Derflow: Distributed Deterministic Dataflow Programming for Erlang

Manuel Bravo
Université catholique de Louvain
angel.bravo@uclouvain.be

Zhongmiao Li
Université catholique de Louvain
zhongmiao.li@uclouvain.be

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

Abstract

Erlang implements a message-passing execution model in which concurrent processes send each other messages asynchronously. This model is inherently non-deterministic: a process can receive messages sent by any process which knows its process identifier, leading to an exponential number of possible executions based on the number messages received. Concurrent programs in non-deterministic languages are notoriously hard to prove correct and have led to well-known disasters.

Furthermore, Erlang natively provides distribution and process clustering. This enables processes to asynchronously communicate between different virtual machines across the network, which increases the potential non-determinism.

We propose a new execution model for Erlang, "Deterministic Dataflow Programming", based on a highly available, scalable single-assignment data store implemented on top of the $riak\_core$ distributed systems framework. This execution model provides concurrent communication between Erlang processes, yet has no observable non-determinism. Given the same input values, a deterministic dataflow program will always return the same output values, or never return; liveness under failures is sacrificed to ensure safety. Our proposal provides a distributed deterministic dataflow solution that operates transparently over distributed Erlang, providing the ability to have highly-available, fault-tolerant, deterministic computations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Dynamo; Erlang; Riak

1. Introduction

Erlang implements a message-passing execution model in which concurrent processes send each other asynchronous messages. This model is inherently non-deterministic, in that a process can receive messages sent by any process which knows its process identifier, leading to an exponential number of possible executions based on the number of messages received. Concurrent programs in non-deterministic languages, are notoriously hard to prove correct, and have lead to many well-known disasters. [15]

When reasoning about the correctness of our programs, we treat every message received by a process as a 'choice'. A series of these 'choices' define one execution of a program. Given this, to prove a program is correct requires proving that each of these executions are correct; that is, for each execution all possible inputs are able to be processed resulting in termination. While there is work underway on making this approach more viable [2], we believe that limiting the ability to write non-deterministic code provides a reasonable alternative to exhaustively checking our applications for correctness.

In addition, Erlang natively provides distribution and clustering as part of the runtime environment. This provides the ability to have processes asynchronously communicate across the network between different instances of the virtual machine. When using asynchronous communication across the network, one can provide even fewer guarantees regarding message delivery and reordering [18]. Erlang, in an effort to solve both of these problems, uses programming patterns and libraries (e.g. OTP) that are designed to reduce the number of choices and to maintain invariants for the remaining choices.

We propose a new execution model for Erlang, namely deterministic dataflow programming. This execution model provides concurrency, while also eliminating all observable non-determinism. Given the same input values, a program written in deterministic dataflow style will always return the same output values, or never return. These input values can be data streams as well, which is a natural generalization of functional programming to the concurrent setting. Our proposed solution provides a distributed deterministic dataflow solution which operates transparently over distributed Erlang, providing the ability to have highly-available, fault-tolerant, deterministic computations.

The major contributions of this paper are the following:

- Prototype implementation of a deterministic dataflow extension to Erlang called Derflow, with examples of its usage for common computations.

Erlang '14, September 5, 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-3038-1/14/09…$15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2633448.2633451

- Transparent distribution of computations, through the usage of the Dynamo-inspired [6] distributed systems framework, $riak\_core$. [3].

The remainder of this paper is organized as follows: Section 2 introduces background material related to distributed dataflow programming and the $riak\_core$ distribution model; Section 3 describes the semantics of Derflow; Section 4 discusses the implementation challenges; Section 5 discusses a few application of Derflow; then, Section 6 discusses integration with non-determinism; finally, Section 7 discusses future work and concludes the paper.

2. Background

The following subsections provide background on Dynamo, the $riak\_core$ library, and deterministic dataflow programming.

2.1 Dynamo

Consistent hashing, hash-space partitioning and a configurable data replication factor are the concepts critical for understanding $riak\_core$'s implementation of the Dynamo mode. We discuss in Section 4.2 how Derflow is built on top of $riak\_core$.

2.1.1 Consistent Hashing

The Amazon Dynamo paper describes a key-value based storage system made up of a cluster of nodes, where every node in the cluster stores some subset of the total data. To distribute this data, a consistent hashing algorithm applied to the data's key is then used to determine a token in the hash-space for where this data should be distributed.

2.1.2 Hash-Space Partitioning

The entirety of the hash space is then evenly divided between the nodes. Each even portion of the hash space is called a partition, and each partition is managed by a virtual node. Each physical node in the cluster hosts a number of virtual nodes, one for each partition assigned to that physical node. The hash resulting from running a key through the consistent hashing algorithm determines which partition is responsible for storing the data associated with that key.

2.1.3 Replication Factor

Dynamo replicates data on consecutive partitions. The replication factor N determines the number of replicas. When a key is mapped to a particular partition in the hash-space, the $(N-1)$ consecutive partitions are used to store replicas of the data. This collection of partitions is called the preference list or primaries.

2.1.4 Dynamic Cluster Membership

As the cluster grows and shrinks, partitions are redistributed to nodes, minimizing the amount of partitions that have to move between nodes to cut down on data transfer between nodes. This is a property of the consistent hashing algorithm described in section 2.1.1.

2.2 Deterministic dataflow programming

Deterministic dataflow was first proposed by Gilles Kahn in 1974, in a programming model that is now known as Kahn networks [12]. In 1977, a lazy version of this same model was proposed by Kahn and David MacQueen [13]. However, up until recently this model has never become part of mainstream concurrent programming. This may be due to either the model's inability to express non-determinism or the simultaneous invention of two other models for handling concurrent programming: the actor model (message passing) and monitors (shared state) [9, 10].

However, deterministic dataflow is now becoming a more important model in mainstream programming due to the increasing prominence of parallel computing, both in distributed computing and in multicore processors. Recent examples include the Oz deterministic dataflow execution model [19], the Akka library for concurrent and distributed programming in Scala [1, 20], and Ozma, which is a Scala language extension that adds deterministic dataflow [7].

3. Semantics of Derflow

This section presents the semantics of Derflow in four subsections. First, we focus on the primitive semantics which support deterministic dataflow; then, we introduce data streams, a programming technique that enriches deterministic dataflow. Then, we discuss a lazy execution extension. Finally, we discuss issues of failure handling.

3.1 Deterministic dataflow

The deterministic dataflow model uses a single-assignment store. This store is shared through all the processes that participate in the deterministic dataflow program. We represent the single-assignment store as:

$\sigma = \{x_1, \dots, x_n\}$

where x_i represents a variable declared in σ. The stored variables are called dataflow variables. Dataflow variables are assigned to dataflow values. A dataflow value is either an Erlang term or a previously declared dataflow variable.

Contrary to Erlang variables, a dataflow variable is allowed to be unbound. Thus, the possible states of a dataflow variable are the following: unbound, bound to a term, partially bound. The former is the initial state of a dataflow variable after is created. After the initial state, the dataflow variable can be either assigned to an Erlang term or to another dataflow variable. If the dataflow variable is assigned to another dataflow variable, we say that the variable is partially bound if the second dataflow variable is unbound. Figure 1 diagrams the states that a dataflow variable can visit.

Therefore, the following single-assignment dataflow store is consistent with the previous definitions:

$\sigma = \{x_1 = x_2, x_2 = \varnothing, x_3 = 5, x_4 = [a, b, c], \dots, x_n = 9\}$

where x_1 is bound to another dataflow variable (x_2), therefore, partially bound; x_2 is unbound and x_3; x_4 and x_n are bound to terms.

During the rest of the section, we use the following notation to specify the state of a dataflow variable:

- $x_i = \varnothing$: Variable x_i is unbound.
- $x_i = x_m$: Variable x_i is partially bound; therefore, it is assigned to another dataflow variable (x_m). This also implies that x_m is unbound.
- $x_i = v_i$: Variable x_i is bound to a term (v_i).

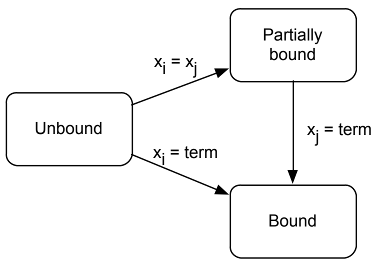

Figure 1. Dataflow variable state diagram from x_i perspective

- if x_i does not appear assigned to anything, it means it is not relevant to which kind of value is assigned.

Each dataflow variable has to keep some extra information in order to implement the primitive operations on which deterministic dataflow relies. A dataflow variable is composed as follows:

$x_i = \{$*value, bound_variables, waiting_processes*$\}$

where *value* is either empty or a dataflow value, *bound_variables* is a set of dataflow variables that are partially bound to x_i, and *waiting_processes* is a set of processes waiting for x_i to be bound. The set of waiting processes is used by the read and the bind primitive operations later described.

The deterministic dataflow model is an extension of the functional programming model with concurrency, dataflow variables and synchronization on them. The model then guarantees that under a particular input, a deterministic dataflow program will always produce the same result. It is well known that determinism is a desired property that simplifies the development of applications.

We now look at which primitives are required to transform a functional program into a deterministic dataflow program. The following primitives we aim to provide are: $declare()$, $bind(x, v)$ and $read(x)$.

$declare()$ creates a new dataflow variable into the single-assignment store. The operation returns the identifier of the newly created dataflow variable. More precisely, this operation can be expressed as follows:

- Before: $\sigma = \{x_1, \dots, x_n\}$
- $x_{n+1} = declare()$
  - create a unique dataflow variable x_{n+1}
  - store x_{n+1} into σ
- After: $\sigma = \{x_1, \dots, x_{n+1} = \varnothing\}$

bind(x_i*,* v_i*)* binds the dataflow variable x_i to the value v_i. More precisely, this operation can be expressed as follows:

- Before: $\sigma = \{x_1, \dots, x_i = \varnothing, \dots, x_n\}$
- $bind(x_i, v_i)$
  - $\forall p \in x_i.waiting\_processes,$ notify p
  - $\forall x \in x_i.bound\_variables, \quad bind(x, v_i)$
  - $x_i.value = v_i$
- After: $\sigma = \{x_1, \dots, x_i = v_i, \dots, x_n\}$

In case the program binds x_i to another dataflow variable (*bind(*x_i*,* x_w*)*), x_i become equivalent to x_w. Thus, x_i will be bound to the same term than x_w when x_w becomes bound (in case it was not bound when $bind(x_i, x_w)$ was issued). Binding x_i with the same value for several times introduces no side effect, i.e. it is idempotent. On the other hand, if x_i was already bound to the term v_w and v_i do not match v_w, the execution of the deterministic dataflow program terminates due to a programming error.

read(x_i*)* returns the term bound to x_i. More precisely, this operation can be expressed as follows:

- Before: $\sigma = \{x_1, \dots, x_i, \dots, x_n\}$
- $v_i = read(x_i)$
  - if $x_i.value == (x_m \vee \varnothing)$
    - $x_i.waiting\_processes \cup \{self()\}$
    - wait until x_i is bound
  - $v_i = x_i.value$
- After: $\sigma = \{x_1, \dots, x_i = v_i, \dots, x_n\}$

Finally, Derflow uses the Erlang *spawn* primitive to add concurrency to the deterministic dataflow model, a fundamental feature of the deterministic dataflow model. Furthermore, useful properties such as transparent concurrency are added. Section 5 shows why transparency concurrency is a desirable property and how programmer can use it.

3.2 Streams

Streams are a useful technique which allow threads, or processes, to communicate and synchronize in concurrent programming. A stream is represented here as a list of dataflow variables, with an unbound dataflow variable as the final element of the list. For instance, a stream variable can be expressed as the following:

$s_i = x_1 \mid \dots \mid x_{n-1} \mid x_n, x_n = \varnothing$

where $x_1, \dots, x_{n-1}$ are dataflow variables either bound or partially bound, and x_n is an unbound dataflow variable.

In order to add streams to Derflow, we extended the metadata kept by a dataflow variable with a new parameter called *next*. This new parameter stores the id of the dataflow variable that represents the successor element in the stream. Thus, a dataflow variable is now composed as follows:

$x_i = \{$*value, bound_variables, waiting_processes, next*$\}$

There are two basic operations applicable to a stream: *produce(x, v)* and *consume(x)*.

produce(x_n*,* v_n*)* extends the stream by binding the tail x_n to v_n and creating a new tail x_{n+1}. It returns the new tail. More precisely, this operation can be expressed as follows:

- Before: $\sigma = \{x_1, \dots, x_n = \varnothing\}$
- $x_{n+1} = produce(x_n, v_n)$
  - $bind(x_n, v_n)$
  - $x_{n+1} = declare()$
  - $x_n.next = x_{n+1}$
- After: $\sigma = \{x_1, \dots, x_n = v_n, x_{n+1} = \varnothing\}$

consume(x_i*)* reads the element of the stream represented by x_i. It returns the read value (v_i) and the identifier of the next element in the stream (x_{i+1}). More precisely, this operation can be expressed as follows:

- Before: $\sigma = \{x_1, \dots, x_i = v_i \vee x_m \vee \varnothing, x_{i+1}, \dots, x_n\}$
- $\{v_i, x_{i+1}\} = consume(x_i)$
  - $v_i = read(x_i)$
  - $x_{i+1} = x_i.next$
- After: $\sigma = \{x_1, \dots, x_i = v_i, x_{i+1}, \dots, x_n\}$

Different processes can read from the stream simultaneously. This do not compromise determinism. Nevertheless, the number of producers is restricted to one in order to keep determinism.

3.3 Laziness

Lazy, non-strict evaluation, or demand-driven execution, delays the evaluation of an expression until the value is needed somewhere else in the program. Lazy execution can improve the performance of programs by avoiding unnecessary computation. Lazy execution also enables the possibility of creating potentially infinite data structures, e.g. infinite lists and infinite trees, since each element will only be created when it is needed by the program.

The intuition of lazy evaluation is simple: a process that wants to assign a lazy variable to a value will be suspended until the value is needed by other process.

The only primitive we need to add is *wait_needed(x)*. This operation suspends the caller process until the dataflow variable

x is needed. As a consequence of this new primitive, the metadata kept by the dataflow variable has to be extended once more. A new parameter called *lazy* is added to the metadata. *lazy* is the set of the processes that called *wait_needed(x)* for the variable x. The dataflow variable is now composed as follows:

$x_i = \{$*value, bound_variables, waiting_processes, next, lazy*$\}$

More precisely, the *wait_needed(x)* primitive can be expressed as follows:

- Before: $\sigma = \{x_1, \ldots, x_i = \varnothing, \ldots, x_n\}$
- $wait\_needed(x_i)$
  - if $x_i.waiting\_processes == \emptyset$
    - $x_i.lazy \cup \{self()\}$
    - wait until a $read(x_i)$ is issued
- After: $\sigma = \{x_1, \ldots, x_i, \ldots, x_n\}$

In case x_i was already bound, *wait_needed(x)* returns immediately.

Furthermore, the primitive *read(x)* has to be changed to notify the processes that called *wait_needed(x)* . More precisely, the new *read(x)* primitive can be expressed as follows:

- Before: $\sigma = \{x_1, \ldots, x_i, \ldots, x_n\}$
- $v_i = read(x_i)$
  - $\forall p \in x_i.lazy,$ notify p
  - if $x_i.value == (x_m \vee \varnothing)$
    - $x_i.waiting\_processes \cup \{self()\}$
    - wait until x_i is bound
  - $v_i = x_i.value$
- After: $\sigma = \{x_1, \ldots, x_i = v_i, \ldots, x_n\}$

3.4 Failure handling

Failures introduce non-determinism. Therefore, a deterministic program can easily become non-deterministic if care is not taken to handle failures in a deterministic manner.

One simple approach to ensure determinism in the presence of failures is to force processes to wait forever if a dataflow variable is either unbound or not reachable. Obviously, this approach does not ensure progress. Consider the following example:

- Process p_0 is supposed to bind a dataflow variable, however fails before completing its task.
- Processes $p_1 \ldots p_n$ are waiting on p_0 to bind.
- Processes $p_1 \ldots p_n$ wait forever, resulting in non-termination.

However, determinism and dataflow variables provide a very useful property for failure handling: redundant computation will not affect the correctness of a deterministic dataflow program. We propose a failure handling model where failed processes or temporarily unreachable processes, can be restarted while still providing the guarantees of the deterministic programming model.

We classify the failures into two groups:

- **Computing process failure:**
  Failure of an individual Erlang process which uses a value in the single-assignment store. Given other processes may be waiting for the result of this processes computation, this can cause the program to block forever.
- **Dataflow variable failure:**
  A dataflow variable stored in the single-assignment store is not reachable. This means that computing processes issuing operations on the unreachable variable will block until the dataflow variable becomes accesible again. This may never happen and the computing process would block forever.

3.4.1 Computing process failure handling

Computing process failures are rather straightforward to handle; execution can continue by re-executing the failing process without having to worry about duplicate processing introducing non-determinism.

Consider the following example:

- Process p_0 reads a dataflow variable, x_1.
- Process p_0 performs a computation based on the value of x_1, and binds the result of computation to x_2.

Two possible failure conditions can occur:

- If the output variable never binds, process p_0 can be restarted and will allow the program to continue executing deterministically.
- If the output variable binds, restarting process p_0 has no effect, given the single-assignment nature of variables.

Derflow does not provide any primitive for handling this computation, as the Erlang primitives are sufficient to handle these failures. Section 5.4 provides an example on how to successfully handle computing process failures.

3.4.2 Dataflow variable failure handling

Dataflow variable failures are more difficult to handle, given that re-execution of a blocked or failed process does not guarantee progress.

Consider the following example:

- Process p_0 attempts to compute value for dataflow variable x_1 and fails.
- Process p_1 blocks on x_1 to be bound by p_0, which will not complete successfully.

The re-execution of blocked process p_1 will result in the process immediately blocking again. Therefore we must provide a way to identify dependencies between processes and dataflow variables in order to provide a deterministic restart strategy which guarantees progress. A common strategy to ensure progress in this situation is to restart the process that declared the failed dataflow variable. In addition, all the processes depending on the restarted process should also be restarted.

We can use the Erlang primitives *monitor/2* and *link/1* to build custom supervision trees which will guarantee a proper restart strategy which will ensure progress. Nevertheless, we still need to provide a way of monitoring and killing dataflow variables of the single-assignment store. To facilitate this, we extend our model with two additional primitives: *monitor(x)* and *kill(x)*. These primitives are inspired by the failure model of Collet [4].

To support these two primitives, we extend dataflow variables as follows:

- We extend dataflow variables allowing them to bind to a non-usable value, represented by $\top$. A *read* or *bind* operation on a non-usable dataflow variable blocks the caller process forever.
- We extend dataflow variables allowing them to track processes which have placed monitors on them. These monitors are tracked to support the $kill$ primitive.

Below is the updated definition of dataflow variables:

$x_i = \{$*value, bound_variables, waiting_processes, next, lazy, monitors*$\}$

The call *monitor(x_i)* sets a monitor to the dataflow variable x_i and returns a stream (initially, an unbound dataflow variable y) that will contain the reachability states that the dataflow variable x_i visits on the node that did the *monitor* call. The new metadata *monitors* is a set that contains all the identifiers of the processes monitoring the dataflow variable.

If the reachability state of x_i changes on a node, the new state is inserted at the end of each monitor stream that was created on that node. A dataflow variable can visit three reachability states: *perm_fail*, *temp_fail* and *normal*. *perm_fail* means that the dataflow variable is permanently unreachable. *temp_fail* means that the dataflow variable is temporarily unreachable but it may become reachable again. Finally, *normal* means that the dataflow variable is reachable. A dataflow variable can only visit the reachability state *normal* after visiting *temp_fail*. Figure 2 diagrams the reachability states that a dataflow variable can visit.

More precisely, the execution of *monitor(x_i)* can be defined as follows:

- Before: $\sigma = \{x_1, \ldots, x_i, \ldots, x_n\}$
- $y = monitor(x_i)$
  - $x_i.monitors \cup \{self()\}$
  - $y = declare()$
- After: $\sigma = \{x_1, \ldots, x_i, \ldots, x_n, y\}$

kill(x_i) sets the dataflow variable x_i to non-usable. It is a synchronous operation; therefore, the caller will block until the operation is completed. All processes monitoring a killed dataflow variable must be notified. This implies that if there are reachability problems the operation may never return. More precisely, the execution of this operation can be defined as follows:

- Before: $\sigma = \{x_1, \ldots, x_i, \ldots, x_n\}$
- $kill(x_i)$
  - $x_i.value = \top$
  - $\forall p \in x_i.monitors,\quad \text{notify } p$
- After: $\sigma = \{x_1, \ldots, x_i = \top, \ldots, x_n\}$

4. Implementation

The following section discusses the implementation of Derflow.

4.1 Derflow API

Derflow currently provides the following functions:

Deterministic dataflow

- {*ok, Id::term()*} = *declare()*:
  Creates a new unbound dataflow variable in the store. It returns the id of the newly created variable.
- *ok = bind(Id, Value)*:
  Binds the dataflow variable *Id* to *Value*. *Value* can either be an Erlang term or any other dataflow variable.
- *ok = bind(Id, Mod, Fun, Args)*:
  Binds the dataflow variable *Id* to the result of evaluating *Mod:Fun(Args)*.
- {*ok, Value::term()*} = *read(Id)*:
  Returns the value bound to the dataflow variable *Id*. If the variable represented by *Id* is not bound, the caller blocks until it is bound.

Streams

- {*ok, NextId::term()*} = *produce(Id, Value)*:
  Binds the variable *Id* to *Value*. It returns the pair composed by the atom *ok* and the variable *NextId* that represents the id of the next element of the stream.
- {*ok, NextId::term()*} = *produce(Id, Mod, Fun, Args)*:
  Binds the variable *Id* to the result of evaluating *Mod:Fun(Args)*. It returns the pair composed by the atom *ok* and the variable *NextId* that represents the id of the next element of the stream.
- {*ok, Value::term(), NextId::term()*} = *consume(Id)*:
  Returns the value bound to the dataflow variable *Id* and the id of the next element in the stream. If the variable represented by *Id* is not bound, the caller blocks until it is bound.
- {*ok, NextId::term()*} = *extend(Id)*:
  Declares the variable that follows the variable *Id* in the stream. It returns the id of the next element of the stream. This function is useful for achieving concurrency in some cases (e.g. The Sieve of Eratosthenes).

Laziness

- *ok = wait_needed(Id)*:
  Used for adding laziness to the execution. The caller blocks until the variable represented by *Id* is needed when attempting to *read* the value.

Dataflow variable failure handling

- {*ok, IdStream::term()*} = *monitor(Id)*:
  Registers the caller as monitor of the dataflow variable *Id*. Returns the head of a stream that will contain the states that the dataflow variable *Id* visits, from the caller process view.
- *ok = kill(Id)*:
  Set the dataflow variable represented by *Id* to non-usable.

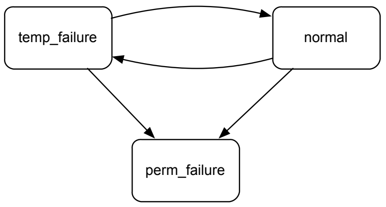

Figure 2. Dataflow variable reachability state diagram.

4.2 Distribution

Derflow is implemented as an Erlang library, which relies on a single-assignment store. This store needs to be accessible by all the processes that participate in the execution of the Derflow program.

4.2.1 Partition strategies

In a single system, the design of such a store is simpler as the memory is accessible and shared by all the communicating processes. Nevertheless, in a distributed fashion, the implementation becomes tricky and keeping consistency guarantees and high grade of scalability is challenging.

We considered three approaches:

- Each dataflow variable has a 'home process', where it was initially created. Therefore, binding the variable always sends a message to the 'home process', which then broadcasts the binding to all the instances.
- Each instance of a dataflow variable knows all the other instances. There are no 'home processes'. Therefore, after binding the local instance, the operation is directly broadcast to the other instances.

- Each computing node has a partition of the single-assignment store. All processes on a given computing node will reference the local partition. Binding a variable sends the operation to the local partition, which will then send it to the partition replicas.

We chose the third approach. In the first two approaches, every process that knows about a particular dataflow variable creates a new instance; therefore, it will eventually participate in the corresponding bind operation. In some cases, the number of instances can be large. This would result in poor performance. Nevertheless, in the third approach, each computing node is responsible for a partition of the single-assignment store; therefore no matter how many processes know about a particular dataflow variable, the binding operation would always be sent to the responsible and to the corresponding replicas.

4.2.2 Design considerations

When choosing to implement our distributed single-assignment store, we examined two possible choices: $riak\_core$ and $mnesia$ [8].

$mnesia$ provides a native Erlang implementation of a relational database management system, which supports atomic transactions and the ability to distribute tables across nodes through replication. However, we look at two specific problems with $mnesia$:

- Problems arise in the presence of network partitions [11] where the $mnesia$ nodes on either side of the network partition are able to make progress independently. Currently, no mechanisms exist for reconciling the changes made to the database when nodes reconnect, nor reasoning about concurrent or causally influenced operations. While the functionality for reasoning about concurrent events is not necessary for the implementation of the single-assignment store, Section 7 discusses a generalization of our single-assignment variables to conflict-free replicated data types, or CRDTs [17], where causality is desired.
- $mnesia$ performs replication to all nodes which share a table of data. This requires writing a custom distribution layer for distributing the data if we want to have it partitioned to ensure even load distribution given dynamic membership and node failures.

Given the background discussed in Section 2.1, $riak\_core$ provides solutions to both of these problems:

- $riak\_core$ provides a dotted version vector [16] and vector clock facility as a causality tracking mechanism which can be used to reason about concurrent operations. In addition, $riak\_core$ provides mechanisms, such as active anti-entropy and handoff, which allow us to reason about divergences between replicas.
- $riak\_core$'s distribution layer provides minimal reshuffling of data, and predictable hashing through hash-space partitioning, consistent hashing, and a virtual node abstraction.

4.2.3 Implementation on $riak\_core$

In implementing the partitioned single-assignment store on $riak\_core$, we made the following design decisions:

- Data is partitioned across a series of nodes, using the hash-space partitioning and consistent hashing techniques described in Section 2.1.1 and Section 2.1.2.
- When declaring new dataflow variables, we write the variable into the replica set for that variable, requiring that the write be acknowledged by a strict quorum to ensure fault-tolerance of the variable as described in Section 2.1.3.
- As dataflow variables become bound, we rely again on a strict quorum to acknowledge the write, and notify all processes waiting for the value that the variable has been bound. Given that $n/2 - 1$ nodes might not accept the write or be available, we ensure that an active anit-entropy mechanism exists to notify any processes on the node which did not receive the update which might be waiting when the bound value is replicated.
- If a strict quorum is not available because of a network partition, operations on dataflow variables do not make progress until the partition has healed.

In the event of ownership transfer, during dynamic membership changes within the cluster, we perform the following:

- Each replica's portion of single-assignment store is transferred over to the target replica. As this occurs, each dataflow variable, if bound, notifies all waiting processes on the target replica allowing any processes which were waiting during the partition to proceed.
- As each variable is transferred over, monitors are removed locally and reapplied for each dataflow variable on the target vnode, given the processes which are waiting.
- Given that the process notification of a bound variable operation is idempotent, duplicate notifications to the same process produces no result.

5. Examples

In this section we describe some use cases for Derflow.

5.1 Concurrency transparency

In Derflow, any function that uses dataflow variables can be run in a different process while keeping the final result same. Thus, programmers can transparently add concurrency to their programs (either parallelism or distribution) in a secure way without thinking about data races and possible bugs.

One such example is a map function, that receives a stream of inputs and applies a function to each element resulting an output stream of equal length. The code in Derflow for a sequential map function is the following:

```
map(S1, M, F, S2) ->
  case derflow:consume(S1) of
    {ok, nil, _} ->
      derflow:bind(S2, nil);
    {ok, Value, Next} ->
      {ok, NextOut} = derflow:produce(S2, M, F, Value),
      map(Next, F, NextOut)
  end.
```

Nevertheless, due to the concurrency transparency property, the programmer could easily upgrade his sequential map to a concurrent implementation without compromising determinism. The code in Derflow for the concurrent implementation of the map function is the following:

```
concurrent_map(S1, M, F, S2) ->
  case derflow:consume(S1) of
    {ok, nil, _} ->
      derflow:bind(S2, nil);
    {ok, Value, Next} ->
      {ok, NextOut} = derflow:extend(S2),
      spawn(derflow, bind, [S2, M, F, Value]),
      concurrent_map(Next, F, NextOut)
  end.
```

In this case, the programmer explicitly specified (by using the primitive *spawn(module, function, args)*) that the evaluation of the function *F* is done asynchronously. Therefore, the map function can read the next element from the input stream without waiting for the function to be evaluated. The concurrent map, when leveraging parallel execution, will be faster than its sequential counterpart.

5.2 Concurrent deployment

In concurrent deployment, we could further leverage concurrency transparency to concurrently and incrementally start new processes according to need. There is no need to start all processes when initializing programs, instead only a few processes will be started at first and they will launch new processes during runtime according to need. The launched processes are executed concurrently and will terminate when it finishes its computation, without affecting the execution of other processes.

The following example is a pipeline that implements the Sieve of Eratosthenes. This program receives a stream of integers and returns a stream with the integers that are prime. At each iteration of the sieve, the stream of candidates is filtered by using the latest prime found. Thus, one filter process is created per iteration. The output of a filter is used as an input of the next filter. Filters are pipelined; therefore, as soon as a filter outputs the first element of its output stream, the next filter can start its execution. The code in Erlang using Derflow is the following:

```
sieve(S1, S2) ->
  case derflow:consume(S1) of
    {ok, nil, _} ->
      derflow:bind(S2, nil);
    {ok, Value, Next} ->
      {ok, SN} = derflow:declare(),
      F = fun(Y) -> Y rem Value =/= 0 end,
      spawn(sieve, filter, [Next, F, SN]),
      {ok, NextOut} = derflow:produce(S2, Value),
      sieve(SN, NextOut)
  end.

filter(S1, F, S2) ->
  case derflow:consume(S1) of
    {ok, nil, _} ->
      derflow:bind(S2, nil);
    {ok, Value, Next} ->
      case F(Value) of
        false ->
          filter(Next, F, S2);
        true->
          {ok, NextOut} = derflow:produce(S2, Value),
          filter(Next, F, NextOut)
      end
    end.
```

5.3 Laziness

The following examples show how the *wait_needed* primitive can be used to implement lazy functions.

The first example implements a lazy version of a sorting algorithm that sorts a list of numbers in ascending order. The Derflow implementation is the following:

```
insort(List, S) ->
  case List of
    [H|T] ->
      {ok, OutS} = derflow:declare(),
      insort(T, OutS),
      spawn(getmin, insert, [H, OutS, S]);
    [] ->
      derflow:bind(S, nil)
  end.

insert(X, In, Out) ->
  ok = derflow:wait_needed(Out);
  case derflow:consume(In) of
    {ok, nil, _} ->
      {ok, Next} = derflow:produce(Out, X),
      derflow:bind(Next, nil);
    {ok, V, SNext} ->
      if X < V ->
        {ok, Next} = derflow:produce(Out, X),
        derflow:produce(Next, In);
      true ->
        {ok, Next} = derflow:produce(Out,V),
        insert(X, SNext, Next)
      end
    end.
```

The primitives that contributes to the laziness of this program are *spawn* on the fourth line of insort and the *wait_needed* function call in the first line of the *insert* function. The *spawn* operation creates a process when an insertion should be executed. The *wait_needed* causes the created process to suspend until the result is needed by some other process. When only partial results are needed for the sorting algorithm, the lazy implementation can have a performance gain over the eager version.

For instance, if only the smallest number of the sorted list is needed, we can simply read the first element of the output list. When the input list is [1,2,3,4,5,6,7,8,9,10], both eager execution and lazy execution performs insertion ten times. However, when the input is [10,9,8,7,6,5,4,3,2,1], the eager version executes insertion for 54 times; in contrast, the lazy version only executes insertion 19 times.

The second example combines lazy execution and eager execution. We implemented a bounded-buffer that connects a producer and a consumer. Thus, the producer only produces on demand when the consumer needs to consume. Nevertheless, the producer is allowed to generate some elements in advance in order to be more efficient. The Derflow implementation is the following:

```
producer(Value, N, Output) ->
  if (N > 0) ->
    ok = derflow:wait_needed(Output),
    {ok, Next} = derflow:produce(Output, Value),
    producer(Value+1, N-1, Next);
  true ->
    derflow:bind(Output, nil)
  end.

loop(S1, S2, End) ->
  ok = derflow:wait_needed(S2),
  {ok, S1Value, S1Next} = derflow:consume(S1),
  {ok, S2Next} = derflow:produce(S2, S1Value),
  case derflow:extend(End) of
    {ok, nil} ->
      ok;
    {ok, EndNext} ->
      loop(S1Next, S2Next, EndNext)
    end.

buffer(S1, BUFFER_SIZE, S2) ->
  End = drop_list(S1, BUFFER_SIZE),
  loop(S1, S2, End).

drop_list(S, Size) ->
  if Size == 0 ->
    S;
  true ->
    {ok, Next} = derflow:extend(S),
    drop_list(Next, Size-1)
  end.


consumer(S2, Size, F, Output) ->
  if Size == 0 ->
```

```
      ok;
    true ->
      case derflow:consume(S2) of
        {ok, nil, _} ->
          derflow:bind(Output, nil);
        {ok, Value, Next} ->
          {ok, NextOut} = derflow:produce(Output, F(Value)),
          consumer(Next, Size-1, F, NextOut)
      end
    end.
```

The above code has three main components:

- The *producer* that only produces items when it is needed. This is achieved by calling *wait_needed* for the next element after it has produced an item.
- The *bounded buffer*: It takes the output stream of the *producer* and the input stream of the *consumer*. It firstly asks for a number of items (*BUFFER_SIZE*) to the *producer* by extending the *producer*'s stream (*drop_list*), then it keeps checking if the *consumer* asks for items. In case the *consumer* has asked, the *bounded buffer* copies an element from the *producer*'s stream to the *consumer*'s stream and extend the *producer*'s stream by one more element.
- The *consumer* that asks for items eagerly.

5.4 MapReduce-style example

We implement a simple framework that can concurrently launch tasks from multiple clients, similar to MapReduce [5]. It combines the use of dataflow variables, concurrency transparency, concurrent deployment, and non-determinism.

In the example, clients send a MapReduce-style task to a proxy through *send_task*. The proxy appends received tasks to a stream and keeps waiting for tasks. The job tracker checks the task stream, spawns mappers and reducers concurrently for incoming tasks and continues checking for tasks.

```
send_task(Proxy, Map, Reduce, Input, Output) ->
  Proxy ! {Map, Reduce, Input, Output}.

jobproxy(TaskStream) ->
  receive
    Task ->
      {ok, Next} = derflow:produce(TaskStream, Task),
      jobproxy(Next)
  end.

jobtracker(Superv, Tasks) ->
  case derflow:consume(Tasks) of
    {ok, nil, _} ->
      io:format("All job finished!~n");
    {ok, Value, Next} ->
      {MapTask, ReduceTask, In, Out} = Value,
      {Mod, MapFun} = MapTask,
      {Mod2, RedFun} = ReduceTask,
      MapOut = spawn_map(Superv, In, Mod, MapFun, []),
      spawn_mon(Superv, Mod2, RedFun, [MapOut, Out]),
      jobtracker(Next)
    end.

spawn_map(Superv, Inputs, Mod, Fun, Outputs) ->
  case Inputs of
   [H|T] ->
     {ok, S} = derflow:declare(),
     spawn_mon(Superv, Mod, Fun, [H, S]),
     spawnmap(T, Mod, Fun, lists:append(Outputs,[S]));
   [] ->
     Outputs
    end.
```

```
spawn_mon(Superv, Mod, Fun, Args) ->
  Pid = spawn(Module, Function, Args),
  Superv ! {'SUPERVISE', Pid, Mod, Fun, Args}.
```

The implementation of the proxy embodies non-determinism, as tasks may be received in different orders due to the process scheduler or network congestion.

However, since the proxy can not predict the arriving order of tasks, it is impossible to write the program in a deterministic way. In fact, this level of non-determinism only affects the order that tasks are launched. Since each task is executed in parallel without interaction between each other, users can not perceive non-determinism.

The job tracker also exemplifies several concepts we proposed. Firstly, the job tracker starts a job when it receives a new task incrementally and does not need to wait for all tasks before it starts any, which is concurrent deployment. Secondly, in each job, mappers and reducers are launched concurrently. This exploits the concurrency transparency property. Each mapper has its own output stream. The reducer reads from the mappers output streams sequentially. Thus, it uses the dataflow variables to synchronize the concurrent execution.

In addition, the example handles computing processes failures. The first argument (*Superv*), of the *jobtracker* function, is the process id of a supervisor process. Thus, all new dataflow processes created in jobtracker (using the function *spawn_mon*) are supervised by it.

According to the semantics of Derflow, redundant computation does not affect the correctness of the program. Therefore, deterministic dataflow functions are idempotent. Considering this property, we implemented a simple supervisor that restarts the failing deterministic dataflow processes when a problem is detected. The code is the following:

```
supervisor(Dict) ->
  receive
    {'DOWN', Ref, process, _, _} ->
      case dict:find(Ref, Dict) of
        {ok, {Module, Function, Args}} ->
          spawn_mon(self(), Module, Function, Args);
        error ->
          supervisor(Dict)
      end;
    {'SUPERVISE', PID, Information} ->
      Ref = erlang:monitor(process, PID),
      Dict2 = dict:store(Ref, Information, Dict),
      supervisor(Dict2)
  end.
```

The above supervisor receives *supervise* and *down* messages. The former is a monitoring request; therefore, the supervisor simply uses the Erlang *monitor* primitive to set the monitor. The latter is received when a monitored process does not exist, it is not reachable or it has died. The supervisor behaves the same in all situations by re-executing the deterministic dataflow process. The supervisor uses *dict* to store the information regarding the monitored processes such as the function executed by the process and its arguments.

Nevertheless, the shown supervisor is only one example. More sophisticated supervisors can be implemented. For instance, the supervisor could behave differently for temporary failures. Then, it can decide to wait longer before restarting the computation. In some cases, it is not efficient to restart the execution.

6. Integration with non-determinism

Deterministic dataflow is a powerful concurrent programming model that eliminates all race conditions by design. However, it is clear that practical applications sometimes need non-determinism. In most cases, the non-determinism is only needed in a small part of the program. But the need cannot be reduced to zero. For example, a simple client-server application needs non-determinism since the server must accept requests from any client. There is only one point of non-deterministic choice, at the server, but it cannot be eliminated. So our deterministic model must cohabit in a simple way with non-deterministic execution. In this section, we show to integrate our model with non-deterministic execution.

6.1 *is_det* primitive

Derflow provides one primitive which allows us to support non-deterministic execution: *is_det(x)*. This operation checks whether a dataflow variable (x) is bound or not, which introduces non-determinism due to different process scheduling or network delays in each program execution.

is_det(x) primitive is useful for stream management. For instance, in a producer-consumer application, where the producer is faster than the consumer, the latter might be interested in only consuming the latest element produced until that point. Thus, it would like to skip some of the produced elements.

More precisely, *is_det* can be described as follows:

- Before: $\sigma = \{x_1, \ldots, x_i, \ldots, x_n\}$
- $bool = is\_det(x_i)$
    - $bool = x_i.value == v_i$
- After: $\sigma = \{x_1, \ldots, x_i, \ldots, x_n\}$

Accordingly, the Derflow API is extended as follows:

- $\{$*ok, Value::boolean()*$\}$ = *is_det(Id)*:
  Returns true if the dataflow variable Id is bound, false otherwise.

A good example of the use of *is_det(x)* is a live-streaming video displayer. The displayer always tries to display the latest frame sent and skip the intermediate ones. A simplified version of this program can be written in Derflow as follows:

```
skip(Input, Output) ->
  case derflow:consume(Input) of
    {ok, nil, _} ->
      derflow:bind(Output, nil);
    {ok, _, Next} ->
      {ok, Bound} = derflow:is_det(Next),
      if
        Bound ->
          skip(Next, Output);
        true ->
          derflow:produce(Output, {ok, Input})
      end
    end.

display(Input) ->
  {ok, Output} = derflow:declare(),
  skip(Input, Output),
  case derflow:consume(Output) of
    {ok, Value, Next} ->
      display_frame(Value),
      display(Next)
    end.
```

The *skip* function traverses the input stream and returns the latest frame until that point. The *display* function displays the frame returned by *skip*.

6.2 Integration with Erlang

One of the main limitations of the deterministic dataflow model is that only one process can write into a stream; therefore, a simple client-server application cannot be implemented. By using communication channels, this limitation can be overcome.

The following example shows how to do this by taking advantage of the message-passing primitives of Erlang. The example implements a monitoring system. It is composed of a centralized component that receives messages from multiple sensor entities placed elsewhere. In this example, we monitor the number of failures per datacenter in a geo-replicated application. There is one sensor per datacenter that sends a failure message to the central component (through a *proxy*) each time a computer is down. The centralized component registers the failures to eventually analyze the statistics. The *proxy* is the component that uses the Erlang communication channels. It receives spontaneous messages from the sensors and serializes them by appending them to an associated stream.

```
observer_proxy(S) ->
  receive
    {Msg, From} ->
      {ok, Next} = derflow:produce(S, {Msg, From}),
      observer_proxy(Next)
  end.

sensor(Proxy, Identifier) ->
  Random = random:uniform(),
  Milliseconds = round(timer:seconds(Random)),
  timer:sleep(Milliseconds),
  Proxy ! {computer_down, Identifier},
  sensor(Proxy, Identifier).

dcs_observer(Input, Output, State) ->
  case derflow:consume(Input) of
    {ok, {computer_down, Identifier}, NextInput} ->
      State2 = register(Identifier, State),
      {ok, NextOut} = derflow:produce(Output, State2),
      dcs_observer(NextInput, NextOut, State2);
    {ok, _, NextInput} ->
      % Ignore
      dcs_observer(NextInput, Output, State)
    end.
  end.
```

The above application is mainly composed by three functions:

- *observer_proxy* that continuously waits for messages. If a message is received, it immediately appends it to the associated stream. It intentionally waits forever if no messages are sent.
- *sensor* that sends a message to the *observer_proxy* every time a computer fails. The computer failure is modeled by a random wait.
- *dcs_observer* that registers the failures by reading the stream associated to the *observer_proxy*.

7. Conclusions and future work

In this paper, we have proposed Derflow, a deterministic dataflow extension for Erlang. Derflow relies on a robust, highly available and scalable single-assignment store built using *riak_core*, a distributed systems framework. We have shown examples of its usage and explained how it can be integrated with non-deterministic computations.

The following paragraphs outline a series of planned extensions to Derflow that will provide a more expressive and complete computational model for large-scale distributed applications.

Generalizing to semilattices Given that our dataflow variables can be seen as simple semilattices with two states: bound and unbound, we would like to extend them to more expressive semilattices used to build CRDTs. This is very similar to the approach taken by LVars [14] to provide deterministic parallel programming. Our work expands on this work by providing this deterministic parallelism across computing nodes, in a fault-tolerant manner.

Similarly to LVars, we would also like to provide a threshold read primitive over these datatypes, which would cause an application to block and synchronize on a value until a particular threshold is passed. However, we are still uncertain what difficulties arise when introducing distribution into this model, given the various failure conditions that can be experienced over computer networks. Furthermore, some CRDTs composed by multiple semi-lattices do not behave monotonically. This may restrict the use of threshold reads.

Extending the Erlang syntax and runtime system Our current model is implemented with a set of library functions. Compiler and run-time modifications can be done to provide a simple syntax for deterministic dataflow programs and to provide simpler ways to control non-determinism in programs. These extensions would provide a much more compelling computational model for the user.

Acknowledgments

We thank Nicholas Rutherford and Sean Cribbs for comments that helped to improve the paper. This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n^o 609551 and by the Erasmus Mundus Joint Doctorate Programme under Grant Agreement 2012-0030.

References

[1] Akka: Building powerful concurrent and distributed applications more easily, 2014. URL `http://akka.io/`.

[2] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. . URL `http://doi.acm.org/10.1145/2535838.2535845`.

[3] Basho Technologies Inc. Riak core source code repository. `http://github.com/basho/riak_core`.

[4] R. Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, Dec. 2007.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. . URL `http://doi.acm.org/10.1145/1294261.1294281`.

[7] S. Doeraene and P. Van Roy. A new concurrency model for Scala based on a declarative dataflow core. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 4:1–4:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1. . URL `http://doi.acm.org/10.1145/2489837.2489841`.

[8] Ericsson AB. mnesia - a distributed telecommunications dbms. `http://www.erlang.org/doc/man/mnesia.html`.

[9] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

[10] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/355620.361161`.

[11] Joel Reymont. [erlang-questions] is there an elephant in the room? mnesia network partition. `http://erlang.org/pipermail/erlang-questions/2008-November/039537.html`.

[12] G. Kahn. The semantics of a simple language for parallel programming. In *In Information Processing'74: Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[13] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *Proc. of the IFIP Congress*, volume 77, pages 994–998, 1977.

[14] L. Kuper and R. R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2381-9. . URL `http://doi.acm.org/10.1145/2502323.2502326`.

[15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. . URL `http://doi.acm.org/10.1145/1346281.1346323`.

[16] N. M. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *CoRR*, abs/1011.5808, 2010.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24549-7. . URL `http://dx.doi.org/10.1007/978-3-642-24550-3_29`.

[18] H. Svensson and L.-A. Fredlund. Programming distributed erlang applications: Pitfalls and recipes. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*, ERLANG '07, pages 37–42, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-675-2. . URL `http://doi.acm.org/10.1145/1292520.1292527`.

[19] P. Van Roy and S. Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[20] D. Wyatt. *Akka concurrency: Building reliable software in a multi-core world*. Artima, 2013.