UNIVERSITEIT VAN AMSTERDAM

Awesomizing the P2DX

AP2DX

Wadie Assal	Jasper Timmer	Maarten de Waard	Maarten Inja
6398693	5995140	5894883	5872464

July 1, 2011

Contents

1 Preface

As a 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 Goals

With the client 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

2.2.1 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 last milestone:

Before the demonstration we will be able to do the following things:

- Planner: We will have a planner class that can specify directions based on the current map position and what part of the map we have not yet discovered.

- Improved Mapper: The mapper will now be able to make an accurate map and find our location on it, while taking the errors in sensor data into account.

- Tests: We will test everything thoroughly.

- Report: We will work on a report, describing our progress, problems and (test)results.

- Documentation: We will work on a proper documentation of our code.

This deliverable was partly reached. We did not make a Planner that uses the map data, but it still makes the robot drive in safe directions. We also have made a mapper. This mapper works real time, but misses some information. This could have been fixed, if we had more time. In the last week, we stopped focussing on the tests, because with only four of us left, the tests stopped being a priority. The documentation was done while coding, so working on a proper documentation was no problem.

2.3   Structure of this document

This document is written for the client as reference about the delivered software. It contains information about the architecture of the program (all the components) and design desicions. Also the development and test process is described. The last part is about what is actually deliverd and what can be improved. In the appendices there is more indepth technical information.

<div align="center">
(a) The original P2DX   (b) The P2DX in USARSim
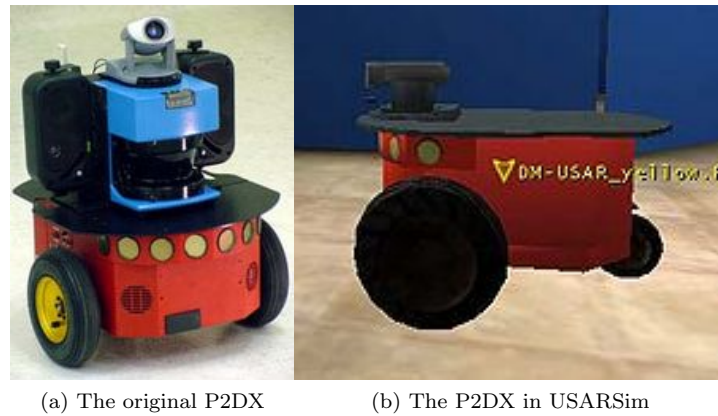
Figure 1: The P2DX we worked with
</div>

3 Architecture

3.1 Introduction

AP2DX is written completely in the Java programming language. This was chosen as it is advertised to be reliable and fast by the company Flowtraders, and it is platform independent. 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 message class 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

3.2 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 (See figure 1) 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

- Laser distance sensor

- Odometry sensor

- Inertial Navigation Sensor

- Camera

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

3.3 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.

3.4 Sensor

The sensor module has two tasks. The first task is forwarding different kind of sensor messages to the mapper module and the reflex module. The second task is to display the sonar and range scanner (laser) sensor data graphically. This allows to see if the data is correctly parsed by the coordinator module and received by this module with one quick glance. Furthermore it can be used for other debug purposes, such as checking correct reflex behavior. See figure **??** for an example of displayed sensor data.

insert picture of sonar and laser data here

3.5 Mapper

The task of the mapper module is to create a map of the environment and to find the position of the robot in this map. This will allow the planner module to plan paths and other behaviour, that is: if it is done correctly!

Luckily, simultaneously localizing and mapping (SLAM) is a commonly encountered problem and many research has been conducted to optimize algorithms. The mapper module uses the *DP-SLAM* [1] implementation in the C programming language. Starting the DP-SLAM program (which was slightly modified for compatibility with the Java module) is done with a system call, after which two threads are started. One to write the correct sensor data to the input stream of the program, and one to read the output stream of the program. The output stream contains information about for example the location of the robot, whether or not something went wrong or when a new map has been dumped to file.

Unfortunately, the mapper module is not multi platform as the DP-SLAM program uses Linux libraries. If this module is run in a non Windows system no map can be created, but sensor important data will still be forwarded to the planner module.

3.6 Planner

The planner controls the robot movement. It uses sonar, odometry and INS data to determine the next action to take. The Planner has three behaviors: drive and turn away from obstacle, check out gaps in the wall and get free from a stuck position.

3.6.1 Behavior: Drive

The robot starts driving when the first sonar data is received. The outer two sonar sensors are ignored, but the inner six are used to determine if the robot is approaching an obstacle and the distance is under a certain threshold constant. Then the Reflex module kicks in and sends a stopmessage to the robot and the Planner. The Planner then checks if the sonars on the left or on the right have the most far view and turns in that direction, until the robot has no obstacle in front of it anymore. This is calculated as follows: If the first of the eight sensors has number 1, and the next number 2, etc. the sensors 3 and 6 are in an angle of 60 degrees. This means it is an equilateral triangle if sensors 3 and 6 can see the same distance, and that means, if the sonars 3 and 6 can see at least as far as the robot is wide, it will be possible to drive forward. A small margin is added to the width to be sure the robot does not get stuck.

3.6.2 Behavior: Check hole

When the sonars 1 or 8, that are perpendicular on the robot's path see a difference between two sonardata sets, that is larger then a certain threshold, say 1 meter, the robot gets curious and wants to to know if there is a hole beside it that it can drive through. It turns 90 degrees in the direction of the hole, scans if the hole is wide enough and drives forward if it is. Is it too small, then it turns back and continues it's original path.

3.6.3 Behavior: Get free

Every time INS data is processed, the planner checks how much the robot has moved since the last INS data. If this is under a certain threshold, the robot increments a counter. If the robot did not move enough after a certain amount of INS data, it goes into 'stuck' mode: stop doing things, drive backwards for a small distance, turn 90 degrees left or right at random and start scanning for a new path.

---

[1] http://www.cs.duke.edu/~parr/dpslam/

$$Message$$

$$AP2DXMessage \qquad\qquad UsarSimMessage$$

$$SpecializedMessage \qquad\qquad (individual messages)$$

$$(individual messages)$$
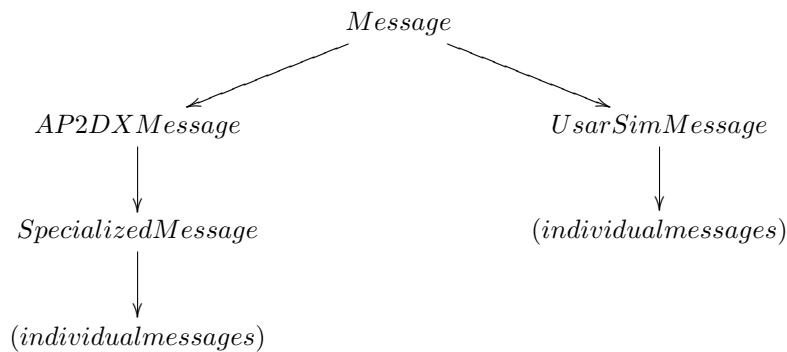
Figure 2: The hierarchy of our message types

3.7 Reflex

The Reflex does only two things. Relay motor messages from the planner to the motor, and stop those messages if there is something in front of the robot, so it has to stop. If it has to stop the robot, it sends a stop message to both the motor and the planner and discards all the movement messages it receives from the planner until the planner sends an acknowledge message.

3.8 Motor

The motor module translates motor messages into motor messages. If it gets forward, 10, it translates to leftwheel 10, rightwheel 10. It does not yet create usarsim message, but it sends this message to the Coordinator and there it is translated and send to the simulator.

3.9 Communication

3.9.1 Protocols

There are two separate protocols in use in AP2DX. The first protocol is the Gamebot protocol[2] 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) sent 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* contain a source module ID, a destination module ID and lastly a type, indicating what kind of message 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.

3.9.2 The Message class

Using two protocols meant that a lot of things had to be done twice, but also that a module had to be able to deal with both protocols. Therefore the base class deals with instances of our abstract `Message` class and has an interface `IMessageReader`. Final modules *know* what kind of protocol a connection uses and can set up the right message reader (`AP2DXMessageReader` or `USARSimMessageReader`) and thus, deal correctly with the incoming messages.

3.9.3 Dealing with 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

---

[2]`http://udn.epicgames.com/Two/UnrealScriptReference.html#Variables`

them in variables in the message. Now, simple setters and getters can be used to modify and read all the different values.

The hierarchy of our messages can be seen in figure 2.

3.9.4   Synchronization between Reflex and Planner

All modules internally use a queue to store all the received messages. Those messages are processed one by one. As soon as the reflex sends a stop message to the motor based on the distance threshold, it starts ignoring all other motor messages it gets from the planner. The planner has to work through it's queue until it reaches the stop message. At this moment, it sends a reset message to the reflex, signaling: everything from now on is meant to get us away from the wall. The Reflex sets a 'getting away from wall' flag and starts relaying again, but is does not watch the distance threshold. When the Planner finished the turn away procedure, it sends a clear message to the Reflex, and from that moment, the robot is again cruising. The Reflex will reset internal state and start again with watching the threshold.

4   Building and Testing

In this section, we will explain the way we tested our program, and automated our build process. Developing and debugging involve a lot of the same actions. To make the development process less repetitive, tedious and time consuming, some manhours were spent on setting up a good working environment. 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.

4.1   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.

4.2   Ant

Java was chosen 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.

4.3   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 home directory on deze.science.uva.nl, during development hours.

4.4   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 dependent 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.

4.4.1   Testreport

TODO: Get test statistics and screenshots

5   Final result

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

6 Discussion

6.1 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.

6.2 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 have better programmed that well, with a nice interface to set and get variables rather than to program it twice.

6.3 Mapper

We downloaded a complicated, sophisticated mapper from the internet. This is good, but the mapper does assume that you always give it correct information. This way, the mapper does not always work.

Also our sophisticated mapper has a slight delay, therefore it misses some information we would put in it.

In hindsight we might have gotten a better mapper if we had downloaded a simpler implementation, and had used that with our program. That would have been better at mapping while driving.

6.4 Autonomous system

As AI-people, we have built an entirely autonomous system. This means that our robot can go where it wants, "when it wants". This is nice, but it does not come in handy, when you want it to do what you want. Sometimes it would have been nice to be able to provide some user input.

6.5 Slow

Our program turned out to be a bit slow, over network. This could be because we print a lot of data, and because every communication is over network. Still, while running everything on one computer, the robot did not crash. This means that the program is fast enough.

bladibla

PlannerProgram (top-left, attributes)

- private double ANGLEUNCERTAIN = Math.toRadians(5)
- private double TURNTHRESHOLD = 1.0
- private double NEEDEDDEPTH = 0.5
- private double PASSHOLEDISTANCE = 0.3
- private double PASSHOLECORNER = 0.3
- private double IGNOREDISTANCE = 0.8
- private double MOVEMENTTHRESHOLD = 0.2
- private int NOMOVECOUNT = 100
- private double BACKWARDDISTANCE = 0.3
- private double travelDistance
- private double distanceGoal
- private boolean firstMessage = false
- private double startAngle
- private double currentAngle
- private double destinationAngle
- private boolean startedTurning = false
- private boolean sonarPermission = false
- private double sonarData[0..*]
- private double lastSonarData[0..*]
- private boolean doHoleScan = false
- private int lastDirection
- private boolean startedTurningBack
- private boolean waitPassingHole
- private boolean passHole
- private boolean stopBot
- private boolean cruising
- private boolean reverse
- private int noMovementCount = 0
- private boolean stuck

Operations
- public void main(String args[0..*])
- public PlannerProgram( )
- public void setTravelDistance(double travelDistance)
- public double getTravelDistance( )
- public void setDistanceGoal(double distanceGoal)
- public double getDistanceGoal( )

Operations Redefined From AP2DXBase
- public AP2DXMessage[0..*] componentLogic(Message message)

InsLocationData (locData)

Attributes
- private double TIMEDELTA = 0.2
- private double lastLocation[0..*] = new double[3]
- private double lastOrientation[0..*] = new double[3]
- private double currentLocation[0..*] = new double[3]
- private double currentOrientation[0..*] = new double[3]

Operations
- public InsLocationData(double loc[0..*], double ori[0..*])
- public void setLocationData(double loc[0..*], double ori[0..*])
- public double travelDistance( )
- public double speed( )
- private void setLastLocation(double lastLocation[0..*])
- private double[0..*] getLastLocation( )
- private void setLastOrientation(double lastOrientation[0..*])
- private double[0..*] getLastOrientation( )
- private void setCurrentLocation(double currentLocation[0..*])
- public double[0..*] getCurrentLocation( )
- private void setCurrentOrientation(double currentOrientation[0..*])
- public double[0..*] getCurrentOrientation( )

SensorProgram

- public void main(String args[0..*])
- private double lastLocation[0..*] = new double[3]
- public SensorProgram( )
- private void doOdometryLogic(OdometrySensorMessage odometrySensorMessage, AP2DXMessage messageList[0..*])
- private void doInsLogic(InsSensorMessage insSensorMessage, AP2DXMessage messageList[0..*])
- private void doSonarLogic(SonarSensorMessage sonarSensorMessage, AP2DXMessage messageList[0..*])
- private void doRangeLogic(RangeScannerSensorMessage rangeScannerSensorMessage, AP2DXMessage messageList[0..*])

Operations Redefined From AP2DXBase
- protected void doOverride( )
- public AP2DXMessage[0..*] componentLogic(Message message)

(drawer, top-right)

- public void sonarRanges[0..*] = new double[0]
- public double rangeScannerRanges[0..*] = new double[0]
- private int sonarThetas[0..*] = {180, 140, 120, 100, 80, 60, 40, 0}
- private double MAX_VALUE = 5.0
- private double rangeScannerFov = 3.1415
- private double rangeScannerResolution = 0.0237992424

Operations
- public void paintSonarLines(double sonarData[0..*])
- public void paintSonarLines(Graphics g)
- public void paintRangeScannerLines(double rangeScannerData[0..*])
- public void paintRangeScannerLines(Graphics g)
- public void paint(Graphics g)
- public void main(String args[0..*])
- public int highestValuedIndexFromArray(double array[0..*])
- public double highestValueFromArray(double array[0..*])
- public double getRangeScannerFov( )
- public void setRangeScannerFov(double value)
- public double getRangeScannerResolution( )
- public void setRangeScannerResolution(double value)

MotorProgram

Attributes
- private double deltaDistanceSinceLastCommand = 0
- private double distanceAim = 0
- private double lastOdometryX = 0
- private double lastOdometryY = 0

Operations
- public void main(String args[0..*])
- public MotorProgram( )
- private void doOdometryLogic(OdometrySensorMessage odometrySensorMessage, AP2DXMessage messageList[0..*])
- private void doMotorActionLogic(ActionMotorMessage actionMotorMessage, AP2DXMessage messageList[0..*])

Operations Redefined From AP2DXBase
- public AP2DXMessage[0..*] componentLogic(Message message)

CoordinatorProgram

Attributes
- package PrintWriter out

Operations
- public void main(String args[0..*])
- public CoordinatorProgram( )

Operations Redefined From AP2DXBase
- protected void doOverride( )
- public AP2DXMessage[0..*] componentLogic(Message msg)

UsarSimMessageReader

Attributes

Operations
- public UsarSimMessageReader(InputStream in)

Operations Redefined From IMessageReader
- public Message readMessage( )

ReflexProgram

Attributes
- public double WALLTHRESHOLD = 0.3
- private double currentDistances[0..*]
- private double previousDistances[0..*]
- private double approaching[0..*]
- private boolean isBotBlocked = false
- private boolean clear
- private boolean firstMessage

Operations
- public void main(String args[0..*])
- public ReflexProgram( )
- public void setCurrentDistances(double currentDistances[0..*])
- public double[0..*] getCurrentDistances( )
- public void setPreviousDistances(double previousDistances[0..*])
- public double[0..*] getPreviousDistances( )
- public void setApproaching(double approaching[0..*])
- public double[0..*] getApproaching( )
- public void setBotBlocked(boolean isBotBlocked)
- public boolean isBotBlocked( )

Operations Redefined From AP2DXBase
- public AP2DXMessage[0..*] componentLogic(Message message)
- public void doOverride( )

AP2DXBase

Attributes
- public Logger logger
- protected Map config
- private long lastDelay
- private AtomicInteger threadCounter = new AtomicInteger()

Operations
- public AP2DXBase(Module myModule)
- protected void doOverride( )
- private void initLogger( )
- protected Map readConfig( )
- protected String getContents(File aFile)
- public boolean UpdateConnection(ConnectionHandler connection)
- public ConnectionHandler getConnection(Module module)
- private boolean addConnection(String ipaddress, int port, Module destination, boolean bidirection)
- public Module[0..*] getPassiveConnections( )
- public AP2DXMessage[0..*] componentLogicCheck(Message msg)
- public AP2DXMessage[0..*] componentLogic(Message msg)
- public AP2DXMessage[0..*] getReceiveQueue( )
- public void setLastDelay(long millisec)
- public long getLastDelay( )

UsarMessageParser

Attributes

Operations
- public UsarMessageParser(AP2DXBase base, Module IAM, Module send, Socket socket)
- public void run( )

SpecializedMessage

Attributes

Operations
- public SpecializedMessage(MessageType type, Module sourceId, Module destinationId)
- public SpecializedMessage(AP2DXMessage message)
- public SpecializedMessage(AP2DXMessage message, Module sourceId, Module destinationId)
- public SpecializedMessage forward(Module source, Module destination)
- public void specializedParseMessage( )

Operations Redefined From AP2DXMessage
- public void parseMessage( )

Operations Redefined From Message
- public void parseMessage( )

AP2DXMessage

Attributes
- private long delay = System.nanoTime()

Operations
- public AP2DXMessage(MessageType type, Module source, Module destination)
- public AP2DXMessage(String in, Module source, Module destination)
- public AP2DXMessage(String in, Module source)
- public AP2DXMessage(String messageString)
- public Object clone( )
- public Map<String, Object> getValues( )
- public int compareTo(Delayed o)
- public long getDelay(TimeUnit unit)
- public long setDelay(long millisec)
- public String toString( )

Operations Redefined From Message
- public void parseMessage( )
- public String compileMessage( )

Logic

Attributes

Operations
- public Logic(AP2DXBase base)
- public void run( )

ConnectionHandler

Attributes
- private Socket socket
- private boolean bidirectional
- private boolean passive
- private PrintWriter out
- private boolean alive = true

Operations
- public ConnectionHandler(boolean usar, boolean bidirection, AP2DXBase base, Socket socket, Module origin, Module destination)
- public ConnectionHandler(AP2DXBase base, Socket socket, Module origin)
- public void run( )
- public void sendMessage(Message message)
- public boolean isConnAlive( )
- private void setConnAlive(boolean value)
- public int getPort( )
- public String getAddress( )
- public Module getModule( )
- public void setBidirectional(boolean bidirectional)
- public boolean isBidirectional( )
- public void setPassive(boolean passive)
- public boolean isPassive( )

UsarSimMessage

Attributes

Operations
- public UsarSimMessage(String in)
- public UsarSimMessage(MessageType type)
- public MessageType getMsgType( )
- protected void parseMessage( )
- protected String compileMessage( )
- public String toString( )
- public String getMessageString( )

Message

Attributes
- protected String messageString

Operations
- public Message(MessageType type, Module source, Module destination)
- public Message(String in, Module source)
- public Message(MessageType type)
- public Message(String messageString)
- public Message(String in, Module source, Module destination, MessageType type)
- public Message(String in, Module source, Module destination)
- protected void parseMessage( )
- protected String compileMessage( )
- public Module getSourceModuleId( )
- public Module getDestinationModuleId( )
- public String getMessageString( )
- public Module setSourceModuleId(Module sourceModuleId)
- public Module setDestinationModuleId(Module destinationModuleId)
- public String setMessageString(String messageString)
- public MessageType setType(MessageType type)

AP2DXMessageReader

Attributes

Operations
- public AP2DXMessageReader(InputStream in, Module source)
- public AP2DXMessageReader(InputStream in, Module source, Module destination)

Operations Redefined From IMessageReader
- public Message readMessage( )

MapperProgram

Attributes
- private String slamDir

Operations
- public void main(String args[0..*])
- public MapperProgram( )
- private void doRangeLogic(RangeScannerSensorMessage message, AP2DXMessage messageList[0..*])
- private void doInsLogic(InsSensorMessage message, AP2DXMessage messageList[0..*])
- private void doSonarLogic(SonarSensorMessage message, AP2DXMessage messageList[0..*])
- private void doOdometryLogic(OdometrySensorMessage message, AP2DXMessage messageList[0..*])

Operations Redefined From AP2DXBase
- public void doOverride( )
- public AP2DXMessage[0..*] componentLogic(Message message)

<<interface>> IMessageReader

Attributes

Operations
- public Message readMessage( )

Module

Literals
- COORDINATOR
- MOTOR
- REFLEX
- SENSOR
- PLANNER
- MAPPER
- USARSIM
- TEST
- ADMIN
- UNDEFINED

Attributes
- public boolean canAcceptJsonMessages

Operations
- private Module( )
- private Module(boolean canAcceptJsonMessages)

Connection

Attributes
- private InetAddress address
- private Socket socket
- private PrintWriter out
- private BufferedReader in

Operations
- public Connection(String addressString, int port, Module module)
- public boolean sendMessage(String message)
- public String readMessage( )
- public boolean close( )
- public Module GetModule( )

PictureViewer

Attributes
- private String image
- private BufferedImage bgImage
- private String pathToTheImage
- private BufferedImage backgroundImage
- public int imageCounter = 0
- private JFrame frame

Operations
- public void main(String args[0..*])
- public PictureViewer( )
- public void run( )
- public void updateImage(BufferedImage backgroundImage)
- public void paintComponent(Graphics g)

Slammer

Attributes
- private Process p
- private BufferedReader sensorReader
- private String pathToSlam = "/home/mpdw/uva/ap2dx/c/dpslam/slam"
- protected BufferedWriter outToFile
- private double x
- private double y
- private double theta
- private double precision
- private String toSendXYTheta
- private String toSendLaser

Operations
- public Slammer( )
- public void run( )
- private void processOutputString(String line)
- public void addXYTheta(double x, double y, double theta)
- public void addLaser(double rangeArray[0..*])
- public double getX( )
- public double getY( )
- public double getTheta( )
- public double getPrecision( )

(Relationship labels: locData, drawer, in, base, baseLogic, receiveQueue 0.., connections 0..*, sendConnection, parser, source, destination, sourceModuleId, destinationModuleId, passiveConnections, 0..*, moduleID, IAM, send, module, pictureViewer, slammer)*