École Polytechnique Fédérale de Lausanne

Software architecture of a formula student autonomous race car

by Adrien Remillieux

Master Project Report

Prof. Mathias Payer
Antony Vennard
Thesis Supervisors

June 10, 2022

Acknowledgments

I'd like to address a special thanks to the Racing Team members, professors and EPFL's personnel that help the project one way or an other. I met many great people and I learned a lot by doing this semester project in the Racing Team.

This report is based on the LaTeX template from EPFL HexHive lab. It was nice from them to publish it on Github for everyone to use [5].

Lausanne, June 10, 2022 Adrien Remillieux

Abstract

In this project we created a framework to build and run the software used in an autonomous race car and manage its dependencies. We went with a microservice approach with a clear separation of concerns between nodes (vision, control, data acquisition...) and they communicate using publish / subscribe mechanisms or RPC. We used ROS (robot operating system) libraries for communication so the nodes can be written in C++ or Python depending on which language fits the task best. We did not use the whole ROS framework because we need to have multiple versions of some common Python libraries and that would have been difficult to accomplish so we built a custom dependency management, build and launch system. We also wanted to be able to launch nodes directly in an IDE to debug them so we reverse engineered how ROS nodes are launched. Finally we used cgroups to limit the number of cores that can be used by a node and to prioritize one node's execution over an other.

Contents

Acknowledgments **2**

Abstract **3**

1 Introduction **6**

2 Background **7**

3 Requirements **8**

4 Design **9**
4.1 Organization of the code and dependencies 9
4.2 Dependency management 11
4.3 Compute resources management 12
4.4 Launch system 12
4.5 Configuration files 13
4.6 Containers 13

5 Implementation **14**
5.1 Dependency management 14
5.2 Launch system 15
5.3 Monolith split 15
5.4 Jetson setup 16
5.5 Fast multidimensional array transfer 17
5.6 IDE configuration 18
5.6.1 Pycharm 18
5.6.2 CLion 18

6 Evaluation **19**
6.1 Publish - Subscribe performance 19
6.2 Development experience 19
6.3 Runtime platforms requirements 20
6.4 Dependency Management 20
6.5 Launch system and node runtime management 21

6.6 Up and running time and difficulty from factory reset 21
6.7 ROS lock-in 21

7 Conclusion **22**

8 Further work **23**

Appendices **25**
A - Launcher output 25

Bibliography **27**

Chapter 1

Introduction

This project started because the prototype program for the autonomous control of the car was getting more and more bloated and harder to work on, so splitting the program into independent parts with clear scope (vision, control, sensor acquisition) was needed. The prototype was also purely sequential and we need to run different algorithms on same or different sets of data at different rates. When looking at possibles ways to improve the situation, ROS seemed to be the best choice because it is a popular framework to program robots as a collection of microservices and to do so it offers communication libraries, build and launch tools. There is also a large set of drivers nodes to interface with sensors and nodes to accomplish a specific task that are community supported.

Splitting the program into nodes allow one person to work on a part of autonomous driving pipeline without having to understand how the whole pipeline works which is necessary as the complexity grows. Clear separation of concerns between nodes also has the benefit of making them swappables with improved versions, simpler and easier to test.

The key challenge and at the same time the cornerstone of this project is ROS. It is very opinionated on how things should be done and if you need to step aside from the "true path" things get difficult quickly. This was the case with our requirement to have different versions of the same dependency between nodes and the initial requirement to make ROS run on an unsupported Linux distribution.

We created eight python packages from the main monolithic program using separation of concern and dependencies as criteria. Each package can have multiple nodes, for example there is one package for the control algorithms that will contain multiple nodes soon. ROS' build, dependency management and launch system was not a good fit for our use case so we built our own but we kept the good communication libraries and the ability to use community made nodes.

Chapter 2

Background

Pub / Sub: Stands for publish / subscribe and is the concept that a program can publish data to a topic (an ID) and subscribe to topics by specifying a function to call when a message is received on a topic. It is often used across a network but works very well locally too.

RPC: Stands for remote procedure call and is the concept that a program can call a function in an other program. It is closely related to pub / sub.

Environment variables: Often shortened env vars. These are variables that can be set by a program that execute an other program to pass informations. On a normal Linux desktop distribution many variables are set by the OS to indicate which user is logged in and what is the language for example.

For this project these variables are of particular interest : `LD_LIBRARY_PATH` tells the dynamic linker to go search for libraries in a semicolon separated list of paths, `PATH` is a semicolon separated list of path where a program that calls an other binary by name will look for the binary and `PYTHONPATH` is a semicolon separated list of paths where Python will look for packages.

Marshalling: it's the process of transforming the memory representation of an object into a data format suitable for storage or transmission. It is typically used when data must be moved between different parts of a computer program or from one program to another [2].

Dependency management: it is the concept of getting the correct version of libraries (from the internet or locally) and making them available to a program.

Compute resources management: Is the concept of restricting the resources available to a program (Eg. the number of cores that can execute it) or prioritizing the execution of a program over an other when there is contention.

Chapter 3

Requirements

The main requirements are:

1. Python and C++ nodes communicate efficiently using pub / sub and RPC. Python nodes can send and receive numpy arrays efficiently. The metrics are ease of use, throughput and latency
2. Develop with all the niceties modern IDEs have to offer, so import resolution of external dependencies has to work and it should be possible to launch a or multiples node in the IDE attached to a debugger for effective debugging
3. Runs on a Nvidia jetson (prod) and on a computer (Simulation / ML Training / Dev), so runs on ARM64 and x86-64 with Ubuntu 20.04
4. Automatic dependency management
5. Multiple versions of some dependencies across the nodes possible because some dependencies require specific and old version of other common dependencies
6. Automatic launch and node runtime management system
7. Guarantee resources to a node and limit the amount of resources a node can consume if there is contention

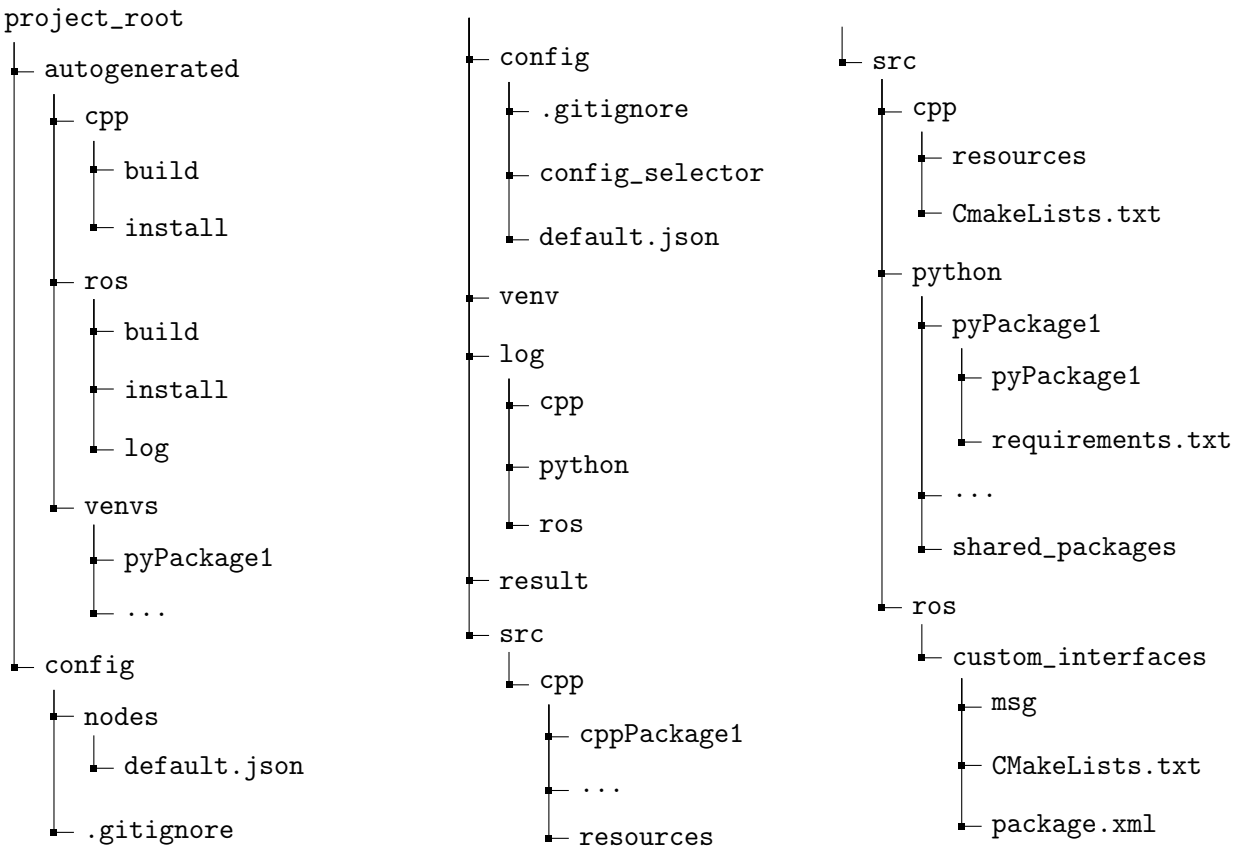
There are also secondary requirements:

1. Be able to run a setup script on a blank Jetson, then to clone the git repository, run the dependency manager tool and finally have a functional control module in $\tilde{3}0$ minutes. No complicated and finicky setup. Hardware should be mostly replaceable and the OS easily reinstallable if something bad happens
2. We should not lock ourselves in ROS to be able to switch to something else more easily in the future if needed because Python and C++ will run anywhere but ROS not

Chapter 4

Design

4.1 Organization of the code and dependencies



The `autogenerated` folder contains files derived from other files. It's possible to delete its content and recreate it using the dependency manager tool. More precisely:

- cpp folder: build contains the build artifacts, install contain the binaries. We chose the default names from ROS for consistency
- ros folder: build contains the build artifacts, install contain the binaries & header files and log the build logs
- venvs folder: contains the Python virtual environments created for each packages with the package-specific dependencies installed

The `config` folder contains the files used to chose many parameters to adapt the program to different scenarios (control of the car, development, simulation...). More precisely:

- nodes folder: contains the configuration to select which nodes to launch and with which resources limits
- .gitignore: used to exclude the config_selector file
- config_selector: used to select which config filename to load. The launcher also look for an environment variable
- default.json: The default configuration. Will be loaded if launcher has no other instructions. This file or other configuration files in this folder must contain a reference to a configuration file in the nodes folder. We made the decision to split the configuration file so that it would not become gigantic. It also makes it easier to test different resources limits

The `venv` folder is the virtualenv used by the IDE to find all the dependencies of the project for code hints, type checks and other niceties during development. It was unfortunately necessary to create this virtualenv with all the most recent versions of the dependencies we use because it is not possible for the IDE to know in which virtualenv to look when a file import a dependency for example. So it is possible that the suggestions will not be correct and in this case it is possible to tell the IDE to use a specific virtualenv for the code hints.

The `log` folder contains all the logs produced by the nodes during runtime. Each python file can call a logger that will automatically log in a file named using the schema "<date>-<program-entrypoint>".

The `result` folder contains all the data we may want to examine after a test or simulation run for example. It's mostly graphs that plot the trajectory of the car between cones for now.

The `src` folder contains all the source code. More precisely:

- ros folder: ros specific sources. The folder custom_interfaces is where we can define C-like structs for inter node communication when the built-in types of ROS are not enough

(strings, bytes, standard numerical types, multidimensional arrays of standard numerical types)

- cpp folder: one folder per C++ package in which there is also a CmakeLists.txt file for instructions on how to build the binaries of this package. The "resources" folder in which there is a simlink to a folder inside "autogenerated/ros/install" that contains ROS custom interfaces (similar to arbitrary C structs) header files. The main CmakeLists.txt file which automatically sets all the necessary environment variables to build ROS nodes without the ROS build system using a python script. The IDE also uses this CmakeLists.txt for the dependency resolution / code hints.
- python folder: one folder per python package and in this folder there is a requirements.txt file and a folder with the same name that contain the sources. The requirements.txt file is processed by the dependency manager tool to install the correct dependencies for the package in its virtual environment

We aimed to have clear separation between the sources, dependencies, binaries, config, logs... with the idea that it should be easy to find something and if it is not where it should be then it does not exist.

4.2 Dependency management

We need to have multiple versions of the same dependency across the codebase and ROS's dependency management system for python is not sufficient. There is no support for the `requirements.txt` file that is the standard to specify a package's dependencies. Instead one has to declare dependencies in an XML file that map these to apt packages names and the version that comes with the distribution is installed if a match is found.

Virtualenvs are designed to allow multiple versions of a dependency to coexist on a system and they have the benefit of not installing these dependencies system-wide but in a folder that is easily deletable. This is very useful because when deleting or upgrading dependencies, some files may be left over (especially when the python dependency compile a C backend during installation). There should not be any leftover files and if any, these files should not cause any issue but with virtualenvs it's so easy start from scratch that it's worth it when we want to guarantee consistent behavior.

Virtualenvs are built into Python so they usually are straightforward to use but ROS tools and nodes are supposed to be launched in a terminal in which a setup script has been executed to add or modify environment variables. After reading the setup scripts and comparing the environment before and after the execution of the scripts we found the necessary environment variables to launch ROS nodes without using ROS tools. Then we altered these variable further by adding the path of a virtual environment to the `PYTHONPATH` and `LD_LIB_PATH` variables and

we managed to launch a Python ROS node that could use the dependencies installed in this virtualenv ! Modifying the same variables allows to build C++ ROS nodes using CMake and launch them.

This paved the way to our current dependency management strategy where the `PYTHONPATH` and `LD_LIB_PATH` variables contain the path to ROS libraries, the node's virtual environment, the generated code for custom messages between nodes (similar to C structs), and a folder that contain dependencies we wrote ourselves that all nodes use one way or an other.

We specifically removed the system libraries from these variables because we don't want the host to have an influence on how the nodes run. We had an issue with a plotting library that was installed as a system dependency and in the virtual environment but the versions were not the same and they didn't have the same set of rendering backend installed. When listing the available backends the library found them all since they had different names but when trying to use them it crashed. This is the kind of bugs we would like to avoid.

For C++ nodes we currently only make the ROS custom interfaces package available to all the nodes. Everything we can do with CMake is possible so we will probably create a folder of libraries we use across many nodes and each complex package will have a resources folder containing its own libraries like the SDK used to control a sensor for example. We will see what is best when we start to write complex C++ nodes.

4.3 Compute resources management

We only really care about this on the production environment which is a Nvidia Jetson running Ubuntu 20.04, so we used cgroups which is something only available on Linux. Cgroups are a way to limit the amount of resources a process can consume or set priorities when there is contention. Unfortunately it is not possible to launch programs in a cgroup with a simple command like for example with `nice` because the cgroup has to exist before it is possible to launch a program in it and only root can modify cgroups. The best solution we found to solve this is to launch the nodes through systemd user defined slice files which implement resources limits using cgroups. These files must be placed in `~/.config/systemd/user`

The launch system can generate all the systemd files during a pre-launch phase and then launch each node that should be in a cgroup via systemd.

4.4 Launch system

The launch system loads the nodes configuration file to know which nodes to launch then sets the environment variables for each node and finally launch them, in cgroups if necessary. We

did not include any dependency graph to launch the nodes in a particular order because we do not think it is necessary when the design is sound. Currently the nodes only output data after having processed all the data they need as input so no motor or steering command should happen before all the nodes are initialized. The launch system also take care of setting the right environment variables for each node.

If one node stops the launcher stops all the nodes because they each are essential for the car to drive correctly. The car could go straight into a wall with even one node failing because the motors output the specified torque until a new torque command comes in. Due to this behavior the node that control the motors attempts to stop them before shutting down. Currently the worst we could expect if the RC car we use for tests has problems is material damage as it is relatively small. The large and much more powerful car used in the competitions has additional safety mechanisms controlled by a human that is always nearby.

4.5 Configuration files

The configuration files are JSON documents because Python's standard library has a parser for it, it can express many kinds of data (list, map, string, int, float, boolean) and it is easy to edit for a human. The lack of support for comments inside a JSON file is regrettable but we found that a readme.md file in the configuration files' folder is better than comments for explaining the different configuration options. There can be many configuration files that we may wish to load for different scenarios and duplicating the comments in each file would be messy.

There is a main configuration file that is used to specify the path of the nodes configuration file and to alter the behavior of each nodes with for example settings for the framerate, resolution and exposure time for the camera node.

The nodes configuration file specify which node to launch with which resources limits, if any. It has a section for Python nodes and a section for C++ nodes because the launcher treat them differently. This file could have been included in the main one but we want to be able to easily switch between resources constraints modes and execution scenarios (test run, simulation, development) without having a huge configuration file.

4.6 Containers

We considered using containers to solve the issues we were facing to get ROS to work on Ubuntu 18.04 by installing a Ubuntu 20.04 container on the Jetson. We eventually decided against it because we don't need isolation (the opposite actually because our code calls CUDA system libraries to access the GPU), more marshalling may happen if everything doesn't run in the same container and the development & build system would me more complex.

Chapter 5

Implementation

We kept the explanation relatively high level with some specific points of interest to not paraphrase the code as it is available and should be understandable.

5.1 Dependency management

The dependency manager tool is implemented as a Python script.

The script first generate the headers and necessary shared libraries of the ROS custom interfaces (C-like structs for when ROS standard types are not enough) that all nodes can use so this needs to happen before the C++ build step. To generate these custom interfaces, the script calls `colcon` which is the ROS build tool with specific instruction on where to build and place the resulting files to respect the organization of the code and dependencies. As for all ROS nodes and tools, colcon needs ROS-specific environment variables to run so we created a function with the lowest common denominator of all these variables and we add as needed. For colcon we had to add the system python packages to the `PYTHONPATH` environment variable because it calls an other tool, catkin that has dependencies there.

Then the script build the C++ ROS nodes by calling CMake with the main CMakeLists.txt in the "src/cpp" folder. Cmake can modify its own environment variables so this CMakeLists.txt calls an other python script that output the correct environment variables so that the nodes can be built. This looks more complicated than necessary but our C++ IDE load the CMakeLists.txt when opening the project so this also makes the IDE load all the correct environment variables too for the code hints and ability to build & launch ROS nodes directly in the IDE. For some reason we had to manually set the python interpreter to be the system one because the Find_python function would select one of the interpreters in a virtual environment of the project randomly. It worked anyway but was weird so we fixed this setting to the system interpreter.

Finally the script iterate over the nodes that should be launched based on the currently active configuration and install the python dependencies for all these nodes. To do so it creates the virtual environment if it doesn't already exists and then calls `pip3` with the requirements.txt of the node's package.

5.2 Launch system

The launch system is a python script that iterate over all the nodes in the active configuration file and launch them. For each node the script use a function to get the node-specific environment variables (since nodes can be in different virtual environment), entry point and cgroups configuration and uses that to launch them.

During initialization, the script delete all the previous cgroup configuration files in systemd if they exist and the ones that are needed during this run will be recreated when the node iterate over the nodes to launch. Since the cost to create and delete these files is negligible and that it happens only once we found this to be the best solution.

The script differentiate between C++ and Python nodes because they have to be launched differently (entry point instead of python3 + entry point) and also between nodes that should be launched in a cgroup and the unrestricted ones because the environment variables have to be set using `systemd-run` instead of via Python's subprocess tool in this case.

The output of each process is captured and displayed on the console asynchronously with each line prefixed by `[<language>-<package>-<entrypoint>]` which give for example `[py-vision-Vision_Module.py] YOLOv5 2022-6-3 Python-3.8.10 torch-1.11.0+cu102 CPU`

A longer example output is available in appendix (the `py-control-stanley` node is launched in a cgroup): A - Launcher output.

If we send a sigint or sigterm signal to the launcher process it repeats this signal to all the launched processes so that the can terminate cleanly. It also sends a sigterm to all the processes if one of them terminate as explained in the design section.

5.3 Monolith split

This split happened in two steps. During the first half of the semester we nicely packaged all the code into python modules which would become nodes later on. We managed to have a relatively simple dataflow between the modules which is data comes in (possibly from multiple other modules) then compute and finally data comes out with no back and forth.

When this was done we forked the pipeline and focused on meeting all the requirements we set before. By the very end of the semester all the requirements were met and we worked on merging our change with the changes of the rest of the team which proved surprisingly difficult because the code had become quite more complex. For example modifications were made to the pipeline to run it in a simulator and to do so the pipeline was passing a simulator object that some module were using to return fake data when running in the simulation. This didn't fit our model at all and we had to delete this code which will come back in the form of a simulator node connected to the simulator that will send data and receive control inputs (steering and torque). We faced a similar situation with the part of the pipeline that saved data from the modules to plot what was happening.

We currently have packages for the camera, control algorithms (multiples implementations of the same functionality), output communication (only to an Arduino to control the wheels and steering for now), localization (multiple implementations of the same functionality), odometry (inertial sensor), video file input (replace the camera in simulations), and vision (detect cones in an image).

5.4 Jetson setup

We need a computer to run the code on the car and we chose a Nvidia Jetson Xavier AGX development kit because of its fairly powerful eight cores CPU and powerful GPU in small form factor. When we started this project Ubuntu 18.04 was the most recent OS available on the jetson. At this point Nvidia had been working on porting Ubuntu 20.04 for almost two years there was no clear release date. Since we started from scratch with the port of the current pipeline to ROS we wanted to use ROS2 to benefit from the improvements and because ROS1 is reaching its end of life but all the supported ROS2 release need an OS more recent than Ubuntu 18.04. We were stuck between a rock and a hard place basically.

We had the option of forcing the Jetson to update to Ubuntu 20.04 which was reported to work but it would probably have broken the IO ports we wanted to use or compiling ROS2 on Ubuntu 18.04 which was also reported to work. We chose to compile ROS on a Ubuntu VM to see how it would go and we only had to manually install a recent version of CMake to compile everything. There was lots of warnings about the unsupported Ubuntu release we were using but the test program from ROS' documentation ran without apparent issues.

When we were done the team was starting to be limited by the ancient version of python (3.6) on Ubuntu 18.04. We could have installed Python 3.8 to match the version of Ubuntu 20.04 but Nvidia had released a developer preview of the Ubuntu 20.04 image for the Jetson so we decided to give it a try. It seemed to work fine and had the latest version of CUDA that our ML libraries use.

To validate the developer preview was usable we wanted to run the pipeline on it unchanged

and for that we had to compile from source an old version of pytorch and torchvision because the version - OS combination wasn't available as a pre-built binary. The compilation is pretty long (3-4h) and we got it right on the second try after editing the source. After installing torch we ran the pipeline and it worked just as well as before so we decided to keep using the developer preview. Soon after that we updated the library we use for vision (Yolov5) to the latest release to switch to pre-built standard binaries for Pytorch and Torchvision

Since we were running a developer preview we wanted to be able to reinstall it quickly should something bad happen so we documented the steps to setup the Jetson after a factory reset (how to build the camera driver, install ROS and a few other things) and made a script. It turned out to be useful when a big update to a point release bricked the Jetson. When this happened we improved the script if it happens again we may end up with a fully automatic script !

5.5 Fast multidimensional array transfer

Most of the communication that happens between the nodes uses multidimensional numpy arrays so we need these transfers to be as efficient as possible. Fortunately ROS messages can be multidimensional arrays but we must be very careful to not create an intermediate python representation of the array. The first library we found to convert from numpy arrays to ROS' multiarrays did that and the performance was terrible. It could only process about 10 frames per second of the camera node with one core pinned to 100% utilization.

It was still a good first step because it got the dimension translation right between the numpy and multiarray representation. By looking at the documentation we found a way to directly copy the raw underlying C buffer from one representation to the other. We just have to specify the buffer's endianness when going from numpy to multiarray because numpy's arrays can have both endianness (on a machine it should always be same but we never know) and multiarray are always little endian based on anecdata. We couldn't not find answer about that in the documentation, there were comments on forums telling it was little endian and no flag in the datastructure indicating endianness. As long as we only communicate locally it should be fine and if we ever switch to a multi host setup we'll look into this again.

The last step was to create function to automatically map numpy array types (uint8, float32, int32...) to multiarrays type and vice versa. It's just a big if, else if on the type and while it's not very good code it makes the API very pleasant to use with only two function functions that are `to_numpy(arr: Multiarray)` and `to_multiarray(arr: np.ndarray)`.

5.6 IDE configuration

This section is made for PyCharm and CLion because this is what we use in the team but it should be applicable to any other IDE.

5.6.1 Pycharm

This assumes you have already cloned the repository into PyCharm. When opening the project PyCharm should automatically create a virtualenv named `venv`. You have to manually merge all the requirements.txt files of the packages into one and then installing them in the virtual environment so that code hints work. When there is multiple version of the same dependency it's preferable to take the most recent one. It's possible to make PyCharm use a package's virtual environment for code hints when needed too.

Then you need to launch `src/launch_and_build/shared.py` and update all your run configurations (saved in git in .idea/runConfigurations) with the environment variables given.

Once the dependencies are installed in the virtual environment you have to go to File -> settings -> Python Interpreter -> gear icon -> click on "show paths for the selected interpreter" -> add the three paths presents in the pythonpath of the generic section of outputed by the `shared.py` tool above. These three paths are the ROS system libraries, the shared packages in the src folder and the custom interfaces in the autogenerated folder.

5.6.2 CLion

To use CLion you need to clone the repository using an other tool (which can be PyCharm) and then create a new project from existing sources by selecting the src/cpp folder in CLion. This is very important because each IDE will create a .idea folder at the root of their project (so .idea for PyCharm and src/cpp/.idea for CLion) and terrible things happen if you open a PyCharm project in CLion and vice versa.

CLion then configures itself automatically using the main CMakeLists.txt in the cpp folder.

Nodes have to be launched with the generic minimal environment variables (set in launch configuration)

Chapter 6

Evaluation

In this section we'll go over the requirements one by one and discuss whether we met them.

6.1 Publish - Subscribe performance

We measured the performance of the publish subscribe performance of ROS using two C++ node that were sending to each other a timestamp and some fake data. ROS has many settings to influence the behavior of the publish subscribe mechanism and we chose the one that optimized the latency by discarding messages if the previous message wasn't handled yet during reception.

When sending every $200\mu s$ the mean between the time of emission and reception is 73'226 nanoseconds with a standard deviation of 36'524 nanoseconds.

Unfortunately we didn't have the time to conduct more in depth experiments for the report but this seems to indicate that in memory message is used and the performance is very good, for small messages at least.

6.2 Development experience

We only tested this with PyCharm and CLion because this what we use in the team and with these IDE, the development experience is very nice once the initial setup is done. To setup the project one has to clone the project, then run the dependency manager tool, update the runtime configuration of each node (saved in git) with the environment variables that one can get by running the `get_env_vars.py` tool then finally manually install all the dependencies in the `venv` virtual environment and add a few path to the pythonpath as explained in the Implementation section. It could be easier but it's already good enough. It is certainly possible to configure

VsCode or any other advanced IDE similarly.

Developing just feels like working on a normal Python project in PyCharm and the same can be said for CLion. The ability to easily disable one of the node in the nodes' configuration file to manually launch it the IDE is very useful when debugging.

While what we currently have is very nice to use and tailored to our needs, we should always carefully consider the cost of rolling our own solutions versus the cost of making something that already exist work for our use case or adapting the way we work to fit the tool's model. If we didn't need multiple versions of the same dependencies, developing the ROS way and using ROS tools may have been simpler. There exist some documentation and plugins to develop in ROS with PyCharm, CLion and VsCode.

In this case we think we made the right choice because ROS in its default form doesn't fit our requirements but it still took a long time to develop and iron out.

6.3 Runtime platforms requirements

The project runs well on the platforms specified in the requirements but during the semester the requirements changed. We now also need to run a simulator [1] that need a GPU on the computers we use to develop to validate correctness of the algorithms and the system as a whole. Due to the GPU requirement we can't run it in a VM and it must run on the host which makes communication harder between the simulator and the VM.

Before switching to ROS the pipeline could run almost anywhere from Windows to M1 macs but now we need to use Ubuntu 20.04 which is a lot more restrictive.

6.4 Dependency Management

We managed to achieve the main goal which was to have multiple versions of the same dependencies across the codebase for Python nodes. We also managed to make everything work with virtual environments which allows us to not pollute the OS of the Jetson and the computers used for the development with dependencies of project and not have dependencies installed in OS influence the execution of the nodes. It took a lot time and effort to achieve but we think it was worth it.

The dependencies that are not available on PyPi (custom pytorch, torchvision and the camera python bindings) currently have to be installed manually in the correct virtual environment. The PyTorch situation is especially complex because depending on the computer we need to install the CPU or one of the CUDA version and on the Jetson a pre-built binary made by Nvidia

that is not on PyPi. We think this is the point where having a dependency manager that can do everything hits diminishing returns because while it's very interesting to do, it takes time and makes the codebase more complex for a task that we don't need to do often.

6.5 Launch system and node runtime management

This part does its job well and is "done" in our opinion. The only addition we could think of is for it to catch sigstop and propagate it to other nodes to help debugging by stopping all the nodes, when the node we're debugging hits a breakpoint.

6.6 Up and running time and difficulty from factory reset

The program to setup the Jetson (Nvidia SDK Manager) needs to download ~14GB from the internet to factory reset the Jetson which can be problematic during competitions abroad for example so the files should be cached on a computer. The setup is also pretty long because while the base image is copied as is on the internal disk, all the libraries like cuda are installed as regular packages.

Once the initial setup is done we need to do own setup which also takes time. The last time we had to do it it took us 6h due to the custom PyTorch we needed to compile and next time 1h30 is a realistic goal with the improved setup script and the standard PyTorch binaries we can download from Nvidia.

6.7 ROS lock-in

The current architecture only uses ROS as a library for inter node communication so the vendor lock-in is minimal. We will also probably keep an architecture centered around nodes and asynchronous communication due the advantages it brings moving forward so the work of creating a dependency manager, build system and launch system should not be lost.

Chapter 7

Conclusion

The project met almost all the requirements set in the beginning and the ones we discovered during the semester with the exception of the automatic dependency management. As explained in the Evaluation section it largely works but some special Python packages have to be installed manually.

Even though ROS was difficult to adapt to our use case we think it was the right choice as it allows us to mix C++ and Python easily to incrementally rewrite the autonomous driving system in C++, has large collection of algorithms and drivers and allows us to run multiple algorithms on same or different sets of data at different frequencies easily with for example our vision code to detect cones and SVO-PRO [4] for odometry that both use the camera.

The desire to transition to a microservice or module based architecture that started this project is also paying off. The control module is now in its own git repository, we pull it using git submodules into the package and the ROS node is just a thin wrapper around it. This allows the people working on this part of the autonomous driving system to focus on it and test it using tailored simulations for example. As each node gains in maturity we expect to continue in this direction.

Chapter 8

Further work

When working on this project we identified areas of improvements that were either out of scope or too time consuming to implement so we list them here:

- ROS is not clear on how the communication between nodes actually happen as it is handled by a library called `eProsima Fast DDS` by default. In memory messaging would be great to avoid marshalling and there exist ways to dictate how the library that ROS uses actually communicate. For example here [3] to force UDP communication
- We need a way to implement backpressure or at least log when messages (publish / subscribe) cannot be processed in time to receive the next one
- We need a VM / server with a Nvidia GPU to run simulations and train new machine learning models. The team develop on a variety of hardware and OS with for example M1 macs, Windows and Linux
- Most dependencies are currently the same across nodes. Do we waste disk space and put significantly more pressure on the L1I cache ? With the current architecture it is easy to create a shared virtual environment that contain the main dependencies
- We need to be careful when limiting the resources available to a node. Some software detect the number of cores and spin the optimal number of threads based on this so if we limit the execution to a fraction of the cores more context switches happen for no useful work
- Sending a sigstop signal to the other nodes when a node that we debug hit a breakpoint would be great. Currently the other nodes continue to run which mean their state has changed when we resume the execution of the node
- A way to automatically install local dependencies into specific virtual environments. This could be useful for PyTorch, TorchVision and the camera python bindings

- Currently the Arduino that control the motors has no concept of heartbeat. We should add that to shutdown the motors if it loses contact with the Jetson
- It would be great if the dependency manager tool automatically installed the most recent version of each dependency in the IDE's virtual environment (venv folder in project root) on each run. We currently have to install and update these dependencies manually to benefit from code hints.

Appendices

A - Launcher output

```
[py-control-stanley.py] Running as unit: run-u2609.service
[py-control-stanley.py] Press ^] three times within 1s to disconnect
   TTY.
[py-external_com-arduino.py] 1654805483.534329 [0]     python3:
   selected interface "lo" is not multicast-capable: disabling
   multicast
[py-odometry-motion_sensor.py] 1654805483.822189 [0]     python3:
   selected interface "lo" is not multicast-capable: disabling
   multicast
[py-control-stanley.py] 1654805484.258926 [0]     python3: selected
   interface "lo" is not multicast-capable: disabling multicast
[py-video_file_input-main.py] 1654805484.308935 [0]     python3:
   selected interface "lo" is not multicast-capable: disabling
   multicast
[py-video_file_input-main.py] 10.00000159010626
[py-video_file_input-main.py] 565.0
[py-video_file_input-main.py] 564.0
[py-localization-ExtendedKalmanFilter.py] 1654805484.635424 [0]
   python3: selected interface "lo" is not multicast-capable:
   disabling multicast
[py-video_file_input-main.py] 563.0
[py-video_file_input-main.py] 562.0
[py-video_file_input-main.py] 561.0
[py-video_file_input-main.py] 560.0
[py-video_file_input-main.py] 559.0
[py-video_file_input-main.py] 558.0
[py-video_file_input-main.py] 557.0
[py-video_file_input-main.py] 556.0
[py-video_file_input-main.py] 555.0
```

```
[py-video_file_input-main.py] 554.0
[py-vision-Vision_Module.py] Loading config specified by
   config_selector file
[py-vision-Vision_Module.py] Loading config specified by
   config_selector file
[py-vision-Vision_Module.py] Loading config specified by
   config_selector file
[py-vision-Vision_Module.py] 1654805485.635148 [0]    python3:
   selected interface "lo" is not multicast-capable: disabling
   multicast
[py-control-stanley.py] [ 6.7559695  -0.17664948]
[py-external_com-arduino.py] 1550 102
[py-localization-ExtendedKalmanFilter.py] [-0.0113202 -4.861501
   1.5720906  1.         0.        ]
[py-video_file_input-main.py] 553.0
[py-video_file_input-main.py] 552.0
[py-video_file_input-main.py] 551.0
[py-video_file_input-main.py] 550.0
[py-video_file_input-main.py] 549.0
[py-vision-Vision_Module.py]
[py-vision-Vision_Module.py] YOLOv5         2022-6-3 Python-3.8.10
   torch-1.11.0+cu102 CPU
[py-vision-Vision_Module.py]
[py-vision-Vision_Module.py] Fusing layers...
[py-vision-Vision_Module.py] YOLOv5n summary: 213 layers, 1772695
   parameters, 0 gradients
[py-control-stanley.py] [ 6.4951663  -0.02434443]
[py-control-stanley.py] [ 6.1415415  -0.03151534]
[py-control-stanley.py] [6.1436715e+00 3.0061312e-04]
[py-external_com-arduino.py] 1550 93
[py-external_com-arduino.py] 1550 93
[py-external_com-arduino.py] 1550 91
```

Bibliography

[1] "Formula Student Driverless Simulator". In: URL: https://fs-driverless.github.io/Formula-Student-Driverless-Simulator.

[2] "Marshalling (computer science)". In: URL: https://en.wikipedia.org/wiki/Marshalling_(computer_science).

[3] "ROS2 Foxy Communication across multiple local users". In: URL: https://github.com/eProsima/Fast-DDS/issues/1750.

[4] "SVO Pro". In: URL: https://github.com/uzh-rpg/rpg_svo_pro_open.

[5] "Thesis Template". In: URL: https://github.com/HexHive/thesis_template.