Increasing Reliability of an Autonomous Vehicle Stack in Robot Operating System 2

Author:

Lemar Haddad (22496083)

Supervisor:

Prof. Thomas Bräunl

School of Electrical, Electronic and Computer Engineering

A thesis submitted in partial completion of the requirements for the degree of

Master of Professional Engineering

(Software)

at

University of Western Australia

June 2022

Word Count: 7997 (including abstract)

THESIS DECLARATION

I, Lemar Haddad, certify that:

This thesis has been substantially accomplished during enrolment in this degree.

This thesis does not contain material which has been submitted for the award of any other degree or diploma in my name, in any university or other tertiary institution.

In the future, no part of this thesis will be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of The University of Western Australia and where applicable, any partner institution responsible for the joint-award of this degree.

This thesis does not contain any material previously published or written by another person, except where due reference has been made in the text and, where relevant, in the Authorship Declaration that follows.

This thesis does not violate or infringe any copyright, trademark, patent, or other rights whatsoever of any person.

This thesis contains only sole-authored work, some of which has been published and/or prepared for publication.

Signature: Lemar Haddad

Date: 4/6/2022

ABSTRACT

The University of Western Australia (UWA) acquired an autonomous shuttle bus, nUWAy, in 2020. Students combined open-source software with their own software to build an autonomous stack, currently in Robot Operating System 2 (ROS2) platform. The aim of nUWAy is to provide a student-run service offering autonomous rides at UWA. The operators of the service are students who should not require any technical knowledge outside of basic training to run the drives. Though autonomous demonstrations have been completed successfully by technical students, there have been overwhelming instances where nUWAy has been unable to complete drives due to software instability issues. These issues require a technical knowledge of the stack to recover from. If left unresolved, these issues will arise in regular drives where non-technical operators will be forced to engage technical resources to recover the software, resulting in unpleasant experiences for passengers and operators. This project aims to improve reliability of the software stack, so that it is more usable by non-technical operators.

Autonomous software issues and failures were tracked and categorised over 56.75 hours, to identify areas of reliability improvement. The focus areas of localisation and launch were selected for improvement. Localisation was migrated from SLAM Toolbox to Adaptive Monte-Carlo Localisation, leading to significant increases for mean time to failure (MTTF) from 8.2 minutes to 55.9 minutes. Furthermore, a software monitoring node (SMN) was designed to identify and recover from failures. SMN handled start-up of the system on 2 PCs, through a desktop icon, and monitors the status of software components. This led to increases for launch MTTF from 31.8 minutes to 1230 minutes. Overall, failures which caused system crashes were reduced, with MTTF of the system improving from 6.5 minutes to 123 minutes.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Professor Thomas Bräunl for supervising my project. His guidance and support were invaluable during this project. I am grateful for the massive learning experience he provided me, and the regular time he dedicated to myself and the other REV students. I would also like to thank Adjunct Associate Professor Robert Reid for the industry advice and experience he was able to provide for the numerous hurdles I faced within the project. Finally, I would like to thank my peers within the REV team for all the work and support which made this outcome possible. Specifically, Kieran Quirke-Brown, Thomas Copcutt, Zhihui Lai, Jai Castle and Zack Wong for their continued work alongside me on nUWAy.

1 INTRODUCTION

In early 2020, the Renewable Energy Vehicle (REV) project acquired an EasyMile EZ10 electric shuttle bus (Figure 1), fitted with sensors for autonomous driving. Since then, the REV team, made up of students and professors at the University of Western Australia (UWA), have built an autonomous driving software stack by combining open-source and student created software. The intent is for the vehicle to achieve SAE level 3+ Automation [1], meaning it can achieve conditional autonomous driving under supervision. The bus, *nUWAy*, is intended to be a platform to offer rides to students at the UWA Crawley campus, operated by non-technical people. These operators will not have any knowledge of the software stack and will only serve to set the route of the bus and intervene when necessary during drives. nUWAy has successfully completed several autonomous drives by technical operators on campus to date [2].

Figure 1: The EasyMile EZ10 shuttle bus, called nUWAy. [3]

The EasyMile EZ10 bus is outfitted with four SICK 2D LMS safety light detection and ranging (LiDAR) sensors, two 4-layer SICK 3D LD-LRS localization LiDARs, and two Velodyne VLP-16 LiDARs with 16 layers. There are mono black and white cameras fitted to the front and back of the bus, as well as two Global Positioning Service (GPS) antennas and an Internal Measurement Unit (IMU) for accurate localisation. *nUWAy* contains two PCs and a Hercules board. A dual-core Intel i7 PC does low-level processing, while an NVIDIA Xavier with a Volta architecture GPU handles most autonomous driving tasks.

The current software is built upon Robot Operating System 2 (ROS2) Foxy, implementing several open-source packages and student-written code. The current system is unreliable and requires a thorough knowledge of the software to operate and ensure that all modules are

operational. The setup or recovery of the software to attempt an autonomous drive can be lengthy, with many failures often encountered during start-up of the system and after extended periods of operation. In this state, nUWAy is unfit for handover to non-technical operators and could result in unpleasant experiences for users of the service as well as operators. This could lead to reluctancy to use the service, a negative image of autonomous vehicles and negative publicity towards the university. As a research platform, solutions are tested on nUWAy for evaluation and research, but there exists a need to focus on improving overall system reliability so that the service is less prone to failure.

We aim to utilise reliability techniques to evaluate the current state of the system and identify areas which would benefit from reliability improvement the most. Fault tolerance techniques will be explored to design and implement a system monitor which is able to recover processes automatically. The goal is to make the service more reliable and operable by non-technical people.

2 LITERATURE REVIEW

2.1 GENERAL RELIABILITY & USABILITY OF SOFTWARE SYSTEMS

The IEEE, in Std. 1633-2016, defines software reliability (SR) as "the probability that software will not cause the failure of a system for a specified time under specified conditions" [4, p. 17]. Furthermore, a fault is defined as "a manifestation of an error in the software" [4, p. 16]. A failure can be a result of a fault occurring and resulting in the loss of expected behaviour of software. SR is different to hardware, in that software will not wear-out or experience increased failures over time unless it is changed. While hardware failures are typically physical in nature, software failures stem from faults in design of the software, attributed to human errors or oversight [5]. Keene [6] presented characteristics to differentiate SR from hardware reliability (HR). Specifically, external conditions don't affect SR but can affect inputs to the software program, software faults usually become apparent under certain conditions, a software fault can cause several system failures or errors, and two identical programs will behave in the same way. This final point has the implication that, unlike HR, SR cannot be increased through running the same program in multiple instances for redundancy. Rather, redundancy can be achieved through different implementations doing the same task, termed design diversity. Furthermore, Keene [6] also introduced the repairable system concept, where periodic restarts can fix software problems.

2.1.1 Software Reliability Assessment

There are three major classes of assessing SR [7]: Black-box reliability (BBR); software metric-based reliability (SMBR); and architecture-based reliability (ABR). BBR is an estimator based on failure observations during testing or operation, wherein the software is given inputs and the user expects outputs, without considering the details of the implementation. SMBR reliability analyses the software implementation itself, considering factors such as code complexity and development process. ABR breaks down a software system into smaller components and predicts reliability of a system through the reliability data of these components.

2.1.2 System Reliability Metrics

A common way of quantifying system reliability is mean time to failure (MTTF), given in Equation 1.

$$MTTF = \frac{Total\ Time\ on\ Test}{Number\ of\ Failures}$$

Equation 1: Mean time to failure

In a repairable system, mean time to repair (MTTR) evaluates the time taken to recover the system from a failure to an operational state. For a system, maximising mean time between failure (MTBF) and minimising MTTR is ideal as the system will possess high availability. System availability is given in Equation 2.

$$Availability = \frac{MTBF}{MTBF + MTTR}$$

Equation 2: Availability. Note MTBF is equivalent to MTTF here.

2.1.3 Software Fault Tolerance

Software that is fault tolerant can detect and recover from a software fault so the system can continue to provide it's specified functionality [8].

2.1.3.1 Single-Version Software Fault Tolerance

These techniques revolve around fault tolerance in a single version of a software system.

2.1.3.1.1 Error Detection

Self-protection and self-checking are important properties of structural modules in software [9]. Self-protection is the ability of a component to protect itself from errors in its input, while self-checking is the ability of a component to prevent propagation of internal errors to other components [10]. T. Anderson [11] presents methods for error detection, as follows.

Replication checks have multiple components to perform the same function. Timing checks can be done for systems with timing constraints. Reversal checks use outputs to determine the inputs for functions which have an inverse. Coding checks have redundancy attached to the information of their outputs, which can be used to check if the output is correct. Reasonableness checks use known properties of the data to detect if there are unreasonable values which indicate error. The trade-off of error detection is system performance, as detection requires extra computation for each error case or code segment that is being checked [10].

2.1.3.1.2 Exception Handling

Functions have preconditions which, if fulfilled, allow them to behave "normally". If violated, an exception handling mechanism allows the program to raise or handle the exceptional case [12]. In fault tolerant systems an exception should be considered in the context of the event that triggered it, effects on the system and mitigation of the exception [13]. B. Randell [14] identified 3 types of exceptions, as follows. Interface exceptions occur when an invalid request is sent to a component, and it is the responsibility of the requestor to handle this exception. Local exceptions occur when an error is detected within its own operation and should be dealt with ideally in a way which the component can continue operation after exception handling. Finally, a failure exception should come from the requested component to notify the requestor that it was unable to fulfil the request it was tasked with. It is therefore essential to design the system around containment of errors, so that they are not propagated to other components.

2.1.3.1.3 Checkpoints and Restart

To recover software in a single version system, the most common technique is checkpoint and restart [6, 13]. Faults in the delivery phase are typically able to be remediated with a restart, as they are state-dependent and transient [10, 15]. In [11], the advantages of checkpoint and restart recovery is that it is independent of the fault or propagation of the fault, can detect unanticipated faults, is generalized and simple. Checkpoints can be implemented as either static or dynamic checkpoints [10]. Static checkpointing sets a checkpoint at the beginning of a component and returns to this point when restarted after an error occurs. This allows error detection at the output, generalised without having to embed these checks within the component. The problem is that the expected time to complete operation of a component increases exponentially as the size of the component increases. Therefore, this type of checkpointing is effective on small modules [10]. Dynamic checkpointing uses state information at various points in execution so that when an error is detected, restart occurs at a point closer to where the error occurred rather than starting from the beginning. In general, it is possible to achieve linear increase in execution

for operation as the component increases in size [10]. Checkpoints for state must be valid, else the error may occur infinitely if invalid state is used.

2.1.3.2 Multi-Version Software Fault Tolerance

These techniques revolve around multiple versions of software to provide a fault tolerant system.

2.1.3.2.1 Recovery Block

The recovery block was developed in the 1990s by B. Randell and J Xu [16]. A system is made up of a set of components, each receives requests and produces responses. A request which cannot be satisfied must return an exception. Recovery blocks work on the three exception types identified in [14], described in section 2.1.3.1.2. Figure 2 illustrates an idealized component.

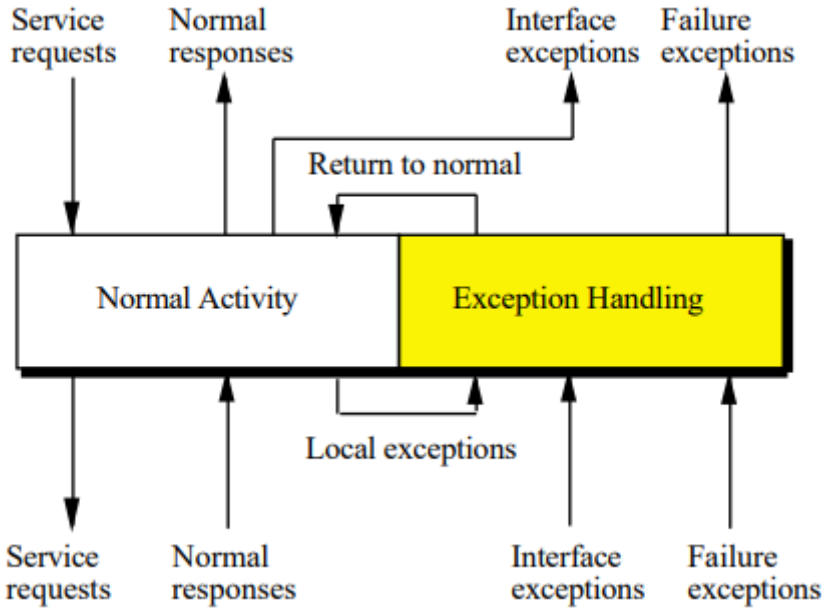

Figure 2: A component, which illustrates the flow of the three exceptions [16, p. 3]

Although this approach can provide software resilient to catastrophic failures, it is argued that this is not enough for fault tolerant software. Therefore, a recovery block (illustrated in Figure 3) should have design diversity through at least two software variants which perform the same operation in different ways, as well as an adjudicator to check the results of the software variants. The adjudicator is shared between all variants, and will chose the primary variant's output as long as it passes the acceptance tests of the adjudicator [17].

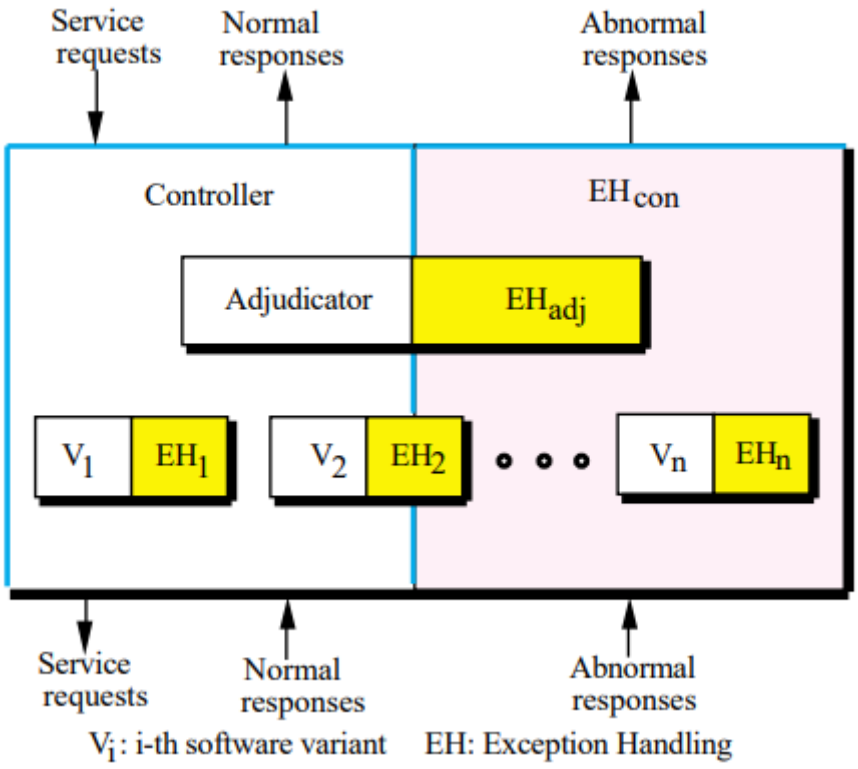

Figure 3: A recovery block component, with variants and an adjudicator [16, p. 4]

The recovery block method is complex and costly to implement due to the design diversity requirement.

2.1.3.2.2 N-Version Software

N-Version software is like N-way hardware. Redundancy of N systems exists with a voter to determine the correct output of the software [18]. The software must be design diverse in its implementation, so that even if one software version fails, the system can continue to function. It is encouraged that different languages, design philosophies and environments are used for each implementation, with development groups having as little interaction between each other as possible, to have the highest level of diversity between designs [8]. It is argued that even with this approach, different teams can still make similar mistakes, which was verified in a 1986 experiment [19]. Also, there is argument around how the voter can be certain which output is correct out of several options. Meanwhile, the recovery block method is more robust as it tries to ensure the program will not reach an incorrect state, whereas there are no controls around exception handling in N-version software. Figure 4 illustrates the model.

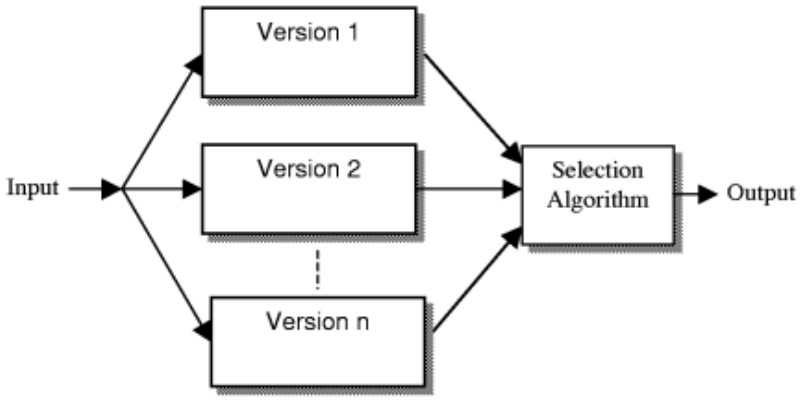

Figure 4: N-Version program model [10, p.19]

2.1.3.2.3 Self-Checking Software

Self-checking software uses multiple software variations as well as an adaptation that each software version has its own independently developed acceptance tests [10, 20]. These acceptance tests differentiate it from recovery blocks. The self-checking software model is illustrated in Figure 5. Output is taken from the highest ranked version which passes acceptance tests. In [21], a simple self-checker is presented as software embedded to continually check results over a large number of executions. It should have a high probability to eventually detect any errors, with a low probability of a false alarm. It is stated that the checks are different to other fault tolerance techniques in that they are statistically independent of the original algorithm and do not double or triple the runtime cost or overhead of the functionality being checked.

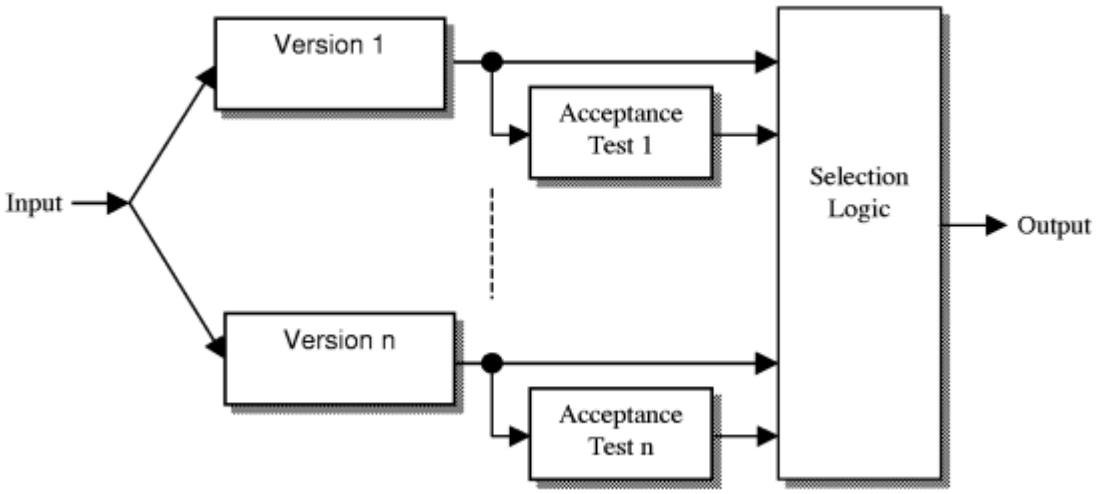

Figure 5: Self-Checking Software with Acceptance Tests [10, p. 19]

It was found in [22], self-checking software was able to detect faults that were not identifiable by N-Version software, as self-checking software is able to identify internal state of a system through tailored acceptance tests. Conversely, the study also found that the effectiveness of the acceptance tests varied greatly from programmer to programmer, with some placing tests in poor places or devising tests which flagged non-existent errors.

2.2 RELIABILITY IN ROS2

While there are several packages in ROS aimed at monitoring [23], fault tolerance [24, 25] and reliability [26], the list for ROS2 is less extensive. The overall ROS2 system is aimed at reliable computing in real time through the utilisation of Data Distribution Service (DDS) [27], but an average system is still made up of many nodes which can fail with the many inputs received in a complex environment. While reliability increases with each release of ROS2 and with regular updates to core packages [28], there is still a need for recovery when failures occur, for a system such as nUWAy to be usable by non-technical users.

2.2.1 ROS2 LAUNCH

ROS2 by default utilises the Launch system to enable running of multiple ROS2 executables, known as nodes. nodes can be launched together with a launch file (LF). Launch will detect if a node's operating system process terminates and report it to the user. Inbuilt into ROS2 Launch is the ability to respawn a node when this happens or shut down every node in the LF if the process is deemed critical by the user [29]. While this is good for some stateless nodes which fail with their operating system being terminated, it does not consider several cases. The first case that has been experienced within our software stack is where the node for safety LiDAR sensors starts before the drivers for the LiDAR are ready. In this case, the node exists but is not publishing anything, and requires a restart of the node after the driver has started to fix. The data is needed for other systems such as localisation and navigation later in the launch sequence, so the entire system must be restarted for operation. The second case is that some nodes can benefit by restarting with checkpointing, where state from before the process terminated can be utilised. An example could be saving the last position the robot was in map coordinates, and re-sending this when the localisation node is restarted, saving operators from having to re-localise the robot.

ROS2 launch supports a lifecycle manager, which can provide deterministic start up and shut down of nodes through node states. The lifecycle manager does not track situations where the nodes have crashed or are in deadlock, after the node has started up.

2.2.2 APEX OS

Apex OS is a ROS2 fork intended for safety-critical applications. The list of intended applications includes shuttle vehicles, and it is ISO 26262 certified [30]. Features include elimination of unsafe code constructs, fully deterministic software execution, lifecycle managed nodes and complete documentation. Apex OS is a commercial product, and the nature of the stringent standards it aims to uphold can rule out some open-source packages, limiting research for university students.

2.2.3 BOND

Bond creates links between two processes, which can time out or be broken by a process, allowing the other process to know if the other has terminated [31]. This can allow processes to implement recovery behaviours if termination is detected. This requires explicit creation of links between programs, by implementing the bonds on both processes. This means open source ROS2 packages need to be modified if they do not use bonds, restricting the ability to stay up to date with the main package branch and requiring knowledge of the package code to modify it for bonds.

2.2.4 SW WATCHDOG

SW Watchdog [32] is a package which implements readers, which listen on topics for output and trigger a transition to the inactive state for a node, upon a node not meeting specified DDS quality of service (QoS). It also implements heartbeat watchdogs, which asserts liveliness of a node irrespective of any published topics. The user is able implement a system-level response to restart a process, for example. This package works does not work with Fast RTPS, since it does not implement QoS. Fast RTPS is the default rmw implementation for ROS2 Foxy. There is also support for checkpoint behaviour which has not been implemented by the developers. SW Watchdog is still in active development and is intended to replace the function of Bond.

3 PROCESS / METHODOLOGY

In this section, we will apply some of the principles covered in the literature review to assess reliability and perform reliability improvement tasks based on the assessment.

3.1 RECORDING FAILURE DATA

To perform reliability improvement, a baseline assessment of reliability is required. This will be henceforth referred to as the initial reliability evaluation. For this baseline, we use BBR testing, as there are many components that work together within our autonomous system. The

aim is to identify where reliability issues lie, including frequency and severity, and use this to prioritise focus to the most problematic components of the system. During a 56.75-hour test period, attempting to autonomously drive on campus, a set of failure modes were defined, and each categorised based on which part of our autonomous system they originated from – localisation, low-level, driving and launch. Furthermore, each failure would be ranked by a severity level, of low, medium, or high. Failure severity is defined based on the impact to the operators and to the availability of the system. This is essentially how much effort it takes to recover the system back to a driveable state.

3.1.1 CATEGORIES

The low-level category encompasses any failure between the PC sending drive-control data and the motor controllers. A student-developed Hercules based interface board (Figure 6) transfers PC commands to motor controller commands for steering and velocity.

Figure 6: Interface board in nUWAy.

The launch category includes anything which prevents the system from launching the autonomous stack correctly. A failure means that part of the system has not started correctly, preventing the system from driving after starting or restarting.

The localisation category includes anything related to providing the transformation between the vehicle's real-world position and position within a map. This includes successfully loading the map, being able to specify an estimate for position with the map, termed a pose estimate, as well as maintaining a reasonable estimation of the vehicle's position within the map over time.

The driving category involves anything which affects the vehicle from getting from a starting position to a goal. This can include being unable to plan a path to a goal, unable to follow a path, operator intervention or failed recovery behaviour.

3.1.2 SEVERITY

Low severity failures are transient failures where the software stack does not crash. These can be recovered by operator intervention. For example, if the system drives to an undesirable position and abandons autonomous control, this can be recovered through the operator providing a new goal pose for the system to replan a drive toward. These low severity failures can usually be reduced over time through further tuning of parameters, which is only possible if drives are regular enough to gather good data. At the time of the initial reliability study, this was not the case.

Medium severity failures are those where one or more components of the autonomous software stack crash. The vehicle is not able to autonomously drive until the crashed system is recovered. This is done through a technical operator restarting the software stack. A non-technical operator should not be expected to identify which component has crashed or how to restart it, so we assume this is unrecoverable without technical support.

High severity failures require at least a power cycle of the entire system. The system is unrecoverable even by restarting the software stack in this case. Sometimes, these issues may not be recovered through a system restart. These are catastrophic, as they render the system undriveable for a large period.

3.2 INITIAL RELIABILITY EVALUATION

From the 56.75 hours of time on test, 686 failures were recorded. This is equivalent to a MTTF of 4.96 minutes. An extract from failures recorded is available in Appendix A. Within this, some tasks have a clear distinction of failure versus success, such as loading a map or driving to a goal position. These tasks are therefore recorded as a failure or a success whenever they are performed. For other tasks, such as correct localisation within a map, there can be a clear failure observed (such as being in an incorrect position within the map and not recovering without operator intervention), but success is observed the rest of the time, so in this case only failures are recorded.

Figure 7 demonstrates that the largest portion of failures experienced where within the localisation category. 60.2%, or 413 failures, were experienced within the testing period. To make a meaningful improvement to reliability, we will focus on this category. The next highest failure statistic came from driving tasks. As driving still required much tuning and relied on reliable localisation and launch to be able to gather driving data for improvement, this category would not be a top priority of focus. The severity of these failures was also mostly low (see

Appendix A). Therefore, it follows that if other failures are reduced, this category would become much easier to refine through tuning. Launch would also be a focus of refinement, as launch related failures prevent both localisation and driving. It is important for operators to be able to rely on a system to start correctly, to be able to reliably operate the service without technical assistance.

The most common failures were of medium severity, meaning parts of the software stack would need to be restarted by technical operators. This is illustrated in Figure 7. These failures would therefore result in downtime of the system while they were manually restarted.

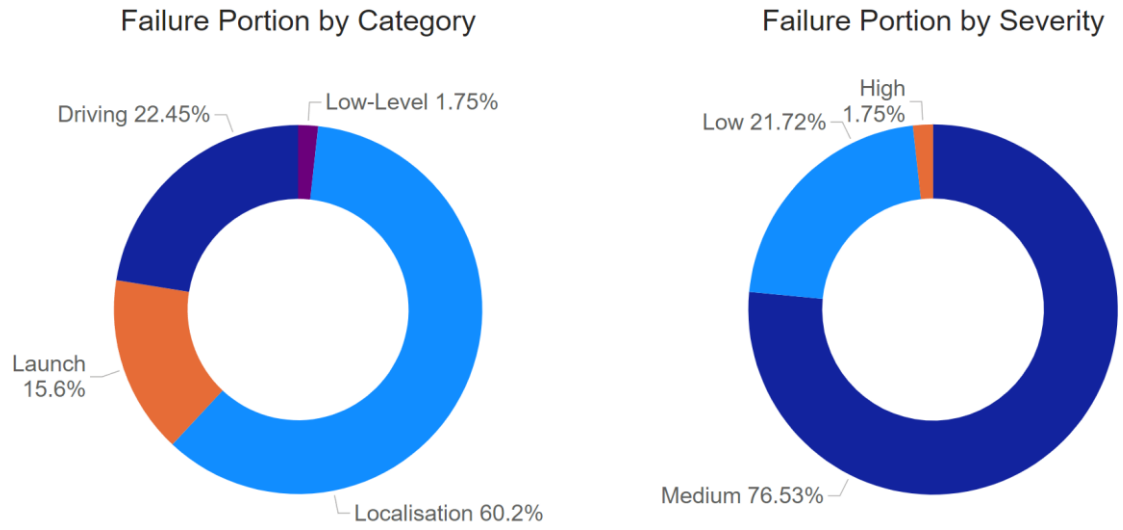

Figure 7: Failure portions by category and severity, from the initial reliability study which yielded 686 total failures over 56.75 hours.

Following the initial reliability study, in the following sections we focus on the localisation stack as well as mitigating launch issues and other general software crashes through a software monitoring node. The overall aim is to improve reliability through reducing medium severity crashes, increasing MTTF and making the recovery of the system automated so that non-technical operators can run the service.

3.3 LOCALISATION STACK

A major area for improvement was within the localisation software stack. The localisation stack is responsible for providing an estimate of a vehicle's position within a generated map, and this is used for path planning in autonomous drives (Figure 8). It is important that this estimated position is accurate, otherwise the vehicle may demonstrate erratic behaviour. We will investigate the current architecture responsible for localisation, and experimentally try to refine our approach in an attempt at improving the reliability of this aspect of the autonomous system.

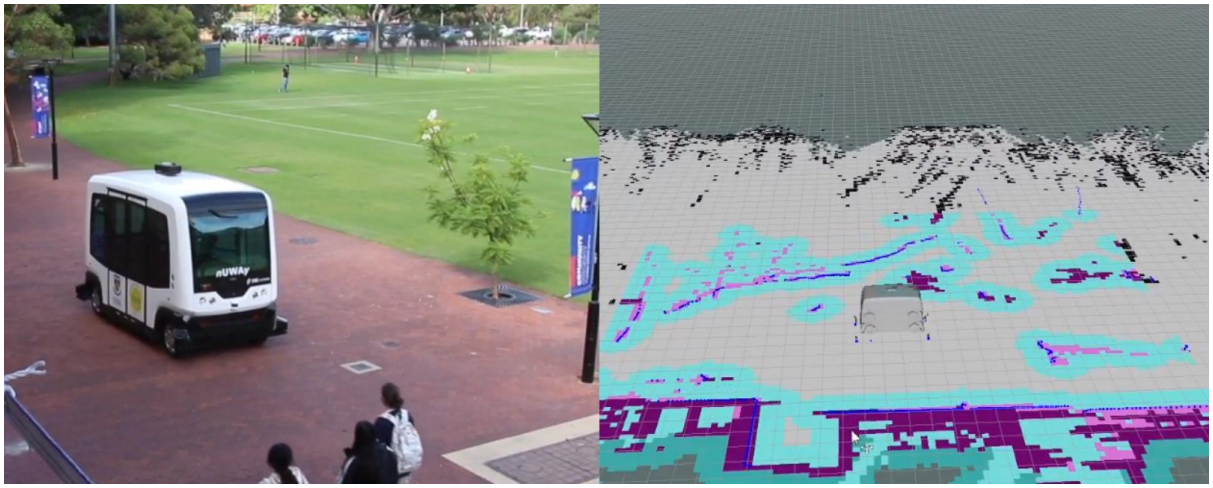

Figure 8: nUWAy (left) in the real world and localised on a generated map (right) using SLAM Toolbox localisation mode.

3.3.1 CURRENT SOLUTION: SLAM TOOLBOX

At the time of the initial reliability study, SLAM Toolbox (ST) was being used for localisation. ST is primarily a mapping tool which builds upon Open Karto [33]. Although the main operation is simultaneous mapping and localisation (SLAM), ST features a localisation mode, which does not save any alterations to the generated map. This is important, as a map's quality may degrade over time if localisation isn't accurate all the time. Figure 9 illustrates a simplified overview of the architecture. ST uses laser scans from the LiDAR sensors as well as IMU information to generate a map of an area, in mapping mode (Figure 10). This map is transferred to navigation software as an occupancy grid, which represents free, occupied, and unknown space around the vehicle. The map frame is also published, with ST providing transformation of vehicle coordinates into map coordinates, to represent the vehicles position within the map. In localisation mode, the map is loaded into an occupancy grid, but is unaltered beyond local changes, which are unsaved. Therefore, the map will not change from day to day but can be used for localising within.

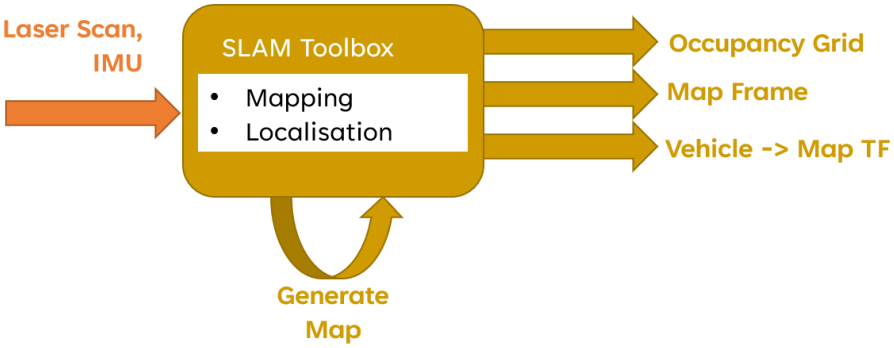

Figure 9: ST uses both laser scans from LiDAR sensors as well as IMU readings for odometery, to generate a map of an area in mapping mode. Localisation mode uses this map to output an occupancy grid as well as a map frame and transformation of vehicle coordinates to map coordinates, which are used for navigating the vehicle around an area.

Figure 10: A map created with ST (left) along with the UWA paths this map represents marked in red (right).

The main reliability issues experienced with ST were loading maps and providing pose estimates. Map loading is the process where a map is deserialised into an occupancy grid, so that it can be utilised for localisation. When a map was attempted to be loaded, it would often

crash the entire ST process, with no descriptive error messages. Each time a map would be loaded, a pose estimate would be required to be provided as an estimate of the vehicle's initial position within the map. This was because GPS coordinates would not align exactly to the map beyond the initial point at which the map started. Sometimes, providing this pose estimate would crash ST, with a similar undescriptive error message. Since many times multiple pose estimates are required for good localisation performance, this would often crash the system, and then the map would have to be reloaded. This would result in a cycle of trying to recover the system on start up. Several experiments would be trialled before reconsidering the entire architecture for localisation.

3.3.1.1 Experiment 1: Parameter Tuning of SLAM Toolbox

Parameter tuning was performed to reduce the computational load ST would require. The most significant parameter change was reducing `resolution` from 0.05 to 0.3 (Figure 11). With this change, we experienced much less lag in our operation with large maps, as well as being able to generate even larger maps without significant performance degradation. Loop closures were also more accurate. Although this was advantageous for mapping, the localisation issues with map loading and pose estimates were not resolved.

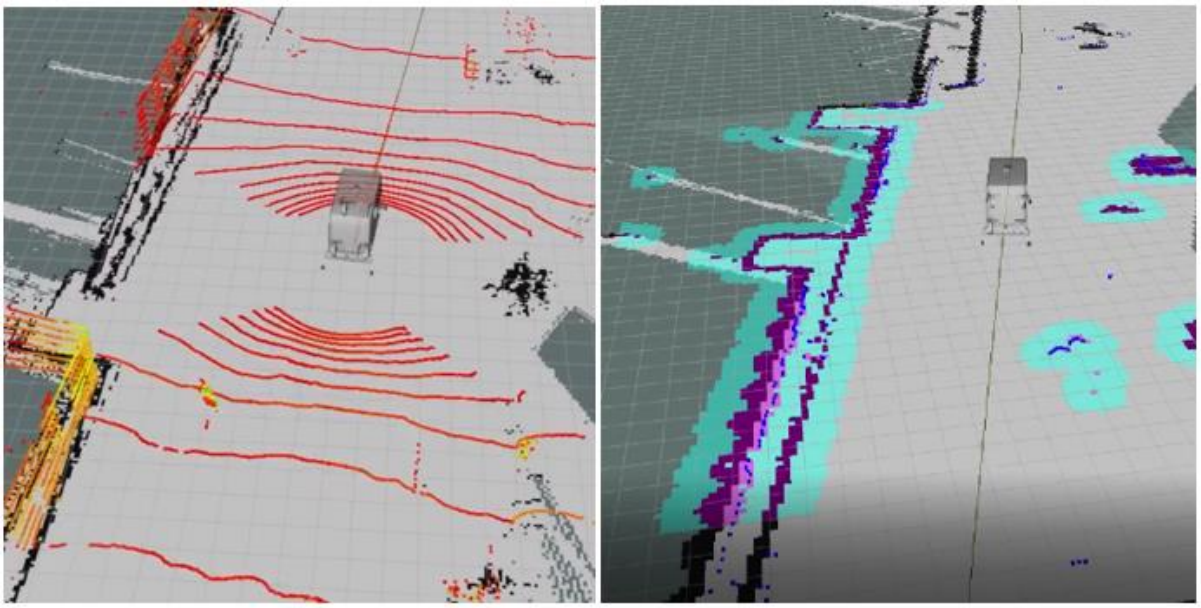

Figure 11: A resolution of 0.05 (left) and 0.3 (right) for the same area of UWA. Note how the scans can converge to a single black line representing a wall on the right.

3.3.1.2 Experiment 2: Different Installation Methods for SLAM Toolbox

ST was initially installed from the deb package manager. Building ST from source did not provide any improvements to our reliability issues. Running our software stack on a more powerful PC also still experienced crashing when loading the same maps. It was recommended

for production robots that ST be installed as a snap, an isolated install. This was claimed to have optimisations improving performance by 10x [34]. Unfortunately, the snap was not compiled for the ARM64 architecture that the Nvidia Xavier PC uses.

3.3.1.3 Final Solution: AMCL for Localisation

Adaptive Monte Carlo Localisation (AMCL) uses an adaptive particle filter size to localise the robot against a map [35]. AMCL uses a static portable gray map (PGM) file, rather than a pose graph for its map representation. While ST serlialises a posegraph and data file for each map it stores, of size approximately 300 megabytes for the campus, AMCL uses a PGM file, size 4 megabytes for the same area. The new solution is illustrated in Figure 12. ST is still used for mapping as it can save maps as PGM files. Map Server then loads in a map into a ROS Occupancy Grid, which AMCL can use to localise against and provide the transformation from vehicle coordinates into map coordinates. This solution was found to have favourable results, which will be demonstrated in section 4.1.

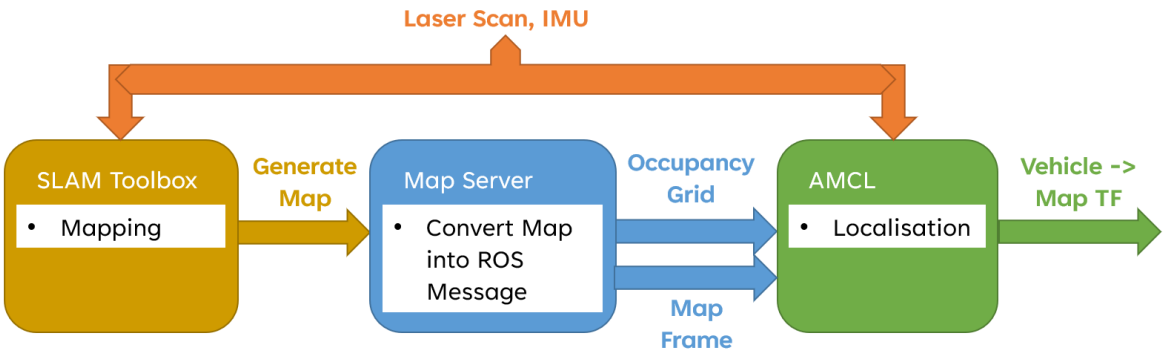

Figure 12: ST is used for mapping and saving a map as a PGM file. A map server will load this static map along with publishing a static frame to represent the map coordinate system. AMCL will localize the vehicle to the map, creating a transformation from vehicle coordinates into map coordinates.

3.4 SOFTWARE MONITORING NODE

To reduce technical operator assistance on board nUWAy, a system is required which acts as a technical operator may, henceforth referred to as software monitoring node (SMN). The SMN should sequentially launch processes, ensuring each has started properly. SMN should regularly monitor ROS2 nodes and topics to ensure they are still running. Finally, the SMN should report any crashes to the operator, and attempt to recover the software stack, by restarting components as well as providing checkpointing where it could reduce the intervention an operator would need to perform. The SMN should be generic, so that it can continue to be utilised as the software stack evolves with more features or different packages.

The aim of SMN is to enable non-technical operators to run nUWAy, even if software crashes occur. SMN is modelled around principles demonstrated in Table 1, from *IEEE Std 1633-2016.*

Table 1: An extract from Table 4 of [4, p. 40]. This protection method has guided the design of our SMN.

Reliability consideration	Applicable methods
Software	
Protection from inadvertent operations, out-of-sequence commands or data, and environmental affects	Self-checking of software events and data integrity. Watchdog timers that monitor the software's operation and trigger a reset or alternate system. Fault/failure detection, isolation, and recovery (FDIR). — Detection—how the SW knows one or more fault or failures have occurred — Isolation—how the SW knows what failed and when — R_____—what the SW should do about it —Recovery—(human interaction, auto-reset, retry rate, revert to backup, etc.) —Reduced operations —Partial) Reset —Restart —Reload —Restore

3.4.1 Design Options: Self-Checking Component

Three design options were considered for the self-checking aspect of the SMN.

3.4.1.1 Option 1: Bond

Bond, presented in section 2.2.3, could be utilised to monitor the software operation and trigger restarts when necessary. This would provide advantages of very fast detection speed and high reliability when detecting failures. This solution requires modification of each ROS2 package used in nUWAy, to create a bond between the package and SMN. This is infeasible when considering the imposed constraint of a generic system which can continue to be used as nUWAy's software changes.

3.4.1.2 Option 2: SW Watchdog

SW Watchdog, discussed in section 2.2.4 contains watchdog timer functionality. Like bond, this allows for fast detection of failures, as well as not requiring modification to each package, unlike Bond. This solution requires either Cyclone DDS or Fast DDS, rather than Fast RTPS, for the rmw implementation that ROS2 uses [32]. Furthermore, each package needs to be encapsulated as a ROS2 lifecycle node. This means modification is required to setup operational states for each package.

3.4.1.3 Option 3: Node and Topic Checker

ROS2 features API calls to retrieve a list of active nodes within the system. It also allows subscription to topics within the system, which can be used to assert liveliness of a process

within the software stack. This does not require any modification to ROS2 packages in the system. The disadvantage of this approach is speed of detection of failure. It often takes several seconds for a node to be removed from the active node list. Topic checks need to allow for significant time redundancy to account for temporary performance degradation when monitoring liveliness of a topic, to avoid false positive failure detections.

3.4.1.4 Final Decision

For the specific conditions nUWAy required, as an experimental vehicle still in active development, Option 3 was selected. It would provide SMN with the most flexibility in configuration. Even though failure detection speed would be significantly slower than bond or SW Watchdog, the SMN is not intended as a safety feature, as the purpose of the operator is to intervene if any undesirable behaviour is observed. Furthermore, Navigation2, responsible for autonomous vehicle control, utilises Bond as well as a lifecycle manager. If any node associated with autonomous driving fails, the entire stack will immediately be shut down, disabling autonomous control.

3.4.2 DESIGN

Figure 13 demonstrates the general operation of the SMN. A configuration file is provided, which defines the ROS2 nodes to launch, in the form of LFs. Also provided is a frequency, which defines how often the checks to the system occur. For each LF provided, the system will start the process within a new window. After it has started, the LF will be checked to ensure it is running. If it has not started properly, the LF is attempted to be recovered through a restart, with a larger timeout period. If recovery is unsuccessful, the system is shutdown, and this is reported to the user. Otherwise, each LF is started sequentially and ensured to be running before the next file is started. Once all the files are running, the system will periodically check the entire system. If a component of the system is unable to be recovered during this phase, the entire system is shut down and started sequentially again, to clear any state-related issues that may have arisen. This behaviour continues until the operator requests that the system is shutdown.

```
function monitor_node(Argument config, Argument frequency) {
    monitor <- new tmux session
    for launchfile in config
        pane <- monitor.create_new_window
        start(launchfile, restart=false, time_to_start=launchfile.tts)
        res <- check_file(launchfile)
        if res is FAILED:
            status <- recover_system(res, files)
            if status is FAILURE
                shutdown_system()
                GOTO top of function Monitor
            endif
        endif
    endfor
    while not SIGINT
        res <- check_system(config)
        if res is FAILURE
            GOTO top of function Monitor
        endif
        sleep(1 / frequency)
    endwhile
    shutdown_system()
}
```

Figure 13: Psuedocode of the overall operation of the SMN.

In Figure 14, an LF is started, or restarted if it is already running. After starting, the system waits a user-specified number of seconds to allow the LF to start. The file is then checked by the process described in section 3.4.1.3, the pseudocode is given in Figure 15. Whenever a topic is checked, the timeout period for declaring the LF as failed is $1/10^{th}$ of the expected frequency of the topic. This allows for any temporary performance degradation of the system, minimizing false positives of LF failures.

```
function start(Argument file, Argument restart, Argument time_to_start) {
    if restart
        stop_file(file)
    endif
    launch_file(file)
    sleep(time_to_start)
}
```

Figure 14: Psuedocode of SMN component which handles startup of a system component.

```
function check_file(Argument file) {
    nodes <- get_node_list()
    if file.nodes not in nodes
        return FAILED
    for topic in file.topics
        res <- check_topic(topic.name, timeout=10*(1/topic.frequency))
        if res is NO_MESSAGES_RECEIVED
            return FAILED
        endif
    endfor
    return RUNNING
}
```

Figure 15: Psuedocode of SMN component which checks if a node within the system is running, by checking the node list as well as listening on topics the node publishes.

The entire system is checked on a user-specified frequency, as shown in Figure 16. If any LFs are found to have failed, attempted recovery, in Figure 17, will be performed. This attempted recovery consists of restarting the LF, and if it is successfully recovered, performing checkpointing behaviour to recover any necessary state information. If the system is not recovered, the timeout for the file to start is increased by 50%, to allow for any system performance degradation. If the LF is unrecoverable, this is reported to the user, and a full system restart is triggered.

```
function check_system(Argument config) {
    for file in config
        res <- check_file(file)
        if res is not RUNNING
            res <- recover_system(file)
        endif
        return res
    endfor
}
```

Figure 16: Psuedocode for the SMN component which checks the entire system, file by file. If a node has failed, it will perform recovery behaviour.

```
function recover_system(Argument errors, Argument launchfile) {
    tts <- launchfile.tts
    while errors is not empty and retries < 3
        tts <- tts * 1.5
        start(launchfile, restart=true, time_to_start=tts)
        errors <- check_file(file)
        retries <- retries + 1
    endwhile
    if errors is empty
        checkpoint(launchfile)
        return SUCCESS
    else
        return FAILURE
    endif
}
```

Figure 17: Psuedocode for the SMN component responsible for recovering a file. It will restart the file, check that it has been recovered, and then perform checkpointing.

4 RESULTS AND DISCUSSION

Another reliability assessment was performed after the SMN and AMCL were implemented into the system. This will be referred to as the final reliability evaluation. Within this evaluation, the same failure modes from Appendix A were recorded, to measure the extent of improvement to our system. Since this testing period was 20.5 hours, compared to the initial period of 56.75 hours, we will use MTTF to make comparisons. We start by comparing AMCL to ST for localisation, followed by discussion of SMN and finally demonstrate overall reliability improvements to the system.

4.1 LOCALISATION RELIABILITY IMPROVEMENTS

Overall, large improvements in reliability were observed from moving our localisation system from ST to AMCL. Our focus will be on the issues of loading maps and providing pose estimates, which were discussed as the most impactful issues of ST in section 3.3.1.

4.1.1 MAP LOADING

Demonstrated in Figure 18, ST was failing approximately 75% of the time when attempting to load maps. Maps of the same size and areas of campus were used for both tests. AMCL, in comparison, was successful in loading the maps over 97% of the time. Note that AMCL was restarted many times in testing to attempt to get a similar frequency of map loading as was being performed by ST, even though this was not required by the system.

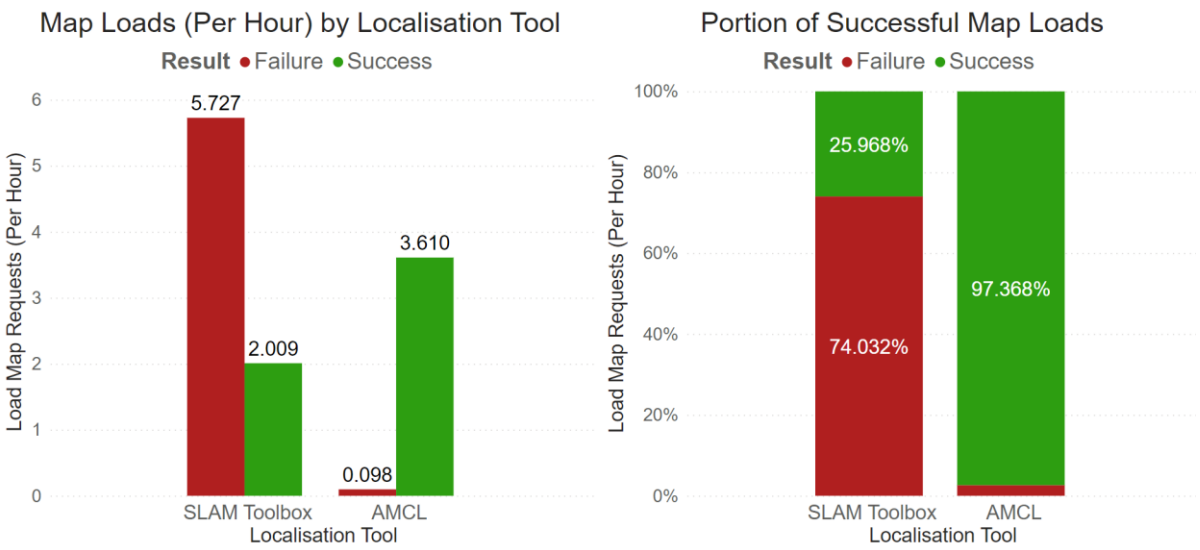

Figure 18: AMCL can successfully load maps far more consistently than ST. The same data is shown in two formats, to illustrate both the rate of failure per hour as well as the portion of successes versus failures.

4.1.2 Pose Estimates

After loading a map, a pose estimate is required to provide a better estimate of the vehicle's position within the map, after which the localisation tool will be able to localise the vehicle. Several of these may be required to refine a good estimate, due to human error when inputting these estimates in the user interface. About 1/3rd of these would fail in ST (Figure 19) crashing the process. This would mean the process would have to be restarted, including reloading the map. With AMCL, the success rate of this operation was over 95%. This had the effect of increasing operator confidence that this operation would work, meaning the operator could give multiple pose estimates in a row to refine their estimate, without worrying about the system crashing. This effect can be seen in the frequency of pose estimates per hour by tool. Even though AMCL maps were successfully loaded about 80% more per hour compared to ST, the number of successful pose estimates for AMCL was 192% higher than ST. Operators were less concerned with providing a good estimate the first time to minimise the risk of the system failing, so would refine their estimate over more attempts. It should also be noted that the failure mode for AMCL pose estimates was different to ST. AMCL would freeze for several minutes before updating the pose estimate in failures recorded, rather than crashing. These instances were still recorded as failures, the response time of the system was unacceptable.

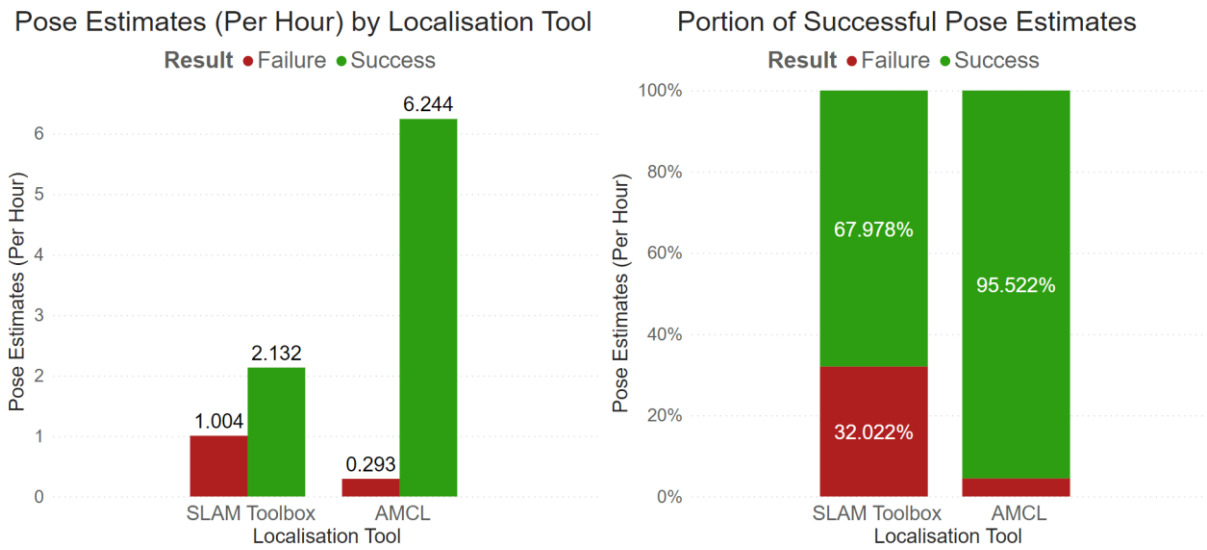

Figure 19: Pose estimate requests in AMCL less likely to fail than within ST. The same data is shown in two formats, to show the rate of failure per hour as well as portion of successes for each localisation tool.

4.2 SOFTWARE MONITORING NODE

SMN was implemented as a ROS2 node within Python, and can be found at [36]. The reasons for using Python was mainly for the libtmux library [37], allowing Tmux sessions to be created and controlled within Python. The system was tested both in a local ROS2 environment with the Gazebo simulator, as well as on nUWAy. Whenever a LF failed, the software would report this to the user. For technical diagnosis, the Tmux window which the failure took place in would also be provided, so that a technical operator could attach to the session and examine the logs for debugging. This is demonstrated in Figure 20.

```
[monitor-1] [INFO] [1650788586.890416507] [monitor]: Stopped process for launch file safety_lidars.launch.py
[monitor-1]     Relevant tmux info (debug):
[monitor-1]         Tmux session: monitor
[monitor-1]         Window: safety_lidars.launch.py
[monitor-1]         Pane Index: 0
[monitor-1]     Please attach to the session for logs, if this shutdown was unexpected.
```

Figure 20: The monitor reporting an issue which has triggered a restart for the safety LiDAR node. Debug information is provided for technical operators.

The most notable benefit the SMN provided was in launch of the system. Firstly, the SMN was packaged as a single desktop icon, which would start a SMN session on both PCs and start everything sequentially. This meant start-up was easy and often successful, even resolving issues such as a race condition we experienced with LiDAR drivers which would cause start up to fail previously. The improvements to launch of the system will be demonstrated in section 4.3.

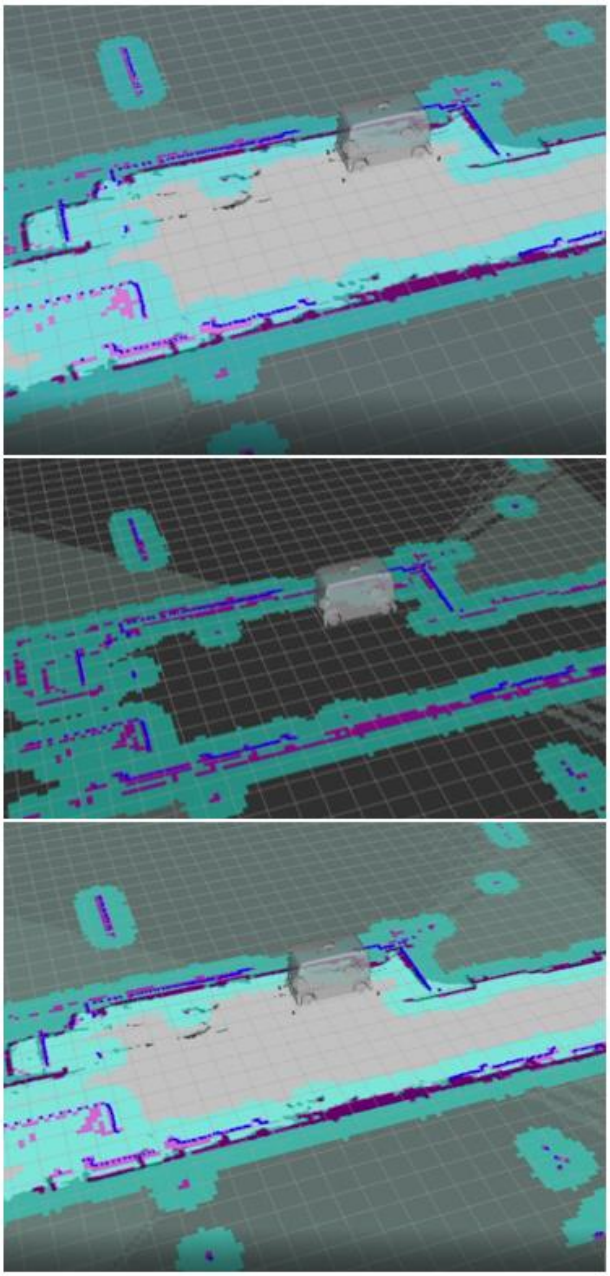

Figure 21: RViz before crashing (top), after being restarted and missing the map (middle). SMN will recover the last loaded map (bottom).

4.2.1 CHECKPOINTING

Checkpoints were implemented in two scenarios. The first checkpointing system implemented was to reload the map if our visualisation software, RViz, had crashed and was restarted. In this situation, usually a technical user would need to send a command to reload the specific map file. Although the system was still capable of autonomous control and this was an issue purely with the visualization software, this would be crucial for an operator to ensure localisation remained correct and monitor the planned path that the vehicle was driving along.

Therefore, this checkpoint was useful for continuing a drive which would otherwise be aborted before the SMN was implemented. This is demonstrated in Figure 21.

Checkpointing was also implemented which could recover the position of the vehicle within the map coordinate system. Periodic vehicle positions were saved, so in the event of a failure of the localisation system the previous known position could be used to reinitialise the vehicle, which could save an operator from having to relocalise the vehicle within the map (Figure 22). While this system was tested and worked as intended while stationary, there was never a scenario where the localisation system crashed during an autonomous drive. Therefore, the benefits it could provide weren't realised in typical operation of the service.

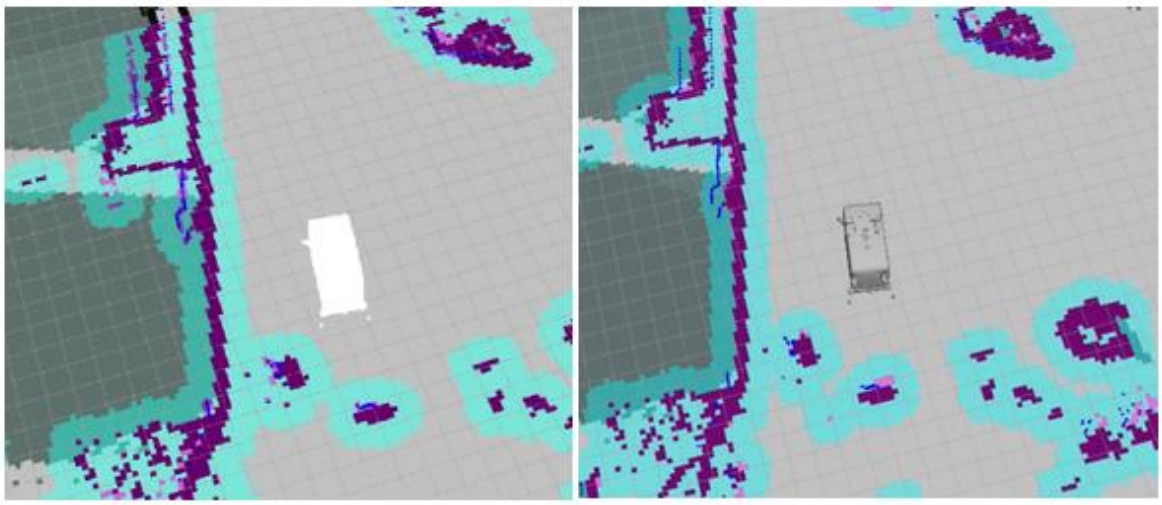

Figure 22: A vehicle's previous position is recovered (right), indicated by a loaded vehicle model, after a failure of the localisation system (left).

4.2.2 LIMITATIONS

Several limitations of the SMN were observed on nUWAy, that were not experienced in simulation environments. There was an issue where sometimes, after a ROS2 node crashed and was recovered by the SMN, the node would no longer appear on the ROS2 node list. Therefore, the SMN would declare the node as failed and continuously restart it, even though it was functioning correctly. Restarting the ROS2 daemon did not resolve this issue. A PC restart seemed to be the only way to resolve this issue. Interestingly, and possibly related, was that sometimes when a ROS2 process would crash, some of its child processes would not stop. This also led to unexpected behaviour, such as unresponsive nodes, even once they had restarted. This could be resolved through stopping all ROS2 processes on the PC.

4.3 Overall Reliability Measurements

Clear improvements were made to the reliability of the localisation stack through utilising AMCL instead of ST, shown in Figure 23. MTTF increased from 8.2 minutes to 55.9 minutes. This means it is significantly more likely that the localisation stack will not experience failure during operation. Furthermore, the SMN allowed significant reliability improvements to be made to launch, increasing MTTF from 31.8 minutes to 1230 minutes. Note that driving and low-level systems were not targeted, so any perceived improvements could be from not recording as many failures in testing or from modifications that other students working concurrently on nUWAy made. With a far less problematic launch and localisation system, opportunity exists to run more regular autonomous drives, which allows data collection and tuning to improve the driving reliability.

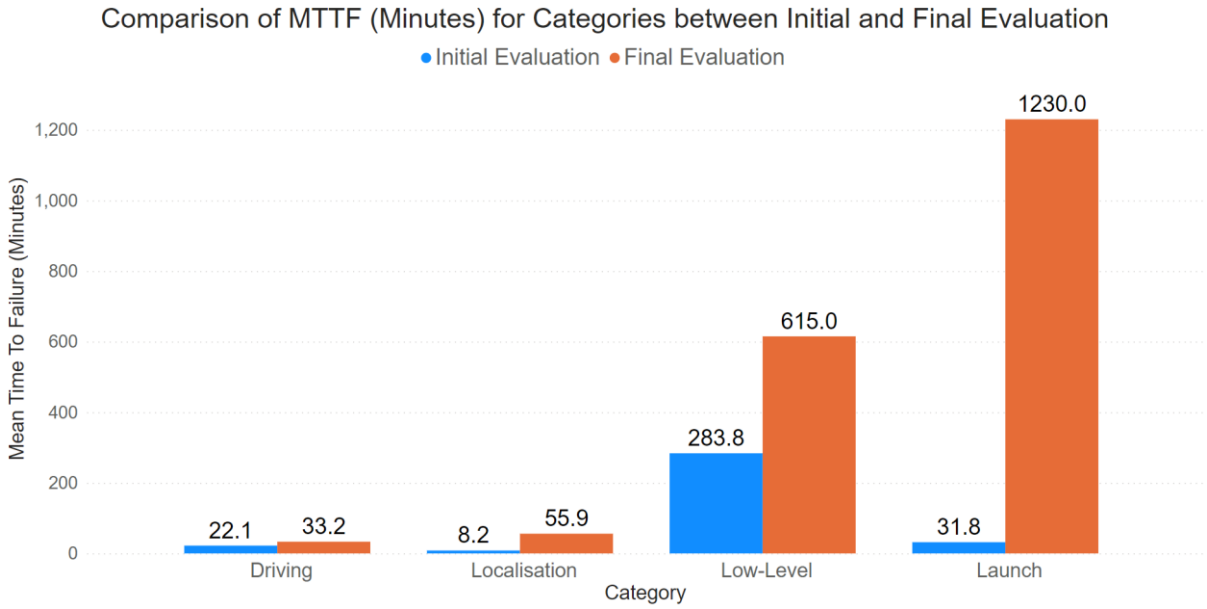

Figure 23: MTTF for the initial and final reliability evaluation shows the increases, particularly in localization and launch which were both targeted.

Medium severity failures, most common within the initial reliability evaluation, have been significantly reduced by reliability improvement tasks, as demonstrated in Figure 24. MTTF increased for medium severity failures from 6.5 minutes to 123 minutes. Therefore, crashes of the system were far less frequent. Furthermore, not illustrated in the figure but important to note, the SMN would restart and recover crashed nodes in most cases. Therefore, the system has become far more useable for operators, as it is far less likely to fail in a significant manner. Note that low and high severity failures were not targeted by the reliability improvement tasks performed in this paper, so the increase in high severity MTTF is due to less failures recorded

in the period of operation. Only 2 high severity failures were recorded in the final reliability evaluation over 20.5 hours.

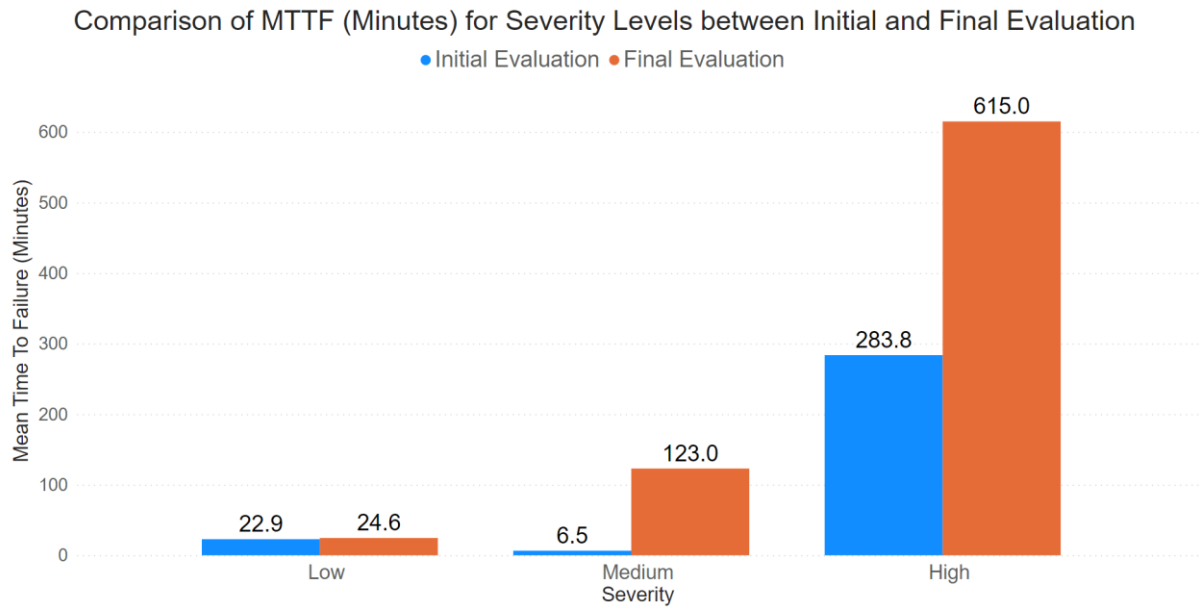

Figure 24: MTTF for severity levels between the initial and final evaluation show significant decreases in targeted medium severity failures..

5 Conclusion and Future Work

This work aimed to evaluate the reliability of the nUWAy autonomous shuttle bus, and to improve reliability of the autonomous system. This is to make the system more usable by non-technical operators, who will be the primary operators of the service on the UWA campus. Using BBR testing, the reliability of the system was evaluated and used to focus on localisation and launch reliability.

AMCL replaced ST for localisation, which saw significant improvements to the reliability of localisation, MTTF increased from 8.2 minutes to 55.9 minutes. A SMN was developed, utilising failure detection, isolation, and recovery techniques to recover from medium severity failures experienced in the system. The implementation of SMN greatly increased the reliability of launch, with MTTF increasing from 31.8 minutes to 1230 minutes. Furthermore, the system was able to detect when ROS2 nodes had crashed and take appropriate recovery behaviour to restart and checkpoint the ROS2 node where applicable. This was packaged into a single desktop icon for maximum ease-of-use by non-technical operators of the system.

Overall, medium severity failures were greatly reduced, with MTTF rising from 6.5 minutes to 123 minutes. This meant the system would experience far less crashing, allowing for greater focus to lie in improving driving systems rather than debugging frequent crashes. While this

work is a good start to improving reliability of the system, it is strongly recommended that future students continue to focus on improving reliability over time. The SMN provides a good basis to improve and expand upon for monitoring the system.

5.1 FUTURE WORK

Suggested based on the limitations of the SMN discussed in section 4.2.2, would be to implement process-level monitoring (PLM), rather than ROS2 topic and node monitoring. It is clear from implementation that there are cases where the ROS2 node list is not representative of the state of the system, causing false failure flagging by the SMN. PLM would also greatly improve the time to detect an error, which is currently slow (magnitude of several seconds to a minute). Either PLM, or SW Watchdog, presented in section 3.4.1.2, could be considered for this. A good investigation would be into mean time to repair (MTTR) of SMN, to measure the time taken to recover the system. This was overlooked in this paper due to time constraints but would allow mean time between failure (MTBF) to be used as a more representative metric of reliability for this system.

Furthermore, multi-version software fault tolerance (discussed in section 2.1.3.2) may be useful for nUWAy. For example, it could be trialled by running multiple driving algorithms in parallel. If the primary driving algorithm were to fail, failover could move to a simpler backup driving algorithm while recovering the primary driving algorithm. This would allow seamless continuation of driving, rather than the vehicle stopping when some failure is encountered.

Finally, upgrading the system from ROS2 Foxy to ROS2 Galactic is recommended for the stability improvements it brings to its core packages [38], in use on nUWAy.

6 REFERENCES

[1] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, J3016_202104, S. International, April 2021. [Online]. Available: https://www.sae.org/standards/content/j3016_202104/

[2] J. Reid, "Students first to build 'brains' of autonomous bus," 18 Jun 2021. [Online]. Available: https://www.uwa.edu.au/news/article/2021/june/uwa-students-first-in-australia-to-build-brains-of-autonomous-bus

[3] T. Braunl. "The REV Project." http://revproject.com (accessed 8 September, 2021).

[4] "IEEE Recommended Practice on Software Reliability," *IEEE Std 1633-2016 (Revision of IEEE Std 1633-2008)*, pp. 1-261, 2017, doi: 10.1109/IEEESTD.2017.7827907.

[5] R. L. Michael, *Handbook of software reliability engineering*. McGraw-Hill, Inc., 1996.

[6] S. J. Keene, "Comparing Hardware and Software Reliability," *Reliability Review*, vol. 14, 4, pp. 5-21, December 1994 1994.

[7] I. Eusgeld, F. Fraikin, M. Rohr, F. Salfner, and U. Schiffel, *Software Reliability*. 2005, pp. 104-125.

[8] C. Inacio. "Software Fault Tolerance." Carneigie Mellon University. https://users.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/index.html (accessed Apr. 21, 2022).

[9] R. J. Abbott, "Resourceful systems for fault tolerance, reliability, and safety," *ACM Comput. Surv.*, vol. 22, no. 1, pp. 35–68, 1990, doi: 10.1145/78949.78951.

[10] W. Torres-pomales, "Software Fault Tolerance: A Tutorial," 12/21 2000.

[11] P. A. Lee, T. Anderson, J. C. Laprie, A. Avizienis, and H. Kopetz, *Fault Tolerance: Principles and Practice*. Springer-Verlag, 1990.

[12] Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers*, vol. C-31, no. 6, pp. 531-540, 1982, doi: 10.1109/TC.1982.1676035.

[13] *Fault-tolerant computer system design*. Prentice-Hall, Inc., 1996.

[14] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220-232, 1975, doi: 10.1109/TSE.1975.6312842.

[15] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," in *Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[16] B. Randell and J. xu, "The Evolution of the Recovery Block Concept," 1995, p. 1.

[17] B. Johnson, "An introduction to the design and analysis of fault-tolerant systems," pp. 1-87, 02/01 1996.

[18] C. Liming and A. Avizienis, "N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELIABILITY OF SOFTWARE OPERATION," in *Twenty-Fifth*

International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'. 27-30 June 1995 1995, p. 113, doi: 10.1109/FTCSH.1995.532621.

[19] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, pp. 96–109, 1986, doi: 10.1109/tse.1986.6312924.

[20] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun, "Definition and Analysis of Hardware and Software-Fault-Tolerant Architectures," *Computer*, vol. 23, pp. 39-51, 08/01 1990, doi: 10.1109/2.56851.

[21] T. Reinhart, C. Boettcher, and S. Tomashefsky, "Self-checking software: improving the quality of mission-critical systems," in *Gateway to the New Millennium. 18th Digital Avionics Systems Conference. Proceedings (Cat. No.99CH37033)*, 24-29 Oct. 1999 1999, vol. 1, pp. 2.D.4-2.D.4, doi: 10.1109/DASC.1999.863702.

[22] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall, "The use of self checks and voting in software error detection: an empirical study," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 432-443, 1990, doi: 10.1109/32.54295.

[23] C. Burbridge. "watchdog_node." Strands Project. https://strands.readthedocs.io/en/latest/strands_apps/watchdog_node.html (accessed 4 June, 2022).

[24] P. Kaveti and H. Singh, "ROS Rescue: Fault Tolerance System for Robot Operating System," in *Robot Operating System (ROS): The Complete Reference (Volume 5)*, A. Koubaa Ed. Cham: Springer International Publishing, 2021, pp. 381-397.

[25] S. Marok, "Flexible Fault Tolerance for the Robot Operating System," Master of Science in Electrical Engineering, Faculty of California Polytechnic State University, Cal Poly, 2020. [Online]. Available: https://digitalcommons.calpoly.edu/theses/2127/

[26] M. Lauer, M. Amy, J. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 7-9 Jan. 2016 2016, pp. 94-101, doi: 10.1109/HASE.2016.30.

[27] Y. Liu, Y. Guan, X. Li, R. Wang, and J. Zhang, "Formal Analysis and Verification of DDS in ROS2," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 15-18 Oct. 2018 2018, pp. 1-5, doi: 10.1109/MEMCOD.2018.8556970.

[28] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The Marathon 2: A Navigation System," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 24 Oct.-24 Jan. 2021 2020, pp. 2718-2725, doi: 10.1109/IROS45743.2020.9341207.

[29] W. Woodall. "ROS 2 Launch System." https://design.ros2.org/articles/roslaunch.html (accessed.

[30] "Apex OS." Apex.AI. https://www.apex.ai/apex-os (accessed June 4, 2022).

[31] S. Glaser. "bond." http://wiki.ros.org/bond (accessed April 5, 2022).

[32] P. Robbel. "SW Watchdog." ROS Safety. https://github.com/ros-safety/software_watchdogs (accessed 5 April, 2022).

[33] S. Macenski and I. Jambrecic, "SLAM Toolbox: SLAM for the dynamic world," *Journal of Open Source Software*, vol. 6, p. 2783, 05/13 2021, doi: 10.21105/joss.02783.

[34] S. Macenski. "Slam Toolbox." https://github.com/SteveMacenski/slam_toolbox (accessed June 1, 2022).

[35] B. P. Gerkey. "AMCL." http://wiki.ros.org/amcl (accessed June 1, 2022).

[36] L. Haddad. "nuway_ros2_monitor." UWA REV. https://github.com/uwa-rev/nuway_ros2_monitor (accessed June 2, 2022).

[37] T. Narlock. "libtmux." https://libtmux.git-pull.com/ (accessed June 2, 2022).

[38] "ROS 2 Galactic Geochelone." ROS.org. https://docs.ros.org/en/foxy/Releases/Release-Galactic-Geochelone.html (accessed 3 June, 2022).

7 APPENDICES

7.1 APPENDIX A: FAILURE LOG DATA

Table 2: An extract from the initial reliability examination, demonstrating descriptions of failures encountered, as well as their category and severity. This includes data from the first 2 days of testing.

Failure	Description	Severity	Specific Category	Date:	7-Feb	9-Feb
				Time Started:	**11:00**	**10:15**
				Pack Up Time:	**16:30**	**16:00**
Hurcules Related						
Major Failure	Screen becomes garbled/no display, unable to rearm or drive, power cycle of board required	High	Low-Level	Failure	0	1
				Success	N/A	
Software Driver Related						
Safety lidar drivers don't load	Not rearmable, no safety lidars on startup, bus pc stack restart fixes	Medium	Launch	Failure	5	4
				Success	0	0
Bus positioning issue	Bus continues driving infinitely on map when stopped, crashing everything.	High	Low-Level	Failure	0	1
				Success	N/A	

	Entire bus shutdown required to fix					
Slam Toolbox Related						
Map load failure (Process dies or Rviz crashes)	Slam toolbox process dies while deserializing a map, or rviz window crashes and cannot be reopened. Full stack restart required	Medium	Localisation	Failure	17	32
				Success	5	11
Unable to stay localized	Bus drifts off map and does not correct back onto it. New pose estimate required.	Low	Localisation	Failure	2	0
				Success	N/A	
Pose estimate causes stack to crash	Multiple pose estimates in a row (and sometimes a single one) cause the entire stack to crash	Medium	Localisation	Failure	6	9
				Success	18	10
Nav Stack Related						
Nav stack does not start properly	Missing parts of the stack such as local costmap or global/local path	Medium	Launch	Failure	0	1
				Success	23	43

	planner. Full stack restart required					
Local planner is unable to follow a path	Bus drives to an undesireable location as a result of poor path following, or poor local path planning	Low	Driving	Failure	1	10
				Success	4	13
Local planner fails to plan a path	Local planner fails to plan a path as a result of dynamic obstacles and does not attempt recovery (vehicle/path constraints, pedestrians, objects blocking path)	Low	Driving	Failure	2	13
				Success	2	0
Performance Related						
Severe lag in stack (Caused by Slam Toolbox)	Loading a large map results in large frame skips and unstable performance, bad path following and localization	Medium	Localisation	Failure	-	-
				Success		
Transform errors (Nav Stack)	Path plan cannot be completed due to transform errors (extrapolation into the past/future)	Medium	Driving	Failure	-	-
				Success		