Awesomizing the P2DX
AP2DX

Wadie Assal 6398693   Jasper Timmer 5995140   Maarten de Waard 5894883

Maarten Inja 5872464

June 25, 2011

Contents

1 Preface

As part of the course Software Engineering and Distributed Applications, of the University of Amsterdam, we are asked to program a distributed application for a virtual robot. The team consists of students with different profiles: three are from Artificial Intelligence, one from Computer Science and ICT at the Amsterdam University of Applied Sciences and one from the pre-master Software Engineering. This melange is both a chance to create something good from different viewpoints as well as a point of attention, because none of us has followed the bachelor Computer Science at the University of Amsterdam.

2 Introduction

For the client a robot controller that can autonomously map an area had to be build. This was attempted in less than four weeks. After two and three weeks there were milestones to show the client the progress.

2.1 USARSim

As simulator USARSim is used. USARSim (Unified System for Automation and Robot Simulation) is an application to simulate the real robot arena's of the IEEE, based on the Unreal engine. The robot should be able to navigate through the 'Yellow Arena'. This is a map with a few static obstacles.

2.2 Structure of this document

This document describes the setup of the development environment and the architecture of the program. In the appendices there is more technical information.

3 Goals

With the client there is agreed to build the following features in the robot-controller:

- Loosely coupled modules based on network communication

- Robot should be safe, i.e. stop for obstacles

- Robot should be able to drive autonomously through the environment

- Robot should be able to create a map of the environment

3.1 Milestones

At every agreed milestone there was a deliverable plannened. This deliverable can contain software and or documentation.

The first milestone:

- Drive: The program should be able to direct the robot through the environment but not yet be able to follow lines or walls.

- Avoid collision: The robot should be able to avoid collision with objects and walls. It will stop, turn to a random angle, and drive on. This way it will cover most of the area without colliding.

- Experiment and content of final report: A draft of the final report, containing a description of the experiment and its contents.

What was actually delivered at this milestone was a program that could only spawn a robot, and Javadoc describing the API of the application. This was partly because writing tests took a lot of time and the baseclass was not yet ready. More about the baseclass later.

The second milestone:

- Avoid obstacles: The robot will be able to avoid the obstacles that cross its path, in stead of stopping and turning a random corner.

- Navigate: The robot will be able to navigate through the room.

- Mapper: A class that creates a map of the room out of the sensor data. In the time of milestone two it does not have to be able to create an entire map and be very accurate, but it will be able to make some implementation

- Improved Sensor: The sensor class will be improved to be able to make an accurate map

- Improved Reflexes: The reflex class needs to be able to use some sensor data to be able to avoid objects appropriately.

The deliverable for this milestone also did not meet the agreed requirements. It did not even met the requirements of the first milestone. A lot of time went to fix bugs in code that could only be tested when the whole framework was programmed. Unforseen problems and horrible bugs soaked up a lot of time. Furthermore some team members were sick.

The final delivable should meet all the goals.

4 Architecture

4.1 Introduction

AP2DX is written completely in the Java programming language. This was choosen as it is advertised to be reliable and fast by the company Flowtraders. This is important for AP2DX, because it should be a safe robot controller, that stops in time and does not harm anyone on it's path. The architecture of AP2DX is based on Object Oriented Programming (OOP). To not repeat the same code again and again for every module, a baseclass was constructed. This class could do things as: read a config file, write to a logfile, accept incoming connections, start outgoing connections based on the config file, handle incoming messages and send responses. The baseclass was designed to be flexible for the needs of every module. Also, the program is heavily multithreaded, to open and check connections and to do the business logic and send messages. As for the messages: a base messageclass was build, to facilitate the communication between USARSim and AP2DX, and between modules of AP2DX. As standard for config file and communication, JSON (Javascript Object Notation) was used.

Create class diagram

4.2 USARSim

The simulator exists of a virtual enviroment where different maps and different robots can be loaded. These robots have different sizes and different features. As example of features of a robot, there are different kinds of sensors and different kinds of wheels. For AP2DX, the robot P2DX was used. This robot is a three-wheel, rear swivel wheel robot. The two frontwheels can be controlled independently of eachother, so it is possible to turn almost in place. As sensors are available on the P2DX:

- 8 Sonar distance sensors

- 1 Laser distance sensor

- Odometry

- Internal Navigation System

- Camera

For the purpose of AP2DX only sonar, laser and INS sensors are used.

insert picture here

4.3 Coordinator

The first module a message on a journey from the USARSim to AP2DX meets, is the AP2DX Coordinator. The Coordinator is programmed to spawn the robot and translate and relay traffic between the simulator and the AP2DX System. That is all it really does. It has a connection with the Sensor module to relay the incoming sensor data and the Motor module to receive commands for the simulator.

4.4 Sensor

When a message is translated by the coordinator the journey continues to the AP2DX Sensor module. This module creates a visualisation of sonar and laser sensor-data in a user interface window, so it is possible to see what the robot sees. Then the message is cloned: it is sent to the Mapper module and the Reflex module.

4.5 Mapper

In the mapper module all the sensor information comes together and here is where the information is going to make sense. The sensor information combined with the position of the robot the mapper module will construct a map. This map is then send to the planner module. If there are any irregularities with the position of the robot it will be corrected here and send to the planner.

4.6 Planner

The planner module is the brain of the software. It directs the robot to unexplored places and solves problems with obstacles. It will make its decisions based on the map send from the mapper module and the obstacle information send from the reflex module. The planner is also the place where the position of the robot is saved. The position is periodically sent to the mapper so the postion can be corrected by the mapper if needed.

4.7 Reflex

The robot is controlled by the reflex module through the motor module. The reflex module will take the responsibility for unexpected situations that could not be foreseen by the planner module. For example low obstacles or unexpected falling object in the path of the robot. The reflex module will stop the robot so the arising situation can be handled gracefully by the planner module. The information needed to make a new route is send to the planner module from the reflex module.

4.8 Motor

The motor modules main function is to provide an abstract layer for the reflex module to control the robot. This will disguise the complexities of controlling the robot. For example if you want to control the robot on a low level, you will have to send a message to drive forward, calculate if the desired distance is covered and then stop the wheel taking into account sliding of the robot because of the momentum. Through the motor module you can control the robot by sending a drive forward message with the desired distance that needs to be covered. The module itself will calculate the distance and stop the robot on the correct place.

4.9 Communication

There are two seperate protocols in use in AP2DX. The first protocol is the Gamebot protocol used by Unreal Tournament to communicate with the simulation. The coordinator module is the only module that uses these messages. First the coordinator module sends a "spawn" message to create the robot in the simulator. Then it continues to read messages (often containing sensor data) send by the simulator and to send messages (containing "drive" commands) to the simulator.

The second protocol is the AP2DX message protocol. These are JSON strings that *always* contains a source module ID, a destination module ID and lastly a type, indicating what kind of message this it is. Other data in these messages might vary. For example; a sonar sensor message has the sonar data in it. A drive message only stores the information on how to drive in the string. This protocol is used to communicate between all the modules in AP2DX. The coordinator module is responsible for translating messages using a kind of protocol to a message using the other kind of protocol.

Using two protocols meant that a lot of things had to be done twice, but also that a module could deal with both protocols. Thefore the base class deals with instances of our abstract "Message" class and has an interface "IMessageParser". Final modules *knows* what kind of protocol a connection uses and can set up the right message parsers ("AP2DXMessageParser" or "USARSimMessageParser") and thuss, deal correctly with the incoming messages.

Different kind of messages need to be dealt with as well. The source and destination module ID take care of getting the message to the right module. The type is used to identify a message. Once a message has been identified as, for example a range scanner sensor message (type: AP2DX_SENSOR_RANGESCANNER), it becomes a "specialized message". A specialized message parses the variables in the string and sets them in variables in the message. Now, simple setters and getters can be used to modify and read all the different

The hierarchy of our messages can be seen in figure X. create and insert picture here (picture; Message to AP2DXMessage and USARSimMessage. AP2DXMessage to SpecializedMessage. SpecializedMessage to all the sensor messages and motor messages and drive messages. USARSimMessage to Jaspers messages)

5 Building and Testing

In this section, we will explain the way we tested our program, and automated our buildprocess. Developing and debugging involve a lot of the same actions. To make the developmentprocess less repetitive, tedious and time consuming, some manhours were spent on setting up a good working enviroment. A git repository was setup at github.com. To develop better code, there was the idea to start with building unittests before actual code. More on unittests later. To automate the compiling, testing, building of Javadoc and creating Java executables (.jar files), an Ant buildscript was created. Next a continuous integration server was set up.

5.1 Git

Git is a distributed version control system, not like Subversion, but like Mercurial. Git allows the developers to work on the same file together and merge the changes. Every commit is only made local and can be reversed. A project was started at http://www.github.com that free Git versioncontrol for opensource projects. It also offers development statistics, wiki and issue management, but those features were not used.

5.2 Ant

Java was choosen for AP2DX, and Java has to be compiled before it can be run. Also, the build directory has to be cleaned, unittests have to be compiled and run, Javadoc has to be published and executables containing the compiled classes had to be generated. This whole process can be automated with an Makefile-like tool, called Ant. An Ant Makefile is called build.xml. When in a terminal in the directory containing build.xml, all the developer has to do it type 'ant' and the process is run. Because developers don't want to wait on javadoc and testing to be completed every time they need to compile and debug, a second build target was created that only did the compiling and generating of jar files, and this could be run with 'ant debug'. Jenkins did run the full buildscript.

5.3 Jenkins

To put everything together and test if it still works, a continuous integration server can be used. Jenkins was choosen for good reviews, and personal experience. What it does, is get every whole hour a clone of the Git repository, execute the Ant script, and publish the testresults. Jenkins is a java program and was run from a student homedirectory on deze.science.uva.nl, during development hours.

5.4 Testing

The original idea for the development of AP2DX was Testdriven development. This involves writing unittests before the actual code using jUnit testing framework. The idea of a unittest is to test the smallest possible unit of code, this is a class method or constructor. A good unittest is independent of other classes. If one class fails, the test for the class that is depenent on the failing class should not fail, because this class may be written correctly. It is possible to make unittests independent on other classes, with class mockups (using the jMockit framework), that are something like dummy objects, that do exactly what you want them to, but this is subject for an other paper.

It is not only important to write tests that test all methods and all possible input and output values, but another measurement instrument for the quality of the tests is line coverage. it is almost never possible to get 100% line coverage, but about 80% line coverage should be well tested code, if all possible input and output values are covered.

Writing of tests was harder than planned and took to much time, so it was decided to stop that and start working on the robot controller. Testing requires a specific mindset and understanding of the process, and because of the socket communications testing was not that simple. it is possible to mock a socket but mocking is difficult also, and it is possible to start a listing socket and let the class under test connect to it and then send it a message, but this requires extra threads which is not very handy for unittests.

5.4.1 Testreport

TODO: Get test statistics and screenshots

6 Experiment

Here we will explain the experiment and how we did the tests. (can be more than one experiment).

7 Results

Here we will discuss the results from our experiment and explain what it means

8 Discussion

8.1 Architecture

The architecture of AP2DX was well designed. What we thought off in the first week was nearly exactly programmed when everything started to work (give or take a few small details, which were easy to add). The framework is robust and more functionallity can easily be added. For example; sensor messages we do not use can be added in mere moments by creating a specizialed message.

But, the question is: Is this a good thing? Obviously it is nice to see the framework working and to know new functions can be added later. But it took us a loot of time to set it up. We did not need to parse all kind of special messages if only use two or three!

Also the base class disallowed us to test anything with the simulator until it was completely finished. Each module could easily be programmed in another programming language, but this pinned us down to use Java, and the base class.

8.2 Message Protocols

Parsing messages We originally thought the messages for USAR and AP2DX were quite similar, but they diverged more and more the further we came to working code. In the end it simply seemed as if we did the same thing twice, with no real advantage. The USAR messages had to be parsed either way. They also had to be compiled either way; there was no alternative method to communicate with the server. In hindsight, we might've better programmed that well, with a nice interface to set and get variables rather than to program it twice.