Autonomy Subteam Research and Development for Fish 'n Ships

Associated Members of the Autonomy Subteam and other contributors

Abstract — This report details the advanced control strategies and research operations undertaken by the Arcturus Autonomy Subteam in implementing the software that powers Fish 'n Ships. We describe both the theoretical underpinnings of our work, its technical significance and general implications, and how each concept or idea was implemented and used in Fish 'n Ships. The scope of our work is broad–encompassing both technical performance improvements specific to the ROS architecture and more theoretical results developed as part of our research and development operations. In full, the result of our efforts has been to enable completely autonomous and efficient localization, mapping, navigation, and error-correction for a small boat based on intelligent perception.

1 Software Deployment Model

This year we have investigated alternatives to our longstanding software deployment model, which has traditionally required the careful installation of certain system dependencies and uses CMake to build various modules required for ROS nodes to work. This system suffered from numerous flaws, leading to an inevitable spiral towards what is known, in the relevant literature, as "dependency hell" [1].

Associated Members of the Autonomy Subteam and other contributors
Massachusetts Institute of Technology (MIT), 77 Massachusetts Ave, Cambridge, MA, email: arcturus-logistics@mit.edu

1.1 Motivations

The main problem that the patchwork dependency installation scheme created non-reproducible environments. It was practically impossible to construct two identical computing environments capable of running the ROS nodes. Additionally, the only way to natively run much of the Arcturus codebase required installing a specific version of Ubuntu, effectively mirroring the environment setup of the computer the nodes would eventually be run on. In addition to system dependencies required to run the ROS nodes, testing required the installation of unrelated visualization components, which often cluttered development environments. Lastly, dependency specification was incomplete, non-standardized, and difficult to follow, leading to manual installation of system dependencies (almost always unspecified) and a collection of random hacks to install remaining packages. This is the textbook definition of "technical debt," or what is now being recognized as "maintenance load" [2].

In this section, we outline how to apply a purely functional software deployment model to a set of ROS modules intended to be run in production on a Jetson Nano. We note that this experiment builds on previous work in this space, most notably [3] and [4]. We begin by outlining some of the immediate benefits of a purely functional software deployment model in the context of the ROS ecosystem, and then move onto some of the practical challenges involved in implementing a pure build system, what progress has been made so far, and what future work remains to be done. We believe that the experiments we have conducted so far lay the groundwork for a completely automated software deployment, management, and testing environment that can be run at scale.

1.2 Background and Related Work

A purely functional software deployment model is one that leverages the principles of purely functional software design to creating package build scripts. The central idea of this model is that every package will be installed using a build function that is "pure" in the mathematical sense, by which we mean that it produces the an output determined uniquely by its inputs. Running this function multiple times with the same input should yield identical outputs. The inputs of such a function are the packages it depends on, and its output is a list of configuration settings and metadata associated with the package. Purely functional build tools can use this metadata to identify the exact version of the package to install and create an environment containing that package with those specific configuration settings applied.

The most notable purely functional build tool in use at the moment is Nix, originally proposed by Eelco Dolstra in his seminal 1978 thesis [1]. While the core idea behind purely functional software deployment is simple–the composition of pure build scripts to create complex yet reproducible computing environments–its benefits are far-reaching and powerful.

We use the Nix packaging ecosystem, `nixpkgs`, a repository containing a set of pure build scripts for a plethora of common system dependencies, along with the Nix build tool, as the basis for our experiment. However, `nixpkgs` does not contain the pure build scripts for ROS packages, so we must rely on another packaging repository to fill in this gap. We find [3] to be the most reliable provider of these packages and incorporate it into our environment setup.

1.3 Potential Benefits

While [3] does not provide a binary cache for the Humble distribution, it is not difficult to setup a Hydra build server that can distribute these binaries for the packages we need. We are currently working on setting up such a server, but we suspect this will greatly speed up package builds, since it will be faster to use Nix's binary cache ecosystem over a local CMake build. Indeed, reports from conferences of Nix academics confirms our prediction [4], suggesting that we can speed up build times approximately five-fold and reduce resource consumption by about 90%.

Of course, this is just one of the many benefits that we obtain by using a purely functional build environment. Aside from enhanced reproducibility, we gain clear and understandable dependency specifications, making it easier to locate and patch security vulnerabilities as well as enabling us to update packages without fear of unforeseen breakages later down the line. We are able to create a unified specification of both system and ROS dependencies for an individual project, and we can also use such a specification to create automated runners for testing and debugging purposes. As should be quite obvious now, the rammifications of creating such a reproducible build system are large, and they can lead to enormous reduction of maintenance loads.

The clear specification of all dependencies also allows Nix to perform "true delta" builds [4], where Nix can identify the exact subset of packages that need to be rebuilt or replaced and apply those changes without performing unnecessary work (e.g. rebuilding dependencies of updated packages that stay the same). Nix eliminates nonsensical bikeshedding [5] over ROS versions and distributions, unifying package declarations in a single monorepo that can be used as an input to a pure build function for the project environment. Compared to Docker, the industry standard for reproducible builds which has been the standard build system on Arcturus on non-Ubuntu systems, Nix offers the huge benefit (in addition to those previously described) of being able to identify and utilize system hardware resources and run code natively without virtualization overhead.

1.4 Implementation Details

1.5 Future Work

Bibliography

1. Dolstra, E.: The Purely Functional Software Deployment Model, (1978)
2. Troy, C.: Stop saying “technical debt”, https://stackoverflow.blog/2023/12/27/stop-saying-technical-debt/
3. Wolsieffer, B.: lopsided98/nix-ros-overlay, https://github.com/lopsided98/nix-ros-overlay
4. Purvis, M., Wanders, I.: Better ROS Builds with Nix, http://download.ros.org/downloads/roscon/2022/Better%20ROS%20Builds%20with%20Nix.pdf
5. jerf: https://news.ycombinator.com/item?id=40701501