Eindhoven University of Technology

Eindhoven University of Technology

MASTER

Realizing Simulated Matches in the RoboCup MSL

Nijland, L.D.

Award date:
2021

Link to publication

Realizing Simulated Matches in the RoboCup MSL

Graduation Project

Student:
L.D. Nijland
Identity number:
0958546

Supervisor:
dr. ir. M.J.G. van de Molengraft

Eindhoven, September 14, 2021

Abstract. To stimulate faster progression in the RoboCup MSL, simulated matches are a valuable addition. This dissertation presents and verifies an architecture that achieves this goal. In this architecture, the simulation software of two teams connect to and communicate with the developed middleware. To facilitate this, software is produced that is easily integrated in the software of the teams. The middleware relies on two protocols. The first is a communication protocol that defines a flow of information and guarantees quality of the shared information. The second is a time synchronization protocol that ensures synchronization of the connected simulators with respect to simulation time. Additionally, a pauser application is developed that executes the time synchronization protocol upon command of the middleware. A containerization tool is proposed that allows the teams to easily share a binary release of their simulation software with others. By deploying the other applications to containers as well, a network of containers is established that together form the architecture. The architecture is tested by hosting a simulated match between two software copies of Tech United which shows that the solution is suited for matches between different teams as well.

1   Introduction

RoboCup is an international initiative that promotes collaborative research into intelligent mobile robotics, artificial intelligence and other related fields. Currently, RoboCup hosts five major competitions, each divided into several leagues among which the Middle Size League (MSL) [1]. In the MSL, two teams consisting of five autonomous robots each play a game of soccer. The teams are free to design their own hard- and software as long as their design complies with the limitations stated in the rules.

Testing on real robots is costly and difficult due to the highly dynamic and aggressive environment the robots operate in. Therefore, simulation is a necessity [2]. A simulation relies on two components: a simulator and a visualizer. The simulator computes the state of the robots and the ball in the simulated environment. The visualizer is used to display the match. Each robot in the simulator is controlled by a duplicate of the software that is originally developed to control a real robot. An important property of the simulator is the simulation time which is the notion of time within the simulator and is uncorrelated to real time.

Most teams, if not all, make use of simulations and their simulators are tailored to fit their specific needs. Some teams are interested in high-level behavior to test strategies, while others value low-level behavior too by simulating the robots very accurately. E.g., this allows to test hardware components such as a motion or vision system. Currently, teams use simulations to test their design during regular game play or in specific game situations such as free kicks, corners or penalties. However, this does not necessarily incorporate opponents. If opponents are added, then these will either be static or controlled by their own software as if they are facing themselves.

According to the teams, considerable progress is made during the annual tournaments. Here, they are confronted with their own weaknesses while learning from the strengths of others. In order to stimulate faster progression, it would be a valuable addition if the teams in the MSL could simulate matches against each other in between the tournaments.

This paper presents and verifies an architecture that realizes simulated matches by connecting the simulation software of two opposing teams. Section 2 presents related research into simulations in RoboCup. Moreover, previous attempts to realize simulated matches in the MSL are discussed. In Section 3, an initial architecture is presented that is based on requirements, preferences and a licensing item that were formulated after meeting with several teams. Section 4 discusses two protocols that cover what information needs to be shared between the teams, how quality of this information is ensured and how the simulators are synchronized with respect to simulation time. Next, Section 5 dives into the software specifics of the architecture. Here, not only a method is proposed that allows teams to share their simulation software with others, but also the final architecture is presented and the software structure is explained. The proposed architecture is put to the test in Section 6 where a simulated match between two copies of the software of Tech United is discussed. Additionally, time synchronization is verified as well. Finally, Section 7 concludes this paper with remarks on the presented work and recommendations for when this project is proceeded.

2 Related Work

The RoboCup Soccer competition hosts two simulation leagues: the 2D and 3D Simulation League. The 2D Simulation League relies on the Soccer Server for simulating matches. This is a tool provided by the organization and is divided in the RoboCup Soccer Simulator Server and the RoboCup Soccer Simulator Monitor [3]. The server provides the virtual match environment and simulates the interactions between the players and the ball. Meanwhile, the monitor shows the match. Each player is controlled by a client, which resembles the brain of a player. In order to facilitate the communication between a client and the server, the organization provides a sample client with a predefined list of commands that can be used to control the players. The 3D Simulation League uses a similar architecture [4]. Here, the players are controlled by agents and each agent represents one player. Communication between the agents and the server is managed by the so-called agentproxy. This component bundles all incoming messages of the agents and forwards them to the server. In return, the agentproxy receives new state information from the server and sends this to the agents. Similarly as in the 2D Simulation League, the incoming and outgoing communication is fixed. Moreover, the league provides the simulation software that is used during tournaments.

Apart from the MSL, there are three other non-simulation leagues in the RoboCup Soccer competition and in two of them, the Humanoid and the Small Size League, simulations are common. The Humanoid League has a simulation branch. Here, the teams are provided with league-standard simulation software. The teams are free to design their own robot model that is used within the simulation. However, the model has to comply with the rules and all communication is fixed [5]. Secondly, the Small Size League is familiar with simulated matches as well. As a matter of fact, due to the current restrictions on organizing large events induced by the COVID-19 pandemic, in 2021 a virtual version of the annual Small Size League tournament was organized. League-standard simulation software was provided and rules for communication with the simulator were defined [6].

The architectures used throughout the various leagues show much resemblance. That is, in all of the leagues the organization provides the teams with simulation software and all communication is fixed. These similarities extend to other competitions as well such as the simulation branch of the RoboCup Rescue competition [7].

Naturally, previous attempts to realize simulated matches in the MSL follow a similar approach. In [8], simulation software based on multi-level abstraction is presented that was meant to become the standard of the MSL. However, this goal was never reached and the project is out of maintenance ever since. A more promising example is Simatch. Since 2016, Simatch has been the official simulation software of the Chinese Simulation MSL and in 2017 it was introduced to the RoboCup MSL as well [9]. As documented on the GitHub page of the MSL [10], it was intended that by the end of April 2018 the simulation software would be operational and the first simulated matches could be played. However, despite the efforts made, this goal was not reached and in the following years no significant progress was made.

Meetings with four different MSL teams (Tech United, Robot Sports, Falcons and CAMBADA) were arranged. During these meetings, it became clear why integrating the Simatch project was unsuccessful. Each team has its own concept of what should be incorporated in a simulation. One clear difference is the level of detail that is used to represent the robots in the simulation. Where some teams believe that the robots can be modeled rather simplistically, others argue that a simulation must portray reality as closely as possible and thus that the robots should be very detailed. Disagreements on design choices like these explain why teams are reluctant to abandon their own simulation software in which they have put much effort and switch to a centralized alternative.

3   Preparation and Initial Architecture

The main goal of the meetings with the four aforementioned MSL teams was to learn the current stance of the teams with regard to simulated matches. Based on these meetings, requirements and preferences are formulated and a licensing item for fair usage of the solution is defined.

3.1   Requirements

From the outcome of the meetings, the following requirements are formulated:

I. Minimal effort is required by the teams in order to realize simulated matches against others.

II. The teams must be able to use their own visualizer.

III. The solution must work regardless of the programming language a team has developed its software in.

IV. The solution must work regardless of the operating system (OS) a team has developed its software on.

V. A fixed flow of information should be defined which ensures that, regardless of any differences in the simulation software of the teams, a simulated match can be played.

VI. Both teams must agree on one ball. That is, they use the same $[x_b, y_b, z_b, \dot{x}_b, \dot{y}_b, \dot{z}_b]$ where x_b, y_b and z_b represent the position of the ball in three dimensional space. $\dot{x}_b$, $\dot{y}_b$ and $\dot{z}_b$ are the corresponding velocity components.

VII. A team must have access to the pose and the corresponding velocity components of the robots of the opponent. That is, $[x_i, y_i, \theta_i, \dot{x}_i, \dot{y}_i, \dot{\theta}_i]$ must be shared between the teams where x_i, y_i, $\dot{x}_i$ and $\dot{y}_i$ refer to the position and linear velocity components of the ith robot, respectively. θ_i is the robot heading which is defined as the angle between the X-axis of the world coordinate frame and the x-axis of the robot coordinate frame of the ith robot measured in counterclockwise direction. Finally, the angular velocity of the ith robot in counterclockwise direction is given as $\dot{\theta}_i$. Figs. 1 and 2 show the world and robot coordinate frame, respectively [11].

VIII. The robots of both teams must agree on one common notion of simulation time.

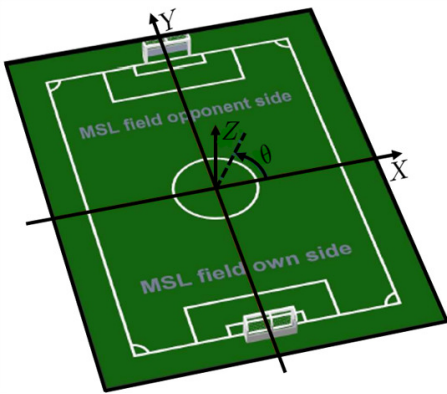IX. The robots of both teams must listen to the same refereeing commands.

Fig. 1: Three dimensional Cartesian world coordinate frame.

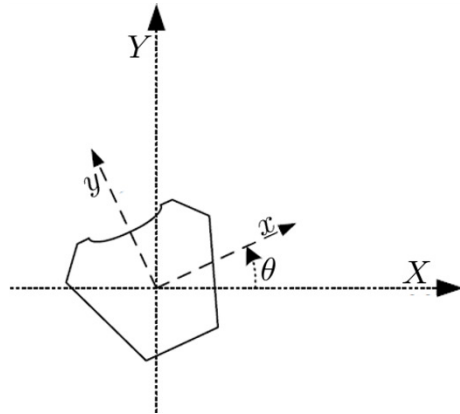Fig. 2: Robot coordinate frame in the XY-plane of the world coordinate frame.

Following the agreements made during the MSL workshop of 2015 [11], positions are defined in meters, angles in radians, linear velocities in meters per second and angular velocities in radians per

second. Moreover, positions and linear velocities should have a minimum resolution of three decimal digits and angles and angular velocities a minimum resolution of four decimal digits.

Furthermore, Fig. 1 implies that each team uses its own definition of the world coordinate frame. Therefore, before using the data of the opponent, the transformations

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \theta' = \theta + \pi (\bmod 2\pi)$$

should be applied. Note that in the second equation $(\bmod 2\pi)$ refers to the modular arithmetic according to which θ stays within the bounds $[0, 2\pi]$.

3.2 Preferences

In addition to the requirements, preferences are defined regarding aspects that teams would like the simulation software to possess, but which are not essential. These preferences are given below:

  I. A simulation should not depend on the availability of another team.

 II. A simulation can be performed on one computer.

III. The solution should not limit a simulation with regards to simulation speed.

The first two preferences are based on convenience. The first grants teams unlimited possibilities to simulate matches. The second preference might be difficult to realize taking the computational effort into account that is required to run all processes involved. Moreover, it might be advantageous to consider distributed computing where components that together form the architecture for a simulated match are distributed over a computer network. The final preference prevents a computational expensive solution that forces the simulation to slow down.

3.3 Licensing

Some of the teams are concerned that teams might be hesitant to provide their latest software release. Since the relevance of a simulated match would be compromised otherwise, the following license item has been defined:

  I. Teams that want to make use of the solution need to provide a binary release of their software that was used during the last official RoboCup tournament together with the corresponding and complete documentation.

Essentially, due to the open nature of RoboCup, the teams are obliged to release their design after each official annual RoboCup tournament. Therefore, providing a binary release of their software to be used for simulated matches should not receive much resistance. Actually, this should work as motivation for the teams to continuously evolve. Additionally, the licensing item also prevents that teams profit from the solution without providing a version of their software at all.

3.4 Initial Architecture

Because of the large diversity in the simulators with regards to what is included in a simulation, it is nearly impossible to re-use one simulator as a league-standard that suits all teams. Moreover, the differences also prevent that combinations can be achieved effortlessly where the opponents in one simulator are controlled by the software of another team.

In Fig. 3, an architecture is presented that connects the simulation software of two teams by means of an application referred to as the middleware. This architecture allows teams to use their own simulator and visualizer. Thereby they are free to determine the level of realism in their simulation software. In this architecture, the simulators send information of their own team to the middleware. In return, they receive information on the opponent from the middleware. These flows

of information are represented by the solid arrows between the middleware and both simulators. The solid arrow between the simulator and the visualizer represents the simulator providing the visualizer with new information to display.
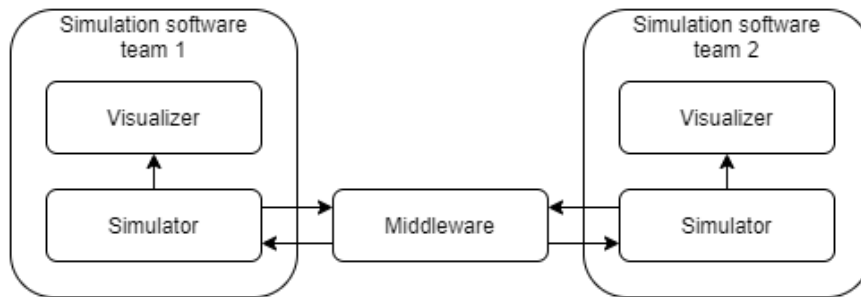

Fig. 3: The initial architecture that is intended to be used to realize simulated matches. The middleware communicates with the simulators of the simulation software of two opposing teams. The visualizer displays what happens within a simulator.

This architecture poses an extra challenge with regards to how teams should share their simulation software. A method must be found that enables the teams to easily share their software with others. Moreover, it must be guaranteed that, regardless of the programming language the software was written in and the OS it was produced on, the software of any particular team works in other environments as well. Preferably, this does not involve installing the software of other teams manually as this would cost much effort.

4   Communication and Time Synchronization Protocol

Now that an initial architecture is chosen, two essential protocols are introduced in this section. First, the communication protocol is explained. This protocol defines a flow of information between the middleware and the simulators. Moreover, the protocol covers how the middleware preprocesses the information to improve its quality before it is sent to a simulator. Secondly, the time synchronization protocol is discussed which ensures that the two opposing simulators are synchronized with respect to simulation time.

4.1   Flow of Information

In the requirements it is established what information is needed in order to realize a simulated match, next it is decided who should provide it. During a real match, the teams are required to send information, e.g., state information of their robots, to the refereeing system for logging purposes. This information roughly corresponds to what is needed for a simulated match. It is assumed that all teams obey this rule and thus that this information is readily available to be sent to the middleware.

Information that is distributed by the middleware to the simulators should correspond to what is available during a real match. It is decided to make use of ground truth information. This gives teams the freedom to determine themselves how information is used. For instance, if one of the opponents is out of sight for all robots of a team, then the corresponding information can be disregarded. Moreover, in reality observations will never be perfect, thus teams might want to add noise to the data before using it.

Three information types are distinguished and each type is analyzed separately. The three types are robot, ball and referee information.

4.1.1   Robot Information

From the perspective of one team, only the pose and the corresponding velocity components of the opponent's robots are missing, i.e., $[x_i,\ y_i,\ \theta_i,\ \dot{x}_i,\ \dot{y}_i,\ \dot{\theta}_i]$. Therefore, the teams need to send this information to the middleware. This implies that a simulator is responsible for computing the states of its own robots only, but not those of the opponent.

The only situation in a simulated match where a team might want to alter the state of a robot of the opponent is when collisions between robots are included in their simulator, but not in that of the other team. Currently, adding collisions in a simulation is of low importance. The rules clearly state that robots should try to avoid physical contact [12]. However, direct contact does not necessarily represent an offense if a robot tries to avoid it, but fails to do so. If contact cannot be avoided, then a robot must by all means minimize the impact by decelerating. Additionally, robots should be equipped to recognize situations of physical contact. This includes both direct pushing and indirect pushing with the ball as its medium by continuously exerting a force on the ball. As soon as a robot senses contact of any form with another robot, it should immediately stop its movement and adjust its moving direction. All of the above support the decision to deny a team to alter the state of an opponent's robot.

If in the future the league evolves to be more contact-based as in human soccer matches, then it will be interesting to include this in the simulations as well. This can then be achieved by, for instance, also sharing force components.

During real matches, the robots acquire images of their surroundings from which they determine the positions and velocities of the opponents. It is possible that teams have incorporated such a perception model in their simulation software as well by using images obtained from the visualizer. If this is the case, the obtained opponent information should first be portrayed in the visualizer instead of using it directly. Afterwards, they can use their perception model to obtain information on the opponents.

4.1.2 Ball Information

In the proposed architecture, both teams simulate their own ball. They should both send the corresponding information, i.e., $[x_b, y_b, z_b, \dot{x}_b, \dot{y}_b, \dot{z}_b]$, to the middleware which on its turn decides which ball to use. If a team is in possession of the ball, then their ball information is used since it is assumed that they will have solid knowledge on the position and velocity of the ball. Their ball will be used until a robot of the opposing team takes over possession, then the ball information of the other team will be used. Note that this also implies that when a robot passes or shoots the ball or when the ball is released accidentally, e.g., due to malfunctioning of the ball handling system, the ball information of the team that was last in possession will be used. The middleware tracks which team is in possession of the ball or was last to be in possession of the ball by using a ball engaged status. This status denotes whether a robot is or is not in possession of the ball and must be shared by both teams for all of their robots in the simulation. In case two robots of the opposing teams claim to be in possession of the ball simultaneously, then the middleware will decide which ball is used. The middleware determines the distance from both robots to the ball and the ball information of the team whose robot is closest is used. In the unlikely event that the distance is equal, the middleware will randomly decide which ball to use.

It is assumed that all teams are able to reset their own state of the ball to the provided position and velocity components when new ball information is received. This is a valid assumption since simulations are typically used to recreate game scenarios such as free kicks where the ball is given a specific position and its corresponding velocity components are set to zero.

Similarly as for the robot information, if a team uses a perception model to obtain the position and velocity of the ball, then they should first show the ball in their visualizer in order to acquire an image that is fed to their perception model.

4.1.3 Referee Information

In a real match, a human referee is in control and communicates with the robots of both teams through a refereeing system called the RefBox. The referee uses a predetermined set of commands that correspond to game scenarios, e.g., a corner awarded to one of the teams. Teams have designed their software to listen when a command is given. The RefBox is an existing project that is available for the teams on the GitHub page of the MSL [13]. It is the responsibility of the teams to make sure that their simulators can cope with the RefBox commands as well. If this is achieved, then the RefBox can be connected to the simulators and a simulated match can be controlled. Since the RefBox is a separate project, it will not be discussed any further.

4.2 Preprocessing of Information

The simulators of both teams have their own notion of simulation time and they can advance through time at different rates. When both the robots and the ball attain high velocities, the corresponding information loses its relevance quickly. Therefore, it is important that the simulators stay synchronized and that the information of the opponent is updated regularly. How synchronization is achieved will be discussed later, the latter will now be elaborated.

To start with, it is required that both teams share information to the middleware regularly. Moreover, it is necessary that, together with the shared robot and ball information, a timestamp t is added that represents the simulation time at which the information was obtained. It is assumed that the simulation time is monotonically increasing. It is the teams' responsibility to ensure this. That is, if a simulator is reset and internally restarts from $t = 0$, then the final simulation time before the reset must be added to the new simulation time before sending it to the middleware.

The middleware makes use of circular buffers in which it stores N sets of information of both teams. Thus, for each team it has N sets of data available accompanied by the timestamps $t_1, t_2, ..., t_N$. In addition, the middleware is equipped with an inter- and extrapolation algorithm. Rather than providing the teams with new information at a fixed rate, the teams can request new information from the middleware. This prevents that a simulator is overloaded with information.

When a team requests new information from the middleware, the time of asking, t_a, must be specified which corresponds to the current simulation time of their simulator. The middleware determines whether inter- or extrapolation is needed. If the asking time is in between the timestamps of two sets of information, thus $t_i < t_a < t_{i+1}$ where $i = [1, 2, ..., N-1]$, then linear interpolation is used. Linear extrapolation is used when the asking time is lower than the timestamp of the oldest set of information or higher than the timestamp of the newest set of information, thus $t_a < t_1$ or $t_a > t_N$. In this case, either the oldest two sets of information, corresponding to the timestamps t_1 and t_2, or the newest two sets of information, corresponding to the timestamps t_{N-1} and t_N, are used. It may occur that the time of asking is equal to the timestamp of one of the sets of information, thus $t_a = t_j$ where $j = [1, 2, ..., N]$. Then, there is no need to preprocess the information and the original information is sent.

Using linear inter- or extrapolation will result in a more accurate representation of the opponent's information in comparison to an alternative where the information corresponding to the timestamp that is closest to the time of asking is used. Since information should be provided regularly to the middleware, the time difference between two consecutive sending instants will be small. Therefore, a linear approximation is suited.

4.3 Time Synchronization

It is essential that the simulation times of both simulators are synchronized. However, the simulation speed of a simulator depends on a combination of multiple factors such as the difficulty of the game scenario for which new robot and ball states need to be determined, and the computing power of the computer that is used. Moreover, no control input is available that allows to alter the simulation speed, unless the possibility to do so is incorporated by the teams. Together, this supports the decision to make use of on-off control where the simulation software of a team is temporarily paused if its simulation time exceeds that of the other team by some threshold ϵ_1.

It is unlikely that the simulation software is paused instantly. It is expected that there will be some delay between the moment that the simulation software is being paused and the moment that it is actually paused because of high overhead. Therefore, to prevent rapid switching between a paused and active state of the simulation software around the threshold ϵ_1, a second threshold ϵ_2 is introduced for which it holds that $\epsilon_2 < \epsilon_1$. The paused simulation software is resumed once the difference in simulation time is below or equal to this second threshold ϵ_2.

In Fig. 4, a visual representation is given. The simulation software of two opposing teams 1 and 2 are represented by S_1 and S_2, respectively. The corresponding simulation times are denoted by t_1 and t_2. In the top state, both S_1 and S_2 are active and the difference in simulation time is below or equal to the first threshold, $\|t_1 - t_2\| \leq \epsilon_1$. In the left state, S_1 is paused and S_2 is active, the difference in simulation time exceeds the second threshold, $\|t_1 - t_2\| > \epsilon_2$, and the simulation time of S_1 is larger than that of S_2, $t_1 > t_2$. In the right state, S_1 is active and S_2 is paused, the difference in simulation time exceeds the second threshold, $\|t_1 - t_2\| > \epsilon_2$, and the simulation time of S_2 is larger than that of S_1, $t_2 > t_1$. Initially, both teams start in the top state at $t_1 = t_2 = 0$.

The transitions e_1 to e_6 denote a switch from one state to another because of a change in conditions and are listed in Table 1. Transition e_1 occurs if the difference in simulation time exceeds the first threshold, $\|t_1 - t_2\| > \epsilon_1$, and if the simulation time of S_1 exceeds that of S_2, $t_1 > t_2$. In result, S_1 is paused. Transition e_2 is similar. Again, the difference in simulation time exceeds the first threshold, $\|t_1 - t_2\| > \epsilon_1$, but now the simulation time of S_2 exceeds that of S_1, $t_2 > t_1$. Thus, S_2 is paused. To return from the left or right state to the top state by means of transitions e_3 or e_4, respectively, the difference in simulation time must be below or equal to the second threshold $\|t_1 - t_2\| \leq \epsilon_2$. Then, the previously paused simulation software is resumed. Finally, it is possible to switch between the left and right state. From the left state, transition e_5 is made to the right state in case the difference in simulation time remains above the second threshold, $\|t_1 - t_2\| > \epsilon_2$, but the simulation time of S_2 now exceeds that of S_1, $t_2 > t_1$. In result, S_1 is resumed and S_2 is paused. The exact opposite applies to transition e_6 from the right to the left state.
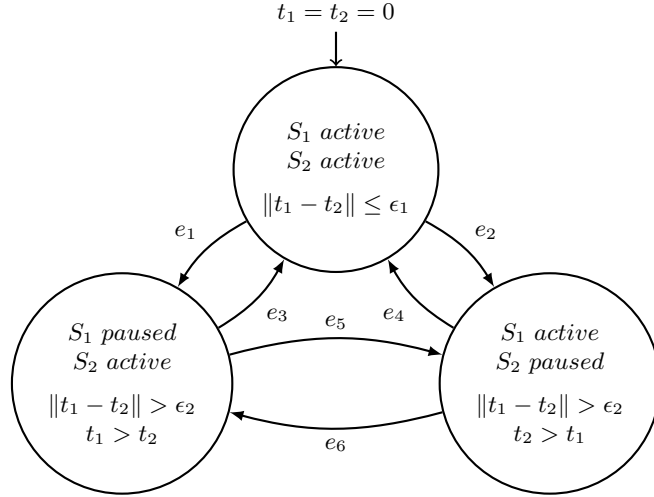
Fig. 4: A visual representation of the three states of a simulated match. In the top state, both S_1 and S_2 are active and $\|t_1 - t_2\| \leq \epsilon_1$, in the left state S_1 is paused and S_2 is active, $\|t_1 - t_2\| > \epsilon_2$ and $t_1 > t_2$, and in the right state S_1 is active and S_2 is paused, $\|t_1 - t_2\| > \epsilon_2$ and $t_2 > t_1$. The transitions e_1 to e_6 are explained in Table 1.

Table 1: Description of the transitions e_1 to e_6 as shown in Fig. 4.

event	conditions	action
e_1	$\|t_1 - t_2\| > \epsilon_1, t_1 > t_2$	pause S_1
e_2	$\|t_1 - t_2\| > \epsilon_1, t_2 > t_1$	pause S_2
e_3	$\|t_1 - t_2\| \leq \epsilon_2$	resume S_1
e_4	$\|t_1 - t_2\| \leq \epsilon_2$	resume S_2
e_5	$t_2 > t_1$	resume S_1, pause S_2
e_6	$t_1 > t_2$	pause S_1, resume S_2

5 Architecture Definition

In this section the final architecture is explored. First, it is covered how teams should share their simulation software. Next, the final architecture is explained. Finally, the software structure on which the architecture relies is elaborated.

5.1 Sharing of Simulation Software

Preferably, playing a simulated match should not depend on the availability of others. Therefore, teams should share their simulation software with others. Regardless of the combination of the programming language and the OS used by others, a team should be able to access and run the simulation software of another team. It is important that sharing, accessing and running the software will not be much effort. A solution to this problem is to make use of containers. A container allows a developer to isolate an application from its current environment and guarantees that it functions in any other environment as well.

Docker is the industry standard to build, share and run containerized apps [14]. In order to produce a so-called Docker container, a Docker image is used. A Docker image is an executable package of software that includes everything, i.e., scripts, dependencies and settings, that is needed to run the application. An image also provides an isolated filesystem to the Docker container. In order to create a Docker image, a Dockerfile is needed. A Dockerfile is a sequence of instructions that are executed consecutively and is used to assemble a Docker image.

In the context of simulated matches in the MSL, the teams are required to build a Docker image including their simulation software. These images should be made available on the Docker hub, a public library for sharing Docker images. Building a new image containing an updated version of the simulation software requires little effort since the same Dockerfile can be used. That is, assuming only internal changes in the software have occurred while the fundamentals, e.g., the dependencies, remain unchanged.

During a simulated match, multiple Docker containers will be running simultaneously. Docker containers are able to communicate with each other when they are connected to the same Docker network. Time synchronization can be achieved by using the *docker pause* and *docker unpause* commands. These commands suspend or resume all processes in the indicated container, respectively.

Docker runs on Linux, Windows and MacOS. Thus, regardless of which OS a team uses, they are able to containerize their simulation software. The four MSL teams that have been interviewed all use a Linux-based OS. Since this is a popular choice among software developers, it is expected that the majority of the remaining teams make use of some Linux distribution as well. However, in case a team makes use of another OS, Docker can be used in combination with a virtual machine. Virtual machines resemble an entire computer system, i.e., they include an OS. For example, this allows to run a virtual machine hosting a Linux-based OS on a computer that uses a Windows OS. This allows a Linux-based Docker container running in the virtual machine to communicate with a Windows-based Docker container running on the host machine.

5.2 Final Architecture

Fig. 5 shows the final architecture that is used to realize simulated matches. The figure shows five Docker containers that run their own application or group of applications. Two new applications, the so-called pausers, are shown. These pausers are able to pause or unpause a Docker container running the simulation software of a team in case of de- or re-synchronization, respectively. These actions are represented by the dashed arrows. The middleware tells a pauser in case it must perform either of these actions as shown by the solid arrow from the middleware to a pauser.

As discussed in the preferences, it is expected that a distributed computer network will be used. If this is the case, then the containers of a pauser and the simulation software of one team will be running on another computer. It is impossible to pause or unpause a container running on another computer, hence why this is delegated from the middleware to the pauser applications.
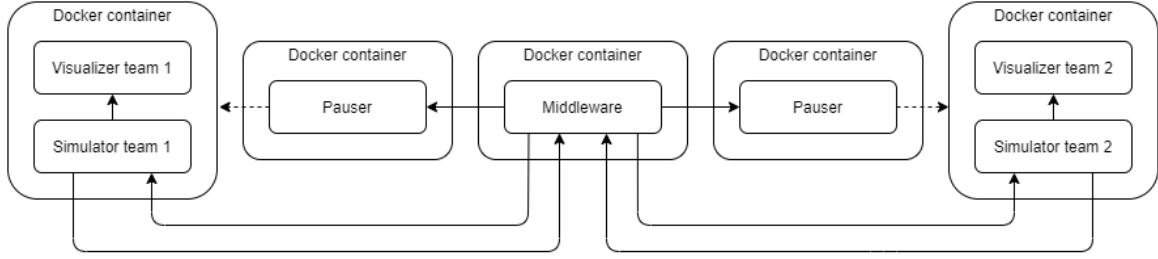
10

Fig. 5: The final architecture that is used to realize simulated matches consisting of five Docker containers that all run their own application or group of applications. Communication is represented by means of a solid arrow, the execution of a Docker command is shown as a dashed arrow.

5.3 Software Structure

Previously, it is defined what information will be shared and who will share it, but not yet how it should be shared. This involves choosing a communication model and a connection type. It is important that the chosen model allows two-way communication. Moreover, no messages should be lost and neither of the teams must be able to gain an unfair advantage because of the connection type. For example, if the middleware always checks for and responds to incoming messages of one team first, then the other team waits indefinitely before their message is handled. Finally, the teams must be able to reconnect to the middleware if accidentally disconnected.

The simulators of the teams and the pausers are connected to the middleware by means of network sockets. A socket is an endpoint for sending and receiving information in a network and a connection can be established regardless of the programming languages used. The socket connections rely on a combination of the transmission control protocol (TCP) as its transport layer and the internet protocol (IP) as the network layer. The transport layer is responsible for the communication between applications while the network layer is responsible for the communication between computers. TCP is a reliable protocol since it guarantees that the information is received exactly the same as that it was sent, i.e., without losses and in the same order. Moreover, TCP is a connection-oriented protocol. In these types of protocols, a connection must be established before data is exchanged.

The middleware performs three phases: initialization, communication and time synchronization. Initialization starts by the middleware opening a socket for the pausers. It then waits for the pausers to make a connection and once both have connected, it opens the socket for the teams. Note that both sockets are designed to accept two incoming connections and before both positions are filled, the middleware will not continue.

Next, the middleware enters the communication phase. It starts by checking whether there is an incoming connection request on the socket of the teams. Once both simulators have connected, a start signal is given after which a simulated match commences. Next, the middleware listens for activity on the socket of the teams. This can either be that one of the teams has disconnected, or that a message is received on the socket connection. In case of a disconnect, the container of the other team is paused. The team that lost connection can reconnect to the middleware. This new connection will be accepted when the middleware enters the communication phase again. Afterwards, the simulations can continue. Two types of messages can be received. Firstly, a team can ask for new information at a particular time. The middleware first preprocesses the information based on the provided asking time before it replies by sending the information. Secondly, a team can also send new information. The middleware stores this information in its circular buffers. If both teams have sent a message to the middleware, then both messages will be handled in the same communication phase. This prevents that the middleware prioritizes either of the teams. Next, a switch is made to the time synchronization phase.

The time synchronization phase represents the time synchronization protocol. Based on the simulation times that are received during the communication phase, it is decided whether action is required and if the simulation software of a team must be paused or resumed. If so, the middleware orders the corresponding pauser to execute this action. Afterwards, the middleware returns to the

communication phase. The pseudocode of the time synchronization protocol is given below and corresponds to Fig. 4 and the transitions given in Table 1. For clarity, some cases where no action is required are added as well.

Algorithm 1 Time Synchronization Protocol

1: **if** $\|t_1 - t_2\| \leq \epsilon_1$, S_1 is active and S_2 is active **then**
2:      Nothing changes
3: **else if** $\|t_1 - t_2\| > \epsilon_1$, $t_1 > t_2$, S_1 is active and S_2 is active **then**
4:      Pause S_1
5: **else if** $\|t_1 - t_2\| > \epsilon_1$, $t_2 > t_1$, S_1 is active and S_2 is active **then**
6:      Pause S_2
7: **else if** $\|t_1 - t_2\| > \epsilon_2$, $t_1 > t_2$, S_1 is paused and S_2 is active **then**
8:      Nothing changes
9: **else if** $\|t_1 - t_2\| > \epsilon_2$, $t_2 > t_1$, S_1 is active and S_2 is paused **then**
10:      Nothing changes
11: **else if** $\|t_1 - t_2\| \leq \epsilon_2$, S_1 is paused and S_2 is active **then**
12:      Resume S_1
13: **else if** $\|t_1 - t_2\| \leq \epsilon_2$, S_1 is active and S_2 is paused **then**
14:      Resume S_2
15: **else if** $\|t_1 - t_2\| > \epsilon_2$, $t_2 > t_1$, S_1 is paused and S_2 is active **then**
16:      Resume S_1 and pause S_2
17: **else if** $\|t_1 - t_2\| > \epsilon_2$, $t_1 > t_2$, S_1 is active and S_2 is paused **then**
18:      Pause S_1 and resume S_2
19: **end if**

For the teams to connect to and communicate with the middleware, software has been written that needs to be implemented in the existing software of the teams. The software works as follows: First, a team connects to the socket connection set up by the middleware. Afterwards, it waits for a start signal before its simulation can start. Once started, the teams can either ask for new information or send new information. If a team asks for new information, the software automatically switches to a read-state where it waits to receive the information.

All software is written in the C++ programming language. Since teams are free to develop their software in whatever language they are comfortable with, all information should be shared in a JavaScript Object Notation (JSON) format. Not only is this format lightweight, it is completely language independent and thus enables communication between software written in different programming languages.

6 Results

In this section, the architecture is applied to simulate a match between two copies of Tech United. Thereby, the middleware, the pausers and the additional software to be implemented by the teams are verified. First, it is explained how the produced software is implemented in the software of Tech United. Next, the setup and the results of a simulated match are discussed. Finally, synchronization with respect to the simulation time of the two connected simulators is verified.

6.1 Software Implementation for Tech United

During matches, the robots of a team rely on cooperation for optimal behavior and decision-making. A common approach to realize such cooperative sensing is to produce a blackboard [15]. In the MSL, various teams, among which Tech United, have incorporated the Real-Time Data Base (RTDB) produced by CAMBADA for inter-robot communication. Each robot has a replicate blackboard that is divided in a private area for local information and a shared area for global information. The shared area is divided in multiple sections, one for each robot. One of the sections is designated to the robot itself and is broadcast to its peers while the other sections are filled with information received from the others [16], [17]. The RTDB is also suitable for simulations [18].

For Tech United, it is possible to access the RTDB externally. Therefore, it is unnecessary to incorporate the produced software in the software of Tech United. Alternatively, a separate application, referred to as the info exchanger, is produced that facilitates the communication between the simulator and the middleware. In particular, the info exchanger reads information from a database and shares it to the middleware or receives new information from the middleware and publishes it to a database. Once information is stored in the database, e.g., the poses of the opponents, then it will used for future decisions. Additionally, the visualizer is updated as well.

6.2 Tech United vs. Tech United

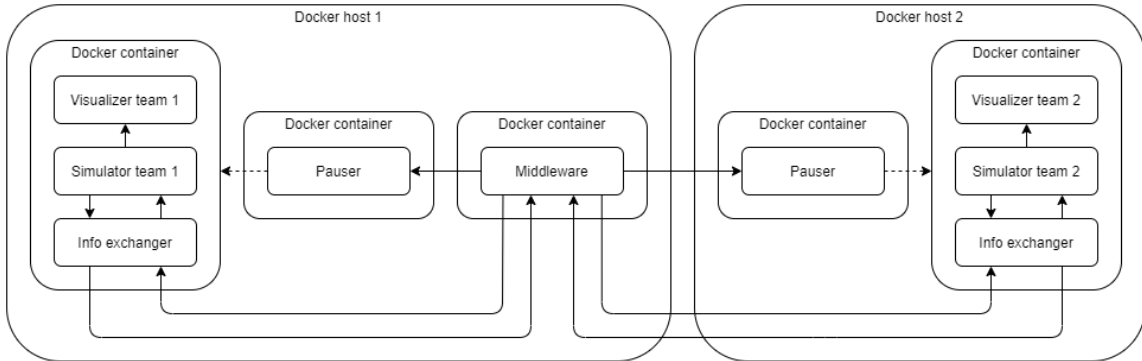The architecture used for a simulated match between two copies of Tech United is shown in Fig. 6.

Fig. 6: The architecture that is used to realize a simulated match between two copies of the Tech United software. Five Docker containers are shown that are divided over two Docker hosts. In the containers where the simulation software is included, an info exchanger application is added which enables communication between the middleware and the simulator.

In comparison to Fig. 5, there are some changes for this particular configuration. First, the info exchanger is added which enables the connection and communication between the middleware and a simulator. It is impossible to run multiple instances of the simulation software of Tech United on the same computer because the communication between the robot software and the simulator of one team will interfere with that of the other team. Therefore, the containers are divided over two computers, the Docker hosts. In order for containers on different hosts to communicate with each

other, they must be part of the same Docker swarm and be connected to the same overlay network, a network type that allows communication between containers on different hosts.

A wired Ethernet connection is used between the Docker hosts because of its low latency in comparison to a wireless alternative. Important aspects of a connection are network speed, which is the amount of data that is sent or received in bits per second, and latency, which is the delay between the moment data is sent by one computer and received by another one. High latency can compromise network speed. However, from testing it is found that the average round-trip latency, which is the time it takes for information to reach its destination and return, of 100 samples is only 0.64 ms. Thus, the expected latency is only 0.32 ms.

The rate at which new information is requested by the info exchanger is set to resemble reality. During a real match, the vision model of a Tech United robot acquires and processes sensor data at a rate of 50 Hz. The info exchanger requests new information at the same frequency. Additionally, new information is sent at this frequency as well, but requesting and sending new information alternate each other. Thus, every 10 ms in simulation time either of the two actions is performed. Network speed is optimized by compressing the data into the aforementioned JSON format.

It is found that there exists a delay in the *docker pause* command. This can be problematic if a *docker unpause* command follows shortly after. What happens is that the pause command is still being processed while the unpause command is already executed. The latter returns that the container cannot be unpaused, since it is not paused. Hereafter, the pause command is executed which pauses the container while it should be active again. From testing it is concluded that there should be a minimum of 250 ms between a pause and an unpause command. Note that this value depends on the hardware that is used and might be different in another setup.

The above delay can compromise a simulation. In order to prevent this from happening, the thresholds ϵ_1 and ϵ_2 are tuned such that the commands do not follow each other within 250 ms apart. However, in contrast to the minimum time between two commands, which is based on real time, the thresholds are based on simulation time. Therefore, no relation between the two can be found. The thresholds $\epsilon_1 = 200$ and $\epsilon_2 = 50$ ms are found to be suitable for a simulated match.

Finally, it is chosen to set the length of the circular buffer to $N = 25$. Since new data arrives approximately every 20 ms, the sets together span about 500 ms. Thereby, the middleware should have more than sufficient data available to guarantee quality of the information since it accounts for situations where a simulator is being paused, but remains active for a while due to the delay.

In Fig. 7, a visual representation is given of a moment in a simulated match as shown on two different computers. The computers both run a copy of the software of Tech United. The robots in one simulator are the opponents in the other simulator and are shown as circles. Moreover, it is seen that both simulations are closely synchronized. That is, the positions of the ball and the robots on one computer and the corresponding ball and obstacles on the other computer are nearly identical.

(a) Computer 1.   (b) Computer 2.
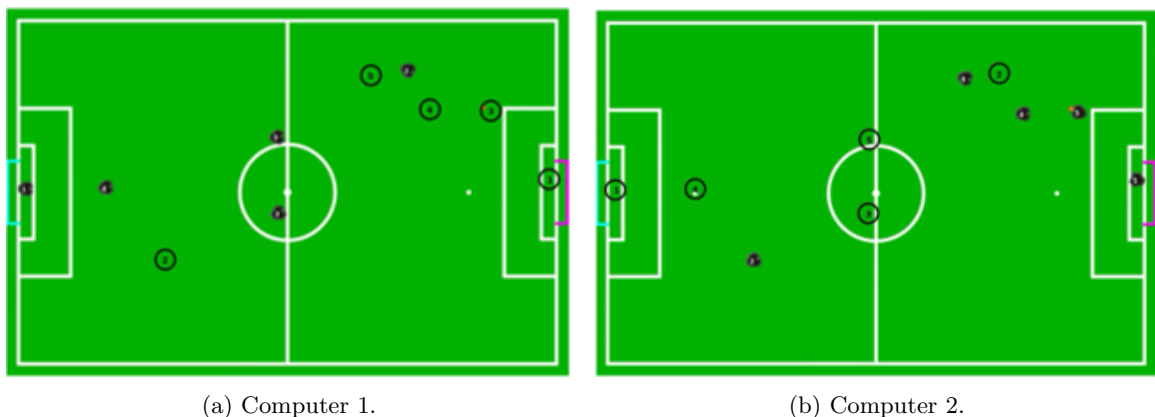
Fig. 7: An instant in a simulated match between two copies of the Tech United simulation software based on the architecture proposed in Fig. 6. The robots on one computer are shown as opponents, represented by circles, on the other computer.

6.3 Verification of Time Synchronization

Naturally, the difference in simulation time between two copies of Tech United will remain small since both are computationally equally expensive to run. To force some action from the time synchronization protocol, and thus the pausers, one of the containers running the simulation software is temporarily paused. The results are shown in Fig. 8. Here, the progression of the simulation times t_1 and t_2 of the two opposing simulation software copies S_1 and S_2 and the difference in simulation time are shown, respectively. Both are plotted over the samples k. A sample is produced every time new information is received from either of the teams and consists of the new simulation time of that team and the most recent simulation time of the other team. As seen in the right plot of Fig. 8, the difference in simulation time exceeds the threshold ϵ_1 four times. Apart from these moments, the difference in simulation time remains below 100 ms almost the entire time.
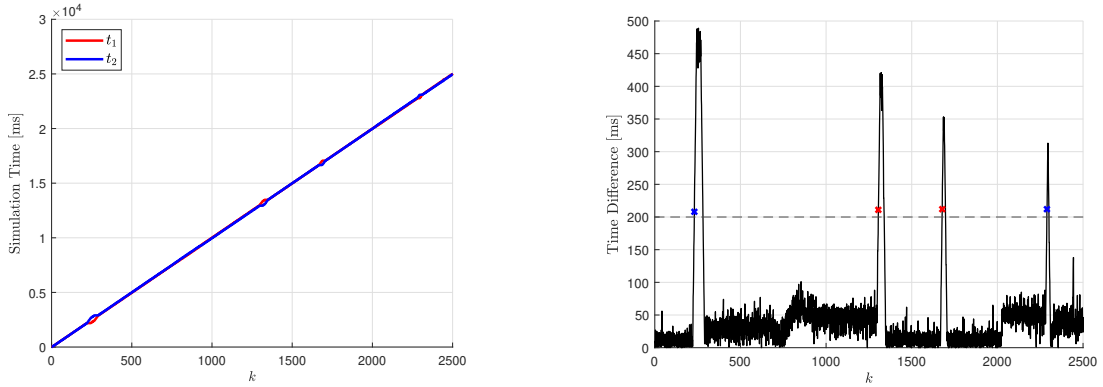

Fig. 8: The progress of simulation time (left) and the difference in simulation time (right) in a match.

Fig. 9 zooms in on a moment where the simulation times drift apart. At $k = 1667$, the container corresponding to the blue graph is paused resulting that at $k = 1678$ the difference in simulation time exceeds ϵ_1. Thus, the container of the red graph needs to be paused. However, because of the delay in the *docker pause* command, the container will remain active for some time as is observed. At $k = 1685$, the container of the blue graph resumes. Then, at $k = 1690$, the container of the red graph is finally paused. From here, the difference in simulation time starts to decrease again. At $k = 1705$, the difference in simulation time is below the second threshold ϵ_2 and thus the container of the red graph is resumed. It is seen that there exists a small delay in the *docker unpause* command as well since it takes till $k = 1707$ before the container actually continues.
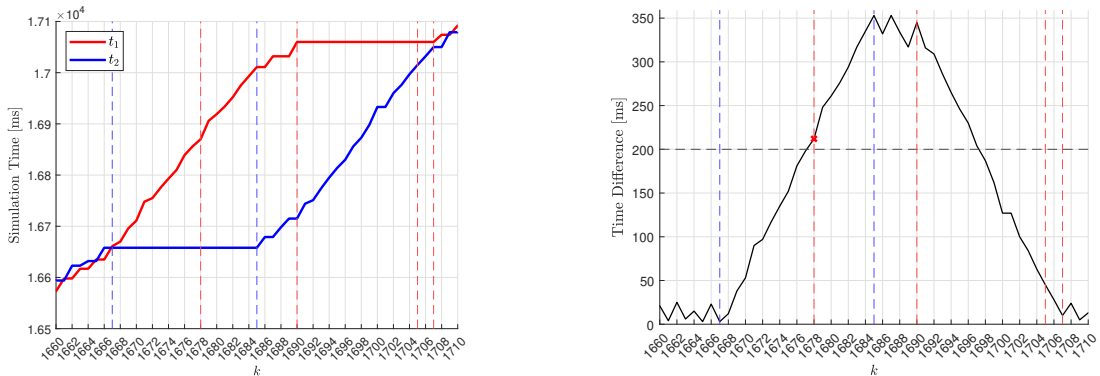

Fig. 9: The progress of simulation time (left) and the difference in simulation time (right) in a match zoomed in on $k = 1660$ to $k = 1710$.

7 Conclusion and Future Work

This dissertation proposes and verifies an architecture implemented in a Docker environment that is suited to realize simulated matches in the RoboCup MSL. The architecture allows teams to continuously develop their simulation software in a way they seem fit. Moreover, little effort is required by the teams to be able to participate in simulated matches. Finally, deploying the software into a Docker image requires little effort and updates in the software are easily implemented in the Docker environment as well.

However, there are still some improvements to be made. At the time, it is possible to simulate regular game play, but not an official match because the RefBox is not incorporated into the architecture yet. Consequently, this implies that it is not yet possible for teams to easily simulate particular scenarios such as corners or free kicks against others. For now, this can be circumvented by adapting the initial position of the robots and the ball. However, this will not be effortless and for each game situation a new Docker image has to be produced.

Secondly, the linear inter- and extrapolation algorithms in the communication protocol are vulnerable to low update rates. That is, the lower the update frequency, the lower the quality of the provided information by the middleware, especially in situations where the ball or the robots attain high velocities. A solution is to define a minimum update rate. It is suggested that teams send new information at least 20 times per second. An improvement to the protocol would be to replace the linear approximations by a higher order algorithm. However, these protocols are computationally more expensive. Alternatively, the current protocol can be complemented with such a sophisticated algorithm and switch from the linear approximations to it in case of low update rates.

On the other hand, very high update rates also compromise the solution. From testing it is found that, on average, a time increment in the Tech United software is 11 ms. This means that an update rate of approximately 90 Hz would be possible. However, requesting and sending new information at such high rates is computationally more expensive. As a result, a simulated match will not be able to run smoothly and the robots and the ball will have a jerky movement in the visualizer. Based on experiments using the Tech United software, a maximum rate of 50 Hz is advised.

Next, the architecture can become more robust to the delays in the *docker pause* and *docker unpause* commands. It would be an improvement if a pauser first externally obtains the state of the container to check if it is paused before an unpause command is executed rather than assuming it is instantaneously paused. Since the delays depend on the computational power of the hardware, the architecture will additionally be more robust to hardware changes as well.

The info exchanger application designed for Tech United can be improved as well. Particularly, the accuracy of the shared information and the timestamp can be increased. For each piece of information stored in the RTDB, the time since it was published can be derived. For an update frequency of 50 Hz, the age of information is approximately 10 ms. At the time, this age is neglected.

Finally, even though the MSL is an open-source league, it is understandable that teams might be hesitant to share a Docker image that includes sensitive information exposing their strategies or tactics. Therefore, it should be explored if it is possible to restrict access to certain files or directories inside a Docker image. Another solution would be that teams encrypt these files themselves or by including it more securely in their software. For now, a gentleman's agreement should be made stating that teams will not explore the contents of a Docker image of others, but only use the image for the mere purpose of simulating a match.

To ensure that the architecture will be used in the MSL, it is essential that other teams get involved as quickly as possible. At the time of writing, this is work in progress. In collaboration with Robot Sports, the developed software for connecting to and communicating with the middleware is being incorporated in their software and afterwards their simulation software will be deployed to Docker. It is expected that once the first simulated match between Tech United and Robot Sports can be played, other teams will be motivated to participate as well. A manual has been written that explains how the produced software should be incorporated and used. Additionally, this manual supports the teams on how to produce a Docker image. Another manual has been written which contributes to the transfer of this project to any successor. This manual elaborates how to build the images of the middleware and the pausers and how to start a simulated match.

References

[1] RoboCup Middle Size League Homepage: `https://msl.robocup.org`. Last accessed Nov 2020

[2] Yao, W., Dai, W., Xiao, J., Lu, H., Zheng, Z.: A Simulation System Based on ROS and Gazebo for RoboCup Middle Size League. In: 2015 IEEE International Conference on Robotics and Biomimetics (ROBIO), pp. 54-59. (2015) https://doi.org/10.1109/ROBIO.2015.7414623

[3] The RoboCup Soccer Simulator Users Manual for the 2D Simulation League: `https://rcsoccersim.github.io/manual/index.html`. Last accessed Oct 2020

[4] RoboCup Soccer Simulation 3D League Rules for 2021: `https://cdn.robocup.org/ssim/wp/2021/06/Rules_RoboCupSim3D2021.pdf`. Last accessed Oct 2020

[5] Virtual Humanoid Soccer Competition: `https://humanoid.robocup.org/hl-2021/v-hsc/`. Last accessed Oct 2020

[6] Simulation Setup for the Virtual Small Size League Tournament: `https://github.com/RoboCup-SSL/ssl-simulation-setup`. Last accessed Oct 2020

[7] Agent Development Framework Manual for the RoboCup Rescue Simulation League: `https://roborescue.github.io/rcrs-docs/rcrs-adf/manual.pdf`. Last accessed Oct 2020

[8] Beck, D., Ferrein, A., Lakemeyer, G.: A Simulation Environment for Middle-Size Robots with Multi-level Abstraction. In: RoboCup 2007: Robot Soccer World Cup XI. Lecture Notes in Computer Science, vol 5001, pp. 136-147. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68847-1_12

[9] Zhou, Z., Yao, W., Ma, J., Lu, H., Xiao, J., Zheng, Z.,: Simatch: A Simulation System for Highly Dynamic Confrontations Between Multi-Robot Systems. In: 2018 Chinese Automation Congress (CAC), pp. 3934-3939. (2018). https://doi.org/10.1109/CAC.2018.8623698.

[10] RoboCup MSL GitHub Page `https://github.com/RoboCup-MSL/MSL-Simulator`. Last accessed Nov 2020

[11] Data Structure for World Model Sharing in MSL: `https://msl.robocup.org/wp-content/uploads/2018/08/MSL_WMDataStruct.pdf`. Last accessed Mar 2021

[12] MSL Rulebook 2021 v22.0: `https://cdn.robocup.org/msl/wp/2021/02/Rulebook_MSL2021_v22.0.pdf`. Last accessed Mar 2021

[13] The RefBox Project: `https://github.com/RoboCup-MSL/RefBox2015`. Last accessed Mar 2021

[14] Docker Homepage `https://www.docker.com/`. Last accessed Aug 2021

[15] António, J., Neves, R., Azevedo, J., Cunha, B., Lau, N., Silva, J., Santos, F., Corrente, G., Martins, D., Figueiredo, N., Pereira, A., Almeida, L., Lopes, L., Pinho, A., Rodrigues, J., and Pedreiras, P.: CAMBADA soccer team: from robot architecture to multiagent coordination. https://doi.org/10.5772/7353

[16] Almeida L., Santos F., Facchinetti T., Pedreiras P., Silva V., Lopes L.S.: Coordinating Distributed Autonomous Agents with a Real-Time Database: The CAMBADA Project. In: Aykanat C., Dayar T., Körpeoğlu İ. (eds.) Computer and Information Sciences - ISCIS 2004. ISCIS 2004. Lecture Notes in Computer Science, vol 3280. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30182-0_88

[17] Santos, F., Almeida, L., Pedreiras, P., and Lopes, L.: A real-time distributed software infrastructure for cooperating mobile autonomous robots. In: 2009 International Conference on Advanced Robotics, pp. 1-6, IEEE, Munich, Germany (2009)

[18] Pedrosa, P.: Simulated environment for robotic soccer agents. (2010)

Declaration concerning the TU/e Code of Scientific Conduct for the Master's thesis

I have read the TU/e Code of Scientific Conduct[i].

I hereby declare that my Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct
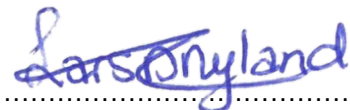
Date

29-09-2021
....

Name

L.D. Nijland
....

ID-number

0958546
....

Signature

....

Submit the signed declaration to the student administration of your department.