NeuROS

A Distributed Integration Framework for Heterogeneous Neurorobotics Systems

By

Perry, Lee

MSc Robotics Dissertation

Department of Engineering Mathematics
UNIVERSITY OF BRISTOL
&
Department of Engineering Design and Mathematics
UNIVERSITY OF THE WEST OF ENGLAND

A MSc dissertation submitted to the University of Bristol and the University of the West of England in accordance with the requirements of the degree of MASTER OF SCIENCE IN ROBOTICS in the Faculty of Engineering.

September 4, 2024

Declaration of own work

I declare that the work in this MSc dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Lee Perry

September 4, 2024

Acknowledgement

First and foremost, I would like to thank Martin Pearson and Thomas Knowles for their highly valued guidance and support. It was an absolute pleasure to work with the both of you and I hope this project proves to be a useful tool for their future endeavours! Without Tom's help debugging my NRP installation, I might never have properly completed this project. Secondly, I would like to thank my tutor and mentor Paul O'Dowd for his eagerness to read my work and offer such incredibly insightful feedback. I have learnt a lot from you. Thirdly, I am hugely grateful to my employer for allowing me the part time schedule I required to attend this course. Juggling these things has not been easy and I appreciate the trust that was placed in me to do so. Finally, to my family and friends, who have all had to put up with me working all hours of night and day while I chip away at this project.

Abstract

Number of words in the dissertation: 15,000 words.

Artificial Intelligence (AI) is a revolutionary technology that is transforming many fields of research. Neurorobotics specifically leverages developments within AI in order to build more intelligent and capable robots. This approach has been proven highly successful across a broad and diverse range of sub-domains, including localisation, control, grasping, vision, scene understanding and planning. Physics simulators have become the industry standard tool for training and testing these systems at scale. Integrating these components together is often a complex task, both in terms of managing rich software requirements and orchestrating the data flows between them.

This dissertation presents NeuROS, an *entirely new framework* that builds upon the knowledge gained from existing projects, combining their strengths and addressing their weaknesses. It delivers a novel, highly flexible, ROS2-based framework for the integration of neurorobotics components, lowering the barrier of entry for neurorobotics research. NeuROS has been designed and implemented from the ground up with the requirements of neurorobotics developers in mind. Support for several industry standard components is included out-of-the-box, offering high degrees of reuse and extendability. Containerisation is utilised for the management and deployment of software dependencies. Native graphical interfaces are employed to increase usability while the open source back-end offers extreme flexibility and extensibility.

The flexibility provided by NeuROS is so high that several different orchestration approaches are possible within the framework. This research therefore additionally provides a comparative analysis using various performance metrics to investigate the relative merits of each orchestration approach. Finally, a comparison of these orchestration approaches is compared against one of the most well established alternatives in the field, and some existing neurorobotics research is partially reproduced. The flexibility provided by NeuROS is shown to offer potential performance enhancements over what is currently on offer, by way of per-connection customisation.

Contents

1 Introduction

1.1 Motivation & Problem Statement

The ongoing revolution in Artificial Intelligence (AI) is producing perhaps the most important general-purpose technologies of our era [1]. Continually improving AI systems have presented new opportunities across many subdomains within the field of robotics, where such AI-enabled robot systems are becoming increasingly popular, automated and capable [2]. Many of these systems are biologically inspired [3], and often based on neural networks [4] with domain-specific modifications widening their performance and breadth of applicability. To date, neural networks have been used for robot grasping [5], object detection [6] [7], scene understanding [8], simultaneous localisation and mapping [9], global path planning [10], robot control [11] and much more. The term *neurorobotics* [12] has been coined for this combination of embedded biologically inspired neural networks within robots.

Such neural networks typically require training against large amounts of input data, which are too costly and impractical to obtain from the real world. Instead, it has become increasingly popular to use *synthetic data* [13], generated artificially via alternative means. Within neurorobotics, this typically involves the use of physics simulators [14] for the generation of realistic sensor inputs, such as camera images and Inertial Measurement Unit (IMU) readings. Furthermore, it is far more convenient for researchers to test and debug their robots in simulated environments, rather than in the real world. Not only is this practice quicker, but it also offers greater variability in the range of environments that can be investigated. Thus, an efficient neurorobotic software development process is often based on biologically inspired artificial neural networks providing the cognition and physics simulators providing the sensing and actuation.

It is apparent from previous research [15] that such complete neurorobotic systems typically necessitate very high complexity, comprising multiple specialised and sophisticated heterogeneous software components. Since much of the systems behaviour derives from a closed perception-

action loop between the robot and it's environment, these systems often require highly complex orchestration. Orchestration of such systems is often performed at various levels of abstraction, with concurrent execution, arbitrarily definable topology and flexible message content. This poses a significant problem as there are very few options available for the integration of such complexity [16] [17] [18]. Of the solutions that do exist, each meets only a subset of the aforementioned requirements. The restrictions they impose include inflexible architectures, lack of integration with industry standard tools, fixed messaging intervals, limited message data types, restrictive interfaces and lack of standardisation. Several such frameworks will be discussed in more detail in Section 2.2, however there is currently no single framework supporting all features required to make it entirely fit for purpose. Notable research by Calvo-Fullana et al [19] and Rovida et al [20] have produced similar generalised frameworks for other domains using ROS [21] as the underlying message transport layer, in order to address and resolve similar integration problems. Bentaleb O. have presented [22] the successful adoption of containerisation approaches, such as Docker [23], for the packaging, deployment and maintenance of complex software requirements.

This dissertation presents NeuROS, an entirely new framework as a novel contribution to the field. The design and implementation of this framework enables unrivalled functionality, especially within the domains of flexible orchestration and containerisation. It will leverage several modern robotics software libraries that compliment such goals. In doing so, this work will provide a useful tool to the neurorobotics community, with improved orchestration and management of highly complex neurorobotics systems.

1.2 Aims & Objectives

Aims

This work aims to answer the following questions:

- Is it possible to build a neurorobotics integration framework with fully distributed orchestration, allowing for more flexible orchestration, including hybrid approaches?
- What are the impacts of these various different orchestration approaches on performance?

- Can containerisation be used to encapsulate the software requirements of individual nodes, while still maintaining a high level of performance and integration?

Objectives

- Design and build a new neurorobotic integration framework that supports:
  - Highly flexible architectures, with minimal restrictions on component network topology and no mandatory centralised controller.
  - A broad range of orchestration approaches, inspired both from existing literature and any additional novel ideas that show promise.
  - Support for industry standard neurorobotics tools, such as Gazebo and NEST.
  - Encapsulation of software dependencies for individual components via containerisation, offering easy extension and a high level of modularity.
- Conduct a comparative performance analysis:
  - Implement several example projects based upon at least one representative neurorobotics use case, which showcase a broad range of orchestration approaches.
  - Implement an equivalent project in a leading competitor framework.
  - Conduct a comparative performance analysis covering at minimum: System load (CPU utilisation, memory consumption, etc), packet delivery rates and execution speed.
  - Use the performance analysis to inform any potential implementation of subsequent performance enhancements, and re-evaluate the resulting system.

1.3 Contributions

This work delivers an entirely new framework, NeuROS, with the following novel features:

- A building block for *distributed orchestration* techniques, as presented in Section 3.6.1.
- Various orchestration techniques offering customisable trade-offs between speed and accuracy, in Section 3.9.2. With an associated comparative performance analysis, in Chapter 4.
- Individual node containerisation, for ease of packaging and deployment.

2 Literature Review

This section will begin by reviewing some existing neurorobotics research, which will be used as a representative test case for our later performance analysis. After which we will discuss some of the existing neurorobotics integration frameworks, contrasting their features with those delivered by NeuROS. The goal of these comparisons is to learn from their successes and remedy any shortcomings as much as possible. Next, we will take a more detailed look at some of the neurorobotics software components that might typically be integrated together, such as specific physics simulators and neural network model libraries. Then we go on to explore some of the ways these components can be integrated together, forming the nodes of a network connected together by a messaging transport layer. Finally, we will discuss containerisation, for the purpose of encapsulating the individual software requirements of each of these nodes.

2.1 Typical Use Cases

In order to understand the exact requirements of the neurorobotics domain, it is beneficial to review a typical use case and try to understand how the requirements placed on the integration framework.

2.1.1 Integrating Spiking Neural Networks and Deep Learning Algorithms on the Neurorobotics Platform

Prior research [15] into the integration of Spiking Neural Networks (SNNs) and Deep Learning Algorithms on the Neurorobotics Platform (NRP) has demonstrated the need for more effective neuroscience modelling integration frameworks, the limitations of which are discussed the following paragraph. This research was aimed at estimating the orientation of a rodent's head within a simulated environment. The baseline approach used purely inertial odometry from an IMU sensor to calculate the rotation, which was shown to produce drift and accumulate error over time, due to the use of purely idiothetic information. The researchers were able to eliminate this error by in-

troducing a Predictive Coding Network (PCN) to estimate the head rotation from allothetic visual information. Furthermore, the PCN could also reliably predict periods of high error associated with the rotation estimate. Future work was suggested, where this error prediction could be utilised as a trigger for learning. Both odometry calculation and rotation prediction were passed into a SNN, with the most active neuron over a 40ms interval being chosen as the most likely rotational distance. All of the components in this research, including the simulator, PCN, odometry calculation and SNN, were integrated together using the NRP.

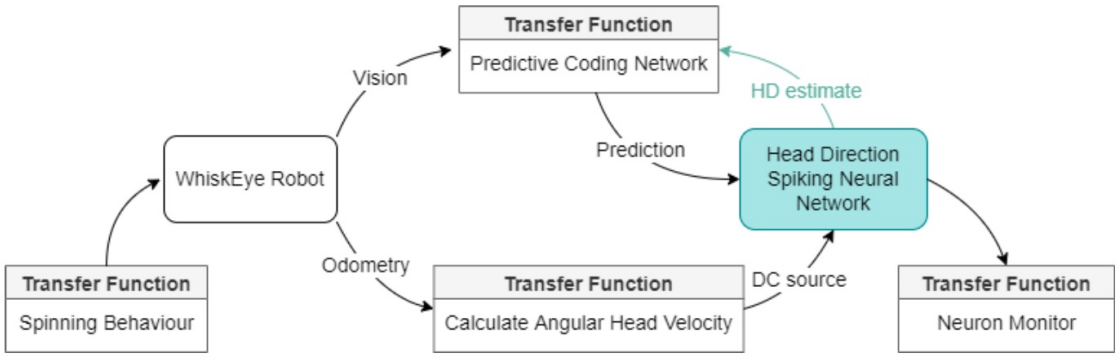


Figure 2.1: Typical Use Case Architecture [15]

The limitations of this existing NRP framework were described as including: The lack of software separation requiring bespoke Python virtual environments to be implemented by the user, a very limited messaging architecture forcing some communication to be performed via global variables, a lack of documentation, and the introduction of custom Gazebo GUIs which forced users out of their familiar workflows. NeuROS aims to address each of these major shortcomings by providing flexible architecture, containerised nodes, fully documented code and native GUIs. The simulation was also shown to run in slower than real world time and the researchers suggested potential improvements for offloading some of the processing to the GPU. This existing research provides a concrete example of the types of components that need to be supported by NeuROS, typical data flows (see Figure 2.1), and some known shortcomings of existing solutions. As such, design considerations and performance analysis for NeuROS were based around this specific use-case, and this research task was adopted as an representative test case.

2.2 Existing Frameworks

2.2.1 The Neurorobotics Platform

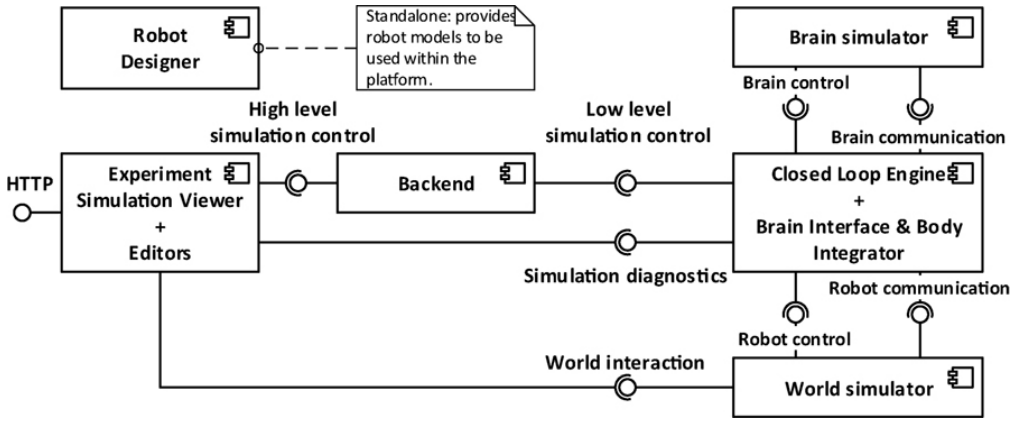


Figure 2.2: The NRP Architecture [24]

The Neurorobotics Platform (NRP) [25] [24] is a platform developed by the Human Brain Project [26] for the integration of simulated physics and brain models. The platform aims for ease of use, with users with little or no prior neuroscience or robotics knowledge. Gazebo [27] is provided as the default physics simulator. In contrast with NeuROS, direct programmatic step control is not provided to the user. Instead, the NRP orchestrates components via the Closed Loop Controller, which executes brain and physics simulations in parallel for a specified time increment. Note that the real-time duration of time may vary, as the NRP will wait for both tasks to complete before starting a new update. Having a centralised controller necessitates a relatively fixed hub-and-spoke architecture. In contrast to this NeuROS uses distributed messaging, meaning it does not require either a central controller or a fixed time step, and can thus synchronise between components with arbitrary topology.

Currently the NRP only really connects NEST with Gazebo, extending further requires a significant amount of work. NeuROS however, will provide NEST and TensorFlow out-of-the-box, with easy extensibility of any other software component through the use of containers. The NRP uses Transfer Functions to communicate between the physics and brain simulations, and convenient Python decorators are provided to ease implementation. However these Transfer Functions hard-code their related ROS topics, which prevents re-usability. Instead, NeuROS retains the Python

decorator approach but separates the communication layer from the user plugins as much as possible, in order to promote reuse of these components.

The NRP provides many custom web-based interfaces on top of existing tools, such as the Experiment Viewer instead of the native Gazebo interface. NeuROS aims to remain more lightweight, providing access to the native interfaces and focusing more narrowly on the integration and modularisation aspect. There is no additional logical separation of software dependencies provided by the NRP, meaning that mutually incompatible software components cannot be used in a single experiment. Conversely, NeuROS solves this problem through the user of containers, again providing further modularity and simplifying deployment.

2.2.2 BRAHMS

BRAHMS (not an acronym) [17] is a highly generalised and flexible framework for the integration of biological computational models. It is motivated by the requirement for large and complex end-to-end biological simulation, for both understanding, modelling and potential use as bio-mimetic robot controllers. There is a big emphasis on code sharing and collaboration, which reduces the time, cost and expertise required to develop such large and complex systems. This is a feature that NeuROS takes great inspiration from, promoting modularity and reuse wherever possible. BRAHMS is configured centrally via SystemML, which is useful for a human reader to easily understand the complete system structure. NeuROS adopts this approach but instead defines projects in JSON, which allows for an additional layer of error-checking via JSON schema validation, utilising commonly available Python libraries [28]. Whereas initialisation in BRAHMS is an iterative procedure, where each process may be called repeatedly, NeuROS leverages information from this central configuration to synchronise initialisation between nodes. BRAHMS is not only centrally configured, but orchestration is also centrally controlled with updates scheduled at regular tick intervals. In contrast NeuROS offers decentralised execution, which allows for more complex architectures and subsystem synchronisation capabilities. Whereas BRAHMS is purely fully deterministic, NeuROS offers a variety of synchronisation strategies, allowing for greater flexibility and customisable trade-offs between speed and strict accuracy. Finally, BRAHMS integrates Webots [29] for physical simulation, but NeuROS opts for the more widely used Gazebo simulator, offering greater collaboration potential and easier project portability.

2.2.3 Ikaros

Ikaros [18] is an open infrastructure for system-level cognitive modelling, including databases, computational models and functional data. It bundles a large number of modular components, which provide an effective balance between flexibility and reuse. This is an aspect that NeuROS borrows from, applying the same methodology to component design. Components within Ikaros are purely for cognitive computation, with no direct support for physics simulators. For this purpose it has been well proven in the field, with over 40 publications using it for both cognitive modelling [30] and robot control [31]. This is in clear contrast to NeuROS which aims to model both the brain, robot body and the physical environment. All components in Ikaros implement their inputs and outputs as matrices of floats, which makes it very well suited to artificial networks. However, such a low level approach makes it somewhat less suitable for tasks that require symbolic processing rather than numerical computation. This was an intentional design choice to make it fast for real-time robot control, and also to make Ikaros more portable to a range of different hardware. In contrast, NeuROS takes a slightly higher level approach, with a number of ROS2 and Gazebo-defined message types that have specific semantic meaning e.g. camera images or IMU data. In this regard the two frameworks target slightly different use-cases. Ikraos is tick-based, with a centralised controller known as the "kernel" responsible for scheduling and passing data between components. NeuROS is conversely highly distributed with no central controller and direct node-to-node data transfer.

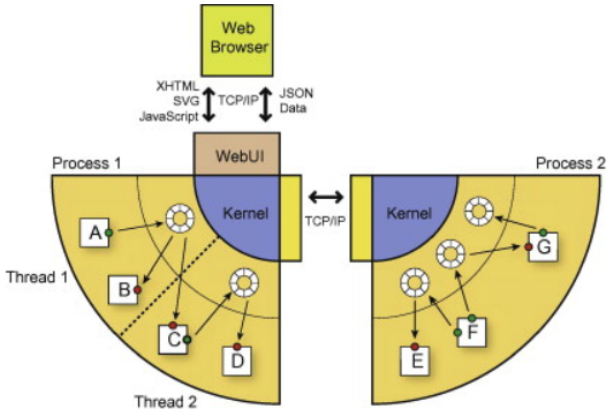


Figure 2.3: Ikaros Architecture [18]

Since Ikaros requires components to be written in C++ and statically linked into the kernel executable, any change to the code will require a recompilation and re-linking. By implementing

NeuROS in Python we avoid this extra development step at the cost of runtime performance. In general, Ikaros is an inspiring framework, though slightly lower-level than the use case NeuROS aims to target. There are certainly aspects of Ikaros, such as runtime speed, with which NeuROS will not be able to compete. Nonetheless there are several features in NeuROS that Ikaros lacks, such as more flexible message content and integrated physics simulation.

2.2.4 ROS-NetSim

ROS-NetSim [19] is an architecture for the synchronised execution of both physics and network simulators, concurrently via the use of custom synchronisation messages. It advocates for many important design considerations which NeuROS aims to emulate. Firstly, there is a strong emphasis on being lightweight and modular. In particular, ROS-NetSim is completely transparent to each of the integrated ROS nodes, as shown in Figure 2.4. This means that users can integrate their existing ROS nodes without the need for modification, thus promoting code reuse.

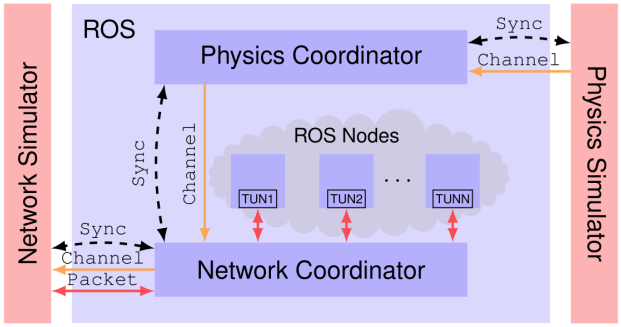


Figure 2.4: ROS-NetSim Architecture [19]

Secondly, ROS-NetSim aims for an architecture that is agnostic to the choice of network or physics simulator. Again, this promotes flexibility and modularity, and NeuROS additionally adopts this aim. It is notable in this regard that each physics simulator requires a small piece of additional code to integrate it into ROS-NetSim. Since there is no existing universally common interface among physics simulators, this is a necessity that NeuROS will also require.

2.2.5 SkiROS

SkiROS [20] is a highly modular system for integrating a semantic database with autonomous robots via ROS, in order to execute complex skill-based tasks. The rough architecture is shown in

Figure 2.5. This project attempts to make full use of the *network effect*, where lots of developers are able to work together as the platform library grows. This is made possible through a strong design ethos of providing building-blocks which developers can then utilise to build higher-order processes. Custom plugins can be implemented in order to provide additional skills, action primitives, world state conditions, discrete reasoners and task planners. A custom GUI allows non-expert users to visually build tasks, inspect the physical and abstract interactions, and monitor sensor data in real time.

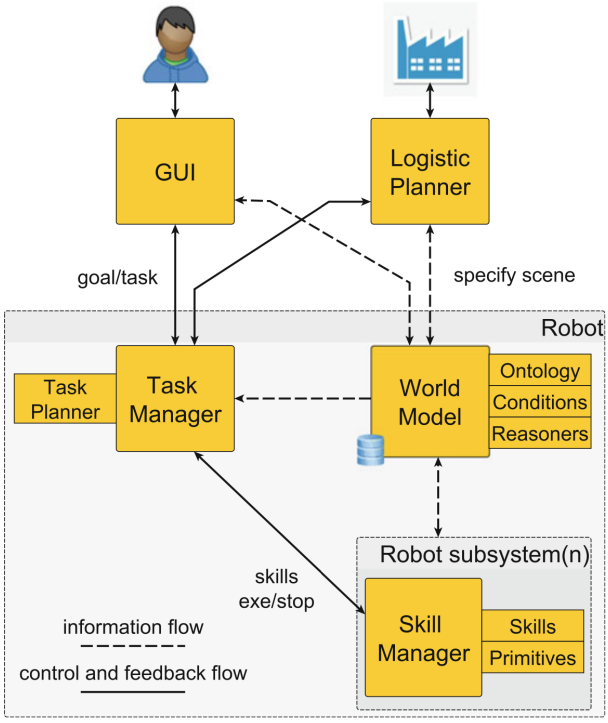


Figure 2.5: SkiROS Architecture [20] (where squares represent ROS nodes and rectangles represent plugins)

NeuROS borrows heavily from many of these concepts. Providing off-the-shelf modules and primitives alongside user-definable custom implementations is extremely effective for providing both flexibility and ease-of-use. Therefore, NeuROS will also utilise this form of plugin-driven functionality. Code reuse and modularity should also be maximised wherever possible, in order to reduce the need to duplicate work and deliver the most functionality per unit of user input. Furthermore, the leveraging of GUIs to aid debugging through system transparency, is very likely to increase user friendliness and therefore also make widespread adoption more likely.

2.3 Simulation & Modelling

Simulating the environment is essential for providing realistic input to sensors and modelling the real-world impact of actuator commands. A large number of physics simulators exist for this purpose, with various advantages and disadvantages, from which a single one is selected. NeuROS will also need to perform neurological computation, and will therefore build upon several carefully evaluated and selected industry standard tools, which are reviewed as follows.

2.3.1 Gazebo

Gazebo [27] [32] has been chosen as the most widely used [33] physics simulator in the field of robotics. It has a plugin-based architecture, where support for specific physics engines, sensors and actuators can all be loaded dynamically at runtime. The simulated robot(s) are described by a URDF file which defines it's physical properties, including sensors, actuators and links. These attributes make it extremely modular and flexible, complimenting the ethos of NeuROS extremely well.

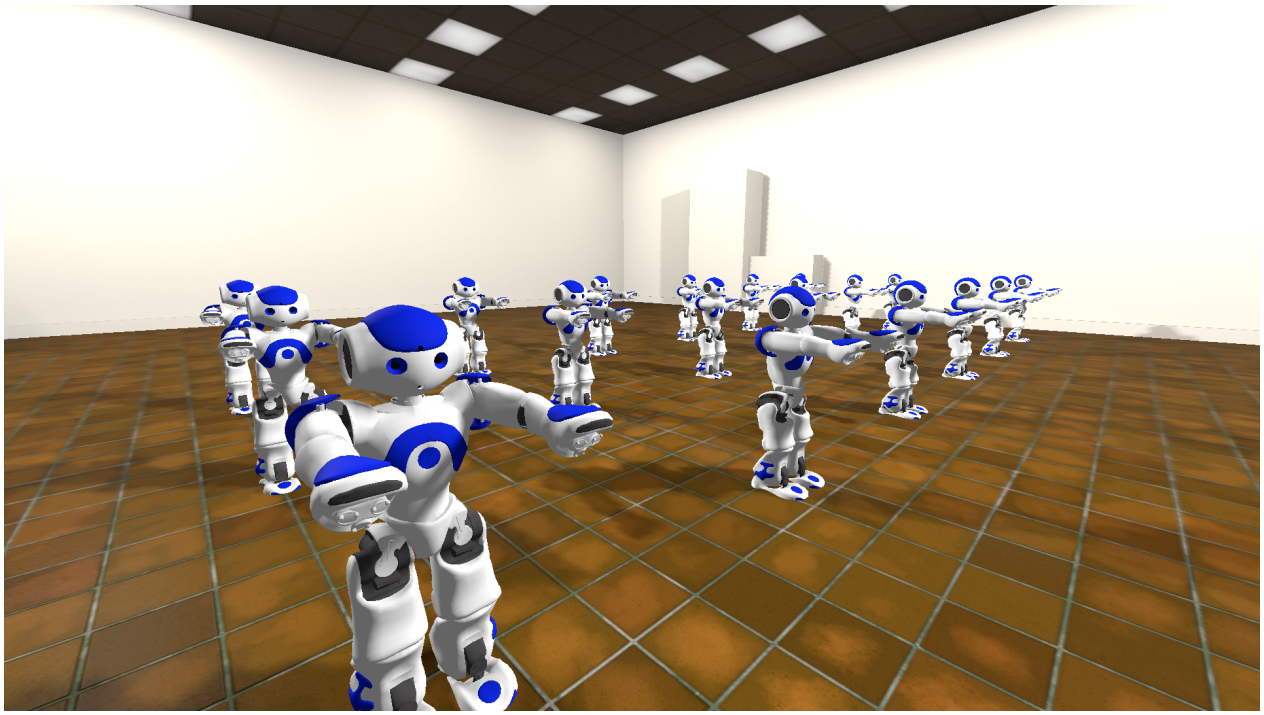


Figure 2.6: Gazebo [32]

The older legacy version of Gazebo is known as "Gazebo Classic", while the modern version

was previously known as "Ignition Gazebo" and later became "Gazebo Sim". Gazebo Sim brings with it a huge architectural rewrite which allows, among many other things, the ability to step through simulation time programmatically. In contrast with older client-server architectures this allows for stricter control, providing the opportunity to have deterministic and reproducible execution. This is leveraged by NeuROS in order to allow neurorobotics experiments to be repeated exactly as before. Such a feature is of significant importance to researchers for many reasons, including debugging and performance comparisons. For the purposes of this research "Gazebo" will be used to refer to Gazebo Sim unless otherwise stated.

2.3.2 Spiking Neural Networks

Spiking Neural Networks (SNN) [34] are a specific type of Artificial Neural Network, designed to model the function of biological neurons more closely. Each neuron in a SNN can receive both excitatory and inhibitory inputs, which contribute (either positively or negatively, respectfully) to the probability of that neuron spiking. It has been demonstrated [35] that not only can SNNs solve all of the same problems that non-spiking networks can, but that they do so more computationally efficiently. Since they are both more computationally efficient and closer to the biological function of neurons, they are particularly suitable to NeuROS.

NEural Simulation Tool (NEST) [36] is a widely used tool for building SNNs, capable of simulating very large numbers of neurons. It is particularly well suited for models of sensory processing, such as those performed by a mammalian visual or auditory cortices. NEST is implemented in C++, with several Python interfaces also available. Therefore, it is highly suitable for the purpose of neural simulation and NeuROS packages PyNEST [37] as one of its standard containerised nodes.

2.4 Integration

In order to integrate physics and brain simulators together, NeuROS will need to pass sensor data output from the physics simulator to the inputs of the one or more brain simulators. Once the brain(s) have computed a decision, any output actuator commands will then need to be passed back to the physics simulator in order to control the robot. Therefore NeuROS requires a transport layer which is flexible both in terms of topology and message content.

The modularisation of software components is widely practiced by software engineering professionals. It has been demonstrated [38] that this approach saves both time and money. This is mostly achieved by reducing duplicated work, and therefore also reducing the time and cost associated with maintaining and bug fixing a larger and redundant code base. NeuROS strongly encourages this approach via the use of containers, with software components encapsulated into indivual modules as much as is practically feasible.

2.4.1 ROS2

For many years Robot Operating System (ROS) [21] has been the industry standard middleware used for building robotic systems [39]. It is a highly distributed event driven system, which promotes decoupled architectures and the easy integration of heterogenous software components. For this reason and others, several existing integration frameworks are built using ROS, including the Neurorobotics Platform (discussed in a later section) [24] [16], SkiROS [20] and ROS-NetSim [19]. Where NeuROS differs in this regard, is in the adoption of ROS2, the latest version of the Robot Operating System. ROS2 is generally more distributed than ROS1, with no master node required for communications. Instead it utilises a new Data Distribution Service with configurable Quality of Service, which has been demonstrated to deliver improved performance [40]. Other beneficial features include easier cross-machine communication, meaning that NeuROS nodes can be executed seamlessly across different physical or virtualised platforms. This makes it easier to leverage specialised hardware, or even on-demand resources such as cloud hosting.

By itself, ROS2 is slightly too low level to provide the functionality that NeuROS would ideally present to users. By allowing users too much low level control over the messaging layer, the ability to add over-arching checks and controls is lost. Furthermore, at minimum each node would require knowledge of it's direct neighbours, which ties them to a specific architecture, or at minimum a specific set of publisher/subscriber topics. In order to remedy these problems, NeuROS builds a slightly higher abstraction layer, hiding the communication details from the individual nodes. This allows NeuROS to implement synchronisation strategies on top of ROS2, in addition to making individual nodes better separated and more modular. Connections between nodes can then be defined at the project layer, and communication with external pure-ROS2 nodes can still be supported (albeit with reduced control).

2.4.2 Containerisation

Containerisation is a well established technology [22] for the deployment and maintenance of complex software across a vast range of domains. Tools such as Docker [23] allow custom built images containing collections of software to be isolated from the host environment and run with near-native performance. The alternative to this approach is to manually install each software dependency on any machine required to execute a given software component, and this comes with greater costs and potential for human error. Furthermore, conflicting software requirements on the same machine are relatively common, and often prove fiddly or impossible to resolve. So far as can be determined, the only existing neuroscience modelling integration framework that provides any support for containerisation is the Neurorobotics Platform, with a single downloadable image containing the complete system. In contrast, NeuROS will utilise Docker to containerise each individual node. With this approach, NeuROS will provide straight forward deployment of nodes on different machines, improved collaboration possibilities and greater separation of concerns. The level of dependency management and component modularity provided by this approach will make it trivial for nodes with conflicting software requirements to be executed within the same computational model. Providing this level of flexibility and ease-of-use takes the proposed NeuROS functionality beyond any existing integration framework.

2.5 Summary

From the above literature review it can be understood that there is currently no framework which delivers on all of the following desirable NeuROS features:

- *Highly flexible distributed architecture:* The NRP and Ikaros both have a spoke-and-wheel architecture, SkiROS is entirely fixed, ROS-NetSim is centrally coordinated through the network coordinator and BRAHMS is unrestricted but centrally controlled. NeuROS is unique in this regard in that it will provide both an unrestricted and fully distributed architecture with no central controller or limitations on node connectivity. Therefore NeuROS offers greater flexibility in terms of how complex the research can be, allows computation to be distributed over a wider network of resources, and offers greater modularity and reuse.

- *Flexible orchestration (tick based or fully distributed event driven):* BRAHMS and Ikaros both use a form of tick controller, while the NRP uses it's Closed Loop Engine (CLE), SkiROS is scheduled via the central planner and ROS-NetSim is coordinated by the physics and network coordinators. Again NeuROS is unique in that it offers multiple forms of orchestration including distributed event-driven blocking conditions in addition to tick controllers. By definition, tick controllers involve parts of the system sitting idle for some period of time. By providing alternatives to tick controllers, NeuROS has the potential to improve computational performance by more fully saturating the processing resource of the host machine. The NRP's CLE offers a closed perception-action feedback loop which NeuROS will have to replicate to some degree, via the manual construction of a circular topology between appropriate component nodes.
- *Modularity and containerisation:* BRAHMS, Ikaros, the NRP, SkiROS and ROS-NetSim all aim to provide some level of modular components. However none of them comes close to the modularity provided by NeuROS, where each individual node is a separate container with it's own individual software requirements. This should enable researchers to not only include more complex and even mutually incompatible software in their experiments, but also to package and share components more easily, reduce implementation effort.
- *Up-to-date physics (non-classic Gazebo) simulation:* Both the NRP and ROS-NetSim integrate with Gazebo Classic, while BRAHMS integrates Webots. None of the reviewed integration frameworks competes with NeuROS, which supports the more modern Gazebo variant, with it's architectural improvements and ability to execute synchronously as a library.
- *Extensive documentation:* While the NRP approach appears to be more focused on providing an easy-to-use black box, where end users are not expected to get their hands dirty. In contrast NeuROS aims to be transparent, enabling understanding through documentation and encouraging users to understand the inner workings and design. It is hoped that this approach allows for a more flexible and powerful tool, for those users who wish to build more complex experiments, and to make full use of such flexibility. The NRP, BRAHMS and Ikaros provide online documentation. No documentation could be found for ROS-NetSim and the SkiROS documentation appears to be unmaintained (at the time of writing the link on the SkiROS

GitHub page [41] gives a 404 error). NeuROS is at least as good if not better, with detailed documentation of every class and method, auto generated on installation and complete with UML diagrams for ease of understanding.

So far as can be ascertained NeuROS is completely unrivaled in it's provision of containerisation for software dependency separation, fully distributed architecture utilising the latest ROS2, modern physics simulator and highly flexible orchestration offering a range of customisable trade-offs between speed and accuracy. Thus, this represents a clear niche for NeuROS and delivers much needed functionality that researchers are currently without.

3 Research Methodology

This project has ambitious goals and consequently a significant amount of software design and implementation was required. The following sections explain in some depth the rationale behind each of the major design decisions, and methodology for the implementation and evaluation. NeuROS is mostly comprised of a single Python application of approximately 3,000 lines of code. All 3rd party software is bundled in several accompanying containers, which are launched on demand as required by any given project. We will start with the system architecture, as designed by this work, and the method by which reliable session management was achieved. Following that we cover containerisation and the development of a generalised node structure. Next we explore customisation via project configuration and the bespoke user plugin mechanism created specifically for NeuROS. After that we move on to orchestration, hooks, input/output conditions and scheduling, which comprises the main focus on the work and serves to give NeuROS it's unprecedented flexibility. Then we explain how several 3rd party tools were integrated. Finally, we present the methodology used for evaluation, involving a representative neurorobotics experiment orchestrated via various techniques under NeuROS. A timetable for this work is given in Appendix A.5.

3.1 Architecture Design

In order to allow users to define projects with highly flexible node topology, the architecture of NeuROS has been designed such that the resulting ROS2 node topology is generated entirely at runtime. Thus the internal architecture of NeuROS bears little relation to the resulting structure of running projects. Instead, we can think of NeuROS as a factory for highly complex ROS2 networks of containerised nodes.

Figure 3.1 shows a class diagram for the main components of NeuROS. There are two main entry points; *Install* (install.py) and *Launch* (launch.py).

Install is the first NeuROS entry point a user must run after downloading the repository. First impressions are important, and it is hoped that making the installation processes as streamlined

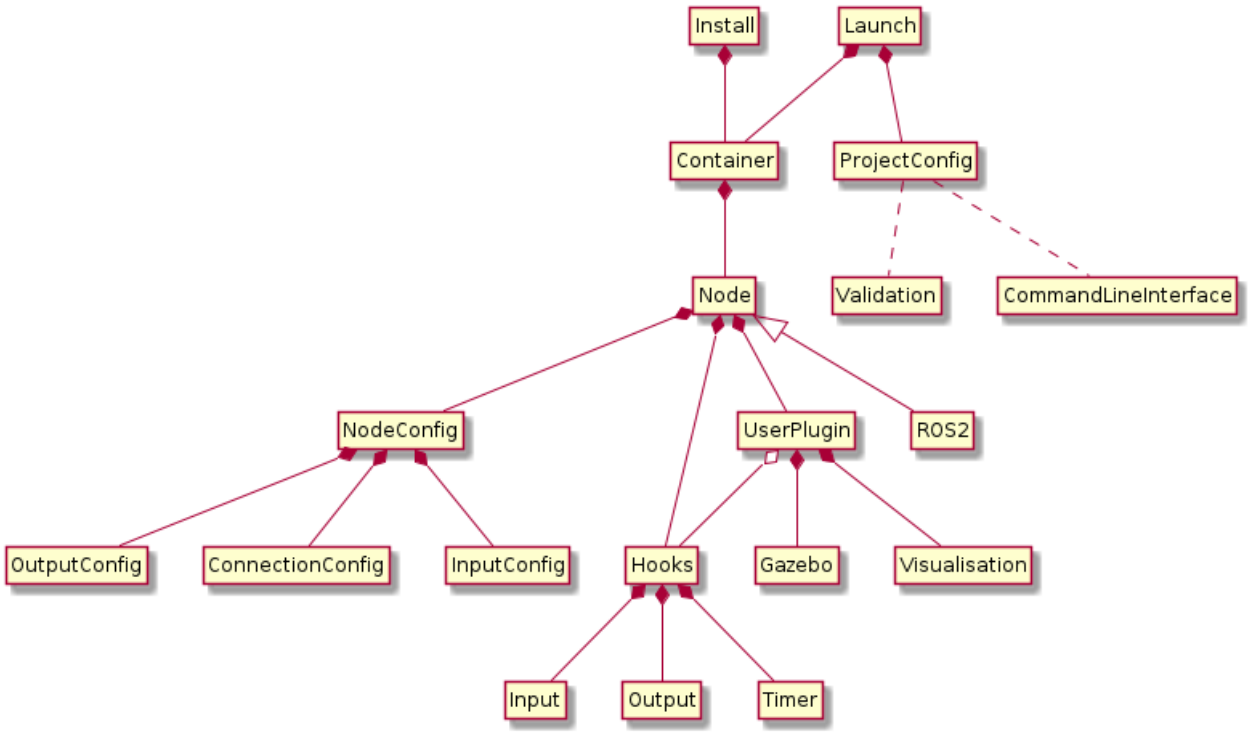


Figure 3.1: High Level NeuROS Architecture Design

as possible increases the chances that NeuROS will be widely adopted. Therefore, *Install* checks that all system requirements are met, downloads and then builds all required bundled software components. This ensures that installation is as simple as running a single command, making it as straightforward as could reasonably be expected.

Launch is responsible for loading *ProjectConfig* from the project path specified by the user via the *CommandLineInterface*, inspecting it and starting a variable number of *Container* instances as specified. The *Validation* module includes features such as project schema validation, and thus must be performed against the entire project configuration at this stage, prior to the config being split between nodes. It is also a widely accepted good software engineering practice to validate inputs as early as possible.

UserPlugin is a placeholder for user specified plugin code, which will be explained more fully in Section 3.5. A variable number of *Node* instances are created at runtime, and their relationships are dynamically configured via the associated *InputConfig*, *OutputConfig* and *ConnectionConfig*. *Timer* can be used for implementing tick-based synchronisation strategies. *Hooks* provide registration and storage of *UserPlugin* callback functions. A more in-depth explanation of these

orchestration techniques is provided in Section 3.6. The *Visualisation* features are also controlled from the *UserPlugin*, allowing customised data visualisation and integration with ad-hoc 3rd-party tools. Similarly, *Gazebo* is launched directly from the *UserPlugin*, adhering to the aim of being lightweight and allowing the user as much direct control of native 3rd-party interfaces as possible.

3.2 Containerisation

During installation NeuROS downloads and installs a containerised fixed version of ROS2 [42] via a *docker pull* command. A handful of additional standard NeuROS containers are built upon this image, each including the addition of a single 3rd party tool, which then become available system-wide. This approach guarantees a fixed ROS2 version for each node, which ensures stability, while also allowing easy upgrade via the modification of a single root image specification.

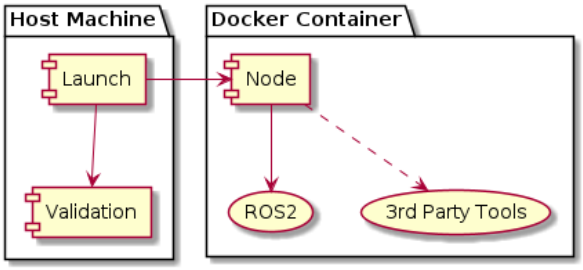


Figure 3.2: Node Containerisation

Users must then specify a single container to be used for each node as shown in Figure 3.2, which may include those available to the system but not provided by NeuROS. This approach ensures almost complete logical separation of node software dependencies, allowing users to package virtually any tool they wish to be used by a NeuROS node. This provides an extreme level of flexibility and modularisation, even supporting the case where two nodes have mutually incompatible software dependencies.

3.2.1 Session Management

Launching and managing large numbers of containerised processes is not a trivial problem. During the development of NeuROS a considerable amount of time was spent ensuring that this functionality was robust. One particular problem was that some processes were not always correctly

terminated following the user issuing Ctrl+C. There exist several tools designed for orchestrating docker, such as Docker Compose [43], however these were deemed too heavy-weight for NeuROS and also appeared to exhibit similar known problems [44]. Instead, a solution was found through a deeper understanding of Unix process groups [45].

When multiple processes are launched from a single teletypewriter (TTY i.e. a user terminal), only one of them is considered to be in the foreground process group. This means that when Ctrl+C is issued by a user, only one of the NeuROS node processes receives the signal and is terminated, leaving the others running in the background. In order to remedy this issue, *Launch* must monitor when any of the known processes have exited and signal termination to all of the remaining processes via a SIGKILL. This approach ensures clean shutdown both in the case that the user issues Ctrl+C and when one of the processes has crashed.

3.2.2 Logical Network Separation

Since NeuROS uses ROS2 for communicating between nodes, it is also capable of allocating nodes to different physical machines and executing across a distributed network. This is a huge benefit in terms of leveraging on-demand compute power, such as the cloud, but opens the door for potential interference between different experiments communicating via identically named topics. ROS2 provides a solution to this problem, called *domain IDs* [46], and NeuROS exposes this functionality via it's command line interface. When running on a network, users can choose an arbitrary (though expected to be unique) integer as the domain ID and specify it whenever launching NeuROS nodes for a single experiment. This allows for logical separation of physically collocated nodes on the same compute resource, as shown in Figure 3.3.

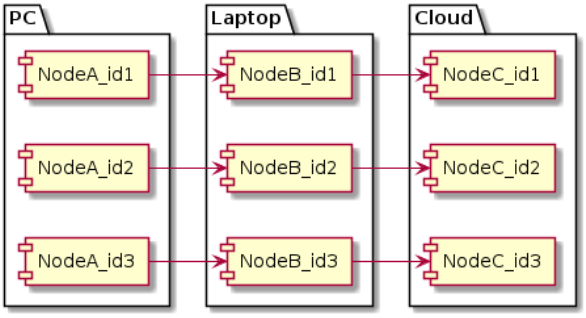


Figure 3.3: Logical Node Separation via Domain IDs

3.3 Nodes, Packages & Workspaces

Since each node will run in a dedicated Docker container, we can utilise mounted volumes to present each one an identical file system structure with varying contents. This enables the repeated reuse of a single generic ROS2 package and node, instances of which can load a particular node configuration file *node.json* from a fixed file path and perform node-specific behaviour. The repeated reuse of this single ROS2 node avoids the need for users to rebuild anything at all, even after significant project architecture modification (in stark contrast with Ikaros, where any change requires recompilation and relinking).

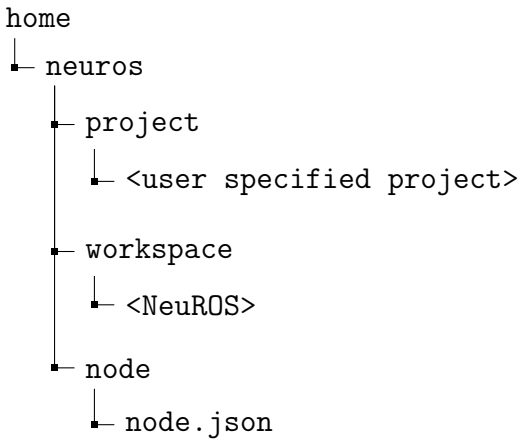


Figure 3.4: Container Directory Structure

Figure 3.4 shows the directory structure as seen by a NeuROS node container. *node.json* is a temporary file generated only when a node is launched and contains just the information pertinent to that specific node, such as it's configuration and connections with any immediately neighbouring nodes. This lack of project-wide knowledge encourages node implementations to remain modular and reusable.

3.4 Project Configuration

3.4.1 Centralised vs. Modular

In general, NeuROS can be understood as providing centralised project configuration with distributed execution. However, when it comes to configuring a project we have two conflicting re-

quirements: Modularity and ease of use. In order for nodes to remain modular they should be configured in a distributed fashion at the node level. Whereas users would presumably find it easier to modify a complete project configuration if it was all located centrally in a single file. In an attempt to support the best of all worlds, NeuROS can be configured entirely centrally, but also supports the ability to import node configurations from dedicated files. When launched, NeuROS combines any imported node definitions into a central configuration. As mentioned previously, each node is then only provided runtime access to configuration relevant to it and it's immediate neighbours. All connections between nodes are specified only in the central configuration and therefore it is possible to import nodes that are used by other projects but connecting their inputs and outputs together in differing ways. By striking this balance, it is believed that NeuROS is able to maximise ease of use without compromising any of the benefits of modularity.

3.4.2 Schema Validation

Users are expected to make configuration mistakes, thus in order to improve ease of use NeuROS provides comprehensive error checking and reporting. Since each project is configured centrally, efficient schema validation can be applied at launch. The applied schema can be found in Appendix A.6. Configurations are implemented in JSON and therefore NeuROS utilises the *json-schema* [28] Python library, which gives clear error messages indicating the cause of any validation failures. If NeuROS is launched with a project configuration that contains errors, it reports the error and exits immediately.

3.4.3 Environment Configuration

In order to promote reuse and modularity, it is beneficial for users to be able to pass arguments to their plugins. This is problematic given that NeuROS always launches the same ROS2 node with the same command, differing only by the node configuration file content presented to the container. Therefore NeuROS allows environment variables to be specified in the configuration file and injected into the node's container at runtime.

To facilitate the referencing of project resources, such as trained brain models or physics model meshes, NeuROS has been designed with two special operators:

- **#** Indicates a path relative to the standard *project* directory. This is useful when referencing static resources that are included in a users project directory.
- **@** Indicates a path relative to the standard *workspace* directory. This is useful when referencing components that must be built and as a result are contained in the NeuROS installation directory e.g. custom Gazebo plugins or message types.

Utilisation of these operators allows plugins to reference resource using relative paths, which helps to reduce explicit configuration dependencies and promotes modularity.

3.5 User Plugins

In order to provide flexibility, both in terms of customisation and extendability, NeuROS requires the use of a per-node user plugin mechanism. The Python programming language was chosen for a number of reasons. Firstly, it is the most widely used language in the field of machine learning [47], and is therefore likely to be well suited to brain modelling. Supporting Python based plugins also allows users to interact with any 3rd party Python libraries, including but not limited to TensorFlow, NEST and Gazebo, providing extensive flexibility and potential for future extension. Finally, user specified code can be loaded dynamically at runtime via Python's *importlib*, avoiding the need for any recompilation or relinking. This increases ease of use and is likely to speed up the users software development life cycle.

3.5.1 Separation of Node and Communication Layer

In order to promote modularity and reuse, each node should know as little about any specific project as possible. It is therefore beneficial to separate the communication layer, which must contain information about the network topology, from the user plugin code. This allows user plugin code to be reused in different projects, where inputs and outputs maybe be sourced from or sent to different neighbouring nodes. Therefor both the ROS2 communication topics and message types are only explicitly defined in the project configuration, where they are additionally assigned aliases. In the user plugins themselves, only the aliases are referenced, and thus the flexibility of any arbitrary project node topology is maintained.

3.6 Orchestration

Many of the previously reviewed integration frameworks, such as the NRP, specify strictly controlled orchestration, often in the form of centralised controllers. In contrast, NeuROS aims to provide a higher degree of flexibility, and therefore the orchestration approach is considerably looser and even potentially fully distributed.

3.6.1 Hooks

User plugins are able to define and register Python functions, upon which they become *hooks*, which may accept inputs and provide outputs. In order to provide intuitive syntax, these have been designed and implemented by this project using via Python decorators, in a similar fashion to the NRP, using the input and output configuration aliases. In order for one of these hooks to be executed there are two conditions that must be met:

- **Input condition**: All of the hooks inputs must have been received.
- **Output condition**: Any other nodes in the project that receive an output from this hook must be ready to receive.

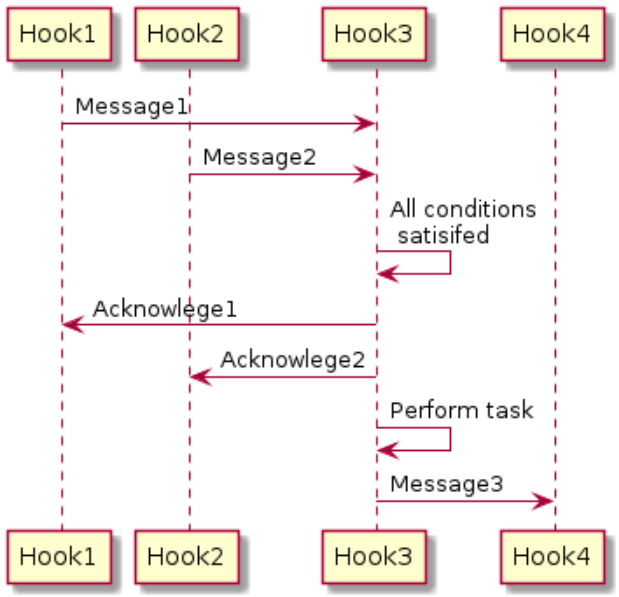


Figure 3.5: Distributed Execution

The concept of "ready to receive" must be clarified in order to fully understand the power of this approach. Figure 3.5 shows a sequence diagram for an example communication involving four hooks. For this example, let's assume they are all in different nodes, although in general any configuration is supported. *Hook3* requires two inputs (*Message1* and *Message2*) and produces a single output (*Message3*). We can assume all output conditions are satisfied before the sequence starts. First, *Hook1* and *Hook2* execute in parallel (since they are not in the same node) and send their outputs (*Message1* and *Message2* respectively) to *Hook3*. Upon receiving them, *Hook3* has all conditions met and begins execution, which triggers acknowledgements to be sent back to *Hook1* and *Hook2*. At that point, *Hook1* and *Hook2* have their output conditions satisfied again and are able to resume execution. This includes potentially sending any new outputs to *Hook3*, which would be queued until *Hook3* had completed its current task. Depending on the duration of each task there is likely a period when all three hooks are executing in parallel. Finally, once *Hook3* has completed it sends its output on to *Hook4*. It can be seen from this example that distributed blocking mechanisms based on input and output conditions can be used to synchronise parallel execution across multiple nodes without any risk of data loss or unsafe access.

3.6.2 Initialisation Hooks

Node initialisation can be performed via a specialised form of hook, which has two notable modifications relative to the standard hook. Firstly, it can only be executed after a registration process has completed. The registration process requires that for all outputs provided by a given node, every receiving node must both be running and have sent a registration request. Registration requests are sent periodically until the first message is received on that connection. This process ensures a highly robust and synchronised node initialisation procedure, allowing nodes to be started in any order or even on any machine, where nodes will not begin their main execution loops until all other dependant nodes are ready. The second modification of an initialisation hook is that it can only be executed once. After the first execution it is effectively deleted from the known set of hooks, ensuring that each node is only ever initialised once. Note that there is nothing stopping users from defining multiple initialisation hooks per node, which is supported with the warning that the order in which they execute is undefined.

3.6.3 Timed Hooks

The design of NeuROS also provides a specialised form of hook which can be triggered periodically. Compatibility is maintained with the rest of the system by applying the normal input and output conditions, with the additional condition that the timer must have also expired. Many existing frameworks rely on synchronisation through a centralised tick controller, which may have some benefits in certain cases by simplify and enforcing a stricter orchestration. Implementing a similar tick controller in a project via the NeuROS timed hook is relatively trivial, a basic example of such a setup is given in Section 3.9.2. This approach of providing users with the tools to build the system that they want provides the flexibility of choosing which orchestration approach is most suitable. This is demonstrated by the project in the *examples/1_tick/clock* directory of the accompanying repository [48].

3.6.4 Communication Protocols

Performance Metrics	**Performance Comparison**		
	TCP	***UDP***	***SCTP***
PLR	Good	Worst	Best
ETE Delay	Good	Best	Good
Jitter	Good	Best	Good
PDR	Good	Worst	Best
Fairness	Best	Worst	Good
Throughput	Good	Worst	Best

Figure 3.6: TCP vs. UDP Performance [49]

To further increase the flexibility of the NeuROS communication layer, the additional concept of a *discard limit* has been introduced. This defines the number of sequential messages that can be sent over a specific input / output pair connection, even when the receiver has not indicated it is ready, and therefore potentially lost. For a zero data loss connection this setting should be configured to zero. For systems that can tolerate some amount of data loss, this parameter offers a

customisable guarantee on the upper limit. In certain cases it may be desirable to stream messages on a purely "best efforts" basis, i.e. an infinite discard limit, and this can be indicated in the project configuration by omitting the discard limit entirely.

The discard limit feature works by leveraging some of the new Quality of Service [50] features in ROS2. In the normal cases, where a discard limit is specified, a TCP based connection is used via the *rclpy.qos.ReliabilityPolicy.RELIABLE* reliability setting. This guarantees the delivery of messages and enables accurate tracking of the count of discarded messages at any given time. In the case that there is no discard limit, a UDP based connection is used instead via *rclpy.qos.ReliabilityPolicy.BEST_EFFORT*. Previous research [49] shown in Figure 3.6 demonstrates that UDP offers reduced jitter and delay at the cost of losing any delivery guarantees, making this protocol highly suitable when delivery guarantees are not required.

3.6.5 Input / Output Modifiers

Outputs can be sent in a one-to-many connection, where one node broadcasts data to multiple others. In order to support the reverse of this operation, a many-to-one connection, this dissertation developed an *All* modifier, which can be wrapped around an input in the hook definition. Doing so modifies the input condition such that it is applied individually to each of the incoming connections i.e. an input message must be received from each one of the connected nodes. In such case, the corresponding function argument becomes a list, where each element of the list is a message from one of the nodes. This mechanism allows for many things including distributed computation, where a large task can be decomposed into multiple smaller homogeneous tasks, distributed to multiple nodes and reassembled once they have all completed. The width of such parallel computation is specified entirely by the project configuration, with zero impact on individual node structure, demonstrating the effectiveness of previous modularisation efforts. This is demonstrated by the project in the *examples/2_synchronisation/voting* directory of the accompanying repository [48].

Sometimes it is useful to specify that a hook should conditionally produce an output. For this purpose this dissertation provides an *Optional* type which can be wrapped around an output in the hook definition. The same output condition applies as for a non-optional output, since NeuROS has no means of inspecting and predicting whether a hook will produce the output or not (i.e. it is assumed that it always will). *Optional* can also be applied to inputs, indicating that a

hook can be executed even if that particular input has not yet been received. In such a case the corresponding function argument will be set to *None*. This could be useful e.g. if a hook expects inputs to be received at different frequencies. This is demonstrated as an output modifier by the project in the *examples/2_synchronisation/tennis* directory and an input modifier in the *examples/3_physics_simulation/elevator* directory of the accompanying repository [48].

3.6.6 Synchronisation & Scheduling

The lack of a centralised controller entails distributed and localised synchronisation between nodes. NeuROS delivers this functionality by only executing hooks when the input and output conditions are satisfied, and therefore requires some form of hook state management and prioritisation. For this purpose, significant investigations were conducted into several potential approaches, and their relative strengths and weaknesses were compared:

- **Synchronous Service Calls** Some initial ideas were explored around using ROS2 *Services* [51] for leveraging blocking calls between nodes in order to orchestrate some level of synchronisation. This would be extremely simple to implement, requiring almost no additional NeuROS code at all. Unfortunately this was quickly established to dramatically increase the chances of deadlocks, due to the single threaded nature of the default message executor. Therefore the user would need to be extremely careful when designing their messaging structure, and NeuROS would likely become unmanageable for anything but the simplest of projects.
- **Multithreaded Executors** As a potential workaround to the above deadlocks, some further investigation was conducted into running multithreaded executors [52]. While this does solve many of the problems of using *Services*, it would also require all user code to be written in a thread-safe manner, thereby significantly increasing the implementation difficulty and reducing user friendliness.
- **Semaphores** Another potential architecture would involve giving each hook a dedicated thread, which by default would be blocked waiting for a mutex-based semaphore. Upon all input and output conditions becoming satisfied for that hook, NeuROS could release the

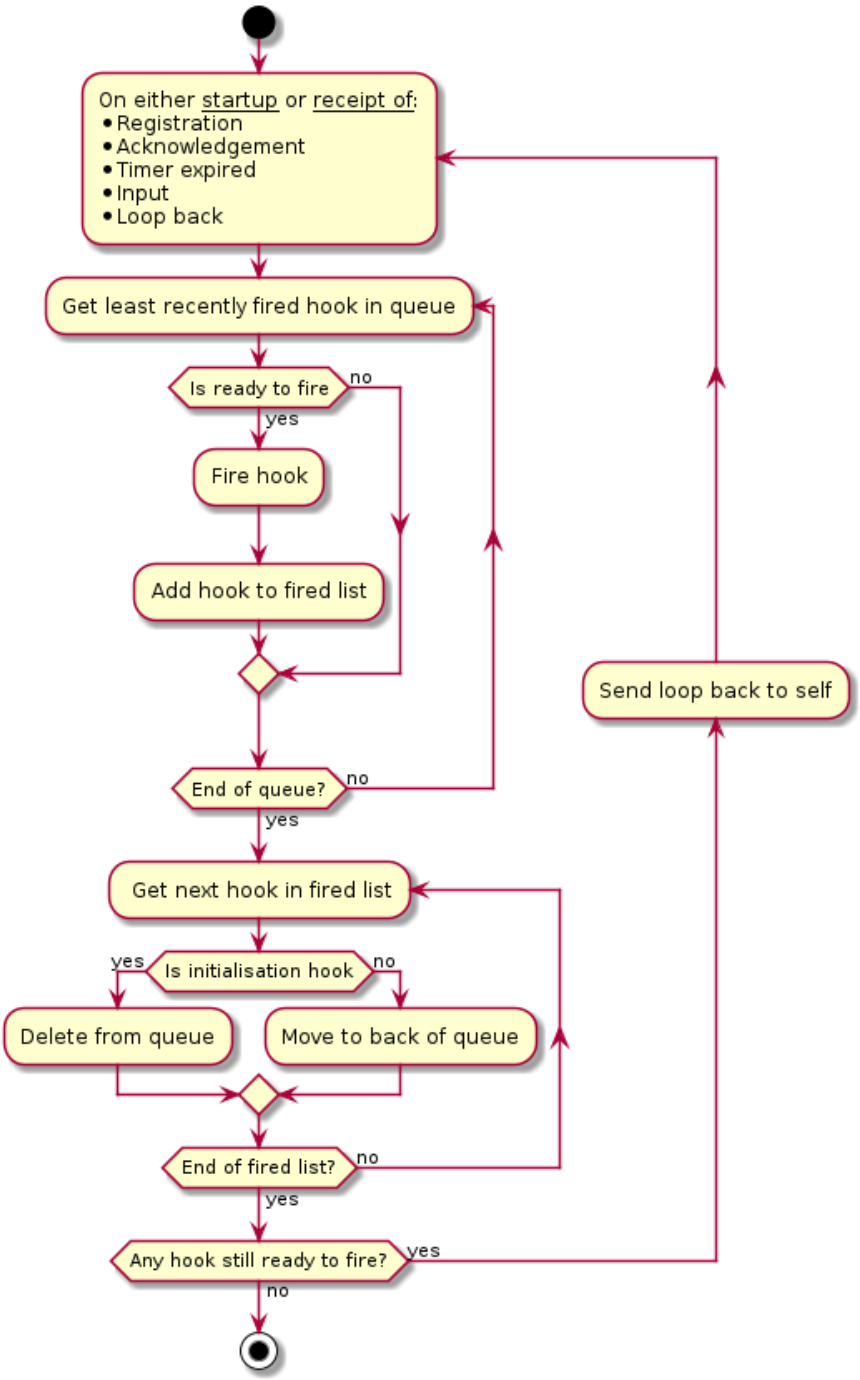


Figure 3.7: Priority-based NeuROS Message Queue

semaphore allowing the hook to execute. Unfortunately this approach also requires all user code to be thread-safe, and was thus deemed over-complicated and error prone.

- **Priority Queues** Eventually the idea of priority-based message queues emerged as a good solution and was therefore adopted for the final NeuROS implementation. Priority queues are widely used for many applications throughout the domain of software engineering and algorithm design [53] [54]. In the case of NeuROS they can be utilised to evenly allocate CPU time to each of the registered hooks. If we imagine that two hooks both produced outputs for the same connection with a zero discard limit, disregarding input conditions, whichever hook was declared first would always be given preference. Therefore we need to execute hooks in the order of longest-waiting first, and priority queues are an appropriate tool for such a task. The hook priority queue is ordered by the last executed time, with the least recently executed hooks at the front of the queue. When inputs or output acknowledgements are received, the hook closest to the front of the queue who's input and output conditions are fully satisfied is then executed. Upon execution it is moved to the back of the queue and the process starts again, ensuring all hooks are given equal opportunity to execute. This approach means that it is both easy and highly efficient to identify the next hook to execute, since the logical time-based priority matches the physical search order. Each hook in the priority queue is only ever executed at most once per receipt of a message. If after executing, hooks remain in the queue that are ready to execute again, then a special *loop-back* message is sent by the node to itself, in order to indicate that the queue should be processed again. This approach allows any other incoming messages to be processed before the second execution of any given hook, which is an important detail to ensure the correct processing of optional inputs. This approach also allows hooks to schedule differing and even dynamic execution frequencies, which offers increased flexibility when compared with competing platforms such as BRAHMS [17]. The complete process used by NeuROS to manage its priority-based message queue is shown in Figure 3.7.

3.6.7 Message Types

In order to keep node implementations decoupled from their configuration, NeuROS nodes have been design such that they do not have access to their message types directly. Instead, the user plugin requests that the node creates the output message for a named connection (as defined by the project configuration), and then populates the fields of the resulting object. This approach allows message types to be changed with zero modifications to the user plugin (with the restriction that the new message type must use the same field names as those used by the existing plugin). The standard message types from ROS2 and Gazebo are all supported out-of-the-box, in addition to any custom user defined message types as required.

3.7 Gazebo Integration

NeuROS offers physics simulation through the integration of of a 3rd party simulator, Gazebo [32]. This is an extremely rich, actively developed and complicated simulator, with many potential interfaces and versions available.

3.7.1 Versions & Compatibility

There is a great deal of confusion amongst the online community [55] surrounding the naming of Gazebo. The original project was named Gazebo, with a later significant rewrite introducing the name Gazebo Ignition. More recently, Gazebo Ignition has been renamed to Gazebo and the original legacy project was renamed to Gazebo Classic. This research has adopted the most recent naming convention throughout.

Gazebo was selected due to a number of notable advantages over Gazebo Classic:

- **Active Development** Version 11 is the final release of Gazebo Classic and is scheduled to become end-of-life on 29th January 2025 [56]. From this point onwards, the Gazebo will be the only Open Robotics simulator in active development. In order to benefit from future features and bug fixes NeuROS will adopt this version from the outset. Other competing platforms, such as the NRP [25], use Gazebo Classic. Eventually the entire Gazebo community

will need to migrate their projects, and this may make NeuROS a more attractive migration option. Unfortunately, using cutting edge software brings it's own problems, with the development team still ironing out many bugs. Thus, during NeuROS development, problems were often encountered which required a significant investment of time to resolve or find workarounds [57]. The C++ Plugin API has been entirely rewritten and migrating from Gazebo Classic is a non-trivial task [58]. However, the expectation is that this migration be worth it in the long term.

- **Simulator as a Library** One of the greatest strengths of Gazebo over Gazebo Classic is it's ability to be run as a library of a separate user process. This allows clients to programmatically execute a physics update for an exact specified amount of time, synchronously, and without the use of any network based transport messages. This is an essential feature in order for NeuROS to provide synchronous orchestration techniques. Unfortunately, the existing Python bindings are relatively limited [59] and therefor all of the joint command and sensor data must still currently travel via transport messages. The non-determinism that this introduces can not currently be entirely eliminated, though this should become possible as the Python bindings become more feature complete. Gazebo additionally allows the physics updates to be run entirely asynchronously, thereby allowing NeuROS to provide an additional real-time execution mode.
- **Modular Architecture** Further benefits include the more modular design of Gazebo. In particular, this allows for the substitution of physics and rendering engines, which provide users with greater opportunities for customisation and extension. This fits very well with the design ethos of NeuROS, which promotes modularity and customisation wherever possible.

Since all NeuROS user plugin code interacts only with the provided Gazebo wrapper (gazebo.py) and never directly with the simulator itself, this should help to encapsulate the simulator functionality and hide version-specific changes from the user as much as possible. Version upgrades are thus made relatively trivial to implement through modification of the corresponding Dockerfile and gazebo.py.

3.7.2 Gazebo Bridge (& other External ROS2 Nodes)

The Gazebo messaging system does not interface with ROS2 directly, and therefore an additional middle layer (supplied by Open Robotics [60]) is required to translate packet data from one messaging system to another, as is shown in Figure 3.8.

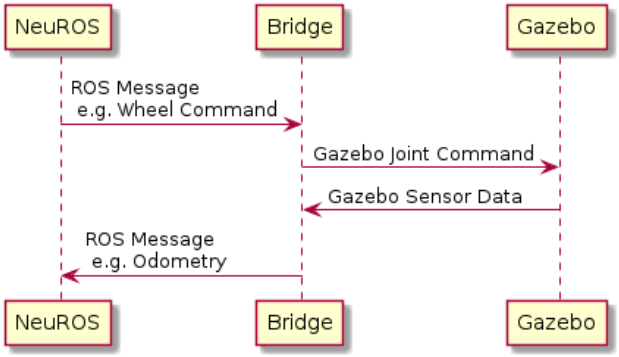


Figure 3.8: Gazebo Bridge

Support for Gazebo bridge requires two modifications to NeuROS. Firstly, support was added directly to the NeuROS message routing for sending and receiving against external ROS2 topics. Note that this offers extreme flexibility, with NeuROS now able to interface with *any* other external ROS2 system. This feature is limited by the loss of NeuROS-specific orchestration tools, such as the *discard_limit*. Secondly, the *gazebo_type* properties are parsed by gazebo.py and, in combination with the standard *type* property, used to launch an appropriate instance of the official Gazebo bridge process. This approach hides a large amount of complexity from the user, maximises simplicity and consequently helps to increase ease-of-use and reduce barriers to entry.

3.7.3 GUI

One of the central design considerations of NeuROS is to make use of existing interfaces and tools as much as is feasible. This brings numerous benefits including reduced maintenance, leveraging of existing open source tools and perhaps most importantly, consistency and familiarity for existing users of those tools. This latter consideration provides two of the well established user design principles of the Simplicity, Consistency, Universality, Flexibility and Familiarity (SCUFF) approach [61] that are adopted by NeuROS in order to increase usability. As such, the native Gazebo interface can be accessed through NeuROS and, when executing Gazebo asynchronously, the simulation

can even be paused and restarted arbitrarily as required.

3.8 User Interaction

3.8.1 Graphical Tools & Visualisations

In addition to the native Gazebo GUI there are a number of additional graphical tools incorporated into NeuROS for user convenience.

Matplotlib

Matplotlib [62] is a tool for plotting graphs that additionally supports animations. NeuROS leverages this library and wraps it in some highly simplified and user friendly classes inside the visualise.py module. This enables users to easily plot line graphs over time, for quick and easy real-time data visualisation.

rqtgraph

rqtgraph [63] is graphical tool provided by Open Robotics for visualising the ROS2 node graph. NeuROS leverages this tool, which can be launched by passing the *–node-graph* parameter to launch.py, in order to present node relationships to the user. In fact, this can almost be considered to be a visualisation tool for the entire NeuROS project configuration file, with the omission of message types, discard limits and environment variables. It is hoped that this will aid system design, maintenance and performance tuning by making the underlying system structure more transparent.

rviz

rviz [64] is yet another graphical tool provided by Open Robotics for the purpose of visualising data published on ROS2 topics. Generally, this allows users to easily visualise and plot inter-node messaging. This is an extremely useful tool for debugging purposes and can be launched within NeuROS by passing the *–visualisations* parameter to launch.py.

3.8.2 Data Logging

ROS2 also provides several in-built mechanisms for logging, or which NeuROS makes full use. Firstly, the console logger can be accessed by user plugins via the *node* parameter passed to all plugin hooks. In actual fact, the entire ROS2 node can be accessed in this manner, giving users access to the entire ROS2 Node API. Once again, the design philosophy of NeuROS is to reuse existing tools that the user is familiar with, rather than to incur additional development cost developing bespoke solutions. This type of logging is generally used for displaying information to the user, either to report execution progress or to aid debugging. Secondly, ROS2 provides a concept known as *bag recording*, whereby all of the messages corresponding to a particular topic (or set of topics) are recorded to a database. This can be used to record an entire ROS2 session, and NeuROS thus inherits and leverages this functionality. In order to increase ease of use, bag recording can be enabled with *launch.py –record* and each topic can be recorded by passing *–topic <topic>*.

3.9 Performance Metrics

3.9.1 Experiment Architecture

In order to assess the performance of NeuROS, a project closely based on the previously reviewed research [15] was implemented. The exact WhiskEye robot model from the previous research was ported to the latest version of Gazebo, updating the definition file accordingly. An equivalent NEST model was utilised, which was capable of estimating the robot head direction given the IMU readings. This model was hand tuned using the real-time data visualisation tool, until the resulting estimates closely tracked the ground truth. Due to time constraints, the predicting coding network from the original network was not fully integrated into this experiment. Therefore, this architecture represents a slightly simplified version compared with that presented in the original research. Nonetheless, it includes both physics simulation and neural computation and therefore represents a complete and representative neurorobotics experiment. Figure 3.9 shows the architecture, as is implemented in the *examples/4_neurorobotics/whiskeye* directory of the accompanying NeuROS repository [48].

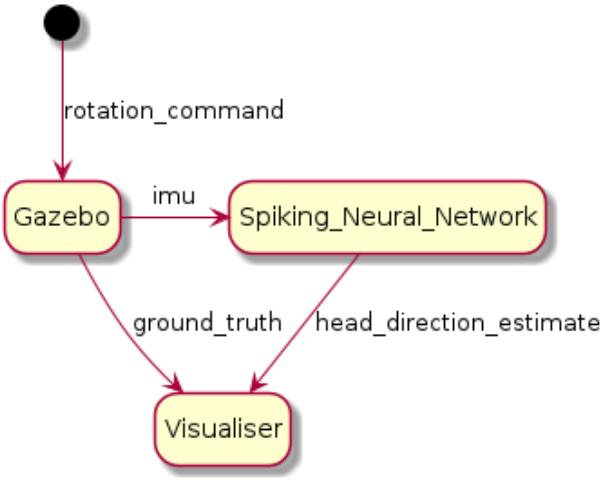


Figure 3.9: Complete Neurorobotics Example Architecture

As shown in Figure 3.9, there are three main connections in this experiment: The IMU readings are sent from Gazebo to the spiking neural network. Additionally, Gazebo sends the ground truth to the visualiser node. Finally, the spiking neural network sends the head direction estimate to the visualiser node. All three of these connections will be impacted by the four orchestration techniques presented in the following section.

Upon launch of the experiment, the Gazebo node sends itself a rotation command using the NeuROS initialisation hook, upon which the simulated robot begins rotating on the spot in an anti-clockwise direction. The ground truth and head direction estimate messages are logged to a file via the previously discussed logging system, and the experiment is terminated once 20 seconds worth of physics has been simulated.

3.9.2 Orchestration Techniques

NeuROS is unique amongst neurorobotics integration frameworks for it's breadth of supported orchestration techniques. Four were chosen for the purpose of the initial comparative performance analysis, and an additional one *Hybrid Approach* based on the lessons of the previous four. Giving the following five in total:

- **Tick Controlled at 20ms, 60ms and 100ms:** For this approach, physics simulation updates were scheduled to occur at fixed real time intervals, regardless of the state of the other nodes. The tick controller is implemented using a NeuROS timed hook and invokes the syn-

chronous Gazebo step function. With this tick controlled orchestration, no other guarantees are in place, meaning that the discard limit is set to zero for every connection. This represents the crudest form of orchestration, but may be suitable for certain real time applications. Depending on the duration of the interval, very likely either one of two things will happen: Either the duration will be too short, and data loss will occur when busy nodes are sent new data to process. Or, the duration will be too long and all of the nodes will be sat idle waiting for the next tick. Therefore it is expected that there will be a very apparent and crude trade-off between speed and accuracy, depending on the tick interval duration.

- **Strict Synchronisation:** During this type of orchestration every connection will have a discard limit of zero, meaning they are transported using TCP and should be guaranteed to be delivered. Additionally, new physics updates will only be executed synchronously once the downstream nodes are ready to receive new data. Overall, this should absolutely guarantee zero data loss across the entire network, but will also have the greatest overhead and therefore likely the slowest execution speed.
- **No Discard Limit:** This orchestration approach builds upon the last but removes the discard limit from all connections, allowing data to be dropped and using UDP as the underlying transport protocol. It retains the synchronous execution of Gazebo, meaning that physics updates are still individually invoked from NeuROS in a relatively tight loop enable via the previously discussed loop-back message. This approach is intended to shed light on the overhead involved on the discard limit synchronisation.
- **Asynchronous Gazebo:** Finally, this orchestration approach executes Gazebo asynchronously, with NeuROS relinquishing all control. Gazebo is free to execute as quickly as it possibly can, and all internal NeuROS messaging is delivered via UDP with zero delivery guarantees. This approach should help to understand the overhead of synchronous Gazebo control, where physics updates are invoked from NeuROS. Very likely, this approach will provide the fastest running speed, while placing the experiment at high risk of significant data loss.
- **Hybrid Approach:** This approach combines two of the above, where *Strict Synchronisation* was employed purely for the critical IMU connection, while the other two connections were reduced to *No Discard Limit*. For this orchestration technique Gazebo was invoked

synchronously in order to guarantee the SNN node was ready when the IMU data was published. This approach demonstrates a flexible best of both worlds, where critical data can be guaranteed and non-critical (purely user interface) data can be delivered on a "best efforts" basis, delivering both speed and accuracy as required.

3.9.3 Baseline Comparison

In order to assess the performance of the NeuROS against the current state-of-the-art, an equivalent experiment was conducted running inside a locally installed instance the NRP. Note that it isn't entirely fair to compare an experiment running under NeuROS to one running under NRP, given the intentionally different versions of Gazebo, different NEST models, updated WhiskEye robot textures, use of ROS2 vs. ROS1, and differing Python versions. These differences are largely impossible to avoid, however the underlying principles should remain the same. That is to say, the experiment running under NeuROS should be able to estimate the robot's head direction to a comparable degree of accuracy when compared with the experiment running under the NRP, as per the original research [15]. Given these considerations, only a high-level visual comparison of the results was used to ascertain they are at least approximately equivalent.

3.9.4 Model Accuracy

During execution, both the ground truth output from Gazebo and the head direction estimate output from the SNN are logged against the current simulated time. Upon completion of all experiments, the head direction estimate and ground truth data can be plot on a single graph in order to compare the accuracy of the model estimates as the robot rotates. As preciously mentioned, only a high level visual inspection will be conducted to ensure the models are at least approximately performing the correct behaviours, as model prediction accuracy is not the main focus of this work.

3.9.5 Execution Speed

To measure execution speed, the simulation time as published by the Gazebo clock is periodically logged alongside the real world time. This allows us to measure how quickly the physics simulation is progressing. A slower simulation speed here likely indicates that Gazebo spent more time waiting

for the rest of the network to catch up. Since the simulation speed is also available within the NRP, this gives as a directly comparable execution speed metric. The simulation speed as a percentage of real time speed is calculated with:

$$simulated\_duration = simulated\_end\_timestamp - 0$$

$$real\_time\_duration = real\_time\_end\_timestamp - real\_time\_start\_timestamp$$

$$speed\_percentage = (simulated\_duration * 100)/real\_time\_duration$$

3.9.6 Packet Delivery Rates

Depending on the orchestration approach, there are two potential causes for packets to be lost within NeuROS. Firstly the presence or absence of a discard limit causes NeuROS to utilise either TCP or UDP respectively for the underlying transport, with the latter reducing delivery guarantees to a "best efforts" basis. Secondly, running Gazebo synchronously or asynchronously has similar trade-offs. While running asynchronously likely provides the fastest execution time, it also risks losing data if the rest of the network cannot keep up with the progression of the physics simulation. Thus data loss must be captured in two separate ways:

Firstly, for each connection within NeuROS, each send or receive operation is logged to the console and counted upon completion of the experiment. The number of dropped packets for any particular connection can then be calculated as:

$$dropped\_packets = number\_sent - number\_received$$

Secondly, the number of sensor packets received over the experiment time frame was compared against the expected number of sensors packets as specified by the Gazebo sensor frequency. If these two values differ then a packet must have been lost somewhere between Gazebo sending it and the NeuROS node receiving it:

$$dropped\_packets = (experiment\_duration/sensor\_frequency) - number\_received$$

3.9.7 Tick Interval Regularity

In order for timed hooks to function well as tick controllers, they will ideally maintain a highly regular and consistent interval. In order to measure the ability of NeuROS to perform this function,

a simple timed hook was executed in isolation. During that period the real world time of the firing of the timed hook was recorded via the logging system. After five minutes had elapsed, a histogram and corresponding statistical analysis was constructed for the deduced interval period.

3.9.8 System Load

In order to gain greater insight into how the system was performing, the CPU utilisation and memory consumption were recorded via the *monitor.py* script. This script was implemented by leveraging the 3rd party psutil library [65]. Once per second psutil is used to obtain and log the CPU utilisation (per-core percentage) and memory consumption (used and percentage) via the following two psutil functions:

$$cpu\_utilisation = psutil.cpu\_percent(percpu = True)$$

$$memory\_consumption = psutil.virtual\_memory()$$

For NeuROS based projects *monitor.py* was started by passing the *–monitor-system-load* parameter to launch.py, whereas for the NRP-based experiment it must be started manually. While collecting load metrics, it makes sense to monitor the system-wide load, as opposed to per-process load, in order to capture the complete set of processes launched by these highly complex software projects. This is especially true for comparisons against the NRP, since these systems are out of our control and have the ability to launch arbitrary processes that we have no knowledge of. Thus, it is important that during these data-capture experiments the system is fully isolated, with no other tasks running on the system and any external network connections disabled, in order to capture only the load introduced by the experiment.

3.9.9 Hardware

All experimentation was performed on an Asus UX32A laptop, with 10GB RAM, an Intel quad core i5 3317U CPU and a 512GB SSD. This is very modest hardware for such demanding tasks, which should actually help to highlight any performance bottlenecks.

4 Results

Instructions for installing NeuROS can be found in the Appendix A.1. All experiment code, raw logs, performance metrics calculations and data plots can be found in the accompanying GitHub repository [48].

4.1 Model Accuracy

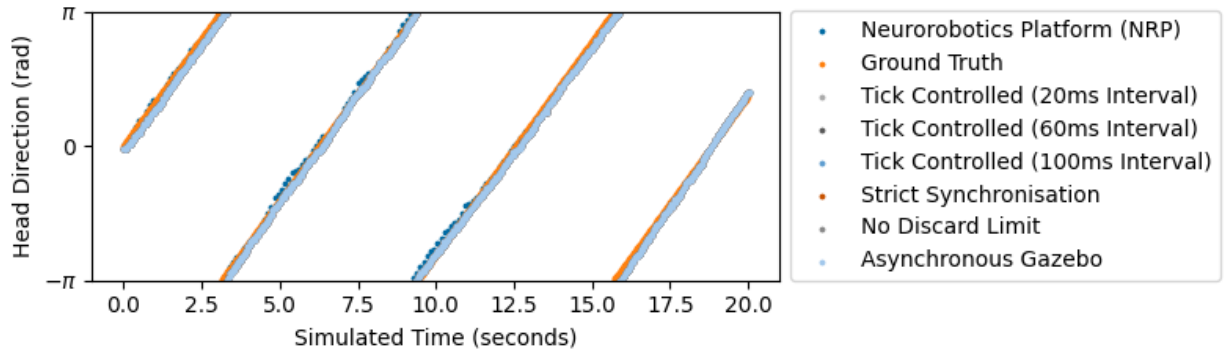


Figure 4.1: Comparison of Head Direction Estimates

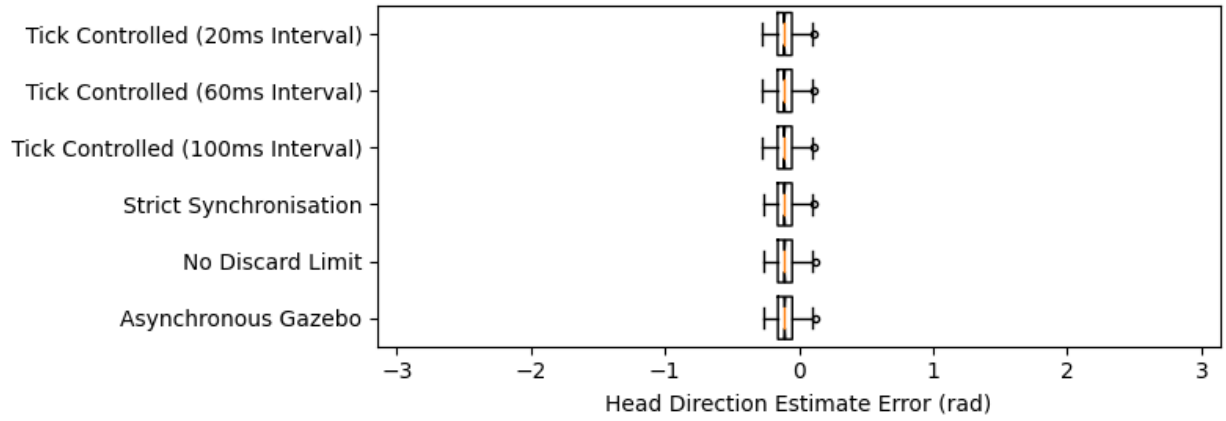


Figure 4.2: Comparison of Head Direction Estimate Error

Figures 4.1 shows a time series of the spiking neural network's ability to track the robot's head direction as the robot rotates, while running under both the NRP and all four of the investigated NeuROS orchestration techniques. Figure 4.2 shows the error exhibited by each of the four NeuROS orchestration techniques when compared with the ground truth. It can be seen that these are all virtually identical in terms of model accuracy, with no significant impact from varying the orchestration approach. This is due to the overall high packet delivery rates for the critical IMU connection, as explained in Section 4.3.

4.2 Execution Speed

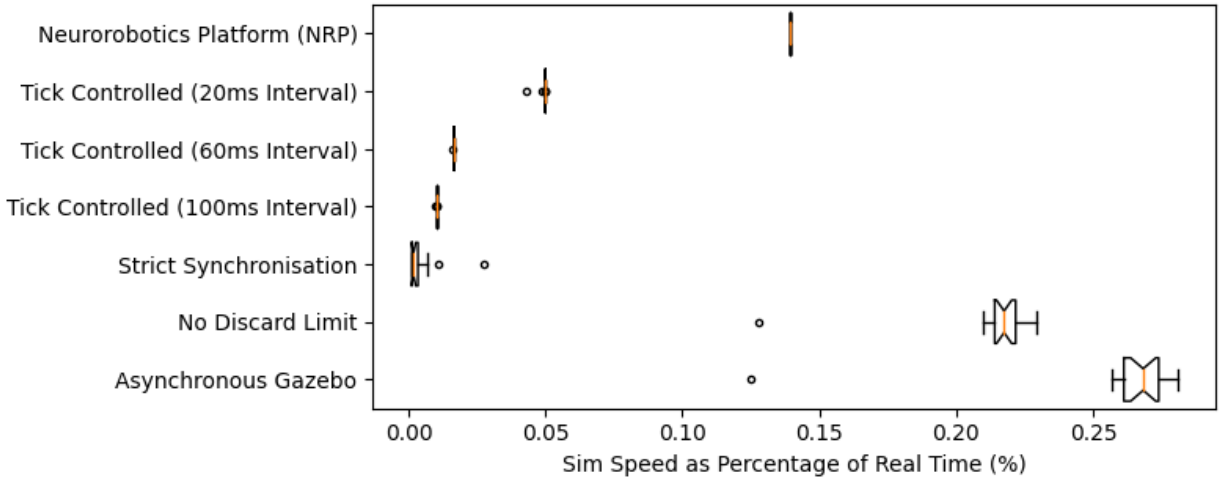


Figure 4.3: Comparison of Execution Speed

Figure 4.3 shows the speed of both the NRP and all four NeuROS orchestration techniques as a percentage of real time. (Note that the variance in NRP execution speed could not be obtained due to the difficulty in extracting precise measurements from a running NRP instance, and is therefore represented in the diagram as 0. This does *not* mean the NRP execution speed has zero variance.) Individual execution speed time series plots can be found in Appendix A.2.

Firstly, it can be seen that while the NRP only achieved less than 15% real time speed, the NeuROS Asynchronous Gazebo approach achieved around 25%, representing a significant potential speed improvement. However, it should also be noted that the *Strict Synchronisation* approach achieved less than 1%, which would represent a significant speed decreased when compared with

the NRP. The *Tick Controlled* techniques had the least variance, as would be expected from a fixed tick interval, and *Asynchronous Gazebo* gave the greatest, as is a likely a result of the complete lack of synchronisation.

The greatest speed improvement within NeuROS appears to be the use of *No Discard Limit* connections. Plausibly, this is due to the halving of the number of packets that need to be transmitted, while also allowing nodes to become unblocked earlier as downstream nodes are effectively always ready. The obvious downside of this approach is the potential for data loss, as will be covered in a later section. Nonetheless, given that this approach can be configured on a per-connection basis, this would appear to offer great potential benefits for non mission-critical data.

We can also observe that tick interval duration is inversely proportional to speed, as one might reasonably predict. This is because the longer tick intervals potentially leave the system idle for longer periods of time, resulting in less CPU utilisation and consequently less computational progress. This hypothesis is supported by Figure 4.6 where CPU utilisation during the 100ms approach is generally operating within a lower range when compared with that of the 20ms approach, as will be explained more fully in more detail in a later section.

4.3 Packet Delivery Rates

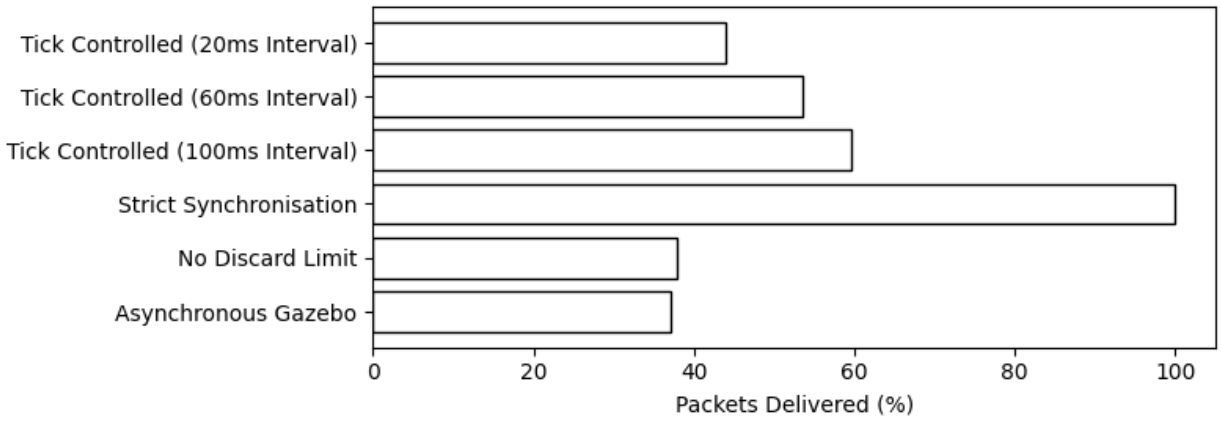


Figure 4.4: Comparison of Packet Delivery Rates

The delivery success rate of Gazebo packets was found to be 100% in all four of the analysed orchestration techniques. Note however that the *Asynchronous Gazebo* approach does not guarantee

this, while the others do. This result is likely due to the physics simulation being the slowest component in the experiment, and therefore downstream nodes are always waiting idle when new Gazebo outputs are published.

Figure 4.4 shows the internal NeuROS packet delivery rates for each of the orchestration techniques. It can be seen that only the *Strict Synchronisation* approach achieved 100% packet delivery success. It is implied [24] [16], though not verified by this work, that the NRP also achieves 100% delivery success. Furthermore, it can be seen that tick interval duration is proportional to packet delivery success. This is expected behaviour, as having a longer tick interval duration gives downstream nodes more time to complete their tasks and become ready again, before receiving new packets. Finally, we can see that both the *No Discard Limit* and *Asynchronous Gazebo* techniques had poor delivery success rates of slightly under 40%. This is likely generally unacceptable amounts of data loss, however may be tolerable for certain situations, as follows:

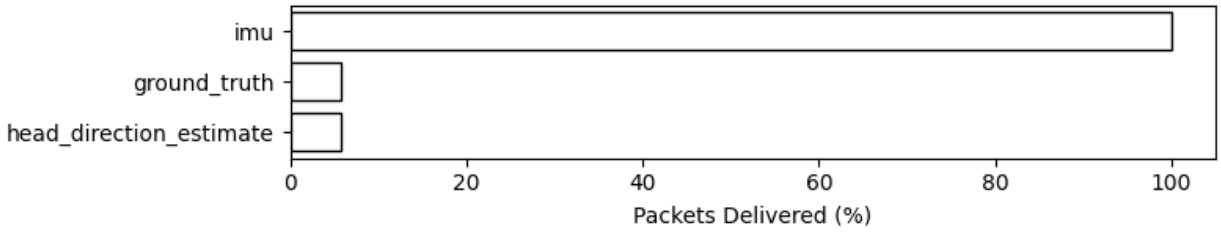


Figure 4.5: Asynchronous Gazebo Send / Receive Packet Counts

Figure 4.5 shows a breakdown of each packet's sent and received counts for the *Asynchronous Gazebo* orchestration approach. We can see that of the three packet types, the IMU connection achieved 100% delivery success and the other two just 5.7% each. This is due to the former being sent from the Gazebo node to the SNN node, with no other nodes sending data to the SNN and Gazebo taking longer to update than the SNN. This is important as it meant that the SNN node was almost always ready to receive new packets once Gazebo has finished. In contrast, the latter two message types were both sent to the same node, the visualiser, which would then become busy while it re-rendered the displayed line plot and drop any subsequent packets until finished. Since the connection from the Gazebo node to the SNN node achieved 100% delivery success, and the other two connections were for real-time visualisation purposes only, this surprisingly does in fact represent an *acceptable* delivery success rate overall. Therefore, in general we might argue that the *Asynchronous Gazebo* orchestration approach is suitable in cases where the nodes immediately

downstream from the Gazebo node complete more quickly that the Gazebo node. Clearly, this could only be determined on a case-by-case basis.

4.4 System Load

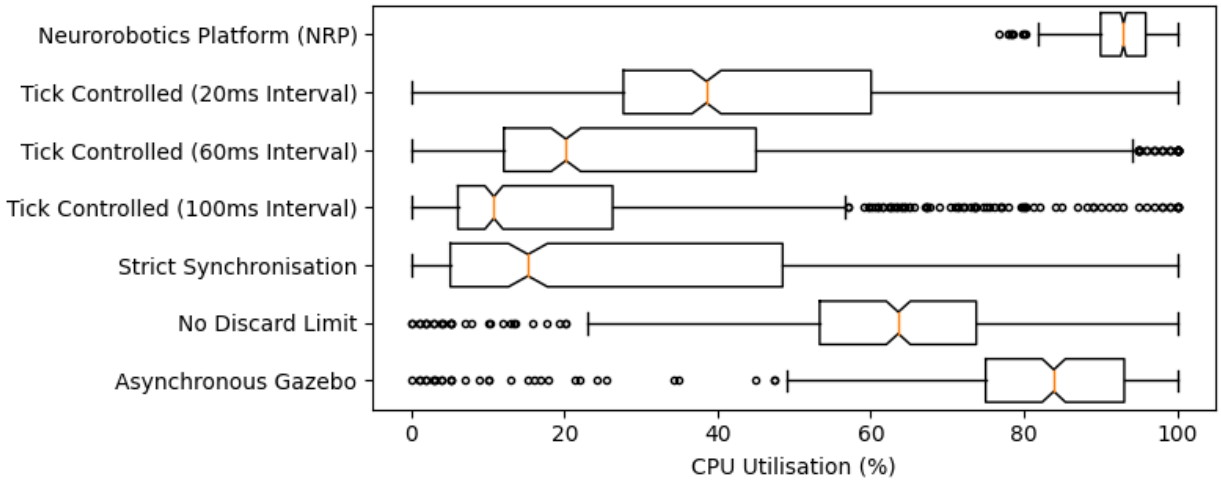


Figure 4.6: Comparison of CPU Utilisation

Figure 4.6 shows the mean CPU utilisation for each of the four NeuROS orchestration techniques investigated, in addition to the NRP. Individual CPU utilisation time series plots can be found in Appendix A.3. We can see that the NRP is extremely good at saturating the CPU, with more that 90% mean utilisation over the course of the experiment. That is extremely high and likely goes some way to explain the NRP's ability to execute more quickly than some of the other techniques, by better utilising the computational resources available. However, it is interesting to note that the *No Discard Limit* and *Asynchronous Gazebo* techniques used significantly less CPU (around 62% and 82% respectively) while still executing more quickly than the NRP. It can also be seen that tick interval duration is inversely proportional to mean CPU utilisation, which is expected given that longer durations will likely leave the system sitting idle for some periods of time. Finally, the *Strict Synchronisation* approach achieves surprisingly low mean CPU utilisation, presumably due to each of the nodes requiring different durations of time to execute, and therefore some nodes waiting idle while other are still running. It is particularly interesting that the NRP doesn't suffer from this same problem. This suggests that either there are some inefficiencies in the *Strict Syn-*

chronisation approach, or perhaps that the data visualisation node is taking significantly longer to execute than the others. This latter idea is explored more fully in Section 4.6.

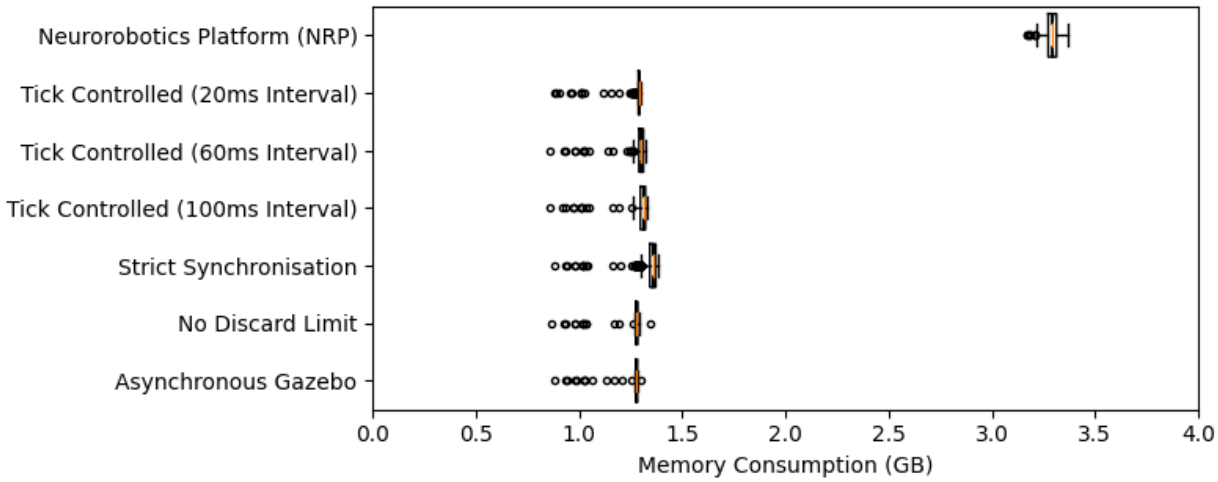


Figure 4.7: Comparison of Mean Memory Consumption

Figure 4.7 shows the memory (RAM) consumption of each of the four orchestration techniques in addition to the NRP. Individual memory consumption time series plots can be found in Appendix A.4. It can be seen that each of the NeuROS orchestration techniques consumes approximately equal amounts of memory, with little to no significant difference among them. In contrast, the NRP appears to consume around three times that of NeuROS. This could be due to the additional complex graphical interfaces and software stacks that the NRP includes, and offers some confirmation that NeuROS is in fact more lightweight in it's approach. There is very little variance within each orchestration approach as once these systems are up and running the memory consumption does not vary significantly.

4.5 Tick Interval Regularity

Figure 4.8 shows a histogram of the tick controller project's interval durations. 100% of hook invocations occurred within 10ms of the specified interval, 99.7% within 5ms, 88.2% within 2ms and 61.1% within 1ms. For comparison, Figure 4.9 shows the same interval achieved by a pure ROS2 Python node, entirely independent from NeuROS. The purely ROS2 node achieved 100% of invocations within 10ms of the specified interval, 100% within 5ms, 97% within 2ms and 73.6% within 1ms. This indicates that there is a small but measurable overhead incurred when running

under NeuROS, in particular in the increase of variance. The standard deviation for the NeuROS based tick controller interval was 1.3ms, whereas for the pure ROS2 node interval it was just 0.9ms, a difference of approximately 0.4ms. This increased variance is plausibly due to the message queue management logic, and in particular the nondeterministic nature of Python's memory allocation during insertion of priority message queue elements.

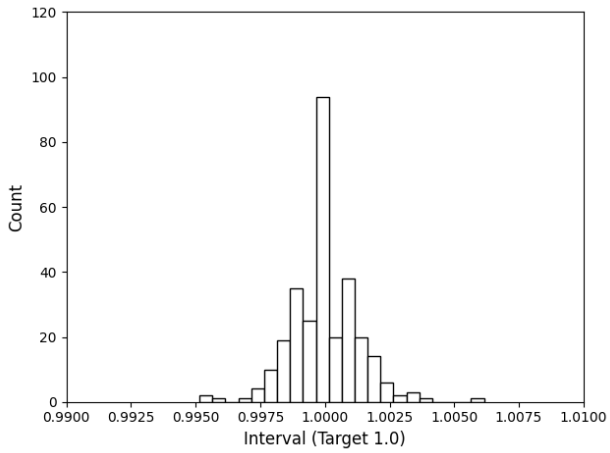


Figure 4.8: NeuROS Tick Controller

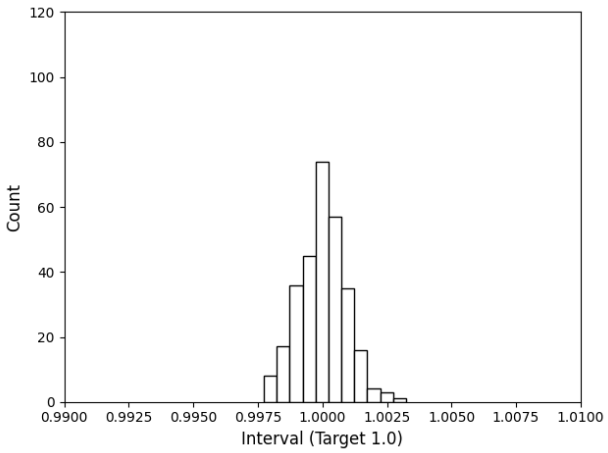


Figure 4.9: Pure ROS2 Python Node

4.6 Hybrid Approach

Given that neither *No Discard Limit* nor *Asynchronous Gazebo* dropped any packets on the IMU connection, it appeared possible that synchronisation of the ground truth and head direction estimate connections are responsible for the dramatic slowdown exhibited by the *Strict Synchronisation* approach. Therefore a *Hybrid* approach was also investigated, as mentioned previously in Section 3.9.2.

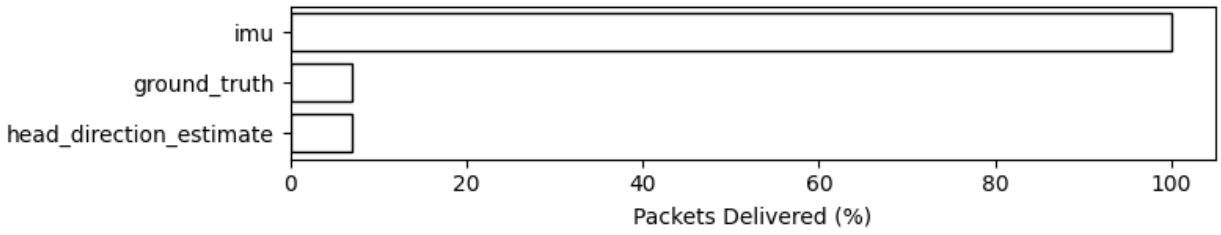


Figure 4.10: Hybrid Approach Gazebo Send / Receive Packet Counts

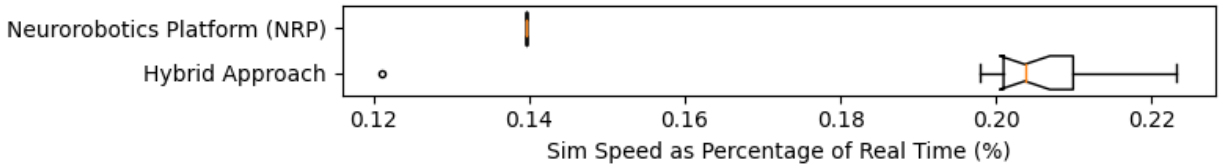


Figure 4.11: Comparison of NRP vs Hybrid Execution Speed

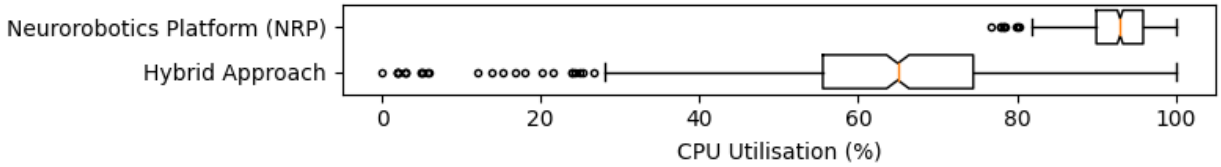


Figure 4.12: Comparison of NRP vs Hybrid Mean CPU Utilisation

This approach offers both a 100% delivery guarantee for critical IMU data, as evidenced by Figure 4.10, and a significant speed improvement when compared to the NRP. Figure 4.11 shows that while the NRP achieved 14% real time speed, the *Hybrid* approach was able to achieved 20%, a 43% speed increase. (Again, please note that the NRP execution speed is shown as having artifically zero variance.) As shown in Figure 4.12, while the NRP averaged around 95% mean CPU utilisation, the *Hybrid* approach required just 62%, a 35% reduction. Note that the *Hybrid* approach's CPU utilisation distribution most closely matches the *No Discard Limit* approach. By sacrificing the visualisation data guarantees, the *Hybrid* approach is able to guarantee the same model accuracy with a 43% increase in execution speed and 35% decrease in mean CPU utilisation.

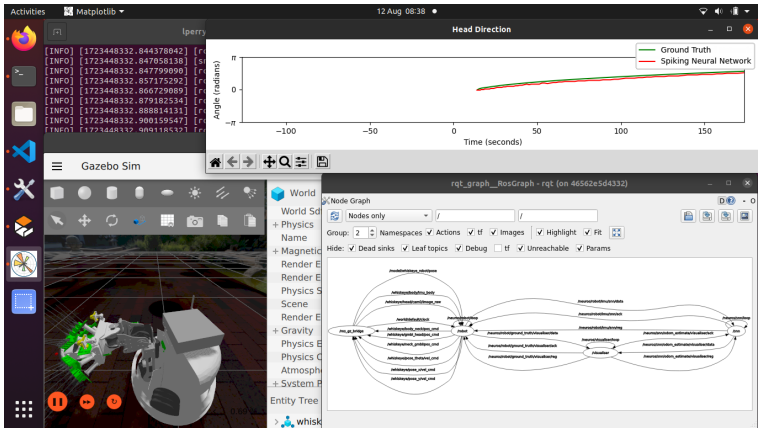


Figure 4.13: Demonstration of Head Direction Estimation Running in NeuROS

5 Discussion and Conclusion

This dissertation introduces NeuROS, an entirely new neurorobotics integration framework with highly modular containerised nodes and unrivalled orchestration flexibility. NeuROS was developed with an extremely ambitious set of aims, intended to compete with well established and far better funded platforms such as the NRP. This has largely been achieved through the leveraging of existing open source software and strict avoidance of repeating any work. By giving users direct access to native GUIs and APIs, NeuROS not only reduced it's own development cost, but also enabled users to make better use of the tools they are familiar with. An in-depth literature review has enabled a thorough understanding of the state of the art and areas which could be improved. The software implementation has leveraged the latest physics simulators, messaging transport layers and containerisation methods to deliver a modern and well designed piece of software. The comparative performance analysis presented in Chapter 4 has demonstrated both the utility of this framework's ability to partially reproduce some existing neurorobotics research, in addition to it's unprecedented orchestration flexibility. Several of the demonstrated orchestration approaches showed potential benefits beyond that which is currently on offer. The *Hybrid* orchestration approach in particular out-performs the NRP in terms of execution speed and CPU overhead, while simultaneously offering similar packet delivery guarantees for critical connections and resulting model accuracy.

5.1 Achievements and Limitations

This research originally set out to answer three main questions:

Is it possible to build a neurorobotics integration framework with fully distributed orchestration, allowing for more flexible orchestration, including hybrid approaches?

This has been demonstrated to be possible via the range of distributed orchestration techniques utilised by the representative neurorobotics example project. Chapter 3 presents an in-depth rationale and methodology for how these were implemented. The novel "ready to receive" concept, invented by this dissertation and presented in Section 3.6.1, has proven to be a highly effective building block for higher level distributed orchestration techniques. Per-connection configurability has even allowed for hybrid orchestration approaches where multiple are combined within a single running project, depending on individual connection requirements. Chapter 4 presents not just the basic efficacy of these orchestration approaches, but also the increased flexibility and potential performance improvements beyond that of the NRP.

What are the impacts of these various different orchestration approaches on performance?

The expected speed vs accuracy trade off has generally proven to be true, as is most crudely demonstrated by the tick controllers, where interval durations are inversely proportional to packet delivery rates. These are likely best suited to real-time applications.

The *Strict Synchronisation* approach is the only orchestration approach which offers equivalent guarantees to the NRP, however it appears to be slower. This seems to be due to the contention and speed of the visualiser node, which is used to plot data from both the ground truth and head direction estimate connections. This approach is suitable when no amount of data loss can be allowed on a particular connection, and was utilised in the *Hybrid* approach as discussed shortly.

There were also some more surprising findings. The *Asynchronous Gazebo* approach offers no packet delivery guarantees whatsoever yet still managed to deliver 100% of critical IMU packets successfully. Dropping these synchronisation guarantees allowed this approach to deliver a 79% increase in execution speed and 17% decrease in CPU utilisation when compared with the NRP. This approach could be applicable either when packet deliveries are not critical, or at least when Gazebo is expected to be the slowest component and some small amount of data loss is not catastrophic. It would be interesting to conduct further experiments with connections across a busy network, in order to determine if this delivery rate degrades accordingly.

The *No Discard Limit* approach is similar to *Asynchronous Gazebo* except NeuROS is responsible for stepping the physics simulator. For the *Hybrid* approach the *Strict Synchronisation* approach was utilised for the single critical connection and *No Discard Limit* for the other two visualisation connections. This results in a system that provides the same guarantees as the NRP for the critical IMU channel, but sacrifices visualisation fidelity for a 43% increase in execution speed and 35% decrease in CPU utilisation when compared with the NRP. This kind of flexible per-connection orchestration is simply not possible with any competing integration frameworks.

Can containerisation be used to encapsulate the software requirements of individual nodes, while still maintaining a high level of performance and integration?

The encapsulation of bespoke software requirements for individual nodes through the use of containerisation has been largely successful, even allowing a specific version of NEST to be used for the neurorobotics example project as was discussed in Section 5.3. However, there remains a notable limitation: The version of Python included in the container must be compatible with ROS2. This is because both the ROS2 communication layer and NeuROS user plugin are loaded into the same process, having a different version of Python for each is not currently possible. It is difficult to imagine how this could ever be supported, without resorting to a fully C++ implementation of NeuROS. This limitation aside, any software which is available on Ubuntu can be included in a NeuROS node without affecting any of the other nodes in the project, or the ability of that node to integrate. Furthermore, all of the performance analysis were conducted with containerised nodes, indicating that performance is also not badly affected. However, it would be interesting to investigate if NVIDIA GPU acceleration for docker containers [66] could be supported in the future. Not only would this potentially improve Gazebo simulation performance, but also GPUs are increasingly being used for the execution of neural networks [67].

5.2 Contribution to the Field

The contribution of this project to the field is twofold:

Firstly, by the development of NeuROS, a novel and lightweight integration framework that delivers unrivaled flexibility and modularity. This project enables neurorobotics researchers to have

unrivalled control over the way their projects are orchestrated. This control was enabled via the development of a novel building clock for distributed orchestration, as presented in Section 3.6.1, and can be used to implement project-specific performance improvements such as reducing delivery guarantees on non-critical data. This in turn may help researchers to more easily train or test their neural computation models with realistic sensor inputs and hardware physics interactions. The node containerisation allows researchers to better manage the inherent complexity of the domain, and to package, publish and share individual nodes more easily. This could lead to increased collaboration, potentially eliminating some repetition of work and reducing the cost of research and development. Furthermore, this easy package management could allow neurorobotics experimentation to be run on widely available computing resources, such as the cloud, and physically remote researchers from all over the world might collaborate on such projects more easily.

Secondly, to provide a performance analysis and comparison of the various orchestration approaches available under NeuROS against a well established competitor, the NRP. The results showed that the provided flexibility provides enhanced performance, outperforming the NRP with a 43% increase in execution speed and 35% decrease in mean CPU utilisation for the given use case.

5.3 Feature Set Status

Feature	Status
Highly flexible architecture	ACHIEVED
Broad range of orchestration techniques	ACHIEVED
Range of 3rd party tools	MOSTLY ACHIEVED
Access to native GUIs	ACHIEVED
Access to full 3rd party APIs	ACHIEVED
Encapsulation of software dependencies	MOSTLY ACHIEVED
Modularisation of components	ACHIEVED
High degree of extensibility	ACHIEVED
Thorough documentation	ACHIEVED

Highly Flexible Architecture ACHIEVED

NeuROS is capable of supporting more flexible typologies than that of the main competing framework i.e. the NRP.

Demonstrated by: The circular architecture of the example tennis project, the one-to-many and many-to-one relationships in the voting project, the incorporation of external ROS2 nodes in the physics simulation experiment, the replication of NRP-based architectures by the representative neurorobotics experiment.

Broad range of orchestration techniques ACHIEVED

NeuROS supports a far broader range of orchestration techniques than any other of the previously reviewed frameworks. It comes complete with fully distributed blocking mechanisms, tick-based controllers, fully synchronous/asynchronous Gazebo options, customisable per-connection delivery guarantees and input/output constraint modifiers, all enabled through the novel techniques presented in Section 3.6.1.

Demonstrated by: Each of the example projects, in particular the variants of the representative neurorobotics experiment.

Range of 3rd Party Tools MOSTLY ACHIEVED

This feature has not been implement as fully as was originally intended, due to significant time constraints. Approximately half of the originally planned tools are supported.

Demonstrated by: The Gazebo simulator and NEST node containers as utilised by the physics simulation project and complete neurorobotics experiment..

Access to Native GUIs ACHIEVED

NeuROS provides direct access to the native Gazebo interface, in addition to the standard ROS2 rviz and rqtgraph graphical visualisation tools.

Demonstrated by: Passing –node-graph or –visualisations to launch.py, Gazebo window launched by the representative neurorobotics experiment, as shown in 4.13.

Access to Full 3rd Party APIs ACHIEVED

Since the 3rd party tools are directly imported into the user plugins, their full native APIs are guaranteed available as standard.

Demonstrated by: The representative neurorobotics experiment node plugins e.g. snn/plugin.py use of NEST.

Encapsulation of Software Dependencies MOSTLY ACHIEVED

This feature has been relatively successful via the application of containerisation. However, some software inter-dependencies do still remain, such as the inclusion of a ROS2-compatible version of Python.

Demonstrated by: The isolation of the Gazebo simulator to only the physics node of the physical simulation example project. The isolation of a specific version of NEST to only the snn node in the representative neurorobotics experiment.

Modularisation of components ACHIEVED

This has been highly successful as is apparent by the huge amount of code reuse across all of the example projects.

Demonstrated by: The reuse of all of the standard NeuROS modules and containers across most of the example projects. The reuse of the same two nodes in all three of the physics simulation project variants. The reuse of four nodes across many variants of the representative neurorobotics experiment.

High Degree of Extensibility ACHIEVED

This feature has been extremely successful, allowing for various projects to be implemented with different architectures, orchestration techniques, message content, 3rd party tools and user plugins. In the author's opinion, the NeuROS framework feels lightweight, flexible and very accommodating of individual project requirements.

Demonstrated by: The complete neurorobotics experiment inclusion of it's a custom container for a specific version of NEST, all of the example projects custom user plugin code, the variety of

example projects and flexibility with which nodes can be recombined into new projects.

Thorough Documentation ACHIEVED

NeuROS provides automatically generated and guaranteed up-to-date documentation for every single module, class, function and class member variable in the entire system, which is far beyond any of the reviewed competing frameworks.

Demonstrated by: The documentation generated on install: ./docs/build/html/index.html

5.4 Future Work

This work has not analysed and quantified the impact of running NeuROS experiments across multiple physical machines, something which was explicitly included in the design considerations. It would be interesting to understand how this affects packet delivery rates when used on a busy network or even across the internet. In particular, does this prevent the *Asynchronous Gazebo* and *No Discard Limit* approaches from achieving 100% delivery on the IMU connection?

As previously mentioned, it may be possible to make better use of hardware acceleration by adding support for NVIDIA GPU acceleration for docker containers. Given the potential speed improvements for both physics and neural simulation alike, this appears a fruitful endeavour. It would also be beneficial to perform a lower level performance analysis. In particular, could the priority queue be made more efficient by reducing the amounts of insertions and deletions. How much of a performance overhead does Python bring, and could a C++ implementation offer greater performance still? This would also solve the aforementioned Python compatibility problem.

Finally, while it is hoped that researchers will find NeuROS to be an easy-to-use and intuitive framework, no user study was conducted to confirm this was achieved. This is because of the highly niche nature of the domain and lack of appropriate participants. Once released to the public, such a group of people may become contactable and such a user study might become feasible. This could present a useful opportunity to increase both the ease-of-use and utility of NeuROS features. One minor yet notable usability enhancement could be to automatically generate an rviz configuration file from the centralised NeuROS configuration. This would be extremely useful, relatively easy and would more fully integrate rviz into the NeuROS framework.

A Appendix

A.1 Installation

In order to install NeuROS:

1. git clone https://github.com/EMATM0055-2023/ematm0055-2023-LeePerry
2. cd ematm0055-2023-LeePerry
3. ./install.py

A.2 Execution Speed Time Series

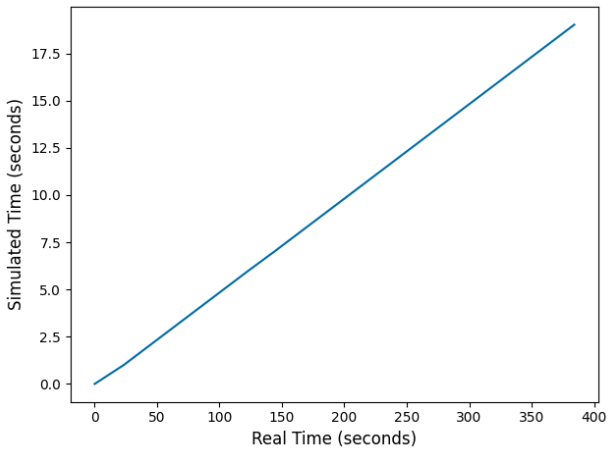


Figure A.1: Tick Controlled (20ms) Execution Speed

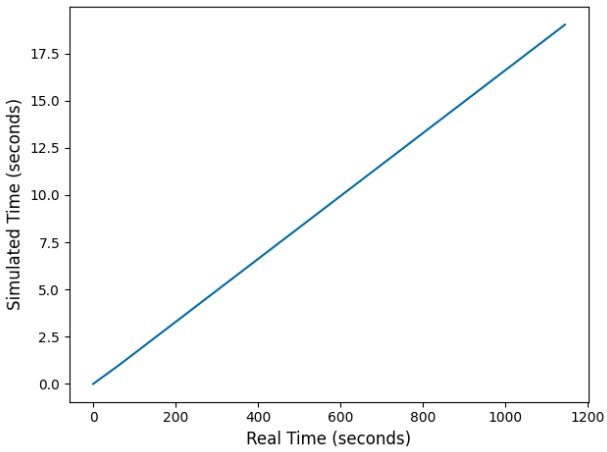


Figure A.2: Tick Controlled (60ms) Execution Speed

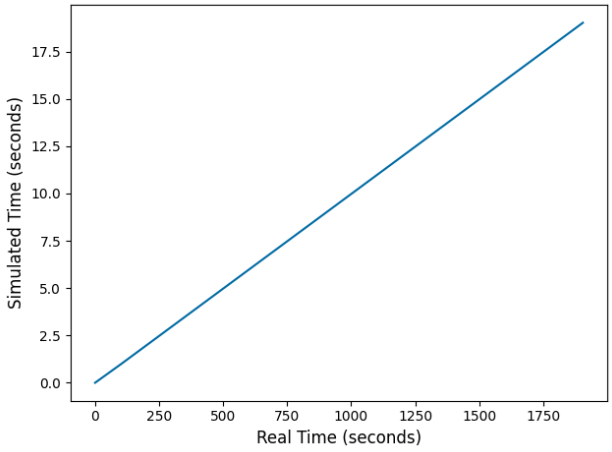


Figure A.3: Tick Controlled (100ms) Execution Speed

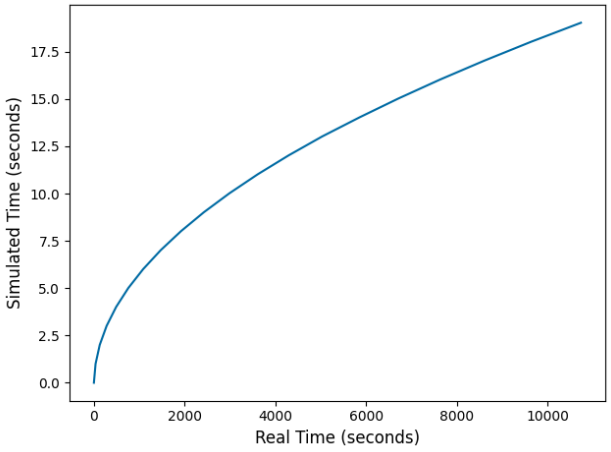


Figure A.4: Strict Synchronisation Execution Speed

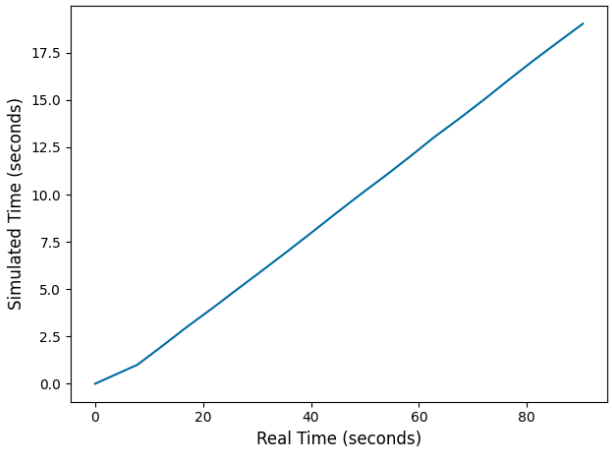


Figure A.5: No Discard Limit Execution Speed

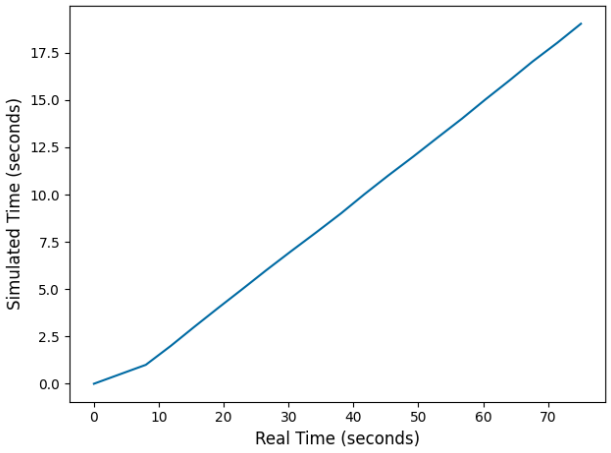


Figure A.6: Asynchronous Gazebo Execution Speed

A.3 CPU Utilisation Time Series

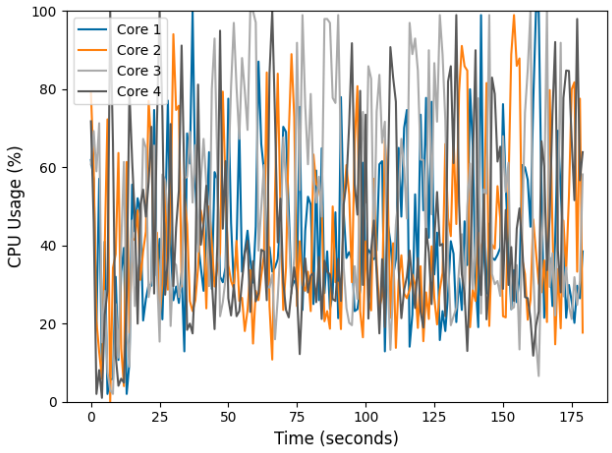


Figure A.7: Tick Controlled (20ms) CPU Utilisation

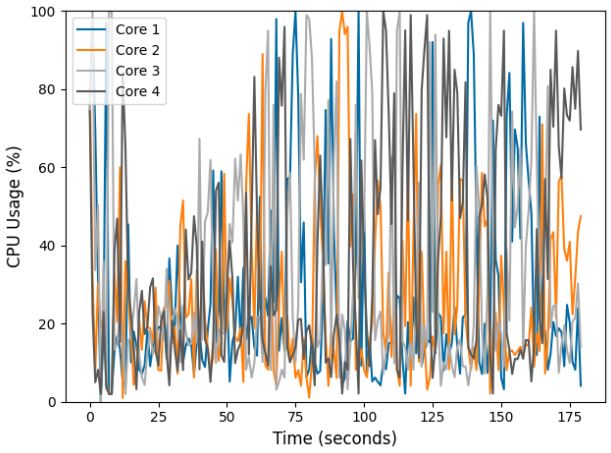


Figure A.8: Tick Controlled (60ms) CPU Utilisation

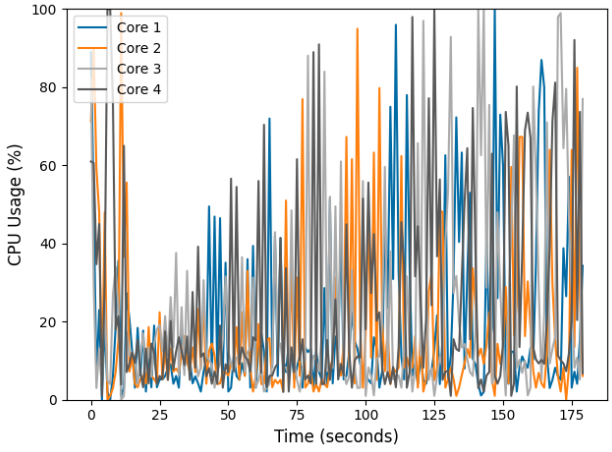


Figure A.9: Tick Controlled (100ms) CPU Utilisation

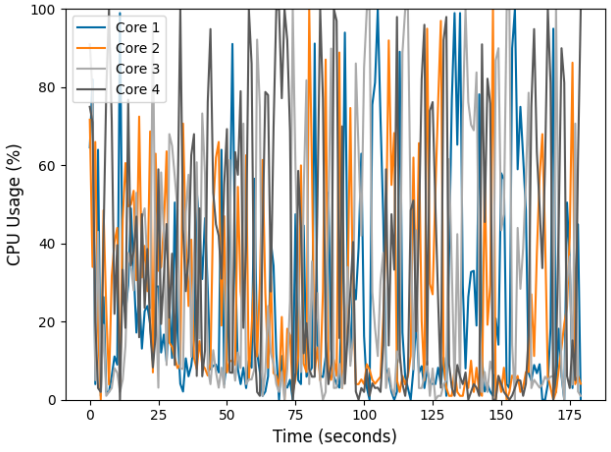


Figure A.10: Strict Synchronisation CPU Utilisation

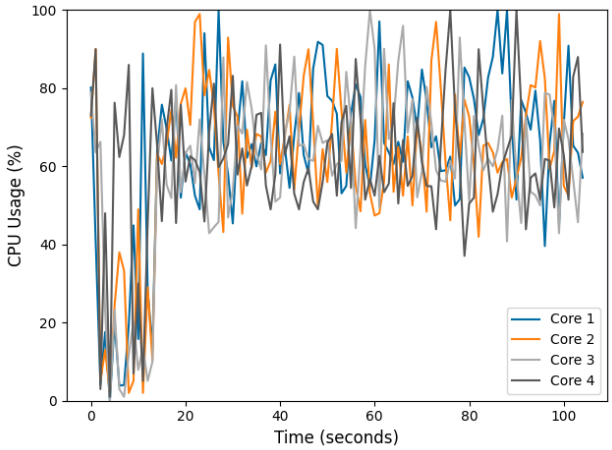


Figure A.11: No Discard Limit CPU Utilisation

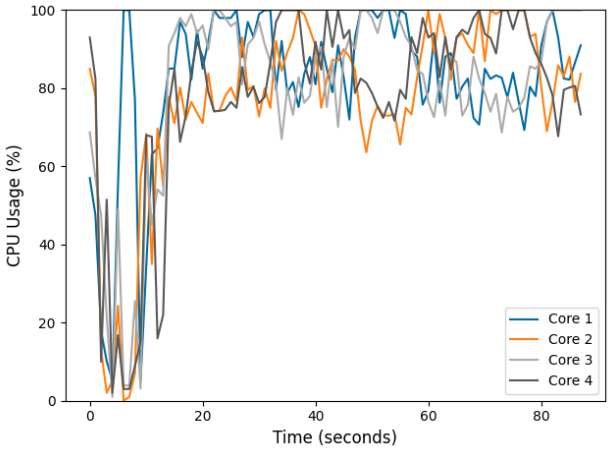


Figure A.12: Asynchronous Gazebo CPU Utilisation

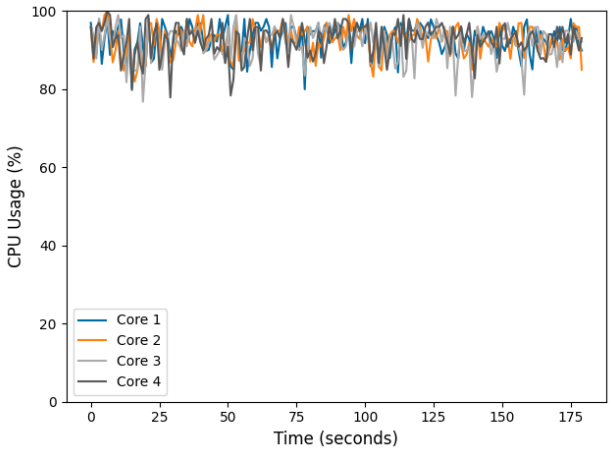


Figure A.13: NRP CPU Utilisation

A.4 Memory Consumption Time Series

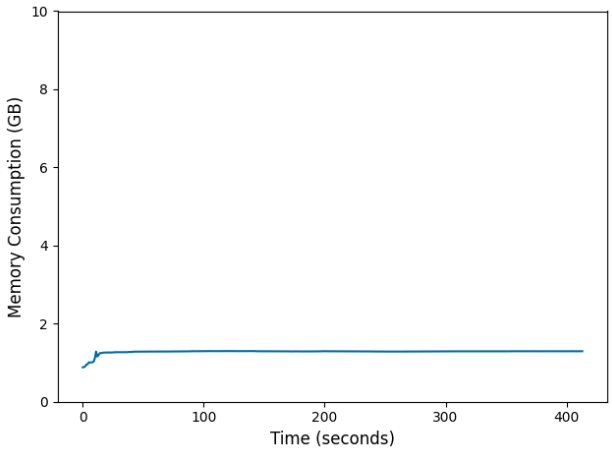


Figure A.14: Tick Controlled (20ms) Memory Consumption

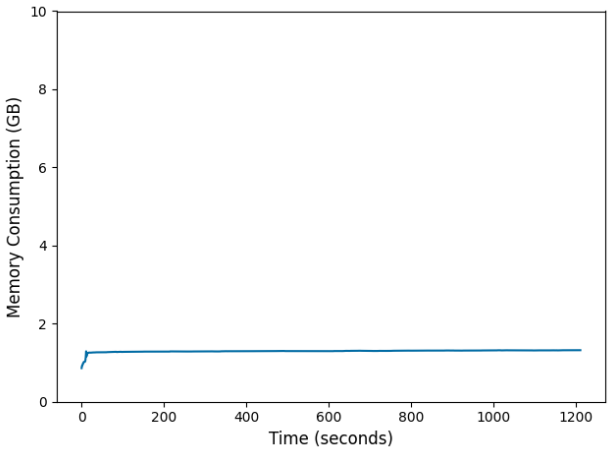


Figure A.15: Tick Controlled (60ms) Memory Consumption

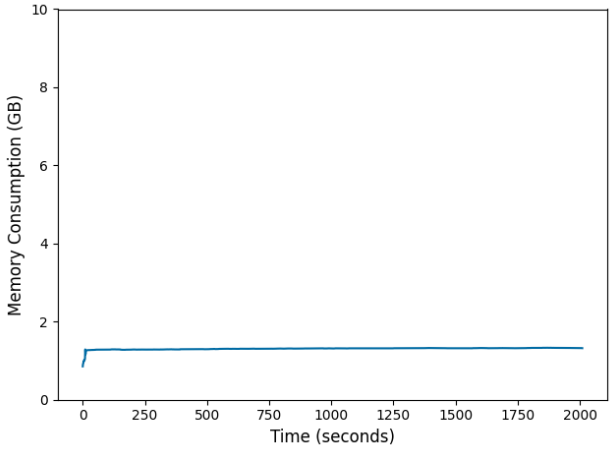


Figure A.16: Tick Controlled (100ms) Memory Consumption

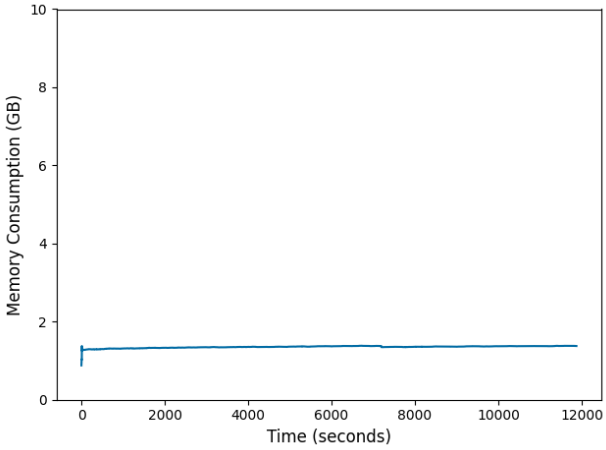


Figure A.17: Strict Synchronisation Memory Consumption

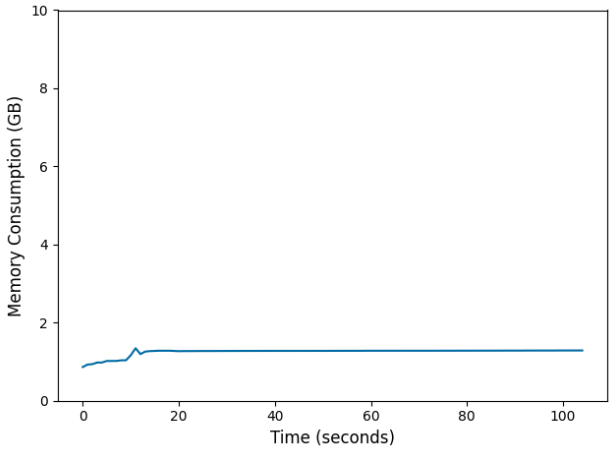


Figure A.18: No Discard Limit Memory Consumption

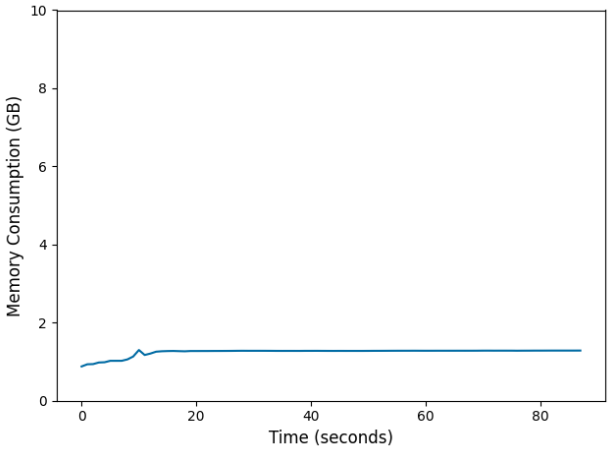


Figure A.19: Asynchronous Gazebo Memory Consumption

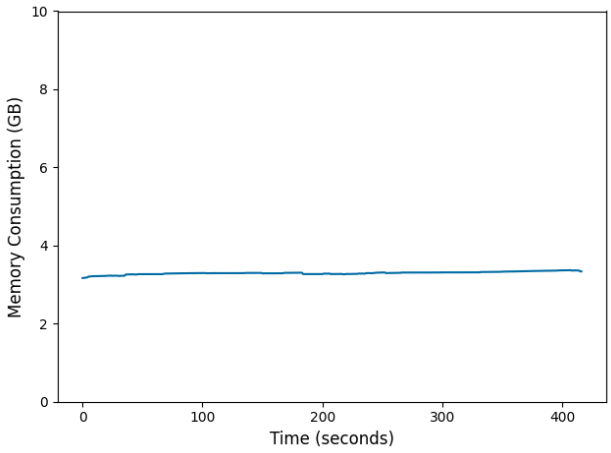
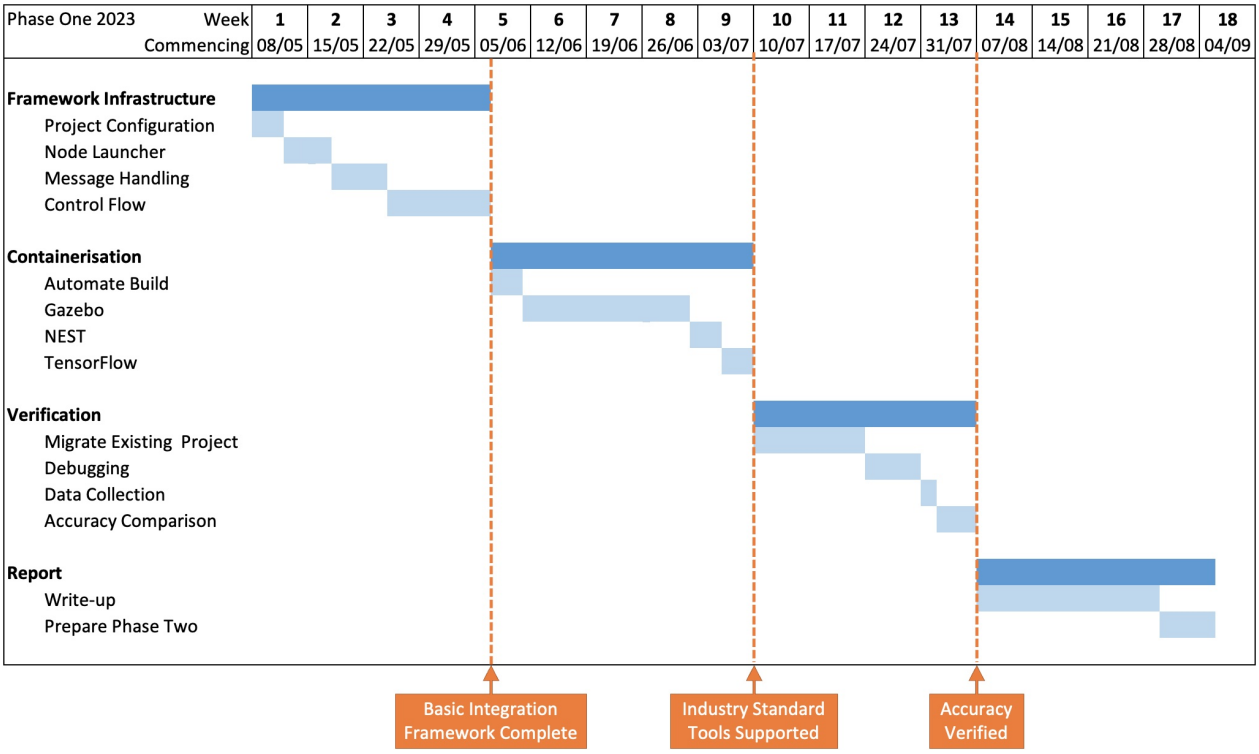


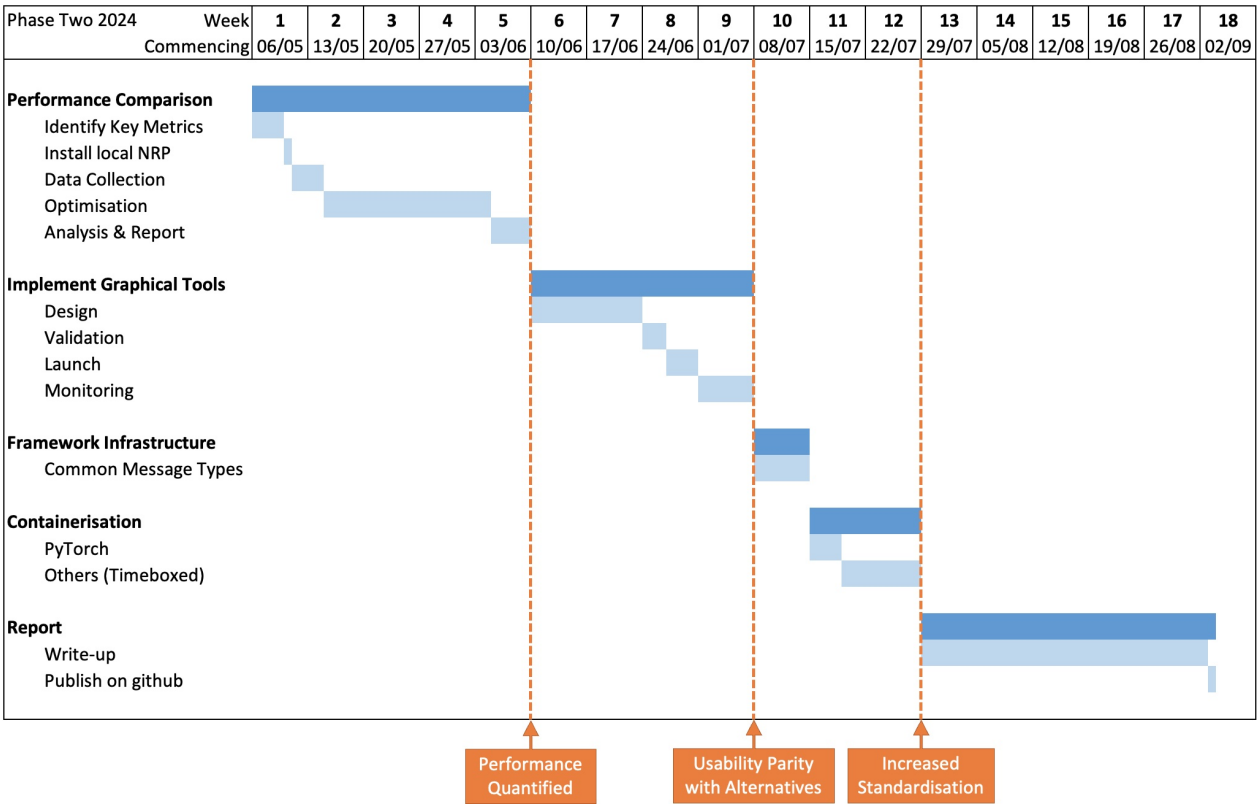
Figure A.20: NRP Memory Consumption

A.5 Planned Timetable

A.5.1 Phase One



A.5.2 Phase Two



A.6 Project Configuration Schema

```
SCHEMA = {
    "type" : "object",
    "properties" : {
        "nodes" : {
            "type" : "array",
            "minItems" : 1,
            "items" : {
                "type" : "object",
                "properties": {
                    "name"      : { "type" : "string" },
                    "container" : { "type" : "string" },
                    "plugin"    : { "type" : "string" },
```

```
            "inputs" : {
                "type" : "array",
                "items" : {
                    "type" : "object",
                    "properties": {
                        "name"          : { "type" : "string" },
                        "type"          : { "type" : "string" },
                        "external_topic" : { "type" : "string" },
                        "gazebo_type"    : { "type" : "string" }
                    },
                    "required" : [ "name", "type" ],
                    "additionalProperties" : False
                }
            },
            "outputs" : {
                "type" : "array",
                "items" : {
                    "type" : "object",
                    "properties": {
                        "name"          : { "type" : "string" },
                        "type"          : { "type" : "string" },
                        "external_topic" : { "type" : "string" },
                        "gazebo_type"    : { "type" : "string" }
                    },
                    "required" : [ "name", "type" ],
                    "additionalProperties" : False
                }
            },
            "environment" : {
                "type" : "object",
                "additionalProperties" : { "type" : "string" }
            }
        },
        "required": [ "name", "container", "plugin" ],
        "additionalProperties" : False
    }
```

```
        },
        "connections" : {
            "type" : "array",
            "items" : {
                "type" : "object",
                "properties" : {
                    "source_node"       : { "type" : "string" },
                    "source_output"     : { "type" : "string" },
                    "destination_node"  : { "type" : "string" },
                    "destination_input" : { "type" : "string" },
                    "discard_limit"     : { "type" : "integer", "minimum"
                                                              : 0 }
                },
                "required" : [ "source_node",
                               "source_output",
                               "destination_node",
                               "destination_input" ],
                "additionalProperties" : False
            }
        }
    },
    "required" : ["nodes"],
    "additionalProperties": False
}
```

References

[1] E. Brynjolfsson and A. Mcafee, Artificial intelligence, for real, *Harvard business review*, vol. 1 2017, pp. 1–31, 2017.

[2] T. H. Davenport, *The AI advantage: How to put the artificial intelligence revolution to work*. mit Press, 2018.

[3] A. Hunt, N. Szczecinski, and R. Quinn, Development and training of a neural controller for hind leg walking in a dog robot, *Frontiers in Neurorobotics* [online], vol. 11 2017, 2017, ISSN: 1662-5218. DOI: 10.3389/fnbot.2017.00018. available from: https://www.frontiersin.org/articles/10.3389/fnbot.2017.00018.

[4] B. Cheng and D. M. Titterington, Neural networks: A review from a statistical perspective, *Statistical science* 1994, pp. 2–30, 1994.

[5] J. C. V. Tieck, H. Donat, J. Kaiser, *et al.*, "Towards grasping with spiking neural networks for anthropomorphic robot hands," in *Artificial Neural Networks and Machine Learning–ICANN 2017: 26th International Conference on Artificial Neural Networks, Alghero, Italy, September 11-14, 2017, Proceedings, Part I 26*, Springer, 2017, pp. 43–51.

[6] J. Redmon and A. Farhadi, Yolov3: An incremental improvement, *arXiv preprint arXiv:1804.02767* 2018, 2018.

[7] J. Kim, S.-P. Kim, J. Kim, *et al.*, "Object shape recognition using tactile sensor arrays by a spiking neural network with unsupervised learning," in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2020, pp. 178–183.

[8] Y. Gu, Y. Wang, and Y. Li, A survey on deep learning-driven remote sensing image scene understanding: Scene classification, scene retrieval and scene-guided object detection, *Applied Sciences*, vol. 9, no. 10 2019, p. 2110, 2019.

[9] C. Duan, S. Junginger, J. Huang, K. Jin, and K. Thurow, Deep learning for visual slam in transportation robotics: A review, *Transportation Safety and Environment*, vol. 1, no. 3 2019, pp. 177–184, 2019.

[10] L. Zhang, Y. Zhang, and Y. Li, Path planning for indoor mobile robot based on deep learning, *Optik*, vol. 219 2020, p. 165 096, 2020.

[11] H.-T. Nguyen and C. C. Cheah, Analytic deep neural network-based robot control, *IEEE/ASME Transactions on Mechatronics*, vol. 27, no. 4 2022, pp. 2176–2184, 2022.

[12] A. Knoll, M.-O. Gewaltig, J. Sanders, and J. Oberst, Neurorobotics: A strategic pillar of the human brain project, *Science Robotics* 2016, pp. 2–3, 2016.

[13] S. I. Nikolenko, *Synthetic data for deep learning*. Springer, 2021, vol. 174.

[14] J. Collins, S. Chand, A. Vanderkop, and D. Howard, A review of physics simulators for robotic applications, *IEEE Access*, vol. 9 2021, pp. 51 416–51 431, 2021.

[15] Rachael Stentiford, Thomas C. Knowles, Benedikt Feldoto, Deniz Ergene, Fabrice O. Morin, and Martin J. Pearson, Integrating spiking neural networks and deep learning algorithms on the neurorobotics platform 2023, 2023.

[16] E. Falotico, L. Vannucci, A. Ambrosano, *et al.*, Connecting artificial brains to robots in a comprehensive simulation framework: The neurorobotics platform, *Frontiers in Neurorobotics* [online], vol. 11 2017, 2017, ISSN: 1662-5218. DOI: 10.3389/fnbot.2017.00002. available from: https://www.frontiersin.org/articles/10.3389/fnbot.2017.00002.

[17] T. Prescott, Brahms: Novel middleware for integrated systems computation, *Frontiers in Neuroinformatics* [online], vol. 2 Jan. 2008, Jan. 2008. DOI: 10.3389/conf.neuro.11.2008.01.051.

[18] C. Balkenius, J. Morén, B. Johansson, and M. Johnsson, Ikaros: Building cognitive models for robots, *Advanced Engineering Informatics* [online], vol. 24, no. 1 2010, pp. 40–48, 2010, Informatics for cognitive robots, ISSN: 1474-0346. DOI: https://doi.org/10.1016/j.aei.2009.08.003. available from: https://www.sciencedirect.com/science/article/pii/S1474034609000494.

[19] M. Calvo-Fullana, D. Mox, A. Pyattaev, J. Fink, V. Kumar, and A. Ribeiro, Ros-netsim: A framework for the integration of robotic and network simulators, *IEEE Robotics and Automation Letters* [online], vol. 6, no. 2 2021, pp. 1120–1127, 2021. DOI: 10.1109/LRA.2021.3056347.

[20] F. Rovida, M. Crosby, D. Holz, *et al.*, Skiros—a skill-based robot control platform on top of ros, *Studies in Computational Intelligence* [online] May 2017, pp. 121–160, May 2017. DOI: 10.1007/978-3-319-54927-9_4.

[21] Open Robotics, *Robot operating system*, https://www.ros.org, [Online; accessed February 2023], 2023.

[22] S. A. e. a. Bentaleb O. Belloum A.S.Z., Containerization technologies: Taxonomies, applications and challenges, *The Journal of Supercomputing* [online], vol. 78 2022, pp. 1144–1181, 2022. available from: https://link.springer.com/article/10.1007/s11227-021-03914-1.

[23] D. Inc, *Docker*, https://www.docker.com, [Online; accessed February 2023], 2023.

[24] F. Roehrbein, M.-O. Gewaltig, C. Laschi, G. Klinker, P. Levi, and A. Knoll, "The neurorobotic platform: A simulation environment for brain-inspired robotics," in *Proceedings of ISR 2016: 47st International Symposium on Robotics*, 2016, pp. 1–6.

[25] T. U. München, *Neurorobotics platform*, https://neurorobotics.net, [Online; accessed February 2023], 2023.

[26] EBRAINS, https://www.humanbrainproject.eu/en, [Online; accessed February 2023], 2023.

[27] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, Ieee, vol. 3, 2004, pp. 2149–2154.

[28] J. Berman, *Jsonschema 4.21.1*, https://pypi.org/project/jsonschema, [Online; accessed February 2024], 2024.

[29] C. Ltd., *Webots simulator*, https://cyberbotics.com, [Online; accessed February 2023], 2023.

[30] P. Björne and C. Balkenius, A model of attentional impairments in autism: First steps toward a computational theory, *Cognitive Systems Research*, vol. 6, no. 3 2005, pp. 193–204, 2005.

[31] M. Johnsson and C. Balkenius, Neural network models of haptic shape perception, *Robotics and Autonomous Systems*, vol. 55, no. 9 2007, pp. 720–727, 2007.

[32] Open Robotics, *Gazebo*, `https://gazebosim.org/home`, [Online; accessed February 2023], 2023.

[33] J. Collins, S. Chand, A. Vanderkop, and D. Howard, A review of physics simulators for robotic applications, *IEEE Access* [online], vol. 9 2021, pp. 51 416–51 431, 2021. DOI: 10.1109/ACCESS.2021.3068769.

[34] F. Ponulak and A. Kasinski, Introduction to spiking neural networks: Information processing, learning and applications, *Acta neurobiologiae experimentalis*, vol. 71, no. 4 2011, pp. 409–433, 2011.

[35] W. Maass, Networks of spiking neurons: The third generation of neural network models, *Neural networks*, vol. 10, no. 9 1997, pp. 1659–1671, 1997.

[36] M. Gewaltig and M. Diesmann, NEST (NEural Simulation Tool), *Scholarpedia* [online], vol. 2, no. 4 2007, p. 1430, 2007, revision #130182. DOI: 10.4249/scholarpedia.1430.

[37] J. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig, Pynest: A convenient interface to the nest simulator, *Frontiers in Neuroinformatics* [online], vol. 2 2009, 2009, ISSN: 1662-5196. DOI: 10.3389/neuro.11.012.2008. available from: `https://www.frontiersin.org/articles/10.3389/neuro.11.012.2008`.

[38] A. Martini, E. Sikander, and N. Medlani, "Estimating and quantifying the benefits of refactoring to improve a component modularity: A case study," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2016, pp. 92–99. DOI: 10.1109/SEAA.2016.48.

[39] M. Quigley, K. Conley, B. Gerkey, *et al.*, "Ros: An open-source robot operating system," in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[40] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.

[41] frovida, *Skiros github*, https://github.com/frovida/skiros, [Online; accessed July 2024], 2024.

[42] O. Robotics, *Ros2 humble*, https://hub.docker.com/layers/osrf/ros/humble-desktop/images/sha256-fc438a5a3cbadde88cd25014b398978f34b51b8cfcb611c4e6180ef5c8b39 [Online; accessed March 2024], 2024.

[43] D. Inc, *Docker compose*, https://docs.docker.com/compose, [Online; accessed March 2024], 2023.

[44] D. Inc, *Docker compose*, https://github.com/docker/compose/issues/7135, [Online; accessed March 2024], 2023.

[45] C. Ltd., *Ubuntu process identifiers*, https://manpages.ubuntu.com/manpages/xenial/man7/credentials.7.html, [Online; accessed March 2024], 2019.

[46] Open Robotics, *Domain id*, https://docs.ros.org/en/foxy/Concepts/About-Domain-ID.html, [Online; accessed July 2024], 2024.

[47] S. Raschka, J. Patterson, and C. Nolet, Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence, *Information* [online], vol. 11, no. 4 2020, 2020, ISSN: 2078-2489. DOI: 10.3390/info11040193. available from: https://www.mdpi.com/2078-2489/11/4/193.

[48] L. Perry, *NeuROS*, https://github.com/EMATM0055-2023/ematm0055-2023-LeePerry, [Online; accessed September 2024], 2024.

[49] D. Madhuri and P. C. Reddy, "Performance comparison of tcp, udp and sctp in a wired network," in *2016 International Conference on Communication and Electronics Systems (ICCES)*, 2016, pp. 1–6. DOI: 10.1109/CESYS.2016.7889934.

[50] Open Robotics, *Quality of service settings*, https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Quality-of-Service-Settings.html, [Online; accessed June 2024], 2024.

[51] Open Robotics, *Understanding services*, https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html, [Online; accessed June 2024], 2024.

[52] Open Robotics, *Executors*, https://docs.ros.org/en/foxy/Concepts/About-Executors.html, [Online; accessed June 2024], 2024.

[53] R. G. M. Jr., Priority Queues, *The Annals of Mathematical Statistics* [online], vol. 31, no. 1 1960, pp. 86–103, 1960. DOI: 10.1214/aoms/1177705990. available from: https://doi.org/10.1214/aoms/1177705990.

[54] D. H. Larkin, S. Sen, and R. E. Tarjan, A back-to-basics empirical study of priority queues, in *2014 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pp. 61–72. DOI: 10.1137/1.9781611973198.7. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973198.7. available from: https://epubs.siam.org/doi/abs/10.1137/1.9781611973198.7.

[55] Reddit, *Differences between gazebo classic, ignition gazebo, gazebo*, https://www.reddit.com/r/ROS/comments/17b0ouo/differences_between_gazebo_classic_ignition, [Online; accessed June 2024], 2024.

[56] Open Robotics, *Gazebo 11*, https://classic.gazebosim.org/blog/gazebo11, [Online; accessed June 2024], 2024.

[57] Open Robotics, *Gazebo bugs*, https://robotics.stackexchange.com/questions/104316/joint-position-controller-plugin-doesnt-subscribe-to-any-topics, [Online; accessed June 2024], 2024.

[58] Open Robotics, *Gazebo plugin migration*, https://gazebosim.org/api/gazebo/3.0/migrationplugins.html, [Online; accessed June 2024], 2024.

[59] Open Robotics, *Gz-sim*, https://github.com/gazebosim/gz-sim, [Online; accessed June 2024], 2024.

[60] Open Robotics, *Ros 2 integration*, https://gazebosim.org/docs/garden/ros2_integration, [Online; accessed June 2024], 2024.

[61] R. Picking, V. Grout, J. McGinn, J. Crisp, and H. Grout, Simplicity, consistency, universality, flexibility and familiarity: The scuff principles for developing user interfaces for ambient computer systems, *International Journal of Ambient Computing and Intelligence (IJACI)*, vol. 2, no. 3 2010, pp. 40–49, 2010.

[62] The Matplotlib development team, *Matplotlib*, https://matplotlib.org, [Online; accessed June 2024], 2024.

[63] Open Robotics, *Rqt_graph source code*, https://github.com/ros-visualization/rqt_graph, [Online; accessed June 2024], 2024.

[64] Open Robotics, *Rviz source code*, https://github.com/ros-visualization/rviz, [Online; accessed June 2024], 2024.

[65] Giam Paolo, *Psutil*, https://github.com/giampaolo/psutil, [Online; accessed August 2024], 2024.

[66] NVIDIA, *Installing docker and the docker utility engine for nvidia gpus*, https://docs.nvidia.com/ai-enterprise/deployment-guide-vmware/0.1.0/docker.html, [Online; accessed August 2024], 2024.

[67] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng, "Performance analysis of gpu-based convolutional neural networks," in *2016 45th International conference on parallel processing (ICPP)*, IEEE, 2016, pp. 67–76.