate that a slight improvement in normalised worst-case response-time is achieved by applying the synthesis algorithm to ROS 2 test applications with a larger average chain length (> 6). This should prove practical to developers seeking to achieve smaller response-times on large applications.

7.4 Findings

7.4.1 Summary

We summarise the results of the evaluations below:

1. **Overhead of executor implementations.** (section 7.3.1) Comparing the overhead of the different executor implementations revealed that the Standard executor suffers from large outlier overhead spikes, regardless of the impact from publish-subscribe activity or utilisation level. We also observed that the Standard executor is more

sensitive to utilisation. At 60% utilisation, it experienced larger median overhead response-times that both new executor implementations, and lower median overhead response-times at 15% utilisation.

2. **Effect of utilisation on executors.** (section 7.3.2) When examining the effect of utilisation on the predictability of ROS 2 applications, we find that the applications experince deadline misses under the Standard executor at just 10% utilisation, while applications under the New (TD) executor do not experience any until 60% under a rate-monotonic priority assignment policy. We also find that the average mean normalised jitter across all utilisation levels is 286% larger for applications run with the Standard executor, than for applications run with the New (TD) executor, and 252% larger on Standard than for applications run with New (PC).

3. **Effect of chain length on executors.** (section 7.3.3) An investigation into the impact of chain length revealed that after a chain length of 6 (with proportionally scaled computational work and period), no applications run under either of the new executor implementations experienced any deadline misses under rate-monotonic priority assignment. Meanwhile, applications run with the Standard executor experienced more than 20 at absolute minimum across all lengths. In its best-case scenario, the mean normalised jitter experienced by test applications under the Standard executor is 189% that of New (TD) (at chain-length 2). On average across all chain lengths, it is 360% larger.

4. **Evaluating executor choices for high-priority time-sensitive callback chains.** (section 7.3.4) The impact of prioritising a chain in an application without rate-monotonic priority assignment revealed that across all chain lengths, the priority chain enjoys an average normalised worst-case response-time just 27% that of the same application run under Standard for the New (PC) executor and New (TD) executor. Meanwhile, the selected non-priority chain suffers from an average worst-case response time that is 17% and 15% larger across all chains when run under the New (PC) and New (TD) executors respectively. We also see that the worst case difference in mean normalised jitter is 135% for the non-priority chain, but 1300% for Standard when compared to the priority chain.

5. **Effect of priority synthesis algorithm.** The priority synthesis algorithm was applied to test-applications in which the length of the chains was varied between 3 and 9, with a step of 3. Two different synchronisation probabilities were applied (15%, and 30%), and the utilisation dropped to 30% to compensate for the increase that would occur when chains synchronised with one another. The Standard executor was pitted against the New (TD) executor, and the New (TD) executor with the synthesis algorithm. The results showed that the priority synthesis algorithm achieved small improvements in the normalised worst-case response-time at larger chain lengths, but performed similarly or worse at small chain lengths.

Chapter 8

Conclusion

We begin the conclusion of this thesis by summarising the research conducted chapter by chapter (section 8.0.1), along with the contributions. Then, we provide a discussion on the limitations and strengths of the work in section 8.1. Finally, we provide a set of future work in section 8.2.

8.0.1 Summary

To properly contextualise the contributions of this paper, it is necessary to revisit the research objectives first outlined in section 1.3. These are:

1. To identify and implement the necessary changes to the ROS language-specific client-facing libraries that improve the real-time capabilities of the ROS scheduling model. This entails enabling support for (1) the prioritisation of callbacks and (2) preemptive callback execution.

2. To explore how an improved scheduling model can be leveraged to create more predictable ROS applications by synthesising callback priorities for applications. The objective being that the synthesis scheme reduces the end-to-end response-time of time-sensitive callback chains.

In Chapter 2, we set the groundwork for improving the real-time capabilities of the ROS scheduling model by exploring the various communication patterns of ROS along with the execution model of its de-facto scheduler. Chapter 3 then provided motivational examples for how ROS applications could benefit from callback prioritisation, preemptive execution, and a priority assignment strategy. We also distilled the problem into a set of minimal, formal requirements upon which a solution can be built. Next, the problem was further explored in Chapter 4, where related work and material was reviewed and discussed. Following that, Chapter 5 presents our solution in the form of two new executor implementations and a priority synthesis algorithm. A feasibility test for individual callback chains based on the graph model is also presented, along with a schedulability test for applications with harmonic chains. Chapter 6 introduces our novel testing framework, which enables the rapid generation and assessment of ROS 2 applications. Finally, Chapter 7 demonstrates how both new executor implementations and the priority synthesis algorithm can improve the predictability of ROS 2 applications in terms of response-time, jitter, and deadline misses.

8.0.2 Contributions

Our contributions in this work are threefold:

1. Two ROS 2 executor implementations (named Producer-consumer and Thread-dispatch), supporting **callback prioritisation** and **preemptive callback execution**. Both of which are qualities not present in the standard solution.

2. A priority synthesis algorithm which automatically **derives priority values for callbacks** based on a given set of callback chains and chain priorities, with handling for shared callbacks and synchronisations.

3. A feasibility test which determines if any given callback chain cannot complete within its deadline, operating on the graph model of section 3.3.1.
4. A schedulability test for ROS 2 applications which have chains with harmonic periods, enabled by reducing the ROS graph model to a task set under fixed-priority preemptive scheduling.
5. A toolchain that enables the **automatic generation** of ROS 2 applications for rapid testing and development.

An assessment of whether the research objectives were satisified by the contributions is entirely justified by the results. In terms of the results, we find that the new executor implementations succeed in improving the timing-predictability of ROS 2 applications in a number of aspects. When comparing the new executor implementations to the standard, we found that the New (TD) executor was able to run all given test-applications with up to 60% utilisation without a single deadline miss. By contrast, the standard executor suffered deadline misses at just 10% utilisation. In terms of jitter, we found that under rate-monotonic scheduling, test-applications across a series of chain lengths experienced, *at minimum*, 189% the amount of mean normalised jitter using the standard executor than the New (TD) executor. Next, an investigation into high priority chains revealed that the new executor implementations are able to deliver smaller worst-case response times at relatively modest cost to the system. For example, we saw that the worst case difference for mean normalised jitter between the New (PC) executor and the standard is 135% for a randomly selected *non-priority chain*, but *1300%* for the standard when compared to the priority chain. Finally, we see that for systems with longer callback chains, the priority synthesis algorithm can improve median response times up to 7% at chain lengths of 9 for 30% synchronisation chance.

8.1 Discussions

The research conducted in this thesis has not come without its share of critical limitations. We enumerate these below:

1. We have not evaluated our solution on systems with a very large number of chains, callbacks, or executors, as we have focused on test-cases that resembled those covered in the case studies of section 7.1.2. Consequentially, the results are not generalizable to systems with 10s or 100s of callback chains or executors. Likewise, consideration was not given to how data dependencies and message sizes affect the performance of executors.
2. The executor implementations developed are based entirely on *static* priorities. This means that scheduling algorithms that depend on dynamic priorities (e.g earliest deadline first (EDF)) are not supported. This is due to the distributed nature of ROS 2, which complicates the communication of priority chances across scheduler instances (executors).
3. Despite the test cases being inspired by real-world applications, there exists no real-world proof of concept for the contributions. The testing conducted was on synthetically generated time-driven ROS 2 applications. This will not always compare to real-world applications, which must deal with the occurrence of sporradic events.
4. Despite an extensive look into the communication patterns of ROS, there remain many unknowns in regards to how other components of ROS (e.g. DDS and the underlying OS) contribute to the predictability of applications. These factors can be investigated as future work, as they require more elaborate operating systems knowledge
5. The focus on the publish-subscribe paradigm means that the new executor implementations do not support other (newer) mechanism of ROS. This means that ROS 2 constructs such as services cannot be used, in turn limiting the applicability of the solution.

8.2 Future Work

A number of angles exist for future work with respect to the contributions made. We enumerate some of these below:

1. **Data-dependencies.** In addition to providing a set of standard message formats, ROS enables developers to create their own messages through a convenient file format in combination with the package system. This means developers may rapidly integrate custom messages in their applications. The effect of how heterogenous message sizes can affect predictability of callback chains is not explored in this work. Furthermore, developers may use the publish-subscribe communication mechanism of ROS to access data logically separated into other nodes within the same process or even that of a different process. In a single-threaded executor model, such as that present by default in ROS, simultaneous accesses to shared data in a node is automatically sequentialized. However, the new executor models raise a concern of race conditions, as callbacks have the ability to preempt one another.

Bibliography

[1] Ian Chen Alejandro Hernández Cordero, Tully Foote. Drone demo in ros 2/gazebo/rviz 2. https://github.com/osrf/drone_demo, 2019.

[2] N Audsley, A Burns, M Richardson, K Tindell, and A J Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[3] Marya Belanger. Ros 2 client interfaces (client libraries). https://index.ros.org/doc/ros2/Concepts/About-Client-Interfaces/, 19-02-2021.

[4] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, may 2005.

[5] Brian Gerkey. Ros, the robot operating system, is growing faster than ever, celebrates 8 years. https://spectrum.ieee.org, 2015. Last accessed: Feb. 15, 2021.

[6] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B. Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] Yorick De Bock, Sebastian Altmeyer, Thomas Huybrechts, Jan Broeckhove, and Peter Hellinckx. Task-set generator for schedulability analysis using the taclebench benchmark suite. *ACM SIGBED Review*, 15:22–28, 03 2018.

[8] Saeid Dehnavi, Dip Goswami, Martijn Koedam, Andrew Nelson, and Kees Goossens. Modeling, implementation, and analysis of xrce-dds applications in distributed multiprocessor real-time embedded systems. 02 2021.

[9] Open Source Robotics Foundation. Ros client library api stack, 08-10-2016.

[10] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, may 1976.

[11] Brian Gerkey. Why ros 2? *ROS 2 Design*.

[12] Carlos San Vicente Gutiérrez, Lander Usategui San Juan, Irati Zamalloa Ugarte, and Víctor Mayoral Vilches. Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications.

[13] Vijay Pradeep Josh Faust. Package summary. http://library.isr.ist.utl.pt/docs/roswiki/message_filters.html#Policy-Based_Synchronizer_.5BROS_1.1.2B-.5D, 19-11-2021.

[14] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[15] The Linux man-pages project. *sched(7) - Linux manual page*, August 2019. release 5.10.

[16] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, jan 1973.

[17] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. pages 1–10, 10 2016.

[18] micro ROS. Real-time executor. 2020.

[19] M. S. Morshed, S. Meeran, and A. Jain. A state-of-the-art review of job-shop scheduling techniques. *International Journal of Applied Operational Research - An Open Access Journal*, 7:0–0, 2017.

[20] Nathaniel Hamilton Feiyang Cai Tim Darrah Shreyas Ramakrishna Ayana Wild Patrick Musau, Diego Manzanas Lopez. The flash van. https://github.com/verivital/F1TenthVanderbilt, 2019.

[21] Michael Pinedo. *Scheduling : theory, algorithms, and systems*. Springer, New York London, 2008.

[22] Bernard Roy and B Sussmann. Les problemes d'ordonnancement avec contraintes disjonctives. *Note ds*, 9, 1964.

[23] Yukihiro Saito, Futoshi Sato, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. ROSCH:real-time scheduling framework for ROS. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, aug 2018.

[24] Alex Simpkins. Real-time control in robotic systems. In *Robotic Systems - Applications, Control and Programming*. InTech, feb 2012.

[25] Emin Gun Sirer and Robbert Van Renesse. Synchronization classic problems, July 2013.

[26] Hideki Takase, Tomoya Mori, Kazuyoshi Takagi, and Naofumi Takagi. mROS. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies - HEART 2019*. ACM Press, 2019.

[27] Andrew Tanenbaum. *Operating systems : design and implementation*. Prentice-Hall, Englewood Cliffs, N.J, 1987.

[28] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingson Lv, Qungxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ros 2 executors ROS. In *2020 IEEE 41st IEEE Real-Time Systems Symposium (RTSS)*. IEEE, dec 2020.

[29] Thomas Gubler. Ng drones. https://wiki.ros.org/ng_drones, 2019. Last accessed: Feb. 17, 2021.

[30] William Woodall. Glossary. https://docs.ros2.org/foxy/glossary.html#term-client_library, 08-10-2016.

Appendix A

APPENDIX TITLE

A.1 Executor Implementations

A.1.1 Single-threaded executor

```
void
SingleThreadedExecutor::spin_some(std::chrono::nanoseconds max_duration)
{
  // Fetch and store current timestamp
  auto start = std::chrono::steady_clock::now();

#ifdef DEBUG
  cpu_set_t cpu_set;
  std::cout << std::string("\033[1;33m") + "SingleThreadedExecutor: Pinned to
    cores {0}" +
    std::string("\033[0m\n");

  CPU_ZERO(&cpu_set);
  CPU_SET(0, &cpu_set);
  if (0 != pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpu_set)
    ) {
    throw std::runtime_error(std::string("pthread_setaffinity_np: ") +
      std::string(std::strerror(errno)));
  }

#endif

  // Create callback
  auto callback_should_still_spin = [max_duration, start]()
  {
    // Case: Spin forever
    if (std::chrono::nanoseconds(0) == max_duration) {
      return true;
    }

    // Case: Spin for duration
    return ((std::chrono::steady_clock::now() - start) < max_duration);
  };

  // Check if already spinning
  if (true == spinning.exchange(true)) {
    throw std::runtime_error("spin_some() called while already spinning!");
  }

  RCLCPP_SCOPE_EXIT(this->spinning.store(false); );

  while (rclcpp::ok(context_) && (true == spinning.load()) &&
    true == callback_should_still_spin())
  {
    AnyExecutable any_exec;

    // Non-preemptable call
#ifdef PROFILE
```

```
47      long long overhead_ns;
48      if (get_next_executable(any_exec, &overhead_ns)) {
49  #else
50      if (get_next_executable(any_exec)) {
51  #endif
52
53  #ifdef PROFILE
54        g_n_processed_callbacks_++;
55        g_cumulative_processing_time_ns_ += overhead_ns;
56  #endif
57        execute_any_executable(any_exec);
58      }
59    }
60
61  #ifdef PROFILE
62    long long avg_processing_time = g_cumulative_processing_time_ns_ /
        g_n_processed_callbacks_;
63    syslog(LOG_INFO, "{.value: %lld, .mode: 0}", avg_processing_time);
64  #endif
```

Listing A.1: **Single-threaded standard executor**

A.1.2 Thread-dispatch executor

```
1  // todo
```

Listing A.2: **Thread dispatch executor**

A.1.3 Producer-consumer executor

```
1  void PreemptivePriorityExecutor::multiplex (std::chrono::nanoseconds
       max_duration,
2    std::vector<Consumer<Callback_Ptr> *> *consumers_p)
3  {
4    std::vector<AnyExecutable> ready_executables;
5    int max_allowed_priority = d_priority_range.u_bound -
6      d_priority_range.l_bound - 1;
7
8    // Mark start
9    auto start = std::chrono::steady_clock::now();
10
11   // Create callback for checking time
12   auto should_still_spin = [max_duration, start]() {
13     if (std::chrono::nanoseconds(0) == max_duration) return true;
14     return ((std::chrono::steady_clock::now() - start) < max_duration);
15   };
16
17   // Spin indefinitely, or for a duration if specified
18   while (rclcpp::ok(this->context_) && spinning.load() &&
19     true == should_still_spin())
20   {
21     // Wait for work up to a timeout
22     wait_for_work(std::chrono::nanoseconds(100000000));
23
24     // Check if timeout occurred or should stop spinning
25     if (false == spinning.load() || !should_still_spin()) {
26       break;
27     }
28
29     // Clear and fetch new ready executables
30 #ifdef PROFILE
31     long long processing_start_time_ns = get_timestamp_ns();
32 #endif
33     ready_executables.clear();
34     memory_strategy_->get_all_ready_timers(&ready_executables, weak_nodes_);
35     memory_strategy_->get_all_ready_subscriptions(&ready_executables,
       weak_nodes_);
36
37     // Sort all ready executables into the correct consumer queues
```

```
38      for (auto e : ready_executables) {
39        Callback_Ptr item = executable_to_callback(e);
40
41        // If nullptr, then it was already handled
42        if (item == nullptr) {
43          continue;
44        }
45
46        // Check the priority value
47        if (e.callback_priority < 0 || e.callback_priority >
48          max_allowed_priority) {
49          throw std::runtime_error("Executable had invalid priority: " +
50            std::to_string(e.callback_priority) + ", only allowed [0," +
51            std::to_string(max_allowed_priority) + ")");
52        }
53
54        // Extract the consumer for the given callback priority
55        Consumer<Callback_Ptr> *consumer = consumers_p->at(e.callback_priority);
56        auto m_p  = consumer->mutex();
57        auto cv_p = consumer->condition_variable();
58        {
59          std::lock_guard<std::mutex> temp_lock(*m_p);
60          consumer->enqueue(item);
61 #ifdef DEBUG
62          std::cout << "[Executor]: enqueued " << item->description() << std::
      endl;
63 #endif
64        }
65
66        // Notify the work thread (lost if not sleeping)
67        cv_p->notify_one();
68      }
69
70 #ifdef PROFILE
71      long long processing_time_duration_ns = get_timestamp_ns() -
      processing_start_time_ns;
72      if (ready_executables.size() > 0) {
73        g_n_processed_callbacks += ready_executables.size();
74        g_cumulative_processing_time_ns += processing_time_duration_ns;
75      }
76 #endif
77    }
78
79 #ifdef PROFILE
80    long long avg_processing_time = g_cumulative_processing_time_ns /
      g_n_processed_callbacks;
81    syslog(LOG_INFO, "{.value: %lld, .mode: 1}", avg_processing_time);
82 #endif
```

Listing A.3: **Producer-consumer executor**

A.2 Evaluation framework

A.2.1 Merge/synchronisation algorithm

Algorithm 3 Computation of merges/synchronisations to perform

1: **procedure** RandomOperation(*callbacks*, p_{merge}) ▷ Computes a series of random operations to perform between callbacks (by index)

2: ▷ Init action set
3: $actions \leftarrow \emptyset$

4: ▷ Create array of indices for the callback set
5: $n \leftarrow$ length of the ordered set *callbacks*
6: $ns \leftarrow$ ordered set of positive integers across range $[0, n)$

7: ▷ Loop until no more merges possible
8: $stop \leftarrow FALSE$
9: **while** size of $ns > 0 \wedge stop = FALSE$ **do**
10: $x \leftarrow$ length of the ordered set ns
11: $chances \leftarrow (x * (x - 1))/2$
12: $wasAction \leftarrow FALSE$
13: **while** $chances > 0 \wedge wasAction = FALSE$ **do**
14: $r \leftarrow$ random $\mathbb{R}$ between 0.0 and 1.0
15: **if** $r < p_{merge}$ **then**
16: $i \leftarrow$ random valid index across ns
17: $j \leftarrow$ random valid index across ns
18: **if** $i = j$ **then**
19: Rerun current loop iteration
20: **end if**

21: ▷ Add an action between $ns[i]$ and $ns[j]$
22: $actions \leftarrow actions \cup \{(ns[i], ns[j])\}$
23: $ns \leftarrow ns \setminus \{ns[i]\}$

24: ▷ Stop loop since need to recompute chances
25: $wasAction \leftarrow TRUE$
26: **else**
27: $chances \leftarrow (chances - 1)$
28: **end if**
29: **end while**

30: ▷ If all chances exhausted, then stop
31: **if** $chances = 0$ **then**
32: $stop \leftarrow TRUE$
33: **end if**
34: **end while**
35: **end procedure**

A.2.2 Generated assets

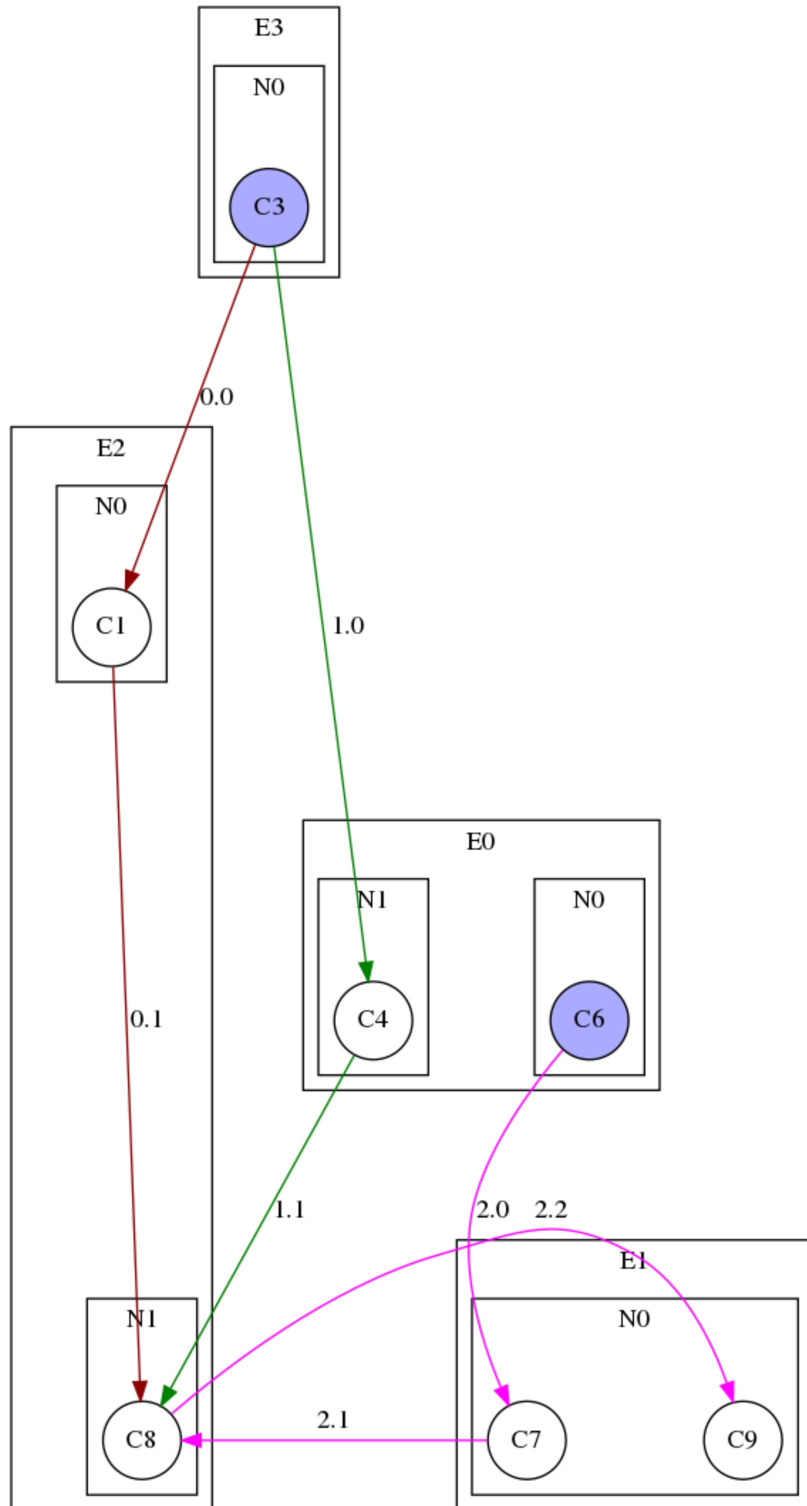


Figure A.1: **Generated application asset. Edges are labeled in form $X.Y$, where X is the chain ID, and Y the number of hops in that chain as of that edge. Callbacks are indicated with circles, and have a C prefix. Enclosing boxes labeled prefixed with N represent nodes, while outermost boxes labeled with E represent executors**

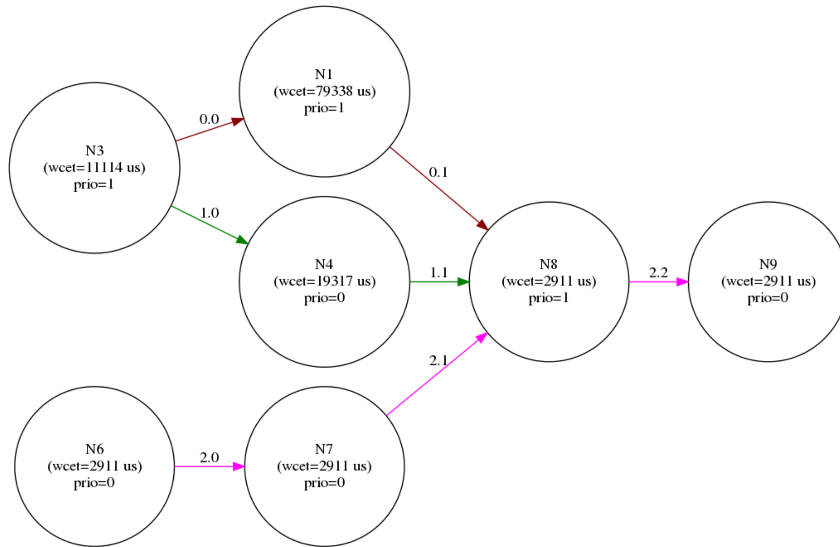


Figure A.2: **Generated chains graph asset Edges are labeled in form $X.Y$, where X is the chain ID, and Y the number of hops in that chain as of that edge. Callbacks are indicated with circles, and have a C prefix. Callbacks have their priority and WCET of each node labeled within them**