ThunderVolt Description Paper 2021

Felipe Gomes de Melo D'Elia[1], Gabriel Cosme Barbosa[1], Leonardo Isao Komura[1],
Letícia Miyuki Kimoto[1], Lucas Haug[1], Lucas Tonini Rosenberg Schneider[1],
Maria Eduarda Dall'Orto de Araujo[1], Ricardo Tamay Honda[1] and Thalles Carneiro da Silva[1]

***Abstract*— ThunderVolt is the Very Small Size Soccer team of ThundeRatz, the robotics group from Polytechnic School of the University of São Paulo. The team was first present in the Iron Cup 2020, and since then, the code was restructured for the Iron Cup 2021, and now again for the LARC 2021. In this paper, we present the work that have been develop so far in different areas, such as mechanics, embedded devices, game strategy and simulation.**

I. INTRODUCTION

The *IEEE Very Small Size Soccer (VSSS)* competitions have grown a lot in the national robotics scenario, mainly because robot soccer is a challenging and stimulating problem that requires a complete engineering solution. That is why some members of ThundeRatz, the robotics team from the Polytechnic School of the University of São Paulo, started developing the ThunderVolt project in 2018.

The solution designed for ThunderVolt can be divided into four different modules: the mechanics, responsible for the physical specification of the robot; the embedded devices, consisting of the hardware and firmware used on each of the team's robots and on the RF transmitter; the game strategy, where the decision-making and the navigation systems are implemented; and the simulation, needed to develop the game strategy without relying on a physical robot. This paper describes in more details each of theses modules and how they were developed.

II. MECHANICS

Our mechanical project can be divided into three main aspects, the drive system, the internal frame and the uniform. The rendered CAD assembly is shown as the Figure 1.

The drive system was made using Pololu's Micro Metal Motors, and in order to optimize space within the internal frame of the robot, the motors were not aligned and a pair of gears were used to transmit the power from the motor output to the wheels of the robot.

With the intent to determine the ideal gear ratio of the motors, a mathematical model to determine what was the best gear ratio to a specific wheel was made. There are two cases in our model, when there is no sliding (Figure 2) and when there is sliding (Figure 3).

Where:

- m the robot's mass
- R the robot's wheel radius

[1]All authors are university graduate students at the Polytechnic School of the University of São Paulo and members of the ThundeRatz Robotics Team. contato@thunderatz.org

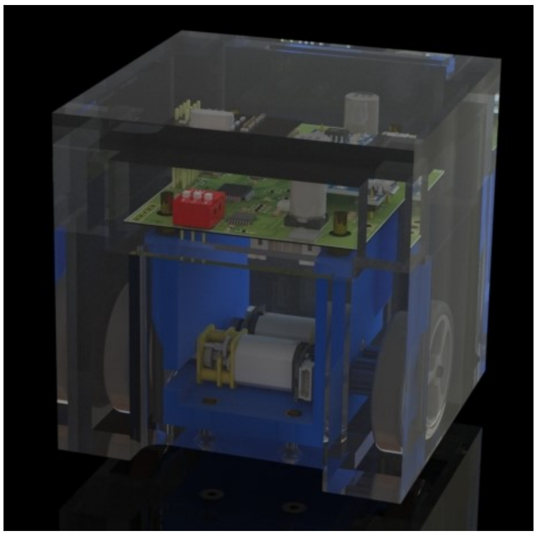
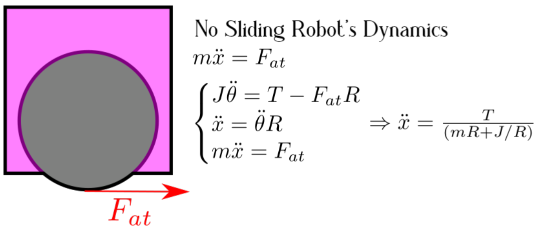


Fig. 1. Robot's Assembly


No Sliding Robot's Dynamics
$m\ddot{x} = F_{at}$
$\begin{cases} J\ddot{\theta} = T - F_{at}R \\ \ddot{x} = \ddot{\theta}R \\ m\ddot{x} = F_{at} \end{cases} \Rightarrow \ddot{x} = \frac{T}{(mR+J/R)}$
F_{at}
Fig. 2. Dynamic Robot Model No Sliding Case

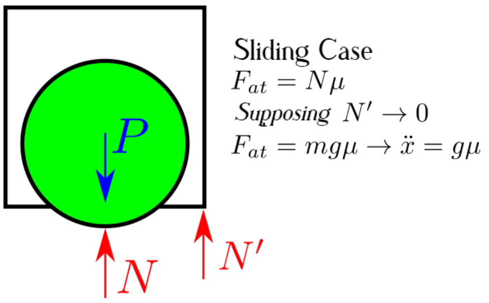


Fig. 3. Dynamic Robot Model Sliding Case

- F_{at} frictional force on the wheel
- μ friction coefficient of the wheel
- J moment of inertia of the wheel
- T motor output torque
- $\ddot{x}$ the robot's acceleration
- $\ddot{\theta}$ motor angular acceleration
- N normal force on the wheel

So, when plotting our $\ddot{x}$ in function of the mass, we have:

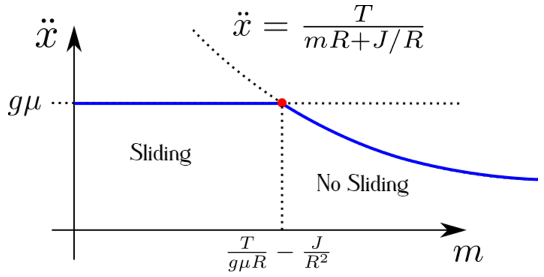


Fig. 4. Acceleration in function of Robot Mass

From the inflection point of the graphic, we can see that in order to have no sliding, we have to attend to this relation:

$$m \geq \frac{T}{g\mu R} - \frac{J}{R^2}$$

We can simplify our equation by not considering the inertia of the wheel, solving for the Torque we have:

$$T \leq mg\mu R$$

The output torque can be written as $T = GearRatio * T_0$, where T_0 is the torque of the motor before the gearbox transmission. So we have:

$$GearRatio * T_0 \leq mg\mu R$$

The assembly mass shown in SolidWorks is 125 g, g is 9.81 m/s², the friction coefficient μ is 2.2 and T_0 is 0.0014388 Nm [20]. For a wheel diameter of 32 mm we have that our gear ratio is in between 15.0 +- 3.6. In order to be well comfortable in the no-sliding area of the graphic, a **10:1 gear ratio gearbox** was selected for the drive system.

The robot uniform has the function to cover the internal frame and protect all it's internal components. Three types of uniforms were developed, each one with its own geometry to drive the ball. Two uniforms have a C type geometry and one has an ellipse type driving ball geometry.

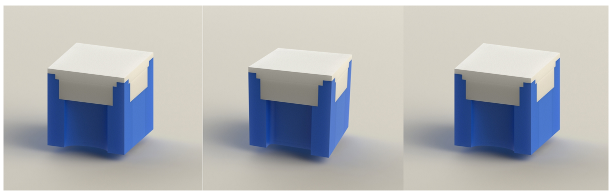


Fig. 5. The three robot's uniforms

The parts of the uniform will be 3D printed and to determine the most appropriate filament to print the parts, a research was made, analyzing all kinds of filaments that are widely available. It was concluded that ABS filament was the most appropriate due to its mechanical properties and also its ease of printing.

The main part of the robot's frame is also ABS 3D printed, due to its complex geometry. The electronics are mounted in the robot using some brass pins that are inserted in the main 3D printed frame. The uniform is mounted into the main frame by some brass screw threads.

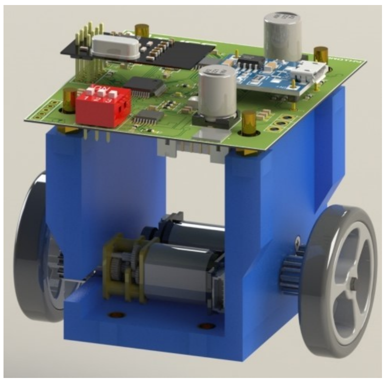


Fig. 6. Robot's Internal Frame

III. EMBEDDED DEVICES

For its complete operation, the project uses two types of embedded devices: the robots and a transmitter dongle. Where the transmitter is connected to a USB port of the computer running the game strategy and is responsible for receiving the speed commands for each robot from the connected computer and transmitting them to the robots using the Nordic Semiconductor's radio frequency module nRF24L01. The robots then receive the speed commands using the same type of radio module and control their motors accordingly. The specification of the electronics and the firmware of both devices will be described in the following subsections.

A. Electronics

Although the operation of the robot and the transmitter dongle could be described as a complex behavioral, because of its operating logic, the electrical system that powers and controls the basic functions of the robot and the dongle are rather simple. The transmitter dongle being currently utilized is very straightforward, is just a NUCLEO-F303RE board, integrated with a nRF module, however an original PCB is being designed by the team at this moment. With regards to the robots, ThunderVolt's board was fully designed by the members of ThundeRatz, via the circuit board designer program Altium Designer, and its main points will be highlighted below.

1) Microcontroller: The microcontroller utilized in the ThunderVolt project is the STM32G474CBT3, by STMicroeletronics. It is responsible to do the main control of the robot, therefore, it has influence in almost every electronic

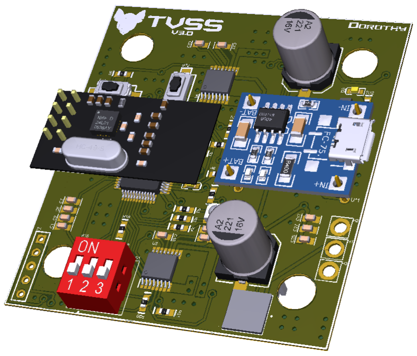


Fig. 7. 3D model of the PCB (”VSS”) from Thundervolt

system in the machine. Furthermore, it has an important relationship with the transceiver, who forwards the commands which it converts into signals that manage other components, controlling the movements of the robot during a match.

2) Transceiver: Responsible for receiving the speed commands sent by the computer, and repassing it to the microcontroller, the nRF24L01 transceiver, by Nordic Semiconductors, is a crucial component of the project. Its behavioral is described in the firmware section of this paper (III-B).

3) Boost voltage regulator: In the ThunderVolt robots, a single Li-Ion battery powers all the electric parts of the robot. But, for the motors' operation a 9V voltage is required. With that in mind, it was chosen for this project a tension boost converter, the TPS61089RNRR, by Texas Instrument. This component receives the 4.2V from the battery and converts it in 9V, which powers the motor driver.

4) Charging System: One remarkable feature in this robot is its charging system. To facilitate the handling, it was designed an electronic system that allows the user to charge its battery without taking it out. It is noteworthy that, while charging, the board is powered by the same USB that provides energy for the battery. In addition, there is a circuit, using an operational amplifier in comparative mode, that turns on a led notifying when the charging is complete.

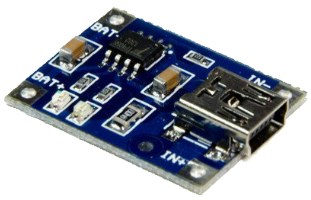


Fig. 8. Battery charger TP4056, from NanJing, that assists on the charging

5) H-Bridge Motor Driver: The hardware in charge of controlling the DC motors of the robot is the H-Bridge Motor Driver DRV8874, by Texas Instrument. Its operation is based in a PWM signal controlled by the microcontroller, which varies based on what the situation demands, regulating the motor current accordingly.

B. Firmware

The firmware of both embedded devices was initially based on the STM32ProjectTemplate [9], an open source template created by ThundeRatz to simplify the firmware development for the STM32 family of microcontrollers [22] using the STM32CubeMX software [14]. Besides that, ThundeRatz wrote the STM32Guide [24], a public guide to help programming the STMicroelectronics' microcontrollers.

The transmitter firmware specifically then was developed to receive a packet from the computer running the game strategy via serial communication, using the microcontroller's UART. The received packet was structured to contain a start byte, then six bytes of speed data, two for each robot, then a Cyclic Redundancy Check (CRC) byte for checking errors and finally a stop byte. After the the 6 bytes of speed data are received, they are sent, sorted according to the robot numbers, through the Serial Peripheral Interface (SPI) to the nRF24L01 module, which sends the data to all robots. In order for the microcontroller to communicate with the RF module, the team developed a library to handle this communication, the STM32RF24 [10].

Lastly, the robot's firmware is responsible for receiving the transmitter packet using SPI and the nRF24L01, then parsing it to get its data depending on the robot's number and then controlling its motors with Pulse Width Modulation (PWM) according to the received data.

IV. GAME STRATEGY

The game strategy was implemented using the *Robotic Operating System (ROS)* [8] framework, which provides many tools and libraries that are useful when developing robotic applications.

Also, ROS code is organized into packages, which enhances the modularity and organization of the project. Following these principles, the decision-making system was subdivided in a few tasks: the coach, responsible for the top-level decisions relative to the game state; the behaviors, which describes what the robot should do based on the role it was assigned by the coach; and the positioning, that decides where is the best place for the robot to be given the game state.

Also, the game strategy must provide a way for the robots to follow the decisions made, so a navigation package was also implemented. Inspired by the ROS Navigation Stack [12], the ThunderVolt Navigation provides a global planner, which generates a complete trajectory considering the positions of all the entities on the field; and also a local planner that generates the velocity command for a robot based on the global plan or other input variables. The next sections will discuss each of these parts in more details.

A. Coach

In order to choose which robot is going to have each behavior (eg. goalkeeper, defender and striker), there is a coach that defines this according to the positions of the robots and the ball. The coach is a program that is always analyzing the positions of the players to see if it is necessary to change the behaviors for a better performance.

In the program version that ThunderVolt was used in Iron Cup 2021 (February 2021), the coach was used in more static states, as to define the roles of the robots in a penalty, and a few dynamics states. For the version in LARC 2021, the team wants to apply the coach in more dynamics states, then the changes of the roles could be more dynamic. To do so, there will be used a behavior tree or a state machine.

To understand all the field's states, first, it's possible to divide in 2 moments: before the match starts and after. In the state at the moment the match starts (when the 3 robots are aligned), the behaviors are defined as what is the most common at the games: the striker at the front, the defender in the middle and the goalkeeper close to the goal. After, as the game starts, there are 2 possibilities: our team gets the ball and advances for the foe's side or the other team gets the ball and advances for our side.

If we go for the foe's side, the expected is the striker further ahead with the ball and the defender in the middle of the field. So, if the striker, in any moment, is ahead of the ball, it's necessary to change the roles, so the robot behind the ball could attack and the one ahead of the ball can come back to the middle of the field. This change repeats until we score a goal. In the other situation, when the foe's advances for our side, there will be a limit that, if the ball passes, getting closer to our goal, the robot in front of the ball needs to be more offensive, get the ball and advance to the foe's area, so it turns into the striker, and the other becomes the defender.

B. Behaviors

The team's behavior is determined through a set of behavior trees. A behavior tree is an arrangement of nodes (internal nodes) and leafs (execution nodes). The former controls tree's flow by making use of different node types (e.g.: selector, sequence, parallel or decorator), while the latter describes an action or a condition, and returns a feedback (success, failure or running) to guide internal nodes. This configures a tree that specifies the player's action chain.

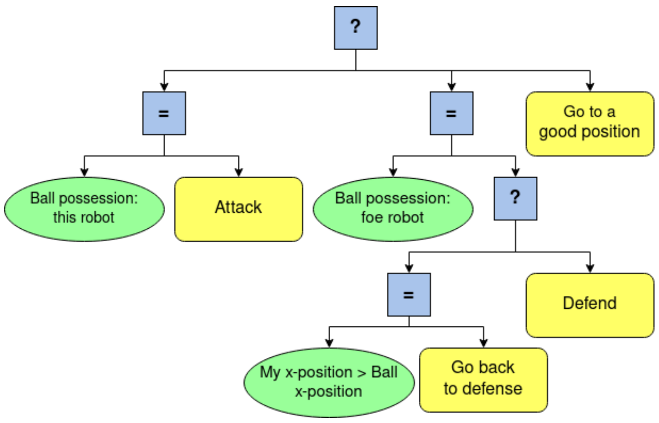

?
=
=
Go to a good position
Ball possession: this robot
Attack
Ball possession: foe robot
?
=
Defend
My x-position > Ball x-position
Go back to defense

Fig. 9. Basic behavior tree implementation

An example of its structure is shown in Fig. 9. This example is a simple implementation of a behavior tree: it flows from left to right, guided by internal nodes (blue box). The selector node "?" tries to run each of its branches one at a time, going to the next branch when receiving a failure feedback, until it receives a success from a leaf or it has no more branches to run. The parallel node "=" executes all of its sons nodes simultaneously and returns failure if at least one of them returns failure. This and the condition nodes (green box) controls which action (yellow box) will be executed.

The benefits of this approach includes its reactivity (due to execution node's feedback) and modularity, since each execution node can be implemented separately and can be used one or multiple times in the tree, resulting in a practical, flexible and efficient tool.

Our team has created 3 types of behavior trees (defender, striker, and goalkeeper) using py_trees [13], a library which helps creating them and allows information exchange between nodes using a blackboard. We are currently adapting that structure using BehaviorTrees.CPP [5], which stores the trees in a .xml file that can be updated at run-time. Above that exists a coach that dynamically allocates the available trees among the players.

C. Positioning

In certain situations, it is necessary for one of the robots to assume a specific position on the field, so that, for example, the chances of catching rebounds increases. With this in mind, an algorithm based on the HELIOS team for 2D Soccer Simulation was implemented [2]. In which favorable positions are defined for designated points on the field, and through a triangulation of the plane, in this case the Delaunay Triangulation, and a linear interpolation algorithm, the player's positions are acquired for the other points where the ball can be.

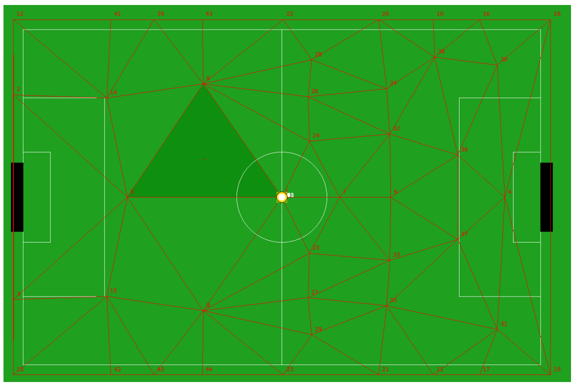


Fig. 10. Delaunay Triangulation on 2D Soccer Simulation field using fedit2

Thus, we have an locally adjustable algorithm that uniformly defines a position for all points in the field with satisfactory accuracy and that doesn't require a lot of computing power.

D. Global Planning

To make the robot go to the desired position, a path is made using vector fields. This simple yet efficient method consists of a function that associates a vector to each point of the field, representing an attraction or a repulsion force.

These set of vectors guides the robot, and can be combined with different fields to create specific paths.

The initial implementation was inspired on NeonFC [6], and makes use of three basic fields: radial, line and tangent. From these, other fields were constructed for many purposes, mainly to avoid obstacles (wall, defense area, other robots) or go to specific points (goal, ball, strategic positions).

We are also analyzing other approaches, such as Visibility Graphs. This type of planning tries to minimize the distance to travel to a goal point by creating routes that passes near to the obstacles, which are represented as polygons (e.g.: circles, triangles, etc). A path is composed of edges that connects two visible vertexes or tangent points of these obstacles, until the goal is seen. The optimized route is created using search algorithms like A* and modelling it through a graph, while using some heuristic methods to make it smooth. This method avoids a big problem that happens in vector field approach: local minimum points, which leads to a dead end path.

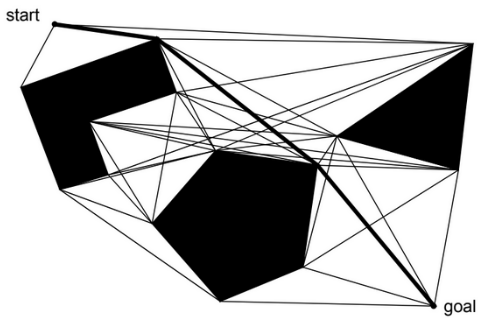


Fig. 11. Visibility graph example [3]

E. Local Planning

The local planning task must be flexible enough to provide a few different ways to control a robot. For the most basic ones, like looking at some direction or spinning, a PID controller was implemented to control the angular position of the robot.

For tasks like following a defined path provided by the global planner a Dynamic Window Approach Controller (DWA) is being implemented. This method consists in sampling all possibilities from the control space of the robot (linear and angular speeds) and simulating the robot's trajectory for each one, as shown on the figure 12.

After generating all trajectories, we must evaluate then based on a few different metrics, like proximity and alignment to the target and proximity to the global path. Those metrics are weighted to compose an score function, this way, the linear and angular speeds associated with the highest-scoring trajectory are sent to the robot.

V. SIMULATION

When developing a robotics application, one of the main obstacles is relying on the physical robots to validate the developed software. So in order to reduce this dependency

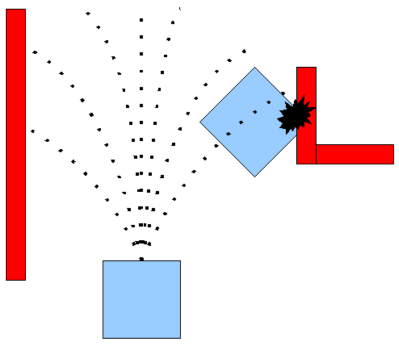


Fig. 12. Marder-Eppstein, E. *DWA Local Planner*. Retrieved from `http://wiki.ros.org/dwa_local_planner`

and facilitate software development, it is necessary to use simulation tools.

Thus, along with ThunderVolt, the team develop an open source IEEE Very Small Size Soccer simulation project entitled TraveSim [15], which supports simulating a single robot, a team and a match, in a 3x3 VSSS field or a 5x5 VSSS field. Similar to ThunderVolt, TraveSim was develop using ROS [8], therefore it was structured based on Gazebo [7], the ROS default simulator. In addition to the ROS integration, Gazebo offers an accurate and effective robot simulator, possible thanks to robust physics engines, like ODE [21], realistic rendering using OGRE [23] and a huge open source community.

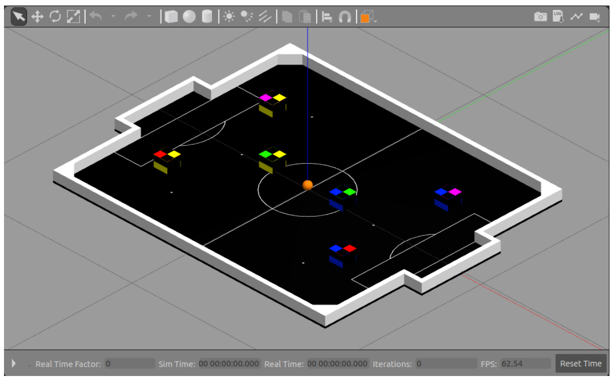


Fig. 13. 3x3 match simulation in TraveSim

As a ROS based project, the communication with TraveSim is achieved using ROS Topics and ROS Services, whose names are described in the TraveSim documentation and in Gazebo's ROS tutorials [19].

TraveSim supports two commands interfaces for controlling the robots:

- Differential drive control: the robot's velocity is represented in two components, a linear component in m/s and an angular component in rad/s.
- Direct motor control: each motor is independently controlled using angular velocity commands in rad/s.

Moreover, Gazebo itself provides an interface to set and obtain all models states, like the positions and velocities of the robots. These states are all relative to the world frame of reference and have their units of measure all in units of the International System of Units.

Lastly, as the communication with TraveSim depends on ROS, with the aim to allow non-ROS projects to use the simulator, another project was developed to provide a layer of adapters, the TraveSim Adapters [16]. These adapters then allow communicating with the simulator through the Google Protobuf library, creating UDP sockets to receive and transmit data. The messages used then to communicate with the simulator remained the same as the VSSSLeague's FIRASim [17], which can be seen in [18], simplifying the use of the simulator.

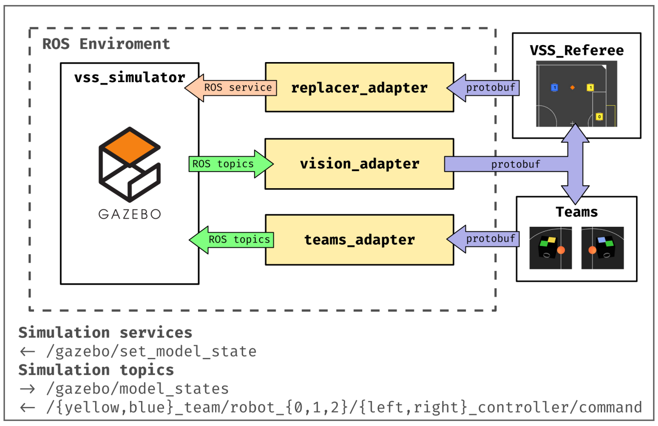


Fig. 14. TraveSim Adapters Architecture

ACKNOWLEDGMENT

The ThunderVolt team would like to thank the ThundeRatz robotics team, Amigos da Poli Patrimonial Fund and Polytechnic School of the University of São Paulo for all their support. In addition to giving recognition to our sponsors STMicroelectronics, Altium, SolidWorks and Circuibras.

REFERENCES

[1] Jong-Hwan Kim et al." *Soccer Robotics*. Vol. 11. Springer Tracts in Advanced Robotics. Springer, 2004. ISBN: 3-540-21859-9.

[2] Hidehisa Akiyama and Itsuki Noda. "Multi-agent Positioning Mechanism in the Dynamic Environment". In: *RoboCup 2007: Robot Soccer World Cup XI*. Ed. by Ubbo Visser et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 377–384. ISBN: 978-3-540-68847-1.

[3] Roland Siegwart et al." *Introduction to Autonomous Mobile Robots*. Vol. 2. MIT Press, 2011. ISBN: 978-0-262-01535-6.

[4] Michele Colledanchise. "Behavior Trees in Robotics". MA thesis. SE-100 44 Stockholm, Sweden: School of Computer Science and Communication - KTH, 2017. ISBN: 978-91-7729-283-8.

[5] Davide Faconti. *BehaviourTree.CPP*. Mar. 22, 2020. URL: https://github.com/BehaviorTree/BehaviorTree.CPP/.

[6] NeonFC. *NeonFC potential fields implementation*. Oct. 31, 2020. URL: https://github.com/project-neon/NeonFC/wiki/Potential-Field.

[7] Open Robotics. *Gazebo*. Version 11.0. Jan. 30, 2020. URL: http://gazebosim.org.

[8] Open Robotics. *Robotic Operating System*. Version ROS Noetic Ninjemys. May 23, 2020. URL: https://www.ros.org.

[9] ThundeRatz. *STM32ProjectTemplate*. Oct. 9, 2020. URL: https://github.com/ThundeRatz/STM32ProjectTemplate.

[10] ThundeRatz. *STM32RF24*. Version v1.0. Nov. 26, 2020. URL: https://github.com/ThundeRatz/STM32RF24.

[11] Eitan Marder-Eppstein. *DWA Local Planner*. Version noetic-devel. Aug. 9, 2021. URL: http://wiki.ros.org/dwa_local_planner.

[12] Eitan Marder-Eppstein. *ROS Navigation Stack*. Version noetic-devel. July 20, 2021. URL: https://github.com/ros-planning/navigation.

[13] Splintered-reality. *py_trees*. May 31, 2021. URL: https://github.com/splintered-reality/py_trees.

[14] STMicroelectronics. *STM32CubeMX*. Version 6.3.0. July 8, 2021. URL: https://www.st.com/en/development-tools/stm32cubemx.html.

[15] ThundeRatz. *TraveSim*. Version 21.04.1. Apr. 7, 2021. URL: https://github.com/ThundeRatz/travesim.

[16] ThundeRatz. *TraveSim Adapters*. Version 1.0. July 5, 2021. URL: https://github.com/ThundeRatz/travesim_adapters.

[17] VSSSLeague. *FIRASim*. June 28, 2021. URL: https://github.com/VSSSLeague/FIRASim.

[18] VSSSLeague. *VSSS Simulation Protobuf Messages*. Apr. 10, 2021. URL: https://github.com/VSSSLeague/vsss_sim_pb_msgs.

[19] Open Robotics. *Tutorial: ROS Communication*. Accessed: 2021-07-20. URL: http://gazebosim.org/tutorials?tut=ros_comm&cat=connect_ros.

[20] Pololu Robotics and Electronics. *Micro Metal Gearmotors*. URL: https://www.pololu.com/category/60/micro-metal-gearmotors.

[21] Russ Smith. *Open Dynamics Engine*. URL: https://www.ode.org.

[22] STMicroelectronics. *STM32 32-bit Arm Cortex MCUs*. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.

[23] OGRE Team. *OGRE*. URL: https://www.ogre3d.org.

[24] ThundeRatz. *STM32Guide*. URL: https://github.com/ThundeRatz/STM32Guide.