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

Alongside the advent of multi-core CPU architectures and increased local storage capacities, large scale distributed systems based on commodity hardware, accommondated for in special facilities like data centers, have proven to be a viable tool for dealing with the ever increasing demand of compute power and storage.

One of the largest drivers of this demand is of course the modern web, including not only all the data accumulation that is taking place on social networks and social media but also providing different kinds of large scale services and platforms like maps and navigations but also e-commerce platforms which today include music and video streaming platforms like *Spotify*, *Netflix* or *Amazon Prime Music* or *Amazon Prime Video* or even transportation-on-demand platforms like *Uber*.

Additionally, the advent of artificial intelligence and machine learning created its own demand for ever increasing large scale neural networks and training data sets. A progression that is deeply coupled with the products of the modern web, providing not only recommondation or prediction engines for the users but also scheduling and load balancing decisions inside the systems, perfectly adapted to the given workload, which allow to fully utilize a given system to capacity without human intervention.

Therefore, in order to provide systems that allow to scale to millions or even billions of users, either in terms of storage, computing, training neural networks or simply content delivery and response times, large scale distributed systems have been constructed which pose new system design challenges not only regarding fault tolerance, network failure and fail over strategies but also in terms of simply how to describe, express and comprehend these vast and infinitely complex systems.

1.1 Goal of this work

To face these challenges and to analyze and explore the concepts needed by programmers to describe, construct and program these system, the following work presents the *Drift Programming Environment*, a collection of different components that allow the programming and orchestration of a distributed system of autonomous services by describing their data dependencies.

The goal of this work is to not only present each individual component of the final result, which includes an abstract, imperative and stateful coordination language, an immutable file system based on distributed message queues and an interactive user interface based on the syntax of Petri Nets, but also to introduce the reader to the set of projects and the people that created these projects, and their ideas and philosophies, that served as inspirations and guidance for the ideas and design decisions included in the *Drift* project.

1.2 Structure of this work

This is done by giving a short summary of the history and evolution of programming languages and their adaption to fundamental changes of the underlying hardware, especially the advent of distributed systems in chapter 2.
Chapter 3 then succesively introduces the related projects and tries to extract and hightlight their relevant aspects to the *Drift* project.
Chapter 4 then introduces the components of the *Drift Programming Environment* by first introducing the Drift language in chapter 4.2 including its implementation in form the of the Drift shell. This is followed by the introduction of the Drift system implementation in chapter 4.3 including an extensive explanation of the error model and distributed error handling procedure. Finally, chapter 4.4 presents the Drift UI prototype which features the Drift shell as introduced in chapter 4.3.1 as well as an interactive graphical representation of the current state of the entire system using the Petri Net syntax and an event time line that allows to effortlessly review previews system states.
Chapter 5 then summarizes the presented work, including all the presented components and their contribution to the overall system and chapter 6 thoroughly discusses possible improvements regarding the implementation of each individual component as it was introduced in chapter 4 as well as possible progressions of future projects.

2 Distributed Programming

In 1937 a paper titled *"On computable numbers, with an application to the Entscheidungsproblem"*, written by Alan Turing, was published [1]. 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 [2], 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 [3] with the first generation using binary machine instructions to todays languages

that allow for higher level programming styles like *object oriented programming* [4] or *functional programming* [5].

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 [6] 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.1 one can see a very minimal multithreaded program written in Python, one of the most used languages today [7].

```
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 [8] 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 2 shows another two examples of how threading is expressed in programming today, namely using the programming languages *Java* [9] and *Go* [10]. 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 2b 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.3 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).
The underlying concept that tries to unify object oriented programming and a distributed or networked system is often summarized as *distributed objects* [11]. 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.

```java
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()'

```java
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

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 [12]. Fig.4 shows an example of a Python binding for using the *Amazon Web Services* (AWS) service called *Simple Storage Service* (S3) [13], [14].
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 [15] depicted in Fig.4 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

```
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'.

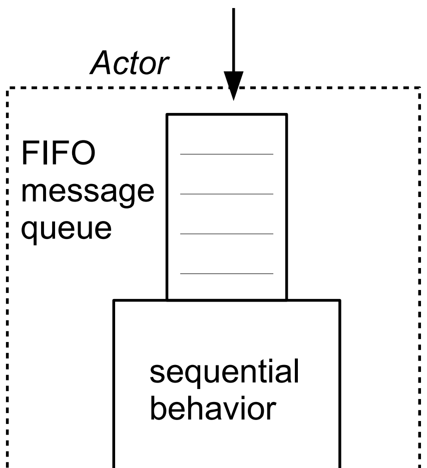
syntax (parameters) or semantics.
This might be acceptable from a business perspective because already existing systems can easily be extended to use these new services but this puts all the responsibilities in the hands of the programmer and also enables vendor lock-in because the semantics are defined 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 languages 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 [16].

On the language side an alternative paradigm to programming called the *Actor Model* [17], [18] has started to become widely adopted by the distributed systems community and implemented by languages like *Erlang* [19], [20] or *Pony* [21].
Since its perception in 1973 by Hewitt et al. [17], 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* [22] and the Milner's *Pi Calculus* [23].
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 [24] and [18] 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 illustrated in Fig.5. 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 2.2 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.
Since Erlang is an actor based language, Fig.6 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

```
-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 *!*

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 [25], 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.

3 Related Work

This chapter will present different ideas and projects which have served as motivation and inspiration not only for the *Drift* language but for the entire *Drift* project.
First two talks by Rich Hickey will be presented and the relevant aspects will be

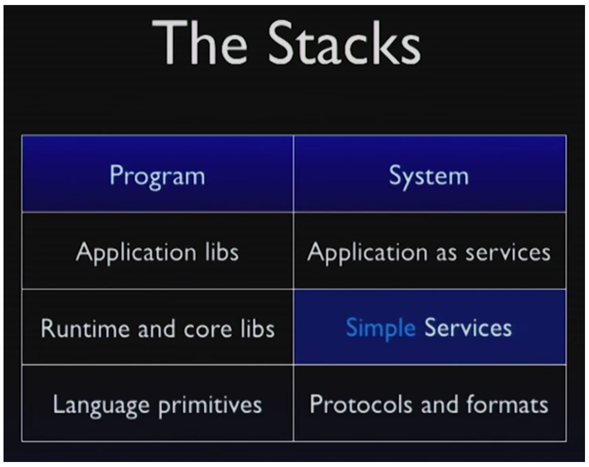
extracted. The first talk discusses the necessity and design of a “Language of the System”, especially in the context of highly distributed systems and the second talk introduces the concepts of *place oriented programming* and *value oriented programming* with an emphasis on the importance of immutable data in programming language design.
Then, as a representative of the *functional programming* approach, *Cuneiform*, a functional scientific workflow language and *HiWay*, its distributed execution engine will be introduced. Both of which are being developed here at the Humboldt University of Berlin at the department of bioinformatics (WBI) and have served as a major source of inspiration and opposing concepts.
After that, core concepts of UNIX-like operating system design and more importantly of the UNIX shell *bash* as one representative of a typicall UNIX shell are being introduced. Especially a mapping between the language constructs of the *bash* shell and core concepts of functional programming languages will be presented.
Furthermore a new distributed data base project will be introduced which tries to combine the functional programming concepts of immutable data and value orientation with the distributed data base context of data accretion, distributed logging, time series data and streaming.
Then the source code version control system *git* will be introduced because it shows how a seemingly stateful system on the user facing side can be implemented on top of an immutable, append only file system of objects addressed by hashes of hashes.
At last a talk by Bret Victor will be summarized in which he presents his idea of *immediate feedback* in the context of programming and tooling for programming. In this talk he showcases multiple prototypes which show how to interactively visualize code and how this could be used to aid the process of developing software. The most relevant feature presented will be a *time bar* which allows to interactively go back and forth through past states of the code and the actively running program.

3.1 The Language of the System

Rich Hickey is most widely known as the creater and *Benevolent Dictator for Life* (BDFL) [26] of *Clojure* [27], [28], [29], [30], a functional programming language and Lisp [31], [32] dialect. Clojure runs on the Java virtual machine to allow Java and Clojure interop in order to utilize the extensive collection of already existing Java libraries.
Clojure is therefor not a *pure* [33] functional programming language but should rather be considered a pragmatist’s compromise of having immutable data by default but allowing for easy I/O and statefulness whenever neededed.
Although Clojure’s *front end* (its syntax) looks a lot like Lisps syntax, which is notorious for its extensive use of parenthesis, Clojure’s *back end* (its implementation) focusses heavily on very modern features like *software transactional memory* [34] and *persistent data structures* [35] aimed at highly concurrent programming.
Hickey himself has been promoting these features as necessary upgrades to the object oriented view on programming in order to fully utilize multicore architectures and

is an avid visitor of conferences of programming language design. He therefor has a considerable collection of talks and lectures in which he explains his rationale behind his design decisions [36], [37].

One of his more relevant talks regarding the work presented here is called “The Language of the System” [1] held at *Clojure Con* 2012 [38], [39]. In it Hickey criticizes what has already been outlined in chapter 2.3, namely that distributed *systems* today are being built indirectly by programming individual units and components and then running them simultaneously, hoping that a distributed system will almost magically emerge but lacking any description whatsoever of that system as a whole.



The Stacks

Program	System
Application libs	Application as services
Runtime and core libs	Simple Services
Language primitives	Protocols and formats

Figure 7: Slide-excerp from the talk “The Language of the System” by Rich Hickey that draws parallels between the composition of (local) programs and distributed systems.

The reason for this, he claims, is that the languages and tools we use to build each individual component have been developed to do exactly this: describe *programs*, describe encapsulated self-contained blocks of computation. They don’t provide any means of describing how to compose these components, because they don’t provide any primitives for how they *communicate*.

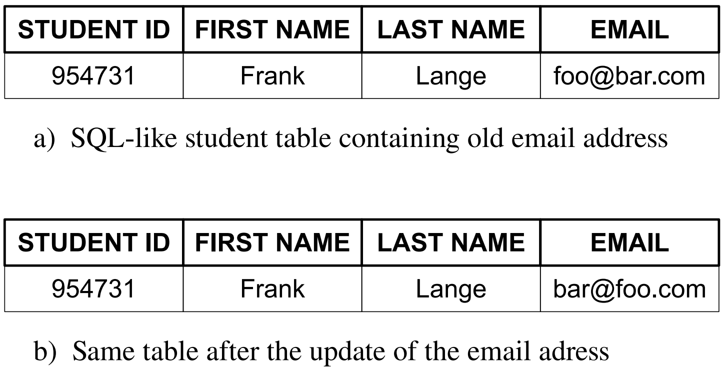
As can be seen in Fig.7 he draws parallels to how we haven been building our programs in order to derive similar properties for our system composition and description. For example he proposes that the language of the system too, would have primitives just like the programming languages that we have been using so far. But instead of having primitives of computation like built-in data types and arithmetic operators, in the systems context these primitives would be primitives of *communication* like communication protocols or data formats. On top of which simple core services need to be available in order to compose higher level application logic, just like core libraries like the GNU C library (glibc) [40] or any other standard library of any common programming language provide essential building blocks for programmers to compose their programs.

So the relevance of this talk to the work presented here is, that it sparked the question of whether or not it would be possible to build such a composition language, a language

[1]In the context of this talk and for the rest of the work presented here the word *system* does not mean *low-level* or *bits and bytes* as in operating system but rather means the system as a whole as from a bird’s-eye view and more as described by something like systems theory.

of the system, that allowed a global perspective without introducing global state and how such a language would look like.

The other talk by Hickey which is relevant to this work and which I would like to summarize here is called "The value of values" and was held at the GOTO Conference 2012 as well as the JaxConf 2012 [41], [42], [43], [44]. Probably being his most famous talk, in it Hickey bootstraps his views on programming and language design by putting *immutable data* at the center of programming and deriving the advantages of core functional programming concepts from that. He introduces two programming paradigms, namely *place-oriented programming* and *value-oriented programming.* A *place* in Hickey's view literally represents a place in reality. Something where one can put things and where they reside. If one replaces the thing that already resides in a place with some other thing, the information of that exchange is never recorded. Places don't record their exchange *history* so to speak. Every exchange *overwrites* the current content of that place. To the effect that the value received from querying a place depends on the point in *time* when the query was executed. The value of the place is inherently intertwined with time, which is what is generally known as *state.*
This represent the bulk of programming language history. It even represents the slots on the band of a Turing machine, which can be overwritten by said Turing machine with symbols defined in its alphabet. We have built hardware like memory and hard drives but also our software like data bases and file systems after this principle. They provide slots of storage which can carry a value but can be overwritten so the new value replaces the old and the value received from querying a place depends on the moment in time of the query.
In order to examplify this concept, Fig.8 shows a very basic representation of what could be a *SQL* table. This table stores data using the schema of *(ID, First Name, Last Name, Email)* which is supposed to represent a registered student. In Hickey's view, each cell of the table is a place, so if a student were to change her email address only the cell containing the email address would change, as shown in Fig.8b. But changing an email address does not invalidate the fact that there has been a point in time in which the student used the old email address, just like electing a new president of a country does not invalidate the fact that the country was once governed by somebody else.



STUDENT ID	FIRST NAME	LAST NAME	EMAIL
954731	Frank	Lange	foo@bar.com

a) SQL-like student table containing old email address

STUDENT ID	FIRST NAME	LAST NAME	EMAIL
954731	Frank	Lange	bar@foo.com

b) Same table after the update of the email adress

Figure 8: Place-orientation examplified by a SQL table, each cell being a place.

It is these *facts* that in Hickey's view represent what is also known as a *value* because

a value, once it exists, can never change. One of the most used and perhaps most accessable examples of values come from mathematics. The value '5' is just that, five. It cannot be altered or changed in any way. Five is always five. Mathematics allows for expressions like $x = 5 + 1$ which binds the expression which uses two values and an operator to the *name* x. One could now *evaluate* the subexpression $5 + 1$ which would yield another value, namely '6', but that would not change either the value '5' nor the value '1'. Five is always five, even when used in compound expressions.
This is exactly what *value-oriented programming* is about. It treats *data* as values, so data is *immutable* by default. Since data is immutable this means that it behaves like facts: once created (once it happened) it stays true forever and since operations on data can never change that data but only create new data, it becomes easy to record the history of steps that led to the creation of data. It's almost like one gets the history of operations on data for free, once data is immutable which is the direct opposite of place-oriented programming.

Both these approaches represent the extreme positions on a spectrum of language and system design. Unfortunately though, both these approaches in their basic form share a mutual problem: they don't scale. Place-oriented systems are very *space efficient* because places are singletons, they are unique and there is only exactly *one* of each place. This means that all producers and consumers or all readers and writers working on a place need to be mutually excluded from doing so. Access to a place needs to somehow be coordinated. One example for such coordination are *mutexes* (also called *locks*). However, this means that with an increasing number of accesses to a place, the contention for that place increases and the time spent on coordinating this access increases, especially if each logical operation needs access to multiple places in order to complete successfully. So place oriented systems scale in terms of space but not in terms of time.
As already outlined, value-oriented systems implement the exact opposite. Because values can never change, it is absolutely safe to allow multiple access to a value, or give each accessor its own copy. To stick with the mathematics example, it is common to see multiple functions using the value '5' as one of their input values. One calculating the square of its inputs the other something completely different. Although both are working with the same *value* they are working with their own copies of it, which don't have to updated or coordinated in any way, because five is always five. The value cannot change. This means that there's virtually no coordination overhead which allows value-oriented systems to scale in time, but because multiple instances or copies of the same value might need to exist, the system does not scale in space.

So the choice every language or systems designer needs to make, is which scalability bottleneck the system will be able to cope with. Is space going to be the issue, so it is important to be space efficient and feasible to pay for that efficiency with coordination overhead, because contention will be sparse? Or is contention going to be the issue, so that it is easier to supply enough space in order to keep the coordination overhead to a minimum?

As I would like to argue, in the context of future distributed systems, the latter approach seems more appropriate because until now, we have tried to simply port our place-oriented systems like file systems and data bases into the distributed context and it didn't seem to work out all that well. I argue that distributed systems are not so much about place-efficient computation, but rather about *communication* and *coordination*. Which means that a value-oriented system will keep the overhead for these things to a minimum and it is way easier to supply enough space, than to make a place-oriented system scale in the distributed context.

3.2 The Cuneiform Language

One area that features a wide variety of custom tailored languages which allow to describe a possibly distributed system from a bird's-eye perspective, is the field of *scientific workflows* [45], [46]. Especially in the context of *bioinformatics*, scientific workflow systems like Kepler [45], Taverna [47] and Pegasus [48], [49] have become an invaluable tool because they allow to utilize distributed systems in order to cope with the sheer amount of data that's been used and might also allow for massive gains from parallelization.
One language that is beeing developed here at the Humboldt Universität zu Berlin, at the department of *Knowledge Management in Bioinformatics*, is called *Cuneiform* [50], [51].
Matching the approach advocated by Rich Hickey and as presented in the chapter 3.1, Cuneiform is a functional workflow language with a formal foundation in the lambda calculus [52], [53] and with an emphasis on immutable data. Another key aspect of Cuneiform is its *foreign function interface* which allows to use already exisiting bioinformatics tools with any modifications. Most other scientific workflow systems force users to reimplemented their desired services using the abstractions given by the framework.

```
deftask untar (<list(File)> : tar(File)) in bash *{
  tar xf $tar
  list=‘tar tf $tar‘
}*

txt = untar(tar: 'corpus.tar');
csv = wc(txt: txt);
result = groupby(csv: csv);
result;
```

Figure 9: A shortened word count example using the functional workflow language *Cuneiform*

Fig.9 shows a shortened and small word count example, taken directly from Brandt et al. [50]. First this example shows how a Cuneiform task is defined using the *deftask*

keyword. The task named *untar* is supposed to deliver the same functionality as the widely known UNIX tool *tar* and therefor expects a *file* as input and returns a *list of files* as output. Showcasing the foreign function interface capabilities, one can see that it is specified that the body of this task will actually be written in *Bash*, the language of the same-named UNIX-like operating system shell.
The actual workflow description is shown in lines 6-9. Since Cuneiform uses *lazy evaluation* the first lines 6-8 don't actually trigger any behavior of the system. They only describe the workflow. Line 9 shows the query syntax of the system, which will finally trigger the actual execution of the specified workflow and print its result.
As one can see the results of the called tasks are bound to names using the = operator syntax, which in an imperative language might be called the *assignment operator*. Since Cuneiform follows the functional programming paradigm, these name bindings, rather than being assignments, are later dissolved using textual substitution.
Each name in the example work flow is used by the following task invocation as an input, therefor creating a zick-zack like pattern which basically describes a sequential data dependency between these tasks.
This shows that the coordination expressed via Cuneiform describes the data dependenies and therefor the *data flow* between tasks which cannot only be a sequential flow, but rather can branch out and join again in any possible way, describing a *directed acyclic graph* (DAG).
Since Cuneiform uses lazy evaluation, lines 6–8 build the DAG which can then be optimized or minimalized when the actual query operator in line 9 triggers its execution.

The next chapter will try to show, how these core concepts of a functional language and its data flow describing qualities can be mapped or emulated by the language used in the common UNIX shell.

3.3 The UNIX shell

Much can be said about the The UNIX time sharing system [54], [55]. It was developed in the early 1970's by Dennis Ritchie and Ken Thompson and featured many new concepts and abstractions which are still widely used today, like the file system and its naming and file descriptor concepts, inter-process communication mechanisms and the command line interface often referred to as "the Shell" [54].
It was also written in a new programming language developed by Ritchie and Brian Kernighan, called the *C* programming language [56]. So it delivered exactly what today is missing in the distributed systems community: a complete stack.
As shown in Fig.10 the operating system and its kernel API are positioned at the bottom of the stack. Representing the foundation by providing basic system primitives in terms of storage and persistence (memory, file system) but also in terms of communication (file descriptors, sockets, pipes) although it can be said that all of these concepts are unified behind an universal channel concept, namely the file descriptor. Hence the proverb "In UNIX, everything is a file".

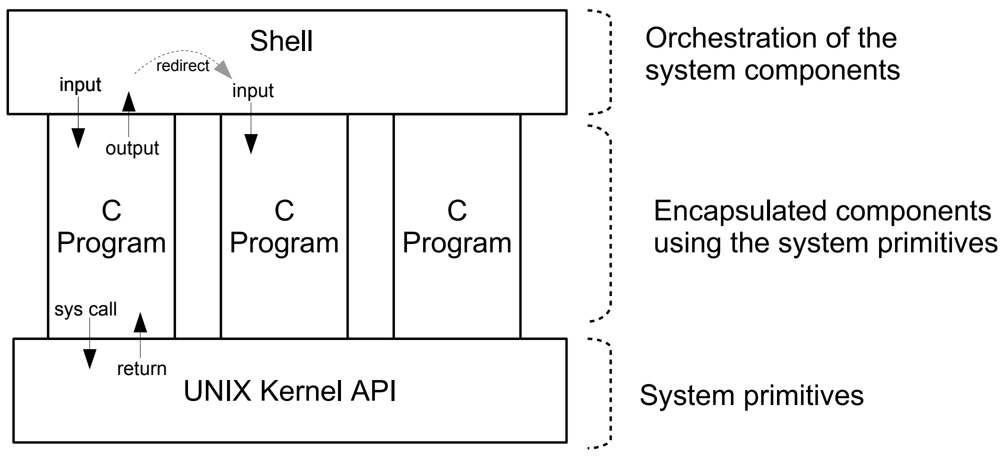


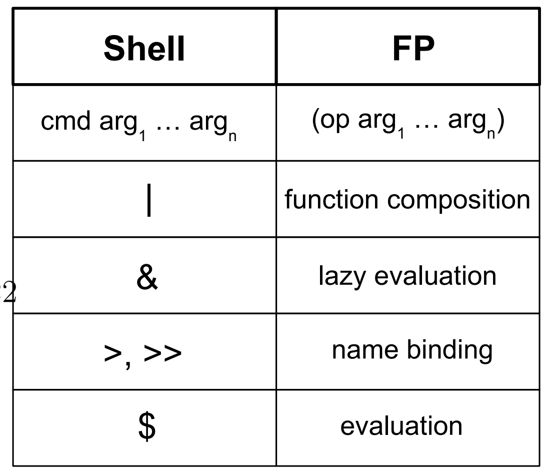
Figure 10: Conceptional stack of the original UNIX system

The system components are instances of the concept of a *process*. Encapsulated units of program instances, using virtualized resources like CPU and memory without any knowledge about other processes and requesting the system's primitives and services via so called *system calls*.

On top of the stack lies the shell, interfacing with the user. Using a different language than the system components, its purpose is to allow the user to control and orchestrate the system by starting programs (creating a new system component by initiating a new process) and by either persisting the output of the program to the file system or chaining subsequent program calls, redirecting the output of one program to be the input of the next program in the chain (*pipe mechanism*). Although it is possible for processes to directly talk to other processes via inter-process communication mechanisms like *named pipes* or *sockets*, the default mode of operation is that each process receives its input data from a designated input channel called *standard in* (stdin) and can choose between two designated output channels, named *standard out* (stdout) and *standard error* (stderr), which direct the output back to the shell who created the process.

This means that each component doesn't know and doesn't have to know how the whole composition, of which it is being part of, looks like. It only knows where its inputs come from and where to produce its output to. In my opinion, this bears much resemblance to the basic notion of a function and function composition in functional programming. Also the notion of autonomous processes seem very similar to what is today known as *Actors* in the *Actor Model*, as introduced in chapter 2.3.

In fact many of the core features and language constructs of a modern UNIX shell like the *Bourne Again Shell* (bash) [57] can be mapped to concepts used in languages following the functional programming paradigm, as can be seen



Shell	FP
cmd $arg_1 … arg_n$	(op $arg_1 … arg_n$)
|	function composition
&	lazy evaluation
>, >>	name binding
$	evaluation

Figure 11: Mapping of concepts from the

from the table shown in Fig.11. First of all the command syntax of the shell as shown in [54] follows the same prefix notation as used in Lisp [31] with the operator first and the following space-separated operands, e.g. `add 3 5` in the UNIX shell and `(+ 3 5)` in Lisp. Another concept is the widely known *pipe operator* `|`. This operator redirects the data received from the default output channels of the program on the left hand side to the default input channel of the program on the right hand side. In the same way how function composition in functional languages as well as plain mathematics *forwards* the result of the inner function as input to the outer function in an expression like $f \circ g$ which can also be written as $f(g(x))$. However, the disadvantage of this mathematical notation is that the data *flows* from the inside to the outside, or from the right to the left. When using the pipe operator the whole statement becomes $g \mid f$ and the data flows from left to right, just like most western cultures read any other form of text.

The *ampersand operator* `&` signals that the spawned process should be run in the background. This means that a new shell prompt is immediately available for the user and new commands can be entered and processed. Leaving the `&`-operator when starting a program blocks the shell for the duration of running the program and prints its output to the shell whenever output is made available by the program. The `&`-operator is useful in order to spawn multiple programs but since no output is shown [2] this could be seen as telling the shell everything that needs to be done before actually querying any results which is exactly what lazy evaluation does in functional programming languages.

The single and double angle bracket operators $>$ and $>>$ express that the output of the spawned program shall be redirected into a file, e.g. `add 3 5 > result.txt`. In order to see why this can be thought of as a name binding one needs to take a closer look at the concept of a *file system.* Of course, many file system implementations are highly complex and use different kinds of strategies and algorithms in order to provide high performance read and write access to storage hardware like hard drives, tapes or flash drives. This implementation side of the file system is what I would like to call the *back end* of a file system and is of less concern for this matter. However, most UNIX based file systems share a common *front end* or user interface which is also defined in [54] which uses the notion of *files* and *directories* in order to create a name tree and defines the naming scheme of this tree by putting a `'/'` as its root and inserting another `'/'` for every level in the tree. So a file named *c* which is contained in a directory called *b* which itself is contained by another directory called *a* would have the name `/a/b/c`. This naming scheme guarantees that each and every file has a single globally unique name because files are leafs in this name tree and there is only a single unique path

[2]In some implementations the output is printed whenever it arrives, interleaving with the current user input

to reach each leaf [3]. So from the user perspective, given such a unique name, the file system fetches the *unstructed* data that is referenced by the name, which is exactly the user facing behavior of what today is most widely known as a *key-value store*. Therefor writing data to a file can be seen as setting the value of a key in the key-value store but since keys are nothing more than unique names, this can also be seen as binding the name of the file to its content, in the same way that a reference points to its data. Lastly, the shell features the `$`-operator which is mostly used to query the content of shell variables, e.g. `echo $FOO`. Since the shell interpreter uses *eager evaluation* [58], i.e. immediately executing each command after its been entered by the user, it is not possible to put commands into variables and have them executed only when the content of the variable is querried, e.g. `FOO=‘cat words.txt | grep dog‘`. But following what has earlier been said about the `&`-operator, *if* the `&`-operator can be thought of as lazy evaluation, then the `$`-operator would be the query operator which triggers command execution. In the same that the `result;` statement in line 9 of Fig.9 querries the result of the described work flow.

This feature mapping shows, that the basic UNIX shell can be thought of as a very basic functional language given only minor modifications and it also challenges the popular belief that imperative languages and functional languages live on opposite sites of the language spectrum.
There is however a very fundamental problem with using the UNIX shell, that is not fixed by interpreting its features as functional and that is of course its statefulness in regard to the underlying file system. Since the file system is provided as a system primitive by the kernel, the shell, being only yet another process running on the system, has access to the exact same file system as any other process running on the system. This means that concurrent access on the same file leads to contention about its content and is a fundamental problem that needs to be dealt with. Unfortunately the original design, though aware of the problem, did not propose any solutions [54, chapter 3.6]:

> There are no user-visible locks in the file system, nor is there any restriction on the number of users who may have a file open for reading or writing; although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice, difficulties do not arise.

In that sense, files behave like *places* as introduced in chapter 3.1 but without providing coordination mechanisms. Since the UNIX shell *language* can easily be mapped to functional language concepts, it would make sense to convert the underlying data model from a place-oriented into a value-oriented one by introducing some sort of an *immutable file system* in which files are represented as *values* instead of as places. Interestingly enough, this idea is currently explored by a new project coming not from the file systems community but rather from the data base community, as will be explained in the next chapter.

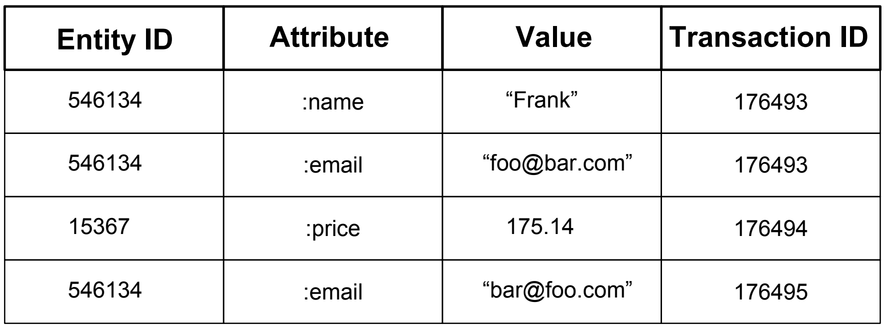
[3]ignoring more advanced features like symbolic links

3.4 Datomic - The Database as a Value

Datomic [59] is another project started by Rich Hickey and is now developed and marketed by *Cognitect Inc* besides Clojure, a Lisp dialect for the JVM, originally conceived by Hickey, as was introduced in chapter 3.1.

Instead of being another functional programming language, Datomic is a distributed data base, delivering classic data base guarantees and features like ACID transactions, joins and a logical query language. The main difference however, is that Datomic is based on the value-oriented idea of immutable data and not based on the idea of mutable cells in a table (as demonstrated by Fig.8 in chapter 3.1) as explained in [60] and [61]. This is why one of Hickey's talks about Datomic [62], [63] is called *The Database as a Value.*

As introduced in chapter 3.1 Hickey views data values as facts. Therefor a data base's role of storing data is nothing more than the accretion of facts, in the same way that history is an accretion of facts, like "the king died" or "Today I moved to Berlin." and even though new facts can be recorded concerning the same subject or *entity*, these new facts do *not* invalidate that there was a point in time in which the old fact was true. New facts do not overwrite old facts.



Entity ID	Attribute	Value	Transaction ID
546134	:name	"Frank"	176493
546134	:email	"foo@bar.com"	176493
15367	:price	175.14	176494
546134	:email	"bar@foo.com"	176495

Figure 12: The information model schema used in Datomic.

So in order to talk about facts at different points in time, Datomic uses a single schema information model which defines the structure of every recorded fact. As shown in Fig.12 every fact is a 4-tuple consisting of an entity ID to uniquely identify subject for which a new fact is recorded for, the attribute of the entity for which this is new information, the actual value itself that we want to record and a transaction ID that serves as a logical time stamp in order to allow facts to be recorded at different points in time. The idea of logical time is of course in reference to what is known as a *Lamport Clock* [64] and also *Vector Clocks* [65], [66] and has been used by other distributed storage systems like Google's *Bigtable* project [67].

In contrast to the example shown in Fig.8 in chapter 3.1 the example shown here in Fig.12 tries to show how a possible snapshot of the distributed log of facts might look like in Datomic. As one can see the first transaction (#176493) sets two attributes of the same entity, namely its name and email address. The next transaction sets

the price of some completely different entity and then another transaction is issued concerning the already existing entity from the first transaction, recording a new email address for that entity.
This of course means, that any reader still reading the old email address won't be interrupted and can still perceive his outdated view of the world and any new reader can decide whether he wants to receive the most recent value or the history of a value. This lends itself to using timed window operations in order to select only a fraction of the history and brings this system closer to the field of data streams and event stream processing. However, this means that the accretion of facts does not scale in space but needs virtually no coordination overhead and therefor scales in time, as has already been hinted at in chapter 3.1.
The relevant aspect of the Datomic project for the work presented here, is the fact that Datomic, despite being a data base, does not provide any storage capabilities. One needs to supply a storage solution on top of which Datomic can run. Interestingly enough, *any* already existing storage solution, from well known SQL data base systems, to No-SQL systems and key-value stores can be used to host Datomic, because the schema enforced by Datomic can be mapped to any known way of structuring data. Take a key-value store for instance. In order to store the values saved in Datomic the key would simply be the 3-tuple of $(Entity, Attribute, Transaction)$ and the value would be just that, the value.
This means that the key insight of Datomic is not to be a full fledged distributed data base in and by itself, but rather to be a very thin data interface that creates *the illusion of immutable data* on top of already existing distributed but mutable, place-oriented storage systems.

I guess it is safe to say, that this is Hickey's most important message: that functional programming at its *core* is not about functions being first class citizens, not about monads or type classes, not about partial application or currying, not about implicit parallelization, independent term evaluation or lazy evaluation. It's about *immutable data* and nearly every important aspect about functional programming can be derived from this single fact.
Unfortunately Hickey draws another conclusion from that, which I would like to argue is debatable, namely that this immutability should be propagated all the way up to the user interface (the language). The next chapter will show an example that does not follow this approach and the following chapters will try to explore how to build the illusion of mutability on top of an immutable system just like Datomic creates the illusion of immutability on top of a mutable system.

3.5 Git

Git is a so called *distributed version control system* [68], [69] created by Linus Torvalds [70] in order to help the development process of the Linux operating system kernel, also created by Torvalds. Since first conceived in 2005 [71], *git* has become one of the most used tools in modern software development today and very many things can be

said about how to effectively *use* git in order to organize source code versioning and the whole collaborative software development process [72].
Unfortunately the information about its interal structure and workings is disproportionally rare compared to the massive amounts of tutorials and usage guides. However, it is the combination of the internal structure of git and its user interface that is of relevance to the work presented here.

Since git was originally conceived to be a version control system for source code and source code is most widely stored in files, git operates on the file level and tracks changes to lines in files. So from the user perspective, these are normal file system files which are edited and stored just as any other files. Git does not interfere with any operations of the underlying file system. It is therefor completely invisible during the file editing process.
After a change of a file has been saved (persisted to the underlying file system) the user can issue git commands, e.g. using the git command line tool, in order to instruct git to track and *note* these changes. In order to advance the git repository to a new state, these changes need to be *commited*. This will create a new vertex in the history log of git and will advance the current head pointer of the git repository to this new version. It's important to note, that there is no fixed commit size. Commits can contain any number of changes, even of different files. It is up to the user to decide the *distance* between steps.
Git then stores the content of the tracked files as so called *blobs* and addresses them by the hash of their content. Therefor behaving like a *content addressable file system* underneath. In order to store the file and directory structure a tree structure is used in which blobs represent files and subtrees represent directories, as shown in Fig.13 which was taken from [73]. It's important to note that if a file is changed by the user and that file is then tracked and the change committed to git, the hash of the content of the file will be different compared to the earlier version of the file. Therefor git will store the complete file again, now with its new content and under the new hash. Only in later stages of the repository will git use *delta encoding and compression* in order to eliminate duplicate content, i.e. when pushing to a remote repository. Keeping every single version of file identified by its content hash is what allows git to be able to go back in time and fully restore *any* version of the repository ever committed.

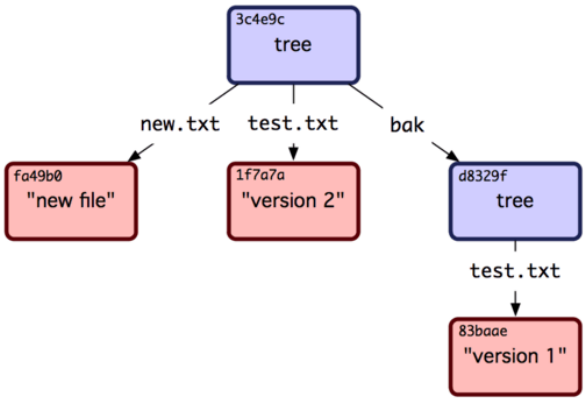


Figure 13: Example of Git's internal tree structure based on content hashing.

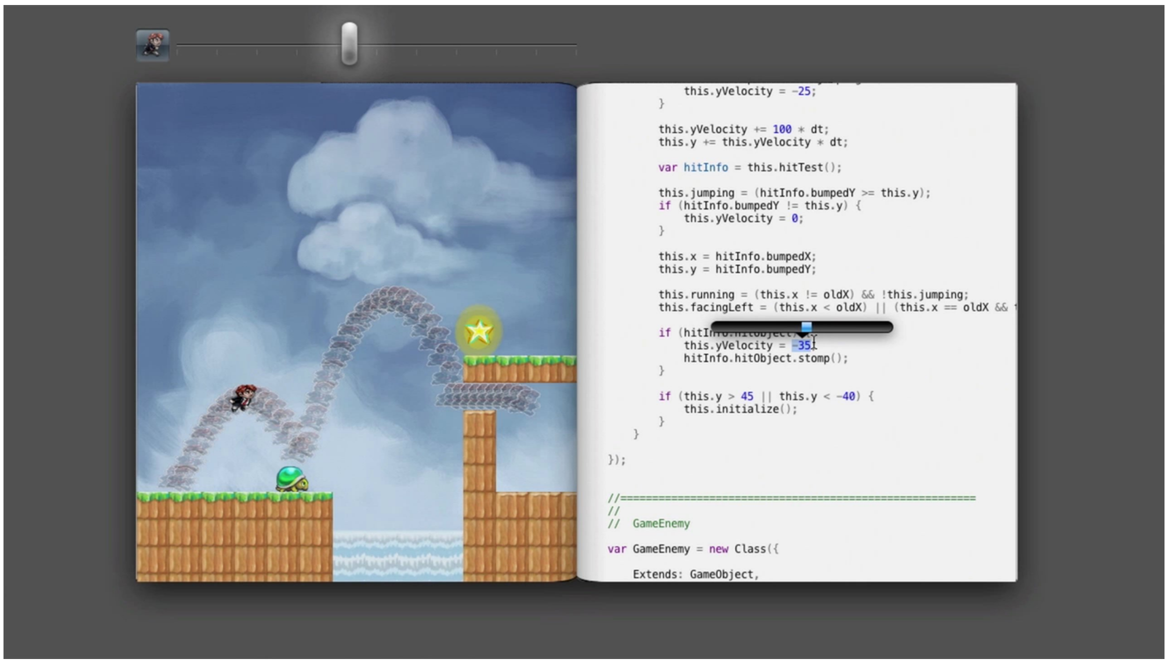
This is all completely hidden from the user. From the user perspective the file that she just changed, really changed. Because git does not interfere with the underlying file

system a change in a file is persisted and permanent. It's the user instructing git to take a *content snapshot* of the current state of the repository containing the content of all its files and directories when issuing a commit command.
So the content snapshot becomes an immutable *fact* and is appended to the internal repository history. Just like a fact is appended to the fact log that is Datomic, as introdudec in chapter 3.4. But since the file really changes for the user, she always immediately sees the latest state of the world but has the possibility to check out earlier versions of the world.
Therefor the user interface is stateful. If a file is deleted it's gone from the directory. But it is possible to recreate the directory state in which it existed, including its content, if a snapshot of that state was recorded. This means that it is possible to change a file and then revert the change or to create a file and then delete it again without git ever noticing. The user has full control of the history of the repository and needs to decide which intermediate states of the repository become immutable facts by instructing git to take a content snapshot. But since the changing of the current files and also the recreation of earlier versions happen in the extact same directory, the user experience feels more like working in a stateful environment and persisting the current *state* down to the immutability layer. It feels like a stateful system on top of an immutable system, instead of just dumping backups to a data base.
So it is not necessary to stick to immutability all the way up to the user interface in order to benefit from the advantages of an immutable system because user actions aren't always final. They don't always represent irreversible facts that should always stay true. As I would like to argue, most people iterate when working. They have an idea that seems plausible, implement that idea and then realize that they haven't thought of this and that, or something else doesn't work out as planned. So they change, adjust and iterate until a satisfying solution is found or the necessary resources for doing so have run out. Which is exactly what is best represented by a stateful environment.
But there will be intermediate steps that seem significant, and these can be recorded as immutable facts in order to allow for backtracking or to serve as documentation or as cached results for other work in progress that somehow arrives at the same intermediate result.

3.6 Immediate Feedback

After having introduced the notion of immutable data and its advantages in systems design but also the usefulness of state and where it can be a viable tool, the last piece that is needed in order to introduce the *Drift* project is how to interface with the user. As introduced in chapter 3.2, scientific workflow languages and systems deal with the need of the user to compose complex work flows from individual parts while maintaining a bird's-eye perspective of the whole system. As this can be done textually, some languages even choose a graphical notation, e.g. YAWL [74].
Although the visual representation of the system will be discussed in later chapters of this work, one of the most interesting aspects of interface design today in terms of

describing and composing the system is what has been described by Bret Victor as *immediate feedback.* [75], [76], [77].
After having worked at Apple Inc. as a *Human Interface Inventor*, Victor gained attention in the interface design community mostly through his talks, namely *Inventing on Principle*, *Stop drawing dead fish*, *The Future of Programming* and *Media for Thinking the Unthinkable* [77], [78], [79], [80].
Especially in his first talk *Inventing on Principle* he introduces his own principle which runs like a common thread through all of his work: the immediate coupling between the user and the system. Although not limited to the area of computer systems and programming, Victor critizises the cumbersome and stop-and-go nature of modern programming using text editors and compilers.
In his view, most programmers use a text editor in order to write down a program, immitating and emulating the steps a computer would take in their head, when specifying the program. Then, when the programmer is optimistic that his program will provide the desired result it gets compiled, showing any possible syntactic mistakes only after the compilation and not during the phase of actually writing the program. When the compilation succeeds, the program is executed and if the run time behavior differs from what was expected, the machine is stopped and the programmer is back at the text editor trying to figure out his mistake by emulating the execution step by step in his head. Tools like debuggers make this process more accessible by showing for instance the contents of memory addresses, but they do not eliminate the cognitive overhead of emulating the steps by the programmer in order to figure out *why* the program behaves the way it behaves. Debuggers only provide a framework for doing so. To Victor this work flow is extremely slow, error prone and opposite to any other creative work flow like drawing or playing an instrument. Because when drawing, the artist immediately sees the effect of applying the chosen amount of paint to the canvas, therefor showing a feedback loop between the actions of the user and the response of the system. So in order to create this feedback loop and to allow for an immediate action-response cycle, he presents a programming environment prototype, as shown in Fig.14.
This prototype showcases how the development of a 2D video game could look like. As one can see the coding environment not only shows the program code on the right hand side, but also shows a live and *running* version of the code on the left hand side. So that whenever the user changes the value of, say the gravitational attraction in the game, the effects of that change can immediately be seen and experienced by playing with the Mario-like figure. Unfortunately this still does not completely solve the problem of having to emulate the actions of the computer as a programer because what is the correct value for the gravitational attraction or velocity in order for Mario to enter the little gap shown in Fig.14? Although the change is immediately shown in the running program, the programer would still have to use a trial-and-error approach of figuring out the correct value by doing the jump in order to see whether or not the chosen value works as expected.
In order to eliminate this tinkering and to close the feedback loop even further, the environment features a *time bar* which records the history of the trajectory of the



```
            this.yVelocity = -25;
        }

        this.yVelocity += 100 * dt;
        this.y += this.yVelocity * dt;

        var hitInfo = this.hitTest();

        this.jumping = (hitInfo.bumpedY >= this.y);
        if (hitInfo.bumpedY != this.y) {
            this.yVelocity = 0;
        }

        this.x = hitInfo.bumpedX;
        this.y = hitInfo.bumpedY;

        this.running = (this.x != oldX) && !this.jumping;
        this.facingLeft = (this.x < oldX) || (this.x == oldX &&

        if (hitInfo.hitObject) {
            this.yVelocity = -35;
            hitInfo.hitObject.stomp();
        }

        if (this.y > 45 || this.y < -40) {
            this.initialize();
        }
    }

});


//=========================================================
//
//  GameEnemy

var GameEnemy = new Class({

    Extends: GameObject,
```

Figure 14: Screenshot from the talk *Inventing on Principle* by Bret Victor (14:13) showing a prototypical live-coding environment.

Mario-like figure, as shown on the top of the screen. So now the jump has to be recorded only once, and using *slide bars* in the code editor in order to sweep across the value spectrum of a variable, the environment can show how the trajectory of the jump would have looked like using the value currently set in the code editor.

So now the programer can slide across different values, finding the exact one that allows for the Mario-like figure to enter the litte gap below the star, without having to manually trial-and-error different values.

This concept of immediate feedback might seem alien in the context of imperative languages and ahead-of-time compilation but is actually not new in the context of functional programming.

Because functional programming languages won't easily allow for state and base their concept of computation on the evaluation of mostly side-effect free expressions, even Lisp [32] featured what would later become known as a *read-eval-print-loop* (REPL), an interactive shell that eagerly evaluated the expressions typed in by the user. Although such an interactive shell does not immediately allow for the fluent parameter tuning as shown by Bret Victor, it still delivers a more immediate feedback loop than ahead-of-time compilation and because of that allows for easier visualization of the current user actions and the resulting state of the system.

4 Drift

The last chapter, chapter **??**, introduced multiple seemingly unrelated software projects and also the people behind them and their ideas and principles. However unrelated these projects seem, they each contribute to the work presented in the following chapters, namely the *Drift* project.
The first idea that was introduced in chapter 3.1 was the need for a language of the system. A language that would allow to orchestrate a possibly distributed or even microservice based system from a bird's-eye perspective. In order to allow such a system's perspective without depending on global state in the distributed context and to keep the complexity arising from the combinatorial explosion of contending and interleaving interactions of the autonomous and independent components of the system to a minimum, the importance of immutable data in the form of value-oriented programming was introduced.
Chapter 3.2 showed a modern contender for such a language and system, namely Cuneiform: a functional scientific workflow language, utilizing most of the functional language toolbox like lazy-evaluation, single-assign name bindings and independent subexpression evaluation as defined by the Church-Rosser Theorem [81], [82]. However, using Cuneiform and similar systems means fully accepting the paradigm of functional programming and abandoning any imperative roots of programming, which seems rather radical given the wide adoption of imperative languages which shaped the way people think about programming for generations.
In order to look for alternatives, chapter 3.3 went back in time and took a closer look at a system that has already been around for decades, namely the UNIX operating system. The interesting aspect when it comes to system orchestration and coordination from a bird's-eye perspective is of course the UNIX shell. Given that the operating system allows for multiple independent processes to run seemingly autonomous, the shell not only provides mechanisms to spawn these processes but also to connect the data flow between them using well-known abstractions like pipes and files. Furthermore, the chapter 3.3 showed, that when taken seriously, the constructs offered by the shell language can even be mapped to functional language concepts as used by languages like Cuneiform. This revealed that the UNIX shell offers roughly the same capabilities in terms of orchestrating a system of independent processes *plus* being imperative, widely-known and interactive.
Especially the interactiveness delivering immediate feedback is something that provides multiple benefits as was introduced in chapter 3.6 and was something that for the longest time was second nature in the realm of functional but not imperative programming.
The "big problem" so to speak, when it comes to the design of the UNIX shell as well as the operating system itself and the idea of porting it to the context of distributed systems is of course the file system or more generally the data layer. Since its design follows the place-oriented paradigm as introduced in chapter 3.1, any subsequent implementations using distributed file systems have stuck to the same paradigm and therefor run into the same issues.
So in the same way that chapters 3.2 and 3.3 are supposed to challenge the widely

accepted believe of incompatiblity between functional and imperative language concepts, chapters 3.4 and 3.5 try to challenge the preconceptions of distributed immutable data systems and their implications. So chapter 3.4 introduces Datomic, a distributed immutable facts log when viewed from the user perspective, but when looked at from the implementation perspective, Datomic only creates the illusion of immutability by putting a thin immutability layer on top of widely-known and already existing mutable distributed storage solutions. Git on the other hand, as introduced in chapter 3.5, is the complete opposite. It provides the user with a mutable and stateful perspective on her own file system built on top of an immutable content addressable file system underneath.
This again challenges the widespread believe that immutability or statefulness are system design primitives that define the system to its core and must be applied throughout the whole system stack, up to the user interface.

So naturally the challenge arose, of whether or not it would be possible to build an imperative and distributed shell language with a flat learning curve using well known imperative concepts but with the same orchestrational powers and interactiveness as provided by functional languages on top of an immutable distributed file system in order to utilize the benefits of the value-oriented approach and to keep the contention and complexity of the whole system to a minimum.

The following chapters will present the current result of that challenge by first introducing the *Drift language*, a tiny and abstract microservice coordination language built on the concepts of names, namespaces and services. Then its distributed implementation including the immutable *Drift file system* will be presented, which provides the illusion of a file system by utilizing distributed message queues. Finally the *Drift UI* will be presented which is implemented as a web interface for any modern browser and contains not only the shell but also a visual live-representation of the system state using the well known Petri Net syntax as well as a time bar that allows to revisit old system states.

4.1 The Importance of Names

Developing a programming language can be an overwhelmingly ambitious task. Most often it's only that when one sets out to do so, that one realizes how many features even basic languages offer: arithmetic, different types of loops, conditionals, logical operators, built-in data types and a type system that allows for arbitrary user-defined types. Depending on the overall paradigm the list might continue with things like classes, objects, inheritance, polymorphism and generic types or things like type classes, higher-order functions, partial application or monads and do-notation.
Although it would have been nice to have at least scratched on some of these aspects, it became clear right at the beginning that most of these features would be left untouched given the scope and resources of the work presented here.
But that is not necessarily a bad thing. Constraints often times have the benefit that

they force resources to be spent on what really matters and so in that same spirit the goal for the Drift coordination language was to carve out what is *essential* to programming. It could almost be said that the goal was to find the smallest possible language that would not allow for any construct to be taken out and still be offering the same possibilities.

So what is the essence of programming?
Well that depends on what the user is supposed to be able to express in that language. If it's computation through calculation, built-in arithmetic operators like $+$ or $*$ might seem a good idea. If the user should be able to express things that model the real world, like in many simulation contexts for example, having constructs that represent real world objects and allow for equipping them with properties like color or weight could be another sensible approach, built on basic arithmetic. This would naturally lead to built-in data types for which these basic arithmetic operators are defined and would build up to a type system which could deal with arbitrary user-defined types and the operations defined on them.
But defining these structures, functions or classes and modelling their properties and functionalities is only one part of constructing a program because these structures in and by themselves have nothing on which they could execute their behavior on and are not connected in any way in order to produce a final output as the result of multiple functionalities chained together.
A program does not only exist as the mere sum of its parts but also of a description of how these parts interact and compose and where data comes from and where it goes which is what is mostly described inside the *main-method* which virtually all programming languages independent of their paradigm have in common because it serves as the universal starting point of a program.
Most languages use the same syntax and semantics for both purposes: for describing the individual components and for describing their often times more abstract orchestration. This is demonstrated by the examples shown in Fig.15.
The left hand side figure, Fig.15a, shows an example using the C programming language. In it, a new compound user-defined data type *point* is declared and defined, consisting of two integers representing the x and y coordinates of a point. Then a function is declared and defined which takes such a point as an input and returns the sum of its coordinates.
These are the blue prints of the components of this tiny system. Inside the main-methode an instance of a point is created and its coordinates are set to the values 3 and 5, which are part of the source code itself. Then a function call is issued using a copy of this point instance which creates a single instance of the mentionend function whose return value is then immediately printed to the screen.
So inside the main method the actual *business logic* of the program is described by composing and chaining instances of the described components and inserting data into the system and letting it flow through the components in order to produce the desired output.

```c
#include <stdio.h>

struct point {
  int x;
  int y;
};

int point_sum(struct point p) {
  return (p.x + p.y);
}

int main() {

  struct point p;
  p.x = 3;
  p.y = 5;

  printf("%d\n", point_sum(p));

  return 0;
}
```

(a) Component definition and orchestration in C.

```java
public class Point {
  private int x;
  private int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getSum() {
    return (this.x + this.y);
  }

  public static void main(String[] args){

    Point p = new Point(3, 5);
    System.out.println(p.getSum());

    return;
  }
}
```

(b) Component definition and orchestration in Java.

Figure 15: Examples of component definition and orchestration done in languages that use the same language for both tasks.

The same thing happens in the example on the right hand side, in Fig.15b, this time using the object oriented programming language Java. In this example a class *Point* is defined, also using two integers to represent its coordinates. The class defines two methods: a constructor method for ease of instantiation and a *getSum(int, int)* method to again calculate the sum of the coordinates.
As with the C example on the left, this plain class definition by itself would not make for much of a program. The business logic is again defined in the main-method, where an instance of the Point class, a Point object, is instantiated (again with data contained inside the source code file itself) and the result of a call to its instance method *getSum* is printed.

However, it is possible to disentangle the component description from the higher level component composition. One example of this has already been introduced, namely the UNIX stack as shown in Fig.10. In this stack, each component is a UNIX process, dealing only with the task of processing the data its been given and producing its

output respectively. The higher-level business logic of the system is then defined using the shell which can be used to instantiate components (processes), just like objects are instantiated in Java, and describe the data flow between them. The difference is, that the shell language, the *coordination language* is completely different compared to the *computation language* used to describe the computation done by each individual process.
So which of the two, computation or coordination, is actually more relevant in the context of programming a distributed system? Well, how would one distribute a computation like $1 + 2$? Computation in and by itself has something local about it because the operator needs to have perceived or received its operands in order to carry out its computation. The *distributedness* actually emerges because although each individual operation is carried out on a local machine, the actuall composition of multiple operations and data flow through them spans multiple locations, i.e. multiple machines.
So as I have already tried to argue in chapter 2.1, for the context of distributed systems, it's not so much computation that defines the programming of distributed systems (because of computation having this inherently local aspect to it, we can for the most part reuse the already existing computation languages) but rather *coordination* and *communication* between the components and operations residing on different machines.

So the question of what is the essence of programming changes in the context of distributed systems and therefor for the context of this work to: what is the essence of composition?

To me, as far as I can work out, the essence of composition is simply *data* and *functionality*. Whether functionality is provided by plain functions modelled after the example of mathematical functions, or with objects that encapsulate functionality via methods, or with actors or processes or microservices is not really important.
I believe the user needs these two things: she needs to be able to identify and talk about her data and she needs to be able to identify and invoke her services with that data. This also includes identifying the data that might be the result of such a service invocation.
So what's the most natural way for *people* to identify things? By giving them *names*. Names that carry *meaning* and therefor provide semantics (for humans).

Meaning is an interesting word and one can easily find examples where it is ascribed to mechanisms that do not really provide it. For example one of the fundamental books on programming language semantics and type systems, *Types and Programming Languages* by Pierce [83] contains the following quote by Mark Manasse on page 208:

> The fundamental problem addressed by a type theory is to ensure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.

In this quote, type systems are identified as the source of meaning for programs and there is some truth to that. But I would like to argue that it is not the type system itself that provides the meaning, but rather the *names* of the types that do so.

```
void runServer(ServerSocket serverSock) {
  serverSock.listenOnPort(9000);
}
```

```
void runServer(Uhjgj abcdeg) {
  abcdeg.inufdhg(9000);
}
```

```
void runServer(143 abcdeg) {
  abcdeg.inufdhg(9000);
}
```

Figure 16: Example showing the importance of *names* for human understanding and reasoning about code.

Consider for example the three versions of the same function in Fig.16. The first version represents the kind of code that is used in software development today, using meaningful names not only for the data types but for the variable names as well. This makes it easy for humans to understand why a server socket has a method that triggers listening on a port and why this method receives a number as input. Most of the meaning is conveyed through the names, if chosen appropriately.
In the second version the names are obfuscated but still used as expected by the compiler. This will happily compile, but most of its meaning is gone. The third version reduces the type identifier to a number. Most language grammars do not allow for type names to be numbers, so this won't compile but the type checking engine itself would probably still accept it. Because to the type checker, the type is just an ID and as along as it can check whether or not the type with the ID 143 provides a method with the used identifier and signature, it would not raise any flags.

The *meaning* is not conveyed by types. It's conveyed by the *names* of the types and of course the names of variables, methods, classes, objects, and so on.

```
int add_one(int n) {
  return (n - 1);
}
```

Figure 17: Example showing a bug that cannot be detected by the compiler or type checker.

Consider another example, shown in Fig.17. This example shows a simple function whose name suggests that it returns its argument incremented by one. Unfortunately its code does the exact opposite and returns the decremented input. This will also happily compile because *add_one* is a valid function identifier and *n - 1* is a valid operation on integers. But we as people will immediate spot the mismatch between the meaning of its name and its provided functionality.
So when one tries to reduce coordination down to its core parts, namely data and services, the essential abstract thoughts that can be expressed by the programmer are something like "do this on that", or "do this on that and put the result there", or "do this on that and then that and put the final result over there". Interestingly even on such an abstract and rather comical scale, these abstract statements can be categorized into two groups. They are either describing a functionality chain, where each component depends on the output of its predecessor and they all form a sort

of pipeline through which the data flows sequentially or the services are completely independent of one another.
So naturally the formulation of data and services forms a dependency tree or, depending on the allowed complexity, a directed acyclic graph (DAG). Which can be used effectively for visualizations as will be shown in later chapters.

4.2 The Drift Language

And so, in that same spirit, the basic concepts of the *Drift* language are *names* to represent data and *services* to represent functionality.

```
.> Cat mydata
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
.>
```

(a) Simple service invocation.

```
.> myresult = Cat mydata
.>
```

(b) Service invocation with result bound to a name.

Figure 18: Basic service invocation mechanism in Drift.

Fig.18 shows the basic concepts offered by the *Drift language* as implemented by the *Drift shell*. In the Drift language, service names are required to start with an upper case letter, whereas names representing data are to be started with a lower case letter. The $. >$ symbolizes the shell prompt. The first example of Fig.18a shows the most basic use case: a simple service invocation. Following the tradition of the UNIX shell, Drift uses the same order of command and arguments, namely $Service\ name_1\ ...\ name_n$. Depending on the called service it would also be possible to invoke services without any parameter names.
Since Drift is eager-evaluated the service shown in Fig.18a is immediately started. In this case it's a service called *Cat* in reference to the widely-known UNIX tool, which prints the content of given files to the screen. So the *Cat* service also returns the data of its arguments and since there are only names in Drift, the *Cat* services takes names as input and returns the data associated with them.
Since the data returned by the invocation of the *Cat* service is not bound to any name, it just gets printed to the shell. When all the data that is available behind the name(s) given to *Cat* has been printed, a new shell prompt is shown and further commands can be entered by the user.
Fig.18b shows the same service invocation but this time the result of the service call is bound to a name, using the assignment syntax as known from other imperative languages. Again the *Cat* service is invoked immediately but since its result is bound to a name, nothing is printed to the screen except a new prompt so that new commands can be entered.
In order to query the data that is represented by a name, Drift offers the $-operator, a designated query operator, as shown in 19a. This query operator will receive *all* the

```
.> myresult = Cat mydata
.> $myresult
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
.>
```

(a) Using the $-operator to query the data behind a name.

```
.> Cat mydata
Lorem ipsum dolor sit amet,
[cancel]
.>
```

(b) Canceling an ongoing data query.

Figure 19: Basic query and query cancelation mechanism in Drift.

data available behind a name and print it straight to the screen.

However, since services are invoked immediately and therefor executed immediately, it is possible that the service has not yet produced a single data item or has not yet finished processing and has therefor not finished producing all of its output. This means that it is absolutely possible that either no data or only some data is shown, depending on what's available when the user issues the query.

Drift is not an *all or nothing* batch system where the user either sees no output, or all the output atomically. It's a streaming system in which data is observable whenever it is available. This is based on the observation that batch processing, even distributed batch processing, is only *a special case* of stream processing and not the other way around, as was also proclaimed in [84, chapter 1]. This fits perfectly with the overall approach of immediate feedback because not only the commands of the shell are eagerly evaluated and because of that the services immediately spawned, but also the data produced by the individual components flowing through the system can be immediately observed by the user.

Of course it would defeat the purpose and reactiveness of the system if the user would have to wait for all the data to be finished, once querying a name. Therefor query cancelation has been implemented as shown in Fig.19b. When data is printed to the screen, either because the result of the producing service invocation was not bound to a name, or because a name was queried using the $-operator, the display of data can be canceled simply by pressing *q* and new prompt will be shown so that new actions can be taken.

Therefor printing data to the screen has not the role of presenting real results, like in many batch processing systems. It rather becomes a tool for the user to sneak a peek at what the system is currently doing in order to assess whether or not the system is working towards the desired result.

So where do all the available services and names come from? In the same way that the UNIX shell does not provide much built-in functionality except for only a very few built-in commands, the Drift shell also does not provide any services. Services need to be inserted into a *service registry*.

This is a designated service invisible to the user that the Drift shell assumes to be

available at all times. When a command is entered, the shell will consult the service registry about whether the service exists and whether its usage corresponds to its specification stored in the service registry.
However the shell does not *download* the required service in any way. It only fact-checks the service before issuing the entered command as a task to the back-end system, as will be described in chapter 4.3. If a service is unknown to the service registry, a *service unknown* error will be printed to the screen and a new prompt is shown.

The availability of names is different. One of the design goals of the Drift shell is to present the user with its own cloud so to speak. Therefor any Drift shell invocation starts a new *session* using a UUID as a unique session identifier. In the current implementation, this UUID is also generated and handed out by the service registry.
Following this principle, any session starts with a completely empty *namespace* containing no names. However, before the actual session starts and the entering of commands is allowed, the user is presented with a setup phase. In this *import phase* she can only use the *import* keyword to *upload* files from the local file system of where the Drift shell is run into the Drift system.
When the user has finished the import phase, the actual session begins and normal command input is possible. The *import* keyword and therefor any further imports are disabled.

```
Session: e3d78ad5-898...
.> import data.csv
.> import test.txt
.> q
--------------------
.> ls
  data.csv
  test.txt
.> result = Cat test.txt
.> ls
  data.csv
  result
  test.txt
.> export result as result
.> q
```

Figure 20: Example of a short Drift session.

Fig. 20 shows an example session including the import phase. Here two local files, namely `data.csv` and `test.txt` are uploaded to the Drift back-end. When the import phase is finished by pressing *q*, the normal session begins. This example also introduces one of the few built-in commands of the Drift shell, namely `ls`. These built-ins are again lent from the original UNIX shell and UNIX file system, so `ls` is of course used to list all the available names in the current namespace.
Now with these imported names and associated data, services can be invoked and their result can be bound to new names. In order to recognize whether a name resulted from an import or was later created during the session, only the imported names are allowed to contain dots and file endings like `.txt`, `.csv` or `.tar`.
In the same way that the import keyword during the import phase allows to fill the initial namespace with names, the *export* keyword allows to *download* the data referenced by a name to the local file system under a given name. In the current implementation this can be done at any time during a session but it could also be a valid alternative to put the export behavior into a seperate export phase before closing the session.

The local file in which the data is stored is placed inside a directory with the same name as the session ID, eliminating the possibility of name clashing between different sessions (of possibly different users). However, the user of a session is responsible for preventing name clashes from within a session when using the *export* keyword.

So far *names* and *services* and their basic usage have been introduced. Initial data can be uploaded from the local file system into the Drift back-end and imported as names into the initial name space of a session of the Drift shell. Resulting data from service invocations can be either printed to the screen or bound to names using the assignment syntax known from other languages and the namespace can be explored by using well known UNIX file system commands like `ls`.
However, there are services whose result is not just singular data. As has already been mentioned, one possible file extension for imports is `.tar` and so naturally one service which is able to deal with such imports is called `Untar*`, with the `*` on the right hand side being of significance because it indicates that the output of `Untar*` is not just data (a single file) but rather a bunch of names. Which is why the concept of *names* in the Drift language is expanded to also include *namespaces*, which directly correspond to directories in the original UNIX file system.
Since the result of an invocation of `Untar*` is a namespace, it wouldn't really make sense to be able to print it to the screen. But it would also not make sense to bind a bunch of names to only a single name, as was shown so far.
Therefor, the result of services returning a namespace must also always be bind to a namespace.

```
Session: e3d78ad5-898...
.> import comments.tar
.> q
--------------------
.> ls
  comments.tar
.> res/ = Untar* comments.tar
.> ls
  comments.tar
  res/
.> cd res/
.> ls
  c1
  c2
```

Figure 21: Example of a name space binding.

As shown in Fig.21 *namespaces* also start with a lower case letter, just like names but namespaces must always be trailed by a `/`. This and the additional `*` allow the interpreter to syntactically verify that the issued command is at least valid in terms of its basic format. If the format is invalid an error is printed to the screen to inform the user and the command is not accepted.
The resulting namespace then of course contains the unpacked files as new names. This allows the namespace of the whole session to become a namespace tree in the same way that the UNIX file system offers a naming tree, with `/` as a delimiter. Using another shell built-in, `cd`, the user can also *walk* the namespace tree just as with the usual UNIX file system, as is also demonstrated in the example shown in Fig.21.
This is important because since there are no variables in the Drift language, only names and namespaces one could now say that either the original file system was liftet up into the language or the language allows its variable-space to be traversable like a tree. Both

interpretations, I think, are valid.
But not only the variable-space or data-space forms a tree. As was already mentioned, also the orchestration of services and the data flow between them forms a tree or possibly even a DAG.
In order to allow for such composition, the Drift language also features the `|`-operator (pipe operator) which behaves exactly like the pipe operator already known from the UNIX shell as introduced in chapter 3.3. Fig.22 shows one possible use of the pipe operator in order to count the number of lines of comments that might have been left under an online video or blog post.
This again showcases the `*` syntax, which is not only needed in order to indicate that a service *produces* a namespace but also to indicate that a service *receives* a namespace as input. Therefor the interpreter can immediately check, (with help of the service registry) whether or not the basic formats of names, namespaces and services are valid. But this is not only supposed to help the interpreter. It is of course done with the best intentions for the user, so that he, given meaningful names and these basic format identifiers, might intuitively understand even longer chains.

```
Session: e3d78ad5-898...
.> import comments.tar
.> q
--------------------
.> linesOfComments = Untar* comments.tar | *Concat | LineCount
.> $linesOfComments
  4653
.>
```

Figure 22: Example of using the pipe operator and `*` syntax.

However, it is not immediately obvious why such an operator is actually needed. As was shown so far, the result of every single service invocation, whether it returns singular data or a namespace, could be bound to a name or namespace respectively. Leaving it at that, would create the same zick-zack pattern as demonstrated in Fig.9 in chapter 3.2. Unfortunately this would lead the user into a direction that is opposite to the core philosophy of the Drift language.
In Drift, there are only names. Not because this makes things easy but rather because it is the *names* that carry all the semantics for us programmers and the goal of the Drift language is to eliminate every unnecessary clutter and focus on the essence, focus on the names and chose them wisely in order for them to convey as much *meaning* as possible.
But not every single service invocation is full of meaning. Sometimes a certain combination of service calls, maybe even in the specified order provide a meaningful transformation. Instead forcing the user to provide meaningful names for every single invocation would lead to meaningless placeholder names like `ccd` or `idx2`.
So in the same way, that *Git* puts the responsibility of chosing when and how much to

commit into the hands of the user, the |-operator allows the user the chain together multiple service invocations sequentially and then giving the result of the whole chain a meaningful name. So it is still possible to revert to single-stepping each service call if deemed necessary but arbitrary batch sizes of services are also possible. It is up to the user to chose wisely.

```
Session: e3d78ad5-898...
.> import data.csv
.> q
-------------------
.> a = A data.csv
.> b = B a
.> c = C a
.> d = D b c
```

Figure 23: Example showing how to construct dependency graphs.

But sequential chains of service do not make for full fledged DAGs. Fig.23 shows an example of the well known diamond formation. So simply reusing the names that have been created through earlier service calls and binding their results to new names in order to make them reusable later allows for the creation of arbitrary dependency graphs. These graphs naturally lend themselves to be visualized as will be shown in chapter 4.4.

So far, one could argue that besides the eager evaluation, the Drift language behaves like a basic functional language. There are no variables so there is no contention about the values of such variables (the name *variable* itself suggests change over time, also known as state) and data produced by services is only loosely bound to names as is mostly done in functional languages [4].

```
 1  .> a = A data.csv
 2  .> b = B a
 3  .> a = C
 4  .> ?b
 5    B a
 6  .> ?b.a
 7    A data.csv
 8  .> ??
 9    B (A data.csv)
10  .>
```

Figure 24: Example showing state in the language and how service invocations behave as closures.

However, Fig.24 shows how state is included in the Drift language because names, even when services have already been started with these names as input, depending on the data they represent, can be overwritten and bound to new data any time.

The rather abstract example shows how a name `a` is created by starting the service `A` with the input name `data.csv` which can savely be assumed to be an imported name, because only import names are allowed dots and file endings like `.csv`. While this `A` service might be running, another service, `B` is started, using the ouput of `A` as input. Since data is made available to these names as soon as it is available there might be a data stream between these two services, who also form a sequential data dependency. But it is also possible, that the user enteres line 2 after the invocation of service `A` in line 1 has already finished. Then all the data is available and `B` can pull that data depending only on its own processing speed.

[4]by the use of a so called *let*-expression [85]

Then, in line 3, the name `a` is re-bound to the result produced by the invocation of `C`. So when now querying the data behind `a` using the `$`-operator, the user would see the data produced and streamed by `C`. If the user queries the data available behind the name `b` however, it would see the result of B *still consuming from* `a`. Therefor the invocation of `B` on `a` *captures* the value of `a` indefinitely and is therefor not concerned by any later change. Therefor service invocations behave exactly like *closures*, a concept from the area of functional programming describing function invocations that capture their environment (variable names and their values) [86], [87].

Unfortunately this can make it difficult to remember how the data currently shown by a query on a name came to be. In order to help with that, the Drift Shell offers another built-in feature, namely the *history query* operators `?` and `??`.
As shown in line 4-7 in Fig.24, the simple history query operator `?` can be used for a quick resolution of the statement that was issued in order to produce the data that is currently bound to a name. In the case of the name `b` that statement was the invocation of `B` on `a`.
But since the value of `a` has changed since the original invocation, the simple history query operator `?` can further be used to resolve *any* name in a statement and show the statement that created that name, at the point in time when the overall statement was issued, as shown in line 6 and 7. When the overall statement was issued, the name `a` was bound to the result of the invocation of `A` on `data.csv` and not to the result of `C` as is now the case.
So the simple history query operator `?` can be used to arbitrarily traverse the whole history tree of any given name or statement. The full history query operator `??` is then a mere shortcut that immediately unveils the complete history tree of a statement. This should be used carefully, since all the intermediate names are being resolved, it is up to the user to reconstruct the meaning of the issued statements, as shown in line 9.

```
.> a = A data.csv
.> b = B a
.> ls
.> rm a
.> ls
  b
  data.csv
.> ?b
  B a
.> ?b.a
  A data.csv
.>
```

Figure 25: Example showing how names can be removed.

One interesting aspect that follows from this is shown in Fig.25. Since service invocations capture the values of their input names so that they can be overwritten, these names cannot only be overwritten but also flat out deleted. In order to that, the Drift shell offers yet another built-in command called `rm`.
Although the deleted name is no longer shown by `ls` and therefor currently not available for any commands, its value is still captured by the invocation of `B` just as it was captured in Fig.24. In that sense the Drift language allows for state, because the value of its variables, its names, depends on the moment of time when they are queried. But it also works like *Git* in the sense that the user can observe her behavior imme-

diately: things that are deleted are gone but underneath everything depending on those things still works. And so the next chapter will show how these semantics allow for and are used for building a distributed system, orchestrating multiple independent services on multiple machines in a fault-tolerant manner.

4.3 Drift System Implementation

The last chapter conceptionally introduced the *Drift language* as well as the *Drift Shell*. This chapter will introduce the current implementation of the *Drift* back-end system, based on the ideas presented so far.

Both, the Drift language and shell, are heavily influenced by the idea that most of the semantics of programming for us programmers is contained within the *names* that are being used. Which makes it hard to differentiate between which functionality is defined by the language and which by the shell.

One example of this would be the variable-space which is defined to consist of names and namespaces only, forming the same kind of namespace-tree as a traditional UNIX file system. This tree and variable-space can be walked using the language keywords `ls` and `cd` which are therefor implemented as shell built-ins.

Furthermore some assumptions and invariants of the underlying system's behavior have also been sketched, based on the ideas introduced by chapters 3.1 - 3.6. One example of this would be, that names can be removed from the pseudo-global namespace emulated by each shell session, but the data that is referenced by this name needs to be held on to and made available by the system in case any running service is still consuming from this data. So using the concepts of Rich Hickey as introduced in chapter 3.1, from the user perspective names behave like places but from the system perspective data behaves like values.

Another aspect that has already been scratched upon, is the idea that data should be made observable as soon as it is available. This is based on the realization that batch processing is only a special case of stream processing. In other words: batch data is just streaming data with all the stream snippets that would otherwise trickle down over time having already arrived. Therefor any system that is able to deal with the fact that data might only arrive as delayed chunks, will naturally be able to deal with a delay size of 0.

So in order to build a back-end system that supports these assumptions, one needs to first define the basic tasks of the front-end (the shell), the back-end and how they interface with one another. Fig.26 shows the basic idea of how this looks in the current implementation.

4.3.1 Drift Shell

The shell's job is to present the user with a textual interface and send any valid commands as tasks to the system. Therefor the shell is *not* part of the system per se. It runs on the user's machine and connects to the system via network. This means

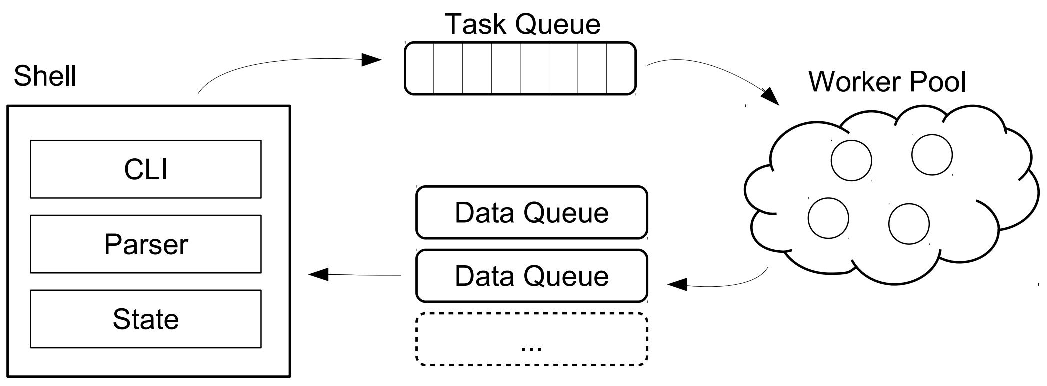


Figure 26: Abstract architecture of the Drift front-end and back-end and the interfacing between the two.

the user can interface with the system virtually from anywhere. This might seem insignificant but another vital task of the shell is to present the user with its own seperated and stateful environment. Each session defines its own names and never interferes with any other ongoing session, in the same way that a user working on a UNIX system is isolated from and should never even notice any other user.
So in order to provide these guarantess, the shell mainly consists of three parts. All parts and therefor the shell itself are currently implemented in Java 8 (official version 1.8). The *command line interface* (CLI) deals with presenting the shell to the user and handling any input and output from and to the user. Any input received is passed to the language parser which was generated using the ANTLR parser generator (version 4) [88]. Additionally in the parsing layer the shell connects to the service registry which is not yet shown in the high-level overview of Fig.26. If any syntactical errors are found or the services do not fit the specification stored in the service registry, a specific error message is printed to the screen.
It is important to note however, that in the current implementation every command entered by the user is fact-checked with the help of the service registry. This could be optimized by caching any information retreived about a service in the client shell, therefor saving round-trips over the network. However, such a caching infrastructure would, depending on the change rate of the entries in the service registry, be vulnerable to stale entries. Therefor some form of cache invalidation or cache coherency protocol would be needed, updating *all* the client caches whenever an entry in the service registry changes. This was considered future work, as will be discussed in the appropriate chapter, chapter 6.
The third aspect of the shell is the internal state of the session which consists of two things. A name table that maps the names the user created in her session to the names that are used withinin the system and the history log for each name and namesapce in the session.
The history log for each name (or namespace) is itself implemented as a *persistent data structure* [89], [90]: an immutable append-only log, just like Datomic as presented

in chapter 3.4. Therefor each name that is created, either via an import or a service invocation, gets its own history log. Each name that occured in the command that created the name is stored as a reference into the history log of that name and therefor the value of that name at the point in time when the command was issued.
Over time this creates a reference tree in which the entry of a name points to old values of the names that were used to create it. This tree can then of course be traversed using the history query operators `?` and `??` as introduced in the last chapter, chapter 4.2. This is also one aspect of the implementation of the closure behavior of service invocations as demanded by the Drift language, as was also presented in the last chapter.

4.3.2 Drift FS

When the command that was entered by the user is valid, a task description is generated and send to the *task queue*. There is only *one* designated task queue. The task description itself is also generated by the parsing layer of the shell.
Since the Drift language does not feature very much syntactical concepts it has a reasonably straight forward grammar. So it could be argued that the usage of the ANTLR parser generator might be overengineering, since most of the more advanced tooling offered by the ANTLR tool is never used. However, given the structure of the parser generated by the ANTLR parser generator, parsing a command given by the user naturally builds up a command description structure, by visiting all the elements of the command down to every name. Therefor the use of ANTLR was not only for automatically generating the parser itself but also for generating the task description as a byproduct of parsing each command.
This task description fulfills two purposes. For one it serves as the obvious task description and contains every information needed by a worker to execute the given task. But it also serves as an identifier because in order to deliver the guarantees demanded by the semantics of the Driftlanguage, one invariant of the system is that the execution of services is *deterministic*.
This means that given two service invocations with the exact same input data, the results of both invocations must be identical. From this it follows that since the result of a task only depends on its direct inputs and not on any further state of the system, the task invocation description, the command that created the task, can be used as an identifier for the result of the task.
This is done via hashing the task description structure that was generated by parsing the given command. The resulting hash value is used to uniquely and globally identify the result of the command in all of the system. It is this hash that is stored in the name table of the shell, that maps the names given by the user to their hashes, their global system names.

Nonetheless, in order for the workers to be able to execute a task, the whole task description structure is send to the task queue. On the other end of the task queue there is a pool of workers. In order to allow for these workers to achieve hight scalability and fault tolerance, it is adviced to run each worker on a seperate machine, as was done in the current implementation.

Any free worker will then pull the new task out of the task queue, execute the task according to the task description and produce its result into another message queue, a *data queue*.

The name of the resulting data queue is of course the hash of the task description structure the worker received. Therefor both, the shell and the system exchange the full task descriptions but use the hash of the description in order to identify where to consume from or produce to. So when the user queries the name, that was bound to the result of the command that was entered earlier, the shell looks up the hash-name from the user-defined name and consumes from the data queue with that name in order to show the result to the user. However, for data that was imported into the system no such task description exists. Therefor the import command is implemented as using the full content hash of the imported data, which unfortunately is expensive in terms of compute resources on the client (the shell).

The whole data layer of the system is implemented using distributed message queues. This was done because of the demand of the Drift language that data needs to observable as soon as it is available. Earlier versions featured a distributed file system like the *Hadoop Distributed File System* (HDFS). Unfortunately it is absolutely not trivial to implement *any* form of data streaming capabilities on top of HDFS, since the file abstraction provided by HDFS does not allow for any contention or publish-subscribe functionality.

At the time of this writing most of the available distributed file systems are still struggling with even providing basic POSIX conformability. HDFS for example is *not* POSIX conform. Unfortunately, even if it was, this wouldn't actually be of much help, because the POSIX file system API does not provide any mechanisms for being notified for file change. Eventhough the Linux kernel does implement such an API, namely the *inotify* interface, this is mostly ignored by both, the POSIX standard and the distributed file system community.

Luckily the distributed message queue community can be considered very active in this regard which is why not only coordination and communication is done via message queues but also storage and persistence. This means that the shell can listen on the resulting data queue represented by name and as soon as the first data token is produced by a worker into that queue, consume this token and present it to the user.

4.3.3 Execution Engine

In order to illustrate the detailed processing of a task, Fig.27 shows the basic steps that are taken by the shell as well as the back-end system. It all starts with the user issuing a command. In this case the command itself is irrelevant, but as one can see it's a command that produces singular data because the user binds this result to the name *foo* [5].

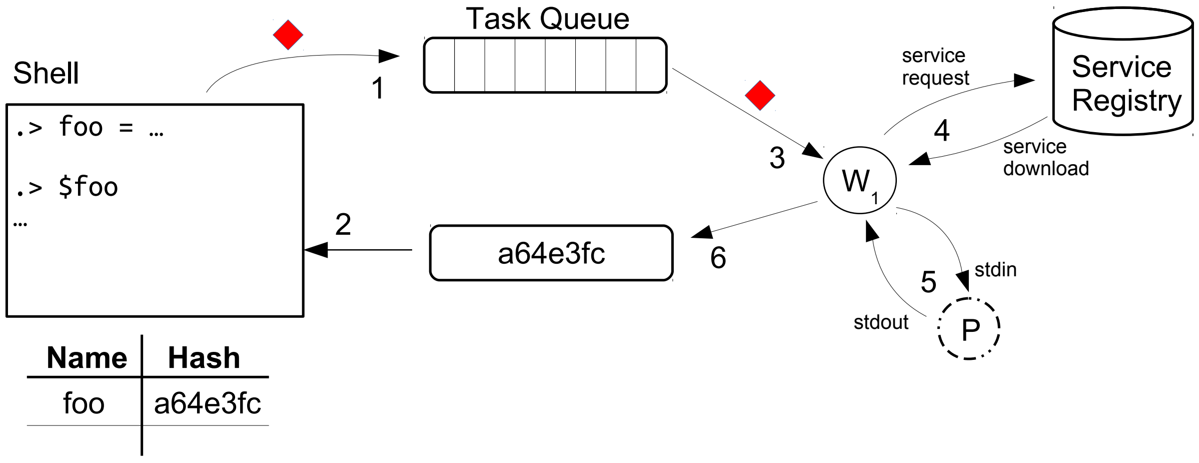


Figure 27: Detailed steps for processing a single task.

The command is then parsed and the task description structure is created. If the command includes any already existing names, the name table is used to resolve these names to their global hashes. When the task description structure is finished, its hash is stored in the name table behind the name *foo*, as shown in the example.
Since names that are used in the command are first resolved to their global hash names, it means that the final hash for the task is calculated over a structure that might already contain other hashes. Therefor the resulting hash might be a hash of hashes and tasks issued may form a *Merkle Tree* [91], [92], in the same way that the objects stored by *Git* in its underlying file system form such a tree, as was discussed in chapter 3.5.

The task is then send to the task queue, which is shown as step 1 in Fig.27. If the command of the user contained the pipe operator, the shell will split the chain up into individual commands and replace the input names of each command with the result hash of the task description structure of its predecessor. The individual tasks are then send to the task queue in the order in which they are read, from left to right. Otherwise it would be possible to construct task chaines with more tasks then available workers, which would bring the system to a halt.
The task queue is implemented as a *RabbitMQ* message queue [93], [94], [95]. It's important to note that the current implementation uses RabbitMQ in the version 3.6.6-1 and that RabbitMQ is run in the fault tolerance and distributed mode, which means that queues are replicated three times and messages are *not* written to disk but rather kept in RAM at all times. These fault-tolerance and scalability measures are provided by the RabbitMQ message broker by default and were used without any modifications except for configurations and setup.

[5]This is of course just a dummy name for the purpose of the example. As was explained earlier, when using Drift names should be chosen wisely in order to convey meaningful semantics.

Since the result of the command is bound to a name, a new prompt is immediately available to the user. It is now assumed, for the sake of the example, that the user immediately wants to see the data of the result and therefor queries the name *foo* using the query operator `$`. This is shown as step 2 in Fig.27. This showcases another feature that is widely adopted by most message queueing frameworks and infrastructure: both, consumer and producer, are able to create queues. That means that whoever requests a queue first, will trigger its creation and any subsequent request for queue creation are being ignored as idempotent by the message broker. So in this case it is assumed that the shell, as the consumer, is first to request data from the queue with the name `a64e3fc`. For step 2 it is further assumed that no data is currently available, so no data is printed to the user and the shell blocks in a sort of receive loop, in the same that an actor would block, waiting for new messages, as was introduced in chapter 2.3.

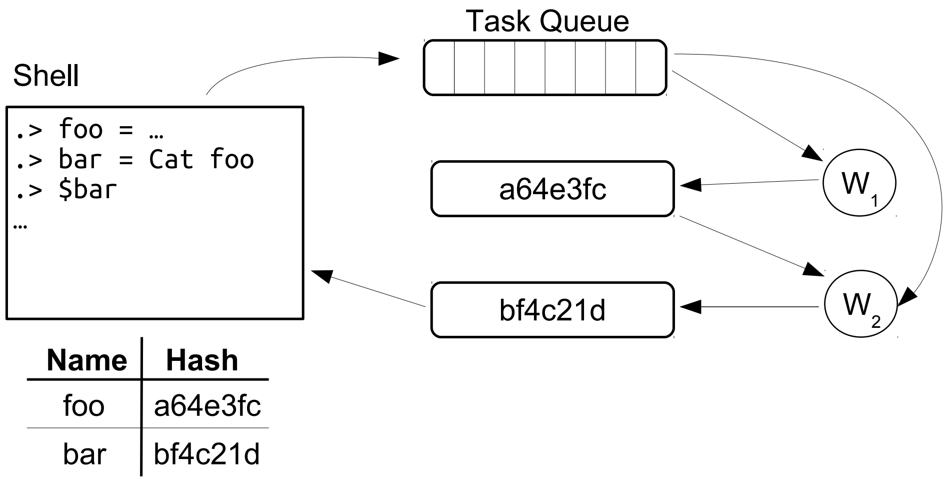
Step 3 shows how the free worker W_1 receives the task from the task queue. As will later be discussed in the error model section, there is no spreading or redundant execution used in order to deal with possible worker failure. A task must only be received by a single worker and must also only be executed *exactly once*. This is another invariant of the system. Guaranteeing that only exactly one worker receives a task from the task queue is again already taken care of by RabbitMQ out of the box. The internal queue scheduler used by RabbitMQ uses round-robin scheduling by default. Therefor a new task is only sent to exactly one worker. If that one fails, another different worker will receive the task. A task is only ever worked at by exactly one worker at a time.
The worker itself has no idea about the possible services that are available in the system. The worker has been implemented as a *generic* worker. This means that the worker downloads the service that is being contained in the task description from the service registry and simply executes the received service, as is demonstrated by step 4. Since the generic worker is currently also written in Java, the downloaded service could also just be Java code and therefor executed by the worker instance natively. However, for the sake of the example it is assumed that the downloaded service code contains instructions to fork another Linux process. This means that arbitrary executables can be executed by the worker, in the same way that the foreign function interface of Cuneiform allows to execute already existing bioinformatic tools without any code modifications, as was introduced in chapter 3.2. However, it needs to be said, that the current implementation assumes that the binaries referenced are already pre-installed on the worker node. However, this could be easily avoided by also storing the needed executable in the service registry and downloading it if necessary.

Step 5 shows how the worker has forked another Linux process. The *stdin* and *stdout* (and *stderr*) channels of the spawned process are redirected by the worker. This allows the worker to receive any needed input from other data queues in the system and forward them to the forked process and receive any output of the forked process and forward this output to the output queue of the worker and therefor the output queue of the task. The forked process at no point needs to know that it's being part of a distributed system of workers and distributed message queues. This means already existing tools can be re-used without any modification. Step 6 shows how the worker forwards any piece of data received from the forked process to the output queue, whose name is the hash of the task description the worker received in step 3. Whenever the worker produces an output data token, anyone listening on this particular output queue will be able to observe the data and so naturally the shell, as one such listener would print the received data to screen.

Unfortunately there is a problem. Fig.28 shows an example including a data dependency between the task whose result is stored behind the name `bar` and the task whose result is stored behind the name `foo` because the service `Cat` takes the data from `foo` as input. Given the eager-evaluation of the shell, however, there are multiple timings in which these tasks might be invoked.

One would assume that the easiest one would be for the user to take a break after she issued the `foo` command and start the `bar` command only after the `foo` has already finished and produced all its output data. Then all the data would be ready even before the second task is started, which could result in the same worker executing both tasks. Unfortunately this is the most problematic case.

The “best” case in terms of implementing the data layer using message queues only would actually be if the user issued both commands immediately and the consuming worker, worker W_2, was already consuming before the first data item produced by W_1 arrived in queue `a64e3fc`.



```
.> foo = …
.> bar = Cat foo
.> $bar
…
```

Name	Hash
foo	a64e3fc
bar	bf4c21d

Figure 28: Example showing the data dependency between tasks.

This is due to the *Abstract Message Queue Protocol* (AMQP) which is implementing by almost all message queue implementations and frameworks. The AMQP defines that, although messages can be fanned out to multiple consumers, a consumer only receives those messages that arrived in the queue *after* he registered at the queue, in the same way that a chat client only receives the chat messages that are posted, after

it joined the chat room.
Since RabbitMQ implements the AMQP, it would've been impossible to implement the data layer using RabbitMQ only, given the eager-evaluation strategy of the Drift language because as was described earlier, it must be possible for consumers to pull data from a queue *after* that data has already been produced to that queue. In fact, consumers need to be able to start consuming from the beginning of the queue, no matter how much messages are already contained in that queue.
Luckily there is a new open source message queue project called *Apache Kafka* which does *not* implement the AMQP [96], [97], [98]. Therefor Kafka queues keep their messages for a configurable retention period, regardless of how many consumers "consume" them. It is debatable whether or not messages are still being *consumed* when they are not actually removed from the queue. Therefor it is also possible to describe the service of Apache Kafka as a distributed log, since messages pile up in the queues without ever being deleted if the retention period is set to infinity as was done in the current implementation.

Because messages that have been produced to a Kafka queue are never deleted, it is possible for a consumer to connect to a queue that already contains messages and read the messages in the order of their arrival, starting from the very first messages. One could say that the consumer can read the message *history* of a queue, in the same way that *Datomic* allows the accretion of facts to build up history, as was introduced in chapter 3.4.

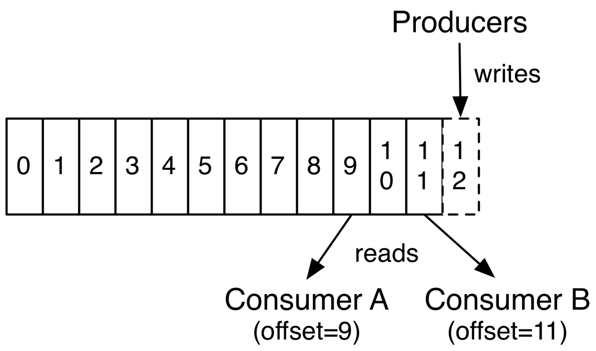


Figure 29: Example showing multiple consumers independently reading from the same Apache Kafka queue at different positions.

Furthermore, this *immutability* of the messages allows for multiple consumers to read and consume from the same queue *concurrently*, as shown in Fig.29 which was taken from [96]. In RabbitMQ and other message queues that implement the AMQP consumers are actually contending for messages if they are consuming from the exact same queue. A fan out behavior is also possible, this however needs all the consumers to have already registered at the fan out component (which is called *Exchange* in RabbitMQ) before messages are being produced. In Kafka any consumer can start consuming whenever he likes at whatever position or message in the queue and completely unaffected by any other consumer. Therefor the actual *data queues* in the Drift system implementation are Apache Kafka queues.
Because of this immutability and message store rather than message queue nature, Kafka implements a *pull-model* instead of a *push-model* as used by other message queues like RabbitMQ. This pull-model has the client starting at a specified message

index called *offset* and then pull for new messages for a specified duration. When the poll duration is over, the poll returns all messages that could be fetched during the poll as a batch.
The current Drift implementation uses the Kafa Java API in version `0.10.2`. Unfortunately even in this latest Kafka version, the polling for data is also used to also signal consumer liveness to the Kafka broker. Kafka therefor effectively multiplexes the data channel (polling for data) with the signal channel (liveness via heartbeats), resulting in the consumer having to constantly repoll in order to not be marked as disconnected by the Kafka server. This is still true even when the queue the consumer is polling on is *deleted* using the external admin tool provided by Kafka. Therefor an already ongoing poll is *not* interrupted by the queue deletion and any subsequent poll will recreate the queue because consumers can also create queues as was explained earlier. So any external *problem* with a queue must be signaled to consumers via messages inside the queue. The only exception that can currently be used to interrupt poll is a so called `WakeupExecption` which must be thrown using the `Kafka.Consumer` object itself, by another thread from inside the consumer.

This is implemented differently in RabbitMQ. Since RabbitMQ implements the push-model, the user specifies callback methods which are invoked whenever a message is delivered to the consumer and this callback interface also allows to specify a method which is triggered whenever the queue on which the consumer is currently waiting for messages is deleted. A single delete-action on a queue will trigger the delete callbacks on *all* consumers currently listening for messages on that queue.
Therefor a single logical queue as used by the Drift system always uses both: an Apache Kafka queue for the plain data and a RabbitMQ queue for signaling any errors to all consumers.
Fig.30 tries to illustrate the full extent of the Drift back end. It shows two workers forming a data dependency. The first worker, W_1, received the diamond task which specified to consume from the input queue `c4d245`. Since this is only a single data queue, this represent data that was once imported because of the import either failing or succeeding there is no need any further signal queue for intermediate task failure. The data queues (Kafka queues) are named with normal letters whereas the signal queues (RabbitMQ queues) are named with italic letters.
After having downloaded the required service from the service registry, worker W_1 spawns another Linux process P_1 as specified by the command it received. It then forwards any data item it received from the input data queue to this process and any data item produced by the process to its output data queue. Worker W_2 does exactly the same independent of W_1. It receives the triangle command, downloads the required service and spawns the service as a new Linux process. The triangle command contained the hash `a64e3fc` as an input queue for W_2, so therefor W_2 consumes from the ouput queue of W_1 and produces to its output queue, whose name is the hash of the command structure that already contained the hash of the output queue of W_1, as was explained earlier. Since the user queries the result of `bar` and not the result of `foo`, the shell then consumes from the output queue of W_2 and shows every data item

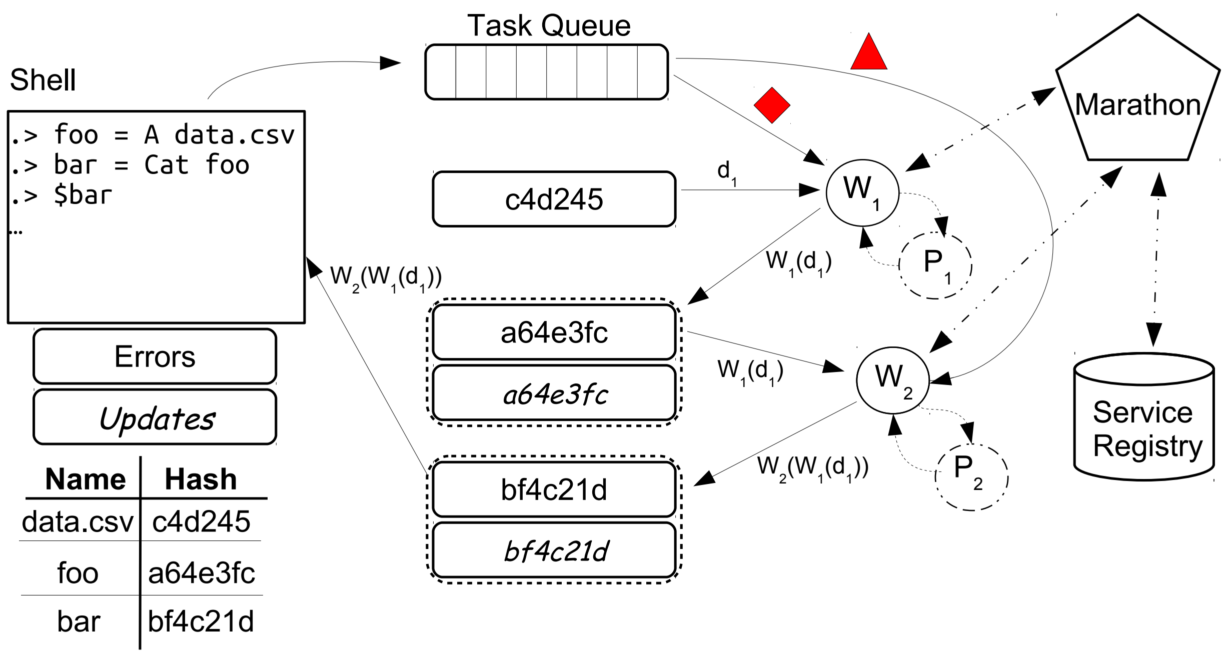
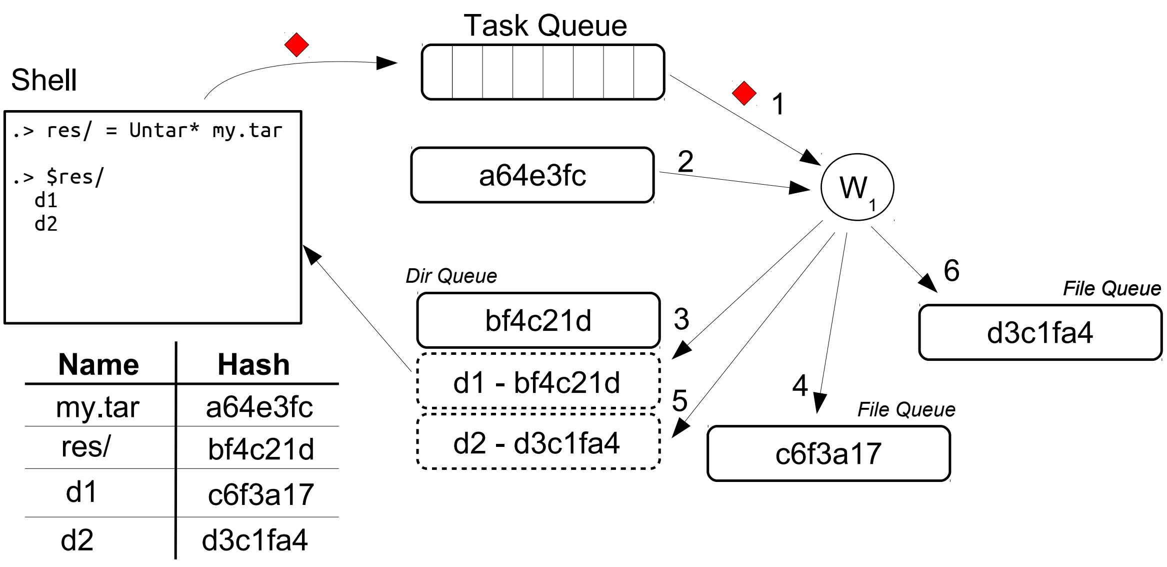


Figure 30: Example showing the full extent of the Drift back end. Two workers are running, forming a data dependency with one worker consuming from a data queue created by an import and the other working consuming from the output data queue of the first worker.

received to the user.

The examples so far have shown the detailed execution of services dealing only with regular data which is either bound to a single name and then later queried by the user or directly printed to the screen when received by the shell. But as was introduced in chapter 4.2 there are also tasks that receive or produce a *namespace*.
Figure 31 shows how the creation of a namespace is implemented by the Drift back end. As can be seen the user issues the `Untar*` command that was already introduced earlier. The asterix on the right side of the command name indicates that it produces a namespace and therefor its result must also be bound to a namespace, as indicated by the `/` in the name `res/`.
After commiting the task to the task queue the shell inserts its hash into its name table, so the namespace `res/` is listed by `ls` and is also queriable. Querying a namespace results in effectively querying what is called the *directory queue*, because of the close resemblance of names and namespaces to directories and files in the original UNIX file system.
The worker receives the task, which is shown as step 1, and then receives its input data from the appropriate queue, namely the queue that contains the `.tar` file which that was created by an import as indicated by the name. This is shown as step 2. Now for every resulting name in the namespace the worker will first *publish* its name and its



```
.> res/ = Untar* my.tar

.> $res/
  d1
  d2
```

Name	Hash
my.tar	a64e3fc
res/	bf4c21d
d1	c6f3a17
d2	d3c1fa4

Figure 31: Example showing the creation of a namespace using directory queues that contain the original file name and their hash name and the actual data queues representing files.

hash to the directory queue with the hash being again the content hash of the extracted file in the case of Untar*. Therefor the this name would now be available as a name inside the res/ namespace in the shell and could be used for further commands. This is shown as step 3 in Fig.31. The worker will then start to fill the data queue of the published name which is step 4. When it finished the first name, it will publish the next name and start filling its data queue until every name in the namespace has been created, which is illustrated by step 5 and 6 [6].
So therefor not only the data behind names is streamed, but also the content of namespaces is stream as well as the data behind the names contained in that namespace.

Going back to the full back end overview as shown in Fig.30, a new element besides the already introduced *Service Registry* is shown, namely *Mesos Marathon* [99]. Marathon is a distributed fault tolerant watch dog process that monitors services spawned through its interface. If a failure of such a process is detected by Marathon, the failed service gets restarted without any further effort by the programmer or system administrator. Marathon is one component of the *Mesosphere* project which started out as a distributed resource negotiator called *Apache Mesos*, which is now it's probably most widely known component [100], [101]. Since Marathon monitors Mesos containers, the workers as well as the service registry are executed as Mesos containers.

[6]In this example the dashed queues do *not* represent any signal queues but are just supposed to illustrate the entries of the directory queue.

]

So therefor all of the core components of the back end, both message queue frameworks including the task queue and Marathon are run in distributed mode. That means that the message queues are automatically replicated three times and keep all their messages *only* in RAM. Disks are never used. Marathon itself is also replicated three times with one master and two slaves in order to provide fault tolerance.
It's important to note that these are the fault tolerance capabilities that these tools offer by default. Apache Kafka for instance comes with its own Apache ZooKeeper instance for internal distributed coordination and distributed consensus that itself is also run in distributed mode, meaning being replicated amongst multiple nodes [102], [103].

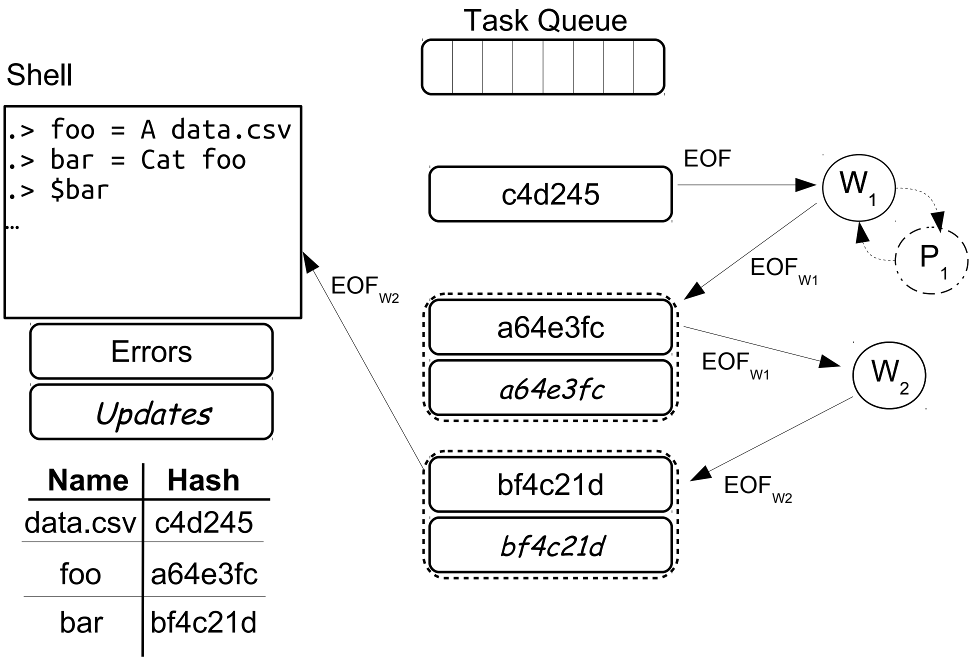
Although all of the used tools are open source software, with their code being freely available via *Github*, all these tools were used without any code modifications. Therefor most of the fundamental problems of programming a distributed system, like distributed messaging or distributed consensus are being taken care off. The main challenge therefor lies *not* in resolving these already solved problems but rather in learning how to utilize these tools given the vast variety of tools, options, configurations and customizations. Fig.32 summarizes all the tools and their versions as used by the current implementation.

Tool	Version
Java	8 (official 1.8.0_121)
ANTLR	4.6
RabbitMQ	3.6.6-1
Apache Kafka	0.10.2
Apache Mesos	1.1.0-1
Mesos Marathon	1.4.1
Apache ZooKeeper	3.4.8

Figure 32: Overview over the tools and their versions as used in the current implementation.

4.3.4 Error Model

Since now that all of the components of the Drift back end have been introduced, the last thing that needs to be discussed is the handling of errors. The basic failure model that is assumed for the components of the back end (the system) is *crash-recovery*. This is realized by using Mesos Marathon for spawning, monitoring and restarting the worker processes and the service registry. The used message queues again offer the restarting of lost queues out of the box.
Before introducing the error cases, Fig.33 shows the normal mode of operation. This uses the same example as was shown earlier, omitting the unnecessary parts. So again one can see the two tasks W_1 and W_2 processing what has been bound by the user to the names `foo` and `bar` respectively.
Worker W_1 consumes its input from a data queue which was created by importing a local file. After the import has finished, the import action itself inserts a special coordination token, an `EOF` token, into the data stream. When worker W_1 has consumed all the data token available in this input queue, he will consume the `EOF` token. This



```
.> foo = A data.csv
.> bar = Cat foo
.> $bar
…
```

Name	Hash
data.csv	c4d245
foo	a64e3fc
bar	bf4c21d

Figure 33: Example showing the successful completion of two tasks and the propagation of the `EOF` token.

signals the end of the input data stream and triggers the termination of the spawned process P_1. After the process has been gracefully terminated, W_1 also sends an `EOF` token to its own output queue and starts listening for new tasks at the task queue again. Therefor, naturally, whenever W_2 or any other consumer reaches the end of the data stream produced by W_1, it will encounter the `EOF` token of this stream which again triggers the creation of its own `EOF` token.
This `EOF` mechanism is of course also used when dealing with the creation of namespaces and therefor directory queues, as for example shown in Fig.31. Each individual name will be finished by inserting the `EOF` token into its data queue and at last the worker will finish up the directory queue by also inserting the `EOF` token.

This showcases an important overall property of the Drift back end implementation: the only thing that *any* worker needs to know is only its immediate environement, i.e. where to read its input from and where to produce its output to. If a worker is finished, it closes its ouput stream by producing a last `EOF` token, without having any idea whatsoever about who or how many other workers might wait for the next token and without any global knowledge about the overall data dependency graph built up by the user.
This is heavily inspired by *Petri Nets*, a theoretical model for describing distributed systems [104], [105]. The core concepts of Petri Nets are *places*, literally places where tokens can reside and *transitions*, actions that consume tokens from their input places and produce tokens to their output places. The relevant aspect of Petri Nets is that

whether or not a transition is *enabled* (can fire) only depends on the number of tokens on all of its input places. Therefor each and every transition is enabled independent of any other transition and carries *no* global knowledge about the net it is being part of or any other global state which is the key property for scalability in the context of distributed systems.
In that same spirit, every worker in the Drift system has no idea about the overall structure of the system or the complexity of the data dependency tree that was and is being created by the user. Every worker directly consumes from the inputs contained in the task description and produces to an output queue named after the hash of the task description itself.
Therefor, when the `EOF` token is produced by one worker, it will slowly but definitely trickle down whatever data dependency tree has been created by the user. Reaching every listening worker without ever knowing about them. Every listener will then create its own `EOF` token, therefor propagating the token one level further down the dependency tree.

Given the invariant of the system that service invocations are *deterministic* and using the described setup, the system differentiates between two error cases: either a *system failure* or a *user failure*.
A user failure is encountered whenever a task issued by the user made it to the task queue but leads to an error when it is finally executed by the worker. For instance this could be case whenever the user issues a command in which a service consumes some input but the process that is spawns expects a different input format as provided by the input queue specified by the user. Naturally this will lead to an error produced by the forked process and will result in the cancelation of the task.
Unfortunately these types of errors, the user errors, cannot be resolved by the system. Under the assumption and system invariant that tasks are deterministic, it holds that a task that led to a processing error, will recreate the same error if executed again. Therefor restarting the task using a different worker would only lead to the same error result. Therefor the whole task needs to be marked as erroneous and to do that the worker will produce the only other special control token besides the `EOF` token, a `ERR` token. This `ERR` token is inserted into the output data queue of the task and the worker will signal the task completion to the task queue, so no other worker will receive that task. Therefor the error result is forever persisted with this specific task invocation.
Interestingly, if the error only happened *after* some resulting data tokens were produced by the task, the error token is only added to the output stream. This means that no further data will be added to the ouput, but the data that was produced before the error occured can still be consumed from the queue until the `ERR` token is reached.
if a consuming task reads the error token from at least on of its input queues it will gracefully terminate its own processing and produce its own `ERR` to its own output queue accordingly. Therefor a single error will slowly but definitely propagate through the whole dependency tree, in the same way the `EOF` token is propagated, as was shown in Fig.33.
Additionally any worker who encounters an error, whether produced by its own pro-

cessing or by consuming the `ERR` token from one of its inputs, will produce an error notification to the `Errors` queue. This queue is read by the shell in a seperate thread which then marks and updates the names managed by the shells internal name table to indicate that certain names resulted in an error. This is done to prevent the user to issue commands including names for which it is already known that they resulted in an user error. So if a name is used for which this information is already contained in the `Errors` queue, the command will be blocked and an appropriate error message is displayed to the user.
Again, it's important to note that because of the eager-evaluation and immediate feedback strategy of the Drift language and therefor its shell, it is possible that the users issues a command which contains a task that will soon encounter an error, or maybe even has already encountered an error but the information, the message itself, has not yet reached the `Errors` queue or is just being processed by the shell. So the task will be send, will be executed and then encounter the `ERR` token in one of its input streams just as described earlier.
The data a task produced before it might encountered an error is kept, in order for the user to be able to inspect what might have gone wrong. So eventhough the user cannot use the erroneous name besides overwriting it with a new command, she can always query the faulty name and the data available behind that name will be printed by the shell until the `ERR` token is reached.
Since task invocations in terms of user failure have this all-or-nothing property, because of the task determinism invariant, dealing with such errors is actually straight forward. The other class of errors, the system failures, are actually a bit more involved and are the reason why each Kafka data queue is mirrored by a RabbitMQ signal queue.

The error category of *system failures* encompasses anything related to the execution of the workers or the distributed message queues. This means crashing, running out of disc space, memory or any other form of resource or simply not being reachable via the network. It is thought of being the responsibility of the system administrator to fix such errors in terms of providing the necessary hardware. Any software monitoring and restarting as well as replicating is either done by Marathon in case of the workers and service registry or in case of the in-memory messages handled by the distributed message queue brokers themselves.
Since the user is not supposed to deal with those kind of failures it wouldn't make sense to present them to the user in any form. Therefor it is another invariant of the system that the user should never be able to recognize any form of system failure. Especially not when observing data. This is done as illustrated by Fig.34.
When a worker crashes or is unable to continue, it is assumed that any process it might have spawned will also be terminated and therefor discontinued. This is shown as step 1 in the example.
Since the task queue (RabbitMQ broker) will have detected the missing worker but did not receive a task completion notification, it will redistribute the task to any other worker currently ready to receive new tasks. In the example above this is worker W_3 as shown by step 2. Given the invariant of the system that a task result must be

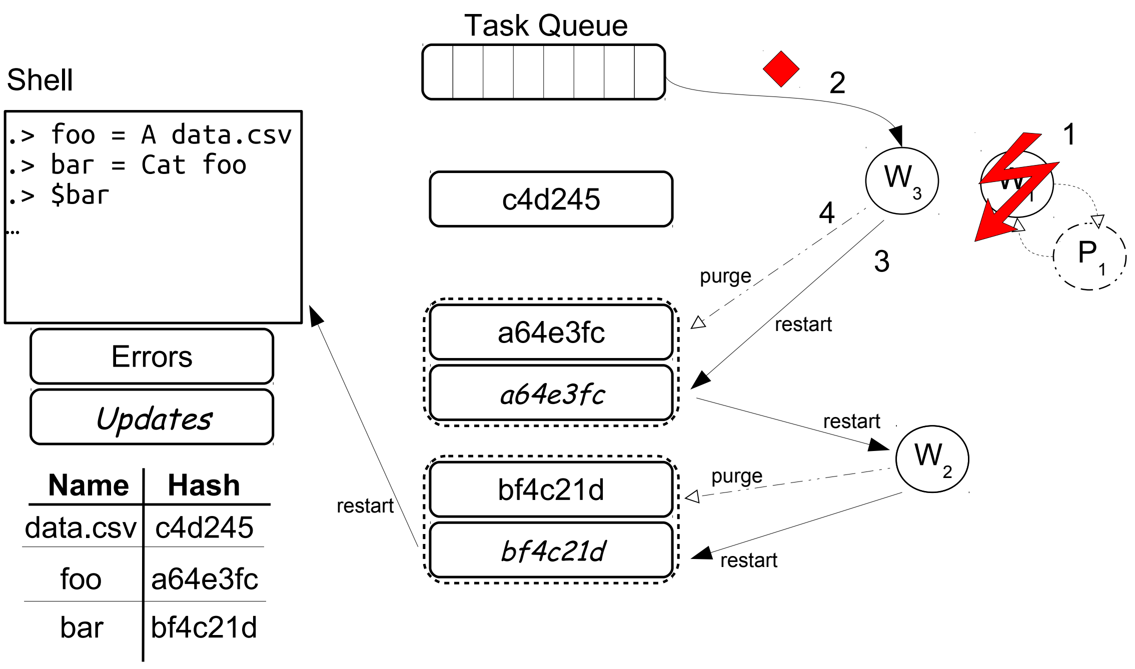


Figure 34: Example showing a worker crash and failover strategy including signal queue deletion and purging of the data queue.

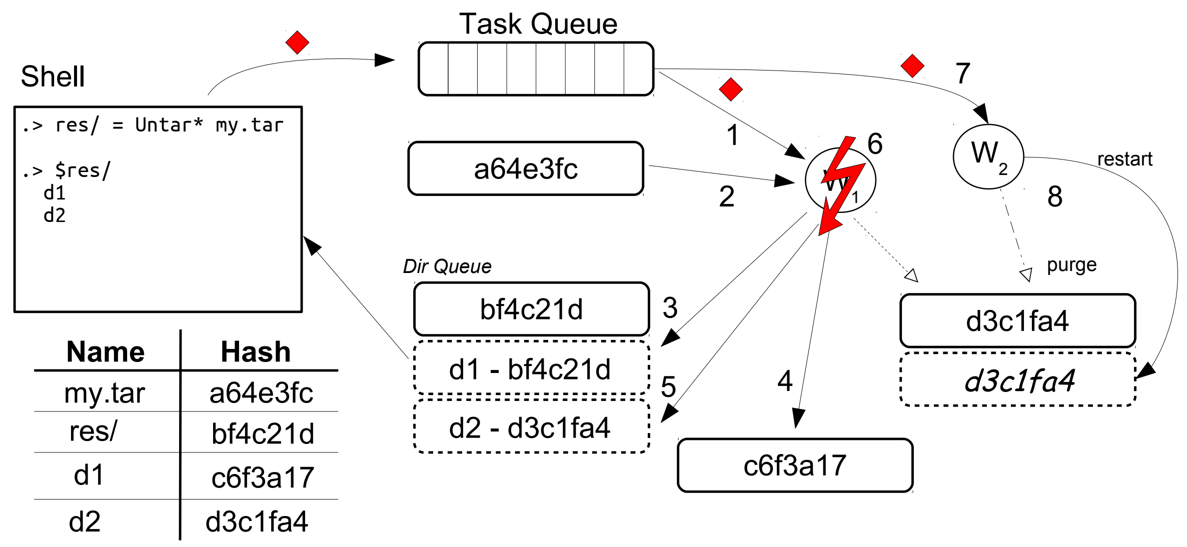
presented as if this task has been executed *exactly-once*, any worker receiving a new task first checks whether or not the expected result queue named after the hash of the task description itself already exists. Unfortunately this information alone is not enough. As was explained earlier, any consumer could have also already triggered the creation of said result queue by listening for data on it. Although possible this is rarely the case in terms of other workers given the order in which tasks are inserted into the task queue and therefor producers and consumers are spawned but one must not forget, that the shell is also considered a consumer.
Therefor it is possible that the shell is already waiting for the result of a task for which the corresponding worker has just received the task description. To account for that a worker not only checks for the mere existence of the result queue but also reads the last token that was produced to that queue. If that token is not the `EOF` nor the `ERR` token, it inferes that the task has already been started by another worker which must have crashed.
The worker W_3 then starts the clean up process as illustrated by step 3 and 4. It first *deletes* the signal queue. This will trigger an immediate notification and callback invocation at *every* listener without knowing them directly (steo 3). Then, it sets the retention period for the corresponding Kafka data queue to the minimum value (step 4). The Kafka broker has been configured to aggressively poll any retention period change, so that any retention period change will be carried out almost immediately (in around 10 seconds). Lowering the retention period of a queue to a couple of seconds, triggers the deletion of any message after that duration. Therefor reducing the retention

period to its minimum value effectively purges the content of a queue which takes on average about 60 seconds even for large queues. After sending the retention period change, W_3 will wait for two minutes before restarting the execution of the task. That means recreating the signal queue and spawning any necessary process and resetting the retention period of the data queue to infinity. It will then refill the data queue that now is assumed empty.
Any worker that received the delete trigger via the signal queue will do the same for its own signal and data output queue. That means it will first delete its own output signal queue, purge its output data queue, terminate its process and then wait for two minutes before resetting the retention period of the data queue and restarting the process. Therefor the resetting also propagates through the whole dependency tree without any component ever knowing its complete structure, just as the `EOF` and `ERR` tokens propagated through the network. Admittedly, setting the retention period of the Kafka data queues to its minimum value in order to delete already produced messages before the crash seems like a hack. Unfortunately, given the immutable nature of the design of the Kafka message queues, even the newest version does not provide any mechanism for explicitly deleting messages from queues.
This reset procedure assures, that when an `EOF` or `ERR` token has been produced to a data queue, the content of the data queue looks like as if the task that produced it was only executed exactly once. This whole procedure is also retried indefinitely because it is only triggered by system failures and it is assumed that these are taken care of by the system administrator as soon as possible.

However, there is one last problem. It is true that the final result looks like the task that produced it was only executed once, but the user could have queried the data queue at any intermediate point in time, just when some data had already been produced but the worker crashed. In order to hide the re-execution of a task and refilling of the data queue after a restart from the user another feature of Kafka message queues is used because Kafka queues not only store plain messages but rather *key-value* pairs where the key can be chosen by the producer and the value represents the actual message.
In the Drift system, the keys are managed by the corresponding worker, who will increment the key for every insert into the data queue. But the shell also retreives these keys whe consuming messages that are then immediately printed to the screen. So for example imagine a case in which the user queried the result of on ongoing task, which has already produced three output items, a, b, c when the user issues the query. The task is now about to produce its next output item d and suddenly crashes but a, b and c have already been printed to the screen in the shell. The whole failover procedure that was described above is triggered and so the data queue is resetting meaning a, b and c are deleted. The shell however still waits for the next new data item d. The next message it will receive from the data queue however will be a again, since the whole task is restarted from scratch.
So instead of buffering already received messages and comparing any new incoming messages with the already received ones, figuring out which one have already been displayed and which haven't, the shell simply stores the key of the last message that

was printed to the screen. Any message that is further received with any key lower than that will be discarded. Only the data item with the next highest key will be shown. Therefor observing the data from the user perspective will always look like a single continuous execution of the task.

The same approach is of course also used when dealing with system failures during the creation of a namespace. Figure 35 tries to illustrate an error case based on the example shown in Fig.31.



```
.> res/ = Untar* my.tar

.> $res/
  d1
  d2
```

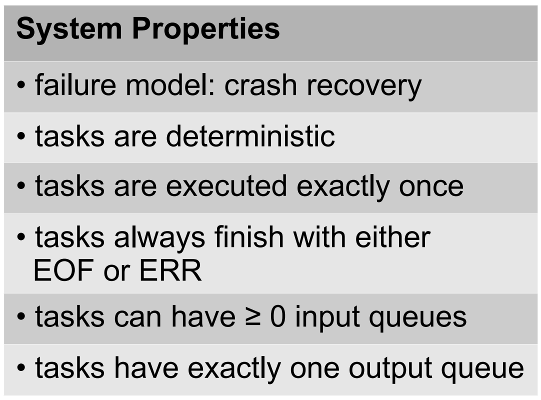
Name	Hash
my.tar	a64e3fc
res/	bf4c21d
d1	c6f3a17
d2	d3c1fa4

Figure 35: Example showing the handling of a worker crash during the creation of a namespace.

As was shown earlier, the worker W_1 first received the `Untar*` task. It therefor received the `.tar` file from its input queue and first published the name of the first result name to the directory queue, making it available to the user, before then uploading its content to its data queue, as shown by steps 1-4. After finishing step 4 by inserting the `EOF` token into the data queue of the first name, worker W_1 published the next name to the directory queue and started uploading its data to the appropriate queue. So now it is possible that the user has already queried the content of the second name in that namespace and has already been streamed the first data items as they were produced by W_1.

It is now assumed for the sake of the example that W_1 crashes. As already introduced Worker W_2 will now receive the task and is supposed to finish the namespace creation. So W_2 also pulls the input data from the input queue specified in the task description but instead of restarting the whole namespace creation process it first checks the directory queue (whose name it can calculate by hashing the task description it received) for any already published names. For each name that has already been published it then checks the last data item in that name's data queue. If the last item is not an `EOF` or

`ERR` token it triggers the same reset-and-purge process that was already introduced. Therefor each unfinished name is recreated from scratch, which is again signaled to anyone listening on that name's queue. But every name that was already finished by any worker that tried to create that namespace before is left untouched.
Therefor the same mechanism is applied in the shell, which as was assumed already queries the second name in the created namespace. Only messages from that name's queue with a key higher than the latest received key are shown to the user, therefor streaming a name from a streamed namespace behaves exactly the same as dealing with any other name.



System Properties
• failure model: crash recovery
• tasks are deterministic
• tasks are executed exactly once
• tasks always finish with either EOF or ERR
• tasks can have ≥ 0 input queues
• tasks have exactly one output queue

Figure 36: Table summarizing the properties and invariants of the Drift system implementation.

This concludes the handling of any system failures. As a side note it should be mentioned that the final design of using the `EOF` and `ERR` tokens as control tokens and inserting them into the data stream was not at all obvious at the beginning. Different designs were implemented that featured multiple Apache ZooKeeper instances which allowed crashed workers to recover their last state and continue where they crashed. Although these designs had their advantages in terms of failover delay, the overall design of the back end system became ever more complex and harder to reason about. Using the control token approach all the used ZooKeeper instances could be eliminated and the overall design became much simpler. The price that is being paid for that simplicity is that the recomputation done by the failover worker restarts from scratch by purging every item that had already been produced to the resulting data queue. But given the overall and inherent complexity of the distributed system context this price was deemed acceptable because the recomputation can easily be further shortened as will be explained in chapter 6.
Figure 36 summarizes the system properties and invariants of the current Drift back end implementation that have been described so far.

The only thing that so far has been shown on in the examples but not yet talked about is the `Update` queue, which is for example shown in Fig.33 and is implemented as a RabbitMQ queue compared to the `Error` queue, a Kafka queue.
The reason why the `Update` queue has not been introduced yet is because it is mostly irrelevant to the overall back end system implementation. Its sole purpose is to serve as an event interface for any front end implementation like the Drift shell. Any action that is undertaken by the back end system is sent as an event to the `Update` queue. So whenever a worker receives a new tasks, or finishes as task (with either `EOF` or `ERR`) or restarts a task, it sends a short event description to the `Update`. These can then be used for example by any visual front ends to dynamically update the visualization of

the system that is shown to the user, as will be described in the next chapter when the *Drift UI* will be introduced.

4.4 Drift User Interface

To complete the implementation of the *Drift* programming environment, this section will introduce the system's visualization. The idea for this visualization is heavily based on the concept of *immediate feedback* that was introduced in chapter 3.6 as well as on the original idea of a *language of the system* as introduced in chapter 3.1.
Therefor the overall goal of the system's visualization is to give the user a bird's-eye perspective of the overall system. This overview should show all the data dependencies that have been built-up by the user but should also dynamically react to either new instructions by the user but also new events in the system itself, e.g. the finishing of a task.
Furthermore this visual representation should not replace the original shell. It was not supposed to be a visual language but rather a graphical representation of the data dependencies created by the user. Using only a textual interface like the shell, makes it *easy* for the user the built-up large structures with only a few lines of commands. This can mislead the user into thinking that the system is still small and managable when in reality it is a complex dependency graph containing multiple interconnected data dependencies that are not easy to grasp. Therefor the visual representation is thought of as a guideline for the user, mirroring her commands and showing their immediate effects to the overall system.

Since other workflow systems use a direct acyclic graph (DAG) to encapsulate and represent the data dependencies as described by the user, the first idea was to also use a DAG as a dynamic representation of the system. Whenever the user would enter a new command, a new vertex would appear in the DAG and whenever a task finished the system would insert the event into the `Update` queue that was introduced in chapter 4.3 and the DAG would eliminate the corresponding vertex and update its view.
At first this was implemented in exactly that fashion. Unfortunately this forced the decision whether vertices in the DAG represent services and where therefor named after the name of the invoked service, or whether vertices represent the data, being named after the names given by the user. Although it would've been possible to allow the user to dynamically switch between both views, the DAG itself was discarded as a fitting visual representation. A DAG only describes the dependency aspect of its vertices. If there is a path from vertex A to vertex B then there is a dependency between A and B. If no such path exists, A are B independent of each other. That is what is captured by the DAG.
However, the Drift language's core concepts are *data* and *functionality*. Therefor both concepts exist in the language: *names* (data) and *services* (functionality). So a visual representation would need to have two different concepts in order to represent both, names and services. Another important aspect of the services orchestrated by the Drift system is their locality in terms of their inputs, outputs and state, as was introduced in

chapter 4.3. This means that no single service invocation ever has nor needs *any* global information or global state. Its execution and its outputs only ever depend on the data contained in its input queues and output is only ever produced to its own output queue. As was already mentioned in the last chapter, this is heavily reminiscent of transitions in a Petri Net because they too only ever consume from their input places and only ever produce to their output places.

Interestingly, since Petri Nets too are based on two core concepts, namely *places* and *transitions*, they offer different syntax to talk about both concepts. Figure 37 shows a very rough sketch of the idea of a Petri Net. The Petri Net syntax uses circles to represent places and squares to represent transitions. Therefor Fig.37 shows a transition A consuming from two input places and producing to three *different* output places, $1, 2$ and 3. These places are then also input places for the following transitions B, C and D which each only depend on their single corresponding input place.

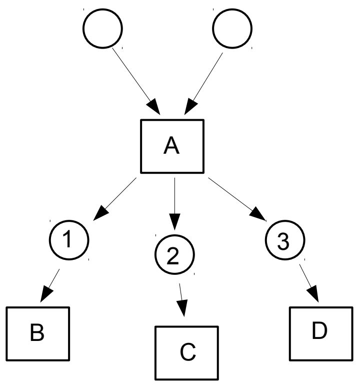


Figure 37: Rough sketch example of a Petri Net.

This works perfectly fine when the Petri Net is static and modelled without any dynamically changing parts. However, given the goal of immediate feedback the Drift shell uses eager-evaluation and the system is composed *along the way* whenever the user enters new commands. Therefor it is not possible to infer or predict how many consumers a given service invocation will have. In the worst-case scenario all available workers will at some point consume from the same input data source at the same time. That means that building the system defensively one would have to add as many output places to any service invocation as there are workers in the system.

This makes sense in the formalism and realm of Petri Nets because here, as shown in Fig.37, transition B could really consume (meaning eliminate) tokens from place 1 but not from place 2. Therefor the *firing* of transition B is completely independent from the firing of transition C because their inputs are different. If one would change the given model, having transition A produce to only a single output place and having transitions B, C and D all consume from this single place would also drastically change the semantics of the net. Because in that case tokens consumed by B would no longer be available to transitions C and D. All three transitions would naturally fight for the tokens because consuming a token is a destructable action.

However, given that the output queues in the Drift back end are implemented using *immutable* Apache Kafka queues, multiple consumers can consume from the same queue concurrently without any contention. Therefore, instead of modelling the producer-consumer relationships between A and B, C and D with different independent output places, a single output place can be used, as shown by Fig.38.

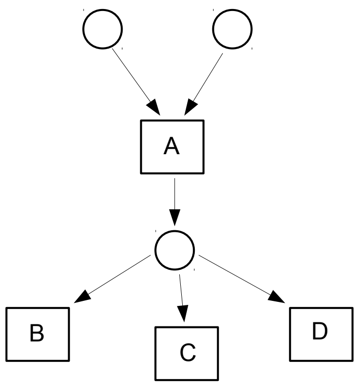


Figure 38: Minimized net with *different* semantics.

It cannot be highlighted enough that using the original Petri Net formalism the nets presented in Fig.37 and 38 would have drastically different semantics as explained earlier. However, the Petri Net *syntax* supports exactly the same concepts as the Drift language and is therefor a better fit in terms of visualizing a Drift system than a simple DAG, mapping names to places and services to transitions. So *only* the *visual* Petri Net syntax and not the formalism itself is borrowed in order to visualize a drift system.
So in order implement such a visualization the browser was chosen as a common platform that allows for ease of use and requires no installation whatsoever, using HTML, CSS and JavaScript for implementing the website/web interface.
Figure 39 shows the final prototype of that user interface. As one can see the page is split into three vertical parts. On the left hand side there is a full Drift shell implementation in the browser. This embedded shell works exactly like the stand alone shell that was implemented in Java, as was presented in chapter 4.3.1 and was implemented using the JavaScript library *jQuery Terminal* [106].

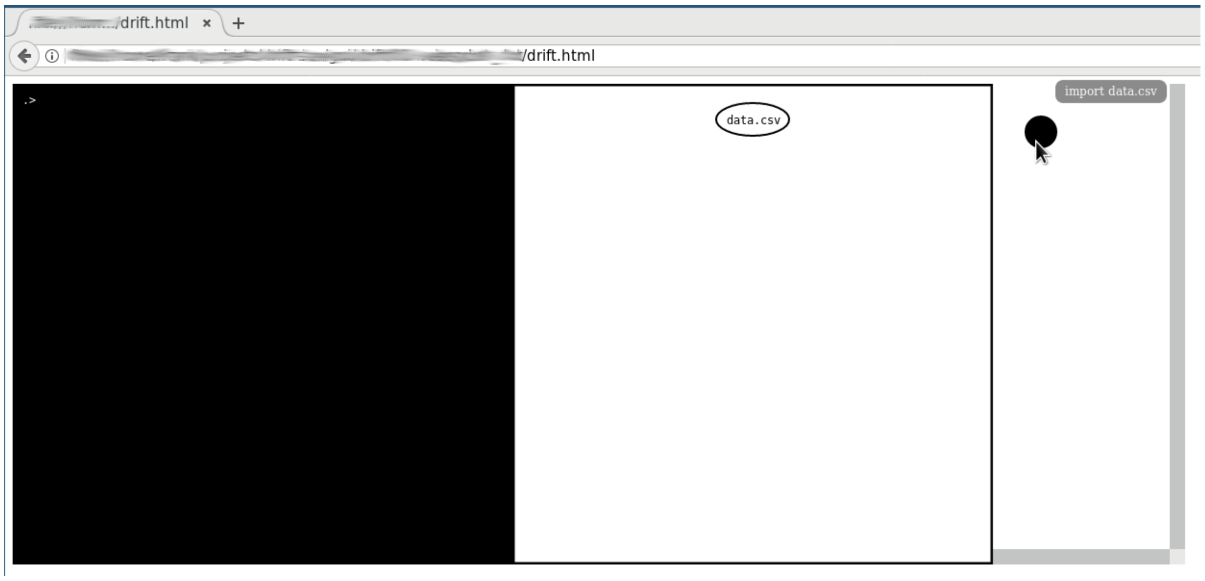
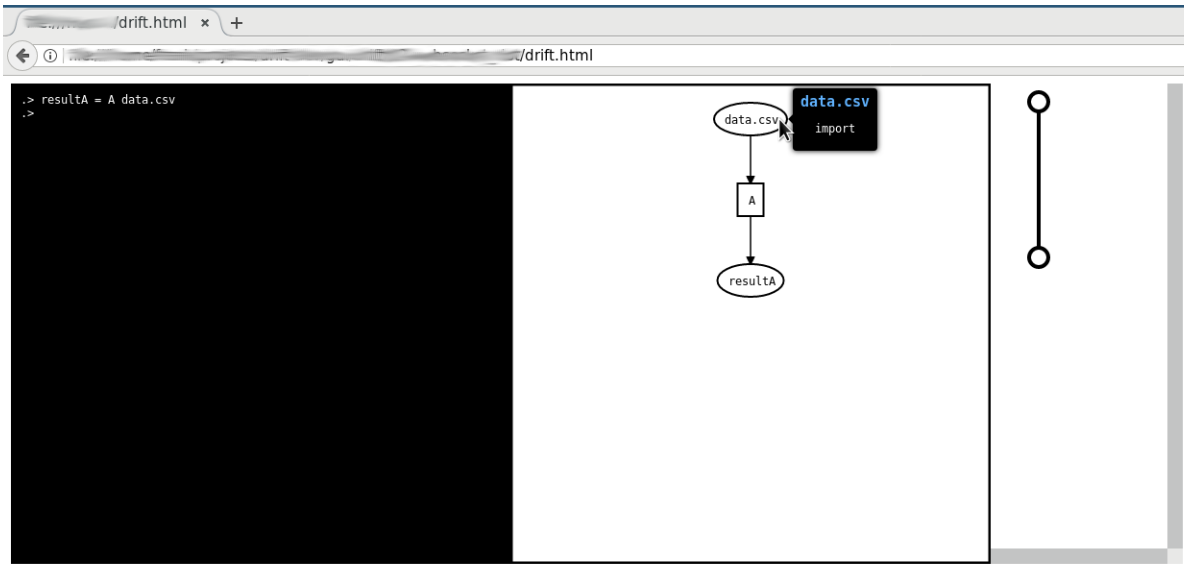


Figure 39: Initial state of the system after the import of the file `data.csv`. Imports are also stored as events on the the time line.

In order for this shell to communicate with the Drift backend from within the browser a small websocket server was implemented in Java which only serves as a gateway, forwarding commands from the shell to the back end and pushing new messages from the `Update` queue that was presented in chapter 4.3.4 to the browser using the widely known `dot` file format to describe the new graph that should be printed which resulted from the system update [107], [108]. Since this being a full Drift shell, all the features that were presented in chapter 4.3.1 are present here as well. Only the `import` section works differently. Because one cannot access arbitrary local files from within JavaScript and the browser, the user needs to select to files which shall be imported via a GUI wizard, which is the same that is used for whenever files need to be uploaded on any other website. User selected files can then be read by the browser and therefor their content can be uploaded to the Drfit system.

Next to the shell there is an HTML5 canvas containing the graphical representation of the system using the already introduced Petri Net syntax. Since most Drift sessions start by importing some local files into the system, the graph printed on the canvas starts out containing only unconnected places (circles) with the given name written inside of them. The graph is constructed in-memory using a JavaScript library called *graphlib.js*. The layout of the constructed graph on the canvas is automatically generated using the *dagre.js* library and the final printing of the graph onto the canvas is done using *d3.js* [109], [110], [111].

Any new command by the user immediately updates the graph shown on the canvas. Services used in the command are drawn as transitions (squares) and connected to their input places accordingly. If the result of the command is bound to a name, a new place is drawn and connected to the service as its output place, as illustrated by Fig.40. Also a new snapshot of the current graph is stored and available as an event in the time line for later review. Furthermore each graph element also offers a mouse-hover feature, where information about the service or name is available, e.g. if it was created via an import.



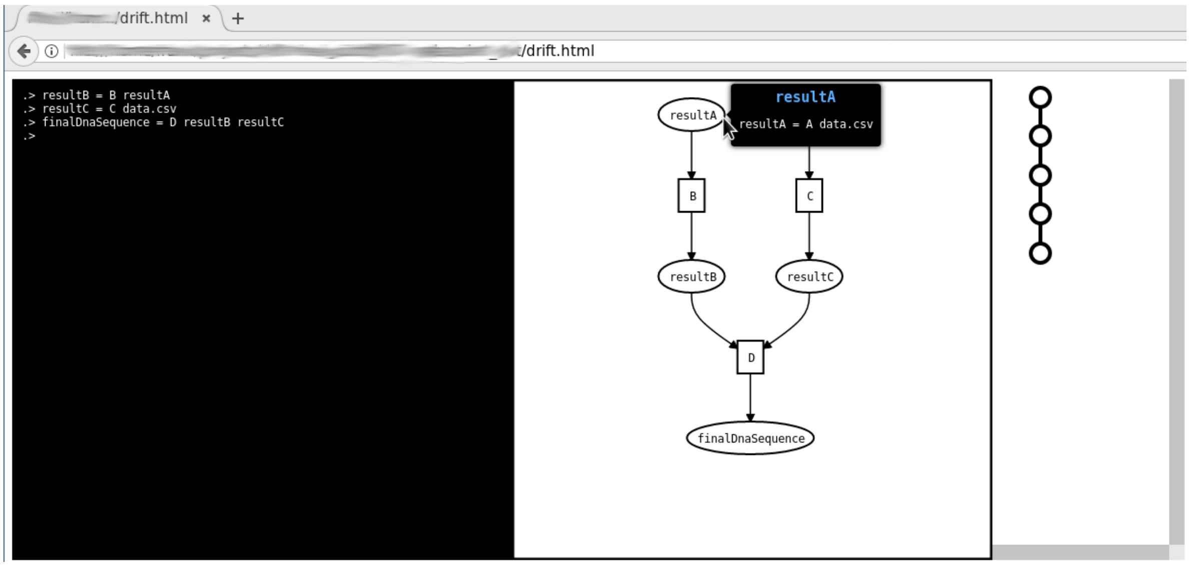
```
.> resultA = A data.csv
.>
```

Figure 40: A simple service invocation which invokes the service in the back and also immediately updates the system view.

If a service has finished producing its output by sending and `EOF` token to its output queue, the update sent to the `Update` queue will trigger the removal of the edge connecting the service to its output place. The same thing happens when a service consumes an `EOF` token for one of its input queues. It sends an event notification to the `Update` queue which will send a new graph to the browser missing the edge between that service and the corresponding input place.

This means that the user can watch and observe when data has been finally consume or produced without querying its content. However, live-querying the streamed data in the shell is still possible.

But this liveness of the system and its graphical representation also has its drawbacks. Imagine for example that the user entered a lot of commands, quickly building up a large graph. Because of the spacial limitations of the shell, some of the earlier commands might have already been pushed out of the scope of the shell terminal. If the user now wants to look up what command created a specific place, she would have to scroll up, looking for the name and command that created *that* particular place.

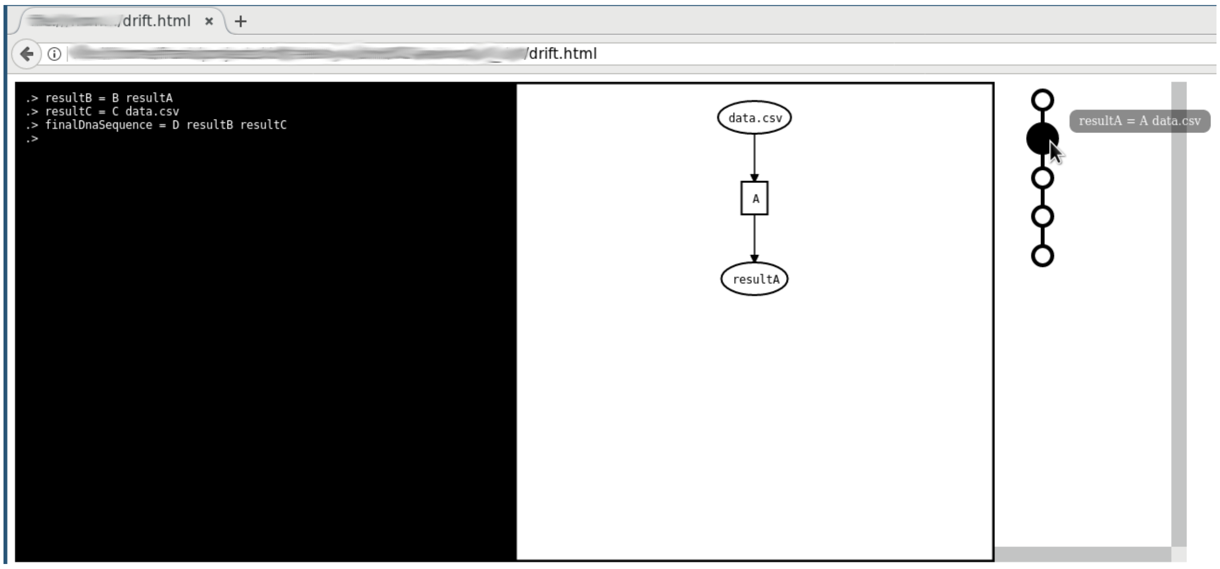


```
.> resultB = B resultA
.> resultC = C data.csv
.> finalDnaSequence = D resultB resultC
.>
```

Figure 41: Example a more complex dependency graph but because service `A` has already finished, the source of `resultA` is no longer shown in the graph.

This is illustrated by Fig.41. Here the name `resultA` does not contain a typical file ending like `.csv`. Therefor it must have been created by an earlier service invocation and not an import. But because the service that created this name has already finished, it was deleted from the system graph and only its result, `resultA`, remains. Since then the user has used the result of her earlier invocation and built-up a dependency tree in which some parts consume data from that earlier result.

In order to avoid having to search for the command that created the name `resultA`, the graph provides a mouse-hover feature. If the user hovers over a place, a pop-up will contain the command that created that place, as is shown in Fig.41. This is the graphical equivalent of the history operators `?` and `??` that were introduced in chapter 4.2. Of course the mouse-hover pop-up will contain the original command and input place names, even when some of the original inputs have now been overwritten by newer commands.



```
.> resultB = B resultA
.> resultC = C data.csv
.> finalDnaSequence = D resultB resultC
.>
```

Figure 42: Example showing how a mouse-over over the event and therefor command that created the resultA name shows the snapshot of the system graph at the time the command was issued.

In order to not only view the command that created a name but also review the system state at the time the command that created that name was issued, the user can hover over the corresponding event in the time line and the canvas will immediately show the snapshot of the graph that was stored. Therefor old system states are stored and the user the review the history of the whole system, in the same way that a user can review old system states in *Git* as well as *Datomic*, as was introduced in chapter 3.5 and 3.4. The idea of the event time line is of course heavily inspired by the time bar presented by Bret Victor, as was introduced in chapter 3.6.

So this concludes the Drift user interface, which after the Drift language and shell and the Drift back end including the immutable Drift file system, is the last part of the *Drift Programming Environment*. The Drift UI is a graphical web interface which offers the same shell that could also be used as a stand alone shell but additionally offers a graphical representation of the whole system using the widely known Petri Net syntax with a slightly different semantics based on immutable data.

The system graph is updated either by commands of the user or by updates from the individual and independent components of the distributed system. Therefore, although the individual components of the system do *not* maintain any global state, the UI can still create the system overview from the individual pieces of information.

5 Summary

In summary this thesis proposed a novel programming environment in the area of distributed programming, called the *Drift* programming environment. The environment consists of *i*) the Drift language, a compact and abstract distributed coordination language focussing on the concepts of *data* and *functionality* and emphasising the importance of *names* for conveying meaning in programming *ii*) the Drift shell an implementation of the Drift language in form of an interpreter, *iii*) the Drift distributed execution environment including the Drift FS, a novel distributed and immutable file system based on distributed message queues and *iv*) the Drift user interface, a web interface featuring the Drift shell and a graphical representation of the Drift execution environment based on the well-known Petri Net syntax.

To set the context for the work presented here, chapter 2 first introduced the concept of *distributed programming* by giving a short summary of the history and evolution of programming languages and their adaption to fundamental changes of the underlying hardware. This was done by first looking at the adaption of programming language constructs regarding the change from single-core to multi-core CPUs in chapter 2.1 and continued with looking at another adaption, namely from single machine computing to distributed computing using multiple machines connected via a network. Therefore chapter 2.2 introduced the approach of *distributed objects* and chapter 2.3 introduced the *actor model*, using examples to highlight their advantages but also their shortcomings.

In order to deal with these shortcomings and to finally introduce the Drift project, chapter 3 first introduced different software projects and people that created these projects as well as their ideas and philosophies regarding software, system design and system architecture. Although these projects might seem unrelated at first glance, they all contributed to the ideas and design decisions of the overall Drift project. Therefore, for each project, the relevant aspects regarding their influence on the Drift project were introduced and highlighted.
This was done by first introducing Rich Hickey and his ideas of the *language of the system* and *value-oriented programming* in chapter 3.1. These ideas are then picked up in chapter 3.2, by introducing *Cuneiform*, a functional scientific workflow language. Cuneiform is developed here at the *Humboldt-Universität zu Berlin* and combines the ideas of immutable data from the real of functional programming with the bird's-eye perspective of describing a system from the scientific workflow community.
Chapter 3.3 then took a closer look at history and introduced the UNIX time sharing system including its command line shell. Given the new concepts of processes, a UNIX system can be seen as a system of independent processes working concurrently. Therefore the UNIX shell becomes a coordination language, orchestrating these processes and their data flow, much like a workflow language. Chapter 3.3 then went on by introducing a mapping between the language concepts the UNIX shell and language concepts known from functional programming languages like Cuneiform, showing an almost one-to-one correspondence except for the global shared mutable state which is

the UNIX file system.
Chapters 3.4 and 3.5 then introduced two rather new projects, namely *Datomic* and *Git* which both deal with the aspect of *immutable data.* Datomic, being a distributed and immutable facts log, presents its user the illusion of immutability though being implemented on top of already existing mutable distributed storage solutions like key-value stores or SQL data bases. Git on the other hand is implemented as an immutable content-addressable file system which presents its user with a mutable and stateful interface based on ordinary files and file access. Considering these two opposing projects challenges the widely-accepted believe of an insurmountable divide between both approaches: mutability and statefulness versus immutability and functional programming.
Lastly, chapter 3.6 introduced Bret Victor and his idea of *immediate feedback*, which, in the field of programming, can be seen as the goal of creating a tight feedback loop between the programmer typing code or issueing commands and her being presented with the system's response. This ties with the concept of the shell as introduced in chapter 3.3 which immediately runs its user's commands, as opposed to the more traditional approach of first writing all of the code, compiling it and then running it.

Chapter 4 then presented the Drift project. It started by first focussing on communication as the essence of distributed programming and highlighting the importance of *names* for conveying the meaning and semantics of a program in chapter 4.1. Then, the imperative and stateful Drift coordination language was presented in chapter 4.2, using multiple code examples to gradually introduce the language concepts based around names, namespaces and services. Alongside the language itself, the shell was introduced, which features some of the language keywords as shell built-ins as well as letting the user traverse the names and namespaces like he would a typical UNIX file system.
Then the Drift back end implementation was presented including the Drift file system, which is done by gradually building up the system from the basic communication between shell and system up to how different message queues are used for either data and signaling and how the file system that is presented to user was implemented using these message queues, especially given the mapping of directories to namespaces. Furthermore, the error model and fault tolerance mechanisms used by the Drift system were presented, using a control token mechanism as well as already established recovery software.

Lastly the Drift UI was presented, which was implemented in the form of a web interface for modern browsers. It featured a full fledged Drift shell as well as a graphical representation of the system orchestrated by the user. Instead of a direct acyclic graph (DAG) which is often used by similar system, the also widely-known Petri Net syntax with an alternative semantics was used. This allowed to naturally represent the Drift language concepts of *names* and *services* as places and transitions in the Petri Net syntax. Since this graphical representation is not only updated by commands of the user, but also by events in the system, e.g. finishing of a task, it is possible that elements are removed from the graph without user intervention. In order to allow the user to recap and reconstruct earlier system states a time bar was presented that presents all the events in the system and which allows the user to print older system graphs by simply mouse-hovering over the event entry in the time bar.

All the components presented in chapter 4 as a whole represent the *Drift Programming Environment*. A distributed orchestration system focused on coordinating the communication between its services, built upon the idea of distributed immutable data which allows for virtually no contention or coordination overhead and therefore no global coordinator.
Instead of sticking to this immutability even up to the user interface, forcing the user into the paradigm of functional programming, an imperative, stateful language is used, built on top of that immutable system, that allows the user to stay in the real of imperative programming if that feels more natural. To make dealing with this introduced state more managable, a graphical representation based on Petri Net syntax is used, that allows the user to immediately observe its actions and data but also allows to revisit earlier system states.

Any component of the *Drift Programming Environment* is a full fledged research topic in and by itself. For example, language design offers many aspects that have *not* been considered in the work presented here as will be discussed in chapter 6, but also the system implementation offers infinite possibilities for improvement in areas like scheduling, load balancing or garbage collection, just to name a few.

However, the main contribution of the work that was presented here is not seen as the quality of each individual component. All of the implemented components should rather be seen as prototypes and proof-of-concepts that explore novel ideas in their individual areas. The main contribution of the *Drift Programming Environment* is the collection and integration of all of these prototype components into one large system and environment, merging and composing them in a way that they as a collection fit the overall goal of this work and represent more than just their overall sum, challenging the widely-accepted believe that there is a fundamental difference between imperative, e.g. stateful languages and functional, e.g. immutable languages, as has been *the* dominating discussion for the last decades in the area of programming language design.

6 Future Work

As was summarized by the last chapter, the *Drift Programming Environment* is a collection of different components ranging from a coordination language and interpreter to an abstract distributed file system and graphical user interface.
Since each of these components should be seen as a prototype or proof-of-concept, they naturally leave much to be desired and explored. So this chapter will discuss further developments of each component which were deemed out of the scope of the

work presented here due to time and other resource limitations.

Starting with the Drift coordination language, one of the next steps would be to provide a formal specification of the language. So far the language has only been defined in terms of its implementation and some prose, trying to explain its concepts. A formal semantics would be needed in order to decouple the definition of the language from its current implementation but would probably also allow further reduction and understanding of the language constructs. Moreover, the semantics of the language would naturally lend to also providing a type system, which could further enhance the meaning given appropriate names and allow for sound guarantees in terms of stuckness or error handling.
Furthermore, in order to fully evaluate the helpfullness of the language overall, user studies would be needed to assess whether or not the rather abstract and concise syntax and semantics of the language suffice in order for users to orchestrate their distributed systems.
The Drift shell could be improved in terms of its error output but also in terms of its input handling, providing features like autocompletion for commands or a command history. Additionally, as was already mentioned in chapter 4.3.1 it could make sense to include a service info cache into the shell, which would reduce the number of round trips to the service registry in order to assess whether the service used in a given command are used correctly.

The Drift back end could be improved immensely at almost any aspect. One of the most obvious of these aspects is the inability of the system to deal with its own resource limitations. As was already hinted at in chapter 3.1, systems build on the idea of immutable data tend to not scale in terms of space. In this case the only space that accumulates over time is memory used to store data (messages) in the message queues. The system assumes that there will always be enough memory to allow the user to create new names and therefore spawn new messages while indefinitely keeping every other message queue ever created.
This could be dealt with by garbage collecting, e.g. deleting, queues where every active user session has either deleted or reassigned the name that was once bound to that particular queue (reference counting). This would eliminate the caching effect and would force future tasks to recomputate the content of the queue even though its result was already computed once, but would on the other hand allow the system to continue to function and not run into memory issues.
Another aspect of garbage collection concerns services that are running even though their result is currently never used. Imagine for example a user issuing a command, immediately querying its output and seeing that she made a mistake. The user can then immediate reassign the name to a new service invocation but the old service invocation is still running, even though its result will probably never be used. So canceling the tasks that are executed unnecessarily would free up the workers to execute tasks that are actually needed and would therefore increase the throughput of the system.

Another issue concerning workers is scheduling, which of course is a huge research topic in and by itself. The current implementation uses round-robin scheduling by the task queue, distributing incoming tasks evenly among all available workers. This was deemed reasonable under the assumption of a homogenous cluster in terms of hardware. But given slight variations in terms of available CPU power, memory or bandwith, this could of course be replaced by a scheduling component that takes all these different aspects into account. Maybe even learns or adopts to different workloads on the fly.
Additionally another aspect of the scheduling concerns the order in which tasks are scheduled. The current implementation treats all tasks as equal, whether or not they are considered producers or consumers. So for the sake of the example one could imagine a system with only two workers W_1, W_2 but 4 tasks queued in the order: C_2, P_2, C_1, P_1 with the right most task, P_1, being the next to be executed. Given the two workers, W_1 would execute P_1 and W_2 would execute C_1, effectively idling, waiting for input from W_1.
However, if the system would schedule all the producing tasks first, W_1 would still execute P_1 but W_2 would now execute P_2, therefore really computing in parallel and always guaranteeing that whenever a consumer task is executed the input data for that consumer is always already available. But this would not only mean reordering the task queue, which might not be that difficult but also infering which task is categorized as a producer and which as a consumer, which might not at all be trivial or even possible to figure out.

Given the web interface I personally don't see that much room for improvement. Of course, this being only a prototype and me not being a web developer, probably every button or element could be implemented nicer and prettier given all the features the modern web has to offer. However, the only feature I would actually really add besides much overdue cosmetic work, is for the user to be able to write certain system snapshot to a file and maybe load them from a file in order to resume from that snapshort, either for backup purposes or simply convenience when moving between systems. But given the sandbox nature of the browser and its JavaScript engine due to important security concerns when browsing the web, this seems rather unlikely.

These conclude the major implementational weak points which would likely need to be addressed in any further development of the language and system.
Otherwise I can only encourage future projects to challenge wide-spread believes about imperative or functional languages or mutable and immutable data, especially in the context of distributed system, by trying to emulate one with the other in order to gain further knowledge and understanding about which concepts are needed on which layer of the system stack to fully utilize their advantages and neglect their drawbacks.

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].

[26] Wikipedia, “Benevolent dictator for life — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/Benevolent_dictator_for_life, 2017. [Online; accessed 08-February-2017].

[27] R. Hickey, “The clojure programming language,” in *Proceedings of the 2008 symposium on Dynamic languages*, p. 1, ACM, 2008.

[28] R. Hickey, “Clojure,” *h ttp://clojure. org*, 2010.

[29] https://clojure.org/, 2017. [Online; accessed 08-February-2017].

[30] https://clojure.org/about/rationale, 2017. [Online; accessed 08-February-2017].

[31] P. H. Winston and B. K. Horn, “Lisp,” 1986.

[32] G. Steele, *Common LISP: the language*. Elsevier, 1990.

[33] Wikipedia, "List of programming languages by type — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/List_of_programming_languages_by_type#Functional_languages, 2017. [Online; accessed 08-February-2017].

[34] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[35] B. Keck, C. Nguyen, and L. Radden, "Persistent data structures,"

[36] InfoQ. https://www.infoq.com/profile/Rich-Hickey, 2017. [Online; accessed 08-February-2017].

[37] D. Walters. https://changelog.com/posts/rich-hickeys-greatest-hits, 2017. [Online; accessed 08-February-2017].

[38] R. Hickey, "Rich Hickey — the language of the system." https://www.youtube.com/watch?v=ROor6_NGIWU, 2012. [Online; accessed 08-February-2017].

[39] http://2016.clojure-conj.org/, 2016. [Online; accessed 08-February-2017].

[40] F. S. Foundation. https://www.gnu.org/software/libc/, 2017. [Online; accessed 08-February-2017].

[41] R. Hickey, "The value of values - Rich Hickey." https://www.infoq.com/presentations/Value-Values, 2012. [Online; accessed 08-February-2017].

[42] R. Hickey, "Rich Hickey — the value of values." https://www.youtube.com/watch?v=-6BsiVyC1kM, 2012. [Online; accessed 08-February-2017].

[43] G. Conference. https://gotocon.com/cph-2012/, 2017. [Online; accessed 08-February-2017].

[44] J. Conference. https://en.xing-events.com/jaxconf2012.html, 2012. [Online; accessed 08-February-2017].

[45] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[46] A. Barker and J. Van Hemert, "Scientific workflow: a survey and research directions," in *International Conference on Parallel Processing and Applied Mathematics*, pp. 746–753, Springer, 2007.

[47] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, *et al.*, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[48] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *undefined*, pp. 11–20, Springer, 2004.

[49] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[50] J. Brandt, M. Bux, and U. Leser, "A functional language for large scale scientific data analysis," in *BeyongMR, ICDT/EDBT Workshop*, 2015.

[51] M. Bux, J. Brandt, C. Lipka, K. Hakimzadeh, J. Dowling, and U. Leser, "Saasfee: scalable scientific workflow execution engine," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1892–1895, 2015.

[52] A. Church, "A set of postulates for the foundation of logic," *Annals of mathematics*, pp. 346–366, 1932.

[53] H. P. Barendregt *et al.*, *The lambda calculus*, vol. 3. North-Holland Amsterdam, 1984.

[54] O. Ritchie and K. Thompson, "The unix time-sharing system," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1905–1929, 1978.

[55] M. J. Bach *et al.*, *The design of the UNIX operating system*, vol. 1. Prentice-Hall Englewood Cliffs, NJ, 1986.

[56] B. W. Kernighan and D. M. Ritchie, *The C programming language.* 2006.

[57] C. Ramey, "Bash, the bourne- again shell," in *Proceedings of The Romanian Open Systems Conference & Exhibition (ROSE'94)*, pp. 3–5, 1994.

[58] Wikipedia, "Eager evaluation — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/wiki/Eager_evaluation`, 2017. [Online; accessed 16-February-2017].

[59] Cognitect, "Datomic: A distributed database for serious information systems.." `http://cognitect.com/datomic`, 2016. [Online; accessed 16-February-2017].

[60] R. Hickey, "The architecture of datomic - rich hickey." `https://www.infoq.com/articles/Architecture-Datomic`, 2013. [Online; accessed 16-February-2017].

[61] R. Hickey, "The datomic information model - rich hickey." `https://www.infoq.com/articles/Datomic-Information-Model`, 2013. [Online; accessed 16-February-2017].

[62] R. Hickey, “The database as a value - rich hickey.” https://www.infoq.com/presentations/Datomic-Database-Value, 2012. [Online; accessed 16-February-2017].

[63] R. Hickey, “The functional database - rich hickey.” https://www.infoq.com/presentations/datomic-functional-database, 2012. [Online; accessed 16-February-2017].

[64] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[65] C. J. Fidge, “Timestamps in message-passing systems that preserve the partial ordering,” 1987.

[66] F. Mattern *et al.*, “Virtual time and global states of distributed systems,” *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[67] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[68] G. Project. https://git-scm.com/sfc, 2017. [Online; accessed 17-February-2017].

[69] Wikipedia, “Git — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/Git, 2017. [Online; accessed 17-February-2017].

[70] L. Torvalds, “Google tech talk: Linus torvalds on git.” https://www.youtube.com/watch?v=4XpnKHJAok8, 2007. [Online; accessed 17-February-2017].

[71] G. Project. https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git, 2017. [Online; accessed 17-February-2017].

[72] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O’Reilly Media, Inc.", 2012.

[73] G. Project. https://git-scm.com/book/en/v1/Git-Internals-Git-Objects, 2017. [Online; accessed 17-February-2017].

[74] W. M. Van Der Aalst and A. H. Ter Hofstede, “Yawl: yet another workflow language,” *Information systems*, vol. 30, no. 4, pp. 245–275, 2005.

[75]

[76] Wikipedia, “Bret Victor — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/Bret_Victor, 2017. [Online; accessed 20-February-2017].

[77] B. Victor, “Bret Victor — inventing on principle.” https://vimeo.com/36579366, 2012. [Online; accessed 20-February-2017].

[78] B. Victor, “Bret Victor — stop drawing dead fish.” https://vimeo.com/64895205, 2013. [Online; accessed 20-February-2017].

[79] B. Victor, “Bret Victor — the future of programming.” https://www.youtube.com/watch?v=8pTEmbeENF4, 2013. [Online; accessed 20-February-2017].

[80] B. Victor, “Bret Victor — media for thinking the unthinkable.” https://vimeo.com/67076984, 2013. [Online; accessed 20-February-2017].

[81] A. Church and J. B. Rosser, “Some properties of conversion,” *Transactions of the American Mathematical Society*, vol. 39, no. 3, pp. 472–482, 1936.

[82] Wikipedia, “Church–Rosser theorem — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/Church%E2%80%93Rosser_theorem, 2017. [Online; accessed 21-February-2017].

[83] B. C. Pierce, *Types and programming languages.* MIT press, 2002.

[84] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *Data Engineering*, p. 28, 2015.

[85] Wikipedia, “Let expression — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/Let_expression#Let_definition_defined_from_Lambda_calculus, 2017. [Online; accessed 23-February-2017].

[86] P. J. Landin, “The mechanical evaluation of expressions,” *The Computer Journal*, vol. 6, no. 4, pp. 308–320, 1964.

[87] Wikipedia, “Closure (computer programming) — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/Closure_(computer_programming), 2017. [Online; accessed 24-February-2017].

[88]

[89] N. Sarnak and R. E. Tarjan, “Planar point location using persistent search trees,” *Communications of the ACM*, vol. 29, no. 7, pp. 669–679, 1986.

[90] H. Kaplan, “Persistent data structures,” in *IN HANDBOOK ON DATA STRUCTURES AND APPLICATIONS, CRC PRESS 2001, DINESH MEHTA AND SARTAJ SAHNI (EDITORS) BOROUJERDI, A., AND MORET, BME," PERSISTENCY IN COMPUTATIONAL GEOMETRY," PROC. 7TH CANADIAN CONF. COMP. GEOMETRY, QUEBEC*, Citeseer, 1995.

[91] R. C. Merkle, “A digital signature based on a conventional encryption function,” in *Conference on the Theory and Application of Cryptographic Techniques*, pp. 369–378, Springer, 1987.

[92] Wikipedia, “Merkle tree — Wikipedia, the free encyclopedia.” `https://en.wikipedia.org/wiki/Merkle_tree`, 2017. [Online; accessed 24-February-2017].

[93] Pivotal, “Rabbitmq.” `https://www.rabbitmq.com/`, 2017. [Online; accessed 24-February-2017].

[94] Wikipedia, “RabbitMQ — Wikipedia, the free encyclopedia.” `https://en.wikipedia.org/wiki/RabbitMQ`, 2017. [Online; accessed 24-February-2017].

[95] A. Videla and J. J. Williams, *RabbitMQ in action.* Manning, 2012.

[96] C. Inc., “Apache kafka — a distributed streaming platform.” `https://kafka.apache.org/intro`, 2017. [Online; accessed 27-February-2017].

[97] R. Ranjan, “Streaming big data processing in datacenter clouds,” *IEEE Cloud Computing*, vol. 1, no. 1, pp. 78–83, 2014.

[98] N. Garg, *Apache Kafka.* Packt Publishing Ltd, 2013.

[99] M. Inc., “Marathon — a container orchestration platform for mesos and dc/os.” `http://mesosphere.github.io/marathon/`, 2017. [Online; accessed 27-February-2017].

[100] M. Inc., “Mesosphere — hybrid cloud platform.” `https://mesosphere.com/`, 2017. [Online; accessed 27-February-2017].

[101] M. Inc., “Apache mesos — a distributed systems kernel.” `https://mesos.apache.org/`, 2017. [Online; accessed 27-February-2017].

[102] A. ZooKeeper, “Apache zookeeper.” `https://zookeeper.apache.org/`, 2017. [Online; accessed 27-February-2017].

[103] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems.,” in *USENIX annual technical conference*, vol. 8, p. 9, 2010.

[104] W. Reisig, *Petri nets: an introduction*, vol. 4. Springer Science & Business Media, 2012.

[105] Wikipedia, “Petri net — Wikipedia, the free encyclopedia.” `https://en.wikipedia.org/wiki/Petri_net`, 2017. [Online; accessed 27-February-2017].

[106] J. Jankiewicz, “Jquery terminal — jquery terminal emulator.” `http://terminal.jcubic.pl/`, 2017. [Online; accessed 02-March-2017].

[107] “The dot language.” http://www.graphviz.org/doc/info/lang.html, 2017. [Online; accessed 01-March-2017].

[108] Wikipedia, “DOT (graph description language) — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/DOT_(graph_description_language), 2017. [Online; accessed 01-March-2017].

[109] C. Pettitt, “Graphlib — a directed multi-graph library for javascript.” https://github.com/cpettitt/graphlib, 2017. [Online; accessed 01-March-2017].

[110] C. Pettitt, “Dagre — directed graph renderer for javascript.” https://github.com/cpettitt/dagre, 2017. [Online; accessed 01-March-2017].

[111] M. Bostock, “D3 — data-driven documents.” https://d3js.org/, 2017. [Online; accessed 01-March-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 May 12, 2017 ....