Implementing EcoTruck in Functional Languages

By
Nikolaos Bezirgiannis

Supervisor
Dr I. Sakellariou

Supervisor
Dr K. Margaritis

Bachelor Thesis

Dept. of Applied Informatics,
University of Macedonia

January 25, 2011

Abstract

In this thesis, we present the motivation, design and implementation of a Recycling Management software platform. Potential users of this platform include companies and individuals that wish to dispose their idle paper quantity for recycling. The goal of the application is to enhance the collection of recyclable materials by making it faster, more efficient and environmental-friendly. The recycling trucks construct and follow optimal drive paths, that have the effect of lowering fuel consumption and transportation costs while raising the overall service throughput.

The concept is based on a previous work, entitled "EcoTruck", that took part in the Greek National Software Contest *xiriafia.gr* and won the *1st* prize. By pairing MultiAgent theory with modern concurrent and distributed technologies, we provide an improved re-implementation of that platform.

Contents

Chapter 1

Introduction

1.1 Environmental Terms

The protection and preservation of the natural environment has evolved into a major problem that modern man has to deal with. The main cause of the problem is the rapid population growth, that led us to continuously searching for and using more and more natural resources. This situation takes place in such a high pace, that nature finds it hard to replenish in a timely manner the vast amounts of the natural resources that are consumed.

Two basic methods to treat this problem is the Recycling and Reusing of products.

1.1.1 Recycling

The term Recycling refers to the processing of used materials so as to transform them once again into useful materials that can be reinserted to the production cycle. It is the third component of the well-known 3R hierarchy "Reduce, Reuse, Recycle".

Recycling aims to:

- Help to reduce pollution caused by waste.
- Require much less energy that we would otherwise need to build fresh products from raw materials.

Recycling can:

- Decrease the financial expenditure on building products. The use of raw materials in the production normally costs much more than using recycled materials.
- Preserve natural resources for future generations.

Recycling is an ancient practice for the preservation of the natural environment. There are even records in writings of Plato around 400 B.C.. However, this method is finding application only in the last 40 years. The reason for this is the huge industrial growth that we meet in our days.

The industry trying to satisfy the needs of the modern man has increased the offer of their products binding however a big part of the natural production factors. The fear of depletion of these factors is the main reason for the widespread adoption of recycling practices.

Greece, however, does not follow this global trending. According to statistical evidence, our country is placed in the last position of the European ranking on environmental and recycling issues. This is due to the fact that recycling can not be thought as a simple process. It entails proper environmental culture, public awareness and financial investments for acquiring special equipment and using latest technologies.

There is a great variety of recyclable materials. This includes materials with widespread use such as metal, glass, paper and plastic. In practice, only a few of them are actually recycled because in many cases there is no significant benefit from recycling them. This thesis is concerned exclusively with paper recycling, because the material exhibits remarkable gain by recycling it and there are still margins for application.

Paper Recycling

Because of the huge amount of paper used daily, important ecosystems are threatened by its consumption. Forests that required over 1000 years to grow can be vanished in just 12 minutes. The excessive logging does not only create a sad image; it ruins the fertility of the ground and has a negative impact on the water cycle.

The environmental damage does not stop with the logging of the trees. To turn the paper into something useful, the factories process the logs heavily. Often, this process results to considerable pollution of the aquatic ecosystems and also of the atmospheric air.

Thus, if we lessen the unnecessary paper consumption and improve our recycling programs we can ease the pressure on the forests, which can allow the environmentally correct management of the forest and will prevent the devastation of our ecosystems.

In Greece, paper consumption has surpassed the 800,000 tons per year. Of these 800,000 tons, about 300,000 tons of paper are thrown each year, which their production costs translates into:

- 12 million square metres of forest
- 100 million cubic water (equal to the consumption of Attica region for 100 days)
- 1.5 to 2 billion kilowatts per hour (equal to the energy needed to power 1 million households for a period of 4 months)

Recent studies carried out have shown that recycling of 1 ton of paper will:

- Prevent the logging of 17 trees
- Save 30 cubic metres of water
- Consume 2,700 kilowatts less for the production of paper products
- Reduce by 73% of the atmospheric pollution
- Spare 320 litres of gasoline

Note: The numbers [11] above are used as information data for the software application that is presented here.

1.1.2 Reusing

Reusing is the process of exchanging useful products with the goal of saving up resources, money and time. In contrast with recycling these products do not get any processing.

This practice is not very well-known compared to recycling, but can contribute enough to the overall performance of the goal, as the difficult part of reprocessing is removed.

Paper Reusing

Specifically for paper, nature puts some constraints on the viability of the reusing methods. With further and further reprocessing, the paper material wears out. Because of its reduced quality, the recycled paper is found in low-value products (e.g. cartons, wrappings). Also, the paper products are specially built in such a way that are intended for specific customers and include logos or symbols making these imprinted products hard to trade.

1.2 Problem

We will examine only recycling and not the part of reusing, not because we find it unimportant, but because of the small application and attention it receives in Greece.

We can track many deficiencies in the management of the procedure of paper recycling by municipals all over the country. Until now, the only method used by authorities for recycling paper carton boxes involves telephone notifications. Generally, this method includes the following steps:

1. Big companies and shopping centres inform by telephone the service qualified for this task

2. After much communication a municipal truck picks up the paper quantity

As we can observe the service mechanism applied is not well-thought-out and in many situations can be cumbersome.

Particularly, some of the drawbacks are:

- The notification process is not automated.

  The truck must continuously be in touch with the municipal office or the companies-clients for clarifications and further details or any changes that have to be made. This should take place in a timely fashion, nearly real-time. Also we could speed up the communication if every user of the system handles the connections with other users in parallel.

  But a telephone communication can be neither constantly online nor concurrent. The truck can handle only one client at a time. Consequently the truck will lag in serving companies and fail in a large number of them.

- The truck does not follow a certain plan.

  As a result its actions will not be coordinated. Efficient scheduling of these actions is arbitrary and does not follow any pattern.

- There is no cooperation between the trucks.

  The trucks operate autonomously. The service office does not allow "team-play" between the trucks so as to achieve their common goals.

- The trucks operate inefficiently.

  There is no search for the shortest path to a client, because the truck does not know about all available routes. The very important traffic factor is not counted in.

- Recycling bins are not a "silver bullet".

  Of course the municipals have taken care of putting recycling bins in various places inside the city with the goal of speeding up the recycling process and making it more widely used. However the problem remains mainly for two reasons:

  1. The companies have a vast amount of paper to recycle, which does not normally fit into bins.
  2. The bins are bulky.

     The bins reduce the space for parking lots with any of the consequences that has. Also bins don't scale up. Just adding more bins does not solve the problem and it could make it worse.

- There is no public awareness on environmental issues and mainly on the recycling methods.

  People think that recycling is a difficult and time-consuming procedure, because it demands collecting and sorting the materials in the house and finding a nearby bin to throw them in.

1.3 Motivation

This thesis tries to present an alternative method of managing paper material, to implement it in an efficient application and to analyze the results and its capabilities.

1.3.1 Room for improvement

- Truck will follow a plan.

  Each truck will construct his own plan in order to sort his intended actions. The plan will be dynamic; that means the truck can re-order its steps in real-time. The long term goal is the overall improvement of the entire system.

- Timetables for collecting and transferring are constructed.

  Given the traffic factor inside the city roads, the trucks could create and update live their timetables in order to select each and every time the shortest path to the client. In this way, waiting time to collect the quantity can decrease dramatically and there is a speedup in client dispatching.

- The heavy computing parts become automated.

  The planning and scheduling are problem hard enough to be solved by a human in a timely manner and in most cases these solutions are inefficient.

  There is no need anymore for human intervention for the communication, the management, the construction of the plans and the scheduling of the trucks. All these are being taken care of by the application.

  The result of this will be faster servicing and throughput increase, because the system can handle more clients and new trucks will be able to participate in the overall process.

- The service centre becomes unnecessary.

  Communication will take place *directly* from a user to another user of the system without the need for a "middleman" (i.e. the municipal office). This has the negative effect of increasing the messages exchanged between the users, but in this way the system can be more fault-tolerant. A possible failure in the service centre will not bring down the whole system and moreover the running and queued jobs will still execute.

1.3.2 Technological advance

The recent rapid progress of computers and networks has created opportunities for new methods of work and communication. Furthermore their universal dominance have made the use of these new technologies more affordable.

This is the main reason which turns the idea into an actual application. Nowadays, it is easy enough to install and operate a computer system in a truck environment. Also the cost of a wireless connection has reached such a low level that makes the whole idea feasible.

1.3.3 Approach

The multiagent approach seems to be ideal for attacking the recycling management problem. In a nutshell, clients as well as trucks take the form of agents that communicate continuously. The coexistence and the coordination of agents for achieving their common goals creates a multiagent system. The theory behind multiagents is discussed later on.

Our multiagent system is implemented in Erlang, a specifically designed language for describing distributed systems. The features and the components of the Erlang language, as well as the Erlang OTP platform, which facilitates in writing such complex systems, are described in detail in the 3rd chapter.

Chapter 2

Multi-agent Systems

2.1 Intelligent Agents

2.1.1 What is an Agent

There exist many different interpretations of the term "agent", depending on which context it is used in. The general meaning of the word encompasses a behaviour of action; an agent is an entity (it could as well be a person), who has been given the authority to act on behalf of another or provide a service.

In Artificial Intelligence, the concept of the *agent* plays a central role. For this reason, there is an ongoing fight on which interpretation of the term better describes its purpose. The truth is, that all these explanations can be both wrong and right at the same time; it depends on what capabilities you want to give to your agent. We aspire to the simple notion of the Physical Agent: an entity which percepts its environment through sensors and acts upon that environment through actuators.

We should establish the characteristics we think are important to better conceptualize the essence of the intelligent agent:

Autonomy

Normally, what we do is to build an agent and assign it a series of tasks that we want it to accomplish. However, we do not instruct it on what steps (actions) it should take to fulfill our goals. In this sense, the agent is autonomous, for the reason that it thinks and decides how it must act, so as to better carry out our assignments.

Rationality

When the agent thinks and acts in such a way so as to better achieve its goals, we say that it is being rational.

Proactivity

The agent should be proactive, that means, in certain cases it should in advance take actions to encounter possible future situations happening. It should be able to make "the first move" to come closer to its goals and to bring itself in a better position in the future.

Reactivity

Some environments are in constant flux and have a very dynamic nature. Agents that operate in these environments should behave the same; they should modify their objectives in a similar fashion to the changes happening in the environment. These changes could possibly mark a goal as no more valid or even unattainable. The agent should realize this and dynamically modify its future tasks.

Social Ability

The agents should exhibit the ability of interacting with other thinking entities (being this a human person or another agent) so as to come closer to fulfilling their own goals.

2.1.2 Agent Environments

What follows is a short categorization [13] of the properties of environments.

Fully observable vs partially observable

If the sensors of an agent can fully capture the state of the environment at any particular time, we say that the environment is fully observable. On the other hand, in a partially observable environment, some of its attributes cannot be noticed and the agent should keep an internal state containing previous attributes that it has sensed.

Deterministic vs stochastic

If the next state of the environment can be fully determined just by applying the action onto the current state, then the environment is said to be deterministic; in any other case, where there are "hidden" factors that affect the environment then we should call it stochastic.

Episodic vs sequential

In an episodic environment, the agent picks its next action based not on previous actions or older states but just on the current episode. In a sequential environment, the choice of the next action is influenced from former actions and experiences.

Static vs dynamic

If the environment does not change during the time the agent takes to choose its next action, then it is called static. Otherwise, if time can alter the environment then we say it is dynamic.

Discrete vs continuous

The environment that has a finite number of states is a discrete environment. However, if the state of the environment has attributes that are expressed in continuous variables, then the environment is in that way called continuous.

Single agent vs multiagent

If an agent only perceives the environmental objects and acts upon them, then this environment is a Single-Agent system. On the other hand, if it interacts with other agents by competing with them or cooperating, then we have a Multi-Agent system.

2.2 MultiAgent Systems

A MultiAgent System (MAS), as we said earlier, is composed of a number of different agents that continuously interact with each other, possibly through messages. MASs are basically trying to attack particular problems that would otherwise by very difficult or even impossible to solve.

MultiAgent Systems should impose the following characteristics [16] :

- Every agent has only a local view of its environment and normally communicates with other agents to obtain a "bigger picture of the world".
- There exists no central administrative control system.
- Information is scattered among agents and data are decentralized.
- Computation, and thus communication, is asynchronous.
- Key factor in all MASs is *time*. The communication between the agents must be done in real-time and the overall performance of the system is time-sensitive.

2.2.1 Understanding

Agents sense their environment, then try to understand its surroundings and environmental objects and finally act upon them. An agent can give a somewhat different meaning to the same exact object compared to another agent. And that is all fine, until we go to implement a MAS.

Agents living inside a MAS interact wit each other by exchanging their views and beliefs on objects that surround them. We have to be sure that agents give the correct and precise meaning in every part of their acquired knowledge. This is achieved by defining ontologies.

An *ontology* is a formal set of terms that describe the knowledge about a particular domain. Every new term is defined by writing down a structural composition of older terms. In this way, we introduce new entities by abstracting over past definitions. We can be sure that the agents who share the same ontology, will interpret these newly defined terms in an identical manner.

Some ontology languages to write down our ontology in, are the Web Ontology Language (OWL) and the Knowledge Interchange Format (KIF). Another way to express our ontology is to define custom XML documents, although that would most likely be a very strenuous effort, because it requires a low-level coding style and we lose the ability of reusing previously defined ontological terms.

2.2.2 Communication

Now that we have concluded on how the agents will uniformly express their knowledge, we must take a look on how the communication between them will be realized.

We can model the agent communication infrastructure around the well-established *Speech Act Theory*. This theory, developed by the philosopher John Austin, can help us analyze utterances from the perspective of their function, rather than their form [14] . Austin described three distinct types of acts based upon their functions: locutionary, illocutionary and perlocutionary acts.

In this sense, words are not just "inanimate" objects, but their utterance can designate action. In agent communication we can define some performative verbs, a list of common words that agents will exchange through messages to describe their actions.

As we said before, agents that form a MAS interact with each other using messages. The messages must be written in a mutually understood language, otherwise an agent would not be able to interpret the message's contents. There have been proposed several

Agent Communication Languages (ACL), though only two of them are really well-known: the KQML and the FIPA-ACL.

What follows is an example of a `REQUEST` performative, written in the FIPA-ACL language. Its syntax resembles Lisp, because it uses S-expressions to represent data:

```
(request
    :sender (:name customer1@debianlap:8080)
    :receiver (:name df@debianlap:8000)
    :ontology recycling-ontology
    :language FIPA-SL
    :protocol inform
    :content
        (update-position (:lat 40.625606) (:lng 22.960431)))
```

2.2.3 Cooperation

The MAS agents exchange messages with each other, not because they are in a "mood for chatting", but with the goal to cooperatively carry out some work. We should make clear that, in a cooperative MultiAgent environment, the agents of the system do not necessarily have to share the same goal. Problems are decomposed to smaller subproblems and then each subproblem is delegated to an agent to solve it. The subsolutions are then synthesized to one whole solution. This activity is called *task sharing*.

Task sharing can be a straightforward process, if the agents given exhibit quite identical capabilities. Then, we can assign tasks to them in no particular order. However, in most cases, the agents are very different to each other, so we have to specify for each task at hand, an agent we believe is appropriate to accomplish it.

This task allocation can be achieved by a higher-level task sharing protocol, called *Contract Net Protocol* (CNP). This protocol defines an interaction pattern where an initiator agent announces some task it wants to get done, the other agents respond with an offer (most likely a cost or time to fulfill the task) and finally the initiator delegates the task to the agent(s) with the best offer.

What follows is a graphical representation of the CNP and after that a detailed description of the protocol:

The *Initiator* agent sends Call-For-Proposal (CFP) messages to the *participants* of this negotiations. The participants respond with either a PROPOSE, including the bidding offer, or a REFUSE, stating possibly the reason for the refusal.

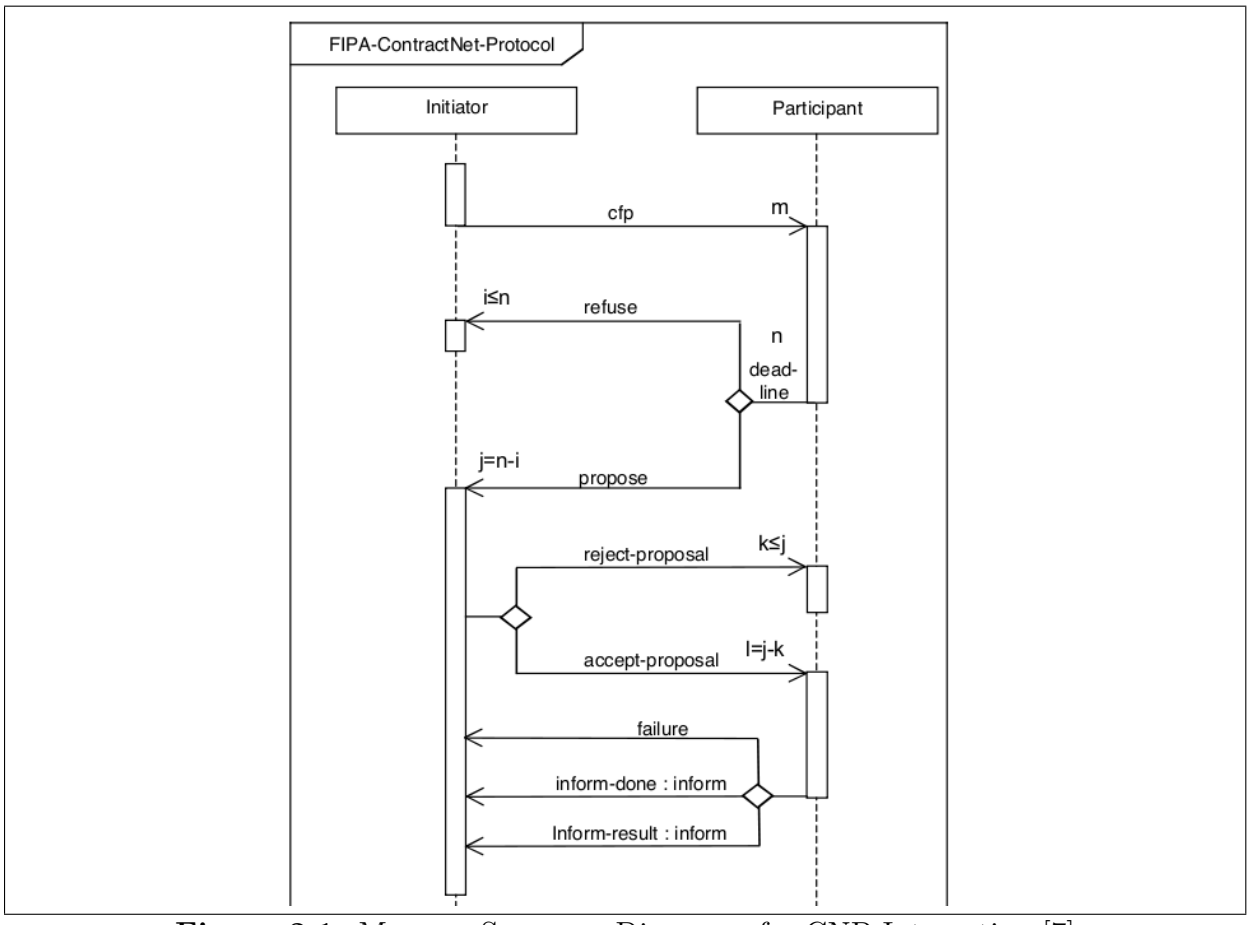


Figure 2.1: Message Sequence Diagram of a CNP Interaction [7]

Then, the initiator *ranks* the bids and awards the agent(s) with the *best bid*, an ACCEPT_PROPOSAL. The rest of agents will receive a REJECT_PROPOSAL performative.

The accepted agents are committed to carry out the job that were delegated to them by the initiator. When they finish they have to announce to the initiator that they have completed the task sending back an INFORM message with any necessary extra information. If the accepted participants fail to perform the task, they could issue a FAILURE to the initiator and then the initiator will decide, how it should handle this problematic situation.

2.2.4 Coordination

Besides committing to a specific task, a MAS agent should also commit to the joint goal of the MultiAgent system. That is, it should give its best effort to fulfill its assigned task,

without dishonoring the overall aim of the MAS.

Committed tasks can possibly conflict with each other and in a situation like this, there should be a resolution of the conflict either by negotiating or by coordinating the individual tasks.

2.3 The Foundation for Intelligent, Physical Agents

In 1996, a non-profit organization was conducted in Switzerland, to specify a set of standards for the communication and interaction of agents and to define an implementation of such Agent Platforms. It took the acronym FIPA, which translates to "The Foundation for Intelligent, Physical Agents". From time to time, large tech companies were members of this organization. The FIPA was later on accepted to be an official IEEE standards committee.

The FIPA's Mission as stated by the organization itself is:
"The promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings."

Several agent platforms have adopted the FIPA standard and are said to be fully FIPA-compliant. One of the protocols that the standard brings forward is the **Contract-Net** protocol. Messages are written in **FIPA-ACL**, which was illustrated earlier.

2.4 Conclusion

This chapter tries to be a short introductory summary to the theory behind multi agents; we only saw a glimpse of the otherwise wide research domain of MultiAgent Systems. We gave a simple definition of the Agent entity and how the agents understand, communicate and work together to solve bigger problems.

Chapter 3

Erlang Programming

3.1 Introduction

There are literally thousands of programming languages out there and probably there will be more to come. Some of them are very powerful or run amazingly fast; others are more expressive or easier to understand. But few of them really enjoy high popularity and most programmers are familiar with only a handful of them.

But why use Erlang? Why we need yet another language?

While it is possible given any capable enough language to design complex systems or solve hard mathematical problems, it is not certain if this is going to be an easy process. Its all coming down to using the "right tool for the job".

And Erlang can really shine if you want to:

- Write a program that can easily scale.
- Build a fault-tolerant application that can be upgraded on-the-fly.
- Build a mission critical server product.
- Design a soft real-time system.
- Use a language that has been heavily tested in industrial products.
- Write in a functional style.

3.1.1 History

The name "Erlang" can refer to either the Danish mathematician and engineer Agner Krarup Erlang or the abbreviation **Er**iccson **Lang**uage.

It all began when a small team of programmers in the Ericcson Labs was investigating a suitable language to power their telecom products. After a lot of consideration they decided to ditch the proprietary languages used by Ericcson at that time and build a new language designed with concurrency and fault tolerance in mind.

Erlang is influenced by functional languages such as ML and Miranda, concurrent ones such ADA and Modula. It is borrowing its syntax and many other things from Prolog. Even the first virtual machine targeted for Erlang was Prolog-based.

Years passed and Erlang started to find more and more practical use inside the Ericsson company. At the same time wonderful new features were being added to the language.

In 1991 Mike Williams decided to rewrite the virtual machine in C. This gave Erlang the speed it needed to implement soft real-time systems.

In 1996 Erlang hit a major milestone when the "Open Telecom Platform" framework (OTP) was released, a set of great tools and libraries to build robust and fault-tolerant distributed systems.

Finally, after supporting hundreds of commercial applications and being tested on real-world scenarios, the language was released as open-source in 1998, using a similar license to MPL, the Mozilla Public License.

Since then, Erlang is gaining very much in popularity. The community is ever-growing and its adoption is rapidly expanding mainly because of the recent need for building web-oriented distributed systems.

3.1.2 Describing the Language

When people talk about programming languages or compare them (Warning: this can lead to endless flamewars!), they try to fit the languages around predefined categories and programming paradigms, even if sometimes this is not possible.

Erlang is described by many as multi-paradigmatic (i.e. you can express the same program solution in different ways), although the Erlang community and the recent addition of the OTP framework encourages writing with a specific programming style. In this way it is easy to find certain distinct characteristics of the language:

Functional

It is a purely functional programming language at the core but impure at a broader sense, because it allows to incorporate computational effects to be run inside functions; that is , by design, it restricts the places you can run your side-effects so that pure code does not mix up with impure functional code.

It is eagerly evaluated, offers single-assignment variables and immutable data structures. It has no-shared state and relies heavily on pattern-matching; a feature mostly used in functional programming. We will talk about it later on.

Dynamic typing

You don't have to add any type annotations to the language constructs, because the type checking takes place at runtime, raising runtime exceptions when something goes bad. Most of the time, writing an Erlang program feels like "scripting". However this gained flexibility comes with the cost of type safety and sometimes could result in late uncaught bugs at production stage.

To address these problems, a pair of tools have emerged: TypEr and Dialyzer. TypEr is a type checker for type annotated Erlang programs but can also work as a type annotator, inferring types for erlang without any type specifications. Dialyzer is a static analysis tool that is used by many developers as an auxiliary tool for catching common programming errors and software discrepancies.

Concurrency-oriented

Erlang is built with concurrency in mind. The *process* is the most fundamental concept of the language. An erlang process is somehow the smallest execution unit that you model around your program's logic.

Unlike system processes and OS threads, Erlang processes are extremely lightweight and memory efficient. Reports have shown that you can spawn millions of processes simultaneously while the system stays fully responsive.

The communication between processes is realized through message passing, a share-nothing philosophy, where each process sends and receives small messages that contain valid erlang data structures.

VM

Source code written in Erlang is translated into intermediate files, called .BEAM

files, which essentially is bytecode that gets executed by the Virtual Machine. This allows code to be distributed to any platform given an Erlang VM exists for it.

The VM includes a garbage collector and a scheduler for each OS thread. All processes live inside the VM and get scheduled accordingly.

Distributed

A distributed Erlang system is composed of a number of different Erlang nodes running in one or more computers. Each node is a unique instance of the VM.

The extra semantics added to the language can alleviate the development of such distributed applications. In Erlang, specifying or referring to either a local or a remote node makes no difference.

3.1.3 Special Features

Here are some of Erlang's distinct features that you will not probably find in other languages:

Binaries and the bit syntax

The Binary data type is a special Erlang data structure, a built-in construct of the language. It is a sequence of bytes (unsigned 8-bit bytes by default), that are used to store and manipulate large amounts of raw data.

Just to make things even easier, the language offers the bit syntax for manipulating binaries in a high-level manner. The bit syntax is just like pattern-matching with the difference that where you would normally work on typed values, now you pattern-match on "untyped memory".

Common use cases include:

- Designing, building and parsing of network protocols.

  "..a bit program can read like the high-level specification of a protocol rather than its low-level (and opaque) implementation." [6]

- Serializing loads of data for later storing or transferring them.
- Swapping high-level data structures (e.g. strings and tuples) with binaries in order to boost performance and gain in space efficiency.

Hot code loading

In large industrial systems or live production servers, downtime is absolutely crucial and must be minimized. Often, special situations arise and the system software must be upgraded (or downgraded):

- When the system must be patched against a bug.
- When a new application feature rolls out.
- When you need to prototype through trial&error.

In most conventional languages and runtime systems you must bring down the whole or a part of a live system to upgrade its software. This inserts important downtime and a possible loss of data.

By contrast, software written in Erlang can be upgraded during runtime with a technology called *hot code loading* or *hot code swapping*. In most occasions, hot code loading can be as easy as a normal function call.

The OTP framework offers some great tools to safely upgrade a running system. And if things go badly, you can of course roll back to a previous version of the code; again by not taking down the system.

Fault-tolerance

Joe Armstrong, one of the creators of the Erlang language, famously said in one of his theses "Do not program defensively. Let it crash". The *let-it-crash* philosophy came to be one of the greatest aspects of the language. Although by definition it comes to contradiction with fault-tolerance, in essence they mean very much the same.

When you use defensive programming in your applications, many times you end up with "polluting your code with needless guards trying to keep track of the wreckage" (see Java exceptions). With Erlang you write down your code failure-free; you deal about failures only later on. This results in a cleaner separation of the program's logic and the error handling code.

When an error occurs in a process, it is common to crash the process and let the error propagate on "neighbouring processes" (i.e. processes that depend on each other). The rest of the system that don't depend on the crashed code will remain unaffected and will still run. The point is that the part of the system that went down, hopefully, will be restarted by a higher-hierarchy process (called the supervisor). This hierarchy

is defined in supervision trees, a core idea of Erlang/OTP discussed below in further detail.

Soft real-time applications

If high availability and throughput are demanding properties of a system, you must ensure that your application can gracefully handle high loads on peak times.

Erlang is a perfect fit for such an application. This can be attributed to the fact that each process runs in total isolation. It has its own memory and its own garbage collector. In this way, an Erlang system does not suffer from garbage collection pauses.

It is not surprising that many servers and proxies are written entirely in Erlang. It comes so natural to spawn a separate process for each client connection; a thing that you wouldn't dare to do in any other language.

The OTP framework

Lessons learned building thousands of concurrent and distributed applications by Ericsson in Erlang through the years, have led to the creation of the OTP framework, an extension to the traditional Erlang. The OTP stands for Open Telecom Platform and unlike its name suggests, it is not limited to telecommunication applications. Practically it is a middleware for Erlang development.

The programmers can use a common ground to facilitate development time and effort by constructing behaviours, a formalization of design patterns. Furthermore, code written with OTP is more portable and maintainable.The joining of individual applications becomes such an easy process, for the reason that OTP applications share a similar programming logic.

Besides this collection of libraries, the OTP framework includes a set of other powerful tools, such as a distributed real-time database called Mnesia, complete web and ftp servers and a CORBA Object Request Broker.

Language interoperability

Often, you have to design large GUI applications or do heavy number crunching in a distributed fashion. Other times you have to add concurrency levels on top of an already built web app or an old C codebase. Instead of rewriting the whole thing (with possible loss in performance, because Erlang is "not good with numbers") in Erlang, you can use the interfacing capabilities of the language.

While most other languages provide a Foreign Function Interface (FFI) library to link with code written in other languages, Erlang follows a different approach. It extends the message passing paradigm, through ports and linked drivers, to interact with foreign code. To the programmer, it feels like surrounding the foreign code with a thin layer of an Erlang process; communicating with this process remains just the same.

How about talking to an Erlang node from a non-erlang-powered system? Erlang/OTP again has a solution for this, offering rich C and Java libraries so that you can masquerade your program to look like a usual Erlang node. The communication between such nodes works flawlessly.

Multicore-ready

The recent increase in multicore processors' availability is pushing popular languages to add the necessary ingredients for supporting Symmetric Multi Processing (SMP). While many have succeeded to utilize the advantages of the SMP technology, others have failed to come up with an efficient parallelism design or deliver a working implementation (see the Python GIL). But Erlang had a solid concurrency model for years. The only thing left was to translate it to SMP.

The SMP support for the language was added later on and became stable in the R11B release (in 2006). Briefly, the implementation involves spawning a separate process scheduler for every core/processor in the system. The schedulers pick up their next jobs from one common run queue, which is a shared data structure protected with locks.

In most cases, the developer can enjoy the SMP performance gain without making any changes to the program. Applications in Erlang are written with a lot of individual processes that allow fine-grained parallelism.

3.1.4 Case Studies

Many Ericsson network hardware products shipped are powered by Erlang and lately the language has been used by web companies as a mean to achieve scalability, high-availability and robustness in their web services.

Ericsson Switches

Erlang has been used in many successful network products such as broadband, GPRS

and ATM switching solutions. However, after Java becoming the new trend in the late 90s, Ericsson banned Erlang for new projects in favor of more mainstream languages.

Facebook Chat

The entire Facebook instant messaging system is powered by ejabberd. Ejabberd is an Erlang-written, open-source, extensible application server which speaks the Extensible Messaging and Presence Protocol (XMPP, formerly Jabber). Right now, it is the only solid solution for setting up a robust chat service and is widely-used by many sites and companies for private internal communication.

Amazon SimpleDB

"Amazon SimpleDB is a highly available, scalable, and flexible non-relational data store that offloads the work of database administration. Developers simply store and query data items via web services requests, and Amazon SimpleDB does the rest." *(from the Amazon site)*

It is build around CouchDB, a new NoSQL Erlang-powered database. Instead of storing rows of data inside tables, CouchDB saves JSON documents and serves them via a RESTful interface. Also, it offers a master/slave replication scheme that increases throughput and fault-tolerance. In case of a failure on the master server, a slave instance can take charge.

3.2 The Basics

This is a short introduction to the language syntax with some examples to help you write a simple program in Erlang. The language is said to be rather small in size, providing only the essential abstractions for modeling a functional application.

3.2.1 Datatypes

Integers

In Erlang integers denote arbitrarily large whole numbers with bignum conversion; that is when an integer cannot fit in a word it is converted to its bignum representation, which uses arbitrary number of words for storing it.

```
1
19
1000000000000000005
```

An integer can be positive or negative and is expressed in base 10 by default. The notation Base#Value can be used to write integers in other bases.

```
1> 2#101010.
42
2> 16#FFFF.
65535
```

Characters

Characters are just ASCII integer values and can be written with the $Character syntax.

```
3> $A.
65
4> $\n.
10
```

Floats

Floats are Erlang's real numbers. They are 64-bit double-precision floating point numbers.

```
3.14159 -5.4E-9
```

Arithmetic Operators

There exists the classical operations on integers and floats: addition, subtraction, multiplication, and division:

```
5> 7+5.
12
6> 7-5.
2
7> 7*5.
35.
8> 7/5.
1.4
9> 7 div 5.
1
10> 7 rem 5.
2
```

Atoms

Atoms are symbolic, constant literals that are used to make code more readable. The data type is inspired by Prolog and is analogous to a Lisp symbol. For people not familiar with these languages, it can be thought of as a *huge enum*.

They must be started with a lowercase letter and can be followed by any alphanumeric character or the symbols _ and :

```
hello
i_am_not_a_string
root@localhost
nikos@bezirg.net
```

If you want to start it with an uppercase letter or include other symbols, you can enclose the atom inside single quotes as follows:

```
'Hello World!!'
'uom@195.251.209.3'
```

The only operation on atoms is just the comparison, albeit a very efficient one. The implementation of atoms includes a big table inside the VM that stores all atoms at creation time. Once an atom has been created, it stays until the system stops; atoms are not garbage-collected. Another characteristic is that atoms remain in the object code, so debugging becomes easier.

Booleans

The atoms `true` and `false` are used as the boolean values. They are the result of comparisons:

```
11> 0==1.
false
12> 1<100.
true
13> a>z.
false
```

These are the logical operators:

```
and andalso or orelse xor not
```

Lists

Lists are the classical data type found in all functional languages. Essentially it is a collection of elements that can grow or shrink in size. An example:

```
[5,1,$d,mpla,true]
```

The empty list is denoted with `[]`.

Unlike the Haskell language, Erlang list elements do not need to be of the same type.

List processing takes place at the left part of the list. With the *cons* operator (|), We can split a list into its first element, the *head*, and the rest of the elements, its *tail*. The tail of the list must be itself a list. In the same way, we can add an element into the beginning of a list.

```
List = [Element | List] or []


[a,b,c] <=> [a | [b,c]] <=> [a | [b | [c]]] <=> [a | [b | [ c | []]]]
```

Strings

Strings is the same as a list of characters. Erlang does not treat them differently. The syntax is:

```
14> "ABC" == [65,66,67]
true
```

Atoms vs Strings

It is possible to use strings in place of atoms. However, strings are not very memory-efficient and string comparison is much slower than comparing atoms.

In the other way around, using atoms for strings could also work, but in practice, because of the atoms not being garbage-collected, there is a limitation on the number of atoms that can be created. So, the use of auto-generated atoms (via `list_to_atom` operation) should be avoided.

Tuples

Tuples are just like lists but with a different implementation and no cons operator. To manipulate tuples you use some provided built-in functions.

Compared to lists, tuples are a little more efficient in memory and also in access times. But when you should use the one or the other type? Generally this rule exists: Use lists when you have a variable number of items and use tuples for a fixed number of items. Also, handling huge lists tend to be slow.

Records

When you have to deal with tuples - with size let's say over 10 elements - the construction of the tuple and the processing of elements becomes cumbersome. For this purpose there is the record data type. Records are "named tuples" in the sense of giving each index of the tuple a unique name. The record fields are accessed by name, just like a C structure.

On the inside, records are built using tuples. During compilation phase, the preprocessor translates them to plain tuple structures. The speed and memory efficiency are identical.

Defining a record

```
-record(name, {field_1 [ = default_1 ],
               field_2 [ = default_2 ],
               ... ,
               field_n [ = default_n ]
  }).

-record(book, {title, author, year=none}).
```

Instantiating it

```
#book{title="An Introduction to MultiAgent Systems",
      author="Michael Wooldridge",
      year=2009}
```

Modifying it

```
Book1#book{year=2007}
```

Accessing its elements

```
Book1#book.title
```

Variables

Combining all of the above datatypes, we can create complex data structures and store them in variables.

Variables must begin with an uppercase letter and be followed by alphanumeric letters, integer and underscores.

```
One = 1.
Two = One + One.
Author = "Nikos".
Thesis = #book{title="some_title", author=Author}.
```

Be careful of the *single-assignment* property of variables. Something like this,

```
X = X+1
```

is not legal in Erlang. Once something is defined, it cannot be redefined. The above statement is not side-effect-free and issues a *destructive update* on the variable, a technique that most common languages rely on. Erlang uses the mathematical notion of a variable, that is a statement about a fact.

3.2.2 Pattern Matching

During an assignment operation, the right-hand side is evaluated and then matched with the left-hand side. If the pattern match succeeds, any free variables of the left-hand side

are bound to the computed values. If it fails to match, an error is raised. An example can clarify things:

```
[X | Xs] = [1,2,3,4] ... X = 1 , Xs = [2,3,4]
{hello, Name} = {hello, nikos} ... Name = nikos
{hello, Name} = {cu, nikos} ... will fail
```

Pattern matching is a feature mostly found in declarative languages.

3.2.3 Functions

In a functional language, functions are the building blocks to construct your program's logic. In Erlang, functions occupy an even more central position, because they have a double role; that of an action and a process. We will check how processes work in a later section. For now, they can be seen as classic C functions.

They have a name and a list of zero or more parameters, enclosed in parentheses.

```
hello(Name) ->
    io:format("Hello "),
    io:format(Name ++ "\n").
```

The statements in the body of the function will execute sequentially and the last expression evaluated, will be the return value of the function. We do not have to write down `return`, like other languages, because in Erlang, every function has to return something. There is no void type. If we want to return a nil value, conventionally we use the `ok` atom.

```
15> hello("World").
Hello World
ok
```

Here, the last statement is a print operation that returns ok, so consequently our function also returns ok.

Most of the times, you write down functions using pattern-matching, that is you write a series of clauses that the system will try to match step by step. If a particular match succeeds, then the body of the matched clause will run. It is common to have a catch-all clause, that always succeeds; otherwise, an error will occur if all other clauses fail to match. An example:

```
inc(X,0) ->
   % that is meaningless, increase sth by 0?
   X.

inc(X,N) ->
    % increase x by n
    X+N.
```

The length of the parameter list is the function's arity. In Erlang functions can exist with the same name but different arity.

```
inc(X) ->
  % increase x by 1
  inc(X,1).
```

This comes handy when building an API, because we can have optional parameters or pass default values to functions.

3.2.4 Modules

In Erlang, the module is the basic compilation unit. Each module contains functions, which as a group form an API. Modules are written in files with an .erl suffix. They contain some built-in attributes, such as compiler directives, a list of exported functions, a version number for hot code swapping, and some user-provided attributes such as author or date.

```
-module(test).
-author(nikos).
-export([inc/1]).
...
...
...
```

3.3 Sequential Erlang

Functional programming relies heavily on *recursion*. Programmers familiar with imperative languages will feel a little out of place at the beginning, because basic imperative constructs, such as *while* and *for* are missing from the language. Instead control flow is done through recursive function calls.

3.3.1 Conditional Constructs

Erlang provides the `case` and `if` constructs for conditional evaluation.

Case

We have seen already pattern-matching done in the assignment and the function level. Now, again, we use the same technique slightly in a different way.

Instead of introducing any new variables or defining functions, we match against an expression and on the success of it, we execute some code. In a way, we map clauses to series of statements, essentially introducing control flow:

```
case Expression of
    Pattern1 -> expression1, expression2, ..;
    Pattern2 -> expression1, expression2, ..;
    ..
    _ -> expression1, expression2 ..
end
```

The *case* forms resemble a *switch* in the C-like languages.

The _ pattern matches any value, so it will always succeed, much like a `default` label in C/Java.

If

The `if` construct is similar to the `cond` construct found in Common Lisp. The syntax is:

```
if
    Guard1 -> expression1, expression2, ..;
    Guard2 -> expression1, expression2, ..;
    ..
    true -> expression1, expression2 ..
end
```

Guards must be expressions that evaluate to a boolean value. From top to bottom, the if construct tries to find a truthful predicate and execute its statements.

3.3.2 Guards

Guards are a useful extension to functions, that can make code much clearer and more readable. In many situations, you do not only want to (pattern-) match against structures

of data, but also on particular values of them. With guards you can create individual clauses for instructed values of the formal parameters of a function. Following a previous example:

```
inc(X, N) when N < 0 ->
   error.
```

Here, it is not meaningful to increase a value by a negative number, so in this situation the user returns a custom `error` atom to signal failure. Later on, we can see how we can raise and handle true built-in exceptions.

Another thing we can do with guards is elementary type checking. Because the language lacks a strong type system, we can check for types calling some Built-In Functions (BIFs) as guards.

```
inc(X,N) when is_list(N) ->
  % the user entered a string (list of chars) instead of a number
  % try to casted to an integer
  Int_N = list_to_integer(N),
  X+Int_N.
```

Be careful, only BIFs can be used as guards. This is mainly for two reasons. First, BIFs are written in C for speed, and second, user-implemented functions could introduce side-effects before the function body is even executed. This can be contrary to the pure nature of functional programming.

3.3.3 Tail-recursion

As we said earlier, recursion is a very widely used technique to express iterative computations in a functional way. Also, it is common through recursion to divide a problem into smaller subproblems (divide&conquer).

The canonical way of writing such recursive functions (often by a straightforward translation from their mathematical forms) sometimes can be very inefficient. Take a look at the classical summation function definition:

```
sum([])           -> 0;
sum([Head | Tail]) -> Head + sum(Tail).
```

The direct recursion used here, although it seems normal, when used on very large lists, will grow a lot in memory.

Instead, using an accumulator:

```
sum_acc([], Sum) -> Sum;
sum_acc([Head | Tail], Sum) -> sum_acc(Tail, Head+Sum).

sum(List) -> sum_acc(List,0).
```

we can sum large lists in a memory-efficient manner. This is achieved through the Tail-Call Optimization (TCO).

When the last call inside a function is a call to the function itself, the call can replace the stack frame of the current procedure with one of the newly called procedure. Tail-recursive functions are implemented in such a way that do not consume extra space on the call stack.

In Erlang, tail-recursion is very significant for the reason that processes, are in a sense "functions that live forever". The looping inside these function-processes is done through tail-call. Without Tail-Call Optimization the processes would build up memory in no time, resulting in the system's memory running out of space.

3.4 Concurrent Programming

A system is said to be concurrent when several of its computations are being executed simultaneously, possibly interacting with each other. In Erlang such a concurrent computation is called a process. The interaction between processes is happening through message passing.

Asynchronous message passing and the whole Erlang philosophy are derived from one of the many theoretical models for describing concurrent systems, the Actor Model [2]. Erlang's concurrency infrastructure is a practical implementation of this mathematical model.

3.4.1 Process

A process in Erlang could be thought of as a "blessed function". In fact, every function already written can be turned into a long-living process without any hard work. This "blessing" is called *spawning* a process.

Spawning a new process

Lets say, we have defined this function:

```
loop() ->
    io:format("Hello world! \n"),
    timer:sleep(5000),
    loop().
```

This function prints a message on the screen every 5 seconds.

```
1> test:loop().
Hello World!
Hello World!
...
```

When we call this function, we observe that the shell becomes locked and we cannot execute anything else. This is happening because the function is being executed by the shell's process itself and cannot pass control to us. What should we do is spawn a new process for our beloved function.

```
2>  Pid = spawn(test, loop, []).
```

The `spawn` BIF takes as input the module, the function name and a list of parameters to be passed to the function (in this case none). Then, it creates a new process and assigns a unique identification number to it, the *Process ID* (PID), much like the Unix PID. The Erlang people (most of them have a telecom background) like to give the "telephone number" as an example of a real-world Process Id. We should store this Pid returned by spawn in a variable for later communicating with the process.

The spawned process will "live" as long as it has more code to execute. When it has nothing left to execute, it is said to terminate *normally*. If an error occurs during the lifetime of the process, it is said to terminate *abnormally*.

It is very important, as we said earlier, when creating the process, to specify a tail-recursive function, otherwise the process will leak memory.

3.4.2 Message Passing

The processes communicate by passing messages to each other. The incoming messages are stored in the order they are delivered in a special queue structure, called the *mailbox*. Every process has its own mailbox and is responsible for reading and deleting its delivered messages.

Following the previous example, sending a message is as simple as writing:

```
3> Pid ! "people".
```

With this command we send to the previously acquired PID of the process a string message. The contents of the message can be any valid Erlang structure.

In this situation, the message will be delivered, however it will not be processed because the loop function does not contain any message processing functionality. That way, the message will stay indefinitely inside the mailbox.

If we rewrite the function like this:

```
loop() ->
    receive
        Name -> io:format("Hello ~s!\n", [Name])
    end,
    loop().
```

load the updated function and respawn a new process:

```
4> c(test). % compiles and loads a module
5> Pid2 = spawn(test, loop, []).
6> Pid2 ! "people".
Hello people!
```

The message is being popped from the mailbox and processed. We wrote a `receive` form, yet again a pattern-matching construct, that is used to try messaging patterns and upon a successful pattern-match execute some statements.

3.4.3 Timeouts

When a process stumbles upon a receive statement, it suspends execution and tries to read the next message from the mailbox. If the mailbox is empty or pattern-matching fails, the process will transition to a WAITING state until a preferred message comes.

Sometimes, it is not desirable to wait for an excessive amount of time a message to be delivered. Maybe after all the message will never come. The `after` statement inserts a timeout in the receive construct and puts a constraint on the time spend by the process on waiting for a message.

```
loop() ->
    receive
        Name -> io:format("Hello ~s!\n", [Name])
    after 5000 -> io:format("No message!")
    end,
    loop().
```

Here, after 5 seconds passed with no incoming message, the process will print instead a "No message!" string and loop again.

3.4.4 Registered Processes

In a live system, there exist processes acting as *services*; that is long lasting activities. Rather than keeping track of their PIDs, you can register an alias for them and reference them from that time on by their unique registered names. The example is self-explanatory:

```
1> Pid1 = spawn(web, blog_server, []).
2> Pid2 = spawn(web, log_server, []).
3> register(my_blogger, Pid1).
4> register(my_logger, Pid2).
5> my_blogger ! restart.
6> my_logger ! off.
```

Here are some BIFs on the registration mechanism:

register(Name,Pid) Registers a process. The Name must not be registered.

unregister(Name) Unregisters an already registered process.

registered() Lists all currently registered processes in the system.

whereis(Name) Returns the PID associated with the Name

3.4.5 Concurrency Pitfalls

Writing concurrent programs is far from being a "walk in the park". Even if your code passes the compiling and testing phases, concurrency errors may not arise, until you finally shipped the application. Erlang, being a robust concurrent language, strives to solve some common concurrency errors but there is still a possibility to stumble upon one of those:

Race Condition

When two or more activities "fight" for a mutual resource, a race-condition could happen. The outcome after a race-condition probably will be totally different from the expected result. Generally, race conditions are very hard to catch and require heavy stress-testing of the system.

Conventional languages cope with this issue by throwing locks and mutexes on every resource. Erlang begs to differ by providing a solid concurrency model. Race conditions are somewhat rare in Erlang, but there are solutions when things go bad.

Deadlock

Since Erlang programs don't use locking mechanisms you would think that the language is deadlock-free. And you are right in some way, because "Deadlock in a strict sense does not exist in an Actor system" [2].

The whole truth is that deadlocks can be seen in an Erlang system, although it is extremely rare. Consider the following situation: Process A sends a synchronous message to process B and waits for a response; in the same time process B makes a synchronous call to A and waits for it to respond. This is clearly a deadlock and the two processes will be stalled.

Process Starvation

One even more rare occasion is *process starvation*. Process starvation - a situation you mostly have to deal in Operating Systems - is noticed when a process does not get any execution time by the scheduler and remains for a long time in a holding state waiting to be scheduled. This typically happens when you mess up priorities.

Although you can play with the "niceness" of processes in Erlang, like you would do in a UNIX system, it is not advised to do so. The process scheduler in the runtime system is said to be very fair and the garbage collection happening on a per process basis makes you not care at all about process starvation.

3.5 Error Handling

There are different exception-handling mechanisms in Erlang and for this purpose there exist three different classes/types of errors to describe in some way failure of execution.

error

Runtime errors belong to this class of exceptions. Also, you can raise yourself an error by calling the BIF `erlang:error(Term)`. Term can be any valid Erlang structure that accompanies/describes this error.

throw

The throw exceptions are essentially non-local returns. They are raised by calling `erlang:throw(Term)`. Their use is discouraged.

exit

Exit are the most interesting of the three. They are heavily used on concurrent/distributed systems to inform "caring" processes about failure in another part of the system. Exit signals can be produced either by linked processes that fail or by explicitly killing a process with an exit(Reason) BIF.

try..catch

How can we handle these exceptions? We surround code that might possibly fail with a try..catch statement. This construct, again, is also using pattern matching. Upon matching with a particular exception a series of statements is executed.

In the example that follows, we run a possible-to-fail computation and catch all exceptions. Depending on the particular type of exception, we return back to the user a tuple including information about the failure.

```
try this_may_fail(X) of
    Val -> no_failure
catch
   exit:Reason -> {exit, Reason};
   throw:Throw -> {throw, Throw};
   error:Error -> {error, Error}
end.
```

3.6 Distributed Erlang

There are some reasons for (re)writing an application in a distributed manner:

- Performance

  We can benefit in speed from distributing the workload of a program across multiple machines.

- Scalability

  There are times that a machine can reach each limits (this is mostly encountered in web services). We want an easy way to withstand a high increase in traffic by just throwing more machines in the system.

- Reliability

  Replicating services and data through many host computers can give us robustness and reliability in case of a failure in one of our machines. Any work that the machine was doing before, is delegated automatically to another machine through *failover* & *takeover* techniques.

- Distributed by nature

  Some applications (such as ours!) are inherently distributed. Working with a sophisticated distributed programming language seems perfect.

3.6.1 Nodes

A Node is just a running Erlang Run-time System (RTS) that has been given a name. We can run as many nodes as we want on the same or on a remote machine. Lets start two nodes on the same host by entering this in one command line:

```
$ erl -sname foo -setcookie 123
Erlang R14A (erts-5.8) [source] [smp:2:2] [rq:2] [async-threads:0]

Eshell V5.8  (abort with ^G)
(foo@debianlap)1>
```

and on the other terminal:

```
$ erl -sname bar -setcookie 123
...
(bar@debianlap)1>
```

Then on we can communicate and send messages in processes running on any node in the same way we did with local processes:

```
(foo@debianlap)1> Pid_Bar ! restart.
```

We send a message to the other node just by having its Process Id. The PID is not only unique to the local machine, but throughout the whole distributed system. This wonderful ability is called *the transparency of communication* and gives us the necessary abstractions upon which we can structure our program's logic, without caring about locality of execution.

We can even utilize the ability of registration and do something like this:

```
(foo@debianlap)2> {my_blogger,bar@debianlap} ! restart.
```

and talk to registered processes that run in remote/other nodes.

3.6.2 Communication

What is happening behind the scenes, is that a *TCP/IP* connection is established for every pair of nodes of the distributed system. Any new node that comes in, will automatically spawn new connections to any other node in the system.

3.6.3 Security

Authentication is, basically, realized with the use of a magic cookie. Nodes that want to start a conversation must set the same cookie, an arbitrary string that is stored locally in a `.erlang.cookie` file or provided as a command-line argument.

This protection method is very rudimentary and should be taken with great care, otherwise a stolen cookie could possibly give full control of the locally running node to an unauthorized remote source.

3.6.4 rpc

The Remote Procedure Call (RPC) is a well-known protocol for interacting between processes and services of large distributed systems. Implementing RPC comes very natural, for Erlang being so concurrency and distribution oriented.

The standard library provides an rpc module for writing rpc-enabled applications. To call a function on another node you just execute:

```
rpc:call(Node, Module, Function, Arguments)
```

3.6.5 epmd

The Erlang Port Mapper Daemon (epmd) is an OS thread that listens (on port 4369 by default) for incoming connection requests by remote nodes and maps them to the appropriate local node.

3.7 Introduction to OTP

The OTP framework provides us the abstractions we need to easily model our concurrent intentions. Here, we take a look on the three generic *behaviours* of the OTP, that is the **Server**, the **FSM** and the **Supervisor**. If you come from an object-oriented background, you can think of the OTP behaviours as an abstract interface or a partially implemented class, where you "fill the left out parts" with your implementation.

Next, we will bundle these behaviours together, along with any metafile, into a package using another behaviour, called the OTP **Application**.

3.7.1 Server

Generally, in a client-server model, the purpose of the server is to accept some calls and respond to them. Also, the server maintains an internal state, which he modifies according to the received messages.

To build a server behaviour, we write down a callback module that will export a set of particular functions.

```
-module(my_server).
-behaviour(gen_server).
-export([init/1,
         handle_call/3,
         handle_cast/2,
         handle_info/2,
         terminate/2,
         code_change/3]).
```

A typical description of these functions:

init(Args) -> {ok, State}

This function is called upon instantiating the server. If everything is ok, it should return the initialized state of the server.

handle_call(Request,From,State) -> {reply, Reply, State}

This function is called when a synchronous request to the server is made. The server possibly alters its state and returns a reply to the caller.

handle_cast(Request,State) -> {noreply, State}

When an asynchronous call (a cast) is made, the server does not reply to the sender, but will just process the request received and consequently change its state.

handle_info(Info,State) -> {noreply, State}

The calls and the casts to the server are OTP-custom-built messages, so it is not advisable to send messages to a gen_server (and generally any OTP behaviour) using the *primitive* message passing we saw in the previous chapter.

However, if this is required, you can implement the handle_info function to handle the classical Erlang messages.

terminate(Reason,State) -> ok

Terminate is, in essence, the opposite of the init function. It contains cleanup code, which gets executed before the server process is terminated.

A termination of a behaviour does not necessarily indicate failure in the program. It is possible that a supervisor signaled a restart or a shutdown of his supervised processes, or the process itself decided to gracefully stop executing.

code_change(OldVsn, State,Extra -> {ok, NewState}

This callback function will run during a live upgrade of the system (using the hot code loading techniques we mentioned earlier). Its main purpose is to safely modify the internal state to use a possibly new format. If this turns out to be a very difficult process, you can of course reset its state.

The interaction with a gen_server is realized through some library functions:

gen_server:start_link(ServerName, Module, Args, Options) -> {ok, Pid}
: Spawns a new child server process and *links* it with the parent process. If the server goes down, the parent process will exit too.

gen_server:call(ServerRef, Request) -> Reply
: Sends a request to the server and waits for a reply.

gen_server:cast(ServerRef, Request) -> ok
: Sends an asynchronous request and returns immediately.

3.7.2 FSM

This behaviour implements an event-driven Finite State Machine (FSM) using a gen_server as the basis. By definition, the loop in an event-driven FSM makes these steps:

1. Consume an event
2. Take some actions
3. Transition to a new state

In the gen_fsm behaviour, the events are signaled with OTP messages. The states are just callback functions inside the module and the actions are normal statements that the fsm executes.

A typical gen_fsm will look like this:

```
-module(my_fsm).
-behaviour(gen_fsm).
-export(init/1,StateA/2,StateB/2,terminate/3,code_change/4).

StateA({switch, on}, StateData) -> % the event is pattern-matched
    ...some actions here...
    {next_state, StateB, NewStateData}.

StateB({switch, off}, StateData) -> % and again here
    ...some actions here...
    {next_state, StateA, NewStateData}.
```

What is missing here, is the normal `init`, `terminate` and `code_change` functions. We should write them in a similar fashion to the gen_server example.

You send events to the FSM process by calling `gen_fsm:send_event(FsmRef, Event)`.

3.7.3 Supervisor

The supervisor is the most important feature of the OTP framework, yet it still remains easy to comprehend. What the supervisor basically does, is to monitor a set of processes and restart them if necessary. These processes can in turn be other supervisors or just workers (i.e. server, fsm or event behaviours).

The parent supervisor with its spawned children form a tree of processes, called the *supervision tree*.

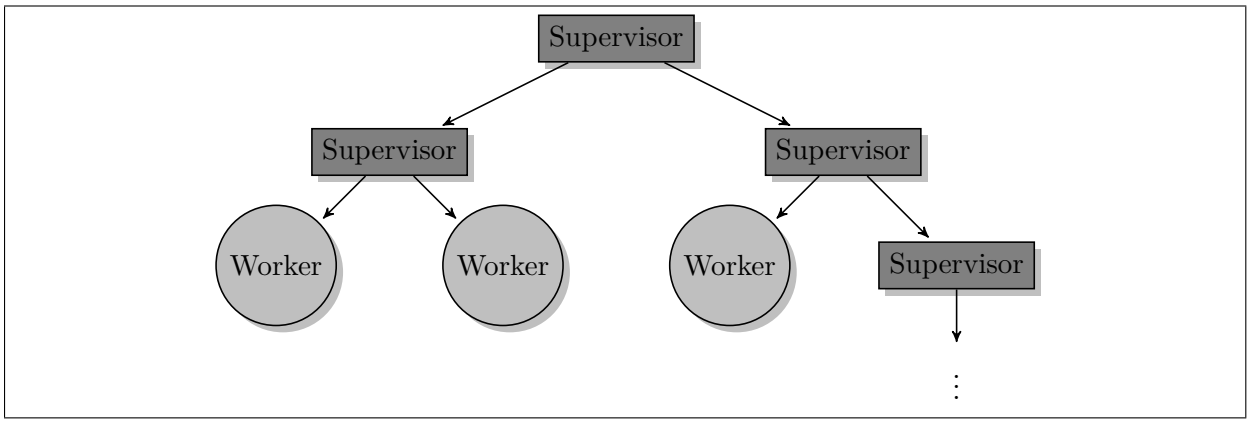


Figure 3.1: An example of a simple supervision tree

The *restart strategy* that the supervisor follows can be one of:

one_for_one
: Restarts only the crashed child process and the siblings remain unaffected.

one_for_all
: Restarts all the child processes when a crash occurs in one of them. This suggests a strong dependency between the siblings.

rest_for_one
: Will restart only the processes that were started after the crashed process. This implies that the child processes were spawned on a dependency row.

The Supervisor and the Child Specifications will go inside the `init` callback function of the supervisor module.

3.7.4 Application

An OTP Application is not used solely as a packaging medium (*library application*) for distributing code; it can act as an entry to, and an exit from your program. This type

of applications are called *active applications*. Every active application should internally specify its interface; that is a *root supervisor* that will spawn all other processes of the system.

The only things you have to implement is the `start` and `stop` functions.

The library as also the active application dictate a directory structure where the modules and the metadata files should live in:

```
application-name/
                |--- src/       Source code (.erl)
                |--- include/   Include files (.hrl)
                |--- ebin/      Bytecode of the source (.beam)
                |--- doc/       Documentation of the code (EDoc)
                |--- priv/      Non-erlang-related files (such as dlls)
```

3.8 Conclusion

Although, we strived to cover the Erlang language from A to Z, we can jokingly say that we stopped somewhere around "T". There are parts that we left out, because we think they are rather unimportant to the implementation of our application, such as the functional merits of the language (i.e. lambda expressions and higher-order functions), the very interesting Hot-code Loading technology, Interfacing with other languages and last of all Socket programming.

Chapter 4

System Analysis

We present here the design and decisions made for our system based on the theory behind MultiAgent Systems. After that, we will see how we came to implement it, with the help of the distributed mechanisms provided by the Erlang language and its OTP framework tools.

4.1 Overview

In the introductory chapter, we gave an early description of the problem we are trying to solve. Now we are going to restate the same problem in a different way, so as to better fit the technicalities of the solution.

The *old-school* paper recycling involves three kinds of individuals that actively participate in the process chain: the companies, the municipal office and the recycling truck vehicles. Lets examine a typical interaction:

After unpacking a newly arrived shipment or when the paper-made products start to wear out, a company employee will realize that some paper quantity is no longer used and has been left over. Then, the person in charge inside the company will call by telephone the municipal office to state that they dedicate a paper amount to recycling.

The office, will then, try to reach out, in sequence, every truck that it administers and announce them about the new request made. The truck driver will examine the request and decide to respond to the office positively by stating the time he/she believes is necessary to carry out the task or negatively by refusing to handle such a request for his/her own reasons.

After that, the municipal office will gather all responses made. It will filter out the

refuses and rank the affirmative responses based on the proposed time of each one. The office will award the best truck (that is the truck with the shortest dispatching time) to carry out the task.

Finally, the winner truck will call the office a last time, to indicate that the task has been carried out. This last call is necessary, because the central office has to be sure that, at the end, the request will be successfully processed. If a lot of time passes with no news from the assigned truck, the office is free to reestablish a negotiation for the request and assign the job to a different truck.

In comparison with the approach described above, our application wants to:

- Get rid of the central office
- Automate the communication mechanisms
- Help the truck driver to manage tasks and transactions

In this way, we can attack the distribution and management bottlenecks found in the aforementioned process.

To put it it simply, what we are looking for is an entity that will handle the tasks for us and communicate with other entities when needed. This entity should:

- Act without guidance (autonomy)
- Take the initiative when suited (proactivity)
- Change its intentions when they are no longer valid (reactivity)
- Interact with other entities so as to better achieve its goals (social ability)

All these characteristics can be found in an *agent*. We will replace every person that takes part in the interaction with an appropriate agent. Specifically, we will construct an agent that will take the role of the customer company and substitute the redundant telephone calls with digital communication. Every truck will be represented by a truck agent that will have the double goal, that of managing its set of tasks and also communicating with other agents.

We can easily recognize a familiar interaction pattern taking place in the *old-school* process. A task is announced by an initiator to a list of participants, then these participants *bid* for acquiring the task, and at the end the task is delegated to the best bidder. This distinctive pattern closely resembles the *Contract Net* interaction protocol.

As we said earlier, one of our intentions is to remove from the system the need for a central administrative authority (i.e. the recycling office). Although this can be easily done, we still need a central point where the agents' identities will be stored. Imagine a MultiAgent system where the agent can interact with the environment but cannot communicate with other agents, because it does not know where they "live". For this reason, we need to set, inside the system, a Directory Facilitator (DF). You can think of the DF, as a phone book but instead of mapping names to telephone numbers, it maps agents to their addresses.

4.2 Modeling

We think it would be helpful to draw a general picture of the concept with all the implementation details removed, so as to better understand the purpose and the nature of our system.

There are three types of entities that form this multiagent system:

- the Directory Facilitator (from now on just **DF**),
- instances of the **Customer** type
- instances of the **Truck** type

The first step an agent (Customer or Truck) has to take to become a member of this system is to login, by passing some user information to the DF. The DF stores and manages a list of all authorized agents. Upon authorization, the agent is becoming aware of all the other connected agents and can interact with anyone he prefers.

A typical situation, after that, involves issuing a recycling request by the Customer. Now that the Customer knows about all subscribed trucks of the system, sends a request to each one of them, stating his position and the paper quantity he wishes to recycle.

The interaction goes into a bidding phase, where Trucks respond affirmatively to the Customer with a bid; that is an estimated time they believe is needed to fulfill the request. Naturally, Trucks can also refuse to such a request, for their own reasons and beliefs.

The Customer receives all bids, processes them, and decides to pick the truck with the best bid as the agent that will eventually perform their request.

The Winner-truck, then, begins to process its assigned job and take the necessary actions to bring itself closer to that goal. Upon arriving to the Customer, it will pick up the paper quantity and mark the job as finished.

Later on, when the truck will become near full, it will head back to the Recycling Factory to unload its quantity and start possibly a new conversation.

Of course, this is just a trivial case where everything went exactly "by the book". In real cases, the interactions involved will be arithmetically larger and more complex, so things can go wrong. For this purpose, we designed a robust, fault-tolerant system, adding to it enough mechanisms to handle such situations arised.

4.3 Specifications

Prior to implementing the actual program, we did discuss and conclude on behaviours such a system must impose:

Online

The system and its services should be online and available all the time, with as little as possible downtime.

Real-time

Generally, in MultiAgent systems, time plays a very critical role. That means, the decisions and actions the agents take is bounded by time. The computations involved and the overall communication process of our system should be extremely fast, even measured in milliseconds. Often, this is not the case, because of the limitations created by slow network connections.

Furthermore, the time spent between issuing a request and finally assigning it to a truck should be as minimal as possible, because otherwise the customer will have the illusion that the system is unresponsive.

Protocol

To make communication easier, the agents should speak a mutually understood language. This is realized by using a well-defined protocol. We came to implement a custom (FIPA Contract-Net-like) protocol.

No loss of information

The system should guarantee that requests never get lost and will some time in the future be processed. Also, whatever will might happen, the contents of these requests should be left unchanged.

Distribution

The program should be distributed in three separate applications. In particular:

- A server-like application for starting the DF.
- A desktop and mobile version of the Customer application to install in companies that want to recycle.
- A mobile version of the Truck software to mount on the truck vehicles.

4.4 Features

There were things that were not initially thought of and not crucial to the system's good operation, but were later added on to give extra power and functionality to the system.

Queuing jobs

Every Truck maintains its own plan, that is a queue of future jobs it intends to execute. The Truck is responsible to keep the plan in a sound state and not include any unreasoning sequence of instructions. Besides adding jobs drawn by requests, it should have the ability to proactively create jobs in agreement with its intentions.

Splitting requests

Requests, as we said earlier, include the paper quantity the customer is offering for recycle. There is a possibility that this quantity exceeds the maximum capacity of even the largest truck in the system, so it cannot be processed by one truck only. Moreover, there are occasions where no truck's current capacity is enough to fit the paper quantity.

In this sense, it comes natural to "break" the requests into smaller ones, that can be processed by more agent trucks. The Customer issuing the request must decide on what exactly the magnitude of this splitting should be.

Google Maps Integration

The Truck application should assist the actual driver of the vehicle with choosing the shortest path along the job to follow; much like a GPS navigation system would do. Instead of using paid, proprietary navigation software, we took advantage of the public web service that Google offers, called Google Maps.

Despite being free, Google's product is superior to other software, in the sense that it has a *global* geographic information coverage. Otherwise, you would have to load the necessary maps into your GPS. This gives to our system the ability to operate on all places Google Maps has information for.

For an added bonus, Google Maps gives nearly-live traffic data for some popular cities. In this way, the recycling operation gains a lot in speed and the service availability increases.

GUI

The implementation phase was followed by heavy testing, where we constructed and ran custom artificial cases and compared their results against optimal behaviours of an ideal system. This process was somewhat cumbersome, so we decided to build a graphical extension to the system to better inspect/monitor the rationality behind "its" decisions.

The GUI we came up with draws, in real-time, a panoramic view of all the logged-in agents of the system and their particular actions. We caught, this way, many bugs and logical errors.

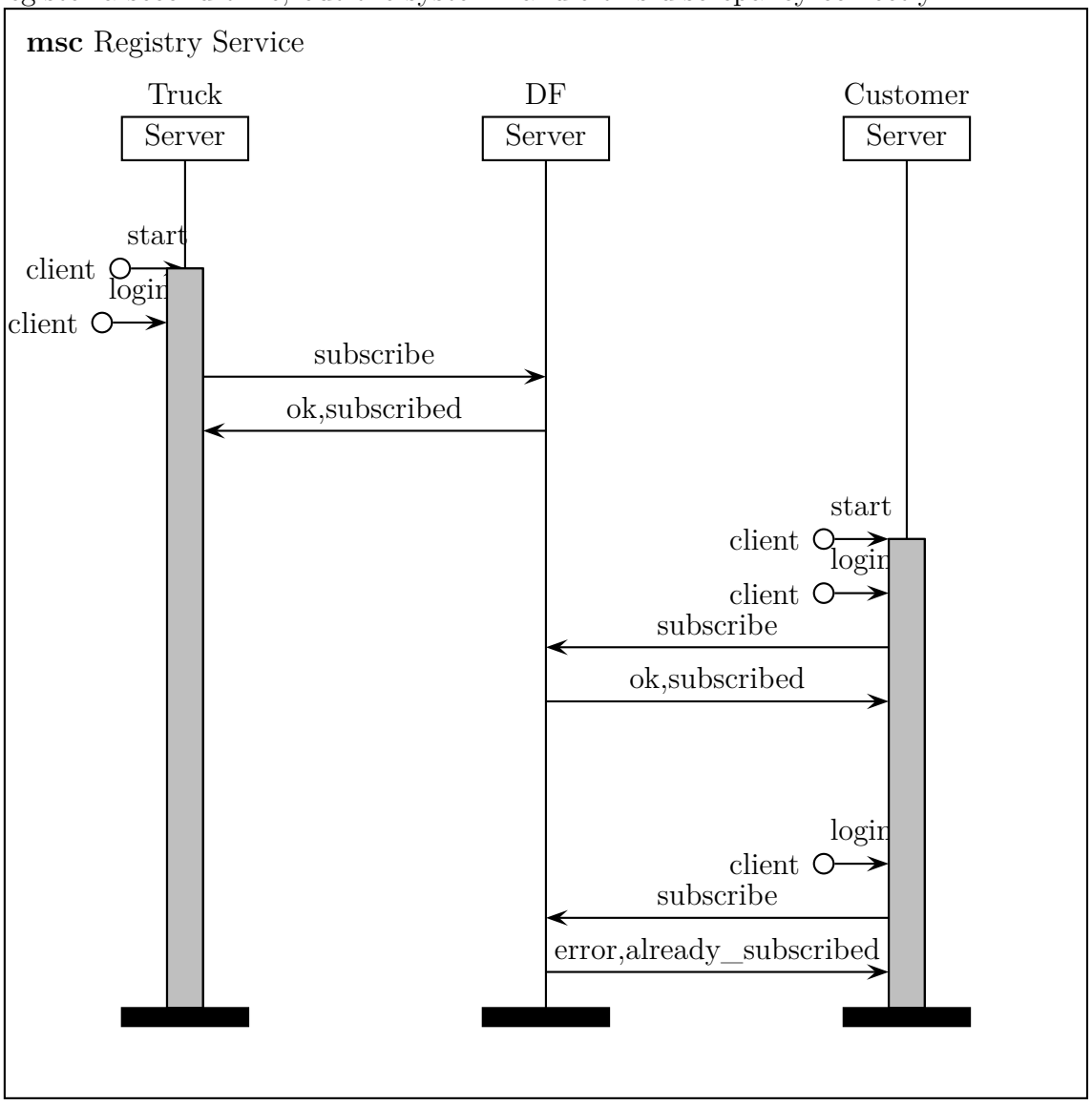
To provide a great user-experience to the clients, being a Truck driver or a Customer, we transformed the previous monitoring application to a web frontend for better interacting with the underlying software.

4.5 Design

The system offers two distinct services, namely the *Registry* and the *Recycling* Service. Although we picture them separately, the truth is that they very much depend on each other.

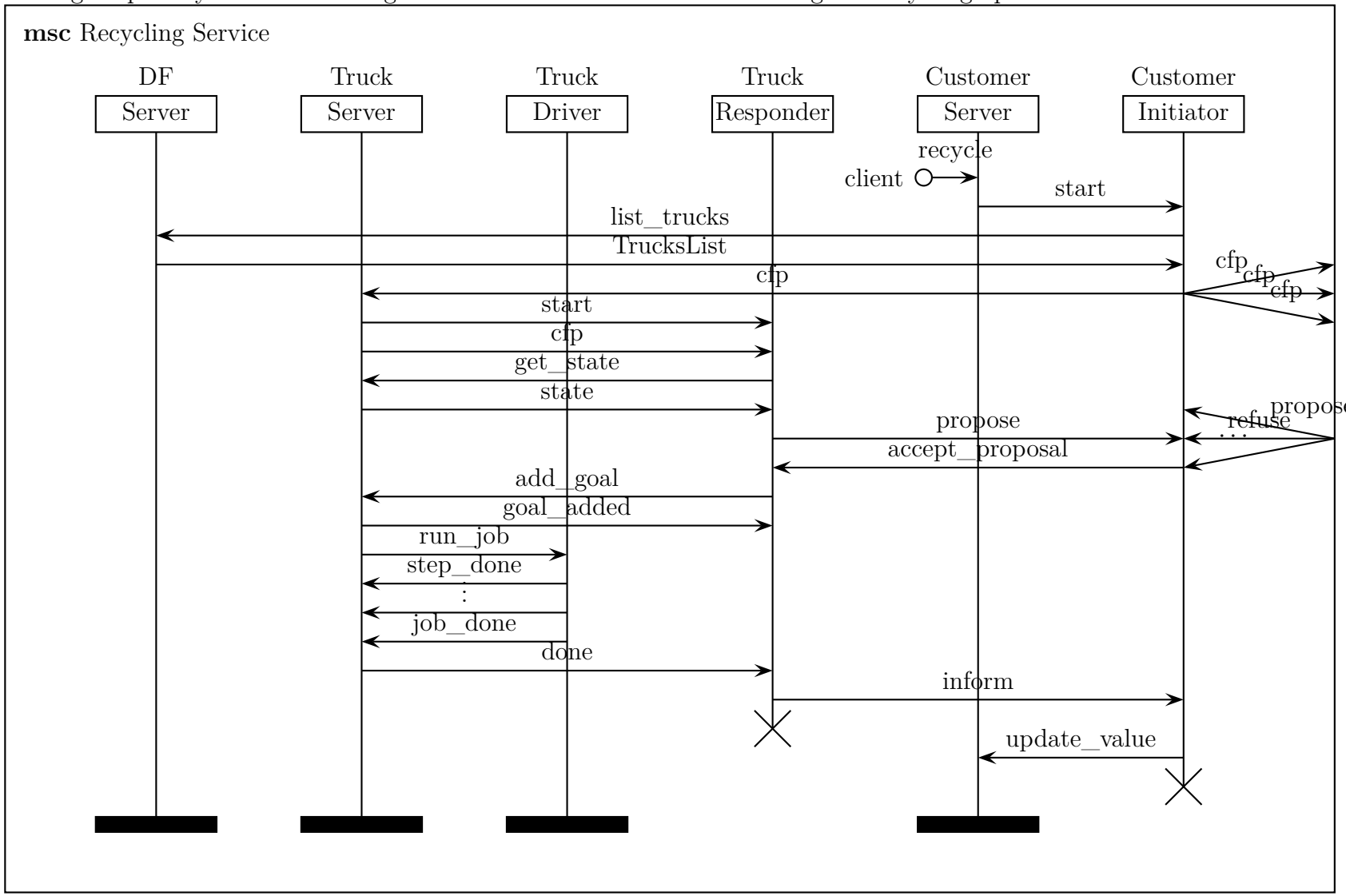
4.5.1 Registry Service

What follows, is a Message Sequence Chart that illustrates a common case found during a Registration interaction. We have two agents, one Truck and one Customer agent, that sequentially – we did that for illustrative purposes, it could as well be done concurrently – try to authenticate and enter the system. At the end, we can see that the customer tries to register a second time, but the system handle this discrepancy correctly.



4.5.2 Recycling Service

The figure portrays a canonical negotiation between a Customer initiating the recycling operation and a Winner-Truck.



4.6 Implementation

We have covered everything we need to implement our program. The system can be split into three different OTP Applications that we describe below.

4.6.1 DF

The DF is a key-ingredient component of the system, for the simple reason that it acts as the sole entry point to the system's "world".

Every agent that wants to become a member of this system has to first "talk" to the DF, giving him some user credentials and getting back an authorization. In this way, we can say that the DF works as the central authority of the system.

The DF stores some extra information about the live state of every online agent with the goal to provide a graphical system-monitoring service.

Another operation that the DF is capable of, is to inform agents about the existence of other agents inside the system. This is used, mainly, by the Customer agents to send their Recycling requests over all subscribed Trucks.

In a possible crash of the DF application, the subscribed agents will still be capable to interact with each other, however no new agent could log in to the system.

Finally, one important thing that should be mentioned, is that since the DF is the entry point of the system, the node location that the DF is running on, should be known by all the agents prior to subscribing to the system.

Supervision tree

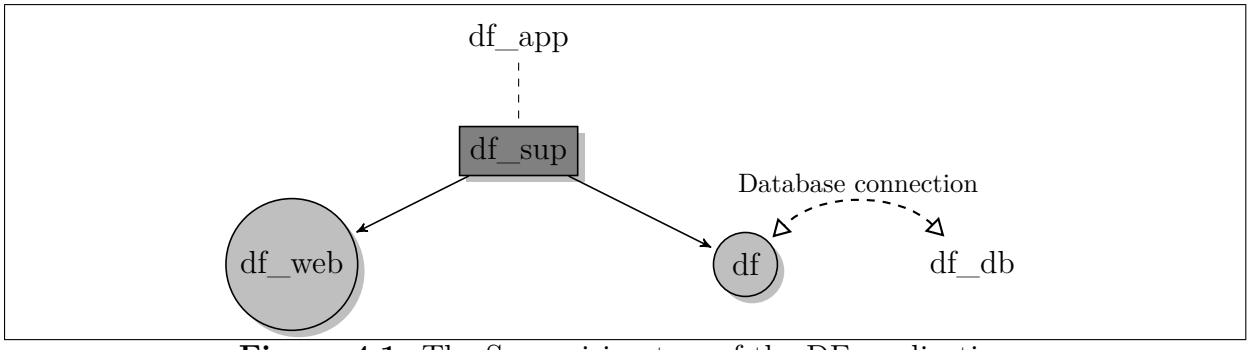


Figure 4.1: The Supervision tree of the DF application

Lets examine briefly some of the functionality of the modules that form the DF.

df_db

We naturally have to store somewhere the logged on agents along with some information about them. We chose to use, for this purpose, the Erlang Term Storage (ETS), which is a mechanism to store items in memory and access them in constant time. It can be described as a fast but very primitive data store. It is packaged inside the standard Erlang distribution, so we didn't introduce any new dependency.

ETS has some extra options, such as *disk saving* or *concurrent access to the database*, but the directory registry does not require any persistency nor any concurrency, so we left them out.

Besides the classic `create_table`, `insert`, `delete` and `member` operations, which are built on top of the ets library API, there are two extra functions: the `list_trucks` and `list_all`, which are, as we will see shortly after, indirectly called by the agents to get "acquainted" with each other.

df

The df module mainly acts as the server-frontend to the database. It is implemented by a **gen_server** behaviour that handles requests, such as:

subscribe
: Log in to the system requires from the agent to state its type (Customer or Truck), its current geographical position and its paper value. The DF also tracks the location of the node where the request came from and does a mandatory check to see if any agent with the same node name has already logged in. If not, he gives authorization.

unsubscribe
: The agent casts a request whenever he wants to log out of the system.

update
: The agent is responsible to provide near live data about his current state. This is done by periodically sending messages to the DF by casting an update.

list_trucks & list_all
: These are the requests the agents send to the DF, where they get "translated" to the corresponding function calls of the df_db module.

df_web

The df_web module implements the server part of the Monitoring Web Interface. It wraps a Misultin[1] server around an OTP gen_server. Misultin is an Erlang library for building fast lightweight HTTP(S) servers.

What is happening during monitoring, is that the client opens a Google Maps frame and makes an in-browser *long-polling AJAX* request to the df_web server. Our server processes this request, fetches all real-time agent data, transforms them into *JSON objects* and ultimately feeds them back to the client's browser to update the map. This procedure is repeated in short intervals.

df_sup

The root supervisor of the application, spawns and monitors the two server processes, the main df server and the supplementary df_web server. If any of them crashes, the supervisor will restart it using a one_for_one restart strategy.

df_app

Except the previously described modules, the application bundles also an open-source JSON encoder/decoder library written in Erlang, called jsonerl. It is used by the web server to encode the agent data.

4.6.2 Customer

The Customer application should be distributed to the companies that take part in the recycling process. There exist equivalent desktop and mobile versions of the application software.

[1] `https://github.com/ostinelli/misultin`

Supervision tree

This simple supervision tree illustrates the simplicity of its model behaviour:

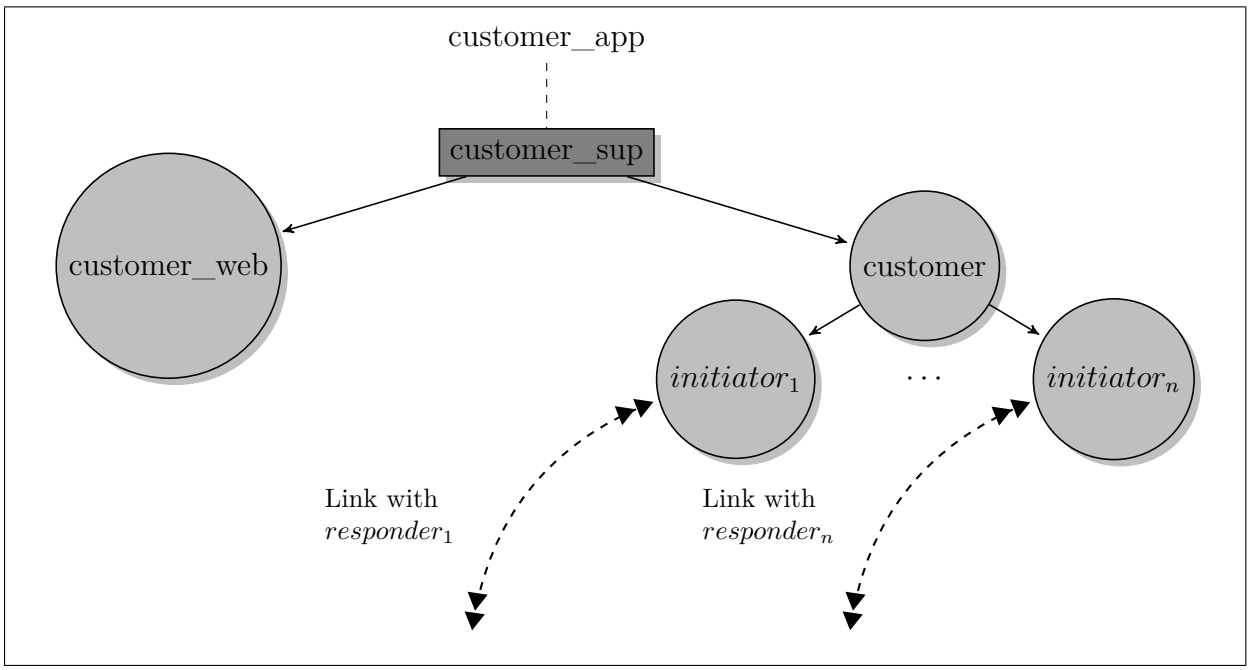


Figure 4.2: The Supervision tree of the Customer application

customer

The customer is an instance of the gen_server behaviour. What it basically does is that it stores the agent attributes and preferences while providing a public API to access or modify them.

Besides storing values, its main duty is to trigger a new *contract-net negotiation* upon receiving a *recycle* command from the client. From now on, when we use the word "client", we mean the person that is operating the machine (in this case the company employee). The server spawns a new customer_initiator process, that will start and manage a conversation with the Trucks.

The customer server is totally independent of the initiator's thread of execution. If the initiator goes down or a failure happens in the established conversation, the server remains unaffected. Built upon this characteristic, the server can spawn as many initiators as he wants based on the "stimulation" he receives from the client.

Unlike the short lifetime span of an initiator, the customer process will stay as long as the application is running.

customer_initiator

The initiator module is, somewhat, the "half" implementation of the Contract-Net protocol. The other half is implemented in the Truck application. The nature of this protocol is not at all like a one-pass communication, but requires successive steps of interaction. That is the reason why we chose to model it in a gen_fsm behaviour. These communicative steps are essentially the transition states of the FSM.

We picked up two snippets of code for a showcase:

```
send_acceptances(BestProposer, Proposers) ->
    gen_fsm:send_event(BestProposer, accept_proposal),
    link(BestProposer),
    lists:foreach(fun (T) -> gen_fsm:send_event(T, reject_proposal) end,
                  lists:delete(BestProposer,Proposers)).
```

In this piece of code, we can observe the functional aspects of the language. First, we send an ACCEPT_PROPOSAL message to the Winner-Truck, but we do not forget to *link* with it. A link with another process can be thought of as a "bidirectional monitor". If any of the linked processes exits abnormally, all other processes in the linked chain will exit too. An abnormal exit could be signaled by a system failure or a network disconnection. Last, we send a REJECT_PROPOSAL to the rest of the proposers, mapping a function over their list.

Next, we can see how the splitting of the Call-For-Proposal (CFP) message is done. The `break_cfp` function takes as input the current contract and breaks it in half, spawning two new separate initiators to start negotiating. After spawning them, the initiator itself stops. Notice, that we have put a threshold on the size of the splitting; if the quantity is below 1kg, the initiator should instead reset. This decision was made to avoid excessive meaningless splitting of CFPs.

```
break_cfp(_Contract = {Position,Value}) ->
    case Value > 1 of
        true ->
            start({Position, Value/2}), % start 2 new inititiators
            start({Position, Value/2});
        false ->
            start({Position, Value}) % small quantity, don't break, just reset
    end,
    {stop, normal, []}.
```

customer_web

The customer_web server is derived from the equivalent df_web implementation. It adds some GUI tools and buttons for the client to operate on.

customer_sup

The supervisor upon starting, will spawn only the web frontend of the application; the client himself should press the *Login* button to state confirmation. After that, the web server will dynamically add the customer process to the supervision tree by calling:

```
supervisor:start_child(customer_sup, CustomerServer)
```

customer_app

There is not really anything new to say about this application, except that we wrote a lot of metafiles that we put inside the `priv/` directory. These were static html files, that the customer_web process was serving.

What it came to be a surprise is that, with the technologies we used to build our system, we didn't have to implement two distinct versions of the same application (one desktop and one mobile). With little to no change, we can make the application run wherever we have an Erlang VM to run on.

4.6.3 Truck

The Truck is the third and last component of the system. This software must be installed on a mobile device running inside the truck vehicle. The application agent should act as an assistant for the driver.

Supervision tree

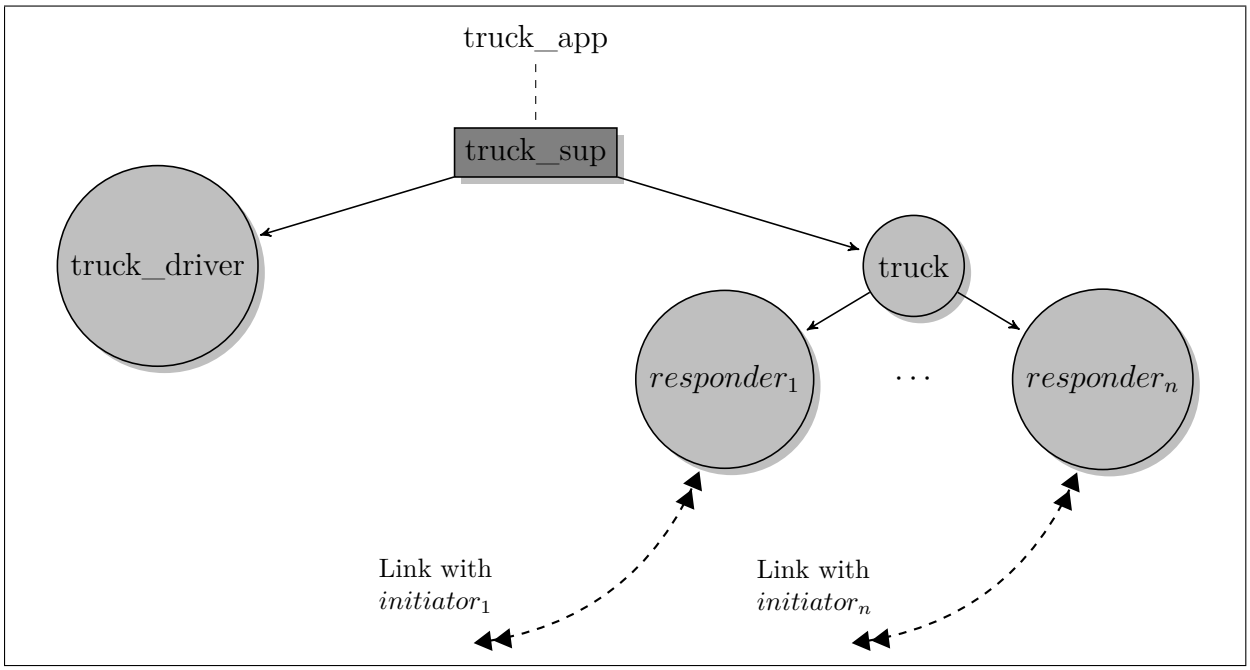


Figure 4.3: The Supervision tree of the Truck application

truck

Much like the customer main server, the truck server is responsible for storing all the agent attributes in its internal state, giving a client API to access them. This API is again built around the gen_server behaviour.

The truck server is listening for incoming CFP requests from the customers. For every CFP delivered, the server is spawning a new truck_responder process, passing to it the CFP message, just as it was received.

We can have many responders negotiating and bidding in parallel without affecting the truck server. However, in some situations, conflicts can arise and we have to deal with them in an effective manner.

truck_responder

This gen_fsm behaviour complements the left out part of the Contract-Net implementation. After the behaviour's instantiation, the responder process receives a Recycling request (a CFP message) and decides if he can sometime in the future fulfill it.

If he is willing to carry out this request, he responds with a Bid, that is a `PROPOSE` performative, which contains the Estimated Time Arriving (ETA), or to put it differently, the Estimated Time to Accomplish the goal.

Otherwise, if he does not feel like committing to the request, because maybe the quantity exceeds his maximum or current capacity, or the request is incompatible with his constructed plan, the responder should send back a `REFUSE` message, stating the reason for doing so.

After the bidding phase passes, the Customer will issue back the results by sending outcome messages to all the Bidders. If the responder receives a `REJECT_PROPOSAL`, he realizes he was not the Best Bidder and having nothing else to do, he gracefully dies.

On the other hand, if the responder receives an `ACCEPT_PROPOSAL`, this essentially means he represents a Winner-Truck. From now on, the Customer believes that the Winner is committed to get the job done.

In a typical situation, this assigned job will be added to the end of the plan/queue and will eventually be picked up and executed by the driver behaviour, sometime later in the future. When the job is finished, the truck_responder will signal a `INFORM` performative back to the Customer and stop execution.

However, there exist some corner cases, where the responder should send a `FAILURE` message to the customer_initiator, to express that he does not believe anymore that he can carry out the job. Warning: this is different than sending an `EXIT` message, signaling an abnormal behaviour of the truck's execution, such as a network error or a system crash. In a `FAILURE` situation, what happens is that although the Truck is in a "healthy" condition, some conflicts have arose that prevent it from acting normally. This is attributed to a "race condition" happening in that particular moment. Consider this case:

Two responders of the same Truck are bidding for two different recycling requests at the same time. Their bids are a computed estimation for completing the tasks based on their current future plan. If they both receive an `ACCEPT_PROPOSAL`, from then on the truck is committed to fulfill both jobs. However, since these jobs cannot execute in parallel and must be run in a sequence, one job is going to be scheduled for later than that it was supposed to be.

In this way, even though both jobs would, in the end, execute successfully, one responder, somehow, would have lied about its ETA. In our design, this "liar"-responder must send a `FAILURE` to the Customer to "let the truth shine".

truck_driver

The truck_driver is, how its name suggests, responsible for guiding the vehicle inside the city roads. Like the truck behaviour, he is also a *singleton* gen_server process, which means that it is a unique instance to the program and stays as long as the application is running.

In a production environment, this module is not going to be necessary. We use it only for **simulating purposes**. When the system is going to deploy, the truck_driver module can safely be removed.

Its main functionality is pictured in this function:

```
Run_step = fun (Step = { _ , Duration}) ->
                    timer:sleep(round(timer:seconds(Duration)*?PROG_SPEED)),
                    truck:step_done(Step)
            end,
```

The driver pops the next step from the job, sleeps for a virtual amount of time necessary to take the step, and tells the truck server to update its position. For having a, somewhat, objective simulation of the system, we added again some virtual *pickup steps*, where the truck_driver spends some time to load the truck, proportionally to the paper amount.

gmaps

This module simply contains some XML parsing functionality, based on the `xmerl` XML parser, which comes with the standard Erlang distribution.

It provides, all and all, three functions for issuing *Google Maps* requests and parsing their results: `compute_eta`, `compute_steps` and `compute_pickup`.

truck_sup

The truck_sup supervisor will spawn and manage the two main servers, the truck process and the truck_driver process. This time, we use the *one_for_all* restart strategy, for the reason that these two processes very much depend on each other.

truck_app

The truck application contains all the previously described modules and nothing more.

4.7 Testing

Throughout the development phase of the system we did systematically run debugging tools to correct some common bugs found. Difficult to catch were the race conditions discussed earlier.

After that and when the program reached a usable enough state, we conducted a series of tests to establish that our system's model is foolproof. The first thing we did, was to instantiate a list of different agents (trucks and customers) and enter them into our system. Then we designed some test cases with arbitrary data and attributes around these particular agents. We solved ourselves these cases on paper and the solutions took the form of blueprints we later tried to match against.

For the small-size cases everything went very well, however for the larger ones we ran into some problems. It was hard to construct cases with many agents, but even harder was to trace the results of the testing. So we decided to build a graphical interface that will allow us to inspect in real-time the overall progress of the system. This interface later came to be the *Monitoring GUI* of our program.

In the figure that follows, you can see a simple test case taking place with 3 truck and 6 customer agents.

Chapter 5

Conclusions

Combining the MultiAgent theory with the concurrency & distribution mechanisms of the Erlang language, we implemented a functional production-capable MultiAgent system for the management of the recycling process.

We strongly believe that the expressiveness, the robustness and the fault-tolerance are some of the most important qualities that a MultiAgent system should meet. And, although Erlang is not specifically classified as a platform for designing such systems, we think that it does have the features required to base upon your very own Agent platform in a simple and elegant fashion.

5.1 Future Work

Some potential features of the system, that we were initially come up with, ultimately missed the final program, because, in that time, they seem to be very time-consuming or hard to implement. Others "came late to the party", so we did not want to interrupt our development progress and left them out.

The most interesting ones:

Dynamic replanning & rescheduling

Every truck agent maintains its own plan that, even though it is possible to append/queue jobs in it, you cannot say that it is very dynamic. A dynamic plan would in real time sort and prioritize its jobs, keeping them always in a consistent state. In this way, we could build a minimal overall travel path of jobs.

Convergence

Lets examine the case where we have a group of customers closely to each other and one sole customer settled in a rather far destination. The truck that will "win" this distant goal, will remain idle after fulfilling it, because in consequent requests he will constantly lose.

If the truck agent, however, could recognize such situations, it would not stay inactive, but try to approach the group of customers, therefore increasing its performance and success rate.

Better Splitting

Committing to the KISS (Keep It Simple, Stupid) philosophy, we strived to keep our system's design and implementation relatively easy and straightforward. With that in mind, a large recycling request just splits in half, creating two more requests that will be processed. Thus, we can say that our splitting algorithm is not that "intelligent".

We could instead take full advantage of the refuse reasons and devise an algorithm, that would break a request into as few pieces as possible, necessary to accomplish the entire goal. In this way, we could gain a lot in availability and increase the overall performance of the system.

Switch from ETS to Mnesia

The database backend of our DF application relies on the Erlang Term Storage (ETS), a somewhat simple storage mechanism, yet very fast. If we, however, replaced ETS with a Mnesia database, we could exploit the fault-tolerance and distribution powers that come for free with the Mnesia system.

There would be some replicated instances of the DF database, where in a possible failure of the main server, a slave instance would take action (failover technique) and become itself the master server. The system would continue to work just fine and its users would not notice a thing.

More testing

Even though we ran a great amount of tests and covered the most common cases, we can certainly use more of it. This would ensure that the system "thinks" and works as expected to.

Benchmark reports

Excessive stress-testing of the program – how well it performs under extreme cases of high load – would yield some interesting results about the real limitations of the system. Built upon that, we can explore new patterns of performance optimization.

Use it in a real environment

If we had the opportunity to deploy in a real-world environment, we could experience, from first hand, the patterns and procedures that emerge "in the wild". We could also examine how the user interacts with such applications and fine-tune them, so as to become more user-friendly.

Finally, our program would transition from merely being a production-capable future application to a fully production-ready system.

Bibliography

[1] U.S. Environmental Protection Agency. Recycling. `http://www.epa.gov/osw/conserve/rrr/recycle.htm`, May 2010.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[4] Joe Armstrong and To Helen. Making reliable distributed systems in the presence of software errors, 2003.

[5] Niclas Axelsson. Erlang for the android platform. `http://www.burbas.se/artiklar/erlang-for-the-android-plattform/`, November 2010.

[6] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly Media, Inc., 1 edition, June 2009.

[7] FIPA. Fipa contract net interaction protocol specification. `http://www.fipa.org/specs/fipa00029/SC00029H.html`.

[8] James Hague. How to crash erlang. `http://prog21.dadgum.com/43.html`, June 2009.

[9] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action*. Manning, November 2010.

[10] The Guides Network. Recycling is important. `http://www.recycling-guide.org.uk/importance.html`, 2010.

[11] Local Union of Korinthia. Paper recycling and its meaning. http://www.anakyklosi.com.gr/site.php?&file=pages.xml&catid=44, 2006.

[12] Ted Patrick. Recycling. http://ted.onflash.org/2008/01/why-erlang.php, January 2008.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall series in artificial intelligence. Prentice Hall, 2 edition, December 2002.

[14] Jennifer Spenader. Speech act theory, introduction to semantics. http://odur.let.rug.nl/~spenader/public_docs/speech_acts_groningen_NOV.pdf, November 2004.

[15] Leon S. Sterling and Kuldar Taveter. *The Art of Agent-Oriented Modeling*. The MIT Press, 2009.

[16] Katia P. Sycara. Multiagent systems. *AIMag*, 12, 1998.

[17] Ulf Wiger. Erlang OTP. http://www.slideshare.net/nivertech/erlang-otp/, December 2003.

[18] Wikipedia. Recycling. http://en.wikipedia.org/wiki/Recycling.

[19] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2nd edition, July 2009.