Turtlebot2 Multi-Agent System

VAS Team - July 2018

NAME	EMAIL	MATRICULATION NUMBER
Valentina Cecchini	valentina.cecchini@student.univaq.it	255596
Andrea Perelli	andrea.perelli@student.univaq.it	254758
Stefano Valentini	stefano.valentini2@student.univaq.it	254825

demo: https://youtu.be/1dfWthhUovk

EXECUTIVE SUMMARY

Overview

The project work aims to implement a multi-agent system that is responsible of providing assistance to warehouse personnel.

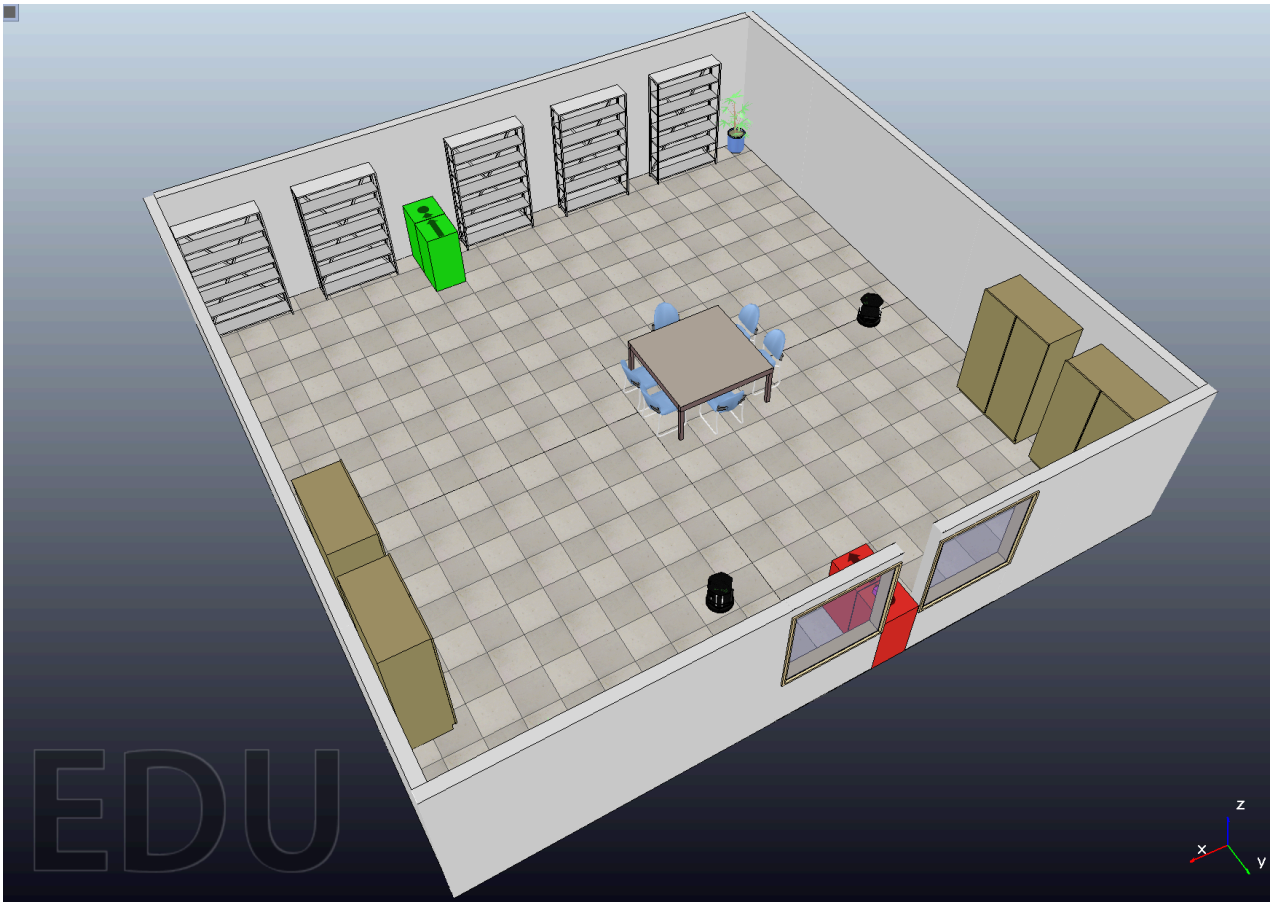
The fleet is composed by Kobuki Turtlebot2 units, the scene represents a warehouse in which a conveyor belt carries packages into the room.

Since the Turtlebot2 units are not provided with arms, one or more operators are needed to help with the fetching/placing of the packages in the respective shelves.

Objectives

- The units must not collide with the objects/people/units in the environment.
- The units must be able to establish (implicit) precedences among each other in order to resolve conflicts and avoid collisions.
- The unit has to understand its status:
  - it has to understand if it is carrying a package or not
  - it has to understand where it has to go (e.g. if it is carrying a package it has to reach the “green” human operator)
- The units have to be able to interact with the human personnel (they have to wait for the human to pick/put the package from/on the unit, etc.).

SCENE



Scene Description

The scene is composed by a producer who spawn objects on a conveyor belt (which simulates the goods arrival), at the end of it a human operator is responsible of putting the packages on the Turtlebot2 units.

On the other side of the room (separated by some ornaments/obstacles), another operator waits for the packages.

When a package is received the operator places it on another conveyor belt which brings it to a sink (that simulates the placing on the shelf).

The operator that is responsible of loading(unloading) the package wears a full-red(green) uniform.

The human operators have been abstracted colouring the conveyor belts in the implementation.

TURTLEBOT2 CAPABILITIES

Functional Specification

- Maximum translational velocity: 70 cm/s
- Maximum rotational velocity: 180 deg/s (>110 deg/s gyro performance will degrade)
- Payload: 5 kg (hard floor), 4 kg (carpet)

Hardware/Sensors

- Kinect
  - Color camera: 640x480 sensor, 640x480@30fps output
  - IR camera: 1280x1024 sensor, 640x480@30fps output
  - Operation range (depth sensor) = 0.8m - 3.5m
  - FOV = 58°H, 45°V, 70°D
  - Spatial resolution (@ 2m distance) = 3mm
  - Depth resolution (@ 2m distance) = 1cm
- Gyroscope
- Bumpers: left, centre, right
- Cliff sensors: left, centre, right
- Wheel drop sensor: left, right
- Odometry: 52 ticks/enc rev, 2578.33 ticks/wheel rev, 11.7 ticks/mm
- Programmable LED: 2 x two-coloured LED
- State LED: 1 x two coloured LED [Green - high, Orange - low, Green & Blinking - charging]
- Sensor Data Rate: 50Hz

USED HARDWARE

We deployed the system on the following configuration(s):

The V-REP simulation run on the following machine:

- OS: Windows 10 x64
- CPU: i7 7700k @4,8 GHz
- RAM: 16 GB DDR4 @2400MHz
- GPU: Nvidia GTX 970

The DALI infrastructure run on a MacBook Pro 13".

By our analysis, the system has the following memory requirements:

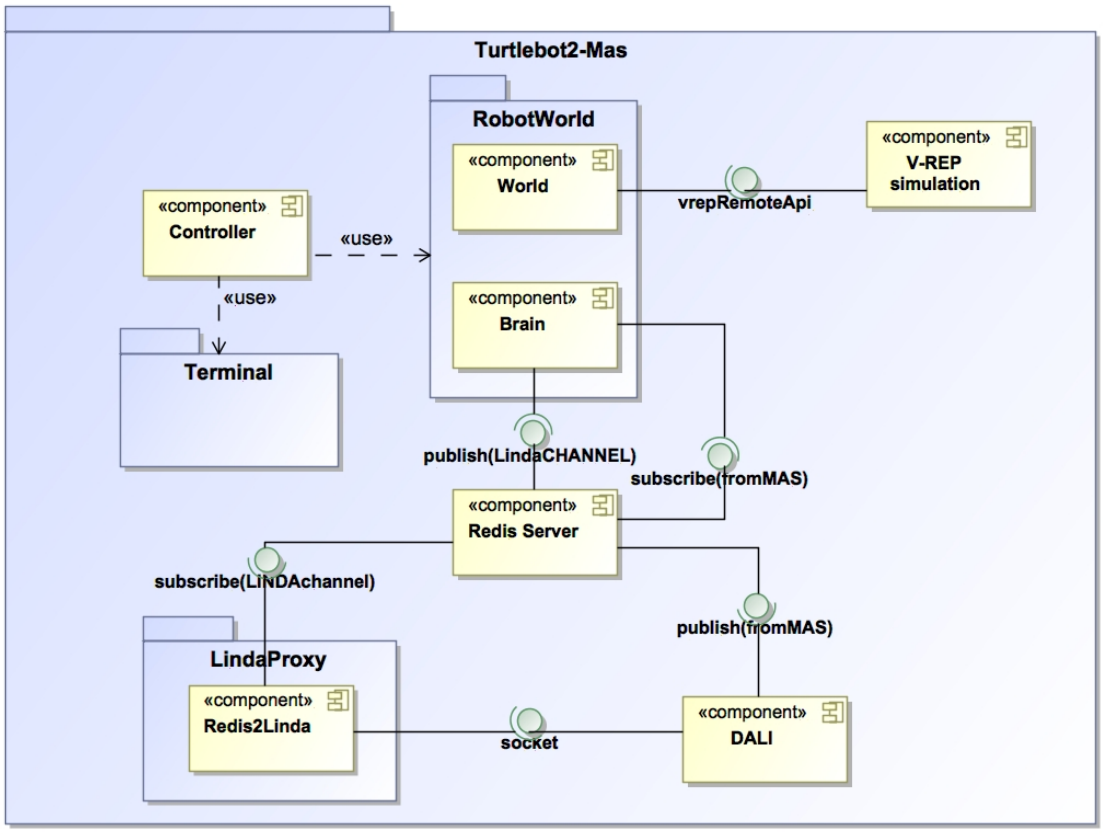
- 278.5 MB for the V-REP simulation
- 186.4 MB for the DALI subsystem.

SYSTEM COMPOSITION

The system is composed by the following components:

- *V-REP* simulator: the host of the simulation.
- Python *Controller*: it is responsible to spawn the processes that will interact with DALI (*Brain* component) and the V-REP simulation (*World* component).
- Python *Brain*: it is the script used to interface with DALI and it represent the reasoning part of the robot.
- Python *World*: it is the script used to interface with V-REP, so it is responsible of elaborate what the robot sense and tell the robot what to do and how, when a particular event occurs.
- *Redis Server: used as a message broker (more explanation in the following sections) to allow communication between the Python Controller and the DALI infrastructure.*
- *Linda Proxy* (Redis2Linda): allows to send messages to Linda by publishing on a redis topic, the proxy translates the message and forwards it to the MAS.
- *DALI* multi-agent system: composed by the agent representations of the physical units, each agent analyse the incoming message and computes an appropriate action.

Component Diagram

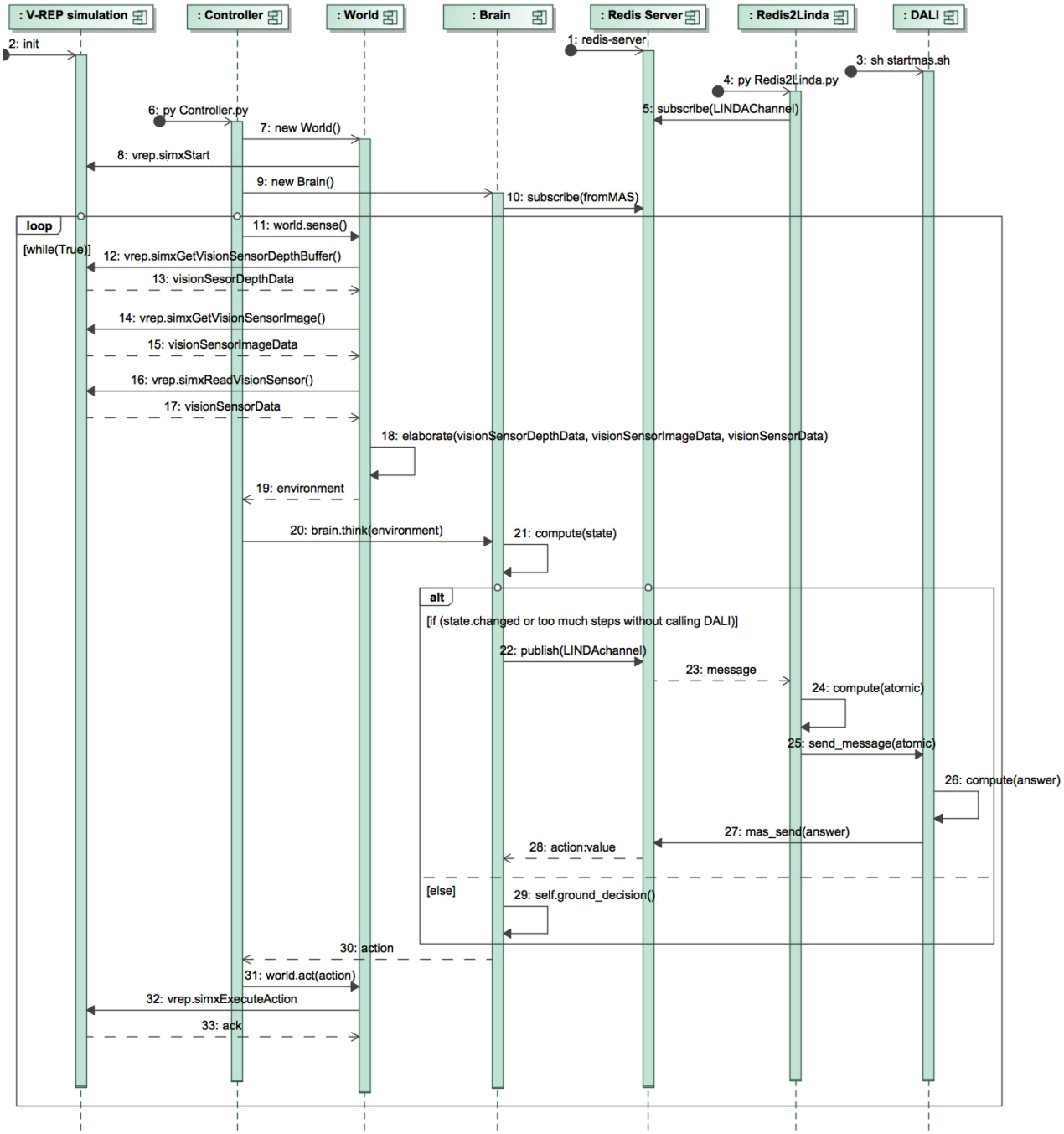


Message passing framework

The typical flow of the information is the following:

1. the python *Controller* fetches, for each agent, its status;
2. if required, the information is transformed in a format that is readable by *DALI*;
3. the message is sent to the *Redis2Linda* proxy that encapsulate it and forwards it to the *DALI* agent;
4. the *DALI* agent receives it (socket), learns the information that is contained in it and computes an answer;
5. the answer (action) is sent back directly to the python controller through *redis*;
6. the *Controller* interprets it and executes it (in the meantime the controller put the physical agents in stand-by while listening for the answer).

Sequence Diagram



The user starts the system by running the *Redis Server*, followed by the *V-REP simulation*, *DALI* and the *Redis2Linda* proxy.

Redis2Linda starts by subscribing to LINDAchannel and waits for incoming messages from the *Controller*. The *Controller* instantiates a *World* object that is used to establish a connection to the *V-REP simulation.* In the same way a *Brain* object is instantiated.

At this point, the application enters in a loop: the *Controller* asks to sense the environment to the *World* object; it interacts with *V-REP* and retrieves the raw data about the environment. When the *World* receives an answer for all the requests it has done, it elaborates what it has received and sends the polished data to the *Controller* that shares this information with the *Brain*.

The *Brain* object computes the current state of the unit; then the *Brain* compares the state that it has just elaborated with the previous one: if the state is changed or a certain number of decisions have been taken without invoking *DALI*, the *Brain* publishes on *LINDAchannel* the state of the unit; then *Redis2Linda* receives the message and "translates" it in a way that *DALI* can understand and then it forwards it to *DALI* itself. At this point *DALI* computes an answer based on what the message contained and publishes the resulting action to *the "fromMAS"* topic; the message is then received by the *Brain*, that in the meantime has stopped the robot while listening for an answer.

If, instead, the state is not changed and so there is no need to invoke the *DALI* subsystem, the *Brain* decides to repeat the previous action.

When the *Brain* makes a decision on which action the robot must do, it communicates it to the *Controller* that pass the action to the *World* object which executes it (by sending commands directly to the *V-REP simulation*).

Then the cycle restarts.

INTERNAL DALI AGENTS REPRESENTATION

Predicates

NAME	DESCRIPTION
vision/2	Contains information about what the agent sees: the first term indicates the color of the object that the agent sees, or 'none' if the agent sees nothing. The second term indicates where the object is w.r.t. the agent dof (right, center, left, near).
depth/1	Used to inform the agent that the depth sensor has detected something that is near or far.
load/1	Tells whether the agent is carrying a package or not.
recentavoidance/1	Used to remember if in the recent past the unit avoided an obstacle.
agentname/1	Used to construct the answer, so the python controller only reads the messages that are actually meant for it.

Rules/Events

NAME	TYPE	STRUCTURE	DESCRIPTION
redis/1	external event	-	Triggered by the arrival of a redis message, the agent cleans the message and invokes the addKnowledge/1 rule.
addKnowledge/1	rule	-	The agent creates an .lp file in which it writes the content of the received message and then it compiles it, learning what it received; to avoid race conditions on the file, each file name is preceded by the current timestamp (through the concatenate/3 rule).
concatenate/3	rule	-	Concatenates a numeric prefix with an atomic suffix; the new atom is stored in "Concatenation".
answer/1	rule	-	Incapsulates the answer operation, the agent builds the answer, send it using the rule mas_send/1 and then forgets what it has learned (to have a clean kb for the next message).
avoid/0	internal event	preconditions	the unit has something near that is not its target
		outcome	answer: turn 40° to the left
unload/0	internal event	preconditions	the unit is near the green operator and it is full
		outcome	answer: unload

NAME	TYPE	STRUCTURE	DESCRIPTION
loadup/0	internal event	preconditions	the unit is near the red operator and it is empty
		outcome	answer: loadup
follow/1	internal event	preconditions	the unit sees its target in a certain position (that is not ‘center’)
		outcome	answer: turn 3° to the direction in which the target is
forward/0	internal event	preconditions	the unit sees its target at the center
		outcome	answer: go forward at a speed of 3
turn/0	internal event	preconditions	the unit is facing the wrong direction
		outcome	answer: turn 20° to the left
wanderAround/0	internal event	preconditions	the unit does not see a target
		outcome	answer: turn 20° to the left
wanderAroundAfterAvoidance/0	internal event	preconditions	the unit does not see a target, but in the step before it avoided an obstacle
		outcome	answer: go forward at a speed of 5