API Specifications for Virtual Soccer Competition [Preliminary]

This is a preliminary document, next major update will be published on March 29th. Expected modifications for this document are mainly clarifications and examples. In case you feel that your team requires support for elements which are not described in this document, contact the TC as soon as possible and we will examine your request.

This document describes how to interface the source code of the teams with the simulator for the competition. More precisely, it focus on:

- the definition of a configuration file provided by team to provide instructions to the simulator
- the protocol used to enable communication between the program controlling the robots and the simulator

In the following, we use the terms:

- *robot* or *robot model* to refer to the virtual robot acting and sensing within the simulated environment and
- *robot control software* to refer to the robot behavior software implemented by teams which runs outside the simulator environment

General aspects

Control of the robots during the game is performed through asynchronous communication based on a custom TCP protocol between the simulator and the robot control software.

The protocol will be based on proto3 messages.

After each step of the physical simulation, the simulator sends a message with all the new data available from the sensors.

The minimal step between two sensors update is chosen as follows:

- Camera Data: minimal time between two consecutive frame is noted `minFrameStep` and is `16ms`
- Other sensors: minimal time between two updates is noted `minControlStep` and will be `4ms` or `8ms` depending on the physical timestep that will be used for simulation.

ROS and ROS2 Bridge

A simple package allowing to make the bridge between the TCP protocol and ROS or ROS2 topics will be provided. It should be executed as part of the robot control software. As a consequence, using those bridges will add a small delay in processing.

Configuration file

The simulator requires a configuration file per team written in `JSON` format containing a dictionary with the number of the robot as key and the following properties as values:

- `robotModelPath: string` *The path to the model that should be used for this robot.*
- `robotStartCmd: string` *The command that should be used to start the robot controller software.*
- `resetConfiguration: Posture` *A dictionary with joint names as keys and their position as values.*
- `halfTimeStartingPose: Pose` *Robot pose at the beginning of a halfTime.*
- `reentryStartingPose: Pose` *Robot pose after a removal penalty.*
- `shootoutStartingPose: Pose` *Robot pose at penalty shootouts.*

A team can choose to start the game with less robots than the maximum allowed in its league, however this cannot be changed during the game. E.g. a team decides to play with only 1 robot and this robot receives a red card, it has to finish the game with no robots on the field.

The `Pose` objects defines the transform used to place a robot, see documentation:

- `translation: float[3]` *The [x,y,z] coordinate at which the robot has to be spawned.*
- `rotation: float[4]` *[rx,ry,rz,angle] with [rx,ry,rz] a normalized vector and angle the the rotation angle in radians*

`Pose` is defined in the following referential:

- Origin is the center of the field at ground level
- X axis points toward the center of the opponent's goal
- Z axis points toward the sky, orthogonal to the ground

The `reentryStartingPose` should be specified with a negative value for y and the robot facing the penalty mark entirely outside of the field. Alternative poses will automatically be adjusted using a mirror symmetry along the XZ plane or offsets along x axis.

Teams are requested to make sure that their initial position is compatible with the rules, otherwise they will risk to commit repeatedly the following offence: *entering the field without the referee's permission* which leads to a red card in case it is repeated twice.

Protocol

Opening the connection to the simulator

To start the communication with the simulator, robot control software should send the following message on a port (to be defined):

- `connect <teamId> <robotId>` with:
  - `teamId`: the identifier used in the GameController
  - `robotId`: the number of the robot that tries to connect (1..N)

The command should fail with an explicit log message if: - If no team with the identifier `teamId` is playing the game - If a connection is already active for the indicated robot.

All connections attempt are logged during the game. In case systematic attempts are logged to connect to a robot from another team, the team performing the illegal connects will be sanctioned.

In case a robot is disconnected during the game, it should be able to reconnect by sending the same initial message.

Closing communication

Once the game is finished, the simulator should send a message announcing the end of the game to each client.

In case a client is not responding anymore or is not treating messages quickly enough, which leads to filling a communication queue on the simulator. The simulator should interrupt the connection with the client in order to maintain the quality of service for other clients and keep simulation at a reasonable speed. Such an interruption should appear in the logs of the simulator.

Sensor messages

The robot will receive a single message with a list of sensor readings of the following types:

- PositionData
  - sensorName : `string`
  - position : `double`
- Accelerometer Data
  - sensorName : `string`
  - values : `double[3]` *[m/s^2], x-axis, y-axis, z-axis*
- Gyro Data
  - sensorName : `string`
  - values : `double[3]` *[rad/s], x-axis, y-axis, z-axis*
- Bump Data
  - sensorName : `string`
  - contact : `boolean`
- Force Data
  - sensorName : `string`
  - value : `double` *[N]*
- Force-3D Data
  - sensorName : `string`
  - values : `double[3]` *[N], x-axis, y-axis, z-axis*

- Force-6D Data *To be confirmed*
- Image Data
  - sensorName : `string`
  - width : `int`
  - height : `int`
  - quality : `int` *100 = no compression, 0 = high compression*
  - data : `char[]` *JPEG encoded data*
  - hFOV : `double` *Horizontal field of view of the robot in degrees*
  - centerX: `double` *Horizontal optical center in pixels*
  - centerY: `double` *Vertical optical center in pixels*
  - radialCoefficients: `double[2]`
  - tangentialCoefficients: `double[2]`

Each team is limited to `50MB/s` for the sum of all the data sent to its robots. In case a team exceeds its budget (on a 1 second based history), packets will be discarded.

Acting messages

Each robot send a single message with a list of commands of the following types:

- SetMotorTarget
  - Type of command: position, velocity, effort
  - Value of the target: `float`
- SetMotorPID: see documentation
  - P : `float`
  - I : `float`
  - D : `float`
- setExposure:
  - sensorName : `string` *sensor should be a camera*
  - exposure : `float` unit: `[J/m^2]`
- setQuality:
  - sensorName : `string`
  - quality : `int` *100 = no compression, 0 = high compression*
- ToggleCamera
  - enable : `boolean`
  - cameraId : `string`
  - NOTE: resolution of camera can't be changed dynamically, however, it is possible to enable or disable cameras at the same position during the game to change between a fixed number of resolutions. Cameras positioned at the same location and orientation with the same field of view are not treated as separate cameras according to the laws of the game.
- setSensorTimeStep:
  - sensorName : `string`
  - timeStep : `int`
  - If request exceeds the minimal timestep for the sensor, minimal

timestep should be used.