2021 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
AUTONOMY, ARTIFICIAL INTELLIGENCE & ROBOTICS (AAIR) TECHNICAL SESSION
AUGUST 10-12, 2021 - NOVI, MICHIGAN

Collaborative Migration of an Autonomous Ground Vehicle Software System to ROS 2

Michael Boulet[1], Ryan DelGizzi[2], Scott Lathrop[3], Brendan Leahy[4], Jake Montez[1], Gary Rucinski[1], Matthew Spinola[5], William Thomasmeyer[6], Jerry Towler[7]

[1]MIT Lincoln Laboratory, Lexington, MA; [2]Stratom, Boulder, CO;
[3]Raytheon BBN, Columbia, MD; [4]Robotic Research, Clarksburg, MD;
[5]Neya Systems, Framingham, MA;
[6]National Advanced Mobility Consortium, Ann Arbor, MI;
[7]Southwest Research Institute, San Antonio, TX

All authors contributed equally to this work.

ABSTRACT

The Robotic Technology Kernel (RTK) is a government-owned library of reusable software modules based on the first generation Robotic Operating System ("ROS 1") that can be formed into "autonomy stacks" for integration onto defense robotic platforms. RTK has been used to demonstrate autonomous ground vehicle capabilities spanning many programs and mission scenarios over the past five years. Future use of RTK is dependent, however, on migrating it to be compatible with the 2nd generation of ROS middleware ("ROS 2") scheduled to replace ROS 1 in May, 2025.

This paper summarizes the methodologies, systems, results, and lessons learned thus far from a project to migrate RTK to ROS 2 for the purpose of informing similar ongoing or future large software-centric activities within the ROS and defense robotics communities. A key conclusion is that a well-defined set of organizational practices and technical guidance can enable a large, heterogeneous team of developers from multiple industry, non-profit, and FFRDC organizations to successfully execute a complex DoD software task.

Citation: M. Boulet, R. DelGizzi, S. Lathrop, B. Leahy, J. Montez, G. Rucinski, M. Spinola, W. Thomasmeyer, J. Towler, "Collaborative Migration of an Autonomous Ground Vehicle Software System to ROS 2", In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 10-12, 2021.

This material is based upon work supported by the Department of the Army under Air Force Contract No. FA8702-15-D-0001 and the National Advanced Mobility Consortium per U.S. Army Contracting Command Technical Direction Letter GVS OTA TR07; 70-201809. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of the Army.

INTRODUCTION

The U.S. Army Robotic and Autonomous Systems (RAS) Strategy identifies the need for advanced RAS capabilities over the near-, mid-, and far-term horizons to address mission challenges, maintain overmatch, and improve the combat effectiveness of the future force. Realizing the Army RAS Strategy requires the large-scale development, testing, iteration, and integration of software-centric technologies, such as autonomous maneuver and decision-making.

The **Robotic Technology Kernel ("RTK")** is a large government-owned and -managed library of reusable software packages which can be combined to form various subsystems comprising an "autonomy stack" for ground vehicles. RTK provides perception and localization modules that use different types of sensing hardware for object detection, ranging, classification, and vehicle pose estimation. RTK offers a variety of user-selectable path planners suited for varying environmental and autonomous navigation requirements. It also includes modules for vehicle management, motion control, communication management, and a variety of supporting development infrastructure for simulation and verification.

Over the past five years, U.S. Army CCDC Ground Vehicle Systems Center ("GVSC") has successfully employed RTK to demonstrate autonomous ground vehicle capabilities (e.g., autonomous route, waypoint, and convoy following) spanning many programs and mission scenarios. It is now a relied-upon source of software for both government and contractor developers on major programs such as Combat Vehicle Robotics (CoVeR), and Autonomous Ground Resupply (AGR).

RTK is based on the **Robot Operating System ("ROS"**), an open-source middleware software framework for robotic applications [1]. The framework provides software modules, tools, and interfaces that simplify the development of robotic behaviors and platforms. ROS enables collaborative robotic software development through common architectural constructs and open development based on the permissive BSD open-source license.

ROS originated in 2007 from a variety of institutions including Stanford University and Willow Garage, ultimately transitioning to Open Source Robotics Foundation and then to Open Robotics ("OR"), a non-profit organization. In 2014, OR introduced a second-generation version of ROS ("ROS 2") to address limitations in the original version ("ROS 1") and to support advanced capabilities, such as multiple robot teaming, real-time performance, and enhanced message and network security. Table 1 provides a summary of key differences between ROS 1 and ROS 2 and the resulting impact on robotic system development. ROS 2 has continued to mature to the point that OR has decided to sunset ROS 1, beginning with the release of its last version (Noetic Ninjemys) in May 2020 and the termination of support in May, 2025.

Given RTK's dependence on ROS 1 and the opportunity to take advantage of the new features built into ROS 2, GVSC decided to initiate the **Modular Autonomy and Robotic Software ("MARS")** project to migrate the RTK software modules from ROS 1 to ROS 2. Beyond ensuring the future sustainability of RTK, the MARS project has four additional goals:

- increase the use of software development best practices, such as unit testing and code analysis, in RTK development.
- increase the number of organizations contributing to RTK development,
- reduce the time needed to effectively use or contribute to RTK through documentation and changes to the RTK source code structure, and
- incorporate cybersecurity features and related best practices.

	ROS1	ROS2	Impact
Communications Middleware	Custom discovery and publish / subscribe	Data Distribution Service (DDS) using the Real-time publish subscribe (RTPS) protocol	• Established, real-time middleware • Support for multiple DDS vendors[1] • Multiple Quality of Service (QoS) settings • Security support (DDS-Security) • Steeper learning curve
Security	None[2]	DDS-Security + SROS2	• Authentication, access control • Data transmit integrity + confidentiality • Security event logging
Build system	Catkin	Ament	New build system to learn
Launch System	XML	XML and Python	• Improved control over launch behavior • More complexity
Languages	C++03, Python2	C++14/17, Python3	Access to modern language features
OS Platforms	Ubuntu 16.04	Ubuntu 20.04, OS X, Windows 10	Larger, actively supported OS footprint
Single-process support	Nodelet, add-on to core ROS	Components, integrated into core ROS	Enhanced support for running multiple modules within a single process
Lifecycle node	No standard approach	Well-defined lifecycle integrated into core ROS	Provides a standard approach to a common usage pattern
Support	Support ends 5/2025		Migration necessity

Table 1: Comparison of ROS 1 and ROS 2 elements. ROS 2 increases use of standard software components, supports additional platforms, and provides new features based on ROS 1 experience.

[1] Each DDS implementation requires an implementation of a ROS Middleware (RMW) to support abstraction of ROS2 topics, services, and actions

[2] ROS1 eventually bolted on an experimental secure ROS (SROS), which included a TLS shim between the network stack and the ROS client library.

CHALLENGES

Migrating an individual ROS software package to the ROS 2 framework requires a large number of changes to the source code and build files. Although the details depend on specific properties of each package, modifications typically include revising the executable and test source code to leverage the ROS 2 API, rewriting launch scripts in ROS 2's Python-based approach, replacing ROS 1 message definitions with their ROS 2 counterparts, and updating the build instructions and metadata to the ament architecture (the ROS 2 build system). Tools exist to automate simple changes in API calls, such as ROS 2's new use of the "msg" namespace. However, the high degree of flexibility deliberately designed into the ROS 1 architecture creates challenges in completely automating the migration due to the many different design patterns present in RTK and conceptual-level differences in ROS 2. The MARS program, along with much of the larger ROS community, has concluded that dedicated engineering effort to inspect and revise the source code is typically necessary to successfully migrate a package. While the effort to complete and test these changes for an individual package can be significant, the scope of the challenge is limited to the package. Migrating a large system of packages at the scale of RTK introduces additional technical and programmatic challenges which must be addressed to achieve the project objectives.

Dependency Management. A ROS package will often depend on functionality provided by other ROS packages. For example, a package named *C* may need to receive and process a message that is defined within a package named *B*. A large ROS system, such as RTK, will typically define hundreds of dependency relationships across the total set of system packages. The ROS build system requires that the migration of packages to ROS 2 be performed in dependency order. In the example above, package *B* must be migrated to ROS 2 before package *C* could be successfully compiled in ROS 2. Therefore, the dependency relationships impose a series constraint in the management of the migration process, limiting the degree of parallelism that can be applied to accelerate the migration timeframe or reduce project risk. A delay in migrating a single package could, if it is in the dependency chain of many other packages, effectively stall the entire project.

Distributed Team Structure. The level of effort needed to migrate a single package will vary widely based on many factors, such as the number of ROS API calls and the number of executable elements. At the start of the MARS program, past experience migrating ROS packages similar to RTK suggested that, on average, an RTK package would require approximately 80 engineering hours. A complete migration of RTK would therefore require at least 15 000 hours of engineering effort. Given the anticipated scale of the migration effort coupled with the need for continued capability development in the ROS 1 code base occupying the full-time effort of existing RTK developers, MARS program leadership concluded that a new team of engineers was needed to execute the migration. Given that ROS2 is a relatively recent development, there are very few developers familiar with it. Therefore, the MARS program elected to compose the engineering team from multiple industry and non-profit organizations. While the distributed team structure has the advantage of leveraging a diverse set of expertise, it also introduces a management challenge in how to effectively coordinate effort across organizational boundaries. An additional challenge is the assembled engineering team's varying and, in some cases, limited direct experience with the RTK code base.

Regressions. Every change made to a code base introduces the possibility of altering the behavior of the software which could impact the functionality or performance of the integrated

system. There are many potential causes of regressions in a ROS 2 migration, ranging from the developer making an error in the conversion of ROS API calls to subtle differences in timing in the underlying ROS communication system. Given the magnitude of changes required to migrate RTK, the introduction of regressions is a near-certainty regardless of the level of engineering expertise or rigor. Therefore, detecting the occurrence of regressions is a key challenge.

Cybersecurity. Security adds complexity to an already complex system, impacting performance. Segmenting the architecture to support concepts such as zero-trust (i.e., least privilege) and resiliency (i.e., continuous delivery of mission-critical functionality despite a cyber-adversary) is difficult without an intuitive implementation that scales with the number of nodes. Comingled ROS and non-ROS interfaces further complicates trust between components and how to mitigate. Adding security modifies the way developers write and test code. Given those challenges, it was important for the MARS program to address cybersecurity from the start of the migration as it forced the broader team to consider the tradeoffs while working to mitigate risk to a level acceptable to stakeholders.

APPROACH

To address challenges identified above, the MARS program developed a package-by-package migration approach that incorporates cybersecurity considerations, emphasizes continuous testing, and leverages an Agile program management methodology. The approach was documented in a pair of documents. An Agile Software Development Plan detailed the process and practices for managing the software effort, which is further described below. A Software Development Plan described the technical approach to the migration. Both documents were updated as needed based on periodic retrospective assessment.

Modular Software Migration. ROS packages function as the atomic unit of dependency, build, and distribution within a ROS system, so the MARS migration effort is naturally divided on package boundaries. It would therefore seem natural to allocate the task of migrating an individual package to development resources, i.e., individual developers or development teams within one participating organization, in a topologically-sorted dependency order. However, collections of packages often have related functionality. In RTK, for example, there are several different vehicle planning algorithms implemented in individual packages. Allocating work to resources based purely on a topological sort may fail to take advantage of similarities across groups of packages. Therefore, MARS grouped RTK packages into 14 subsystems: Autonomy Mode Manager, CAN A-Kit Bridge, Configuration, Diagnostics, IOP Bridge, Localization, Motion Execution, Navigation, Visual Perception, Sensors, Perception, Tools, Vehicle Management System, and World Model. Each subsystem was allocated as a whole to participating organizations to facilitate development of expertise within each subsystem. Developers collaborated across subsystems to identify a package migration order that satisfied dependencies.

The program prioritized migration of packages needed to perform autonomous capability on a specific ground vehicle class, called the MRZR, in order to facilitate system-level testing on a hardware platform. 121 of the approximately 200 packages composing RTK are needed to operate the MRZR. These 121 packages include roughly 5500 files and a total of 1.3 million lines of source code. The remaining packages consist of those needed to control other vehicle platforms, which will be migrated to ROS 2 at a lower priority, and

deprecated packages, which will not be migrated to ROS 2.

The work to migrate an individual package was divided into four steps. The first step requires inspecting the ROS 1 package source code and any available documentation to create an XML-based interface model and behavioral description of its executable elements, i.e., nodes and nodelets. Next, the unit and node-level tests are added to the ROS 1 version of the package to facilitate regression testing. In the third step, the package source code and tests are migrated to ROS 2 with minimal changes to the overall source code structure. This step is intended to be a rapid migration to enable the migration of any dependent packages. In the final step, the source code and tests are revised to meet project quality goals, including resolution of any issues elicited from static analysis tools.

Agile Collaboration. To address the need for management and execution flexibility, the MARS program adopted an Agile Scrum methodology. Beginning with a collection of software requirements or project expectations known as the Product Backlog, Scrum defines a process for selecting items from the Backlog, working on them for a fixed period (a "Sprint"), reviewing the work and making decisions about quality and completeness, and then repeating the cycle to finish uncompleted items or taking more items off of the Backlog.

The Product Backlog for the RTK migration effort consists of one or more of the four steps used to migrate an individual package, described above. At the beginning of each sprint, during the Scrum Sprint Planning meeting, the development organizations state which steps they expect to execute for which packages. The status of each committed step is reviewed at the end of the Sprint at the formal Sprint Review before developing the plan for the next Sprint.

For tracking purposes, each package/step combination is tracked as a Story in the Jira issue tracking system. The Story descriptions and completion criteria are spelled out in the body of the Story definition and are the same for every step regardless of package. The Jira Sub-task issue type is used to list the technical tasks that complete each Story. Sub-tasks are assigned to individual contributors from the partner organizations. Sub-task status is initialized to Backlog when created, and individual contributors change the status to In Progress and Done as work progresses. When all the Sub-tasks for a Story are complete, the status of the Story is set to Done also. The combination of Story, Story Status, Sub-task, Sub-task Status, and assignment provide accurate quantitative insight into the status of the overall effort at every point in time.

Modeling. To document the interfaces and behavior of the executable components within the RTK system, the MARS program developed a set of models utilizing tools and techniques commonly associated with Systems Engineering. The RTK models are based on an XML schema with elements aligned with ROS domain concepts. The schema and structure allows for the ROS interfaces to be captured in a standard way. The ROS concepts defined in the profile include nodes, messages, services, actions, topics, packages, and nodelets.

The Systems Engineering community typically leverages applications based on SysML models, not custom XML schemas. To facilitate collaboration, the MARS program developed a tool to convert from the MARS XML models to a SysML model within the Magic Draw software application.

Testing. The MARS program leverages software testing to detect any regression in package functionality across the migration from ROS 1 to ROS 2. Tests applied to the ROS 1 implementation of a package are expected to execute with the same results in the ROS 2 version. Many packages in the existing ROS 1

RTK code base lack the tests necessary to detect regressions. Therefore, the MARS program must develop a suite of tests for the ROS 1 version of packages.

The MARS program tests code at three levels as appropriate: unit, integration, and system level tests. Unit level tests validate individual functionality of small pieces of code. Integration level tests combine multiple pieces of software and typically require more sophisticated or complex input data. System level tests run large amounts of code, typically in a fully simulated environment.

Developers created function-level unit tests for the ROS 1 version of RTK software packages where the structure of ROS 1 RTK package is amenable to such unit testing. Developers sought to develop ROS 1 unit tests with a structure that would support their use with minimal changes in the migrated ROS 2 RTK package.

Developers created function-level and node-level unit tests for RTK software packages migrated to ROS 2. A goal of 80% line coverage was established and adhered to except for small and well-justified exemptions. Developers were given the freedom to exercise engineering judgement in deciding not to develop tests for certain functions, files, or other units of code. These decisions were documented in Jira.

Developers created node-level tests, i.e., tests that used the ROS interface, for both ROS 1 and ROS 2 RTK packages. Node-level tests were designed to maximize code similarity across the ROS 2 migration. Developing a test harness that isolates the test node's ROS interface from the test logic is one recommended approach.

Many packages in RTK require interaction with hardware, such as sensors and platform control systems. However, not all hardware was always available at test time, making achieving software testing goals challenging. Due to the complexities involved with shipping and running hardware at multiple physical developer locations, the decision was made to test these hardware packages in software-only environments.

Two approaches were used to test these types of packages. First, software emulators were built to mimic the hardware system behavior, such as connecting to a TCP socket and streaming data packets. The second approach was data replay. Data captures from physical hardware, such as TCP packets or serial streams, were replayed to the unit under test..

Development Automation. The MARS program established a suite of tools and infrastructure, known as a DevSecOps system, for automating the build, testing, and reporting process. The primary objective of the DevSecOps system is accelerating MARS development velocity and improving code quality through efficient access to information and artifacts. The ready availability of package status and metrics supports informed collaboration and decision making across MARS participants and stakeholders.

A pipeline was established to provide feedback on all ROS 1 and ROS 2 development work. These builds performed the following tasks for every package being targeted in a particular run:

- Verify dependencies install correctly from configuration files within the package.
- Confirm the source code builds without any errors.
- Run all functional tests.
- Run all static analysis tools.
- Publish dashboard with one-view summary and navigation of CI results.

The static analysis executed includes the following common tools:

- GCov – code coverage during test
- Cloc – count lines of code
- Cpplint – C++ linter
- Flake8 – Python format check
- Lint Cmake – CMake format check
- Pep257 – Python docstring check

- Xmllint – consistent XML style
- Clang-format – C++ format linter
- Clang-tidy – C++ checker for common programming errors

The DevSecOps infrastructure supports system-level testing by integrating a physics and environment simulation engine to accurately model sensor and platform behavior.

Cybersecurity. The MARS program established a cybersecurity team to address challenges from the start of the MARS project. The cybersecurity team provides guidance to the migration team and establishes the technical foundation for enabling security within the runtime RTK system.

Implementation of security controls, through DDS-Security, is a desired end goal of the migrated ROS 2 RTK system. DDS-Security is a security model and service plugin to the DDS specification [2]. It includes plugins for authentication, access control, cryptography, encryption, and logging. The MARS program developed an experimentation environment to assess the impact of security on system performance. The environment uses Apex.AI's open-source performance test package [3], [4]. The team had to make a few modifications as, at the time, the performance test environment targeted ROS 2 Dashing distribution rather than Foxy, which is the targeted distribution for MARS. Specifically, the package's support for DDS-Security in the Foxy distribution was incorrect as the secure ROS 2 (SROS2) application programming interface (API) changed to use enclaves, different policy formats, and environment variable names in Foxy. Rectifying this incompatibility required modification of the code's naming scheme and updates to the tools and startup scripts to support testing with and without DDS-Security.

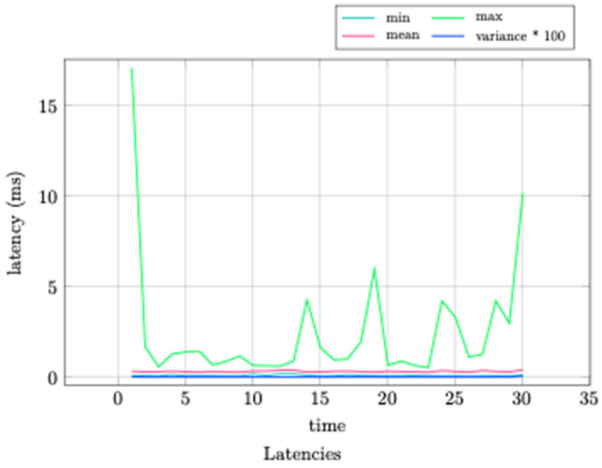
The experiments were run in a VMWare™ Ubuntu 20.04 guest with 16GB of dedicated RAM and 3, 2.9GHz CPU cores. Two ROS middleware DDS implementations were evaluated, FastRTPS and Cyclone DDS. Each experiment involved one publisher sending a 1000-byte message to one subscriber at a frequency of 1000 messages/second over a total time of 30 seconds. UDP was the transport mechanism and the DDS reliability quality of service was set to BEST_EFFORT.

RESULTS

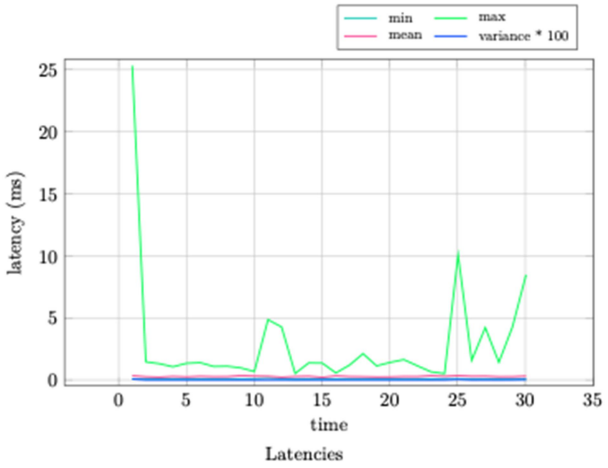
Package Migration. Following a planning and study phase, migration of RTK packages began in June 2020 and is continuing. Over a one-year period, the MARS program has migrated a total of 81 packages to ROS 2 and partially migrated an additional 34 packages. The migrated packages span all subsystems and represent many different types of packages, such as message-only packages, Python-based packages, and packages with C++-based nodes and nodelets.

Developing tests for each package is a substantial component of the MARS effort. The program, to date, has developed 1100 unit and node-level tests that have raised the average code coverage in RTK from 6.6% to 25.7%, an increase of 289%. Furthermore, these tests have identified several errors in the ROS 1 version of RTK that, in addition to being fixed in MARS, have been reported to the upstream ROS 1 RTK project.

Security performance. Figure 1 shows the results of testing ROS 2 message passing latency performance with and without security enabled. The results, shown for the FastRTPS middleware implementation, provide evidence that, on average, the DDS-Security plugins do not appear to have a significant impact on performance. Tests using the CycloneDDS implementation, not shown, give similar results. These experiments will be extended in the next phase of the MARS program to inform decisions as to where to apply DDS-Security controls within the ROS 2 RTK system.



(a) FastRTPS without DDS-Security



(b) FastRTPS with DDS-Security

Figure 1: Latency measurements of ROS 2 using FastRTPS DDS (a) without DDS-Security and (b) with DDS-Security. This experiment is publishing a 1K byte message at 1 000Hz (1000 messages/second) for 30 seconds. The ROS 2 publisher and subscriber are running in the same virtual machine but are transmitting messages across their UDP/IP stack. Latency is the time from when the publisher sends the message to when the subscriber receives the message. The results show a worst-case, maximum of 38% increase in latency, but on average latency is almost identical.

ANALYSIS

Qualitative analysis of the MARS program execution has identified strengths and weakness of the approach used to migrate RTK to ROS 2.

Agile Scrum. The structure of Scrum, which facilitated shared understanding and collaboration across the team, has proven to be an essential component of the migration success. To understand the benefits of the approach, consider an alternative model in which each partner organization managed their own work scope and periodically reported out status. Without the consistency of the package/step structure and tracking in Jira, tracking progress and understanding overall status would have been extremely challenging. The common approach eliminated these challenges. It normalized the language used by individuals from across the project to talk about progress, obstacles, and questions, making discussions efficient. The repetition of the Scrum cycle of planning, Sprint, Backlog review, and Sprint review helped the team fall into a regular cadence in which expectations were known, exceptions could be handled, status could be tracked, and workload was adapted and shifted to take best advantage of available resources.

Scrum was also used to manage the effort to build and support a Continuous Integration / Continuous Deployment (CI/CD) pipeline and to explore the capabilities of the ROS 1 Bridge, a software component that mediates message package exchange between ROS 1 and ROS 2 nodes. Both of these efforts followed the same Scrum methodology described above. In both cases, requirements were less structured than the repetitive package/step pattern used for the migration work proper, but all other aspects of Story definition, Sub-task creation and assignment, planning, tracking, and review were the same.

The ROS 1 Bridge was an unknown quantity to the team. They used an iterative approach to

building the Product Backlog by starting with small projects to demonstrate the Bridge working for simple talker/listener configurations, gradually including the build of the Bridge in the CI/CD pipeline, and finally encompassing inclusion of RTK message packages that had been migrated to ROS 2 to build a Bridge that should work for all packages being migrated.

Collaboration and Diverse Expertise. The effort to transition rosbags, the native ROS data capture format, from the ROS 1 API to the ROS 2 API highlighted the benefits of community-driven development and the importance of coordination tools in efficiently solving problems. Rosbags were central to many of the tests written for the project and were therefore a need in ROS 1 and ROS 2 as a mechanism for persistent data recording.

The rosbag API was expanded during the re-design for ROS 2 to incorporate additional use cases and middlewares. Despite DDS being the default middleware of ROS 2, the implementation of rosbags in ROS 2 includes facilities to record and replay other data formats such as protobuf and ZeroMQ. Additionally, the requirements for rosbags in the ROS 2 environment incorporated determinism, adaptability, scalability, random access, ranged access, variable chunk sizes, and backwards compatibility with ROS 1.

This expansion of use cases caused issues when attempting to translate newly designed tests reliant on rosbags from ROS 1 to ROS 2. First, the rosbag structure was different and required resampling the robags using the new ROS 2 format. In many instances, the interfaces that ROS 1 tests relied on were not available in ROS 2. In these cases, the project communication tools such as Discourse, Confluence, and Jira, coupled with regular Agile Scrum meetings, afforded the developers space to discuss the issues and coalesce on an approach that was both true to the project at hand as well as the long-term goals of ROS 2.

Cybersecurity. The reasons for migrating to ROS 2 vary greatly from project to project, but improved security features in ROS 2 are likely to be high on any list of reasons for making the switch.

Introducing new cyber infrastructure and features to any system can be a daunting task. A port to ROS 2 will include the need for introducing new cybersecurity strategies as well as adapting code to ROS 2 generally. Based on the experience of the MARS program, it is recommended to assess desired security goals at the outset so that related migration challenges can be addressed from the beginning and not as an afterthought.

DevSecOps. The DevSecOps systems proved to be highly effective in providing developers timely feedback on the status of their contributions. In general, the use of pipelines elicited errors in process sooner rather than later. For example, Git strategy errors were caught and corrected early in the code merge process. Additionally, by running in standard container instances, common problems such as code that “works on my machine” were avoided.

The DevSecOps system also helped ensure continuous enforcement of project objectives. The Agile Product Owner was able to quickly establish what lines of code where being executed during the developed tests and make informed decisions about the acceptance of developer contributions. The system established confidence that the resulting merge of development efforts into the mainline repositories was stable and up to project standards.

The DevSecOps processes described here enabled developers, product owners, and stakeholders to execute the RTK conversion with increased velocity and confidence.

Software Testing. Overall, the application of software testing has proven useful in identifying a number of issues with the original ROS 1 RTK

implementation as well as ensuring that the ROS 2 migration did not introduce regressions. Continued development and use of software testing at the unit, node, and system level within RTK is recommended. However, experience from the MARS effort revealed that designing effective tests, particularly for existing software, can be challenging and time consuming. Increased emphasis on test-focused training with additional test code examples at the start of the MARS program would likely have provided value and improved consistency. Furthermore, the development of software components providing a node-level test harness may have accelerated test development by facilitating reuse and commonality.

Technical Debt. Tradeoffs made during early development efforts often result in the accumulation of technical debt associated with software systems. For example, in the case of RTK, it was determined that increased automated unit test code coverage could improve early detection of porting errors. Availability of software design models could improve understanding of how RTK is intended to operate, giving developers an independent check of ported code behavior.

With these expected benefits in mind, the MARS program decided to address both of these areas. Doing so has improved the confidence in both porting decisions and execution. While these were the two areas of technical debt the MARS program chose to address, other programs may want to consider other areas. The key is to assess all areas of technical debt before beginning to port software and decide whether it is finally time to address some areas to accelerate the migration schedule or improve the quality of the outcome.

FUTURE WORK

The MARS program is expected to continue through June of 2022.

Hardware Demonstration. Deploying and demonstrating the ROS 2 RTK system on a physical robotic platform is a key planned milestone. The program has developed a system-level test plan that exercises a suite of RTK features, such as obstacle avoidance and obstacle avoidance. Executing the test plan with an MRZR vehicle running the ROS 1 version of RTK, shown below, established a system-level functionality baseline. The program will repeat the test plan with the ROS 2 implementation of RTK and compare the performance with the ROS 1 version in order to validate that the ROS 2 migration was successful. Additionally, the system will be similarly tested and analyzed with ROS 2 security components enabled.

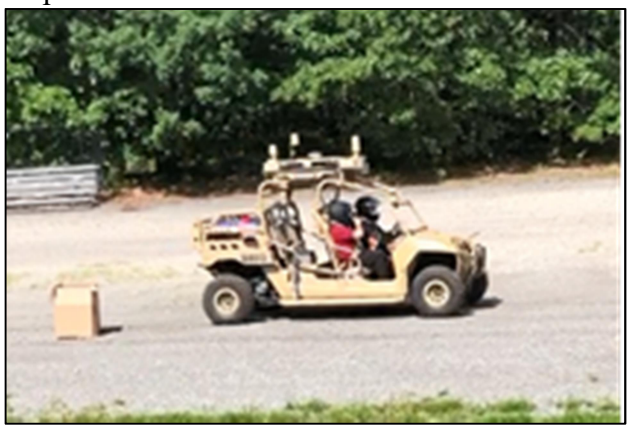


Figure 2: Testing the ROS 2 RTK system on a physical robot platform, such as the MRZR pictured here, is a critical component of the ROS 2 RTK verification effort.

Sustainment. While completing the migration of RTK to ROS 2 is a key objective, the program also seeks to serve as a template for the future development and maintenance of RTK as a common core autonomy library. The MARS program has demonstrated the feasibility of executing a complex software effort by a large team composed of Government and Industry participants. Additionally, the program has demonstrated the value of testing and automation to enhance and maintain code quality. By leveraging the approach used by MARS, the Army

Autonomous Ground Vehicle community could establish a well-engineered, robust, and flexible software system to serve as the foundation for future capability development and transition.

ACKNOWLEDGEMENTS

The MARS program is a large collaborative effort with many participants. The authors acknowledge the effort of the full team in contributing to the planning and results underlying this paper.

REFERENCES

[1] Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." ICRA workshop on open source software. Vol. 3. No. 3.2. 2009.

[2] "DDS-Security Specification Version 1.1," Available: https://www.omg.org/spec/DDS-SECURITY/1.1/About-DDS-SECURITY/ , [Accessed December 15, 2020].

[3] A. Pemmaiah, D. Pangercic, D. Aggarwal, K. Neumann, K. Marcey, "Performance Testing in ROS2," Available: https://drive.google.com/file/d/15nX80RK6aS8abZvQAOnMNUEgh7px9V5S/view [Accessed December 15, 2020].

[4] K. Scott, W. Woodall, C. Lalancette, "2020 ROS Middleware Evaluation Report," TSC-RMW-Reports, November 5, 2020, Available: https://osrf.github.io/TSC-RMW-Reports/ [Accessed December 15, 2020].