Bazel and ROS 2 – Building Large Scale Safety Applications

Kilian Funk, Karl Wallner

October 19-21, 2022 at ROSCon '22
Large scale safety applications. How hard can it be?

Function and Performance in a distributed system

- Efficient communication for large data
- Realtime requirements

Fail-Safe or Fail-Operational Behavior

- Error-detection
- Redundancy
- Special Hardware (e.g. lock step)
- Architectural measures

Well defined Process

- Compliance to State of the Art (e.g. V-Model, ISO 26262)
- Safety Case (HARA, FMEA, FuSaCo, TeSaCo)
- Verification and Validation
- Standardized code creation (four eyes, etc.)
- CI/CD for a large team of developers

- Traceability
- Reproducibility
The safety case relies on our ability to prove transparently:

- why our application behaves like it does
- what exactly we have developed and deployed into the field

Arrows show dependencies that shall be traceable.

- Along the V for artifact creation
- Right to left for verification and validation
The safety case relies on our ability to prove transparently:

- why our application behaves like it does
- what exactly we have developed and deployed into the field

Arrows show dependencies that shall be traceable.

- Along the V for artifact creation
- Right to left for verification and validation
Comparing Colcon/CMake vs. Bazel

**Colcon/CMake**

- Imperative macro language, different (redundant) definitions
- Various different build tools (CMakeLists.txt, setuptools, Make)
- C/C++ as (main) target language
- Caching not safe, due to potentially missing dependencies
- Building in host environment
- Compile time and runtime dependency discovery

**Bazel**

- Declarative, abstract definition of build
- Full programming language (starlark) for extension/customization
- Multi language support
- Integrated (reliable, artifact based) local and remote caching
- (Almost) Hermetic build in sandboxes
- Explicit (full) dependency tree

Disclaimer: Biased towards use case of large scale safety applications!
Things to consider when migrating to Bazel

**Colcon/CMake**

Artifacts are tightly organized in **packages**

Package names must be unique

---

**Bazel**

Strongly **artifacts** oriented; packages are mainly to improve clarity for user

Workspace names must be unique.
Package and target names are absolute paths within a workspace

```build.bazel
# BUILD.bazel
cc_binary(
    name = "simple_publisher",
    ...
)

msg_library(
    name = "my_msgs",
    ...
)

ros_pkg(
    name = "my_cool_pkg",
    executables_lib = ["simple_publisher"],
)```
Package deployment

**Goal:** Provide a means to “install/setup” bazel built ROS 2 packages (C++, Python for now) onto a target for usage with ros2cli

**Non-Goal:** Provide a pre-built ROS 2 package that can be dependent on for building other ROS 2 packages

```
cc_binary:
  :some_executable

$ # Execute directly in your bazel workspace:
$ bazel run :some_executable
```
**Goal:** Provide a means to “install/setup” bazel built ROS 2 packages (C++, Python for now) onto a target for usage with ros2cli

**Non-Goal:** Provide a pre-built ROS 2 package that can be dependent on for building other ROS 2 packages

```bash
$ # Execute directly in your bazel workspace:
$ bazel run :some_executable

$ # Execute installed version:
$ bazel build :install_archive
$ bazel run :install_archive.install -- $INSTALL_PATH
$ source $INSTALL_PATH/setup.bash
$ ros2 run my_cool_pkg some_executable
```
**Goal:** Provide an extensible multi-language concept for message code generation

- `msg_library` rule only provides information about the input
- no output artifact is generated
Message generation

**Goal:** Provide an extensible multi-language concept for message code generation

- `msg_library` rule only provides information about the input
- no output artifact is generated

- `cc_msg_library` aspects are instantiated by the user of a message on demand
- they generate the “linkable” library for the required language
Message generation

**Goal:** Provide an extensible multi-language concept for message code generation

- `msg_library` rule only provides information about the input
- no output artifact is generated

- `cc_msg_library` and `py_msg_library` aspects are instantiated by the user of a message on demand
- they generate the “linkable” library for the required language
**Goal:** Keep as much of the federated repo concept as possible

- Bazel build configuration can be added on top of existing ROS 2
- A pinning mechanism is introduced to ensure reproducibility
Setup a workspace to use the ROS2-Bazel fork

- Install bazel/bazelisk on your host system
- Add the WORKSPACE file to an empty folder
- Add (.bazelrc, .bazelingore, .bazelversion) as needed
- Create your own package within your workspace

```python
# WORKSPACE
workspace(name = "my_cool_workspace")

http_archive(
    name = "ros2",
    url = "https://github.com/ApexAI/rules_ros/.../rules_ros-x.x.tgz"
    sha = "xxxxxx",
)

load("@ros2//bazel/rules_repo:defs.bzl", "configure_ros2")
configure_ros2(distro = "humble")

load("@ros2_config//:setup1.bzl", "setup1")
setup1()

[...]
load("@ros2_config//:setup4.bzl", "setup4")
setup4()
```
Summary

- Bazel is an alternative to the native ROS 2 build system Colcon/CMake
- Traceability from deployed software back to source code is achieved by a complete dependency tree
- Reproducibility can be achieved through a hermetic build
- We have shown how ROS 2 can be set up with bazel including core concepts like message generation and package deployment

We are in the process of open sourcing the contents of this talk:

http://github.com/ApexAI/rules_ros

We are looking forward to your feedback on github.