Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Drift: an imperative programming environment for the cloud

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Frank Lange
geboren am: 08.08.1989
geboren in: Berlin

Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Joachim Fischer

eingereicht am: ..... verteidigt am: .....

Contents

"If I had more time, I would have written a shorter letter."

– Blaise Pascal

1 Introduction

...

1.1 Goal of this work

...

1.2 Structure of this work

...

2 Distributed Programming

In 1937 a paper titled *"On computable numbers, with an application to the Entscheidungsproblem"*, written by Alan Turing, was published [**?**]. In it Turing defines what is now known as a *Turing machine*, a theoretical machine model for executing arbitrary computation. He demonstrates how to formulate instructions for this machine in order to configure its execution behavior and goes so far as to show how this machine could even receive another machine configuration as its input and then execute what the other machine would have done, thereby effectively emulating the given machine.
But he wasn't the only one thinking about computation. As summarized by Denning in [**?**], Kurt Gödel, Alonzo Church and Emil Post all contributed to the task of exploring the boundaries of computation, setting the tone for the next 80 years of research concerning *computation*, *computers* and *automation*.

Of all the proposed approaches of how to tackle the concept of computation and turning it into a tool that could be understood and used by delivering a theoretical framework, one could argue that Turing's machine model emerged as a winner because, although not at the time, it turned out to be closest to what machines became to be. How they were structed and how they basically operated. Still, to this day.
So in order to use or *program* these machines, these computers, programming languages were developed and over the decades have gone through their own evolutionary process [**?**] with the first generation using binary machine instructions to todays languages that allow for higher level programming styles like *object oriented programming* [**?**] or *functional programming* [**?**].

2.1 Language Evolution

But no matter what mainstream programming language one looks at today, they all seem to have one thing in common, which also gets replicated by each new language that arrives: they are all focussing on describing computation. Computation executed

by a single core machine. Because for most of the last century, this was the problem we were facing.
However, with the advent of commodity multicore hardware [**?**] the problem domain changed. Today even a single a machine isn't a single machine anymore, but rather a group of multiple CPU cores that can operate and compute independent of each other. Unfortunately mainstream languages are still struggling with delivering language concepts and *semantics* that allow for utilizing this new hardware. In Fig.**??** one can see a very minimal multithreaded program written in Python, one of the most used languages today [**?**].

```
from threading import Thread

def count(n):
  while n > 0:
    n -= 1

t1 = Thread(target=count,args=(1000000,))
t1.start()

t2 = Thread(target=count,args=(1000000,))
t2.start()

t1.join()
t2.join()
```

Figure 1: Python multithreading example

As was shown by Beazley in [**?**] this multithreaded version is up to two times *slower* than the single threaded version and the reason for this is the so called *Global Interpreter Lock* (GIL) used by the Python interpreter which basically prevents multiple native threads from truly executing in parallel.
This is supposed to show how slow adoption and adjustment in the mainstream programming language market is happening, even for a problem domain that, as I would like to argue, is not even the most recent set of problems programmers today need to deal with.
This shows how important it is to reevaluate exisiting programming languages, their implementations and their methodologies whenever the underlying machine, system or problem domain changes dramatically.

Now one could argue that Python might have been a bad example and that other mainstream languages offer better adoption of multithreaded programming. Fig **??** shows another two examples of how threading is expressed in programming today, namely using the programming languages *Java* [**?**] and *Go* [**?**]. Altough Java and its virtual runtime environment, the *Java Virtual Machine* (JVM) offer the capability of true concurrent execution of threads, threads still only exist as a library feature, not as a language primitive. They are created by overwriting specific methods that are inherited from either another class or an interface and are used like any other *object* in Java, an object oriented programming language.
This is supposed to represent the approach that has been favoured in programming language design of how to deal with a changing environment. Every new or extending aspect is hidden behind the *function call*, a core feature of the language. The result is that the language itself and its semantics virtually never change, and the true meaning of a plain function call is expressed on an external napkin so to speak. This eliminates

all possibilities for the compiler or any other tooling to aid in the development process because to the compiler the function call that is supposed to start a new thread looks exactly the same as any other function call.

Example **??**b shows a newer language called *Go*. In Go threads are called *go routines* because their semantics are defined within the language itself and are created by the keyword *go*. On the one hand this allows any compiler or other tooling to automatically check, at compile time, whether certain rules of behavior are implemented correctly by the programmer but on the other hand it also delivers guarantees of said behavior to the programmer because these go routines will behave the same independent of the execution environment in which this code is run. Said behavior could vary when threads are only implemented as a library.

```java
class Demo extends Thread {

  public void run(){
    System.out.println("foo");
  }

  public static void main(String args[]) {
     Demo obj = new Demo();
     obj.start();
  }
}
```

(a) Java threading example

```go
package main
import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":",
            i)
    }
}

func main() {
  go f("goroutine")
}
```

(b) Go threading example

Figure 2: Threading examples in Java and Go

This shows that language designs do react to dramatic change of the underlying machine model. They include important aspects of the execution environment and hardware as core features of the language in order to provide a programming environment sweetspot: as close to the hardware to be still implemented efficiently but abstract enough so programming becomes intuitive, productive and less error prone. Just like the Turing machine hit that sweetspot on the theoretical side.

However, compared to the switch from single core to multi core CPU systems, the more dramatic paradigm shift in my view was the advent of the *internet*, or large scale networking in general, because it dawned what is now known as the *information age*. Although this is a rather broad term, describing global change in almost all areas of modern society, I believe these changes can also be seen in the area of computing and computation. Because in order to deliver new products like: the world wide web, internet search, internet advertising, social networks, social media, music streaming,

video streaming, messenger services, navigation/maps, e-commerce etc. companies had to build massive computer networks in order to either store massive amounts of data or to simply scale their product to millions or billions of users and although almost all of the current products, apps, or services could not exist without the computation that is implemented inside their core parts, it is *communication* that has become the main obstacle for building large scale *information systems*.

2.2 Distributed Objects

So how did the tools, the programming languages we use, react to this paradigm shift? Well, Fig.**??** shows an exemplary client and server implementation using a framework called *Java Remote Method Invocation* (RMI) which implements the more abstract request-and-response concept of *Remote Procedure Calls* (RPC).

```
public class ServerOperation extends UnicastRemoteObject implements
    RMIInterface {

    private static final long serialVersionUID = 1L;

    @Override
    public String helloTo(String name) throws RemoteException {
        return "hello to " + name;
    }
}
```

(a) Java RMI server providing the remote method 'helloTo()'

```
public class ClientOperation {

  private static RMIInterface look_up;

  public static void main(String[] args) {

    look_up = (RMIInterface) Naming.lookup("//localhost/MyServer");
    String response = look_up.helloTo("Frank");
  }
}
```

(b) Java RMI client calling the remote method 'helloTo()'

Figure 3: Java Remote Method Invocation example

The underlying concept that tries to unify object oriented programming and a dis-

tributed or networked system is often summarized as *distributed objects* [**?**]. As the names suggest, in this programming model, objects cannot only exists in the local address space but can also reside on remote machines with their methods hence being invoked remotely which by itself is not a bad idea.
But when one looks at how languages and frameworks implemented this new model one can see the same pattern as with multithreading libraries because the remote functionality and remote method calls are implemented by inheriting and overwriting certain methods on the server side and on the client side by instantiating a remote object and calling a remote method on that object. Unfortunately using the exact same syntax as for any local function call.
The difference of course is, that the remote procedure call, being remote, has to deal with different error and failure scenarios than any local method call. Hence it might throw some sort of remote exception which, from the view point of the client can be totally suprising, because syntactically there is no indication whatsoever that this is an action that involves the network or any remote machine.
The lesson here of course is the same as with concurrency support in programming languages because the underlying system changed in nontrivial ways but the distributed objects community tried to apply the same object oriented programming concepts and languages to this new domain, neglecting any additional and unique problems of this new setting and hiding all new functionality behind core language features, so that any new semantics were only available on a napkin and tool support was nearly impossible.

Unfortunately today, one could argue that the same approach is being used again in the context of highly distributed systems built after the microservice paradigm [**?**]. Fig.**??** shows an example of a Python binding for using the *Amazon Web Services* (AWS) service called *Simple Storage Service* (S3) [**?**], [**?**].

```
import boto

s3 = boto.connect_s3()
bucket = s3.create_bucket('media.yourdomain.com')

key = bucket.new_key('examples/first_file.csv')
key.set_contents_from_filename('/home/frank/first_file.csv')

key.set_acl('public-read')
```

Figure 4: Amazon AWS S3 storage example using the Python interfacing library 'boto'.

This service is supposed to provide a remote storage solution, by storing arbitrary data objects behind unique keys. As can be seen in the official example [**?**] depicted in Fig.**??** the unique names are being written using the same format as the widely sed UNIX file systems. The example shows how a local file is being uploaded to the service.

This showcases that Amazon S3 is mostly used as a service providing a remote file system to its clients.
But again, from the language perspective of the Python interpreter, these remote calls are just normal *local* function calls because the semantics and functionality of the S3 service only exist in the informal documentation provided by Amazon without any formal specification and without any chance of automated tool support regarding syntax (parameters) or semantics.
This might be acceptable from a business perspective because already existing systems can easily be extended to use these new services but this puts all the responsibilities in the hands of the programmer and also enables vendor lock-in because the semantics are defined by the company providing the abstractions and not only can they change them whenever they choose, but also other companies can have a vastly different APIs and semantics.

2.3 The Actor Model

In the same way that new programming language embrace the multicore nature of todays CPUs, I believe distributed programming and the languages that are used to build highly distributed systems should embrace the distributedness of the systems they describe and should provide special syntax and semantics for the unique problems inherent to the distributed systems domain and there are ideas and languages that support this view, as for example presented by Waldo et a. in [**?**].

On the language side an alternative paradigm to programming called the *Actor Model* [**?**] [**?**] has started to become widely adopted by the distributed systems community and implemented by languages like *Erlang* [**?**] [**?**] or *Pony* [**?**].
Since its perception in 1973 by Hewitt et al. [**?**], the actor model has received a great amount of attention in the academic community and has also been used to model concurrent computation as an alternative to the threading model. It is therefor also closely related to numerous process calculi like Hoare's *Communicating Sequential Processes* [**?**] and the Milner's *Pi Calculus* [**?**].
Erlang is an actor based functional programming language developed by Ericsson, a multinational telecommunications equipment and services company and was designed to be used by their telecommunications infrastructure.

The core concepts of the actor model as outlined in [**?**] and [**?**] can be summarized as follows. The actor model treats *actors* (or processes) as its universal primitives of computation. These actors are autonomous in the sense that each actors executed behavior is independent of the concurrent existence of other actors. However, in order to interact with others, actors can send and receive messages as

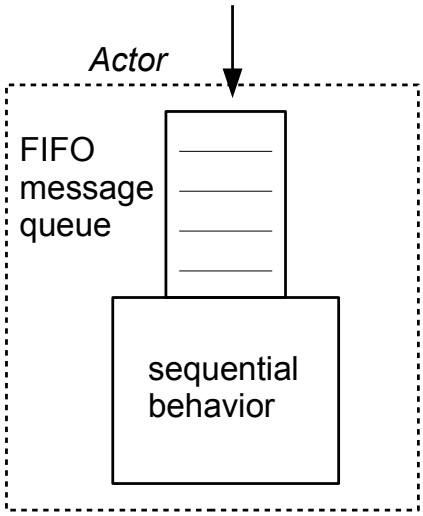

Figure 5: Concept of an *Actor* as defined by the actor model.

illustrated in Fig.**??**. So each actor consists of its single threaded behavior and a single input message queue.
The actor can now decide whether or not and how to react to its incoming messages and can also send messages directly to other actors. Its behavior itself though is single threaded. The concurrent nature of an actor based system lies within the actor model itself because of the actors being executed independent of each other.
This also means that the only means of communication, namely message passing, is *asynchronous* by default. The actor model iself does not define any receipt or acknowledgement mechanism for messages received by an actors input queue. Any such communication protocol must be implemented by the actors themselves, e.g. by programming their behavior to send an acklowdgement to the sender once a message is processed. This means that, using the plain actor model, a sender cannot only not infer whether or not its message was processed or perceived by its receiver, it can't even infer whether or not the message ever reached the receiver.

This seems ludicrous compared to the implicit safety and messaging guarantees promised by the *distributed objects* approach presented in chapter **??** where synchronous messaging is as *easy* as a local function call. But this seemingly pessimistic approach to messaging turns out to be acceptably close that what must be anticipated when dealing with messaging in highly distributed systems on the data center and cloud scale.
So again the actor model seems to provide a sweetspot because on the one hand it encompasses characteristic properties of highly distributed systems, i.e. virtually no message delivery guarantees, thereby forcing the developer to deal with these circumstances in her program structure and application design and on the other hand provides small, independent and single threaded units of computation in order to build more abstract and higher level distributed applications.

```
-module (send_recv).
-compile([export_all]).

serve() ->
  receive
    Request ->
      io:format("received request~n"),
      serve()
  end.

run() ->
  Pid = spawn(?MODULE, serve, []),
  Pid ! Request,
  ok.
```

Figure 6: Erlang example of how to spawn an Actor using the *spawn* function and how to use the messaging operator *!*

Since Erlang is an actor based language, Fig.**??** shows a small example. It spawns an actor executing the user-defined *serve()* function. Spawning an actor is done using the built-in *spawn()* function. The serve() function contains a *receive-loop*, another Erlang built-in indicated by the *receive* keyword. This loop listens for incoming messages and tries to match any incoming message to any of the given patterns. So it can be thought of as the combination of an implicit event loop and a switch-case statement.
In the case of this example any message that consists of only a single atom will be matched to the user-defined Request

pattern, triggering a simple console print and a recursive call to restart the receive loop. This is important because when the receive loop is entered it will wait for incoming messages indefinitely unless a timeout is specified. When a message arrives at the inbox that can be matched against one of the provided patterns it will be processed as programmed. When there are multiple messages in the actors inbox that could be matched against some provided pattern only one of them is being process and it is up to the actor when to reenter the receive loop in order to process the remaining messages.
To address the actor or process created by the call to *spawn()* a process identifier (PID) is returned by spawn, which is the only way to attain such an identifier. In order to send messages to this PID Erlang provides the messaging operator *'!'*.
This shows that message passing is an integral part of the core language, including its pessimistic delivery guarantees (none), and specified by the language itself. These semantics are therefor not loosely defined in some library documentation that could change any minute but are guaranteed to the programmer, independent of the language implementation or execution environment.

Unfortunately there is a down side to this approach. Since the actor model focusses so heavily on its actors, the result is that the actual description of the system happens implicitely and indirectly. It is more like the system *emerges* out of all the little steps and interactions of its components. To the effect that, given a running system of maybe a hundred or a thousand actors [**?**], how would one come to understand what the system as a whole is doing?
There is no other option than to just pick any actor and start reading its code. This means one is forced to reverse engineer the behavior of the entire system from the internal behavior of its parts which, needless to say, can be quite error prone and introduces nontrivial cognitive overhead. In other words, there is no way to take a look at the system from the birds-eye perspective because most people believe this implies introducing some form of global state which would be against any of the goals of the actor model with its asynchronous message passing and independent processes.

References

[1] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[2] P. J. Denning, "What is computation," *Ubiquity*, 2010.

[3] Wikipedia, "Programming language generations — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Programming%20language%20generations&oldid=753335536`, 2017. [Online; accessed 02-February-2017].

[4] B. Stroustrup, "What is object-oriented programming?," *IEEE Softw.*, vol. 5, pp. 10–20, May 1988.

[5] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, (New York, NY, USA), pp. 1–14, ACM, 1992.

[6] S. Olsen, "DailyTech - Here Comes Conroe." `http://www.dailytech.com/HereComesConroe/article3228.htm`, 2006. [Online; accessed 02-February-2017].

[7] TIOBE, "TIOBE Programming Language Index for January 2017." `http://www.tiobe.com/tiobe-index/`, 2017. [Online; accessed 02-February-2017].

[8] D. Beazley, "Understanding the python gil," in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[9] O. Corporation. `https://www.java.com/en/`, 2017. [Online; accessed 02-February-2017].

[10] G. Inc. `https://golang.org/`, 2017. [Online; accessed 02-February-2017].

[11] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn, "Programming with live distributed objects," in *European Conference on Object-Oriented Programming*, pp. 463–489, Springer, 2008.

[12] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *arXiv preprint arXiv:1606.04036*, 2016.

[13] A. Inc. `https://aws.amazon.com/`, 2017. [Online; accessed 02-February-2017].

[14] A. Inc. `https://aws.amazon.com/s3/details/`, 2017. [Online; accessed 02-February-2017].

[15] A. Inc. `https://aws.amazon.com/articles/3998`, 2017. [Online; accessed 02-February-2017].

[16] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Mobile Object Systems Towards the Programmable Internet*, pp. 49–64, Springer, 1997.

[17] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3, p. 235, Stanford Research Institute, 1973.

[18] C. Hewitt, "Actor model of computation: scalable robust information systems," *arXiv preprint arXiv:1008.1459*, 2010.

[19] `https://www.erlang.org/`, 2017. [Online; accessed 07-February-2017].

[20] J. Armstrong, R. Virding, C. Wikström, and M. Williams, "Concurrent programming in erlang," 1993.

[21] `http://www.ponylang.org/`, 2017. [Online; accessed 07-February-2017].

[22] C. A. R. Hoare, "Communicating sequential processes," in *The origin of concurrent programming*, pp. 413–443, Springer, 1978.

[23] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[24] G. A. Agha, "Actors: A model of concurrent computation in distributed systems.," tech. rep., DTIC Document, 1985.

[25] M. Ranney, "What i wish i had known before scaling uber to 1000 services." `https://www.youtube.com/watch?v=kb-m2fasdDY`, 2016. [Online; accessed 07-February-2017].

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den February 7, 2017 .....