API Specifications for Virtual Soccer Competition

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

From now on, this document will only be updated in cases of serious concern.

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 is based on proto3 messages. The current protobuf file can be seen on the Webots github.

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 is `8ms`.

The real-time duration of a `8ms` simulation step might be longer than `8ms` but it is guaranteed to last at least `8ms`.

ROS and ROS2 Bridge

A simple package allowing to make the bridge between the TCP protocol and ROS or ROS2 topics is being developed. If ready before the competition, 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, an example of such a configuration file is provided in the appendix. The configuration file contains a dictionary with two entries:

- `name: string` *The team tag, limited to 12 characters*
- `players: dictionary of Players` *The list of players with their properties*

The `players` entry has the ID of the robot as key and the following properties as values:

- `robotModelName: string` *The name to the model that should be used for this robot. This needs to match the file name submitted in the submission system of the Humanoid League*
- `dockerTag: string` *The Docker image tag containing the control software for this robot. This is optional and defaults to "latest".*
- `dockerCmd: string` *The command used to run the team's Docker image. It will be run like this:* `docker run <some Docker options set by us...> <dockerImg>:<dockerTag> <dockerCmd>`*. The idea of this property is to enable teams to pass different commands based on which robot is used. This property is optional and when this property is not set, the Docker image is run like this:* `docker run <some Docker options set by us...> <dockerImg>:<dockerTag>`*.*
- `halfTimeStartingPose: Pose` *Robot pose at the beginning of a half time.*
- `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 their league by providing less entries to the players dictionary. However this cannot be changed during the game. For example, if 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. A team playing with `n` robots must use robot number `1` to `n`.

The `Pose` objects define 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.*
  - NOTE: while the definition allows to use any rotation axis, only rotations along the z-axis are supported by the AutoReferee (e.g. `0 0 1 3.14`).

`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, the robot control software will have to connect on the address defined with the environment variable `ROBOCUP_SIMULATOR_ADDR` (e.g. `192.168.1.100:10001`). The port defaults to:

- `10000 + ROBOT_ID` for red team
- `10020 + ROBOT_ID` for blue team

The simulator is informed of the IP address of each robot. Therefore, any connection attempts on another robots is denied. Moreover, all connection attempts 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.

When connecting to the simulator, the robot control software will receive a message of size 8 containing the following string:

- `Welcome`: if the connection was accepted by the simulator.
- `Refused`: if the connection was denied.

In both cases, the message is ended by a null character `\0`.

In case a robot is disconnected during the game, it is allowed to reconnect on the same port.

Closing communication

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 interrupts the connection with the client in order to maintain the quality of service for other clients and keep the simulation at a reasonable speed. Such an interruption is noted in the logs of the simulator.

The connection is automatically closed at the end of the game and the robot control software is informed of the current state of the game by the GameController messages.

Sensor messages

The robot will receive regularly `SensorMeasurements` messages containing the following information:

- time : `double` *timestamp at which the measurements were performed in [s]*
- message : `Message[]`
  - message_type: `MessageType` *Error or Warning*
  - text: `string` *textual description*
- accelerometer : `AccelerometerMeasurement[]`
  - name : `string`
  - value : `double[3]` *[m/s^2], x-axis, y-axis, z-axis*
- bumper : `BumperMeasurement[]`
  - name : `string`
  - value : `boolean`
- camera : `CameraMeasurement[]`
  - name : `string`
  - width : `int`
  - height : `int`
  - quality : `int` *UNUSED: only raw images are supported this year. -1 = raw image, 100 = no compression, 0 = high compression.*
  - image : `char[]` *Raw RGB data if quality<0, otherwise JPEG encoded data*
- force : `ForceMeasurement[]`
  - name : `string`
  - value : `double` *[N]*
- force3d : `Force3DMeasurement[]`
  - name : `string`
  - value : `double[3]` *[N], x-axis, y-axis, z-axis*
- gyro : `GyroMeasurement[]`
  - name : `string`
  - value : `double[3]` *[rad/s], x-axis, y-axis, z-axis*
- position_sensor : `PositionSensorMeasurement[]`
  - name : `string`
  - value : `double` *[rad] or [m]*

All the messages sent by the simulator are prefixed with the size of the message that follows. This size is stored on 4 bytes and can be read using `ntohl(uint32_t)`.

Each team is limited to `350 MB/s`. The bandwith is equally distributed between the `N` robots defined in the configuration file of the team. Hence, each robot control software has an individual bandwidth of `350/N MB/s`. In case a robot exceeds its budget (on a 1 second based history), packets will be discarded.

Acting messages

Each robot is allowed to send a single message at each simulation step with a list of commands of the following types:

- motor_positions : `MotorPosition[]`
  - name : `string`
  - position : `double` *[m] or [rad]*
- motor_velocity : `MotorVelocity[]`
  - name : `string`
  - velocity : `double` *[m/s] or [rad/s]*
- motor_force : `MotorForce[]`
  - name : `string`
  - force : `double` *[N]*
- motor_torque : `MotorTorque[]`
  - name : `string`
  - torque : `double` *[N.m]*
- motor_pid : `MotorPID[]` *see documentation*
  - name : `string`

    - PID : `double[3]` *[P,I,D] controller values*
- sensor_time_step : `SensorTimeStep[]`
    - name : `string`
    - timeStep : `int` *Time between two measurements in [ms], disable sensor if 0*
    - If request is lower than the minimal timestep for the sensor (but not 0), minimal timestep will be used.
    - The requested timeStep should always be a multiple of `minControlStep`.
    - NOTE: with respect to Camera sensors:
        * The 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.
- camera_quality : `CameraQuality[]` *UNUSED: only raw images are supported this year.*
    - name : `string`
    - quality : `int` *-1 = raw images, 100 = no compression, 0 = high compression.*
- camera_exposure : `CameraExposure[]`
    - name : `string`
    - exposure : `float` unit: `[J/m^2]`

All messages sent by the robot control software have to be prefixed with the size of the upcoming message stored on 4 bytes in the network byte order. It can be obtained using `htonl(uint32_t)`.

Enabling and disabling sensors

Initially, all sensors are disabled. In order to receive data from the sensors, the robot control software has to explicitly set the time step used to refresh the data, see `sensor_time_step`. It is not required to repeatedly set the `timeStep` of the sensors to receive new data. Once the value is set, the data will automatically be sent to the robot control software in the requested update frequency. Disabling a sensor that has been activated can be achieved by requesting to set the `timeStep` to 0.

Appendix

Example of json configuration

```
{
  "name": "teamA",
  "players": {
    "1": {
      "robotModelName": "robotA",
      "dockerTag": "robotA",
      "dockerCmd": "launchRobot.sh --goalkeeper",
      "halfTimeStartingPose": {
        "translation": [-3.5, -3.06, 0.24],
        "rotation": [0, 0, 1, 1.57]
      },
      "reentryStartingPose": {
        "translation": [-3, -3.11, 0.24],
        "rotation": [0.0, 0, 1, 1.57]
      },
      "shootoutStartingPose": {
        "translation": [2.6, 0, 0.24],
        "rotation": [0, 0, 1, 0]
      }
      "goalKeeperStartingPose": {
        "translation": [-4.47, 0, 0.24],
        "rotation": [0, 0, 1, 0]
      }
    },
```

```
"2": {
  "robotModelName": "robotA",
  "dockerTag": "robotA",
  "dockerCmd": "launchRobot.sh --fieldPlayer",
  "halfTimeStartingPose": {
    "translation": [-3.5, 3.06, 0.24],
    "rotation": [0, 0, 1, -1.57]
  },
  "reentryStartingPose": {
    "translation": [-3, -3.11, 0.24],
    "rotation": [0.0, 0, 1, 1.57]
  },
  "shootoutStartingPose": {
    "translation": [2.6, 0, 0.24],
    "rotation": [0, 0, 1, 0]
  }
  "goalKeeperStartingPose": {
    "translation": [-4.47, 0, 0.24],
    "rotation": [0, 0, 1, 0]
  }
},
"3": {
  "robotModelName": "robotB",
  "dockerTag": "robotB",
  "dockerCmd": "launchRobot.sh --fieldPlayer",
  "halfTimeStartingPose": {
    "translation": [-0.75, -3.06, 0.24],
    "rotation": [0, 0, 1, 1.57]
  },
  "reentryStartingPose": {
    "translation": [-3, -3.11, 0.24],
    "rotation": [0.0, 0, 1, 1.57]
  },
  "shootoutStartingPose": {
    "translation": [2.6, 0, 0.24],
    "rotation": [0, 0, 1, 0]
  }
  "goalKeeperStartingPose": {
    "translation": [-4.47, 0, 0.24],
    "rotation": [0, 0, 1, 0]
  }
},
"4": {
  "robotModelName": "robotB",
  "dockerTag": "robotB",
  "dockerCmd": "launchRobot.sh --fieldPlayer",
  "halfTimeStartingPose": {
    "translation": [-0.75, 3.06, 0.24],
    "rotation": [0, 0, 1, -1.57]
  },
  "reentryStartingPose": {
    "translation": [-3, -3.11, 0.24],
    "rotation": [0.0, 0, 1, 1.57]
  },
  "shootoutStartingPose": {
    "translation": [2.6, 0, 0.24],
```

```
        "rotation": [0, 0, 1, 0]
      }
      "goalKeeperStartingPose": {
        "translation": [-4.47, 0, 0.24],
        "rotation": [0, 0, 1, 0]
      }
    }
  }
}
```