Duckietown Developer

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This book is about how to become a developer in Duckietown. It will guide you through the basics and tools needed for effective software development in Duckietown.

Contents

0.1. What to look for throughout the book

At the end of every section of this book, you will find the subsections:

- Hands on;
- Ask the community;

They will give you some exercises/activities to assess/improve your knowledge and/or prepare your environment for the next section and information about how and where to ask the community for help with that specific section.

Ready? Let’s start!

Part A
Developer Basics

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This section of the book will introduce the basics of software development in Duckietown. The arguments presented in this section are very general and should be clear to any developer (not just in Duckietown).

Contents

UNIT A-1

Developer Basics: ISO/IEC 9126

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

Contents

ISO/IEC 9126 is a international standard for product quality in Software Engineering. It was officially replaced by the new ISO/IEC 25010 in 2011 that introduces a few minor changes.

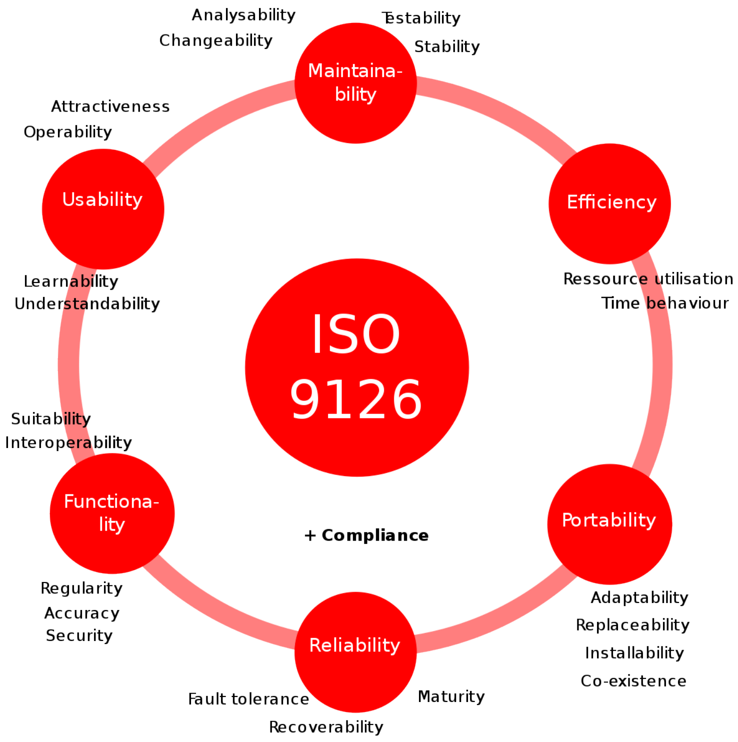


Figure 1.1. ISO/IEC 9126 Standard (source: Wikipedia)

Software development, as any other activities carried out by human beings is subject to human biases. The ISO/IEC 9126 standard's objective is that of aknowledging the most common biases and addressing them by definining clear guidelines about what properties a **good** software product should have.

In this section, we are not going to dive into this standard, but we strongly believe that such standard (and its successor ISO/IEC 25010) should be the best friend of any developer.

Throughout this book, we will mention some of these qualities as we motivate some of the decisions made while creating the Duckietown Development Workflow.

1.1. Hands on

We suggest the reader to get familiar with such standard by using these resources:

- Wikipedia - ISO/IEC 9126
- Official ISO/IEC 9126 (by ISO.org)
- Official ISO/IEC 25010 (by ISO.org)

1.2. Ask the community

If you have any questions about good practices in software development, join the Slack channel #info-developers.

UNIT A-2

Developer Basics: Linux

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This section of the book will introduce Linux distributions and specifically the Ubuntu distribution. We will provide guides for installing Ubuntu in *dual-boot* mode or inside a *virtual machine*.

Contents

2.1. Linux

Linux is a group of free and open-source software operating systems built around the Linux kernel first released in 1991. Typically, Linux is packaged in a form known as a Linux distribution such as Fedora or Ubuntu.

Ubuntu is the Linux distribution officially supported by the Duckietown community.

2.2. Ubuntu

As of this writing, the most recent version of Ubuntu is 20.04 LTS (Long Term Service) which will be supported until April 2025.

2.3. Installation

It is highly recommended to install Ubuntu directly on your laptop or as a dual boot operating system alongside your existing OS. However we also provide some guidance on installing Ubuntu within a Virtual Environment on your laptop.

1) Dual Boot

- First you need to download a `.iso` image file which contains the version of Ubuntu you want. Here is 20.04 LTS make sure to download the desktop image.
- Next, you need a free USB drive with at least 2GB of space. The drive will be completely written over.
- You need some software to write the .iso to the USB. If on Windows you can use Ru-

fus

- Create the bootable USB drive, disconnect the USB then reconnect to your computer.
- Restart your computer
  - If your computer simply boots into the existing operating system you need to change the boot order in your BIOS.
  - Restart your computer again and press the button during startup which lets you into the BIOS. It may say on your computer what this button is but you may need to Google depending on your laptop model. For example Lenovo might be F1 or F2.
  - Look for an option to change boot order and put priority on your USB drive.
- Your computer should now boot into Ubuntu installation and you can follow the instructions for dual boot.

2) Virtual Machine

- First you need to download a .iso image file which contains the version of Ubuntu you want. Here is 20.04 LTS make sure to download the desktop image.
- Download your desired Virtual Machine platform (popular choices are Virtual Box and VMWare).

> Note: Using a Virtual Machine might require some particular settings for you networking settings. The virtual machine should appear as a device on your local network. For example, in VirtualBox, you need to set up a *Bridged Network*. This might differ in other hypervisors.

2.4. Terminal

Some pointers:

- Open a terminal with Ctrl + Alt + T
- `/` is the top level root directoy which contains your
- `~` refers to your home folder located in `/home/` *username*

2.5. Hands on

We suggest that you install a Linux distribution on your computer and get familiar with it before proceeding to the next sections.

2.6. Ask the community

If you have any questions about good practices in installing Ubuntu on your computer or other questions about Ubuntu, join the Slack channel #help-laptop.

UNIT A-3

Developer Basics: Git

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

Contents

Every time there is a large project, with many contributors and the need for code versioning and history, developers rely on VCS (Version Control Systems) tools. Duckietown uses Git as VCS and GitHub.com as a service provider for it. The Duckietown organization page on GitHub is github.com/duckietown.

3.1. Monolithicity VS Modularity

Whether a software project should be monolithic or modular is one of the most debated decisions that a group of developers faces at the beginning of a software project. Books have been written about it. Duckietown started as a monolithic project, and some of us still remember the infamous Software repository, and only later transitioned to a full modular approach.

There are two levels of modularity in Duckietown. We distinguish between **Modules** and **Nodes**. Modules form our first and highest level of modularity, with each module being a collection of nodes. Nodes constitute the smallest software entities, and each node is usually responsible for a very specific task. Nodes are not allowed to exist outside modules. We will revisit these concepts later in the book, but in a nutshell, modules represent high level concepts, like *autonomous driving capability* for a vehicle, while nodes within a module tackle more granular tasks, like *traffic signs detection*.

In Duckietown, code is separated so that each module has its own repository. All official repositories are hosted under the same GitHub organization github.com/duckietown. Be brave, (as of April 2020) we have more than 220 repositories there. You can also have your own modules hosted as repositories on your own GitHub account.

3.2. Git

This section goes through the most common operations you can perform on a git project and a git project hosted on GitHub.com.

1) Terminology

A non-exhaustive list of terms commonly used in git follow.

Repository:

A repo (short for repository), or git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history.

Branch:

Branches constitute threads of changes within a repository. Though we call them branches, do not try to force an analogy with tree branches, they are similar in spirit but quite different in how they work.

A repository is not allowed to exist without a branch, and every operation inside a repository only makes sense in the context of a branch (the *active* branch). Inside a repository, you can have as many branches as you want, but you always work on one branch at a time. Every git project has at least one main branch, usually called the `master` branch.

Use the command `git branch` to see the list of branches present in your repository and which branch you are currently working on.

Though, branches are used in different scenarios, they simply allow groups of developpers to work on their own task without having their work affect or be affected by other groups' work. For example, after a project is released with version `1.0.0`, one team is tasked to develop a new feature for the version `1.1.0` milestone while another team is asked to fix a bug that a user reported and whose patch will be released in the version `1.0.1`.

Branch operations are performed through the command `git branch`.

Commit:

A commit is an atomic change in a repository. A commit is a set of changes to one or more files within your repository. Each commit is uniquely identified within a repository by the hash (SHA-1) of the changes it contains ("plus" a header).

When you create/delete/edit one or more files in a git repository and you are confident enough about those changes, you can commit them using the command `git commit`.

> Note: A commit is not a snapshot (or a copy) of the entire repository at a given point in time. Each commit contains only the incremental difference from the previous commit, called *delta* in git.

A chain of commits in which all the ancestors are included makes a branch. Since every commit is linked to its parent, a branch is simply a *pointer* to a commit (the full chain of commits can always be reconstructed from the commit). In other words, you can think of branches as human friendly labels for commits. Every time you create a new commit, the pointer of the current branch advances to the newly created commit.

Tag:

A tag is a human friendly name for a commit but unlike branches, tags are read-only. Once created, they cannot be modified to point to a different commit.

Tags are commonly used for labeling commits that constitute milestones in the project development timeline, for example a release.

Fork:

A fork is basically a copy of someone else's repository. Usually, you cannot create branches or change code in other people's repositories, that's why you create your own copy of it. This is called `forking`.

Remote:

A git *remote* is a copy of your repository hosted by a git service provider, e.g. GitHub. Remotes allow you to share your commits and branches so that other developers can fetch them. Technically speaking, remotes are exactly the same as local repositories, but unlike your local repository, they are reachable over the internet.

You can use the commands `git fetch` and `git push` to bring your local copy of the repository in sync with a remote, by downloading commits or uploading new commits respectively.

Merging branches:

Merging is the dual operation of creating a new branch. Imaigne you have branched out a new branch (e.g. *new-feature*) from the some branch (e.g. *master*), made some improvements and tested them out. Now you want to incorporate these changes in the *master* branch which hosts your main code. The **merge** operation does exactly that. It takes the changes done in *new-feature* and applies them to *master*.

Often git will manage to apply these changes by itself. However, sometimes if both *new-feature* and *master* changed the same part of the code, git cannot determine by itself which of the two changes should be kept. Such a case is called *merge conflict* and you will have to manually select what should be kept after the merge.

Pull Requests:

If you are working on a secondary branch or if you forked a repository and want to submit your changes for integration into the mainstream branch or repository, you can open a so-called Pull Request (in short **PR**).

A pull request can be seen as a three-step merge operation between two branches where the changes are first *proposed*, then *discussed and adapted* (if requested), and finally *merged*.

2) Common operations

Fork a repository on GitHub:

To fork (creating a copy of a repository, that does not belong to you), you simply have to go to the repository's webpage and click fork on the upper right corner.

Clone a repository:

Cloning a repository is the act of creating a local copy of a remote repository. A repo is cloned only at the very beginning, when you still don't have a local copy of it.

To clone a repository, either copy the HTTPS or SSH link given on the repository's webpage. Use the following command to create a local copy of the remote git repository identified by the given URL.

```
$ git clone REPOSITORY-URL
```

This will create a directory in the current working path with the same name of the repository and the entire history of commits will be downloaded onto your computer.

Create a new branch:

The command for creating a new branch is a little bit counter-intuitive, but you will get use to it. Use the following command to create a new branch:

```
$ git checkout -b NEW-BRANCH-NAME
```

This creates a new branch pointing at the same commit your currently active branch is pointing at. In other words, you will end up with two branches pointing at the same commit. Note that after you issue this command, the newly created branch becomes your active branch.

Working tree:

In git, we use the term *working tree* to indicate all the changes that are not committed yet. You can think of it as your workspace. When you create a new commit, the hash for the current working tree is computed and assigned to the new commit together with the changes since the last commit. The working tree clears as you commit changes.

> **Remark:** You cannot create commits from a clean working tree.

Use the command `git status` to inspect the status of your working tree.

Create a new commit:

Unlike many git operations, a commit is not created by a single git command. There are two steps to follow. First, we mark all the changes that we want to be part of our new commit, second, we create the commit. From your working tree, mark changes to include in the new commit using the command:

```
$ git add FILE
```

The command `git status` will always show you which changes are marked to be used for a new commit and which changes are not. Use the command

```
$ git commit -m "COMMIT-MESSAGE"
```

to create a new commit. Replace *COMMIT-MESSAGE* with your notes about what changes this commit includes.

> Note: Do not underestimate the value of good commit messages, the moment you will go back to your history of commits looking for a change of interest, having good commit messages will be a game changer.

Push changes:

Use the following command to *push* your local changes to the remote repository so that the two repositories can get in sync.

```
$ git push origin BRANCH-NAME
```

3) Fetch changes

If you suspect that new changes might be available on the remote repository, you can use the command

```
$ git fetch origin BRANCH-NAME
```

to download the new commits available on the remote (if any). These new changes will be appended to the branch called `origin/BRANCH-NAME` in your local repository. If you want to apply them to your current branch, use the command

```
$ git merge origin/BRANCH-NAME
```

Use the command `git pull origin/BRANCH-NAME` to perform *fetch* and then *merge*.

Delete branches:

Unlike the vast majority of git commands, git delete does not work on the current branch. You can delete other branches by running the command

```
$ git branch -d BRANCH-NAME
```

If you want to delete your current branch, you will need to checkout another branch first. This prevents ending up with a repository with no branches.

To propagate the deletion of a branch to the remote repository, run the command:

```
$ git push origin --delete BRANCH-NAME
```

Open a GitHub Issue:

If you are experiencing issues with any code or content of a repository (such as this operating manual you are reading right now), you can submit issues. For doing so go to the dashboard of the corresponding repository and press the `Issues` tab where you can open a new request.

For example you encounter a bug or a mistake in this operating manual, please visit this repository's Issues page to report an issue.

GitHub Issues are a crucial part of the life cycle of a software product, as they constitute a feedback loop that goes directly from the end-user to the product developers. You don't have to be a developer or an expert in software engineering to open an Issue.

3.3. Hands on

1) Git

It is strongly suggested to all git beginners to follow the awesome tutorial Learn Git Branching.

Further reading material can be found at the following links:

- Git Handbook
- Basic Branching and Merging

2) GitHub

You can gain access to GitHub by creating an account on github.com (if you don't have one already).

A short GitHub tutorial is available at this link.

It is higly suggested that you setup an SSH key for secure passwordless access to GitHub by following these steps:

1. Generate a new SSH key
2. Add SSH key to your GitHub account.

3.4. Ask the community

If you have any questions about how to use of Git in Duckietown, join the Slack channel #help-git.

UNIT A-4
Developer Basics: Docker

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This section will introduce Docker and the features of Docker that the Duckietown community employs. For a more general introduction to Docker, we suggest reading the official Docker overview page.

Contents

4.1. What is Docker?

Docker is used to perform operating-system-level virtualization, something often referred to as "containerization". While Docker is not the only software that does this, it is by far the most popular one.

Containerization refers to an operating system paradigm in which the kernel allows the existence of multiple isolated user space instances called containers. These containers may look like real computers from the point of view of programs running in them.

A computer program running on an ordinary operating system can see all resources available to the system, e.g. network devices, CPU, RAM; However, programs running inside of a container can only see the container's resources. Resources assigned to the container become thus available to all processes that live inside that container.

4.2. Containers VS. Virtual Machine

Containers are often compared to virtual machines (VMs). The main difference is that VMs require a host operating system (OS) with a hypervisor and a number of guest OSs, each with their own libraries and application code. This can result in a significant overhead.

Imagine running a simple Ubuntu server in a VM on Ubuntu: you will have most of the kernel libraries and binaries twice and a lot of the processes will be duplicated on the host and on the guest OS. Containerization, on the other hand, leverages the existing kernel and OS and adds only the additional binaries, libraries and code necessary to run a given application. See the illustration bellow.

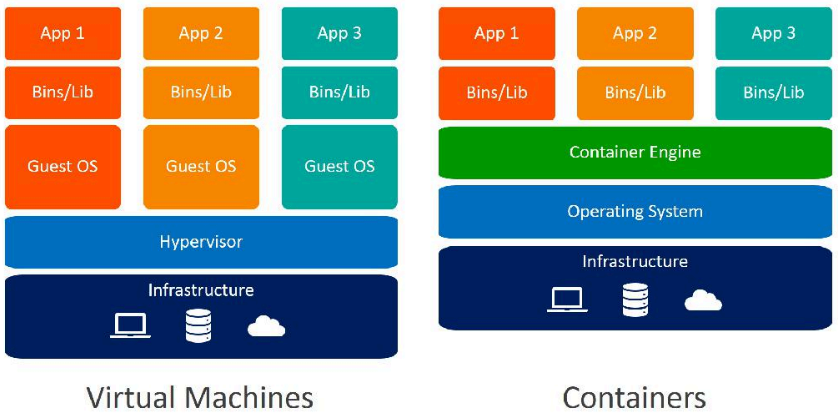


Figure 4.1. Differences between Virtual Machines and Containers (source: Weaveworks)

Because containers don't need a separate OS to run they are much more lightweight than VMs. This makes them perfect to use in cases where one needs to deploy a lot of independent services on the same hardware or to deploy on not-especially powerful platforms, such as Raspberry Pi - the platform the Duckietown community uses.

Containers allow for reuse of resources and code, but are also very easy to work with in the context of version control. If one uses a VM, they would need to get into the VM and update all the code they are using there. With a Docker container, the same process is as easy as pulling the container image again.

4.3. How does Docker work?

You can think that Docker containers are build from Docker images which in turn are build up of Docker layers. So what are these?

Docker images are build-time artifacts while Docker containers are run-time constructs. That means that a Docker image is static, like a `.zip` or `.iso` file. A container is like a running VM instance: it starts from a static image but as you use it, files and configurations might change.

Docker images are build up from layers. The initial layer is the *base layer*, typically an official stripped-down version of an OS. For example, a lot of the Docker images we run in Duckietown have `ubuntu:18.04` as a base.

Each layer on top of the base layer constitutes a change to the layers below. The Docker internal mechanisms translate this sequence of changes to a file system that the container can then use. If one makes a small change to a file, then typically only a single layer will be changed and when Docker attempts to pull the new version, it will need to download and store only the changed layer, saving space, time and bandwidth.

In the Docker world, images get organized by their repository name, image name and tags. As with Git and GitHub, Docker images are stored in image registers. The most popular Docker register is called DockerHub and it is what we use in Duckietown.

An image stored on DockerHub has a name of the form:

```
[repository/]image[:tag]
```

The parts `repository` and `tag` are optional and they default to `library` (indicating Docker official images) and `latest` (special tag always pointing to the *latest* version of an image). For example, the Duckietown Docker image

```
duckietown/dt-core:daffy-arm32v7
```

has the repository name `duckietown`, the image name `dt-core`, and the tag `daffy-arm32v7`, which carries both the name of the Duckietown software distribution that the image contains, i.e., `daffy`, and the CPU architecture that this image is targeting, i.e., `arm32v7`. We will talk about different CPU architectures and why they need to be part of the Docker image tag in the section Section 4.4 - Different CPU architectures.

All Duckietown-related images are in the `duckietown` repository. Though images can be very different from each other and for various applications.

4.4. Different CPU architectures

Since Docker images contain binaries, they are not portable across different CPU architectures. In particular, binaries are executable files that are compiled to the level of CPU instructions. Different CPU architectures present different instructions sets.

Many modern computers use the `amd64` architecture, used by almost all modern Intel and AMD processors. This means that it is very likely that you can find a Docker image online and run it on your computer without having to worry about CPU architectures.

In Duckietown, we use low-end computers like the Raspberry Pi (officially used on any Duckietown device) and Nvidia Jetson. These low-cost computers employ Arm processors that are based on the `arm32v7` instructions set.

Note: Full disclosure, while all devices officially supported in Duckietown are based on 64-bit capable Arm processors, thus using the `arm64v8` instructions set, the Raspbian OS only supports 32-bit, which is the reason why we use `arm32v7` images.

4.5. Working with images

If you want to get a new image from a Docker registry (e.g. DockerHub), you have to *pull* it. For example, you can get an Ubuntu image by running the command:

```
$ docker pull ubuntu
```

According to Section 4.3 - How does Docker work?, this will pull the image with full name `library/ubuntu:latest` which, as of May 2020, corresponds to Ubuntu 20.04.

You will now be able to see the new image pulled by running:

```
$ docker image list
```

If you don't need it anymore or you are running out of storage space, you can remove an image with,

```
$ docker image rm ubuntu
```

You can also remove images by their `IMAGE ID` as printed by the `list` command above. A shortcut for `docker image rm` is `docker rmi`.

Sometimes you might have a lot of images you are not using anymore. You can easily remove them all with:

```
$ docker image prune
```

This will remove all images that are not supporting any container. In fact, you cannot remove images that are being used by one or more containers. To do so, you will have to remove those containers first.

If you want to look into the heart and soul of your images, you can use the commands `docker image history` and `docker image inspect` to get a detailed view.

4.6. Working with containers

Containers are the run-time equivalent of images. When you want to start a container, Docker picks up the image you specify, creates a file system from its layers, attaches all devices and directories you want, "boots" it up, sets up the environment up and starts a pre-determined process in this container. All that magic happens with you running a single command: `docker run`. You don't even need to have pulled the image beforehand, if Docker can't find it locally, it will look for it on DockerHub.

Here's a simple example:

```
$ docker run ubuntu
```

This will take the `ubuntu` image with `latest` tag and will start a container from it.

The above won't do much. In fact, the container will immediately exit as it has nothing to execute. When the main process of a container exits, the container exits as well. By default the `ubuntu` image runs `bash` and as you don't pass any commands to it, it exits immediately. This is no fun, though.

Let's try to keep this container alive for some time by using the `-it` flags. This tells Docker to create an interactive session.

```
$ docker run -it ubuntu
```

Now you should see something like:

```
root@73335ebd3355:/#
```

Keep in mind that the part after `@` will be different — that is your container ID.

In this manual, we will use the following icon to show that the command should be run in the container:

```
$ command to be run inside the container
```

You are now in your new `ubuntu` container! Try to play around, you can try to use some basic `bash` commands like `ls`, `cd`, `cat` to make sure that you are not in your host machine.

You can check which containers you are running using the `docker ps` command — analogous to the `ps` command. Open a new terminal window (don't close the other one yet) and type:

```
$ docker ps
```

An alternative (more explicit) syntax is

```
$ docker container list
```

These commands list all running containers.

Now you can go back to your `ubuntu` container and type `exit`. This will bring you back to you host shell and will stop the container. If you again run the `docker ps` command you will see nothing running. So does this mean that this container and all changes you might have made in it are gone? Not at all, `docker ps` and `docker container list` only list the *currently running* containers.

You can see all containers, including the stopped ones with:

```
$ docker container list -a
```

Here `-a` stands for *all*. You will see you have two `ubuntu` containers here. There are two containers because every time you use `docker run`, a new container is created. Note that their names seem strangely random. We could have added custom, more descriptive names—more on this later.

We don't really need these containers, so let's get rid of them:

```
$ docker container rm container name 1 container name 2
```

You need to put your container names after `rm`. Using the containr IDs instead is also possible. Note that if the container you are trying to remove is still running you will be asked to stop it first.

You might need to do some other operations with containers. For example, sometimes you want to start or stop an existing container. You can simply do that with:

```
$ docker container start container name
$ docker container stop container name
$ docker container restart container name
```

Imagine you are running a container in the background. The main process is running but you have no shell attached. How can you interact with the container? You can open a terminal in the container with:

```
$ docker attach container name
```

4.7. Running images

Often we will ask you to run containers with more sophisticated options than what we saw before. Look at the following example: (don't try to run this, it will not do much).

```
$ docker -H hostname.local run -dit --privileged --name joystick --
network=host -v /data:/data duckietown/rpi-duckiebot-joystick-de-
mo:master18
```

Table 4.1 shows a summary of the options we use most often in Duckietown. Below, we give some examples

TABLE 4.1. DOCKER RUN OPTIONS

Short command	Full command	Explanation
`-i`	`--interactive`	Keep STDIN open even if not attached, typically used together with `-t`.
`-t`	`--tty`	Allocate a pseudo-TTY, gives you terminal access to the container, typically used together with `-i`.
`-d`	`--detach`	Run container in background and print container ID.
	`--name`	Sets a name for the container. If you don't specify one, a random name will be generated.
`-v`	`--volume`	Bind mount a volume, exposes a folder on your host as a folder in your container. Be very careful when using this.
`-p`	`--publish`	Publish a container's port(s) to the host, necessary when you need a port to communicate with a program in your container.
`-d`	`--device`	Similar to `-v` but for devices. This grants the container access to a device you specify. Be very careful when using this.
	`--privileged`	Give extended privileges to this container. That includes access to **all** devices. Be **extremely** careful when using this.
	`--rm`	Automatically remove the container when it exits.
`-H`	`--hostname`	Specifies remote host name, for example when you want to execute the command on your Duckiebot, not on your computer.
	`--help`	Prints information about these and other options.

Examples

Set the container name to `joystick`:

```
--name joystick
```

Mount the host's path `/home/myuser/data` to `/data` inside the container:

```
-v /home/myuser/data:/data
```

Publish port 8080 in the container as 8082 on the host:

```
-p 8082:8080
```

Allow the container to use the device `/dev/mmcblk0`:

```
-d /dev/mmcblk0
```

Run a container on the Duckiebot:

```
-H duckiebot.local
```

4.8. Other useful commands

1) Pruning images

Sometimes your docker system will be clogged with images, containers, networks, etc. You can use `docker system prune` to clean it up.

```
$ docker system prune
```

Keep in mind that this command will delete **all** containers that are not currently running and **all** images not used by running containers. So be extremely careful when using it.

2) Portainer

Often, for simple operations and basic commands, one can use Portainer.

Portainer is itself a Docker container that allows you to control the Docker daemon through your web browser. You can install it by running:

```
$ docker volume create portainer_data
$ docker run -d -p 9000:9000 --name portainer --restart always -v
/var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data
portainer/portainer
```

Note that Portainer comes pre-installed on your Duckiebot, so you don't need to run the above command to access the images and containers on your robot. You still might want to set it up for your computer.

4.9. Hands on

Before you can do any software development in Duckietown, you need to get comfortable with Docker and its tools.

Complete the following steps before proceeding to the next section:

1. Install Docker
2. Orientation and Setup
3. Build and run your image
4. Share images on Docker Hub

If you still feel like there is something that you are missing about Docker, you might want to spend some time going through this guide as well.

4.10. Ask the community

If you need help with Docker basics or the use of Docker in Duckietown, join the Slack channel #help-docker.

Unit A-5
Developer Basics: Duckietown Shell

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This section of the book will introduce the Duckietown Shell (`dts` in short) and the reason behind its creation. In this book, we will use dts commands quite often, make sure you don't miss this section.

Contents

5.1. Brief History

The Duckietown Shell is indeed a shell. It was created in July 2018 to help Duckietown users launch Duckietown demos on a Duckiebot. It became clear pretty soon that having a dedicated shell for Duckietown was a game changer for the whole community. In fact, since the very beginning, the shell had a built-in system for auto-update, which allowed developers to develop new commands or improve old ones and deploy the changes in no time.

Duckietown has a history of using `Makefiles` as a way to simplify complex and operations involving many (usually very long) bash commands. Other developers, instead, preferred bash scripts over Makefiles. And finally, our CI system (based on Jenkins), used `Jenkinsfiles` to define automated jobs.

The Duckietown Shell came to the rescue and unified everything, while Makefiles, bash scripts and Jenkinsfiles slowly started disappearing from our repositories. Today, Docker images to run on Duckiebots, Python libraries published on PyPi and even the book you are reading right now are built through dts.

5.2. Get Started

The Duckietown Shell is released as a Python3 package through the PyPi package store. You can install the Duckietown Shell on your computer by running,

💻
```
$ pip3 install duckietown-shell
```

This will install the `dts` command. The Duckietown Shell is distribution independent, so the first time you launch it you have to specify the distribution of Duckietown software you are working on. You can do so by running the command,

```
$ dts --set-version DISTRO
```

where *DISTRO* can be any of the official distributions of Duckietown software, e.g., `master19`, `daffy`. This will download the commands for the given distribution before the command prompt is shown.

Use the command,

```
$ (Cmd) commands
```

to list all the commands available to the chosen distribution.

You don't really need to run the shell before you can type in your command, for example, you can achieve the same result as above by running,

```
$ dts commands
```

Check before you continue

The nice thing about opening the shell before typing your command is that then you can use the Tab key to auto-complete.

5.3. Installable commands

Some commands come **not** pre-installed. These are usually commands that are either very specific to an application, thus not useful to the majority of Duckietown users, or commands that can only be used during a short time window, like commands that let you participate to competitions periodically organized at international AI and Robotics conferences, e.g. AIDO.

5.4. Hands on

Install the Duckietown Shell as instructed in Section 5.2 - Get Started. Make sure everything works as expected by running the command `dts update` successfully.

5.5. Ask the community

If you have any questions about the Duckietown Shell, join the Slack channel #help-dt-shell.

Unit A-6
Developer Basics: ROS

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

In this section, we will introduce the ROS (Robot Operating System) ecosystem.

Contents

6.1. Hands on

6.2. Ask the community

Part B
Modules

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This section of the book focuses on the concept of software module in Duckietown.

Devices in Duckietown (e.g., Duckiebots, Watchtowers, etc) are not configured to accept code directly. The Operating System running on the on-board computers is configured to accept only code running inside Docker containers. Modules are an easy, robust and effective way of wrapping code into executable Docker images.

Remember, you are not allowed to run any code on any of these devices outside a proper Duckietown module. So, if you have code to run, you need to put it in a module first.

Contents

UNIT B-1
Introduction

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

A software module in Duckietown implements a high level behavior, for example, autonomous driving in Duckietown. A software module is broken into a set of smaller pieces, called nodes. This allows us to tackle a complex problem by leveraging solutions to smaller problems. This approach is very common in software development and is inspired by a military strategy called "divide and conquer" (latin: *divide et impera*) commonly used by the Roman Empire.
Breaking a module into nodes is not trivial, and where you draw the lines between nodes can make a huge difference in the final outcome. Some qualities of the final software product directly affected by this decision are **usability**, **mantainability**, and **portability**.

> **Note:** Read the ISO/IEC 9126 standard to learn more about product quality in software engineering. If you want to become a developer, you might want to bookmark that URL, you will need it, **a lot**.

Contents

1.1. What is a module

A module in Duckietown is a Docker image that complies with a **module template** (more about templates in the next section). Remember the ISO/IEC 9126 standard? well, modules are designed to be highly **portable** and **usable**.

Modules have a pre-defined file system structure with fixed locations for source code and configuration files. File system structure and default locations are template-dependent, check the section **TBD** to understand the different module templates available in Duckietown.

Understanding the pros and cons of using Docker to isolate modules right now is crucial.

Bad news first! The biggest negative effect of using Docker to isolate modules is that by doing so, we are wrapping our source code inside a Docker image. This makes it harder for us to do development, since our code will not be easily accessible through our local file system. This is what scares/frustrates people away from Docker the most. Keep it in mind, if it happens to you, you are not the only one. The Duckietown development workflow explained in this book aims, among other things, at reducing the effect of this code isolation. We will get back to this topic later in the book.

As for the good news, i.e., why using Docker to isolate modules makes sense and our life easier, we could write a book about it, but they will become clear as we proceed.

Duckietown defines a set of module types that you can choose from. The list of module types and their differences will be the topic of the section **TBD**. What is important to know for now, is that a module type defines the environment your code will run in. For each module type, a template repository is provided.

1.2. Module Templates

A module template is a repository, hosted on GitHub space of the Duckietown organization, that lets you build a new module of that type.

Templated repositories on GitHub are special repositories that you can use to initialize an empty repository. Templated repository will provide an initial structure to your empty repository.

Although different template repositories have different files and structures, they all contain: a `Dockerfile`, used to *compile* the entire repository into a Docker image; one or more requirements files, in which you can list the dependencies of your code; a `.dt-project` file that makes it compatible with the duckietown shell. You will find other files as well, but these are the most important ones.

The simplest module template is called `basic` and its template repository is duckietown/template-basic. Since understanding the differences between different templates is outside the scope of this section, we can use any template for the remainder of this section, we suggest using the one above.

1.3. Create your own module

In order to be able to create a Duckietown module, you need to gain access to the module template repositories on GitHub. There are two way to achieve this: you are an official Duckietown developer, thus you are part of the Duckietown organization on GitHub; or, you create a copy (fork) of the template you need on your GitHub account.

If you are not a member of the Duckietown organization on GitHub, you can fork a template on your GitHub account by visiting the module template page on GitHub (e.g., duckietown/template-basic) and click on the Fork button at the top-right corner of the page.

Figure 1.1. Fork button on GitHub

Once you gained access to the template (either by joining the Duckietown developers team or forking the template repository), you are able to create a new repository that will store your module based on the template repository. To do so, go to GitHub, click on the [+] button on the header at the top-right corner and then choose **New repository**. In the *New repository* page, choose the template (e.g., `duckietown/template-basic`) and enter the name of your new module (e.g., `my_module`) as shown in the image below.

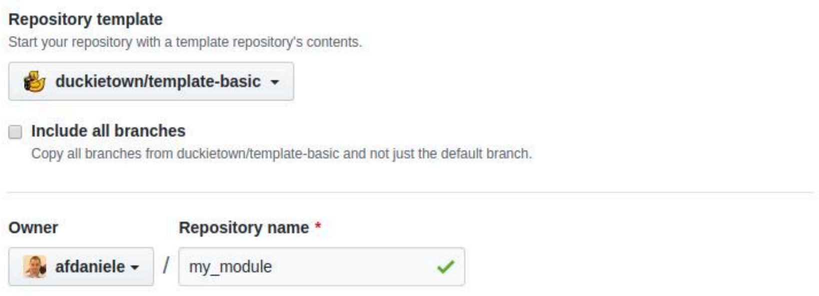

Repository template
Start your repository with a template repository's contents.
duckietown/template-basic
Include all branches
Copy all branches from duckietown/template-basic and not just the default branch.
Owner
afdaniele
Repository name *
my_module

Figure 1.2. New repository with template on GitHub

Click on **Create repository** to create the module repository.

1.4. Build a module

Building a module is very simple. To start, open a terminal and clone a module repository (we created one in section **TODO**).

Templates leave placeholders that you will need to replace with the proper information about your module before you can build it.

Open the file `Dockerfile` using any text editor and look for the following lines at the top of the file:

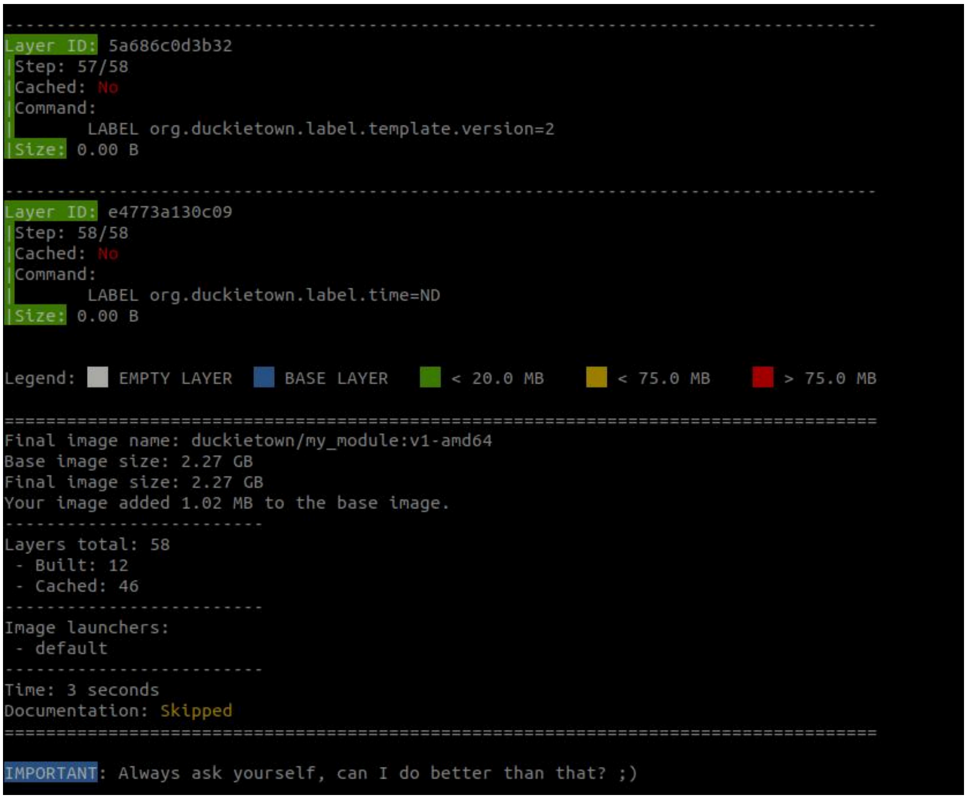
TODO

Replace the placeholders strings with, respectively, - the name of the repository (i.e., `my_module`); - a brief description of the functionalities of the module - your name and email address to claim the role of maintainer;

Save and return to the terminal. Now run the following command to build the module.

```
dts devel build -f
```

The flag `-f` (short for `--force`) is needed in order to allow `dts` to build a module out of a non-clean repository. A repository is not clean when there are changes that are not committed (and in fact our change to `Dockerfile` is not). This check is in place to prevent developers from forgetting to push local changes. If the build is successful, you will see something like the following.



```
-------------------------------------------------------------------------------------
Layer ID: 5a686c0d3b32
|Step: 57/58
|Cached: No
|Command:
|        LABEL org.duckietown.label.template.version=2
|Size: 0.00 B

-------------------------------------------------------------------------------------
Layer ID: e4773a130c09
|Step: 58/58
|Cached: No
|Command:
|        LABEL org.duckietown.label.time=ND
|Size: 0.00 B


Legend: ⬜ EMPTY LAYER  🟦 BASE LAYER  🟩 < 20.0 MB  🟨 < 75.0 MB  🟥 > 75.0 MB

=====================================================================================
Final image name: duckietown/my_module:v1-amd64
Base image size: 2.27 GB
Final image size: 2.27 GB
Your image added 1.02 MB to the base image.
-------------------------
Layers total: 58
 - Built: 12
 - Cached: 46
-------------------------
Image launchers:
 - default
-------------------------
Time: 3 seconds
Documentation: Skipped
=====================================================================================

IMPORTANT: Always ask yourself, can I do better than that? ;)
```

Figure 1.3. Result of command *dtsdevelrun*

Congrats! You just built your first Duckietown-compatible software module.

1.5. Run a module

As stated above, building a module produces a Docker image. This image is the *compiled* version of your source project. You can find the name of the resulting image at the end of the output of the `dts devel build` command. In the example above, look for the line

```
Final image name: duckietown/my_module:v1-amd64
```

1.6. Hands on

1.7. Ask the community

UNIT B-2
Module Types

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

In Duckietown, there are *three* main module types. For each module type we provide a repository template. The module types are:

- **basic** (template: duckietown/template-basic)
- **ros** (template: duckietown/template-ros)
- **core** (template: duckietown/template-core)

We will go through each one of them in details in the next sections.

2.1. Module Type: Basic

The module type **basic** is the one

2.2. Hands on

2.3. Ask the community

Unit B-3
TEMPLATE

Author: Andrea F. Daniele
Maintainer: Andrea F. Daniele

This section of the book focuses on TEMPLATE.

Contents

3.1. Hands on

3.2. Ask the community

Part C
The Duckietown Code-Fu

Author: Aleksandar Petrov

Maintainer: Aleksandar Petrov

This section of the book will discuss a variety of topics pertaining to the style and design choices for the code that runs on the Duckietown robots. If you wish for your code to be merged in our official repositories, read this section extremely carefully and make sure to follow all the guidelines provided here.

Contents

Unit C-1
Structuring a Duckietown repository

Author: Aleksandar Petrov

Maintainer: Aleksandar Petrov

Contents

This section deals with how you should structure your repository and what should go where in it.

1.1. Ask the community

If you have any questions about good practices in software development, join the Slack channel #info-developers.

UNIT C-2
Structuring ROS Packages

Author: Aleksandar Petrov
Maintainer: Aleksandar Petrov

Contents

This section deals with how you should structure your packages and what should go where in them.

2.1. Ask the community

If you have any questions about good practices in software development, join the Slack channel #info-developers.

UNIT C-3
Structuring ROS Nodes

Author: Aleksandar Petrov

Maintainer: Aleksandar Petrov

Contents

This section deals with how you should write the code in a ROS node. In particular, how to structure it. Writing the code of a node goes hand-in-hand with documenting it but this will be discussed in more detail in Unit C-4 - Documenting your code.

3.1. General structure

All ROS nodes should be in the `src` directory of the respective package. If the node is called `some_name`, then the file that has its implementation should be called `some_name_node.py`. This file should always be executable. Furthermore, all the logic of the node should be implemented in a Python class called `SomeNameNode`.

The structure of the `some_name_node.py` should generally look like the following example (without the comments):

```
#!/usr/bin/env python

# import external libraries
import rospy

# import libraries which are part of the package (i.e. in the include
dir)
import library

# import DTROS-related classes
from duckietown.dtros import
    DTROS,
    NodeType,
    TopicType,
    DTReminder,
    DTParam,
    ParamType

# import messages and services
from std_msgs.msg import Float32
from duckietown_msgs.msg import
    SegmentList,
    Segment,
    BoolStamped

class SomeNameNode(DTROS):
    def         (self, node_name):
        # class implementation

if __name__ == '__main__':
    some_name_node = SomeNameNode(node_name='same_name_node')
    rospy.spin()
```

Observe that all nodes in Duckietown should inherit from the super class `DTROS`. This is a hard requirement. `DTROS` provides a lot of functionalities on top of the standard ROS nodes which make writing and debugging your node easier, and also sometimes comes with performance improvements.

In Python code, never ever do universal imports like `from somepackage import *`. This is an extremely bad practice. Instead, specify exactly what you are importing, i.e. `from somepackage import somefunction`. It is fine if you do it in `__init__.py` files but even there try to avoid it if possible.

When using a package that has a common practice alias, use it, e.g. `import numpy as np`, `import matplotlib.pyplot as plt`, etc. However, refrain from defining your own aliases.

The code in this node definition should be restricted as much as possible to ROS-related functionalities. If your node is performing some complex computation or has any logic that can be separated from the node itself, implement it as a separate library and put it in the `include` directory of the package.

3.2. Node initialization

There are a lot of the details regarding the initalization of the node so let’s take a look at an example structure of the `__init__` method of our sample node.

```
class SomeNameNode(DTROS):
    def         (self, node_name):
        super(SomeNameNode, self).        (
            node_name=node_name,
            node_type=NodeType.PERCEPTION
        )

        # Setting up parameters
        self.detection_freq = DTParam(
            '~detection_freq',
            param_type=ParamType.INT,
            min_value=-1,
            max_value=30
        )
        # ...

        # Generic attributes
        self.something_happened = None
        self.arbitrary_counter = 0
        # ...

        # Subscribers
        self.sub_img = rospy.Subscriber(
            'image_rect',
            Image,
            self.cb_img
        )
        self.sub_cinfo = rospy.Subscriber(
            'camera_info',
            CameraInfo,
            self.cb_cinfo
        )
        # ...

        # Publishers
        self.pub_img = rospy.Publisher(
            'tag_detections_image/compressed',
            CompressedImage,
            queue_size=1,
            dt_topic_type=TopicType.VISUALIZATION
        )
        self.pub_tag = rospy.Publisher(
            'tag_detections',
            AprilTagDetectionArray,
            queue_size=1,
            dt_topic_type=TopicType.PERCEPTION
        )
        # ...
```

Now, let's take a look at it section by section.

1) Node Creation

In classic ROS nodes, you would initialize a ROS node with the function `rospy.init_node(...)`. DTROS does that for you, you simply need to pass the node name that you want to the super constructor as shown above.

DTROS supports node categorization, this is useful when you want to visualize the ROS network as a graph, where graph nodes represent ROS nodes and graph edges represent ROS topics. In such a graph, you mught want to group all the nodes working on the `PERCEPTION` problem together, say, to clear the clutter and make the graph easier to read. Use the parameter `node_type` in the super constructor of your node to do so. Use the values from the `NodeType` enumeration. Possible node types are the following,

```
GENERIC
DRIVER
PERCEPTION
CONTROL
PLANNING
LOCALIZATION
MAPPING
SWARM
BEHAVIOR
VISUALIZATION
INFRASTRUCTURE
COMMUNICATION
DIAGNOSTICS
DEBUG
```

2) Node Parameters

All parameters should have names relative to the namespace of the node, i.e. they should start with `~`. Also, all parameters should be in the scope of the instance, not the method, so they should always be declared inside the constructor and start with `self.`.

The parameters should never have default values set in the code. All default values should be in the configuration file! This makes sure that we don't end up in a situation where there are two different default values in two different files related to the node.

In classic ROS, you get the value of a parameter with `rospy.get_param(...)`. One of the issues of the ROS implementation of parameters is that a node cannot request to be notified when a parameter's value changes at runtime. Common solutions to this problem employ a polling strategy (which consists of querying the parameter server for changes in value at regular intervals). This is highly inefficient and does not scale. The `dtros` library provides a solution to this. Alternatively to using `rospy.get_param(...)` which simply returns you the current value of a paramter, you can create a `DTParam` object that automatically updates when a new value is set. Use `self.my_param = DTParam("~my_param")` to create a `DTParam` object and `self.my_param.value` to read its value.

3) Generic attributes

Then we initialize all the non-ROS attributes that we will need for this class. Note that this is done *before* initializing the Publishers and Subscribers. The reason is that if a subscriber's callback depends on one of these attributes, we need to define it before we use it. Here's an example that might fail:

✖

```
class CoolNode(DTROS):
   def         (...):
       self.sub_a = rospy.Subscriber(..., callback=cb_sth, ...)
       self.important_variable = 3.1415

   def cb_sth(self):
       self.important_variable *= 1.0
```

And something that is better:

✔

```
class CoolNode(DTROS):
   def         (...):
       self.important_variable = 3.1415
       sub_a = rospy.Subscriber(..., callback=cb_sth, ...)

   def cb_sth(self):
       self.important_variable *= 1.0
```

4) Publishers and Subscribers

Finally, we initialize all the Subscribers and Publishers as shown above. The `dtros` library automatically decorates the methods `rospy.Publisher` and `rospy.Subscriber`. By doing so, new parameters are added. All the parameters added by `dtros` have the prefix `dt_` (e.g., `dt_topic_type`). Use the values from the `TopicType` enumeration. Possible types list is identical to the node types list above.

> Note: Only declare a topic type in a `rospy.Publisher` call.

3.3. Naming of variables and functions

All functions, methods, and variables in Duckietown code should be named using `snake_case`. In other words, only lowercase letters with spaces replaced by underscored. Do **not** use `CamelCase`. This is to be used **only** for class names.

The names of all subscribers should start with `sub_` as in the example above. Similarly, names of publishers should start with `pub_` and names of callback functions should start with `cb_`.

Initalizing publishers and subscribers should again always be in the scope of the instance, hence starting with `self.`.

3.4. Switching nodes on and off

3.5. Custom behavior on shutdown

If you need to take care of something before when ROS tries to shutdown the node but before it actually shuts it down, you can implement the `on_shutdown` method. This is useful if you are running threads in the background, there are some files that you need to close, resources to release, or to put the robot into a safe state (e.g. to stop the wheels).

3.6. Handling debug topics

Often we want to publish some information which helps us analyze the behavior and performance of the node but which does not contribute to the behavior itself. For example, in order to check how well the lane filter works, you might want to plot all the detected segments on a map of the road. However, this can be quite computationally expensive and is needed only on the rare occasion that someone wants to take a look at it.

A frequent (**but bad design**) way of handling that is to have a topic, to which one can publish a message, which when received will induce the node to start building a publishing the debug message. A much better way, and the one that **should be used in Duckietown** is to create and publish the debug message *only if* someone has subscribed to the debug topic. This is very easy to achieve with the help of `dtros`. Publishers created within a DTROS node exports the utility function `anybody_listening()`. Here's an example:

```
if self.pub_debug_img.anybody_listening():
    debug_img = self.very_expensive_function()
    debug_image_msg = self.bridge.cv2_to_compressed_imgmsg(debug_img)
    self.pub_debug_img.publish(debug_image_msg)
```

Note also that all debug topics should be in the `debug` namespace of the node, i.e. `~debug/debug_topic_name`.

Similarly, a Subscribers created within a DTROS node exports the utility function `anybody_publishing()` that checks whether there are nodes that are currently publishing messages.

3.7. Timed sections

If you have operations that might take non-trivial amount of computational time, you can profile them in order to be able to analyze the performance of your node. `DTROS` has a special context for that which uses the same mechanism as the debug topics. Hence, if you do not subscribe to the topic with the timing information, there would be no overhead to your performance. Therefore, be generous with the use of timed sections.

The syntax looks like that:

```
with self.time_phase("Step 1"):
    run_step_1()

...

with self.time_phase("Step 2"):
    run_step_2()
```

Then, if you subscribe to `~debug/phase_times` you will be able to see for each separate section detailed information about the frequency of executing it, the average time it takes, and also the exact lines of code and the file in which this section appears.

3.8. Config files

If your node has at least one parameter, then it should have a configuration file. If there is a single configuration (as is the case with most nodes) this file should be called `default.yaml`. Assuming that our node is called `some_node`, the configuration files for the node should be in the `config/some_node/` directory.

Every parameter used in the implementation of the node should have a default value in the configuration file. Furthermore, there should be no default values in the code. The only place where they should be defined is the configuration file.

3.9. Launch files

Assuming that our node is called `some_node` then in the `launch` directory of the package there should be an atomic launch file with the name `some_node.launch` which launches the node in the correct namespace and loads its configuration parameters.

The launch file content of most node will be identical to the following, with only the node name and package name being changed.

```
<launch>
  <arg name="veh"/>
  <arg name="pkg_name" value="some_package"/>
  <arg name="node_name" default="some_node"/>
  <arg name="param_file_name" default="default" doc="Specify a param
file"/>

  <group ns="$(arg veh)">
    <node  name="$(arg node_name)" pkg="$(arg pkg_name)" type="$(arg
node_name).py" output="screen">
      <rosparam command="load" file="$(find some_package)/config/$(arg
node_name)/$(arg param_file_name).yaml"/>
    </node>
  </group>

</launch>
```

3.10. Ask the community

If you have any questions about good practices in software development, join the Slack channel #info-developers.

UNIT C-4
Documenting your code

Author: Aleksandar Petrov

Maintainer: Aleksandar Petrov

Contents

This section provides a comprehensive guide to writing inline documentation to your ROS nodes and libraries. We will discuss both *what* should be documented and *how* it should be documented. Head to Unit C-5 - Building the documentation of your code for the details about how to then create a human-friendly webpage showing the documentation.

4.1. Basics about inline code documentation

Documentation is a must-do in any software project. As harsh as it sounds, you can write an aboslutely revolutionary and beautiful software package that can save anyone who uses it years of their time while simultaneously solving all of humanity's gratest problems. But if people do not know that your code exists, have no idea how to use it, or understanding its intricacies takes too much effort, then you will neither save anyone any time, nor solve any problem. In fact, as far as the world beyond you is concerned, all your work is as good as if never done. Therefore, if you wish your code to be ever used, **document** it as extensively as possible!

Inline documentation is a pretty handy way of helping people use your software. On one hand, it is right in the place where it is needed: next to the classes and methods that you are documenting. On the hand, it is also much easier to update when you change something in the code: the documentation of your function is typically no more than 20 lines above your change. On the third hand (should you have one) documentation written in this way inherits the structure of your software project, which is a very natural way of organizing it. On the fourth hand (you can borrow someone else's), there are some really nice packages that take your documentation and make it into a beautiful webpage.

> Note: When we refer to code documentation we mean things like this. The documentation that you are currently reading is called *a book* and exists independent of any code repository.

In Duckietown we use Sphinx for building our code documentation. Sphinx is the most popular way of creating code documentation for Python projects. You can find out more on their webpage and there are a lot of interesting things to read there. Documenting your code is as simple as writing docstrings and the occasional comment. Then, Sphinx takes care of parsing all your docstrings and making a nice webpage for it. However, in order for all this to work nicely, you need to formal your documentation in a particular way. We will discuss this later in this page.

4.2. What should be documented and where?

The short answer to this question is "everything and in the right place". The long answer is the same.

Every ROS node should be documented, meaning a general description of what it is for, what it does, and how it does it. Additionally, all its parameters, publishers, subscribers, and services need to be described. The default values for the parameters should be also added in the documentation. Every method that you node's class has should also be documented, including the arguments to the method and the returned object (if there is such), as well as their types.

Every library in your `include` directory should also be documented. That again means, every class, every method, every function. Additionally, the library itself, and its modules should have a short description too.

Finally, your whole repository, and every single package should also be documented.

Let's start from a node. Here is a sample for the camera node:

```
class CameraNode(DTROS):
    """

    The node handles the image stream, initializing it, publishing
frames
    according to the required frequency and stops it at shutdown.
    `Picamera <https://picamera.readthedocs.io/>`_ is used for handling
    the image stream.

    Note that only one :obj:`PiCamera` object should be used at a time.
    If another node tries to start an instance while this node is run-
ning,
    it will likely fail with an `Out of resource` exception.

    Args:
        node_name (:obj:`str`): a unique, descriptive name for the node
that ROS will use

    Configuration:
        ~framerate (:obj:`float`): The camera image acquisition framer-
ate, default is 30.0 fps
        ~res_w (:obj:`int`): The desired width of the acquired image,
default is 640px
        ~res_h (:obj:`int`): The desired height of the acquired image,
default is 480px
        ~exposure_mode (:obj:`str`): PiCamera exposure mode

    Publisher:
        ~image/compressed (:obj:`CompressedImage`): The acquired camera
images

    Service:
        ~set_camera_info:
            Saves a provided camera info to `/data/config/calibrations/
camera_intrinsic/HOSTNAME.yaml`.

            input:
                camera_info (obj:`CameraInfo`): The camera information
to save

            outputs:
                success (:obj:`bool`): `True` if the call succeeded
                status_message (:obj:`str`): Used to give details about
success

    """

    def          (self, node_name):

        # Initialize the DTROS parent class
        super(CameraNode, self).        (node_name=node_name,
                                         node_type=NodeType.PERCEPTION)
```

The documentation of the node itself should *always* be as a docstring after the class definition. *Do not* put it, or anything else as a docstring for the `__init__` method. This will not be rendered in the final output.

The documentation of the node should start with a general description about the node, its purpose, where it fits in the bigger picture of the package and repository, etc. Feel generous with the description here. Then there is a section with the arguments needed for initializing the node (the arguments of the `__init__` method) which will almost always be exactly the same as shown. After that there is a configuration section where you should put all the parameters for the node, their type, a short description, and their default value, as shown.

This is then followed by Subscribers, Publishers and Services, in this order. If the node has no Subscribers, for example as the camera node, then you don't need to add this section. Note the specific way of structuring the documentation of the service!

Then, every method should be documented as a docstring immediately after the function definition (as the `save_camera_info` example). Again, add a short description of the method, as well as the arguments it expects and the return value (should such exist).

Libraries should be documented in a similar way. However, when documenting libraries, it is important to actually invoke the Sphinx commands for documenting particular objects in the `__init__.py` file. Furthermore, this file should contain a description of the package itself. Here's an example from the `line_detector` library's `__init__.py` file:

```
"""

    line_detector
    -------------

    The ``line_detector`` library packages classes and tools for han-
dling line section extraction from images. The
    main functionality is in the :py:class:`LineDetector` class.
:py:class:`Detections` is the output data class for
    the results of a call to :py:class:`LineDetector`, and
:py:class:`ColorRange` is used to specify the colour ranges
    in which :py:class:`LineDetector` is looking for line segments.

    There are two plotting utilities also included: :py:func:`plotMaps`
and :py:func:`plotSegments`

    .. autoclass:: line_detector.Detections

    .. autoclass:: line_detector.ColorRange

    .. autoclass:: line_detector.LineDetector

    .. autofunction:: line_detector.plotMaps

    .. autofunction:: line_detector.plotSegments


"""
```

You can see that it describes the library and its elements, and then uses the Sphinx commands which will parse these classes and functions and will add their documentation to this page. You can find more details about these functions in Section 4.3 - Style guide.

In a similar way, every ROS package needs a documentation file. This should go in the `docs/packages` directory of your repository and should be named `package_name.rst`. It should describe the package and then should invoke the Sphinx commands for building the documentation for the individual nodes and libraries. See the following example:

```
ROS Package: ground\_projection
===============================

.. contents::

The ``ground_projection`` package provides the tools for projecting
line segments from an image reference frame to the ground reference
frame, as well as a ROS node that implements this functionality. It has
been designed to be a part of the lane localization pipeline. Consists
of the ROS node :py:class:`nodes.GroundProjectionNode` and the
:py:mod:`ground_projection` library.


GroundProjectionNode
--------------------

.. autoclass:: nodes.GroundProjectionNode

Included libraries
------------------

.. automodule:: ground_projection
```

4.3. Style guide

You probably noticed the plethora of funky commands in the above examples. These are called *directives* and we'll now take a closer look at them. The basic style of the documentation comes from reStructuredText, which is the default plaintext markup language used by Sphinx. The rest are Sphinx directives which Sphinx then replaces with markup which it creates from your docstrings.

1) Basic styles

- You can use `*text*` to italize the text.
- You can use `**text**` to make it in boldface.
- Values, names of variables, errors, messages, etc, should be in grave accent quotes:

```
``like that``
```

- Section are created by underlying section title with a punctuation character, at least as long as the text:

```
What a cool heading
===================

Nice subsection
---------------

A neat subsubsection
^^^^^^^^^^^^^^^^^^^^
```

- External links can be added like this:

```
    For this, we use `Picamera <https://picamera.readthedocs.io/>`_
which is an external library.
```

- When describing standard types (like `int`, `float`, etc.) use

```
:obj:`int`
```

- If the type is an object of one of the libraries in the repository, then use the referencing directives from the next section in order to create hyperlinks. If it is a message, use the message type. If a list, a dictionary, or a tuple, you can use expressions like `` :obj:`list` of :obj:`float` ``
- Attributes of a class can also be documented. We recommend that you do that for all important attributes and for constants. Here are examples of the various ways you can document attributes:

```
class Foo:
    """Docstring for class Foo."""

    #: Doc comment for class attribute Foo.bar.
    #: It can have multiple lines.
    bar = 1

    flox = 1.5   #: Doc comment for Foo.flox. One line only.

    baz = 2
    """Docstring for class attribute Foo.baz."""

    def __init__(self):
        #: Doc comment for instance attribute qux.
        self.qux = 3

        self.spam = 4
        """Docstring for instance attribute spam."""
```

You can find more examples with reStructuredText here and here, and detailed specification here.

2) Referencing other objects

You can add a link to a different package, node, method, or object like that:

```
    :py:mod:`duckietown`
    :py:class:`duckietown.DTROS`
    :py:meth:`duckietown.DTROS.publisher`
    :py:attr:`duckietown.DTROS.switch`
```

All of these refer to the `duckietown` Python package. When dealing will nodes, things are a bit trickier, because they are not a part of a package. However, in order to make Sphinx work nicely with ROS nodes, we create a fake package that has them all as classes. Hence, if you want to refer to the CameraNode, you can do it like that:

```
    :py:class:`nodes.CameraNode`
```

Note: We are considering replacing `nodes` with the repository name, so keep in mind this might change soon.

3) Custom sections

When documenting a node, you can (and you should) make use of the following ROS-specific sections: `Examples`, `Raises`, `Configuration`, `Subscribers`, `Subscriber`, `Publishers`, `Publisher`, `Services`, `Service`, `Fields`, `inputs`, `input`, `outputs`, `output`. If you need other custom sections you can add them in the `docs/config.yaml` file in your repository.

4) Using autodoc

We use the autodoc extension of Sphinx in order to automatically create the markup from the docstrings in our Python code. In particular, you can use the following directives:

```
.. automodule:: ground_projection

.. autoclass:: line_detector.ColorRange

.. autofunction:: line_detector.plotMaps

.. automethod:: nodes.CameraNode.save_camera_info
```

You can find more details here.

4.4. Ask the community

If you have any questions about good practices in software development, join the Slack channel #info-developers.

UNIT C-5
Building the documentation of your code

Author: Aleksandar Petrov
Maintainer: Aleksandar Petrov

Contents

This section explains the build process for the Sphinx documentation.

5.1. Ask the community

If you have any questions about good practices in software development, join the Slack channel #info-developers.

Part D
Software Diagnostics

One of the strenghts of Duckietown is that of allowing complex (sometimes state-of-the-art) algorithms to run on low-end computing devices like the Raspberry Pi. Unfortunately, low-end devices are not famous for their computational power, so we developers have to be smart about the way we use the resources available.

The Duckietown Diagnostics tool provides a simple way of *recording* the status of a system during an *experiment*. The easiest way to think about it is that of an observer taking snapshots of the status of our system at regular temporal intervals.

Contents

UNIT D-1
Introduction

Contents

1.1. Running example

Throughout this section we will refer to the toy example of a robot with a single camera (just like our Duckiebots) in which camera drivers produce image frames at a frequency of `20Hz` and we are interested in pushing the camera to its limit, i.e., `30Hz`.

1.2. When do I need it?

You need to run the diagnostics tool every time you have made changes to a piece of code and you want to test how these changes affect the footprint of your code on the system resources and the system as a whole.

Considering our toy example above, we expect that changing the drivers frequency will likely result in higher usage of resources in order to cope with the increase in images that need to be processed. Sometimes these changes have a direct and expected effect on the system's resources, e.g., CPU cycles, RAM, etc. Others, they have effects that are legitimate from a theoretical point of view but hard to exhaustively enumerate a priori, e.g., increase in CPU temperature due to higher clock frequencies, increase in network traffic if the images are transferred over the network.

The diagnostics tool provides a standard way of analyzing the response of a system to a change.

1.3. When do I run it?

The diagnostics tool is commonly used for two use cases:

- analysis of *steady states* (long-term effects) of a system;
- analysis of *transient states* (short-term effects) of a system;

The *steady state* analysis consists of measuring the activity of a system in the long run and in the absense of anomaly or changes. For example, if we want to check for memory leaks in a system, we would run a *steady state* analysis and look at the RAM usage in a long period of time. In this case, we would run the diagnostics tool only after the system reached a stable (steady) state and we don't expect significant events to happen.

The *transient state* analysis consists of measuring how a system reacts to a change in the short run. For example, you have a process that receives point clouds from a sensor and stores them in memory to perform ICP alignment on them every `T` seconds. In this case, we expect that this process will be fairly inactive in terms of CPU usage for most of the time with periodical spikes every `T` seconds. Clearly, small values of `T` mean fewer point clouds to align every time ICP fires but more frequent alignments while large values of `T` mean longer queues of point clouds to align every time ICP fires. We might be interested in tuning the value of `T` so that those spikes do not starve other processes of resources while still maximizing `T`. In this case, we would monitor the system around those ICP events for different values of `T`. In this case, we would run the diagnostics tool at any point in time for a duration of `t > T` seconds so that at least one event of interest (e.g., ICP event) is captured.

Unit D-2
Get Started

In this section, we will see how to perform a diagnostics experiment.

> **Note:** At the end of each diagnostics test, the resulting log is automatically transferred to a remote server. If the diagnostics tool fails to transfer the log to the server, the tests data will be lost and the test need to be run again.

Contents

2.1. Run a (single test) diagnostics experiment

You can run a diagnostics test for `60` seconds on the target device *ROBOT* with the command

```
dts diagnostics run -H ROBOT -G my_experiment -d 60
```

Let the diagnostics tool run until it finishes. A successful experiment concludes with a log similar to the following:

```
. . .
[system-monitor 00h:00m:55s] [healthy] [8/8 jobs] [13 queued] [0
failed] ...
[system-monitor 00h:00m:58s] [healthy] [8/8 jobs] [13 queued] [0
failed] ...
INFO:system-monitor:The monitor timed out. Clearing jobs...
INFO:system-monitor:Jobs cleared
INFO:system-monitor:Collecting logged data
INFO:system-monitor:Pushing data to the cloud
INFO:system-monitor:Pushing to the server [trial 1/3]...
INFO:system-monitor:The server says: [200] OK
INFO:system-monitor:Data transferred successfully!
INFO:system-monitor:Stopping workers...
INFO:system-monitor:Workers stopped!
INFO:system-monitor:Done!
```

The most important thing to look for is the line

```
INFO:system-monitor:The server says: [200] OK
```

which indicates that the diagnostics log was successfully transferred to the remote diagnostics server.

2.2. Visualize the results

The diagnostics server collects diagnostics logs and organizes them according to the given group, subgroup and hostname of the target machine of each test.

To check the outcome of a diagnostics test, open your browser and navigate to https://dashboard.duckietown.org/diagnostics. Tests will be available on this page a few seconds after the upload is complete.

Use the dropdowns `Group` and `Subgroup` to find your experiment and test. Remember, when the subgroup is not explicitly specified with the argument `-S/--subgroup`, `default` is used.

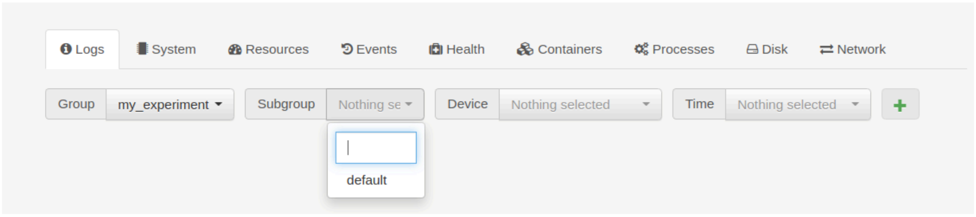


Figure 2.1. Selecting diagnostics test on dashboard.duckietown.org

Use the tabs `System`, `Resources`, etc. to see the content of the diagnostics log.

2.3. One experiment, many tests

In many cases, your experiment is that of comparing two or more configurations or implementations of part of your system. In these cases, you need to run multiple tests as part of a single experiment. The diagnostics tool allows you to declare a group (`-G/--group`) and a subgroup (`-S/--subgroup`) when you run a test. Use the group argument to name your experiment and the subgroup to name the single tests.

Let us recall the example of Section 1.1 - Running example. We want to measure the effects of changing the drivers frequency on the system, so we run (and monitor) the system twice, a first time with the frequency tuned at `20Hz`, and a second time with the frequency at `30Hz`. We call the overall **experiment** `camera_frequency` and the two **tests**, `20hz` and `30hz` respectively. We can use the following commands to run the two tests described above, one before and the other after applying the change to the camera drivers code.

```
dts diagnostics run -H ROBOT -G camera_frequency -S 20hz -d 60
dts diagnostics run -H ROBOT -G camera_frequency -S 30hz -d 60
```

We can now use the Diagnostics page available at https://dashboard.duckietown.org/diagnostics. to visualize both tests side by side. Similarly to what we have done in Section 2.2 - Visualize the results, we will use the dropdown buttons to select our tests and add them to the list. Once we have both on the list, we can move to the other tabs to see how the results of the two tests compare.

UNIT D-3
Reference

In this section, we will describe the various arguments that the diagnostics tool accepts. Use them to configure the diagnostics tool to fit your needs.

Contents

3.1. Usage

You can run a diagnostics test using the command:

```
dts diagnostics run -H/--machine ROBOT -G/--group EXPERIMENT -d/--dura-
tion SECONDS [OPTIONS]
```

3.2. Options

The following table describes the **options** available to the diagnostics tool.

TABLE 3.1. OPTIONS AVAILABLE TO THE COMMAND *dtsdiagnosticsrun*

Argument	Type	Description
`-H` `--machine`	—	Machine where the diagnostics tool will run. This can be any machine with a network connection to the target machine.
`-T` `--target`	localhost	Machine target of the diagnostics. This is the machine about which the log is created.
`--type`	auto	Specify a device type (e.g., duckiebot, watchtower). Use `--help` to see the list of allowed values.
`--app-id`	—	ID of the API App used to authenticate the push to the server. Must have access to the 'data/set' API endpoint
`--app-secret`	—	Secret of the API App used to authenticate the push to the server
`-D` `--database`	—	Name of the logging database. Must be an existing database.
`-G` `--group`	—	Name of the experiment (e.g., new_fan)
`-S` `--subgroup`	"default"	Name of the test within the experiment (e.g., fan_model_X)
`-D` `--duration`	—	Length of the analysis in seconds, (-1: indefinite)
`-F` `--filter`	"*"	Specify regexes used to filter the monitored containers
`-m` `--notes`	"empty"	Custom notes to attach to the log
`--no-pull`	False	Whether we do not try to pull the diagnostics image before running the experiment
`--debug`	False	Run in debug mode
`--vv` `--verbose`	False	Run in debug mode

Part E
Projects

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This section of the book focuses on the projects that are made in Duckietown.

Part F
Benchmarking

Author: Linus Lingg
Maintainer: Linus Lingg

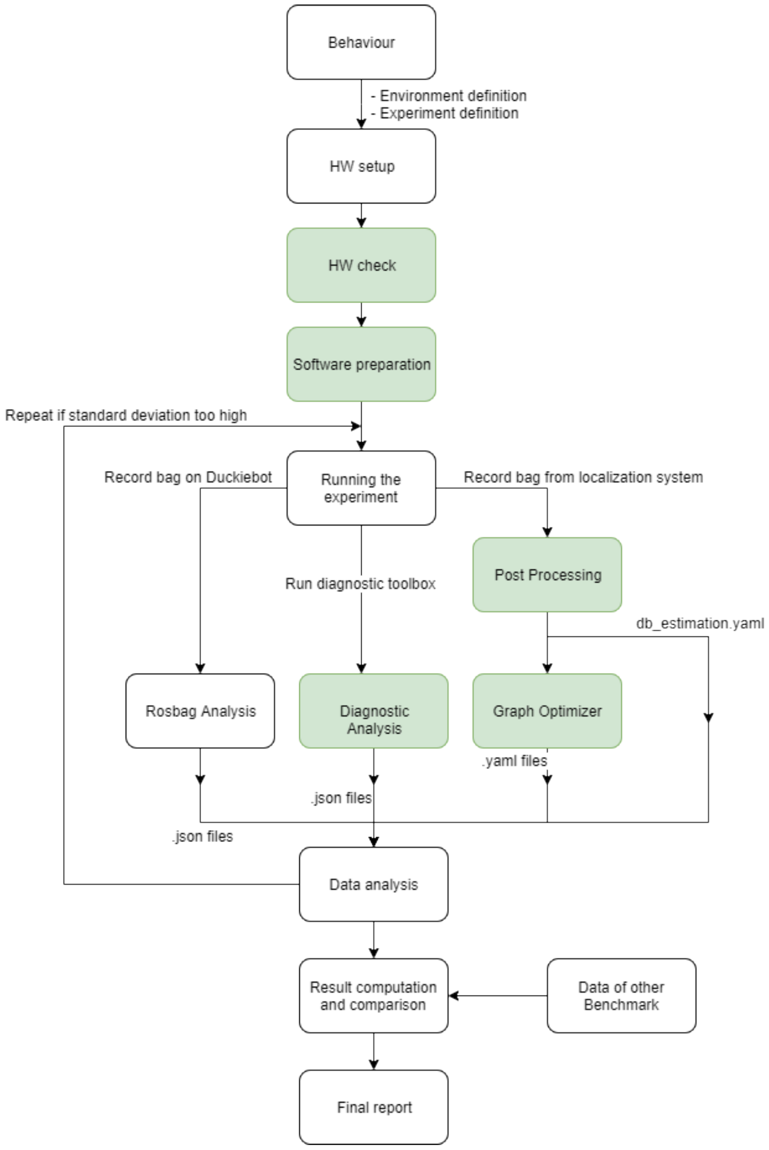
1) Introduction

This section of the book focuses on the concept of behaviour benchmarking in Duckietown. There will be first a brief general introduction to introduce the general architecture of the Behaviour Benchmarking. The most important parts are explained as well as some general information is given. The presentation as well as the repository corresponding to the Behaviour Benchmarking are also available.

Contents

2) General Architecture

Below you can see the general architecture that was set up for the Behaviour Benchmarking. The Nodes that are marked in green as well as the general architecture form are the same for no mather what behaviour that is benchmarked. The other nodes might vary slightly from behaviour to behaviour.



So first of all, the Behaviour defines the experiment definition as well as the environment definition. Based on that the hardware set up is explained and can be done. After this there will be a Hardware check which assures that everything is ready to compute results that can then be compared fairly with other Benchmarks of the same behaviour. Furthermore, the software set up will be explained and can be prepared. With all that done, it is all ready to run the actual experiments in which all the needed data is recorded. It is necessary to run at least two experiments to be able to check if the data received is reliable. Reliable means, that the there was no falsified measurement by the localization system for example, and that “lucky” runs can be excluded for sure. Therefore, the user is asked in the beginning to run at least two experiments, out of those measurements it will then later, based on the repetition criteria, be concluded if this data is reliable or if the user needs to run another experiment. Collecting all the needed data means that on one hand the diagnostic toolbox is running to record all kind of information about the overall CPU usage, memory usage etc as well as the CPU usage etc from all the different containers running and even of each node. On the other hand a bag is recorded directly on the Duckiebot which records all kind of necessary data published directly by the Duckiebot which are information about its estimations, the latency, update frequency of all the different topics etc. And last but not least, a bag is recorded that records everything from the localization system (same procedure as offline localization) which is taken as the Ground Truth . The localization system measures the position of the center of the April Tag placed on top of the localization standoff on the Duckiebot. So each time the position of the Duckiebot or the speed (for example time needed per tile etc) of the Duckiebot is mentioned it is always referred to the center of the AprilTag on top of the Duckiebot. This means that these results require that this April Tag has been mounted very precisely.

The recorded bags are then processed: The one recorded on the Duckiebot is processed by the container `analyze_rosbag` which extracts the Information about latency, update frequency of the nodes as well as the needed information published by the Duckiebot and saves it into json files. The one from the localization system is processed as explained in the classical offline localization system to extract the ground truth information about the whereabouts of the Duckiebot. The original post-processor just processes the apriltags and odometry from the bag and writes it in a processed bag. However, there exists a modified version which extract the relative pose estimation of the Duckiebot and write it directly in a .yaml file (More details are given below). The graph-optimizer then will extract the ground truth trajectories of all the Duckiebots that were in sight and store it in a yaml file. The data collected by the diagnostic toolbox does not need any processing and only needs to be downloaded from this link.

The processed data is then analyzed in the following notebooks: 1. 95-Trajectory-statistics.ipynb: This extracts all the information measured by the localization system and calculates all the things related to the poses of the Duckiebots. 2. 97-compare_calc_mean_benchmarks.ipynb: In this notebook it is analyzed if the measurements are stable enough such they can be actually compared to other results. There the repetition criteria is checked and if the standard deviation is too high the user is told to run another experiment to collect more data. As soon as the standard deviation is low enough the mean of all the measurements is calculated and then stored into yaml file that then can be compared to other results. 3. 96-compare_2_benchmarks.ipynb: This notebook serves to do the actual comparison between two Benchmark results or to do an analysis of your Benchmark (if there is no other one to compare to). It analyzes the performance based on many different properties and scores the performance. In the very end a final report can be extracted which summarizes the entire Benchmark.

So let me specify a couple of things:

- The environment definition includes the map design, the number of Duckiebots needed to run this Benchmark, as well as the light condition.

- The experiment definition includes information about how to run the experiment in which all the data is collected. So where to place the Duckiebots involved, what to do, when the Benchmark is terminated etc. is defined there. Also, the metrics used to actually score the behaviour as well as the actual scoring definition are defined.

- Repetition criteria defines which data we look at to check if the results are kind of stable and therefor reliable. This is to sort out lucky runs for example. The reliability of the data is judged based on the Coefficient of Variation.

- The Software preparation: As this Behaviour Benchmarking is there to actually test the software , most of this is depending on which SW the user wants to test. However, it also includes the preparation of the localization system etc. Moreover, to test a specific component/packages of dt-core, there is a prepared repository to simply add your modified package and build/run it within a specific version of dt-core. This works for all kind of contributions to dt-core that can be used in lane_following and/or indefinite navigation. Also in this package, one can lighten the launch file a bit in case the contribution is not very stable.
- The Hardware Check: is a docker container based on a simple python script that runs the user through a couple of questions and verifies that all is done in a correct fashion and all is prepared to be later fairly compared to other results.

Please note that the analysis is also possible if not all the data is collected, for example if the user does not record a bag on the Duckiebot or if the Diagnostic Toolbox was not running. At the moment the data recorded from the localization system is necessary to get an actual scoring in the end, however this can easily be adapted.

3) General Information

All the packages created for the Behvaiour Benchmarkin can be found here. This repository includes the following new contributions:

1. Packages:
   - 08-post-processing: Slightly modified version of the original post-processor which in addition to processing the bag also creates a yaml file including the relative lane pose estimations made by the Duckiebot.
   - analyze_rosbag: Package to process the bag recorded directly on the Duckiebot and creates several .json file including the needed information about: updated frequency of the nodes that are recorded, latency up to and including the detector node, global constants that were set, number of segments detected at each time and the pose (offset and heading) estimation of the Duckiebot.
   - hw_check: Package that guides the user through a checklist to make sure all the hardware is set up properly and to collect the information about the Duckiebot hardware used for the benchmarking.
   - light_lf: This package is in the end the same as the dt-core, however it allows the user to easily add his contribution and to lighten the launch file for lane following and indefinite navigation.
2. Notebooks:
   - 95-Trajectory-statistics: Analyzes the yaml file created by the graph-optimizer and the one created by the modified post-processor and extracts all kind of information related to the actual location of the Duckiebot.
   - 97-compare_calc_mean_benchmarks: Analyzes the yaml file created by the notebook 95, the json file downloaded from the dashboard, as well as all the json files created by the analyze_rosbag package. It checks the reliability and computes a measurement summary that can be compared to others in the notebook 97

- 96-compare_2_benchmarks: Analyzed the yaml files created by the notebook 97 of two different benchmarks of the same behaviour, or just analysis the performance of one of them if only one is available. It will result in a nice overview of the general performance.

Please note, that for each notebook there exists an example notebook that shows the results computed when the notebook is run with actual data.

4) Goal

The goal of this Behaviour Benchmarking is that for each behaviour and each major version (Ex: Master19, Daffy, Ente etc) there is a huge data set out of which the mean of all the measurements is calculated to build a stable and very reliable reference Benchmark. The developer can then compare its contribution to the results achieved by the global Duckietown performance.

At the moment the user is asked to upload its recorded bags to this link. However, this link should be changed as soon as possible as the storage is limited. The data storage should be automated anyways.

5) Future development

To design a Behaviour Benchmark in the future, all the needed packages and notebooks can be found in this repository, simply (fork and) clone it and then add your contribution. The readme file found in the repository corresponds to the set up and preparation of the lane following benchmark. Just copy it and adapt the sections that need to be changed. The most important sections that need to be adapted are the actual environment and experiment set up and the topics one has to subscribe to for the bag recorded on the Duckiebot.

- If needed adapt the the modified post-processor package by adding your code here, such that other things are extracted out of the bag then just the pose estimation of the Duckiebot for example.
- The Hardware check should remain the same.
- Adapt analyze_rosbag such that the additional information needed for your specific benchmark is extracted out of the bag recorded on the Duckiebot, by editing this file.
- Adapt the notebooks such that they analyze the things needed for your specific benchmark, however, it is strongly recommended to keep the same structure of first analyzing the measurements of the localization system, then check the consistency of the data and summarize it in one yaml file which then can be compared to other Benchmarks.

UNIT F-1

Lane Following Benchmark Introduction

1.1. Environment Definition

- 1 Duckiebot
- No natural light, just be white light coming from the ceiling
- Map: ‘linus_loop’

1.2. Experiment Definition

- Run lane following for 50 seconds
- Termination criteria:
  - Out of sight: 5 sec
  - Crash / Too slow: 15 sec per tile
- Repetition criteria:
  - Consistency in data
- Metrics:
  - Behaviour
  - Engineering data

Type of experiment is pretty obvious and is simply running lane following for 50 seconds. The termination criteria’s, next to just when the bag recording is done, are out of sight of the Duckiebot and crashes. This means that the Benchmark is stopped/shortened to the time where the Duckiebot was last seen if the Duckiebot was out of sight of all the Watchtowers for more than 3 seconds or if the Duckiebot took more than 15 seconds to cross an entire Tile. This is only the case if the Duckiebot crashes or if Lane Following really screws up. In the case of a crash the user is allowed to stop lane following to prevent any damages but he is not allowed to actually move the Duckiebot until the bag has finished its recording.

1.3. Mathematical Metrics Definitions

Below you can see the formulas applied to actually calculate the results.

Arithmetic mean:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Standard deviation (std):

$$s = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

Coefficient of variation (CV):

$$CV = std/mean * 100[\%]$$

1.4. Repetition Criteria

To be able to do a proper analysis of the performance, one has to be sure that the data is reliable. The different data measured was analyzed over a bunch of experiments, and it was concluded that the only data that needs be verified that it is stable is the one coming from the localization system. As you can see here, the data measured on the Duckiebot concerning the update frequency the lag etc. do not change significantly between different experiments of the same code. The same applies for the data measured on the diagnostic toolbox (CPU usage or memory usage etc). Of course the stability of the data is increased the more data there is, but one can clearly see that one gets a good idea of the performance computational wise from one measurement. On the other hand as one can see on the very last slide of the link above, the measurement coming from the localization system can show some weird behaviour. This means, that it can happen, that the trajectory measured by the localization system does not at all correspond to the actual performance of the Duckiebot. To avoid that such falsified data influences the scoring of the actual performance the standard deviation of the measurements of the localization system is checked to assure stable and reliable measurements.

A CV in the range of 20% normally stands for an acceptable standard deviation, this means that the std deviation compared to the mean does not very too much However in this Benchmarking, the purpose of looking at the CV factor is only to be sure that the experiment recorded was not a lucky or unlucky run, therefore, a CV below 100% is already acceptable.

Therefore, the repetition criteria of the lane following Benchmark is, that all the CV of the measurements that are based on the localization system are below 1.0 (=100 %).

To be more precise this means the following: For measurements like the mean of the absolute lane offset (distance and angle to center line) calculated of the localization system is not allowed to vary too much over the different experiment runs. Also properties like Speed, number of tiles covered etc should not have a too large standard deviation over all the experiment runs.

If one of these properties vary too significantly the user is tolled to run another experiment to stabilize the measurements and verify them. Also the user is asked to check for weird trajectory localization done by the localization system and sort this data out.

Like this it is avoided that the results are falsified by measurements that do not at all correspond to the actual performance of the users software.

1.5. Termination Criteria

The Benchmark is officially terminated after the 50 seconds are up. However when the Duckiebot is out of sight for more then 3 seconds or if the Duckiebot takes more then 30 seconds to get across 1 tile the Benchmark will be terminated early. This will be taken into account into the final Benchmark score. An early termination will not be considered as a failure but will just lead to a lower score.

1.6. Scoring

In the Benchmark we consider on one hand the actual performance of the code, which means the actual behaving, and on the other hand we consider the engineering data performance. The engineering data that was recorded during the Benchmark gives you an insight on the CPU usage, update frequency of the different nodes, the latency etc.

The metrics used here to generate a score are the following (please not that in brackets the priorities are noted, H = High priority, M = Medium priority and L = low priority):

1. Behaviour performance
   - Mean absolute distance of the Duckiebot to the center of the lane [m] (H)
   - Mean absolute heading offset of the Duckiebot compared to the reference heading [deg] (H)
   - Mean absolute difference between the calculated/estimated offset by the Duckiebot and the actual offset calculated/measured by the Watchtowers [m] (M)
   - Mean absolute difference between the calculated heading error by the Duckiebot and the actual heading error calculated by the Watchtowers. [deg] (M)
   - Number of crashes (number of early stops due to slow driving or crashes devided by the number of experiments ran) (H)
   - Mean time until termination [seconds] (L)
   - Mean time needed per tile [seconds/tile] (L)
2. Engineering data performance:
   - Mean latency (lag up to and including the detector node) [ms] (H)
   - Mean of the update frequency of the different nodes [Hz] (H)
   - Mean of the CPU usage of the different nodes in the dt-core container [%] (H)
   - Mean of the Memory usage of the different nodes in the dt-core container [%] (L)
   - Mean of the nr of Threads of the different nodes in the dt-core container (L)
   - Overall CPU usage [%] (H)
   - Overall Memory usage [%] (M)
   - Overall SWAP usage [%] (M)

The score then is calculated separately for the Behaviour performance and the Enginieering data performane, where the score is increased by +5 if the property has high priority, +3 if the property has medium priority and +1 if the property has low priority. The overall score is then simply the sum of the behaviour score and the engineering data score.

Please note that the localization of the Duckiebot that is measured by the watchtowers is with respect to the center of the April Tag that is placed on your Duckiebot. This means that all kind of measurements and results that talk about the position of the Duckiebot are refering to the center of the April Tag on top of the Duckiebot. This means that if this Apriltag is not placed very accurately, your results will be false.

Also note that during the final report produced at the very end you will see many different kind of results, also some of it which is not taken into account for the actual scoring. You can easily add a property to the scoring condition or change the priority of the property if you want to focus your score on something specific. This is simply done by changing the lists called: high_prio_beh, low_prio_beh, medium_prio_beh, high_prio_eng, low_prio_eng respectively medium_prio_eng.

1.7. General information

The Benchmarking is set up in a way such that up to to point 6 the procedure stays the same no matter what you are Benchmarking. Starting at point 6 there will be slight changes in the processing and analysis of the data depending on the Benchmark. However these changes are very small and the main changes are within the metrics etc. The goal of every Benchmark is to produce a final report that reports the results and compares them to another Benchmark of your choice. Ideally this benchmark it is compared to is the mean of all the Benchmarks ran all over the world of this type ran with the standard Duckietown software (for example the stable daffy commit of the containers `dt-core`, `dt-car-interface`, `dt-duckiebot-interface`, `dt-ros-commons`). However to be able to compare your Software with another one in any type of Benchmark, you first need to run at least 2 experiments. For each experiment there will be some recorded bags etc which then will be processed, analyzed and evaluated. The resulting evaluations of each experiment you run will then be again analyzed and for all the meaningful measurements, the means, medians and standard deviations are calculated. For each meaningful measurement the coefficient of variation is calculated and if this value is below 1 you get a green light to compute the final Benchmarking report. This means that you have to run at least to experiments and then start running the notebook that calculates the variation of your computed results after each new experiment. So the amount of experiments that need to be run depend on the stability of your results. As soon as you get a green light of all the important results, compute the mean of all the results over all the experiments you ran (at least two). With this .yaml file including all the means, you are finally ready to run the comparison/analysis of your results. This will then generate a nice report that analysis your solution as well as compares it to the results of another Benchmark you selected(can be any Benchmark ran of the same type). Based on the final report file you get at the end you can easily tell if your Software solution did improve the overall Performance or not and where your solution is better and where it is worse.

> **Note:** The procedure explained below runs the diagnostic toolbox, records a bag from the localization system, records a bag directly on the Duckiebot and at the same time the acquisition bridge. Running for example lane_following or indefinite_navigation whilst collecting all that data might not work well as there is not enough CPU. This is because the localization system is at the moment not very efficient. However, this will change in the near future and this issue will be solved.

Nevertheless, at the moment if your Duckiebot does whatever it wants when all is running and you are sure it is not because of your code, try to reduce the data recording to the essential. This means, first just don't run the diagnostic toolbox as this information is not the most crucial. If this still does not help, just record the bag from the localization system as this will give you at least some information about the actual performance of the behaving. In the case where you cannot record all the data, just ignore the according steps in the data analysis and complete the ones based on the bags you actually have. Experiment have shown that the data collected of the diagnostic toolbox do not really change from experiment to experiment, you can also run one experiment just running the diagnostic toolbox and then run this analysis on the data collected there. The same can be done with the bag that is collected on the Duckiebot.

So a possible procedure as long as the localization system is not very efficient to first run some experiments recording just a bag from the localization system, run one experiment running the diagnostic toolbox next to it and then one experiments recording the bag directly on the Duckiebot. The Notebooks analyzing the data can handle all kind of different data recording configurations. However, it is important to say that the more experiments that were run with your code the more reliable the data and therefore, the more accurate the performance scoring.

For each Notebook there is an Example notebook that shows the results/outputs achieved by the notebooks when running them with actual data.

Please note that if you don't have a localization system available, just ignore everything related to the localization system. Then within the notebooks, just upload the Duckiebot relative pose estimation .json file called `BAGNAME_duckiebot_lane_pose.json` created by the analyze_rosbag package and take these measurements as ground truth of the relative Duckiebot pose. In this case all the measurements about the speed, tiles covered per second etc won't be calculated. However you can still get a nice idea about your performance based on the estimations and the engineering data recorded.

Unit F-2
Lane Following - Procedure

2.1. Requires

- Requires: Duckiebot (correctly assembled, initialized and calibrated) in the version you want to Benchmark daffy, Master19 etc.)
- Requires: Localization standoff
- Requires: Autobot
- Requires: 2 straight and 4 curved tiles
- Requires: At least 4 Watchtowers in WT19B configuration
- Requires: X Ground April tags
- Results: Lane Following Benchmarking

2.2. Duckiebot Hardware set up

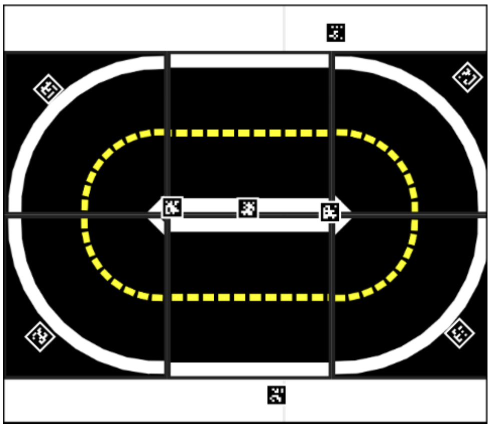
Below a few things you have to be careful with:

- Be careful with placing the April Tag in the very center of the DB as the localization system that measures the Ground Truth expects the April Tag to be exactly in the center! Future work potential!
- Make sure that standoff has an April Tag with a different name as your Duckiebot! So if you use the Autobot18 April Tag for example, make sure your duckiebot is named differently!
- Set the Gain of your Duckiebot to 1.0 by following the instructions found here

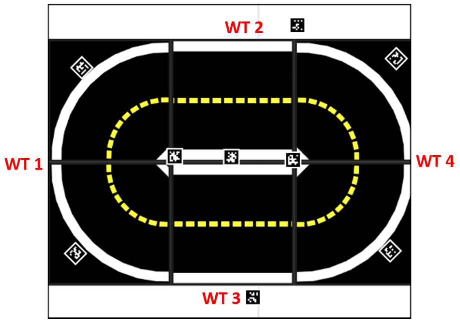
2.3. Loop Assembly and map making

Pleas note that the loop used for these benchmarks does NOT respect the Duckietown specifications and rules as it is not allowed to have two lanes next to each other without any visual barrier. However as all the experiments worked out fine and as the space was limited, the Benchmarks were done on this loop anyways. In the future it is strongly recommended to change the settings to a 3x3 loop that does respect the Duckietown specifications. This can be done without having to change anything else, as the result computed etc will work on any kind of loop.

First assemble the tiles to a 2x3 closed loop as seen below.



Further place the Watchtowers around the loop such that each part of it is seen by at least one Watchtower. To get really accurate results it is recommended to place them as seen below.



Please note that it is recommended to check what the watchtowers actually just see the loop. Therefor place them in a way such that they see as little from the outside of the loop as possible.

Fork and clone the Duckietown-world repository and follow the instructions found here to create your own map. Or simply add the linus_loop folder into the *visualization/maps* folder of your duckietown-world repository and the linus_loop yaml file into the *src/duckietown_world/data/gd1/maps* folder of your duckietown-world repository.

Then place at least 8 Ground April Tags roughly as seen in the images such that each watchtower sees at least 2 Apriltags. After follow the instructions found here to add their exact position to the map.

2.4. Localization Set up

Set up the offline localization following the instructions found here (Steps 3.1-3.5).

2.5. Software preparation

- Make sure your duckietown shell is set to the version `daffy` for the upcoming procedure, even when you flashed you Duckiebot on Master19. Run the command:
  - `dts --set-version daffy`

- On your local computer create a folder called `bag`
- (Fork and) clone the behaviour-benchmarking repository
- Be sure that `dt-core`, `dt-car-interface`, `dt-duckiebot-interface`, `dt-ros-commons` images are updated according to the version you are using.

If not, for daffy pull:

```
* `docker -H BOTNAME.local pull duckietown/dt-core:daffy-
arm32v7@sha256:4c7633c2041f5b7846be2346e0892c9f50987d2fd98d3479ec1a4cf378f52ee6`
* `docker -H BOTNAME.local pull duckietown/dt-car-interface:daffy-
arm32v7@sha256:e3db984157bf3a2b2d4ab7237536c17b37333711244a3206517daa187c143016`
* `docker -H BOTNAME.local pull duckietown/dt-duckiebot-interface:daffy-
arm32v7@sha256:94a9defa553d1e238566a621e084c4b368e6a9b62053b02f0eef1d5685f9ea73`
* `docker -H BOTNAME.local pull duckietown/dt-ros-commons:daffy-
arm32v7@sha256:20840df4cd5a8ade5949e5cfae2eb9b5cf9ee7c0`
```

If not, for Master19 pull:

```
* `docker -H BOTNAME.local pull duckietown/dt-core:daffy-
arm32v7@sha256:4c7633c2041f5b7846be2346e0892c9f50987d2fd98d3479ec1a4cf378f52ee6`
* `docker -H BOTNAME.local pull duckietown/dt-car-interface:daffy-
arm32v7@sha256:e3db984157bf3a2b2d4ab7237536c17b37333711244a3206517daa187c143016`
* `docker -H BOTNAME.local pull duckietown/dt-duckiebot-interface:daffy-
arm32v7@sha256:94a9defa553d1e238566a621e084c4b368e6a9b62053b02f0eef1d5685f9ea73`
* `docker -H BOTNAME.local pull duckietown/dt-ros-commons:daffy-
arm32v7@sha256:20840df4cd5a8ade5949e5cfae2eb9b5cf9ee7c0`
```

Please note, that you do not have to pull the specific tags above if you want to test a new version of any of the containers. However if you want to test a contribution to dt-core which you wrote (for example a new line_detector) it is recommended to pull the according images above.

- If all the images are at the right version you can start the following steps:

For daffy:

```
1. Make sure all old containers from the images `dt-duckiebot-inter-
face`, `dt-car-interface`, and `dt-core` are stopped. These containers
can have different names, instead look at the image name from which
they are run.

2. Start all the drivers in `dt-duckiebot-interface`:

    * `dts duckiebot demo --demo_name all_drivers --duckiebot_name BOT-
NAME --package_name duckiebot_interface --image duckietown/dt-duckiebot-
interface:daffy`


    and the glue nodes that handle the joystick mapping and the kinemat-
ics:

    * `dts duckiebot demo --demo_name all --duckiebot_name BOTNAME --
package_name car_interface --image duckietown/dt-car-interface:daffy`

    Make sure that this worked properly.

3. Within the folder _packages/light_lf_ of the behaviour-benchmarking
repository:
     1. You can **build** the docker container as follows:

        - `docker -H BOTNAME.local build --no-cache -t light_lf:v1 .`

     2. After that, if there were no errors, you can **run** the
light_lf:

        - `docker -H BOTNAME.local run -it  --name behaviour_benchmark-
ing --rm -v /data:/data --privileged --network=host light_lf:v1`
```

For Master19:

```
Follow the instructions found [here](https://docs.duckietown.org/DT19/
opmanual_duckiebot/out/demo_lane_following.html) to start lane follow-
ing, or the instructions found [here](https://docs.duckietown.org/DT19/
opmanual_duckiebot/out/demo_indefinite_navigation.html) to start the in-
definite navigation.
```

2.6. Add your contribution in daffy

To see if you contribution has improved the Lane following just add your contribution into the *packages/light_lf/packages* folder and build the container again:

- `docker -H BOTNAME.local build --no-cache -t light_lf:BRANCH_NAME .`

Then run your version of dt-core:

- `docker -H BOTNAME.local run -it --name behaviour_benchmarking --rm -v /data:/data --privileged --network=host light_lf:BRANCH_NAME`

For example, when you have worked one the lane_filter, then simply add your entire lane_filter folder into the folder *packages/light_lf/packages*. Please make sure that you use the precise name, as then the default version of whatever package is automatically replaced by yours. To get all the different packages in which you can make changes or work in please check here.

If you would like to run indefinite_navigation instead of just lane_following, just uncomment line 3 in the light_lf.launch file found in light_lf/packages/light_lf/launch folder and comment out line 2.

In the end it is the same as if you would simply clone the dt-core repository and building and running this on your Duckiebot. However, it is suggested to develop as you wish and then for the actual Benchmarking to use the way explained above which uses a lighter version of the launch files. This guarantees the benchmarks to be comparable.

2.7. Hardware Check:

First of all, for each Duckiebot involved, run the hw_check you can find within the cloned behaviour-benchmarking repository. Therefor, follow the following steps:

- Go into the folder hw_check by running:
  - `cd ~/behaviour-benchmarking/packages/hw_check`
- Build the container by running:
  - `docker -H BOTNAME.local build --no-cache -t hw_check:v1 .`
- Then run it by running:
  - `docker -H BOTNAME.local run -it --network host -v /data:/data -v /sys/firmware/devicetree:/devfs:ro hw_check:v1`

Then follow the instructions within the terminal.

- When the Docker Container has finished, visit: `http://BOTNAME.local:8082/config` and download the .yaml file with your information in the name.
- Place the .yaml file within the *data/BenchmarkXY* folder of your behaviour-benchmarking repository.
- Furthermore, it is suggested that you set up your own autolab fleet rooster. However, this is not necessary.

2.8. Duckiebot preparation:

To be able to record a rosbag on your Duckiebot please follow the steps below. This rosbag records all messages publisher to the for the specific Benchmark important nodes. The steps below prepare Terminal 4 of the four terminals mentioned below. Please note that the rosbag that will be recorded will automatically be saved on your USB drive.

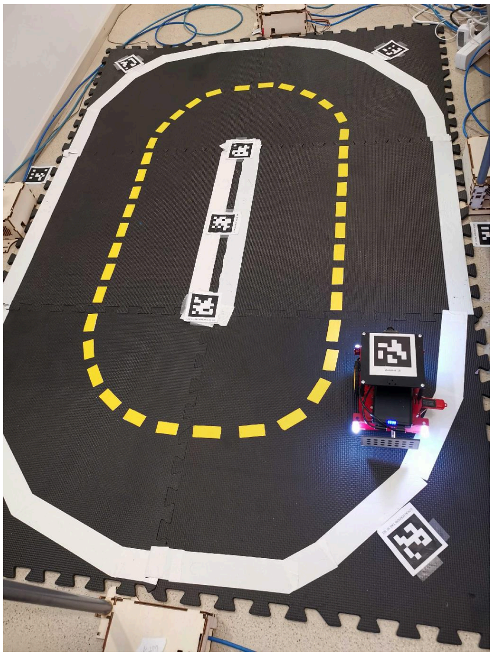
- Plug a USB drive (of at least 32 Gb) into your Duckiebot
- Ssh into your Duckiebot by running:
  - `ssh AUTOBOT_NAME`
  - Within your Duckiebot, create a folder with the name `bag` by running the command:

  - `sudo mkdir /data/bag`
  - Use the command `lsblk` to see where your USB drive is located. Under name you should see sda1 or sdb1 and the size should be a bit less then the actual size of your drive (about 28.7 Gb for a 32 Gb drive)
  - Then mount the folder created above to your USB drive by running:
  - `sudo mount -t vfat /dev/sdb1 /data/bag/ -o uid=1000,gid=1000,umask=000`
  - Then exit your Duckiebot by pressing `Crtl + d`
- Then start and enter a container on your Duckiebot by running:
  - `dts duckiebot demo --demo_name base --duckiebot_name AUTOBOT_NAME`
- Then prepare the command to record a rosbag within that container:
  - `rosbag record -O /data/bagrec/BAGNAME_duckiebot.bag --duration 50 /AUTOBOT_NAME/line_detector_node/segment_list /AUTOBOT_NAME/lane_filter_node/lane_pose /rosout`

Please note that if you are using Master19 you should subscribe to `/AUTOBOT_NAME/lane_controller_node/lane_pose` instead of `/AUTOBOT_NAME/lane_filter_node/lane_pose`

2.9. Place your Duckiebot within the map:

Place your Duckiebot in the beginning of the straight part that is *on the bottom of the loop* relative to the origin of the coordinate system (see image below) of the loop on the outer line such that the Duckiebot drives around the the loop counter-clockwise. Below you see an image on where you should place the Duckiebot, in this image the origin of the global coordinate frame of the map (so the point from where you measured the distances of the April Tags when you added them to the map) lies in the bottom right (in the corner near the Duckiebot).



2.10. Prepare 4 terminals:

Make sure that you carefully read through the steps and the `note` section below before starting up everything. It is important that you start recording the bags at roughly the same time and press `a` to start lane following or indefinite navigation straight after. Therefor first starting the diagnostic toolbox, then start recording both the bags and straight after start lanefollowing by pressing ‘a’.

- Terminal 1: Run the diagnostic toolbox on your Duckiebot:
  - `dts diagnostics run -G Name_BehBench_LF -d 70 --type duckiebot -H BOT-NAME.local`
- Terminal 2: Start the keyboard control on your Duckiebot:
  - `dts duckiebot keyboard_control BOTNAME` To start lane_following press ‘a’ on your keyboard
- Terminal 3: Open a Docker container ros being pre-installed by running the command below or record a rosbag directly on your computer if you have the necessary setup installed:
  - `dts cli`

Then within this container record a rosbag that subscribes everything published by the localization system by running:

  - `rosbag record -a --duration=50 -O BAGNAME_localization.bag`
- Terminal 4: Run the command already prepared above to record a rosbag that subscribes to the needed topics.
- Note:
  - If your Duckiebot crashes, you can stop lane following by pressing `s`, but please

let the recordings of the bags as well as the Diagnostic Toolbox finish normally.

- If your Duckiebot leaves the loop, make sure that you do **not** stop lanefollowing until the Duckiebot was out of sight of the watchtowers for at least 3 seconds **or** just go and grab the Duckiebot when it left the loop and take it completely out of sight manually, then stop the lanefollowing.
- For the BAGNAME please use the following convention:
- `Country_Autolab_LoopName_Date_GithubUserName_HWConfig_SWConfig`
- Where:
  - `Country`: is the country you ran your benchmark (Ex. CH, USA, DE etc.)
  - `Autolab`: is the Autolab you ran your benchmark in (Ex. ETHZ etc.)
  - `LoopName`: is the name of the loop on which you recorded your benchmark, at the moment this should be `linus_loop`
  - `Date`: is the date on which you ran the benchmark in the format DDMMYEAR(Ex. 17022020)
  - `GithubUserName`: is your github username (Ex. duckietown)
  - `HWConfig`: is the hardware configuration of the Duckiebot you used (Ex. DB18, DB19 etc.)
  - `SWConfig`: is the software configuration used on the Duckiebot (Ex. Daffy, Master19 etc.)

2.11. File gathering:

After the rosbag recording as well as the Diagnostic Toolbox have finished you can stop the Duckiebot by pressing 's' on your keyboard. Then do the following steps:

- Exit the container of Terminal 4 by pressing: `crt+d`
- Ssh into your Duckiebot again by running:
  - `ssh AUTOBOT_NAME`
- Within your Duckiebot, unmount the folder by running:
  - `sudo umount /data/bag`
- Then remove the USB drive from your Duckiebot and plug it into your local Computer. Copy the `BAGNAME_duckiebot.bag` that should be on your USB drive into the folder `bag` on your local computer.
- Copy the recorded rosbag of the localization system from the Docker container onto your local computer into the `path_to_bag_folder` (should be simply `bag`) by running:
  - `sudo docker cp dts-cli-run-helper:/code/catkin_ws/src/dt-gui-tools/BAGNAME_localization.bag ~/path_to_bag_folder` or generally:
  - `sudo docker cp Docker_Container_Name:/place_within_container/where_bag_was_recorded/BAGNAME_localization.bag ~/path_to_bag_folder`
- Make sure that both the bags are readable by opening the `bag` folder in a terminal and running:
  - `sudo chmod 777 BAGNAME_localization.bag`

  - `sudo chmod 777 BAGNAME_duckiebot.bag`
- To get the information recorded by the diagnostic toolbox, visit dashboard and login using your Duckietown token. Then navigate to *Diagnostics* and in the drop down menu *Group* select *Name_BehBench_LF* and in the drop down menu *Time* the corresponding time when you ran the Benchmark. After add the data by pressing onto the green plus and download the *.json* file by pressing the Download log button.
- Place the download .json file within you `bag` folder and rename it to `BAGNAME_diagnostics.json`.
- To help the Duckietown community to gather the logs of the recorded bags, please create a folder, named BAGNAME, containing the two bag files as well as the .json file. Make zip of this folder and upload it to the bag folder found under this link.

2.12. Processing the recorded bags:

You need to know where your bag is. The folder containing it is referred as `path_to_bag_folder` in the command below. It is recommended to create new separate folders for each Benchmark (with date and/or sequence number). If you followed the instructions above, your bags are located in the folder `bag`. Example for `path_to_bag_folder` is /home/linus/bag.

- Cd into the package `08-post-processing` found in your `behaviour-benchmarking` repository by running:
  - `cd behaviour-benchmarking/packages/08-post-processing`
- Then build the repository by running:
  - `docker build -t duckietown/post-processing:v1 .`
- Then run the post_processor for the rosbag of the localization system by running:
  - `docker run --name post_processor -it --rm -e INPUT_BAG_PATH=/data/BAGNAME_localization -e OUTPUT_BAG_PATH=/data/processed_BAGNAME_localization.bag -e ROS_MASTER_URI=http://192.168.1.97:11311 -v path_to_bag_folder:/data duckietown/post-processing:v1`

This runs a slightly modified version of the original found here. (To run the original use the following command:

  - `docker run --name post_processor -dit --rm -e INPUT_BAG_PATH=/data/BAGNAME_localization -e OUTPUT_BAG_PATH=/data/processed_BAGNAME_localization.bag -e ROS_MASTER_URI=http://YOUR_IP:11311 -v PATH_TO_BAG_FOLDER:/data duckietown/post-processor:daffy`) Note that when running the original post-processor there won't be a file created called `BAGNAME_db_estimation.yaml` which is necessary for the some of the Benchmarks (Ex. Lane Following)

When the container stops, you should have a new bag called `processed_BAGNAME_localization.bag` as well as a new .yaml file called `BAGNAME_db_estimation.yaml` inside of your `path_to_bag_folder`. (This can take more than a minute, please be patient)

  - Make sure that those files are readable by opening the `path_to_bag_folder` in a terminal and running:
    - sudo chmod 777 BAGNAME_db_estimation.yaml
  - Then place the file called `BAGNAME_db_estimation.yaml` into the folder `~/behav-`

`iour-benchmarking/data/BenchmarkXY/yaml/post_processor`.

- Remember from Unit B-4 - Autolab map, that you created a map. Now is the time to remember on which fork you pushed it (the default is duckietown), and what name you gave to your map (for this Benchmark this should be `linus_loop`). The map file needs to be in the same folder as the rest of the maps. They are respectively the YOUR_FORK_NAME and YOUR_MAP_NAME arguments in the following command line. Please run the graph-optimizer by running:
  - `docker run --rm -e ATMSGS_BAG=/data/processed_BAGNAME_localization.bag -e OUTPUT_DIR=/data -e ROS_MASTER=YOUR_HOSTNAME -e ROS_MASTER_IP=YOUR_IP --name graph_optimizer -v path_to_bag_folder:/data -e DUCKIETOWN_WORLD_FORK=YOUR_FORK_NAME -e MAP_NAME=YOUR_MAP_NAME duckietown/cslam-graphoptimizer:daffy`

This will generate at least one *.yaml* file that will be stored in the folder `path_to_bag_folder`. If you followed the instructions and placed an April Tag with a different name than you Duckiebot on your localization standoff, you should find two *.yaml* files. One will be named like your Duckiebot, and the other one like the name of the April Tag on you Duckiebot (Ex. autobot01.yaml). For the benchmarking we are only interested in the .yaml file that has the same name as the April Tag on top of your Duckiebot has.

  - Make sure that those files are readable by opening the `path_to_bag_folder` in a terminal and running:
    - `sudo chmod 777 APRILTAGID.yaml`
  - Then place this file in the folder `~/behaviour-benchmarking/data/BenchmarkXY/yaml/graph_optimizer`

- For the rosbag recorded on the Duckiebot, run analyze-rosbag by:
  - cd into the `analyze_rosbag` folder found in behaviour-benchmarking repository by running `cd behaviour-benchmarking/packages/analyze_rosbag`
  - Build the repository by running:
    - `dts devel build -f --arch amd64`
  - Then run it with:
    - `docker run -v path_to_bag_folder:/data -e DUCKIEBOT=AUTOBOT_NAME -e BAGNAME=BAGNAME_duckiebot -it --rm duckietown/behaviour-benchmarking:v1-amd64`
  - This will create five `.json` files within the `bag` folder that will be used for the Benchmarking later. The *.json* files are named:
    - `BAGNAME_duckiebot_constant.json`: containing the value of each of the constants that was used for the experiment
    - `BAGNAME_duckiebot_lane_pose.json`: containing the information about the relative pose estimation of the Duckiebot
    - `BAGNAME_duckiebot_node_info.json`: containing information about the update frequency of the different nodes, the number of connections etc)
    - `BAGNAME_duckiebot_segment_count.json`: containing information about the number of segments detected at each time stamp

  - `BAGNAME_duckiebot_latencies.json`: contains information about the latency measured from the very beginning up to and including the detector node
- Make sure that those files are readable by opening the `bag` folder in a terminal and running:
  - `sudo chmod 777 FILENAME.json`
- Then place all those files in the folder `~/behaviour-benchmarking/data/BenchmarkXY/json`

2.13. Result analysis preparation:

- Place the .yaml file created by the graphoptimizer with the name of the April Tag that is on top of your Duckiebot into the *data/BenchmarkXY/yaml/graph_optimizer* folder (please note that it is important that you take the correct .yaml file as the one named after your actual Duckiebot should **not** be placed within the mentioned folder). Then place the .yaml file created by the post_processor called `BAGNAME_db_estimation.yaml` into the *data/BenchmarkXY/yaml/post_processor* folder. Also place all the .json files (the one downloaded from the dashboard as well as the 5 created by the analyze_rosbag container) into the *data/BenchmarkXY/json* folder of your behaviour_benchmarking repository.
- Note that XY stands for a number, so for your first Benchmark name the folder *Benchmark01*.
- Create a virtual environment as you already did for when you added the map to your duckietown-world repository or when you added the exact position of the ground april tags. However, this time, please create this virtual environment within your cloned behaviour_benchmarking repository by following the instructions below:

  a. First, if not already done, install venv by running:
     - `sudo apt install -y python3-venv`
  b. Then, cd into your behaviour_benchmarking repository, and create the venv:
     - `cd ~/behaviour_benchmarking`
     - `python3.7 -m venv duckietown-world-venv`
     - `source duckietown-world-venv/bin/activate`
  c. Now, you can setup duckietown-world. Inside of the virtual environment (you should see "(duckietown-worl-venv)" in front of your prompt line), please run:
     - `python3 -m pip install --upgrade pip`
     - `python3 -m pip install -r requirements.txt`
     - `python3 -m pip install jupyter`
     - `python3 setup.py develop --no-deps`
  d. Then start the notebook:
     - `jupyter notebook`

  If you encounter any issues with the steps above, please click here for more detailed instructions.

  e. Navigate to the notebooks

2.14. Result computation:

In the following it is briefly explained how to achieve some actual results based on the recorded data. The first two steps have to be repeated for each experiment you ran for this Benchmark. They will extract and analyze the engineering data as well as the actual behaviour. The further steps then will help you decide weather you ran enough experiments or not and then finally let you compare your Benchmark to others. Please note, that detailed explanation on what exactly is calculated and how it is analyzed is explained in the notebooks directly. So if you are interested in more detail what exactly happens, please have a detailed look at them and read the comment when running them. Within the final report which will be the end result, you will also fined some detailed information on what is taken into account ect. There are some instructions within the notebooks, however, they are mostly designed to be ran without having to do much, so the following steps won't take long. But please read the instructions carefully. If you followed the instructions above you ran at least two experiments. Please run for each experiment you have already done the first steps before continuing.

- First open the notebook called `95-Trajectory-statistics` and follow the instructions there. To run this Notebook, you will need the file called `AutobotAPRILTAGNB.yaml` found in the folder `~/behaviour-benchmarking/data/BenchmarkXY/yaml/graph_optimizer` (where APRILTAGNR is the number of the April Tag that is placed on top of your Duckiebot) and the file `BAGNAME_db_estimation.yaml` found within the folder `~/behaviour-benchmarking/data/BenchmarkXY/yaml/post_processor`. It will result in a .yaml file called BAGNAME__benchmark_results_test_XY.yaml where XY is the number of the test run. In this file you will find all kind of results considering the actual performance of the behaviour. The file will be stored within the folder `~/behaviour-benchmarking/data/BenchmarkXY/benchmarks/same_bm` and will be further analyzed below. This notebook extracts all kind of data measured by the localization system like the actual trajectory, the absolute mean lane offset of the Duckiebot, the number of tiles the Duckiebot covered etc. For more details on what exactly is analyzed and how the analysis is done, please have a look at the Notebook itself. However, below see below for a list of all computed measurements:
  - The mean of the offset (distance and angle) of the Duckiebot in respect to the center of the lane [m]
  - The number of rounds completed (entirely completed by the center of the April Tag placed on the localization standoff on your Duckiebot
  - The number of tiles covered (center of April Tag completely passed the tile)
  - Avg time needed per tile in seconds
  - Length of the Benchmark in seconds
  - Actual length of the benchmark in seconds
  - Mean absolute lane offset measured by Watchtowers (ground truth) [m]
  - Std of the absolute lane offset measured by Watchtowers (ground truth) [m]
  - Mean absolute relative angle measured by Watchtowers (ground truth) [deg]
  - Std of the absolute relative angle measured by Watchtowers (ground truth) [deg]
  - Mean absolute lane offset measured by the Duckiebot [m]
  - Std of the absolute lane offset measured by the Duckiebot [m]

- Mean absolute relative angle measured by the Duckiebot [deg]
- td of the absolute relative angle measured by the Duckiebot [deg]
- Mean of the absolute difference between lane offset measured by the Duckiebot and by the Watchtowers (ground truth) [m]
- Std of the absolute difference between lane offset measured by the Duckiebot and by the Watchtowers (ground truth) [m]
- Mean of the absolute difference between the relative angle measured by the Duckiebot and by the Watchtowers (ground truth) [deg]
- Std of the absolute difference between the relative angle measured by the Duckiebot and by the Watchtowers (ground truth) [deg]

- Now it is time to see if you have collected enough data, for this, please open and run the notebook `97-compare_calc_mean_benchmarks`. This will open all your result .yaml files you produced above and check if the data is meaningful. This means that it calculates the standard deviation of some of the measurements over the different experiments and puts it in relation with the mean. If the standard deviation for all of the considered measurements is small enough it will then produce a `BAGNAME_benchmark_final_results.yaml` file which includes the mean values over all the experiments ran and saves it in the folder `~/behaviour-benchmarking/data/BenchmarkXY/benchmarks/final`. The Notebook produces some nice visualizations that show the user if its data is stable enough or not and why it is important to have stable data. The user literally gets a green or a red light weather he is ready to actually run the evaluation of the performance or if more data needs to be collected. If the standard deviation is too high, please run another experiment, complete the first step of the result computation and run this notebook again. If not, please upload the resulting yaml file in the `Results` folder found here The .yaml file produced holds the following information:
  - Software information of all the containers like: container name, image name and tag, the base image of the container, the architecture of the container, the branch etc. as well as the constants that were set within the Duckiebot for example the gain, the trim factor etc. These things do not change within the same Benchmark, this means for all the tests you are running with the specific software version all this information remains the same. Therefore, this data is called `static`.
  - Engineering data analysis like the update frequency of the different nodes, the number of segments detected over time, the latency up to and including the `detector_node`, as well as the total overall performance information (CPU usage, the Memory usage and the NThreads). Moreover it includes total performance information of each node of container dt-core. This data changes (at least slightly) between two different tests of the same Benchmark which is why the mean of this data of all the tests ran for one Benchmark is calculated later.
  - Engineering data analysis like the update frequency of the different nodes, the number of segments detected over time, the latency up to and including the `detector_node`, as well as the total overall performance information (CPU usage, the Memory usage and the NThreads). Moreover it includes total performance information of each node of container dt-core. This data changes (at least slightly) between two different tests of the same Benchmark which is why the mean of this data of all the tests ran for one Benchmark is calculated later.

  - Number of experiments ran
  - Run-times of different experiments
  - Info about if enough data collected
  - Analysis of all data from Notebook 95 (Mean, Median, Std, CV etc.)
  - Information about the trajectories

- Then you are finally ready to compare your Benchmark with another one of the same type. For this please run the notebook ´ 96-compare_2_benchmarks´. This notebook will guide you through the analysis and show you the comparison of the two Benchmarks you are comparing. In there you find a nice summary of all the measured results, the metric used and the final results. Please have a look at the notebook 96 example for further details about what the final report includes.

1) Test the code stability

You can test the stability of your code when running some experiments under some specific conditions and comparing the result to the original one. For example you can cover the right white line, or the yellow middle line etc.

2.15. Future work

- Improve the out_of_sight condition such that the Watchtowers can still see the Duckiebot but it counts as out of sight if he is on a tile which is not part of the loop for longer then 5 seconds.
- Also improve the offset calculation and punish the score of the Benchmark if the Duckiebot takes for example a huge shortcut, or give a bigger punishment if he drives on the other lane for too long
- improve the calculation of time needed per tile and make a difference between straight, left and right curved tiles
- save the data online automatically (not up to user) -> Automate data storage
- Automate the Benchmarks. This can be done as soon as the localization system works more reliable and efficient.
- set up remaining Benchmarks. This should not take long at all.
- Set up rest of Benchmarks
- Data collection
- Calculate position of center of mass based on position of April Tag
- Spit out type of tile where Benchmark was stopped in case of an early termination

UNIT F-3
Benchmarks for other Behaviours

3.1. Distance keeping

Prepare two Duckiebots, one dummy and one that is used for the testing. Then set the gain of the ‘dummy’ Duckiebot to 0.6 and make sure the gain of the autobot is at 1.0. Place the two Duckiebots behind each other and and start lane_following on both of them. Let them drive for half a round and then increase the gain of the ‘dummy’ Duckiebot to 0.8. Half a round later increase the gain to 1.0. During this time record the bags on both the Duckiebot and the localization system. For the Duckiebot please record the following nodes:

- /AUTOBOTNAME/vehicle_avoidance_control_node/car_cmd
- /AUTOBOTNAME/vehicle_avoidance_control_node/switch
- /AUTOBOTNAME/vehicle_avoidance_control_node/vehicle_detected
- /AUTOBOTNAME/vehicle_detection_node/detection
- /AUTOBOTNAME/vehicle_detection_node/detection_time
- /AUTOBOTNAME/vehicle_detection_node/switch
- /AUTOBOTNAME/vehicle_filter_node/switch Note that the Duckiebot used as a ‘dummy’ that drives in front does not need to have the acquisition bridge running. Then with the bags recorded run the notebook that analysis the engineering data, this is compatible for any kind of benchmark. Within the notebook 95 add some analysis that compares the distance between the two Duckiebots to the reference distance the Duckiebot in the back was supposed to keep. This can be done very easily by comparing the relative pose of the two Duckiebots. At the same time analyze the calculation the Duckiebot in the back has made and compare its estimated relative distance to the ground truth measured by the watchtowers. Also check how well lane following is doing when having a Duckiebot in front. Please note that you will have to add some lines into the lane_following.launch file to start the vehicle detection.

As a metrics use: for the engineering data the same as for Lane Following, and for the benchmarking the mean difference between the actual distance and the reference distance, the mean difference between the estimated distance by the Duckiebot and the actual distance measured by the watchtowers.

3.2. Red line detection

3.3. LF with Vehicles on same lane

3.4. LF with Vehicles on opposite lane

3.5. LF with Intersection navigation

3.6. Intersection Navigation

3.7. Vehicle Avoidance

3.8. LED detection

3.9. Overtaking

3.10. April Tag detection

Part G
Hardware Benchmarking

In this section the setup and usage of the hardware benchmark infrastructure as well as the benchmark procedure will be explained. In order to facilitate the benchmarking procedure a CLI tool is provided, automating the process.

Contents

Unit G-1
Introduction

In this section the Hardware Benchmark used in my Bachelor Thesis will be introduced. The associated repos are the following:

- Duckietown shell used for the Benchmark CLI tool
- Benchmark Backend, REST API to process data
- Benchmark Frontend, display results

Unless developing, the repos aren't needed as everything is installed via a docker container or the dts.

Contents

1.1. Motivation

In order to assess performance of different software and hardware configurations a standardized benchmarking procedure is needed. Such a benchmark helps do detect inefficiencies as well potential defects in the hardware and will help to troubleshoot nonfunctional Duckiebots. In the development of new software it is crucial to have performance metrics in order to compare the new release against the status quo.

1.2. Experiment

The benchmark uses a standardized sequence of commands and docker images in order to benchmark the performance of the Duckiebot.

1) Material

In order to perform a benchmark the following material is needed:

- Duckiebot to be tested
- A 3x3 Loop (4 Curves, 4 straights)
- optionally 4 Watchtowers and apriltags

2) General procedure

1. start diagnostics
2. start lane following
3. record a bag containing latency information
4. process data and compare against an overall average

UNIT G-2
Procedure

Author: Luzian Bieri

Maintainer: Luzian Bieri

Contents

2.1. Environment

The environmental requirements are minimal. Use at minimum a 3x3 Loop without intersections. Ensure that the map is exclusively illuminated from the ceiling, block all light coming from the side. Use a dark, neutral surrounding for all tiles in order to no disturb the lane detection.

2.2. Setup

If a setup (especially the Duckiebot) is already prepared, **you might continue with the next chapter**. See 2.2.2. *Please mind that the hardware benchmark results aren't fully comparable, as the benchmark aims to compare different hardware setups using the same software.*

1) Duckietown shell

If not installed yet install the newest version of dts (duckietown shell) via the instructions provided by the docs Currently we are using a custom stack of the duckietown shell commands, which fix all software to fixed versions to ensure comparability among different hardware configurations. As they are not in the original repo (yet) they have to be cloned from this fork. We are interested in the benchmarking branch.

Installing:

Use the following commands in a directory of your choice, recommended `/home/user/Documents/benchmarking`.

```
$ cd /path/to/commands/dir
$ git clone git@github.com:lujobi/duckietown-shell-commands.git
$ git checkout benchmark
```

Export the path to the local version of duckietown shell commands

```
$ export DTSHELL_COMMANDS=/path/to/commands/dirduckietown-shell-commands/
```

Set dts to any version:

```
$ dts --set-version RELEASE
```

In order to test whether the installation was successful enter `dts`. The output should look something like:

```
INFO:dts:Commands version: daffy
INFO:dts:Using path '/path/to/commands/dir/duckietown-shell-commands/'
as prescribed by env variable DTSHELL_COMMANDS.
INFO:dts:duckietown-shell-commands 5.0.2
INFO:duckietown-challenges:duckietown-challenges 5.1.5
INFO:zj:zuper-ipce 5.3.0
```

Enable the benchmark command via:

```
$ dts install benchmark
```

Add your dts token via:

```
$ dts tok set
```

Explanation benchmark commands:
The branch benchmarking fixes all software versions to a specific one in order to ensure reproducibility. Additinally the command group benchmark is added. This is used to specify the version of benchmarking software.

2) Duckietown world

In principal this step can be skipped as well if no localization system is around. Please continue with 2.2.3.

Tiles:
For the first benchmark we need the "normal" 3x3 circle circuit. Please ensure that the tiles are cleaned and assembled to specifications. Especially make sure that the road has the correct width.

Lighting:
In order to ensure reproducibility use a well illuminated room. Ensure that the light comes down from the ceiling, such that the bots are not dazzled. **Ensure that no natural light hits the bot.**

Watchtowers:
For the 3X3 circuit use 4 watchtowers in the middle of the circuit. One per corner. Ensure a proper connection and that the watchtowers are close to the corners.

Setup
As of now we use the standard procedure of setting up a loclization system and setting up the the watchtowers. **Thus make sure to use a new Terminal where the `export DT-`**

SHELL_COMMANDS has not been executed. Detailed instructions can be found here. All in all use the following command to initialize the sd card:

```
$ dts init_sd_card --hostname WT_HOST_NAME --linux-username mom
--linux-password MomWatches --country COUNTRY --type watchtower --ex-
perimental
```

Calibration

Use the same calibration procedure as for a standard duckiebot. **But only the intrinsic part.** Instructions.

Starting the camera demo:

```
$ dts duckiebot demo --demo_name camera --duckiebot_name
WT_HOST_NAME --package_name pi_camera --image
```

duckietown/dt-core:daffy

Start the calibration:

```
$ dts duckiebot calibrate_intrinsics WT_HOST_NAME
```

Start collecting data for the calibration. Press on the calibrate button as soon as all bars are green. Click Commit and check under `WT_HOST_NAME.local:8082/data/config/calibrations/camera_intrinsic/` that a file named `WT_HOST_NAME.yaml` exists.

World:

Use the instructions found here to set up the jupyter notebook in order to generate a new map. Note you will have to create your own fork of the duckietown-world. Make sure to leave the repo name as is!

Map

The yaml for a loop with floor around the map and in the center is the following:

```
tile_size: 0.585
tiles:
- - floor
  - floor
  - floor
  - floor
  - floor
- - floor
  - curve_left/W
  - straight/W
  - curve_left/N
  - floor
- - floor
  - straight/S
  - floor
  - straight/N
  - floor
- - floor
  - curve_left/S
  - straight/E
  - curve_left/E
  - floor
- - floor
  - floor
  - floor
  - floor
  - floor
```

Verify the that the map is displayed in the notebook under the name you gave to the file.

Apriltags

We use 4 ground Apriltags placed outside of each corner, moved in 10 cm from both borders. Use this link in order to fill the map with april tags. As such use the command.

```
$ python3 src/apriltag_measure/measure_ground_apriltags.py
MAP_NAME
```

Ensure that you enter your measurements **in Meters**. If you go back to the notebook you should see your map now rendered correctly with Apriltags.

Localization:

In order to set up the basic localization use the following commands on every watch-tower:

```
$ docker -H WT_HOST_NAME.local rm -f dt18_03_roscore_duckiebot-
interface_1


$ docker -H WT_HOST_NAME.local pull duckietown/dt-duckiebot-in-
terface:daffy-arm32v7

$ docker -H WT_HOST_NAME.local run --name duckiebot-interface --
privileged -e ROBOT_TYPE=watchtower --restart unless-stopped -v /da-
ta:/data -dit --network=host duckietown/dt-duckiebot-interface:daffy-
arm32v7
```

It might be that the last command fails. Use portainer to remove the `dt-duckiebot-interface`-Container.

Start the acquisition-bridge on all watchtowers:

```
$ docker -H WT_HOST_NAME.local run --name acquisition-bridge --
network=host -e ROBOT_TYPE=watchtower -e LAB_ROS_MAS-
TER_IP=YOUR_PC_IP -dit duckietown/acquisition-bridge:daffy-arm32v7
```

3) Duckiebot

Hardware Setup:

Assemble the Duckiebot as prescribed in the manual of the respective version of the Kit. Ensure that no wires are touching the wheels, or hinder the benchmark in any other way. Clearly mark the different Duckiebots. Add an Apriltag to your Duckiebot and enter the name below. All in lowercase without whitespaces.

Ensure that no hardware gets mixed between different configurations. Otherwise the whole benchmark will be invalidated.

Init SD Card:

This procedure only works if the special dt-shell-commands is installed. otherwise Proceed with the normal setup.

Decide which software version (e.g. master19, daffy, ente) you want to run. Set the benchmark version to said software version using dts:

```
$ dts benchmark set [!SOFTWARE_VERSION]
```

If the set version should be checked use the following command:

```
$ dts benchmark info
```

Use the init_sd_card command as known. (Some options which could change the software version are disabled) Use said hostname: ![HOST_NAME]

```
$ dts init_sd_card --hostname HOST_NAME --country CH --wifi [!NET-
WORKS]
```

If you are using a 16GB SD-Card, use the `--compress` flag.

4) Initial Setup

If you have a complete set up and calibrated bot you might want to move to the next Unit G-3

After the `init_sd_card` procedure ist over, take any charged battery (which doesn't belong to one of the bots to be tested) and plug the Duckiebot in. After some time the bot should be pingable, then ssh-ing into it should be possible.

Portainer and compose:

Open `HOST_NAME.local:9000` in a browser. As soon as portainer is running, there should be 4 containers one of which is not running (`duckietown/rpi-duckiebot-dashboard`), start that one via portainer. After a short time `HOST_NAME.local` should be reachable. Further progress of the installation can be see there after skipping the login. Finished setting up, enter the your duckietown-token.

Verification TODO delete:

Use this command to test the setup

```
$ dts duckiebot keyboard_control HOST_NAME
```

and this command to test the camera:

```
$ dts start_gui_tools HOST_NAME
```

then use:

```
$ rqt_image_view
```

this should display the live camera feed.

Calibration:

To calibrate the bot we use the same command as is used in the docs.

Camera intrinsic

As the `duckiebot-interface` should be already running use the following command:

Daffy

```
$ dts duckiebot demo --demo_name camera --duckiebot_name HOST_NAME --pack
age_name pi_camera --image duckietown/dt-
core:daffy@sha256:4c7633c2041f5b7846be2346e0892c9f50987d2fd98d3479ec1a4cf378f
```

Master19

```
$ dts duckiebot demo --demo_name camera --duckiebot_name HOST_NAME --image duc
etown/rpi-duckiebot-base:mas-
ter19@sha256:80f23a1835e6b3f9d2606aae54ce824dd13085e3e7491e87c7f0216797964b5c
```

Then run the calibration using the command:

Daffy

```
$ dts duckiebot calibrate_intrinsics HOST_NAME --base_image duckietown/
dt-core:daffy-
amd64@sha256:d72e8a8c3191c146ecc2a812bdd036aaf15076e6c1cb9467304e0e54f9a39a10
```

Master19

```
$ dts duckiebot calibrate_intrinsics HOST_NAME --base_image ducki-
etown/rpi-duckiebot-base:master19-no-arm@sha256:bce-
fefe0e249b8b1fcde21b3eaa6c7af5737fbf92003854376be9541a97257a2
```

info it may take some time until the image is completely downloaded

Camera extrinsic

Then run the calibration using the command:

Daffy

```
$ dts duckiebot calibrate_extrinsics HOST_NAME --base_image duckietown/
dt-core:daffy-
amd64@sha256:d72e8a8c3191c146ecc2a812bdd036aaf15076e6c1cb9467304e0e54f9a39a10
```

Master19

```
$ dts duckiebot calibrate_intrinsics HOST_NAME --base_image ducki-
etown/rpi-duckiebot-base:master19-no-arm@sha256:bce-
fefe0e249b8b1fcde21b3eaa6c7af5737fbf92003854376be9541a97257a2
```

Wheels:

Use thew more detailed explaination https://docs.duckietown.org/daffy/opmanu-al_duckiebot/out/wheel_calibration.html

Use the `gui_tools` to connect to the ROS:

```
$ dts start_gui_tools HOST_NAME
```

the use (in a different terminal) the description from the verification paragraph in order to start the keyboard control.

Then use this command in the `gui_tools` in order to calibrate the bot. Adjust the `TRIM_VALUE` in order to do so. Make sure the *weels can run freely* and that the bot drives straight within 10 cm on 2m.

daffy

```
$ rosparam set /HOST_NAME/kinematics_node/trim TRIM_VALUE
```

master19

```
$ rosservice call /HOST_NAME/inverse_kinematics_node/set_trim --
TRIM_VALUE
```

Unit G-3 Demo

This section presents a demo of the hardware benchmark where all is run on your own machine.

Contents

3.1. Start the API

The *DT_APP_SECRET* and *APP_ID* can be retrieved by either asking via the Slack or by using the Web-Debugger in the Duckietown Diagnostics within a (GET-)request header.

1) Saving data locally

In order to start the API which saves data on your local machine use the following command:

```
$ docker run -v /PATH/TO/DATA/DIR/:/data/ -it -e LOCAL=True -p
5000:5000 -e APP_SECRET=DT_APP_SECRET -e APP_ID=APP_ID --rm duck-
ietown/dt-hardware-benchmark-backend:daffy
```

2) Saving data online

Careful the API address in the frontend is hardcoded and thus needs to be adjusted

In order to start the API which saves data on S3 and a MySQL database use the following command:

```
$ docker run -dit -p 5000:5000 -e MYSQL_USER=DB_USER -e
MYSQL_PW=DB_PW -e MYSQL_URL=DB_URL -e MYSQL_DB=DB_NAME -e
AWS_SECRET_ACCESS_KEY=AWS_SECRET_ACCESS_KEY -e AWS_AC-
CESS_KEY_ID=AWS_ACCESS_KEY_ID -e APP_SECRET=DT_APP_SECRET -e
APP_ID=APP_ID--rm duckietown/dt-hardware-benchmark-backend:daffy
```

3.2. Start the Frontend

Adjust the API address if wanted to run online

```
$ docker run -it -p 3000:80 --rm duckietown/dt-hardware-benchmark-
frontend:daffy
```

It then is reachable under `localhost:3000`

3.3. Run the benchmark

Ensure that Lane following runs on your duckiebot, if necessary consult the resp. documentation.

You need two terminals, another one to prepare the start of lane following as described in the output of the CLI. Plug in a USB Stick int the **top left** port of the Duckiebot.

Now it is time to run the benchmark you can use the `dts`:

```
$ dts benchmark BOTNAME -a YOUR_LOCAL_IP_ADDRESS:5000
```

If your API is running online, enter the `API_URL` including the protocol (i.e. `https://`).

UNIT G-4
Software Architecture

This section introduces the hardware benchmark software architecture. *It is not required to read in order to conduct the benchmark.* It generally consists of three small programs communicating with each other:

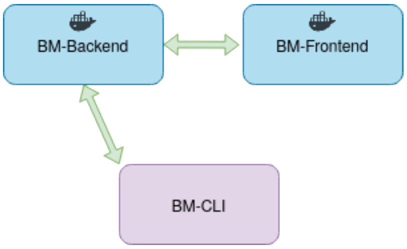


Figure 4.1. Hardware benchmark tools

Contents

In order to **facilitate and unify** the benchmark procedure the above mentioned tools have been created. The optimal setup would be similar to the dts diagnostic tool. One command that starts the benchmark and then uploads all files and displays the data online. Once the API and frontend are online resp. incorporated in the dashboard, this will almost be possible. Only the lane following has to be started by hand.

4.1. Communication

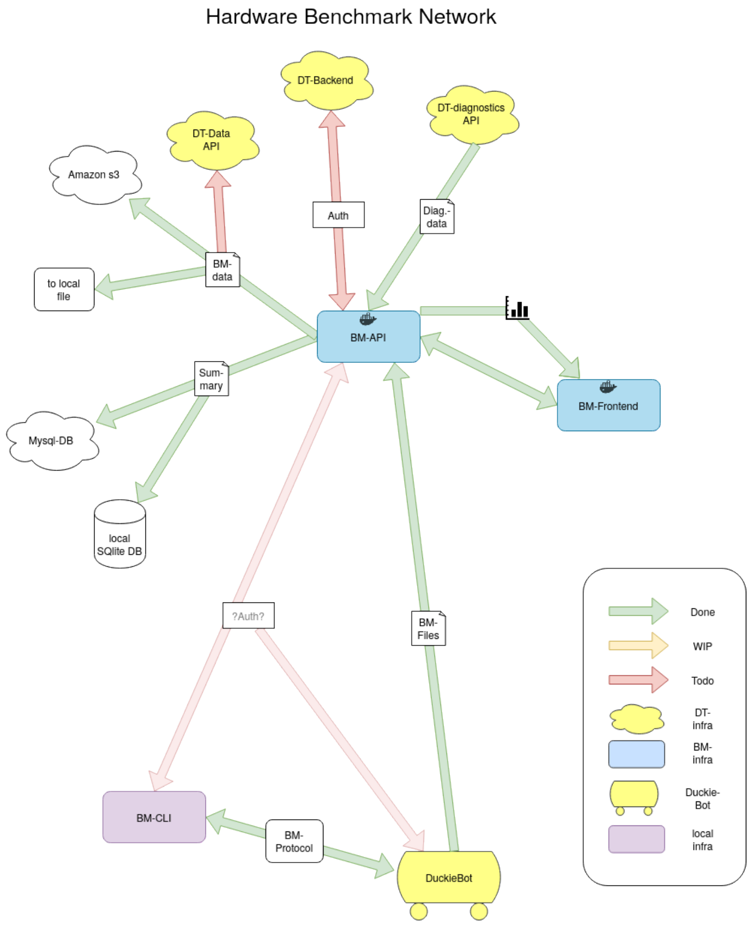


Figure 4.2. Hardware benchmark software communication diagram

The above diagram summarizes the communication between different services. It is obvious that the API is central for the benchmarking procedure.

4.2. API / Backend

1) Endpoints

Once the API is running it offers a Swagger UI which describes all endpoints. It can be accessed at the address of the API (running locally, normally at (localhost:5000)[`localhost:5000`])

2) Data storage

The API provides two different ways to store data:

- Locally on the machine using SQLite and writing files in the specified directory
- Using S3 and a mySQL Database hosted separately

3) Functionality

The API offers two endpoints where the benchmark files can be uploaded via POST request. Once the files, consisting of:

- `meta`, JSON string containing manually written metadata
- `meta_json`, JSON file containing Metadata
- `sd_card_json`, JSON file containing SD Speeds
- `latencies_bag`, Bag containing Latencies, recorded on the bot
- evt. `diagnostics_json`, JSON containing the diagnostic data
- evt. `localization_bag`, Localization bag

are uploaded. The API retrieves all benchmarking measurements from those files, saves them to a file as well as a summary to the Database. If no `diagnostics_json` but the `diagnostics_id` is provided in the request url, the API retrieves the diagnostic data from the dt-diagnostics tool.

4.3. Frontend

The frontend is used to display the benchmark data. As well as a synthetically calculated score.

1) File Upload

The frontend provides an interface to upload data manually, as one may make the benchmark as well "by hand".

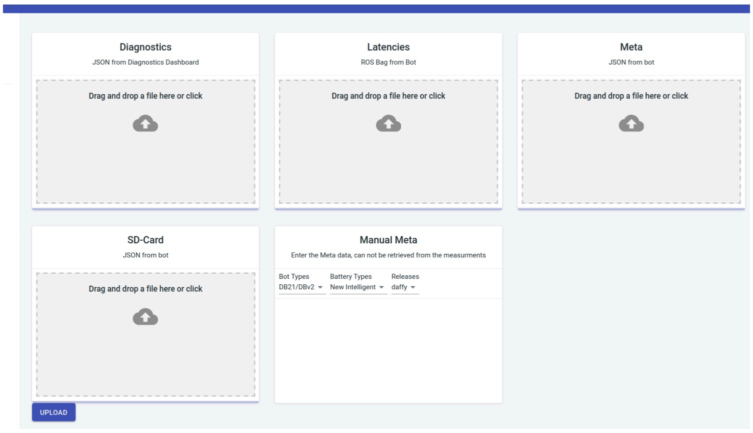

Diagnostics
JSON from Diagnostics Dashboard
Drag and drop a file here or click
Latencies
ROS Bag from Bot
Drag and drop a file here or click
Meta
JSON from bot
Drag and drop a file here or click
SD-Card
JSON from bot
Drag and drop a file here or click
Manual Meta
Enter the Meta data, can not be retrieved from the measurments
Bot Types
DB21/DBv2
Battery Types
New Intelligent
Releases
daffy
UPLOAD

Figure 4.3. Benchmark Upload

2) Score Display

All scores are displayed on the web interface including the possibility to download the saved data and the

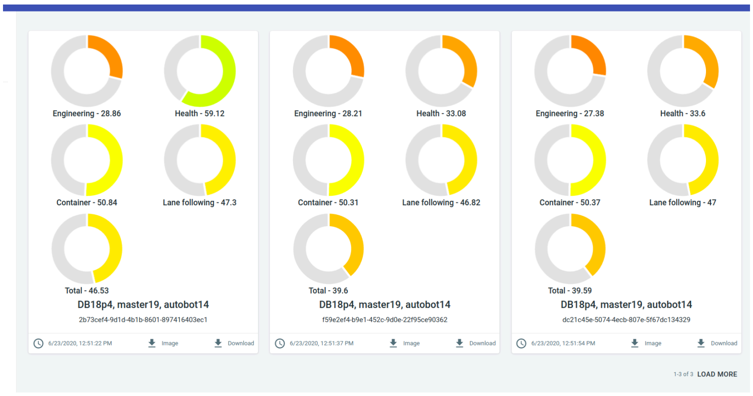


Figure 4.4. Score Display

4.4. CLI

The CLI is part of the Duckietown shell `dts` benchmark command. More explanation can be found in the section Demo.

Unit G-5
Settings

Possible Settings for calculating the BM score

Contents

> **Remark:** In order to edit any function one needs the API in the developer mode, e.g. clone git repo and run from there. Instructions can be found in the repo.

5.1. Weighting Function

In order to adjust the weight function, edit the function `weight_function` in the file `logic/utils/data_collection.py`.

5.2. Weights of single element in benchmark score

Adjustments of weights are made in the dict `averages` in the file `logic/config/config.py`

5.3. New Score Categories

Add another entry to the dict `averages` in the file `logic/config/config.py`

5.4. New measurements

In order to add another measurement which is calculated during the benchmark file analysis edit the function `measurements` in the file `logic/config/config.py`. If more external data is needed to display, the functions calling measurements need to be edited. For further information consult the `README.md` in the repo.

UNIT G-6
Troubleshooting

Troubleshooting section

Contents

6.1. Init_sd_card

Should an error like the one below appear:

```
subprocess.CalledProcessError: Command '['sudo', 'e2fsck', '-f', '/dev/
sdb2']' returned non-zero exit status 8.
```

use

```
$ sudo fdisk /dev/device
```

to delete all partitions on the device. (`d` to delete, `w` to write) **Make sure to edit the correct device, otherwise data will be lost!**

6.2. Render Apriltags

If your map is rendered really tiny. You probably entered the measurements of the april tags in another unit than meters.

6.3. Run Lane following

If the lane following can't be run after the Pre benchmark. Restart the Lane following Container.

Unit G-7
Future improvements

The chapter of benchmarking performance of the Duckiebots is by far not over. This thesis sets a baseline for benchmarking the hardware behaviour. Still all used software plus the applied measurements have room for improvement. Some ideas are presented in this chapter.

Contents

7.1. Measurements:

- Collect more data
- Include localization system data
- Incorporate Battery measurements
- Run benchmark using \verb+daffy+ as software release

7.2. API:

- Incorporate other benchmark procedures e.g. Software Benchmark
- Add authentication via the Duckietown token
- Upload the data to an external host data storage
- Cache overall score in order to reduce response time for resp. endpoint
- Incorporate localization info
- Use \verb+bjoern+ or similar as production \verb+WSGI+ (Web Server Gateway Interface) server, as the Flask development server is technically not save to run in a production environment.
- Save graphs, further reducing the response time

7.3. CLI

- Adjust docker digests for \verb+daffy+, resp. \verb+daffy_new_deal+ container once it is released
- Automated hardware compliance check
- Invoke callbacks in separate threads
- Display link to the new diagnostic result

7.4. Frontend

- Improve frontend, enable direct comparison between average and newly supplied benchmark
- Include frontend into the dashboard
- Frontend make responsive

Part H
Continuous Integration

Author: Andrea F. Daniele

Maintainer: Andrea F. Daniele

This section of the book focuses on the Duckietown CI Infrastucture. This is clearly not something that every Duckietown developer has to be knowledge about but definitely something they should be aware of.

Contents

UNIT H-1
CI Infrastructure

Contents

1.1. CI Infrastructure

Our CI Infrastructure is comprised of two types of nodes, i.e., **master** and **builder** nodes.

We currently have three nodes, one **master** and two **builder** nodes. The `amd64` builder node is responsible for building artifacts natively on `amd64` architecture. The `arm32/64` builder node builds artifacts for the architectures `arm32v7` and `arm64v8` instead.

The following picture depicts how these nodes are connected to each other.

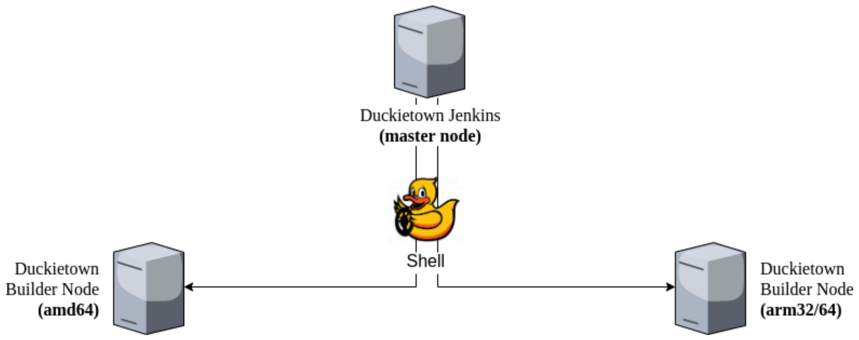


Figure 1.1. CI Infrastructure

1) Master node

The Master node receives notifications from our Git host (github.com) about push events against monitored repositories. Push events are tuples of shape `(repository, branch, commit, author)`. When a push event occurs, the CI master node kicks in and spawns a new **build job**. A **build job** is performed on a **builder** node and has the objective of building a collection of artifacts from the corresponding source code. The event's `repository`, `branch` and `commit` identify the source code version to use.

> Note: Even though `repository` and `commit` are in theory enough to identify the version of the source code, we need the `branch` as well as the artifacts will take its name.

2) Builder node

A builder node is simply a Docker endpoint accessible through the TCP port `2375`. Build jobs are always run inside a Docker container. Some build jobs consists of Docker image builds (and subsequent push to DockerHub).

Unit H-2 CI Builder Nodes

A builder node is simply a Docker endpoint accessible through the TCP port `2375`.

Contents

2.1. Setup a new Ci Builder Node

CI Builder nodes are implemented on AWS EC2 instances. To setup a new builder node, you need to:

1) Create a new EC2 instance on AWS

You can create a new instance by visiting the AWS EC2 Dashboard.

We suggest the following AMIs:

- **arm32/64**: `ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-arm64-server-20181120 (ami-01ac7d9c1179d7b74)`
- **amd64**: `ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64-server-20191113 (ami-00a208c7cdba991ea)`

2) Setup passwordless SSH using the `ci-nodes` RSA keypair

On the Duckietown AWS account, a key-pair called `ci-nodes` holds the public key used to login into any CI-related node.

The private key is neither shared here nor stored on AWS for obvious reasons. If you need it, ask Duckietown administrators.

3) Assign an Elastic IP to your instance.

By assigning an AWS Elastic IP to your instance, you make sure that the IP never changes, so that we can configure other tools with static IPs pointing at each builder node.

4) SSH in and install Docker

SSH into your newly created node and install Docker.

```
sudo apt install docker.io
```

5) Setup crontab

Builder nodes will build many Docker images over the course of a few days. If we don't have a way to keep them clean, they will run out of space in less than a week.

For this reason, we setup two cron jobs. The first one runs every day at 2AM and removes all stopped containers and frees any unused resources (e.g. volumes). The second kicks in 1 hour later, at 3AM and removes all unused images. In both jobs, only resources that are not used for more than 24 hours are freed.

You can setup the two cronjobs by running `crontab -e` and pasting the following lines at the end of the file.

```
00 02 * * *     docker system prune --filter until=24h --force
00 03 * * *     docker image prune --filter dangling=true --filter un-
til=24h --force
```

6) Add the user **`ubuntu`** to the group **`docker`**

Use the following command to add the user `ubuntu` to the group `docker`. This will give your user access to the local Docker engine.

```
sudo usermod -aG docker ubuntu
```

Note: You need to log out and back in for the changes to have an effect.

7) Setup Docker TCP socket

The Master node is the one triggering builds on builder nodes. For this to happen, each builder node has to make the Docker engine available on a TCP port.

In order to enable the Docker TCP socket, open the Docker service configuration file.

```
sudo nano /lib/systemd/system/docker.service
```

Navigate to the `[Service]` section of the file and find the line

```
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/con-
tainerd.sock
```

and replace it with the following

```
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2375 --contain-
erd=/run/containerd/containerd.sock
```

This will give anybody access to your Docker engine over the internet. Do not worry about possible intruders, we will configure the EC2 instance to only accept connections on the port `2375` from the master node.

Save the file and reload (then restart) the Docker service with the following commands:

```
sudo systemctl daemon-reload
sudo service docker restart
```

Test that your change had an effect by executing the command

```
curl http://localhost:2375/version
```

You should get a JSON string with info about the Docker engine. If you get an error, redo this step.

8) Configure EC2 instance to accept connections from the Master node only

Navigate to the AWS EC2 Dashboard page. Then click on `Instances` to the left. Select the newly created builder node and select `Networking -> Change Security Groups` from the `Actions` menu at the top of the list. Add the group `[in]-Docker-API` to the security groups of your instance. This allows your instance to accept connections from the Master node.