University "Politehnica" of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

MASTER PROJECT THESIS

Modern Methods for Communication, Mobility and Portability for the tATAmI platform

Metode moderne pentru comunicatie, mobilitate si portabilitate pentru platforma tATAmI

Scientific Adviser:
S.l.dr.ing Andrei Olaru

Author:
Ing. Ionuţ Cosmin Mihai

Bucharest, 2016

Special thanks to my advisor,Andrei Olaru for all
the guidance, the responsiveness and understanding
over the last two years.

Also, many thanks to all the professors who
inspired me all these years and showed me
what an engineer always should be eager to see:
a new perspective of "how the world is built",
in the shape of machine learning.

Abstract

The instruments for running a house, like the washing machine, the TV or the refrigerator, that once were not interactive, are heading towards connecting with the human users in order to better respond to their needs; the phones are already there. This technology advancement opens a new opportunity for an old concept: Multi Agent System; tATAmI is a MAS framework which simulates an environment and defines components for its agents such that their behaviour can change based on the parts they are composed of.

My contribution for the Master Diploma Project encompasses the implementation of a new communication feature (websocket based), the agents mobility, support for the Android and Raspbian operating systems, new agent components for sensors and other hardware parts and also building a Raspberry Pi example platform for validation and testing.

Contents

Chapter 1

Introduction

Ambient Intelligence is the ability of an artificial environment(i.e. a room, a traffic intersection) to interact with humans in such way that different aspects of their life like safety and work performance are enriched. From smart houses to public transportation and factories, Ambient Intelligence technologies are already present in every day life, the most known are in the area of IoT(Internet of Things). A Multi Agent System(MAS) can be understood as a system distributed over the network where every agent is an autonomous entity.

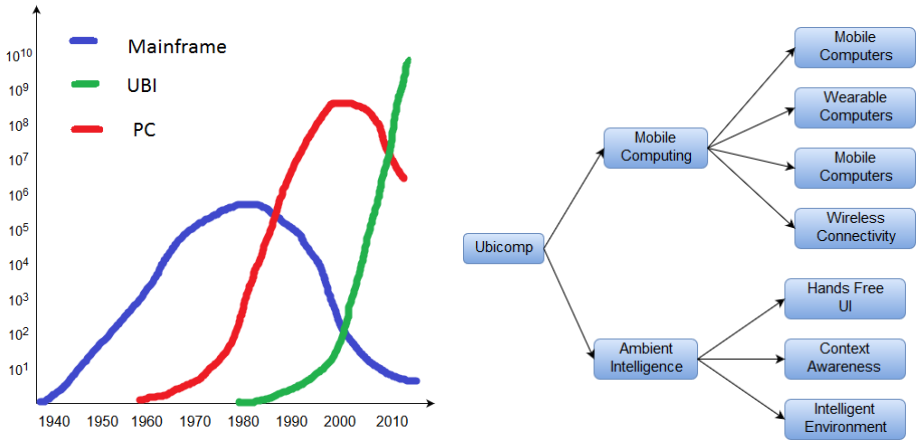
tATAmI stands for "towards Agent Technology for Ambient Intelligence" and it is a component-based MAS which is part of the "tATAmI S-CLAIM(Smart Computational Language for Autonomous Intelligent and Mobile Agents)" platform.

My contribution to the tATAmI framework during the master thesis encompasses several features as follows:

- Increased portability, beside desktop operating systems (which support Java and can make use of a visual interface) it now can be used on Android and Raspbian.
- Simplified and extended communication ability using Websockets.
- Agent mobility; now they can move throughout an entire network instead of only communicating with the other agents.
- Several agent components designed for sensors management and actions.

Nowadays technology evolved enough such that higher levels of generalization and abstraction can be implemented on devices that ten years ago were inaccessible because of processing-power and memory-size restrictions. The passing of time also revealed what concepts are more important and often used.

The increased portability implies that new types of nodes can be added to the MAS. The network we are talking about is not necessarily a Local Area Network, but the ensemble consisting of different operating systems linked by an Internet connection and a server with an accessible IP address and tATAmI installed. The new types of nodes supported can use Android or Raspbian and also feature platform-specific human interfaces (CLI/GUI). In order for the agents to be



(a) Computing paradigms trend (b) Classification of the computing paradigms

Figure 1.1: Ubiquitous computing

able to communicate over the Internet, a new communication system based on Websockets has been integrated. Then, for the agents to be more autonomous and system-independent, mobility has also been implemented, such that every agent can move as it desires. Moreover, in order to use several sensors and hardware components, new agent components have been added to tATAmI.

When reading about AmI I found an interesting explanation in Alcañiz et al [1]. At an abstract level, an intelligent environment is very similar to a PC in the sense that instead of GUI, keyboard, mouse, printer etc. which are ways of interacting with a PC, we can consider sound recognition, voice recognition, and movement patterns to interact with the environment. This is called "Ubiquitous" computing, meaning that the physical environment can be interacted with the same as with a computer, however this feature ought to be balanced by another one such that the user is not overwhelmed with information: Transparency. Transparency in this case is the characteristic of an environment to hide its unintuitive components in such a way that the user is not aware of complex details, for example the integration of a sensor in a table. Current trends indicate a growth of Ubiquitous computing, as can be seen in the Figure 1.1, along with a classification of this paradigm.

Possible user interfaces could be classified as: "Multimodal Input", "Multimodal Output" and "Interaction Management". Where the Multimodal Input is the set of input methods from voice recognition to text recognition, Multimodal Output is the set of the output methods from voice generation to augmented reality and Interaction Management is the whole system which coordinates the inputs and outputs such that they lead to the best possible outcome.

1.1 Usecases

We have developed some use cases in order to show how the improvements that we have integrated into tATAmI could be used in real-life scenarios.

1.1.1 Machine learning model training

Bob created a small robot and a machine learning algorithm, however, in order to get the best results, he has to train the robot intensively for a long period of time. Bob certainly does not have so much time to spend, he almost gives up when he finds out about tATAmI. After he learns to use it, he decides that it would be a good idea to use the robot for data acquisition and testing, the PC for training with the data from the robot (because the PC has more processing power and a large amount of memory), and his phone for a basic remote control. Now he can train his robot from anywhere with an Internet connection, he can stop the process or adjust it, he can use several available resources adaptively and he can oversee the training.

1.1.2 Test automation

Ben works on embedded system X, he needs several scripts to test some use cases, one of them implies shutting down the Ethernet/Wireless connection and then reconnect, another implies specific access rights such that only signed applications can call some core methods, but he realizes that he cannot sign an application for every script and he decides to use a signed tATAmI application. The tATAmI application will be integrated in the test framework, as it has the ability of being autonomous which is a perfect match. The tATAmI framework offers an infrastructure for an agent that can behave autonomously on the target device.

1.1.3 Cluster assistant

Dan has a lot of computers in his cloud company and sometimes computer components fail. By using a tATAmI deployment he knows which one failed and he can rapidly solve the problem which otherwise would have require a stand alone implementation from scratch.

1.1.4 Other possible use cases

- Bayesian Network Simulator using each agent as a node with given probabilities to communicate with each neighbour.
- Security Building Sensor Monitor(i.e. water-meter in a building)
- Urban Traffic Management, each semaphore being an agent

Document structure

The document is organized in six chapters, in the first one, general concepts and use cases are presented, next, you can find a discussion about what already exists, after this the implementation is presented over three chapters: the first one 2 is dedicated to the parts that are common to all the systems, then the next one 5 describes the Android implementation and the fourth 6, the Raspbian implementation. The last chapter 7 is reserved for testing, results and conclusion.

Chapter 2

State of the art

2.1 Overview

At the base of the principles taken into account when implementing tATAmI features stand several other frameworks, I will talk about the principles and about the most important frameworks further. A part of the state of the art was written before beginning to design and implement and the other part after, in order to compare what exists with what is implemented, it is less confusing to know this from the beginning.

- AOPL(Agent Oriented Programming Language) A common practice for a framework is to have an Agent Oriented Programming Language at its top, it has the advantage of ignoring the OOP principles underneath, but the disadvantage of poor performance due to interpretation step, new syntax have to be learned and not in the last place, most of the agent oriented languages are limited and belong to symbolic artificial intelligence, which becomes more and more deprecated. At this level however, the language layer is ignored and every message is treated discretionary by every's agent messaging component. In the future a connection with the S-CLAIM application will be done and tATAmI will have its context-aware language as FIPA standard specifies.
- BDI(Belief, Desire, Intention) BDI is a well known principle in the world of MAS implemented by many frameworks, however, many researchers think that it is a wrong approach because, belief, desires and intentions are fuzzy high level models clustered in the human brain(i.e. Ray Kurzweil in his book "How to create a mind"). The BDI architecture comes from philosophy, more precisely from Bratman 1987, then the theory was formalized by Cohen and Levesque in 1990, by Konolige and Pollack in 1993, and by Singh and Asher in 1990. The above formalizations were made at an abstract level and could not be directly implemented.[17]
- Component based MAS which will be described further because it is the approach I am working with.

2.2 Component based agents

"Component based agents" is a dynamically approach for building an agent behaviour, pushing it closer to a self organized system. Thus the agent behaviour can be changed by controlling the components it is made of(i.e. add a component, remove another). It is a nice generalization not available on many frameworks; now the behaviour emerges from internal and external interactions instead of being programmed explicitly. Next, a classification inspired from Briot et al [7] will be described.

- Cycle-Based architecture – It is somewhat close to how a computer works because it considers the agent actions included in a main loop which runs indefinitely. It works based on a stream of data and control commands generating output streams of the same types(control, data).

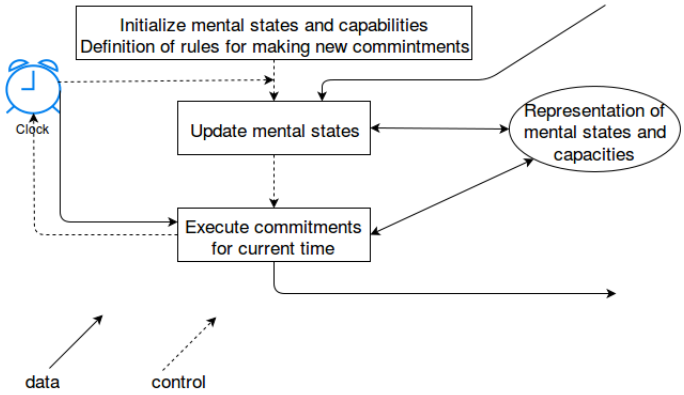


Figure 2.1: A perspective on Cycle-Based Architecture [18].

- View-Based architecture – This one is generic, including several sub-architectures(i.e. VOLCANO, GAM). The point is that the framework has several components, but they are not made to be included in an agent, but to organize the framework. For example the VOLCANO architecture is composed of an agent, an environment, and a part for interactions and organization.
- Level-Based architecture – In this one, there are several abstract models, each is considered a level, for example: world model, social model, self model. InterRRaP [13] is an example with a three level structure: cooperative planning, local planning, and reactive behaviour.
- Behavior-Based architecture – The agent has several behaviors organized in a hierarchy.

2.3 Web based communication protocols

When we are thinking about connectivity over network, one of the first thing that comes to mind is the network socket. A more specific notion is the "Websocket", which works using HTTP protocol over TCP/IP and is usually used in browsers for a permanent connection. In

tATAmI, a communication platform over HTTP has been implemented in order to be able to communicate in a very generic manner on a common used port(HTTP or HTTPS port). Next we will present the available approaches and a comparison between them.

- **Regular HTTP** - The client requests the web page from the server, then the server calculates the response and sends it to the client.

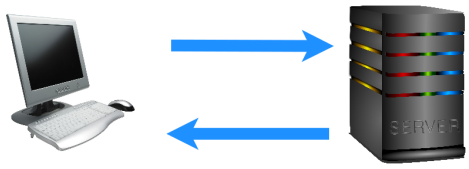


Figure 2.2: The regular HTTP model

- **AJAX Polling** The client requests a web page from a server using regular HTTP, then the requested web page execute javascript code which requests a file from the server at regular intervals. The server calculates each response and sends it back, just like normal HTTP traffic. So the client uses a polling method with a lot of useless requests.

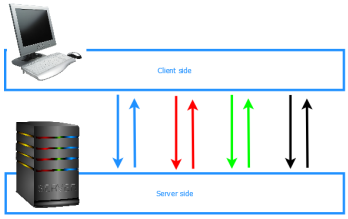


Figure 2.3: Ajax Polling

- **AJAX Long Polling**

  The client requests a web page from a server using regular HTTP, then the requested web page executes javascript which requests a file from the server. The server does not immediately respond with the requested information but waits until there's new information available. When there's new information available, the server responds with the new information. The client receives the new information and immediately sends another request to the server, re-starting the process. The client have to reconnect periodically after connection is closed due to timeouts or data eof.

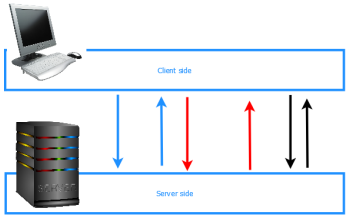


Figure 2.4: Ajax Polling

- **HTML5 Server Sent Events (SSE)** A client requests a web page from a server using regular HTTP. The requested web page opens a connection to the server. The server sends an event to the client when there's new information available. There is a real-time traffic from the server to the client.Only server can send data to client. If client wants to send data to server it would require to use other technology/protocol to do so. Is not possible to connect with a server from another domain.

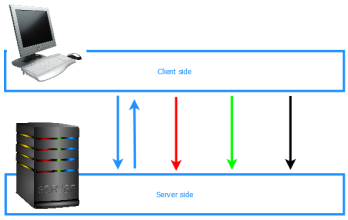


Figure 2.5: HTML5 Server Sent Events (SSE)

- **HTML5 Websockets** A client requests a web page from a server using regular HTTP. The requested web page executes javascript which opens a connection with the server. The server and the client can now send each other messages when new data (on either side) is available. Real-time traffic from the server to the client and from the client to the server are provided. The server must have an event loop. With WebSockets it is possible to connect with a server from another domain. It is also possible to use a third party hosted Websocket server, for example Pusher or others. This way we would only have to implement the client side.

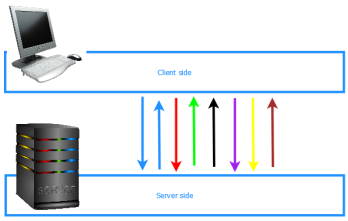


Figure 2.6: HTML5 Server Sent Events (SSE)

2.4 MASS C++

[9] While Java has been the language for the implementation of tATAmI since the beginning of this project, we will also discuss C++. There are several C++ MAS frameworks out there, and MASS C++ is the most representative.

MASS C++ is "intended to parallelize a simulation program that particularly focuses on multi-entity interaction in physical, biological, social, and strategic domain" CITATION. From the paper which describes the framework it can be concluded that it is intended to include the entire functionality of the components and to be scalable for a large amount of agents. It is focused

more on Self Organizing Systems simulations, as it is designed to simulate physics problems, artificial societies etc.

Meanwhile, tATAmI is more of an infrastructure for whatever components are created, meaning that a component can have a subcomponent written in any supported language. Therefore, we are interested in the performance of communication, data transfer rate, agent mobility, and portability. Moreover, tATAmI is more granular, intended to encompass a smaller amount of agents, but each of them with a larger functionality. It would be a loss of time to use C++ for a small performance gain because there is no need of intensive computation and specific memory management.

2.5 JIAC and microJIAC

First of the frameworks studied was JIAC. JIAC is a second version of a framework born from a diploma project developed in 2009 for devices with minimal Java support. Seems like the developers were concerned about the application size and they used ProGuard [1] – an open source tool distributed under GPL licence used for shrinking Java class files, for optimization and obfuscation. With it one can create smaller archives, make reverse engineering harder and detect dead code; some results they obtained are somewhat impressive, for example, for the Ant build tool they obtained a downsize of 90 % (overall results are between 18 %-90 %). Although memory does not seems to be a problem if we think about at least 1GB RAM available on tATAmI-supported devices, I will take it into consideration and maybe add a post processing step in the build process, at least to see how much I could shrink the framework memory footprint.

The parsing and processing of the JIAC is done at compile time. We cannot do this on tATAmI because we want, for example, to be able to modify the components of an agent at runtime. We cannot rebuild the application every time the configuration of the agent is changed because this option is not available on Android and may not be on other embedded systems either. Let's say we could have the resources to compile, even then It would take a considerable amount of time (especially on systems with reduced processing power) or, if the agent would be compiled on a different PC, it would lose autonomy (a PC would always be required). A problem that I also encountered was avoiding Dynamic Class Loading, especially on Android. To solve this, they implemented a reflection class, not applicable to me as this was not needed. I will discuss later how I approached this issue in the Android implementation of tATAmI.

In microJIAC (the Micro Edition JIAC), reflection was not available because they used CLDC (Connected Limited Device Configuration) which is a Java ME (Micro Edition) subset of libraries.

After this study I can conclude that, compared to tATAmI, microJIAC had limited features due to hardware restrictions back in 2006-2009 when it was designed and implemented. Now the memory footprint is not such a big concern as it was then [16].

[1] http://proguard.sourceforge.net/

2.6 JADE

JADE(Java Agent DEvelopment framework) is probably the best known Multi Agent System middleware available, distributed under LGPL license. In JADE, the communication is also essential, they use FIPA ACL (based a lot on KQML), which is a complex and robust communication/coordination/negotiation system. Support for content languages and ontologies, agent mobility and a suite of tools is available, including GUI. It is easy to conclude that JADE is very flexible, with a lot of features, a lot of support for many FIPA standards and so on. At the same time, having a lot of features is a disadvantage, the truth is that many of the design principles are deprecated at this time resulting in slowing down and reducing the portability of the framework.

2.7 Agent Factory & Agent Factory Micro Edition

A lesson that should be learned from this framework is the Eclipse integration, it is a lot easier and attractive for a user to implement an agent component using a graphical interface, however tATAmI project won't include an IDE at this time. Agent Factory is a standard-compliant framework, with a large amount of contributions, resulting in a lot of principles, 80 papers, articles and books which contain information about the implementation of reasoning behind this framework, several Agent Oriented Programming Languages implemented, the nature of agent environment, Algebraic Data Types, Logic-Based AOP Languages, Evaluating Communication Strategies in a Multi-Agent Information Retrieval System and a lot of other symbolic artificial intelligence concepts.

The Agent Factory Micro Edition is a version of the framework optimized for Constrained Limited Device Configuration (CLDC) / Mobile Information Device Profile (MIDP) compiled with a specific compiler crated for intentionally for the framework. AFME defines the term "Perceptors" which are models for generating beliefs about the agent's state and environment. "Actuator" is an abstraction of an object that is doing an action, for example the act of moving the electric motor clockwise. The "ElectricMotorComponent" is an actuator, a term that will be also "borrowed" in this work. Modules are shared references to the same object, for example if a Perceptor is aware of a stone in front of it and an actuator wants to move the stone, then the stone is a Model. In tATAmI however, every component has its own model of a real object which is actually a projection on the respective component. "Services" use the same idea as modules, but they can be shared between several agents, meanwhile modules can be referenced only within the same agent. Other models imply the usage of predicates, plans, reasoning which are not very important for us at this stage.

2.8 Summary

The "Perceptor" is the model for input data(i.e sensors), the "Actuator" is the model for the agent's output(i.e. motor). We will use these two models keeping the name "Actuator", but

“Perceptor” will be replaced with “Sensor”.

ProGuard is a tool for shrinking, obfuscating and optimizing Java byte code used in a post processing step.

The processing power needs to be considered due to the fact that on embedded systems, the processing power is limited although the memory is considerably bigger than when most of MAS Micro Edition were implemented.

Chapter 3

Websocket communication and agent mobility

The communication over the network is performed using a project called Java WebSocket[2]. It implements a communication protocol that allows us to send messages between public-private IPs. Java WebSocket is known to work with Java 1.5 (aka SE 5 / Java 5) and Android 1.6 (API 4). Other JRE implementations may work as well, but haven't been tested.

3.1 The Websocket library project

The project documentation gives two examples in order to run the server and the client side. From the command line, the server is executed with the command:

```
java -cp "build/examples:dist/java_websocket.jar" ChatServer
```

From the command line, the client is executed with the command:

```
java -cp "build/examples:dist/java_websocket.jar" ChatClient
```

In order to use the server, the `org.java_websocket.server.WebSocketServer` abstract class must be extended. The Websocket server establishes socket connections through HTTP. In order to use the client, the `org.java_websocket.server.WebSocketClient` abstract class must be extended; it can connect to a valid WebSocket server. The constructor expects a valid `ws://` URI to connect to.

3.2 Overall websocket implementation

The implementation consist of 4 classes: AutobahnClient which is the implementation of a client using the Web-Sockets library. AutobahnServer is the implementation of a server using the

[2]https://github.com/TooTallNate/Java-WebSocket

Web-Sockets library. WebMessagingComponent is the implementation of the agent's messaging component using the server and the client mentioned above. WebMessagingPlatform is the implementation of the platform which supports communication.

The platform can be a server, client, server and client at the same time, or none. These changes can be made though the scenario file. The configuration is made in "scen:config" tag. The mainHost says whether the platform is client, server or both, the port of the client (mainPort), the port of the server(localPort) and the server host. For example: `<scen:config mainHost="server client" mainPort="9001" localHost="127.0.0.1" localPort="9001" />` In this example, the platform is both server and client, the ports are both 9001 and the host is localhost(127.0.0.1).

When an agent becomes available, it sends an internal message to the server for the registration process. The server keeps a mapping from agent name to the corresponding socket.

The platform keeps a mapping from the name agents to the corresponding MessagingComponent, such that, when a message arrives, a specific method in the specific Messaging component will be triggered from the client through messaging platform.

3.3 Messages

Below is detailed each type of message for server-client control.

- `::internal::` This kind of message is used when the server and the client exchange messages as, for example, the client sends all the agents linked to it to the server in order to register them for routing.
- `::handshake::` This message is sent when the platform wants to start the client but it does not know if the server is running. When this happens, the client keeps sending those types of messages to the server until the sending is successful.
- `::container::` This message contains the name of a container which wants to register to the server such that the server will know where to route an incoming agent.
- `::mobility::` This message contains parts of an agent or even a whole agent.

At first I implemented a working work flow followed by bug fixing and side feature implementation. For example, at first, the destination parameter was an IP address, then I implemented a registration system with the actual names of the containers.

3.4 Limitations of Websocket messaging

The Java_websocket project we are using for communication over network can send strings not bytes. Maximum size of a Websocket frame according to RFC-6455 [8] is $\approx 18 \times 10^{18}$ bytes, thus a sufficient size for any agent one can think of. However, in order to avoid possible limitations

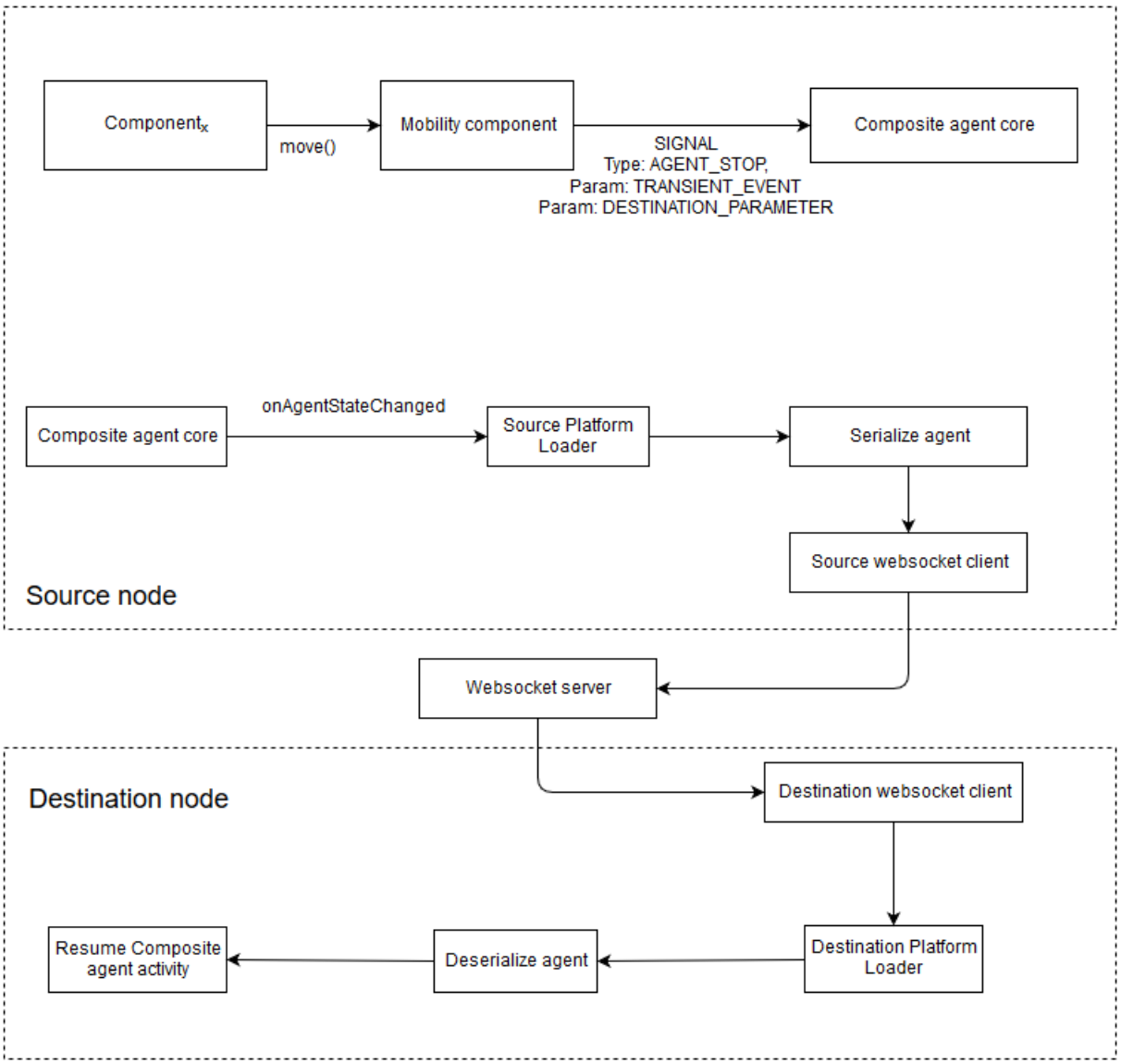


Figure 3.1: The overall Mobility Architecture

from the Java_websocket module and in case it will be changed, we adopted a fragment-by-fragment transport approach. The size of a segment can be adjusted , thus the entire agent can be sent in one transport, in that way the performance is not impacted.

3.5 InputComplexMessageTokenizer Structure

On the source side we adopted a reader monad approach, the context being the serialization of the agent and the reading function is the send function. The purpose of this structure is to receive an array of bytes as an input and to apply a send function over it. First however the input needs to be restructured meaning it must be split based on `BATCH_SIZE` class member.

After that it works exactly as a tokenizer.

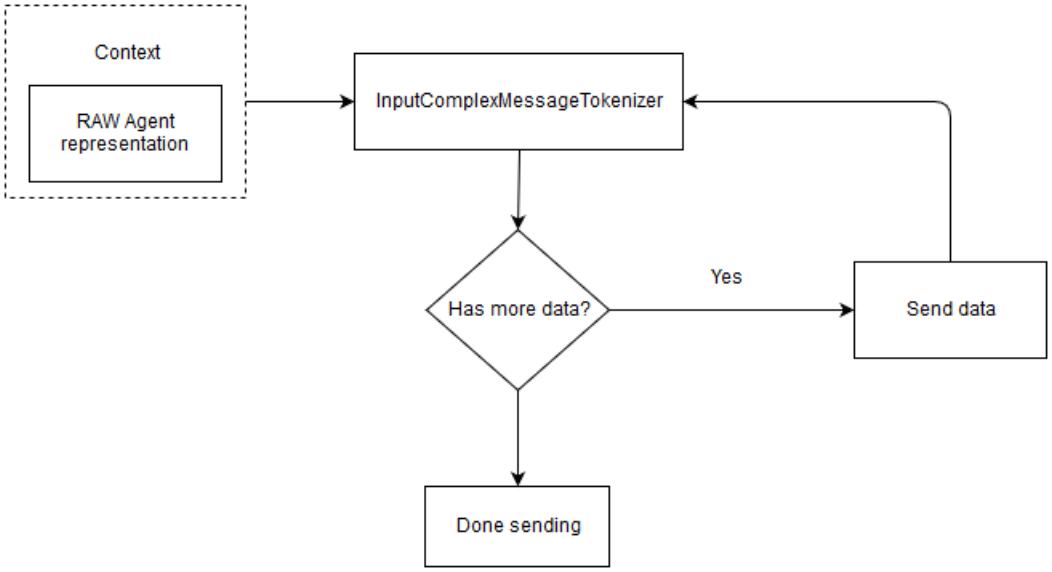


Figure 3.2: The Sender Structure

3.6 OutputComplexMessageAgregator Structure

The component on the receiving side is seen as a writer monad with initial empty context which becomes richer with every received piece of the serialized agent. The usage of this component is the following: Initialize the module(empty context), then, every time a package is available add it to the structure until all packages have been received. Finally extract the context in the form of a bytes array. Aditional parameters: `int mReceivedCount` - how many messages have

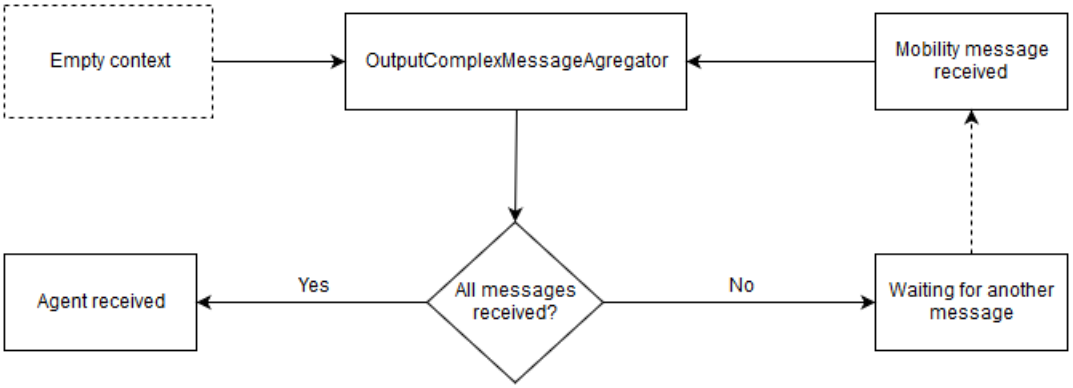


Figure 3.3: The Receiver Structure

been received,`ArrayList<String>` buffer the buffer with all received messages, `int mSize` – the number of messages(at first it is 0, when a message occurs is modified)

3.7 Package Structure

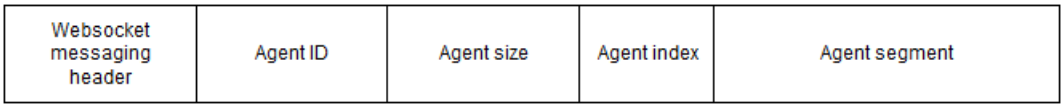


Figure 3.4: The Message Structure

The message that is sent over the network has two header levels: the header imposed by the mobility type of content and the format in which he agent is serialized. Such that the content is composed from: the id of the agent which is transported on other network node, the agent size delimited by "|" characters(The agent size is the length of the string representation), the agent part index and the representation of the agent as a string as shown in the figure 3.4.

3.8 Summary

Websocket communication is, first of all, a more flexible way of exchanging messages and an infrastructure for agent mobility. The implementation uses several types of messages for internal communication and for the agent mobility, two special structures for splitting and aggregating are used. The messaging interface is inspired from JADE.

Chapter 4

Portable core functionality

4.1 Existing Parts

tATAmI is a Multi Agent Framework System designed to be used in Ambient Intelligence environments, it is composed of three main parts: the Agent, the Simulation and the Visualization.

4.1.1 The Composite Agent

The Composite Agent is a generic implementation of the agent containing all the components of an agent and ensuring inter-component communication, the agent state management and the events distribution. The class `CompositeAgent.AgentThread` is the main thread of the agent in which events from event queue are processed and given to the components if it is the case. CompositeAgent also contains control methods for starting, stopping, killing, exiting, state changing, of the agent. I mention this class because here a new state has been added (Transient state) which means the agent is not working and it is in a stopping process.

4.1.2 The scenario file

The scenario is file is an XML file where it is specified the agents configuration to be created on a specific machine; from the name of the agent, its components along with their class paths, to custom parameters and containers name. As an observation, since the Dynamic Class Loading is not used any more and the factory approach taken its place, the class path is more of an unique identification string which has a correspondent in the factory class `ComponentFactory`.

Below can be found an example on how to configure the platform to use Websockets. The `mainHost` attrbute specifies if the the current instance of tATAmI is a server for communication, a client or both. The The `localHost` and The `localPort` parameteres are to configure the client.

```
1 <scen:config mainHost="server client" mainPort="9008" localHost="
    127.0.0.1" localPort="9008" />
2 <scen:platform>
3         <scen:parameter name="name" value="websocket" />
4 </scen:platform>
```

4.1.3 The Bootstrap Process

`Boot` is the root class of the core library. Placed under tatami.simulation, it is where the platform and agents are being loaded according to the XML scenario file. The project platform is made in such a way that the standard agent platform can be changed from the configuration XML file, thus it is loaded at runtime through the reflection mechanism.

4.1.4 The PlatformLoader

The Platform Loader is a standard for the classes responsible with platform loading. It contains an enum, StandardPlatformType, used in retrieving the class name which will be loaded.

4.1.5 JADE based component

Beside the Websocket communication, tATAmI can make use of a JADE based component.

Inside the tATAmI project, JADE platform is located in the package "tatami.jade". It contains the following classes:

- **JadeAgentWrapper** – A wrapper over an AgentManager instance
- **JadeComponent** – Component that handles the interaction with the Jade platform.
- **JadeInterface**
- **JadeLogWrapper**
- **JadeMessaging** – Implements messaging functionality, using the features offered by Jade
- **JadePlatformLoader** – Implementation of PlatformLoader using Jade as an underlying framework for agent communication and mobility.

Inspired from this package, the messaging component over Web-Sockets was implemented.

4.2 Components added for portability

I was thinking about how to implement the framework such that some parts will be common and some of them specific, the shared part is of course the core of the agent consisting of the entire logic excepting the graphical interface.

Until now, the GUI was provided by the `VisualizableComponent`, thus a component of the agent which is somehow embodied in the framework. The first step was to split this component, instead of it, a system independent "ControlComponent" and an "external interface" is used. At the agent paradigm level, the external interface is a receiver of the environment events, in this case for platform and agent control. Every type of interface (GUI/GLI) will work through this programming interface in order to gain control of the framework. The only component that could not be used on the other OS was the graphical interface.

4.2.1 Control component

The control component is an Agent-Level Interface, the requested functionality is summarized below:

- Forward commands to a specific agent - it can choose how to interpret it.
- Create the agents - Based on the scenario file, the framework starts all the agents.
- Start the agents - Each component of each agent is started
- Stop the agents - The simulation is interrupted, however this is not a forced stop.

This component is controlled through the GUI/CLI depending of the system it is installed on.

4.2.2 Agent Logging vs Usual Logging

The output of the agent was taken directly from what the agent was printing, so if the developer printed something to check a variable and if the agent also printed something, it would have been the same thing, so I observed that the log had two different purposes: to print usual logs and to forward agent output and I thought that it would be a good idea to separate this functionality. The outcome is that now the usual Log uses a "\"Log\"" class with static methods(usual implementation, nothing special), meanwhile the agent output is forwarded to an interface that I will describe later, as concern us, for now, let's consider that when an agent needs to log something it will call a method from a singleton class we will talk about later. In other words, there is a log for framework development which is the layer under the MAS paradigm and another log considered as the agent output from the MAS abstraction layer. On the last one (Agent log) I tried to implement a signal-like approach inspired by Qt signals, unfortunately for this case, Java uses listeners, and the listeners needed to be propagated through a lot of classes which would have mean to refactor a significant amount of code just for forwarding some messages. I found the following approach: I created a singleton class, in this class I can also register listeners. When an agent needs to log something, it calls a method from the singleton class(like emitting a signal) then the log call is signaled to all the registered listeners. Yes, a listener is still used, but it does not have to be propagated in the whole framework code. It have to be thread safe of course, but It would have been so even if I would adopted the other code-refactoring-time-consuming-bug-source method. The advantage

is: minimum modifications, lower bug risk, side-effect approach opposite to context approach, we gain simple listener registration.

4.3 Agent Loading

Due to the fact that the Dynamic Class Loading is not available on Android without complicated methods, a factory class for creating each agent has been created.

4.4 Overview

The already implemented parts for simulation, bootstrap, platform loading and, not in the last place, the agent was a good robust base for the tATAmI core. In order to make the framework portable, the component for visualisation has been replaced with a more general control component. A separation between the Agent Log and the Debug Log has been introduced.

Chapter 5

Android-specific implementation

Android can be considered as a specific case of embedded system, due to its privacy and security policies. An application created for Android will not work on other operating systems, there are few hacks and workarounds out there but an official best practice is not available at this moment. Almost all agent-based frameworks have versions for this often used operating system. Relative to tATAmI, the role of this dedicated system can be to control the agents, to oversee their activity and the patterns that emerge from their interactions. The tATAmI Android application is very easy to install, requires permission only for Internet connection, network state and wireless state. In the future maybe it will be available in Google Play although it is addressed to Multi Agent Systems research domain, the users could create agents and play with the behavior determined by different individuals actions.

In order to decide how will the application be modularized and implemented, especially the inter processing art, I've made some research, as follows.

5.1 AIDL

AIDL stands for Android Interface Definition Language, it is an Inter Process Communication(IPC) method used on Android devices and the only one available from user level(In order to be able to use other Linux specific utilities, system privileges or even root is needed). IPC satisfies several needs, one of them is load distribution on different processors, however this is not the case because the agent is targeted to use as less processing power as possible, instead it can be used for exposing a programmable interface for other applications. In our case I studied it to see if it is suitable for splitting the graphical interface from the framework core.

The AIDL requires an .aidl file where every method of the interface is specified, although the format consists of a simplified and modified version of Java code, it has certain limitation, the designer should be careful using this feature. For example, .aidl interfaces does not support overloading. In extent to Java, every non-primitive parameter has a specifier which tells if the class is for input or for output purpose, it is strongly recommended to check if a variable really

needs both input and output property because the marshalling process is expensive. In the build process, based on this .aidl file, java code is generated. Aside of this .aidl, the server and the client needs to be implemented, thus, either I polute the core code with android specific methods, either I create a wrapper to interact with the server.

Actually the AIDL approach has a significant logic behind(relative to the focus of this project) that needs to be implemented, including object serialization which is different than the classical Java serialization. If we talk about this serialization, it needs to be said that any class that will be serialized will implement Parcelable and will define writeToParcel method and other few others. It is simply too much to have to change a lot of already existent classes and to keep them work on other platforms.

PROs and CONs for using AIDL: PROs:

- Modularized framework - simpler to understand
- In case the core crashes, the user does not simply lose the application control, he can, for example restart the core from the interface, or he can get a warning message.

CONs:

- The application could not run with the existing jar only, it needed an Android Application wrapper anyway
- Bigger memory footprint, expensive marshaling for serialization
- Serialization needed for every object, thus the context needs to be recreated
- A security issue maybe, a programmable interface is more vulnerable than a graphical interface.
- Some objects needs to be modified in order to be able to serialize them

Following this case study, I concluded that the best approach is to include the GUI in the main Android project as we anyway have to make different builds for PC, Android and Embedded. AIDL is a good approach for a robust application, but it is the portability enemy.

5.2 XML Parsing and validation on Android

XML is the format of the data which drives the agent behavior, every instance of the framework have a configuration file in which is specified, among other informations, the containers, the agents, their components, global settings for the platform, as the server IP for example. Among the configuration files is also an XML schema which validates them in order to avoid wrong or misunderstanding. On tATAmI PC a custom library was used, but it did not work on Android for several reasons: -Dynamic class loading in Android is not recommended, it depends of the virtual machine the Java code runs on. -The validation didn't work due to an Android bug. -The Android latest version have a runtime permission policy which wouldn't allow us to read the file from disk in the same manner. -Configuration files should have been moved on Android in an unfriendly manner, by using ADB. -The XML library wasn't suitable for reading XML

from input streams(later we will discuss why input streams). Next I will talk about some case study for the above issues that occurred when implementing the Android application.

At first guess I thought that the Dynamic Class Loading is different on Dalvik and ART. Dalvik is the old virtual machine supported by Android, ARM is the new one, both are optimized for running multiple instances of virtual machines. In order to explain the differences I will start with some Java notions. Java came on the market with the advantage that the programs written in this programming language can run on every machine that has Java installed, from here the slogan "code once, run anywhere".This feature came with the cost of translation, more precisely, this is what is happening after writing some java code: The code is translated at build time in byte code, then, at compile time, Java executable translates this byte code into binary machine specific array of instructions for every architecture, that's why Java is slower than C++, but it's faster than interpretation. Due to the fact that Java translates this byte code, it is said that is actually a virtual machine. Next, Android used Java, but it needed a virtual machine that can have multiple instances and keeps a good performance as much as possible, it is a critical issue for this OS because all user level applications uses this virtualization, C++ is even slower in most cases because it is interfaced with Java(Android SDK) through JNI(Java Native Interface). They created Dalvik VM, which uses JIT(Just In Time Compilation) which means that the code is dynamically compiled at runtime. Next, a startup occurred with the intention to optimize this virtual machine, the new one is called ART and have the advantage that it compiles the application into native code at runtime, so the program will install slower because of the compile time, but the runtime is faster as it lacks dynamic compilation. Both classes uses .dex files, so the Dynamic class loading is done in the same way, a .dex file is a compiled Android application code, it can have a maximum size of 64KB. In conclusion I can say that Dynamic Class loading is not recommended in Android, there is very few official documentation and it is very rare used. I replaced the DCL from tATAmI-PC with already loaded classes through a factory design pattern.

In order to keep the same architecture, the configuration files needed to be copied on the device's storage, this could be done in two ways: put them into assets folder, than copy them on storage(at first run) or copy them with ADB/SSH. I tried to copy the files manually on device to see if it really works, after several attempts I concluded that it does not because the SO does not let you to read from the internal storage and certainly does not let you write. I made these attempts on a Nexus 5 phone which does not have an external storage and I could not assume that all phones have. I decided to give up this approach and o keep the recommended Android methodology, implying that the files will be kept in assets folder such that no other permission(Read/Write for Internal/External storage) would be required. This decision generated another problem further described.

The library for XML parsing and validation is `net.xqhs.XML.jar` , however this works only for "File" class, not for `InputStream`, `RandomAccessFile` or other kind of stream, so I had to get the source code and do refactoring such that the library can parse and validate `InputStreams`. Next I got a `SchemaFactory.newInstance()` exception, after frustrating trials I found out that it is a known Android issue ([ISSUE]).

Android has dedicated parsers, however when it comes to validation, they use the same code as the existing library and I reached to the same end point. It seemed like a good idea to search for an alternative for XML parsing and validation on Android. There are few options, the most used is Xerces, however it does not have a validation feature. Therefore I decided to disable XML validation for now.

5.3 Graphical User Interface

On Android, the GUI has a single possible API for implementation: Android SDK, so, there is no flexibility is this regard. The GUI is implemented based on Android SDK along with a small back-end responsible for communication with the tATAmI core. The backend is composed of a listener where the agent output first comes, a "Log Hub" where the messages are scattered aware of the agent who dumped them, the "Backend" class which is a controller for all the back-end structure and an "Agnet Log Backend" where the log for every agent is stored. Keep in mind that the "Agent Log" is not the same with the logs printed to debug the framework and is treated accordingly, in other words it is a higher level of logging.

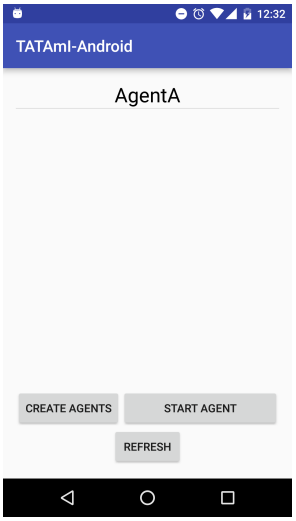


Figure 5.1: tATAmI-Android GUI example

5.3.1 The Main Panel

Is meant to be a welcome page to introduce the user to framework capabilities, contains a scrollable area with a description of what tATAmI is, an image and a button which leads to the Agent Manager page.

5.3.2 The Agent Manager Panel

Contains a list with the created agents, each entry is clickable and composed of a label with the name of the agent and the status of each agent.

1. The list with all the constructed agents
2. A button for constructing the agents
3. A button for starting the agents
4. A dialog box for choosing the scenario

5.3.3 The Agent Panel

This panel is designed for each agent control and monitoring. The user is able to enter commands for the agent through the Control component, to check information from the agent logs and to clear the current output console.

1. A text area where the output of the agent is dumped.
2. A button for clearing the output console.
3. An input text area where commands can be entered
4. A button for submitting the command entered in the input text area

5.4 Summary

The good part of programming for Android is Java, but the way things are working in Android SDK is certainly not the same as usual Java programming, the way classes are loaded, the virtual machines Android is using, issues regarding the XML validation, the restricted GUI library and not in the last place, the Android very restrictive policies make it an anti cross-over programming language, fact that was reluctantly mitigated.

Chapter 6

Raspbian-specific implementation

The other embedded system is Raspbian, a lightweight Linux based operating system created for Raspberry Pi board. Raspberry Pi is a credit card-sized ARM GNU/Linux computer designed, first of all for teaching basic computer science notions in schools. It ought to be very suitable for prototyping and data acquisition, domains which I will also exploit further. Data acquisition is the first step in trial-and-error most trivial machine learning approach, for this I built a basic robot composed of a few sensors and two electric motors. The Raspberry Pi system I have tested on(Raspberry Pi B+) has 1GB RAM, 900 MHz ARM processor, 250 MHz core, 450 MHz SDRAM, 2 overvolt which is just fine for basic agent testing.

It has 40 pins as shown below in Figure 5.1, 26 GPIO, 8 Ground pins, 4 power pins (2x3.3v and 2 x 5v), and two special pins.

There are two "standards" for numbering the pins: BCM and BOARD:

- BCM(Broadcom SOC channel)
- BOARD numbered from left to right and from top to bottom

Different Raspberry Pi boards have their own BCM order, you may want to see your specific board scheme. Before starting to describe the connections, remember that I will refer to the GPIO pins in the BOARD format, because I need to specify which of the Ground and power pins I'm talking about. Meanwhile in the code I've written, the pins are referenced in BCM format. 6.1

6.1 Java RMI

In order for the agents to achieve Raspberry Pi specific capabilities, new components have been implemented for actions and environment perception. First, the hardware components have been tested using the official Python library for accessing GPIO pins, SPI and I2C interfaces. Then the functionality have been reimplemented using a Java library: Pi4J to avoid creating a new process taking into account that the overall hardware resources are limited.

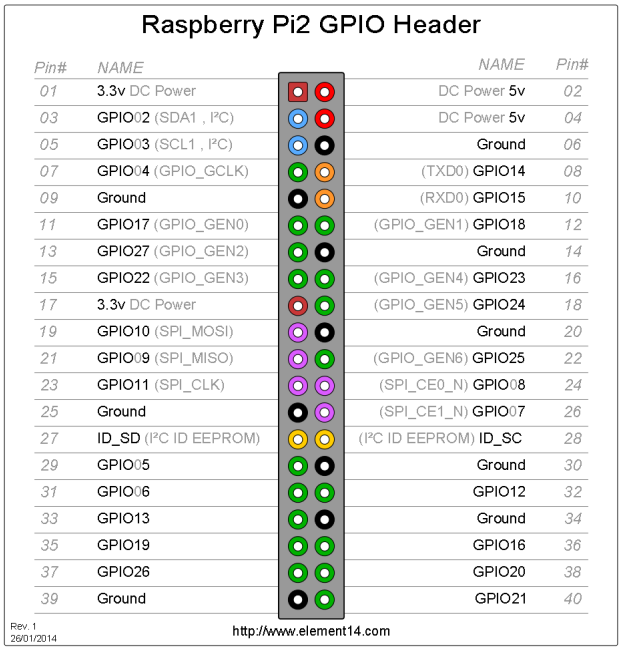


Figure 6.1: Raspberry Pi pins

On Android, a tight connection between the user interface and the core was adopted due to the fact that the AIDL alternative was too specific, however on Raspbian, which is a Linux distribution, a common IPC approach might be used. Because Java is a virtual machine, the Inter Process Communication has poor performance, but it has the advantage that is much more secure. One of the methodology that can be used for such communication would be to implement a socket-based networking created from scratch which is not necessarily complicated, but it raise a high bug risk. Moreover, File-sharing/locking would need a pretty big logic behind for a good producer-consumer algorithm, on the other side a basic implementation would introduce very poor performance issues especially on embedded systems. ActiveMQ or JBoss Messaging could be used, they are libraries built over RMI. At the end of the research, RMI was chosen because it is the most used API in such regards, moreover it offers a flexible over network control although at the moment is used only on loop back network interface.

If, on Android, a listener is used in order for the "Agent Log" to reach the GUI, with RMI this is not possible because the API supports only one way calls, an alternative would have been to use JMX(Java Management Extension) but instead, another server have been created on the client, such that the connection became two way connection.

6.2 Components for using the available hardware

Three types of sensors and two motors have been chosen to show how different types of sensors can be interfaced.

Pi4j is a friendly object oriented open-source(LGPL License) I/O API for using the Raspberry Pi capabilities, this library is used for all implementations of the below components. It is to be mentioned that the numbering of the pins used in Pi4J is different than the original numbering scheme.

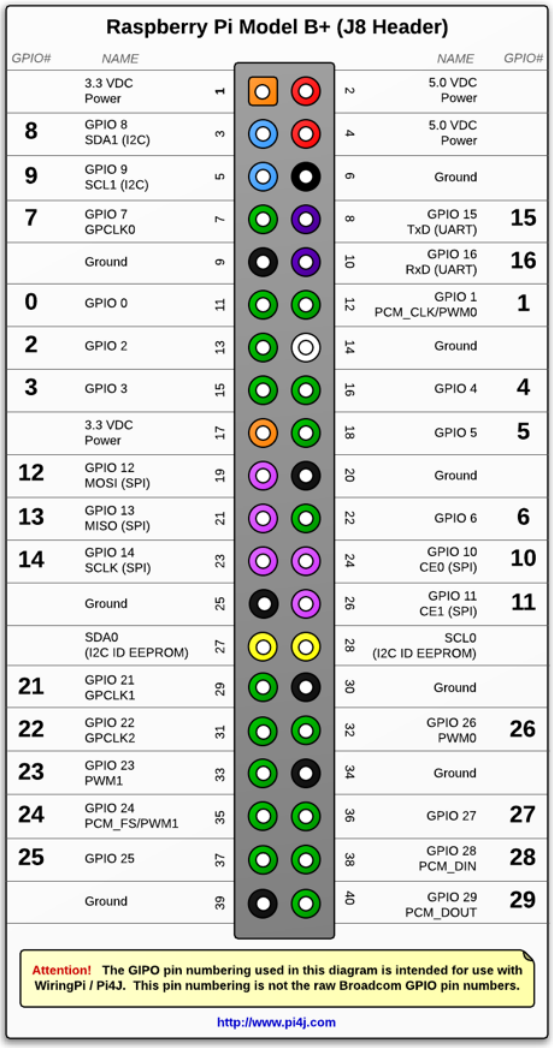


Figure 6.2: Pi4j Numbering scheme

6.2.1 MMA8452Q Component

The MMA8452Q [1] chip is a 3 axis, 12 bit accelerometer interfaced with I2C. It was linked to the GPIO pins in the following way: 3.3V - Pin01, GND - Pin06, SCL - Pin03, SDA - Pin02, the other chip pins are free. To enable I2C on Raspbian do the next steps:

```
1 $ sudo apt-get install python-smbus
```

[1] http://www.nxp.com/files/sensors/doc/data_sheet/MMA8452Q.pdf

```
$ sudo apt-get install i2c-tools
$ sudo raspi-config
```

Next, a semi visual interface will appear, use arrows and enter to navigate and select options. Select "*AdvancedOptions*" → "*I2C*" → "*Yes*" → "*Yes*" Go to "*/etc/modprobe.d/raspi − blacklist.conf*" and comment the line containing "*i2c*" by putting a "#" in front of the row. Then reboot the system.

To see if the sensor is plugged in, type

```
$ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- 1c -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
                                         ..................
```

The testing, Python script I've made consists of some set-up and a reading from i2c interface, then, in the tATAmI component, a timer is used to get samples periodically, in this way the period can be set by the user. Pi4J specific classes for I2c bus and I2c addresses are used.

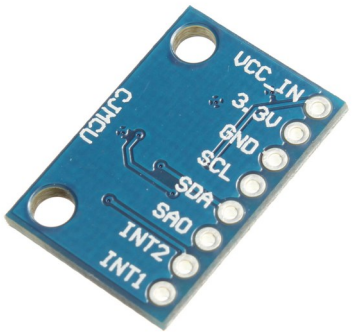


Figure 6.3: MMA8452Q Accelerometer

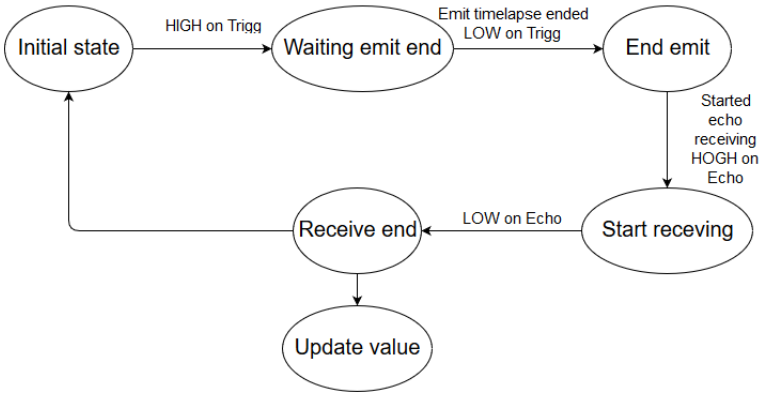
6.2.2 HC SR-04 Component

The hardware component is a HC SR-04 [1] distance sensor, using an ultrasonic sound emitor and receiver it measures the time taken by the sound to return. It is used to measure distances between 2cm - 400 cm and is powered on 5v pin. The chip has 4 pins: Vcc, Trig, Echo and GND, when Trig is on HIGH signal it emits a sound, when Trig is on HIGH signal, it receives the echo, I connected these pins in the following way: Vcc- Pin02, Trig->Pin16, Echo->Pin18 , GND-> Pin06, remember, in BCM format. When the Python script for testing this component was written, I wanted it to work like a getter and retrieve instantly the value representing

[1] http://www.micropik.com/PDF/HCSR04.pdf

the distance. This was not possible, because it take some time for the sound to be emited and received, small amount of time but the point is that the function could not have been synchronous. Thus I implemented a loop which updates a cached value, to do so I used an automata with several states as shown in Figure 6.5 In the HCSR04Componenet however, only two states are used, instead, I'm using a listener from Pi4j library which is triggered when Echo signal is changed from LOW to HIGH or inverse. This way, the receiving moment is recorded and, having also the time when the signal was sent, the distance can be computed.

Figure 6.4: Distance HC-SR04 Sensor



distancesensor

Figure 6.5: Testing script automata

6.2.3 Pressure sensor Component

The sensor[1] is different than the other two because it is analogous, and Raspberry Pi has only digital pins, to solve this I used an MCP3008[2] 10bits Analogue-Digital converter with 8 channels, meaning that 8 sensors can be attached to it. It communicates with the board though the SPI interface. To enable the SPI interface go to "$/etc/modprobe.d/raspi-blacklist.conf$" and comment the "$\#blacklistspi-bcm2708$" line.

The pressure sensor is analogous and has three pins: V_{cc}, V_{out}, GND; V_{cc} and GND are linked to the Raspberry Pi and the V_{out} Pin is linked to the first channel of the $MCP3008$ chip. The sensor used in the project came with an incorporated $10k\Omega$ resistor, it's a good idea to check this because otherwise, the board can be damaged.

[1] https://www.robofun.ro/docs/fsrguide.pdf
[2] https://cdn-shop.adafruit.com/datasheets/MCP3008.pdf

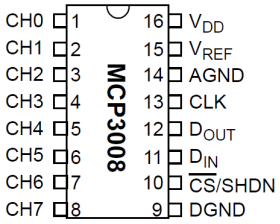


Figure 6.6: MCP3008 AD converter

Figure 6.7:

6.2.4 Electric motor Component

The differentiation of which motor is controlled by the component is done in the configuration file where , the corresponding GPIO pins are configured. Thus, for two motors, two components will be needed.

Figure 6.8: Electric motor driver L298n

In order to control the motor, a L298n driver was used [1] . The electric motor could not be controlled directly from the Raspberry Pi board because the current intensity required to spin is too high. The current intensity needed by a LED is about 20 mA meanwhile an electric motor requires at least 400mA, and the damaging of the board is almost certain.

6.3 Summary

Building the assembly around Raspberry Pi with the above sensors and components is a way of validating the framework and test its performance, it was a lot of fun building it although the

[1] https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf

accelerometer raised some issues, it is recommended to use ADXL345 [1] if the user is a beginner in electronics because the mentioned sensor is the most used accelerometer and a lot of code snippets can be found on Internet. It proved to be a good idea to use Meccano [2] parts as a chassis because it was very flexible and It wasn't necessarily to make a fixed design from the beginning.For example, in this case the chassis was built twice because at some point the parts were too crowded and it was hard to debug. Then in order to connect the Ethernet cable, a wheel had to be removed, but it wasn't a problem at the moment and then it was replaced by a wireless module.

[1] http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf
[2] https://en.wikipedia.org/wiki/Meccano

Chapter 7

Outcome, Testing, How to use

Below are detailed the validation methods, results, limitations and regulations.

7.1 How to use

This section is written from the user point of view, he needs to know how can he create a new agent, a new scenario, new components and not in the last place how to run the framework.

7.1.1 How to create a new component

A component is an independent unit which runs on a separate thread and can communicate with the other components. Usually(The way we are using tATAmI) the structure of an agent is composed out of several generic components and one agent specific component which make use of the other components, of course nothing stop us from implementing a more complex hierarchical structure. For now we will treat the many-to-one components model. First of all the new component which controls the other components have to be created. A component is actually a class which extends `AgentComponent`, an example can be found in Appendix 1("Dummy Component example")8.1. As can be seen, there are several methods to override, I won't detail them here because are really intuitive and comments can be found in the inherited class.However I will mention the handler which is made to receive messages from the other components and represents how the component reacts to the other components behaviour.

The structure of an agent if it isn't already obvious is a tree structure, the more the components are closer to the last level(leaves level) the more generic are (can be used by other agents). In the future when more components will be available, a clustered shape can emerge eventually using a genetic algorithm, probably this would be the best configuration this framework can have in means of agent complexity.

7.1.2 How to create a scenario file

The scenario file is a configuration of a specific platform, a platform can contain several nodes or containers(are the same thing), in the present implementation a platform is the framework process which runs on a single machine. An example of such file can be found in Appendix 2 8.1, as we can see we have a meta declaration of some parameters for the Websocket server and clients and, below, the declaration of a platform which contains a node, on its turn, the node contains an agent composed of several components. The components can have parameters for specific configurations.

7.1.3 How to run the framework

Because the interface for each system is different, there are several ways of utilising the framework.

- on PC

  Load the scenario: start the application

  Create the agents: press Create

  Start the agents: Press Start agents

  Monitor the agent: From the agents new windows you can see every agent's output

- On Android

  Load the scenario: start the application

  Create the agents: Press Agents Manager, Press Create Agents

  Start the agents: Press Agents Manager, Press Start Agents

  Monitor the agent: From the agents list select an agent and check the log

- On Raspbian (from terminal)

  Build: `make build`

  Load the scenario: `make run-raspbian`

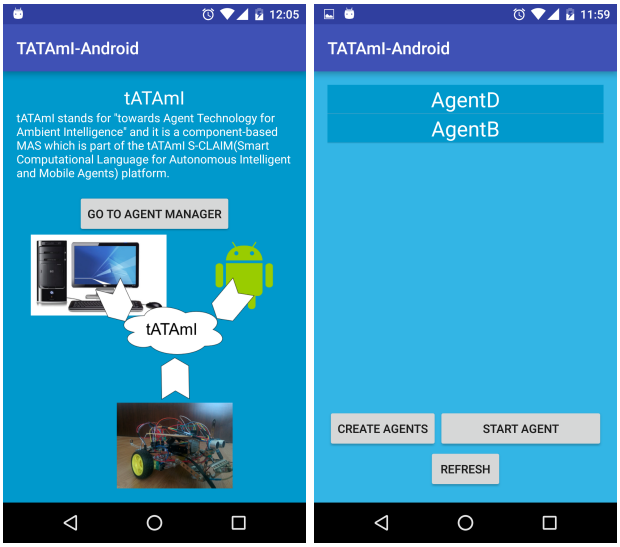
  Create the agents: Open another CLI, go to `tatami_CLI_HMI`, type `./run.sh -create`

  Start the agents: type `./run.sh -create`

  Monitor the agent: `./run.sh -names [name of the agent]`

7.2 Testing and validation

The validation of the Android version has been done manually checking all the GUI elements and their functionality. I verified if the agents are created correctly, if the agents start without



(a) The main Activity (b) Framework control

Figure 7.1: Android application screenshots

errors or crashes and if their output is correct, if the communication server and client work correctly and if the agents mobility is still working. In the figure **??** is a screen example of the Android application.

The testing of the embedded system components has been done at first using some python scripts in order to check that the sensors and the motors are connected correctly. Then, after creating the tATAmI components, the tests were performed again.

Below, in the Figure 7.2 is a picture of the embedded system for framework components development testing and validation. Due to the fact that the testing board is placed on top, the beneath components can't be seen so well. Anyway, in the front can be observed the force sensor, above it is the medium range distance sensor, at the tail of the system is the motor driver, on the test board are placed several 5V $\rightarrow$ 3.3V tension dividers and the MCP3008 AD converter. Beneath the testing board is the battery slot, the Raspberry Pi board, the accelerometer and of course the motors.

7.3 Results

It's somehow hard to evaluate the framework compared to others because: there is no MAS that uses Websockets for communication, there is no MAS that have support for exactly the same sensors we are using. For results I proposed the following tests:

The time an agent takes to move from a device to another.(Table 7.1) This test was realised by transporting an agent forth and back and dividing the time to two.

Sensor samples per second.(Table 7.2)

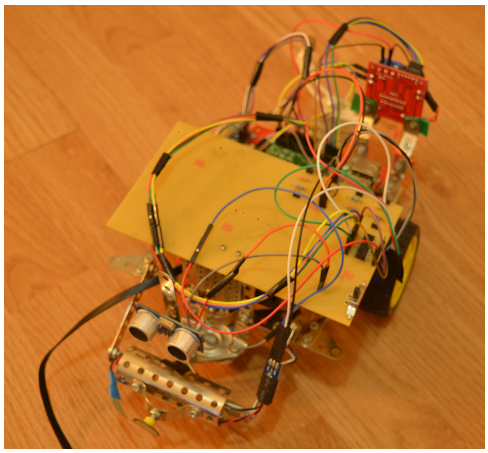


Figure 7.2: Embedded system physical outcome

Table 7.1: Inter device mobility speed

	PC	Android	Raspbian
PC	0.34	0.42	0.57
Android		0.4	0.55

Table 7.2: Samples per second for sensors

	Designed SPS	Real Maximum SPS	tATAmI SPS
HC-SR04	N/A	5	4.6
MMA8452Q	800	647	100
Force	200	170	90

The memory footprint overhead for every system(Table **??**). We can observe that the memory used is very small compared to the available system memory - under 2%.

Raspbian : 37.8MB
Android : 34.7MB
Windows : 36.9MB

During the tests I noticed a severe accuracy drop in the distance sensor relative to the python script due to the listener methodology used by the Pi4J library.This happens because any delay in the listener call is added to the time interval that takes to the sound to leave and come back.

Chapter 8

Conclusion

All the proposed tasks have been completed, including Websocket communication, agent mobility and portability on Android and Raspbian. Working for tATAmI was certainly a challenge, a lot of problems have been solved or mitigated during the development process. Due to the fact that the framework is actually an infrastructure, most of the encountered difficulties were engineering issues, however in order to decide what and how will be implemented, considerable research was needed. In the end we obtained a flexible Multi Agent System capable to harness a larger variety of devices. Far the hardest platform to implement on was Android OS because of its security policies, but in the and everything worked just fine. I enjoyed porting tATAmI on Raspbian because I built a small robot with a Raspberry Pi board along with several sensors.

Ambient intelligence is, in general a domain with many corner cases, design issues, and hard to integrate applications. A new generation of Multi Agent Systems is needed to cope with the increasing Internet of Things products. In the past it was hard to imagine how every device from a house can have the capabilities of a computer because of the size and price tag of such a machine. Nowadays however the computers can be small as a credit card and cheaper than a bus ticket, that being said, with a considerable number of devices in a house, maybe a management framework would be of some use with agents in every electrical socket, in the light bulb switchers, in the washing machine, in the watermeters and so on.

Over the last two years of research a lot of valuable experience in Ambient Intelligence, Ubiquitous Computing and embedded systems have been gained following to be used in real life projects that are to come.

8.1 Future work

Minor features: extend the Control component to receive commands, for example, from an input text. tATAmI will be developed further after this project, one of the main direction I think it should go would be developing a machine learning component using TensorFlow [2]

[2] https://www.tensorflow.org/

for example. Moreover in order to enhance the ML component, a hardware neural network should be considered. Another direction in which the framework can be further developed is Agent Security, it is required that the agent has its own identity and it is hard to be corrupted. Natural Language Processing components, Q Learning components, in other words, tATAmI should be a generic paradigm over many of machine learning sub-domains, the behavior of different components might be interesting. More testing and refining in order to make tATAmI a robust MAS framework. As a personal opinion, I think is important to head this framework towards machine learning and add to it less symbolic artificial intelligence elements.

Glossary

ACL Agent Communication Language. 5

ADB Android Debug Bridge. 9

AIDL Android Interface Description Language. 8, 9

AOPL Agent Oriented Programming Language. 4

ART Ahead of Time Compilation. 9

BDI Belief Desire Intentions. 4

DCL Dynamic Class Loading. 9

FIPA Foundation for Intelligent Physical Agents. 5

IPC Inter Process Communication. 11

KQML Knowledge Query and Manipulation Language. 5

MAS Multi Agent System. 4

RMI Remote Method Invokaation. 11

S-CLAIM Smart Computational Language for Autonomous Intelligent and Mobile Agents. 2

tATAmI towards Agent Technology for Ambient Intelligence. 2

Appendices

Dummy Component Example

```
1  public class DummyComponent extends AgentComponent {
2      String thisAgent = null;
3
4      public DummyComponent() {
5          super(AgentComponentName.DUMMY_COMPONENT);
6          registerHandler(AgentEvent.AgentEventType.AGENT_MESSAGE, new
               AgentEvent.AgentEventHandler() {
7              @Override
8              public void handleEvent(AgentEvent event) {
9                  // Do something with the message
10             }
11         });
12     }
13
14     @Override
15     protected String getAgentName() {
16         return super.getAgentName();
17     }
18
19     @Override
20     protected AgentComponent getAgentComponent(AgentComponentName
           name) {
21         return super.getAgentComponent(name);
22     }
23
24     @Override
25     protected void componentInitializer() {
26         super.componentInitializer();
27     }
28
```

```
29      @Override
30      protected void parentChangeNotifier(CompositeAgent oldParent) {
31          super.parentChangeNotifier(oldParent);
32      }
33
34      @Override
35      protected void atAgentStart(AgentEvent event) {
36          super.atAgentStart(event);
37          if (getParent() != null) {
38              thisAgent = getAgentName();
39          }
40
41          mActive = true;
42      }
43
44      @Override
45      protected void atSimulationStart(AgentEvent event) {
46          super.atSimulationStart(event);
47      }
48
49      @Override
50      protected void atAgentStop(AgentEvent event) {
51          super.atAgentStop(event);
52      }
53
54      @Override
55      protected void atAgentResume(AgentEvent event) {
56      }
57  }
```

Agent Scenario File

```
<scen:scenario xmlns:scen="http://www.example.org/scenarioSchema3"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <scen:config mainHost="server␣client" mainPort="9008"
                localHost="127.0.0.1" localPort="9008" />

        <scen:platform>
                <scen:parameter name="name" value="websocket" />
        </scen:platform>

        <scen:initial>
                <scen:container name="Container">
                        <scen:agent>
                                <scen:component name="parametric" />
                                <scen:component name="control" />
                                <scen:component name="mobility" />
                                <scen:component name="messaging" />
                                <scen:component name="testing"
                                        classpath="
                                           StateAgentTestComponent">
                                        <scen:parameter name="other␣
                                           agent" value="AgentA" />
                                </scen:component>
                                <scen:parameter name="loader" value=
                                   "composite" />
                                <scen:parameter name="name" value="
                                   AgentB" />
                        </scen:agent>
                </scen:container>
        </scen:initial>
</scen:scenario>
```

Bibliography

[1] Mariano Alcañiz and Beatriz Rey. New technologies for ambient intelligence. *Ambient Intelligence*, 3, 2005.

[2] Juan Carlos Augusto and Paul McCullagh. Ambient intelligence: Concepts and applications. *Computer Science and Information Systems*, 4(1):1–27, 2007.

[3] Henri Avancini and Analía Amandi. A java framework for multi-agent systems. 2000.

[4] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007.

[5] Mary Berna-Koes, Illah Nourbakhsh, and Katia Sycara. Communication efficiency in multiagent systems. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 3, pages 2129–2134. IEEE, 2004.

[6] Fabian Breg, Shridhar Diwan, Juan E. Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java rmi performance and object model interoperability: Experiments with java/hpc++. *Concurrency - Practice and Experience*, 10(11-13):941–955, 1998.

[7] Jean-Pierre Briot, Thomas Meurisse, and Frédéric Peschanski. Architectural design of component-based agents: A behavior-based approach. In *Programming Multi-Agent Systems*, pages 71–90. Springer, 2006.

[8] I Fette and A Melnikov. Rfc 6455: The websocket protocol. *IETF, December*, 2011.

[9] Munehiro Fukuda. Mass c++ parallel computing library for mulati-agent spatial simulation. Technical report, UW Bothell Computing and Software Systems, 2015.

[10] Jim Huang. Android ipc mechanism, 2012.

[11] Ray Kurzweil. *How to create a mind: The secret of human thought revealed*. Penguin, 2012.

[12] Jiexin Lian, Sol M Shatz, and Xudong He. Component based multi-agent system modeling and analysis: A case study. In *Software Engineering Research and Practice*, pages 183–189, 2007.

[13] Jörg P Müller and Markus Pischel. The agent architecture interrap: Concept and application. 2011.

[14] Cu D Nguyen, Anna Perini, Carole Bernon, Juan Pavón, and John Thangarajah. Testing in multi-agent systems. In *Agent-Oriented Software Engineering X*, pages 180–190. Springer, 2009.

[15] Andrei Olaru, Marius-Tudor Benea, Amal El Fallah Seghrouchnia, and Adina Magda Florea. tatami 1: Towards agent technologies for ami.

[16] Marcel Patzlaff and Erdene-Ochir Tuguldur. Microjiac 2.0-the agent framework for constrained devices and beyond. Technical report, Technical Report TUB-DAI 07/09-01, DAI-Labor, Technische Universität Berlin, 2009.

[17] Anand S Rao, Michael P Georgeff, et al. Bdi agents: From theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[18] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[19] Vanessa Wang, Frank Salim, and Peter Moskovits. *The definitive guide to HTML5 WebSocket*, volume 1. Springer, 2013.