Proceedings of Australasian Conference on Robotics and Automation, 3-5 Dec 2012, Victoria University of Wellington, New Zealand.

A Novel Approach to Automated Systems Engineering on a Multi-Agent Robotics Platform using Enterprise Configuration Testing Software

Stephen Cossell
Engineer, ScriptRock Inc.
steve@scriptrock.com

Abstract

This paper presents a case study of applying an enterprise grade systems configuration management platform to a set of unmanned ground vehicles and a ground control station. Much like large scale enterprise infrastructure, modern robotics systems are comprised of many different machines communicating over a variety of media, let alone the large number of modules and applications running on each machine. Each module typically has its own configuration settings, with each individual piece of configuration information being crucial to the overall working state of the robotic system. When one configuration item is changed inadvertently, or otherwise, without an operator's knowledge, a manual and lengthy *expedition* through a series of configuration files and command output is usually used to diagnose the cause of the problem. This situation is exacerbated when the platform is used by a range of people for different scenarios on a regular basis. The ScriptRock platform is used by large enterprise software and infrastructure teams to encode system configuration requirements into executable documentation so the underlying environment their applications run on can be validated immediately. The application of the ScriptRock platform to a multi-agent robotic system has shown improved re-configuration times between different use cases as well as a significantly simplified troubleshooting and diagnosis process when the system is found not to be in a working state.

1 Introduction

As more robotic systems are entering mainstream operation they are becoming more complex in order to be able to complete more intricate tasks. In addition, these systems are being given more responsibility in society and as such must be robust and able to function without fault. All modern industrial robotics projects are more than likely to have extensive systems engineering processes tied into their development life-cycles. This process typically involves a thorough set of unit and integration tests to validate each software component of a system, from a functional point of view. It is however difficult to find evidence in published literature of development and testing practices that validate the underlying environment[1] and configuration of software modules themselves.

Software configuration management has been a point of interest for computer science researchers over the last three decades. In the 1980s, research by [Bersoff, 1984] concentrated on identifying and understanding the problem and promoted the need for configuration management in large enterprise systems. By the 1990s, researchers had begun proposing solutions to particular niche problem areas. Work by [Hall et al., 1997] presented a possible solution to configuration management of wireless local area networks. As part of his job maintaining university workstations, Mark Burgess developed CFEngine [Burgess, 1995], a cross platform package for automated system configuration management. While CFEngine provides a useful tool for configuration management, it's potential usage is limited to developers, as configuration *promises* must be encoded in scripts. This is in contrast to the ScriptRock platform, which provides an interface that abstracts underlying code required to validate in individual configuration item [Sharp-Paul, 2012].

In the last decade companies with large dynamic cloud infrastructures such as Netflix [Hoff, 2010], have

[1]This paper uses the term *environment* in a software systems context rather than to describe the physical space an agent interacts with. This includes the underlying operating system, network infrastructure and supporting applications and libraries an application requires to function.

applied custom in-house solutions to test and correct configuration state of their own systems and infrastructure via deliberate and regular breaking of configuration items. This implementation is in line with IBM's manifesto release in [Horn, 2001] requesting a push for the widespread adoption of *Autonomic Computing*.

In the last six years there has been a push in robotics research circles to adopt a common platform for software modules. Platforms such as *aRD* [Hirzinger and Bauml, 2006], *OpenRDK* [Quigley et al., 2009] and the *Robot Operating System* (ROS) [Calisi et al., 2008] attempt to find this common standard, with ROS gaining much adoption in the global robotics community. While these platforms allow rapid prototyping and application of common robotics algorithms, they are still difficult to configure correctly and ensure critical dependencies are satisfied. Although this paper focusses on applying the ScriptRock platform to a custom, in-house robotic system developed within the mechatronics research group at UNSW, future work will attempt to apply a ScriptRock test suite to common ROS modules. The aim is to be able to validate the installation and inclusion of a ROS module by running a ScriptRock test that is distributed with the module.

This paper uses a multi-node system first presented in [Whitty et al., 2010] and later from a more software and networking perspective in [Guivant et al., 2012]. Although these papers describe multi-vehicle systems, a simplified single unmanned ground vehicle (UGV) setup is used in this paper in various common use case scenarios. As presented in Section 3, these scenarios range from direct on board tele-operation to semi-autonomous operation, used commonly for research purposes.

1.1 Outline

The application presented in this paper demonstrates a simple and efficient method for defining and managing system configuration information as well as enabling fast troubleshooting when a configuration item, among hundreds, has been misconfigured. Section 2 begins by giving a high level background on both the UGV and ground control station (GCS) system as well as the high level capabilities of the ScriptRock platform. Section 3 outlines how the ScriptRock platform has been applied to the UGV and GCS system. Sections 4 and 5 respectively discuss the details and results of an experiment used to gauge the increased benefit of using the ScriptRock platform on the UGV and GCS system over an existing manual method.

2 Platform Background

This section gives a high level overview of both the UGV/GCS and ScriptRock platforms in terms of software and network system configuration.

2.1 The Unmanned Ground Vehicle and Ground Control Station Platform

This section gives a high level software module and network layout outline for the Unmanned Ground Vehicle (UGV) and Ground Control Station (GCS) setup discussed in this paper. For a more detailed description of the platform as a whole, see [Whitty et al., 2010] and [Guivant et al., 2012]. For a description of specific individual software components, please refer to [Robledo et al., 2010], [Guivant, 2008] and [Guivant, 2012].

2.2 The ScriptRock Platform

At a high level the ScriptRock platform allows a person to submit system and configuration requirements into an online cloud based platform. Requirement information is submitted using templates that abstract away the technical knowledge required to actually perform a requirement validation. For example, a requirement might be to ensure that a particular program is located in the correct location on a target machine so that the automated start-up scripts can find it properly. Although there are commands that can be run on any mainstream operating system (OS) to check that a particular file or application exists in a given folder, each command is different on each OS and must be manually encoded into a test script. The respective ScriptRock template used for the this type of validation only requires a full path to be entered with a human readable description, with the actual testing process itself abstracted away.

Once a set of requirements have been submitted to the website they are then downloaded as a zip file containing all the requirements translated into executable tests. The zip file contains a shell or batch script that runs the complete set of tests, which generates either plain text or an HTML report containing the results of each test run. If a test fails, a remediation step or set of steps is presented to the tester so they can correct the state of the system and therefore subsequently satisfy the test requirement. As software modules get updated, the online tests validating these modules can be modified and re-downloaded to reflect the change in configuration state. Since the UGVs and GCS are used in different scenarios on a regular basis their underlying configurations change often. A separate ScriptRock test project was set up for each of these scenarios, so that an particular test script can be used to validate the state of a particular desired scenario.

3 Method

The UGV and GCS system is flexible enough to be configured in a number of different ways, but there are three main scenarios that are used most often and are usually required at short notice. These scenarios are direct tele-operation, remote tele-operation and remote point-and-click semi-autonomous operation. Each scenario is outlined in the following sections, along with the modules and infrastructure required and an overview of the types of ScriptRock tests that have been applied to track configuration.

3.1 All Scenarios

Although the modules and devices used in each of the following scenarios differ, there are a number of common configuration items that apply to all scenarios. Each scenario relies on a set of dynamic link libraries (DLLs) as well as certain software interpreters to be installed and be accessible in the operating system's search path. To check a specific DLL is installed and located in a known location, a "`file_exists`" test was created within ScriptRock. Modules are started on each node using a Python script. As a result, a test was created to check that Python was installed on the system and that the correct version of Python was installed.

Even with the simplest scenario of direct tele-operation, modules are required to communicate data over a network connection. Direct tele-operation requires two ethernet connections on board whereas the other scenarios require multiple hops via wired and wireless connections and over different subnets. As a result, a set of "`ping_success`" tests were used to check that the require nodes could communicate with one another. The underlying network configurations of these nodes were also tested by using a combination of "`command_output`" and "`file_contents`" checks. For example, the routing table can be checked on Windows by running the command "`route print`" and checking the output for an expected value. The same can be done on Unix based platforms using the "`route -n`" command.

In the remote tele-operation and point-and-click autonomy scenarios, certain specific settings were required at boot on each of the wireless routers to make sure the UGV's mesh network settings took precedence over the system defaults. A custom script was placed in the Linux start-up directory "`/etc/init.d`" to apply these settings. In addition to checking that the required settings were actually set using custom "`command_output`" tests running commands like "`ifconfig`", an extensive set of "`file_contents`" tests were also used to validate that the custom initialisation script itself contained correct values.

The third major category of tests applicable to all scenarios are the requirements that folder structure, configuration files for custom modules, as well as the modules themselves, existed and were located in the correct locations. A number of "`directory_exists`" and "`file_exists`" tests were used to validate the complete folder structure was constructed correctly.

It should be noted that although this paper focuses on validating that a system is in the correct state as it changes between use cases, these tests have also greatly assisted in rolling out software and network settings to a fresh on board laptop. By running these tests on a regular basis as modules were added, it was immediately apparent that a module or setting was missing without needing to actually start the software up and then manually detect the problem or missing component.

In each of the following sections an infrastructure diagram is used to assist in describing each scenario. Figure 1 outlines each of the diagram components.

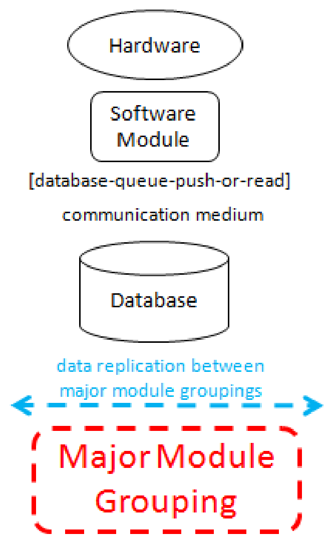
Hardware
Software Module
[database-queue-push-or-read]
communication medium
Database
data replication between major module groupings
Major Module Grouping

Figure 1: Legend used for infrastructure diagrams.

3.2 Direct Tele-operation

Direct tele-operation involved an operator using an Xbox 360 controller to drive the UGV for situations such as Open Day demonstrations, where direct deterministic control is required for OH&S purposes, as well as simply moving the robot from one lab to another. The scenario requires only infrastructure on a single UGV and involves a minimal set of configuration checks to be created, as outlined below.

Infrastructure Layout

The infrastructure used in this scenario is contained within a single UGV. Figure 2 shows how data flows between hardware and software modules in this scenario.

Tele-operated commands are generated by an Xbox 360 controller, which are received by a software module on the UGV's on board laptop. These commands are pushed by the module into the centralised database system.

Laser range finder measurements from the front and back proximity lasers are also pushed into their respective queues in the database. A laser proximity module then reads these queues and determines if an object is too close to allow motion in a particular direction. This module pushes data to a queue in the database on a regular basis, but modifies a specific flag in each pushed record as to whether it can, or cannot detect anything in its proximate range.

The command arbiter module reads new tele-operation commands and laser proximity records from this centralised database on a regular basis and allows the instructions to pass through to the DMC interfacing module if the proximity queues in the database have not flagged any objects as being in the immediate vicinity of the UGV. The DMC module then takes the final filtered driving commands from the database and communicates them to the on board DMC via the local ethernet connection.

Configuration Items

In addition to the general configuration items applicable to all scenarios, a number of "`file_contents`" based checks were created for critical configuration files of modules required to achieve tele-operation. The most important checks for this scenario center around sensor interfacing modules. Each of the three laser range finders attached to the UGV communicates data either via a wired ethernet connection or via a RS-422 serial connection converted to USB. Each of these modules encodes the IP address and port number, or the baud rate and COM port in a configuration file. Ensuring that these values are not only present, but also set correctly, is an important step in validating whether the UGV is configured correctly.

3.3 Remote Tele-operation

Remote tele-operation is used predominantly for teaching purposes during the later stages of undergraduate robot design and autonomous systems courses. In addition to the on board modules used in the *Direct Tele-operation* scenario outlined in Section 3.2, this scenario makes use of a number of machines that replicate data over both a wireless and wired network connection. A detailed overview of the infrastructure is given below, followed by the additional configuration items required for this scenario to run.

Infrastructure Layout

In addition to the modules used on board for laser proximity checks and arbitrating driving instructions to the DMC interfacing module, this scenario relies on a number of network and centralised database replication settings, which are covered in great detail in [Guivant et al., 2012]. This infrastructure is shown in Figure 3.

Data from the UGV is replicated to a teacher's desktop computer over a wireless network connection, using a pair of mesh network routers. The environment the UGV operates in also makes use of a fixed external laser range finder, which acts as a *local positioning system*, much like a GPS unit in outdoor environments. This laser data is replicated over an ethernet connection to the teacher's desktop computer.

Student machines are connected to the teacher's machine via a local subnet over ethernet. Students run a local copy of the centralised database, which listens for broadcasted data from the teacher's machine. In turn the student runs custom written software to read data from their local database, interpret the data and push driving instructions back into the local database. This local database is then configured to replicate driving instructions back to the teacher's machine, which are then in turn replicated back to the UGV's database. Once on the UGV, these driving instructions are handled via the same tele-operation mechanism as in Section 3.2.

Configuration Items

On board, additional "`ping_success`" tests were added for each additional node that data needs to be replicated to, to ensure the underlying medium is configured correctly. In fact, a test suite of network configuration checks was created to test the connectivity of each node to every other node it is required to communicate with. The database replication functionality is also based on an internal code, called an *icode*, that enables data to be received into a correctly configured queue on a destination node. A set of "`file_contents`" checks

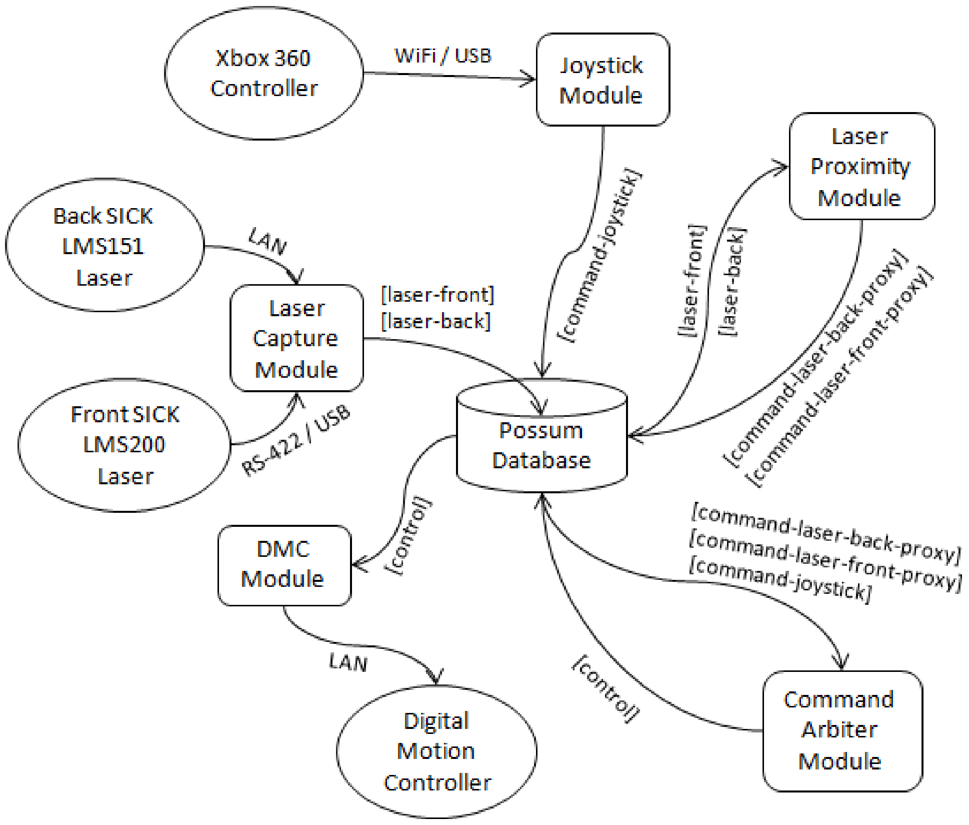

Figure 2: The software and networking component layout used for direct tele-operation.

were created for each replicated queue on each node to ensure the numeric *icodes* were consistent. Custom database queue definition checks were also included for each node to ensure the internal data structures in the database were configured correctly.

Having a test suite designed for the UGV and teaching workstation greatly reduced the set up time required to prepare the machines for lab work. In addition, a *student machine test suite* allowed students to ensure their module set was installed correctly without having to take up a lab demonstrator's time. The *student machine test suite* was also used by lab demonstrators to assist in troubleshooting exotic problems that arose on students' machines brought from outside the lab.

3.4 Semi-autonomous Point-and-Click

The semi-autonomous point-and-click scenario is the most complex scenario outlined in this paper. It is predominantly used for research purposes and involves a number of modules spread over one or more UGVs and a GCS machine. A typical use case involves fusing sensor data and displaying a virtualised reality on the GCS machine. At a high level, an operator can click a ground location on the virtualised reality interface and immediately see a given UGV travel autonomously to the selected destination in reality. The point-and-click functionality can be interchanged with other higher level planner or voice controlled modules that issue desired UGV way points and destinations in the same manner. However, simple virtualised interface point-and-click is used here for simplicity.

Infrastructure Layout

This scenario uses the same configuration on the UGV and wireless routers as the Remote Tele-operation scenario discussed in Section 3.3. The main difference lies in the GCS machine used to remotely control the UGV and the missing laser range finder previously used as a *local positioning system.* Figure 4 shows a high level representation of data flow and modules used in this scenario.

The GCS database receives replicated data that includes raw laser range values from the top mounted rotating laser on the UGV, a pose estimation and DMC wheel rotation and steering positions. This data is then read by both a fusion and laser conversion

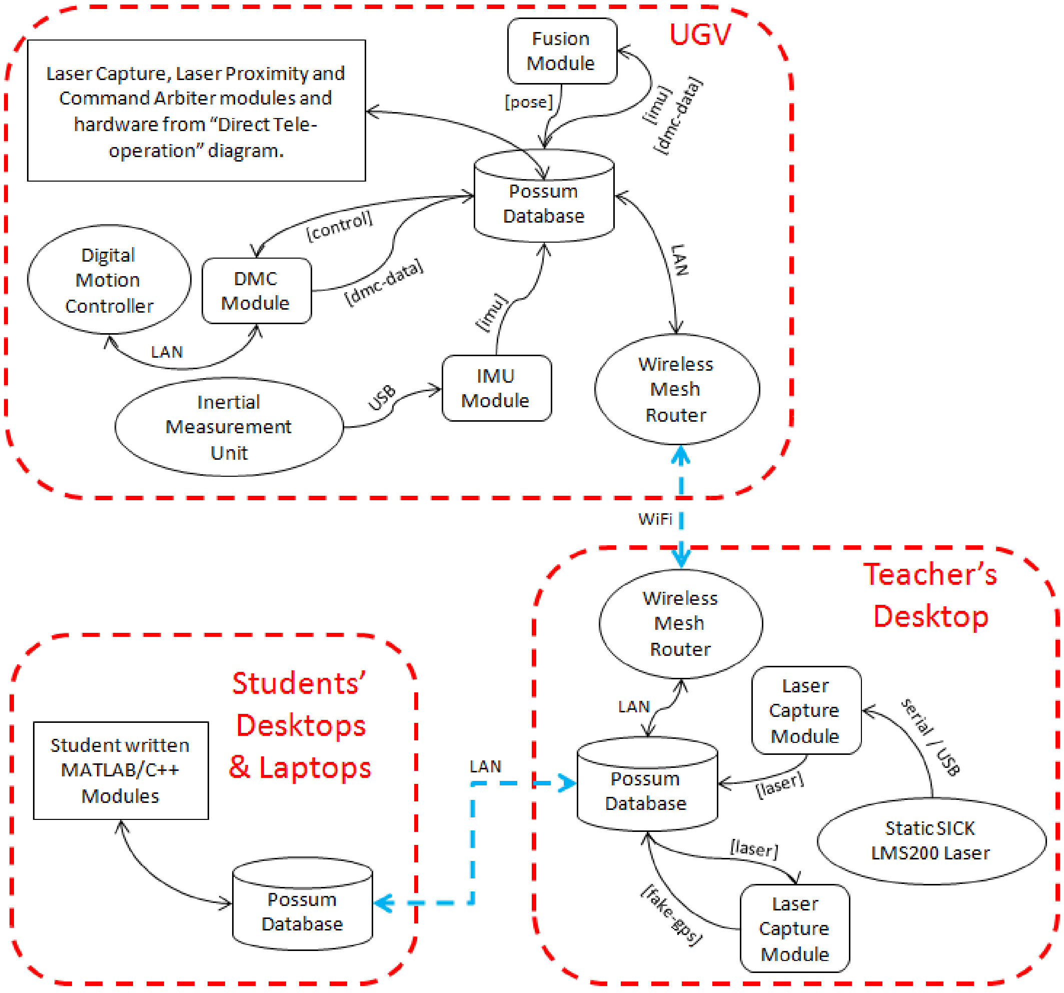

Figure 3: The software and networking component layout used for remote tele-operation.

module to produce a three-dimensional point cloud and occupancy grid, which are pushed back into the GCS database. This data is then read by an OpenGL based visualisation module to display the spatial data in real-time. This interface allows an operator to click on a location on the occupancy grid, which pushes a desired destination into another queue in the database.

This desired destination is in turn read by a planner module, which uses the UGV's current pose estimation, along with the current occupancy grid, to plan a path to that location. This path is pushed to another database queue on the GCS machine and replicated to the UGV. A module on the UGV reads this path and translates it into driving instructions. These instructions are handled in an identical manner to driving instructions in other scenarios and processed through the arbitration module mentioned in Section 3.2.

Configuration Items

Each module requires the correct queue configurations set so that data passes from one module to the next correctly. Each database queue must also be replicated between database instances correctly and as such a range of configuration items are required to be validated via a set of tests.

One example of this involves the point cloud generation module having the correct laser input queue set in its configuration. Since the UGV has three lasers on board, the correct laser queue for point cloud generation must be set, or an incorrect point cloud is generated. This can have a lead on effect to occupancy grid generation and ultimately path planning through an environment that does not match reality. A separate "`file_contents`" test was created for each database input and output queue entry on each module.

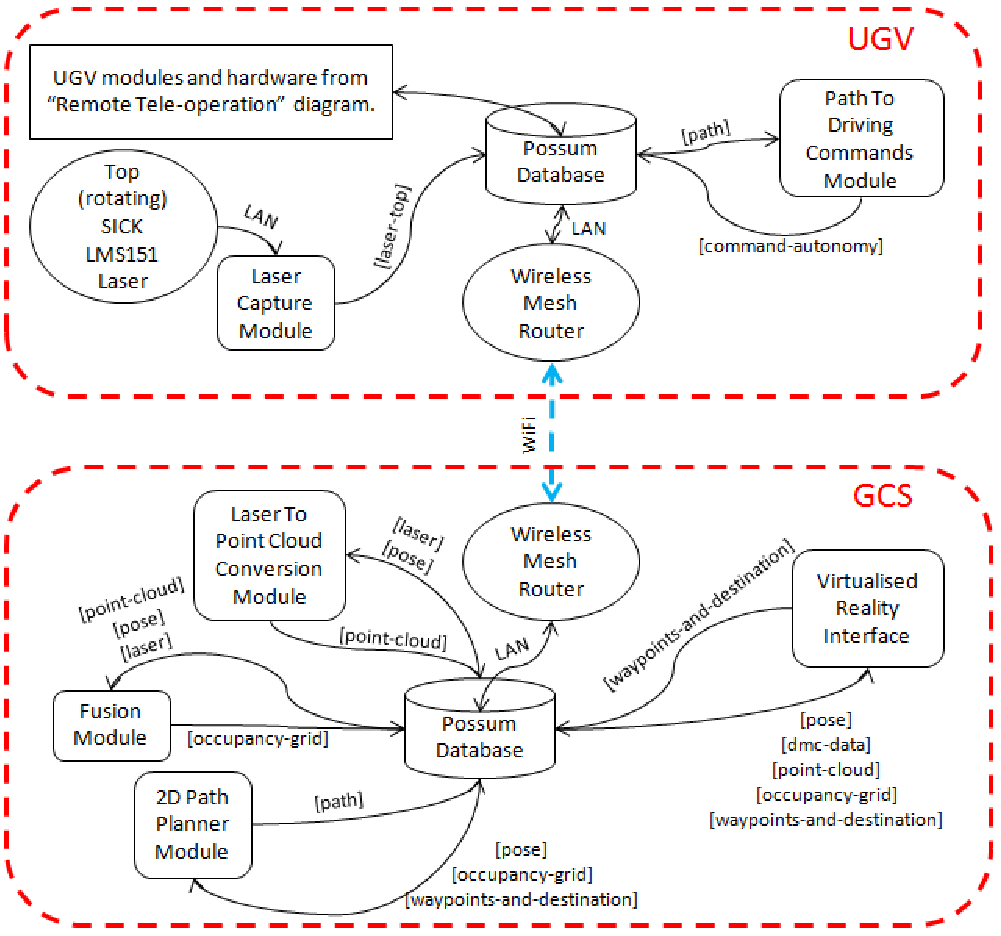

Figure 4: The software and networking component layout used for point-and-click autonomy.

Another example of an insidious misconfiguration is pushing tele-operated driving instructions, autonomously generated driving instructions or laser proximity effecting driving instructions to the wrong queue. On board, the command arbiter module uses inbuilt flags and a hierarchy system to give precedence to certain driving instructions over others. This module relies on the assumption that a given database queue has been configured in another module to push records from the correct location. For example, an emergency mode of tele-operation, that ignores laser proximity data, has the highest precedence in the command arbiter program, but is rarely used in practice. If, for example, the autonomous driving instruction generation module was misconfigured to push data to this queue, then all safety protocols will be unintentionally ignored when the UGV is driving in autonomous mode. Again, a series of "`file_contents`" checks were used to enforce correct queue assignments to all modules.

A final example involves mathematical constants stored in the *planned path to driving instructions* configuration file. This module takes a path and current pose and uses control theory to continually issue driving instructions. These calculations rely on correctly configured control constants as well as knowing the UGV's maximum forward and backward speeds and steering angle limits. A misconfiguration to one or more of these values can lead to erratic motion, which can damage the UGV, let alone objects or people in its operating environment. A number of tests were also included to validate each of these constants individually, with certain domain knowledge encoded into the tests relating to valid ranges of these values for proper operation.

4 Experimental Outline

To test the added troubleshooting benefit of using the ScriptRock platform over an otherwise manual process, an experiment consisting of a series of tests was devised that cover the three main scenarios outlined in Section

3. For each scenario, a set of critical configuration items were selected at random and encoded into a *chaos* script as separate tests. When run, the *chaos* script randomly chooses one configuration item and modifies it into a broken state. This is based on a similar concept used by the Netflix Chaos Monkey [Hoff, 2010].

Each misconfiguration item encoded into the *chaos* script falls into one of the following eight categories:

- a missing DLL file or missing software dependency such as the Python runtime being made inaccessible;
- a missing executable file of a software module;
- a missing configuration file for a given software module;
- an incorrectly set database queue in a module's configuration file for either the queue it reads data from or pushes data to;
- an *icode* misalignment between the two database instances replicating data between one another for a particular database queue;
- an incorrect IP address or port number for database replication settings;
- an underlying network misconfiguration such as incorrect subnet settings or incorrect IP address assignment; or
- any module specific configuration file lines such as incorrect baud rates, invalid IP address or COM port for hardware modules and particular constant configuration definitions required for module calculations.

A total of 111 misconfigurations were spread across three scenario *chaos* scripts. Before each test, a scenario is chosen at random and the respective *chaos* script was run on a given node used in that scenario. A *working state action* was then attempted upon the system unsuccessfully to prove that the *chaos* script did in fact break a crucial configuration item. An experienced engineer that was involved in the design, development and now maintenance of the entire software and networking component set was given access to the system to attempt to diagnose and correct the introduced problem and therefore complete the given *working state action*. Use of every existing piece of software, testing tool and piece of documentation was allowed in every scenario attempt, but the engineer was only allowed access to a ScriptRock test suite for the given scenario on every second attempt. A scenario attempt is deemed complete if the system is able to carry out the respective *working state action*. These are defined as:

Direct Tele-operation Use an Xbox 360 controller to tele-operate the UGV directly over a distance of three metres and return to the starting position, completely demonstrating that it can also steer both left and right.

Remote Tele-operation Be able to use a keyboard based application[2] on the GCS machine to tele-operate the UGV remotely over a distance of three metres, completely demonstrating that it can steer both left and right while driving in both directions.

Semi-autonomous Point-and-Click Given an open area that is traversable by the UGV, issue a UGV desired destination by clicking on a location in a virtualised reality interface on the GCS and have the UGV then autonomously travel to that destination in reality. Then issue another desired destination back to the starting location. In the process, the path chosen must demonstrate that the UGV can move forward and backwards, and be able to steer both left and right during motion.

On each scenario attempt, a timer was started when the engineer was first allowed access to the system. The timer was paused when the engineer believed they had solved the problem. A *working state action* was then attempted immediately to determine if the problem had in fact been solved. Successful completion of the action lead to the paused time being recorded as the official *solve time* of that test. On a unsuccessful action attempt, the timer was resumed from the previously paused state and the engineer was again given further access to the system until they again claimed to have solved the problem. Section 5 outlines the results of this experimental approach applied multiple times and provides an analysis of the results.

5 Experimental Results and Analysis

The experiment outlined in Section 4 was run a total of 50 times, 25 of which allowed the use of the ScriptRock platform and the other 25 without. Figure 5 shows the recorded times of each attempt. The graph clearly demonstrates that using the ScriptRock platform to diagnose system misconfiguration adds an element of determinism and repeatability to the troubleshooting process.

The main observation made from tests allowing the use of the ScriptRock platform was that the engineer would immediately initiate the ScriptRock test suite without considering the use of any other approaches. The test suites created for each scenario took 1m 25s ($\pm$ 10s) to execute and upon viewing the resulting test

[2]A keyboard tele-operation program was used to simulate a student's software module.

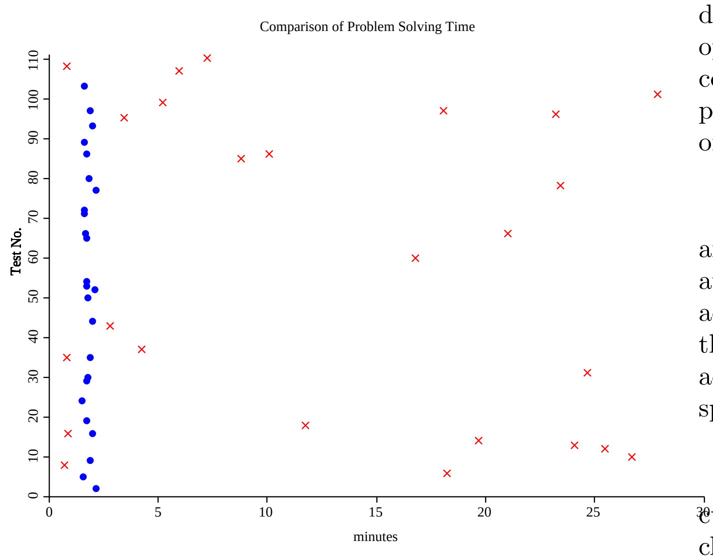

Figure 5: Comparison of time take to solve a given scenario problem. A blue circle represents an attempt where the engineer was allowed to use the ScriptRock platform, while a red cross represents attempts without access to the ScriptRock platform.

report[3], the engineer subsequently solved the problem within a further 10s to 40s.

When not able to use the ScriptRock platform, the engineer appeared to begin the diagnosis process by starting all software start-up scripts to attempt to observe which modules appeared to function normally and which did not. If everything appeared to initialise correctly, the engineer would often then attempt to drive the UGV via tele-operation or, in autonomy related tests, set a desired destination to drive to. The engineer also appeared to make extensive use of the queue activity tab of the centralised database instance on each node to see if certain critical queues were not being populated. The database queue activity and attempted motion testing methods then lead the engineer to begin looking through between one and five configuration files to check configuration items, line by line, until the problem was ultimately solved.

The four quickest test attempts in Figure 5 did not involve any use of the ScriptRock platform. These test all fell into the misconfiguration category of moving or misplacing one of the software module executable files.

[3]The default layout of a results report places the results of all failed tests at the top, followed by a complete set of all test results.

Due to the engineer beginning each non-ScriptRock diagnosis attempt by running a start-up script, the operating system presented a message box when it could not find a particular executable file to run. This provided the engineer with the ability to solve this type of problem consistently within one minute.

Direct and remote tele-operation scenario tests were among the easier misconfigurations to diagnose. By attempting to drive the UGV and by observing the activity of database queues, the engineer could solve the problem in a number of minutes. If database activity appeared to be normal, then configuration files specifically on the UGV were checked to find a solution.

Autonomy related tests were the next most difficult misconfiguration to diagnose as they involved both checking low level tele-operation modules on the UGV as well as sensor fusion and path planning modules on the GCS machine. Some diagnosis attempts were assisted by observing inconsistencies in database queue activity and inter-node database queue replication anomalies. Some tests were also assisted by the fact that the UGV could be tele-operated but not autonomously driven. This lead the engineer to focus the search predominantly on the GCS. One of the more successful and unique test attempts in this category involved the UGV never being able to steer left. This was eventually diagnosed as a constant being set to zero that represented one of two steering angle limits in the DMC interfacing module.

The most difficult group of tests to diagnose involved semi-autonomous operation, with the UGV also not responding to remote or local tele-operation attempts. Here the engineer resorted back to tracking the virtual flow of data from sensors through software modules, database replication and resulting actuator modules. More than half of these type of tests involved the engineer consulting a wiki for information about the usage and configuration of modules.

A statistical summary of the experimental results is given in Table 1, which significantly highlights the more than five-fold improvement in average troubleshooting time using ScriptRock over existing methods. It should be noted that each test attempt presented in this paper involved a single item being misconfigured. Future experiments are planned to test the troubleshooting times of multiple misconfiguration points within a scenario. The hypothesis here is that multiple misconfiguration items may linearly or exponentially make diagnosis more difficult for existing manual processes, but only constantly more difficult for ScriptRock enabled tests.

Test Type	**w/ ScriptRock**	**w/o ScriptRock**
Average Time	1m 47s	13m 16s
Std. Dev.	11.1s	9m 40.5s
Total Time	44m 38s	5h 31m 38s

Table 1: Statistical summary of experimental test times with and without the use of the ScriptRock platform, each over 25 tests.

6 Conclusion

The application of the ScriptRock testing platform to a multi-node robotics system has demonstrated an increased amount of understandability and determinism in managing the software and networking configuration. Undertaking configuration change, or troubleshooting misconfigured nodes, has been reduced to time frames well within five minutes, as opposed to fractions of an hour, using existing manual techniques. As a natural progression these tests have also improved the process of rolling out software and network settings on a new UGV. The progress and ultimate success of a fresh roll out can be measured accurately through the setup process. Making slight configuration changes for use in differing scenarios can also be validated without having to resort to the same previously used manual process. Robotic systems, much like large enterprise grade architectures, have an extensive range of solutions capable of testing whether an application performs correctly, but little is available to test the underlying environment and module configuration. This paper has demonstrated that this innovative application to enterprise systems can also be applied to complex robotic systems to greatly simplify the development, maintenance and use of these systems.

Acknowledgments

The author would like to acknowledge, Dr José Guivant, Mr Mark Whitty and A/Prof Jayantha Katupitiya for their extensive contributions to the design and development of the UGV and GCS system and providing access to the platform. Their assistance in the test creation and testing process is also appreciated.

ScriptRock should also be acknowledged for providing a custom educational licence to the engineers and researchers in Mechatronics at UNSW.

References

[Bersoff, 1984] Bersoff, E. H. (1984). Elements of software configuration management. *IEEE Transactions on Software Engieneering*, SE-10(1).

[Burgess, 1995] Burgess, M. (1995). Cfengine: a site configuration engine. *USENIX Computing Systems*, 8(3).

[Calisi et al., 2008] Calisi, D., Censi, A., Iocchi, L., and Nardi, D. (2008). Openrdk: A modular framework for robotic software development. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 1872–1877, Nice.

[Guivant, 2012] Guivant, J. (2012). Possum robot. *http://www.possumrobot.com*.

[Guivant et al., 2012] Guivant, J., Cossell, S., Whitty, M., and Katupitiya, J. (2012). Internet-based operation of autonomous robots: The role of data replication, compression, bandwidth allocation and visualization. *Journal of Field Robotics*, 29(5):793–818.

[Guivant, 2008] Guivant, J. E. (2008). Real time synthesis of 3d images based on low cost laser scanner on a moving vehicle. In *V Jornadas Argentinas de Robtica (JAR'08)*.

[Hall et al., 1997] Hall, R. S., Heimbigner, D., van der Hoek, A., and Wolf, A. L. (1997). An architecture for post-development configuration management in a wide-area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, MD.

[Hirzinger and Bauml, 2006] Hirzinger, G. and Bauml, B. (2006). Agile robot development (ard): A pragmatic approach to robotic software. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, Beijing.

[Hoff, 2010] Hoff, T. (2010). Netflix: Continually test by failing servers with chaos monkey.

[Horn, 2001] Horn, P. (2001). Autonomic computing: Ibms perspective on the state of information technology.

[Quigley et al., 2009] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). Ros: an open-source robot operating system.

[Robledo et al., 2010] Robledo, A., Guivant, J., and Cossell, S. (2010). Pseudo priority queues for real-time performance on dynamic programming processes applied to path planning. In *Proceedings of the 2010 Australasian Conference on Robotics & Automation*, Brisbane.

[Sharp-Paul, 2012] Sharp-Paul, A. (2012). Scriptrock - frequently asked questions. *https://www.scriptrock.com/faq*.

[Whitty et al., 2010] Whitty, M., Cossell, S., Dang, K. S., Guivant, J. E., and Katupitiya, J. (2010). Autonomous navigation using a Real-Time 3D point cloud. In *Proceedings of the 2010 Australasian Conference on Robotics & Automation*, Brisbane.