

Distributed deterministic dataflow programming for Erlang

Manuel Bravo ¹ Zhongmiao Li ¹ Peter Van Roy ¹
Christopher Meiklejohn ²

¹Université catholique de Louvain

²Basho Technologies, Inc.

Erlang User Conference
Stockholm, Sweden, 2014
June 9, 2014

Overview

- 1 Introduction
- 2 Background
- 3 Semantics
- 4 Implementation
- 5 Examples
- 6 Caveats and future work
- 7 References

- Funded by the European Union
- Focusing on Conflict-free Replicated Data Types (CRDTs)
- Basho, Rovio, Trifork
- INRIA, Universidade Nova de Lisboa, Université Catholique de Louvain, Koç Üniversitesi, Technische Universität Kaiserslautern



- Build a programming model for conflict-free replicated data types (CRDTs). [12]
- Deterministic, distributed, parallel programming in Erlang.
- Similar work to LVars [10] and Bloom. [5]
- Key focus on distributed computation, high scalability, and fault-tolerance.

Conflict-free replicated data types

- Comes in two main flavors: state-based and operations-based.
- State-based CRDTs:
 - Data structure which ensures convergence under concurrent operations.
 - Based on bounded join-semilattices.
 - Data structure which grows state monotonically.
- Imagine a vector clock.

Motivation

- 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.
- Concurrent programs in non-deterministic languages, are notoriously hard to prove correct.

- Treat every message received by a process as a 'choice'.
- A series of these 'choices' define one execution of a program.
- Prove each execution is correct; or terminates.
- Further complicated by distributed Erlang and its semantics. [13]
- OTP is essentially "programming patterns" to reduce this burden.

Contributions

- An "alternative" approach to this non-determinism.
- Deterministic data flow programming model for Erlang, implemented as a library.
- Concurrent programs, which regardless of execution, produce the same result.
- Fault-tolerance and distribution of computations provided by *riak_core*. [3]

Deterministic dataflow programming

- Historically:
 - 1974: First proposed as Kahn networks. [7]
 - 1977: Lazy version of this same model was proposed by Kahn and David MacQueen [9].
- More recently:
 - CTM/CP: Oz [14]
 - Akka [1, 15]
 - Ozma [6]

Single-assignment store

- Relies on a single assignment store: $\sigma = \{x_1, \dots, x_n\}$
- Example: $\sigma = \{x_1 = x_2, x_2 = \emptyset, x_3 = 5, x_4 = [a, b, c], \dots, x_n = 9\}$
- Where:
 - $x_i = \emptyset$: 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.
 - $x_i = v_j$: Variable x_i is bound to a term (v_j).

- $x_i = \{value, waiting_processes, bound_variables\}$
- Where:
 - *value* : empty, or dataflow value.
 - *waiting_processes* : processes waiting for x_i to be bound.
 - *bound_variables* : dataflow variables which are partially bound.

Basic Primitives I

- *declare()* creates a new dataflow variable.
 - Before: $\sigma = \{x_1, \dots, x_n\}$
 - $x_{n+1} = \text{declare}()$
 - create a unique dataflow variable x_{n+1}
 - store x_{n+1} into σ
 - After: $\sigma = \{x_1, \dots, x_{n+1} = \emptyset\}$
- *bind(x_i, v_i)* binds the dataflow variable x_i to the value v_i .
 - Before: $\sigma = \{x_1, \dots, x_i = \emptyset, \dots, x_n\}$
 - *bind(x_i, v_i)*
 - $\forall p \in x_i.\text{waiting_proccesses}, \text{notify } p$
 - $\forall x \in x_i.\text{bound_variables}, \text{bind}(x, v_i)$
 - $x_i.\text{value} = v_i$
 - After: $\sigma = \{x_1, \dots, x_i = v_i, \dots, x_n\}$

Basic Primitives II

- $read(x_i)$ returns the term bound to x_i .
 - Before: $\sigma = \{x_1, \dots, x_i, \dots, x_n\}$
 - $v_i = read(x_i)$
 - if $x_i.value == (x_m \vee \emptyset)$
 $x_i.waiting_processes \cup \{self()\}$
wait
 - $v_i = x_i.value$
 - After: $\sigma = \{x_1, \dots, x_i = v_i, \dots, x_n\}$
- $thread(function, args)$ runs $function(args)$ in a different process.
 - Implemented using the Erlang *spawn* primitive.

Streams I

- Streams of dataflow variables: $s_i = x_1 \mid \dots \mid x_{n-1} \mid x_n, x_n = \emptyset$
- Extend metadata to store pointer to next position:
 $x_i = \{value, waiting_processes, bound_variables, next\}$
- $produce(x_n, v_n)$ extends the stream by binding the tail x_n to v_n and creating a new tail x_{n+1} .

Stream Primitives I

- $produce(x_n, v_n)$ extends the stream by binding the tail x_n to v_n and creating a new tail x_{n+1} .
 - Before: $\sigma = \{x_1, \dots, x_n = \emptyset\}$
 - $x_{n+1} = produce(x_n, v_n)$
 - $bind(x_n, v_n)$
 - $x_{n+1} = declare()$
 - $x_n.next = x_{n+1}$
 - After: $\sigma = \{x_1, \dots, x_n = v_n, x_{n+1} = \emptyset\}$

Stream Primitives II

- $consume(x_i)$ reads the element of the stream represented by x_i .
 - Before: $\sigma = \{x_1, \dots, x_i = v_i \vee x_m \vee \emptyset, x_{i+1}, \dots, x_n\}$
 - $\{v_i, x_{i+1}\} = consume(x_i)$
 - $v_i = read(x_i)$
 - $x_{i+1} = x_i.next$
 - After: $\sigma = \{x_1, \dots, x_i = v_i, x_{i+1}, \dots, x_n\}$

- Provide non-strict evaluation primitive.
- Extend metadata:
 $x_i = \{value, waiting_processes, bound_variables, next, lazy\}$
- $wait_needed(x)$ suspends until the caller until x is needed.
 - Before: $\sigma = \{x_1, \dots, x_i = \emptyset, \dots, x_n\}$
 - $wait_needed(x_i)$
 - if $x_i.waiting_processes == \emptyset$
 $x_i.lazy \cup self()$
wait until a $read(x_i)$ is issued
 - After: $\sigma = \{x_1, \dots, x_i, \dots, x_n\}$
- Modify read operation to notify, if lazy.

Non-determinism

- Provide a primitive which supports non-deterministic execution.
- Introduces non-determinism because it allows a choice to be taken on whether the variable is bound or not.
- $is_det(x)$ determines whether a variable is bound yet.
 - Before: $\sigma = \{x_1, \dots, x_i, \dots, x_n\}$
 - $bool = is_det(x_i)$
 - $bool = x_i.value == v_i$
 - After: $\sigma = \{x_1, \dots, x_i, \dots, x_n\}$

Failure handling

- Failures introduce non-determinism.
- One approach: wait forever until the variables are available.
- Does not ensure progress, for example:
 - Process p_0 is supposed to bind a dataflow variable, however fails before completing its task.
 - Processes $p_1 \dots p_n$ are waiting on p_0 to bind.
 - Processes $p_1 \dots p_n$ wait forever, resulting in non-termination.
- Two classes of errors:
 - Computing process failures.
 - Dataflow variable failure.

Computing process failures

- Consider the following:
 - 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.
- Handled via Erlang primitives.
 - Supervisor trees; restart the processes.

Dataflow variable failures

- Consider the following:
 - 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.
- Re-execution results in the same failure.
- Explore extending the model with a non-usable value.

Deterministic dataflow API

- $\{id, Id::term()\} = declare()$:
Creates a new unbound dataflow variable in the single-assignment store. It returns the id of the newly created variable.
- $\{id, NextId::term()\} = bind(Id, Value)$:
Binds the dataflow variable Id to $Value$. $Value$ can either be an Erlang term or any other dataflow variable.
- $\{id, NextId::term()\} = bind(Id, Mod, Fun, Args)$:
Binds the dataflow variable Id to the result of evaluating $Mod:Fun(Args)$.
- $Value::term() = read(Id)$:
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.

- $\{id, NextId::term()\} = produce(Id, Value)$:
Binds the variable Id to $Value$.
- $\{id, NextId::term()\} = produce(Id, Mod, Fun, Args)$:
Binds the variable Id to the result of evaluating $Mod:Fun(Args)$.
- $\{Value::term(), NextId::term()\} = consume(Id)$:
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.
- $\{id, NextId::term()\} = extend(Id)$:
Declares the variable that follows the variable Id in the stream. It returns the id of the next element of the stream.

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

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

Partition strategies

- Each variable has a home process, which coordinates notifying all processes which should be told of changes in binding.
- Each process knows information about all processes which should be notified.
- **Partitioning of the single assignment store, where processes communicate to the local process.**

Design considerations

- *mnesia*
 - Problems during network partitions. [8]
 - Allows independent progress with no way to reconcile changes.
 - Replication not scalable enough or provide fine-grained enough control.
- *riak_core*
 - Minimizes reshuffling of data through consistent hashing and hash-space partitioning.
 - Facilities for causality tracking. [11]
 - Anti-entropy and hinted handoff.
 - Dynamic membership.

- DHT with fixed partition size/count.
- Partitions claimed on membership change.
- Replication over ring-adjacent partitions. (preference lists)
- Sloppy quorums (fallback replicas) for added durability.

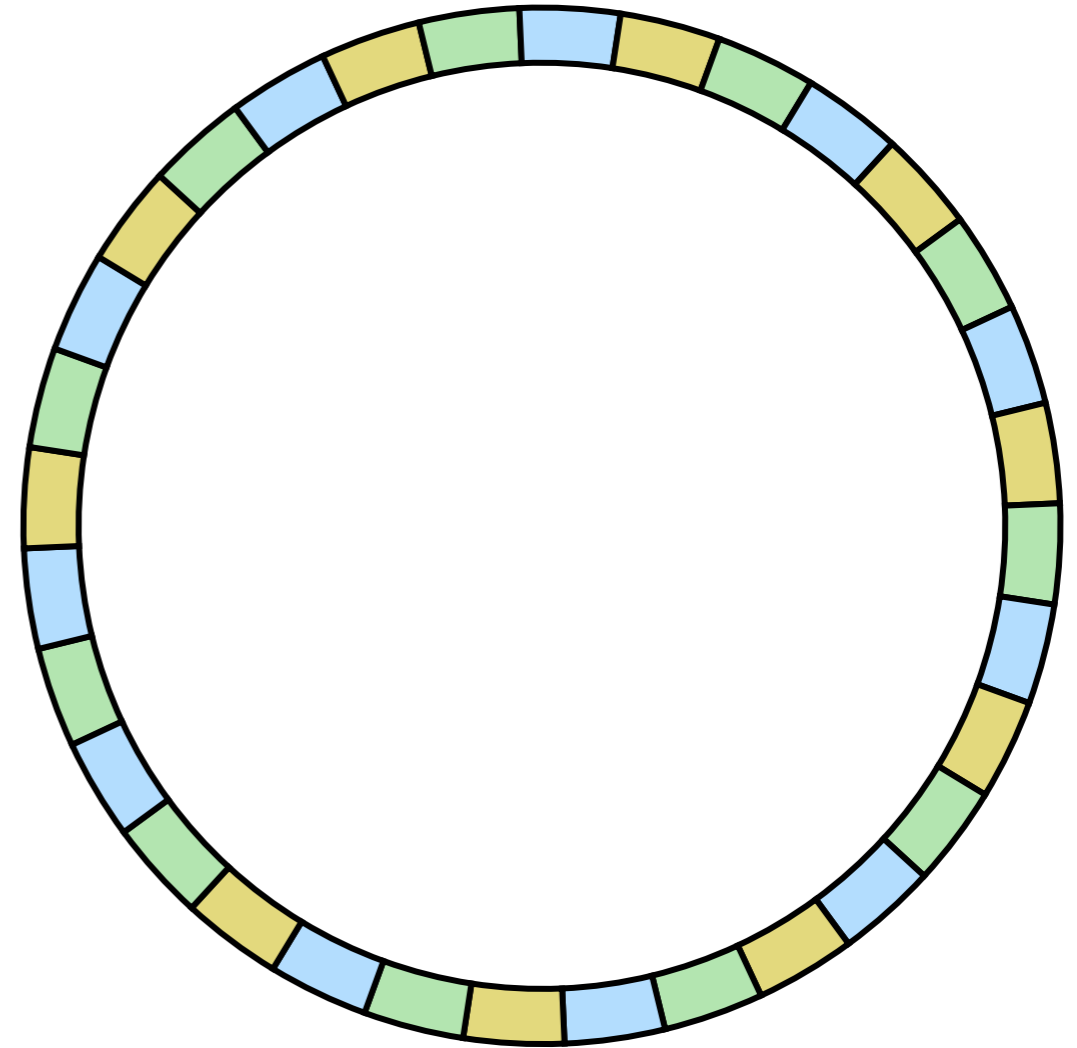


Figure : Ring with 32 partitions and 3 nodes

Implementation on *riak_core*

- Partition the single-assignment store across the cluster.
- Writes are performed against a strict quorum of the replica set.
- As variables become bound:
 - Notify all waiting processes using a strict quorum.
 - In the event of node failures, anti-entropy mechanism is used to update replicas which missed the update during handoff.
- Under network partitions, we do not make progress.
- In the event of a failure, we can restart the computation at any point.
 - Redundant re-computation doesn't cause problems.
- Dynamic membership.
 - Transfer the portion of the single-assignment store held locally to the target replica.
 - Duplicate notifications are not problematic.

Concurrent map example

Concurrent map example

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


Caveats with non-determinism

- Given the following processes: $\sigma = \{x_1, x_2, x_3, x_4, x_5\}$
 - Process p_0 binds x_1
 - Process p_1 reads x_1 and binds x_2 .
 - Process p_2 reads x_2 , does some non-deterministic operation.
 - Using *is_det* on x_6 , which may or may not be bound based on scheduling.
 - Process p_3 reads x_3 and binds x_4 .
 - Process p_4 reads x_4 and binds x_5 .
- Possible failures:
 - If execution fails in p_0 or p_1 , we can restart.
 - If execution fails in p_3 or p_4 , we can restart p_3 and p_4 , and continue on without worrying about non-determinism.
 - If execution fails in p_2 , what do we do?
 - Local vs. global side-effects?

Future work

- Generalize variables to join semi-lattices.
 - Currently a semi-lattice with two states: bound and unbound.
 - Use the diverse set of CRDTs available in Erlang. [4]
 - Provide eventually consistent computations, which deterministic values regardless of the execution model.
- Provide an analysis tool to determine where you are introducing non-determinism.
 - Similar to the Deadalus work. [2]
 - Possible use for Dialyzer here?
- Explore alternative syntax.
 - Parse transformation.
 - Some other type of grammar.
- Make the library a bit more idiomatic.

References I

 Akka: Building powerful concurrent and distributed applications more easily, 2014.

 P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears.

Dedalus: Datalog in time and space.

Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

 Basho Technologies Inc.

Riak core source code repository.




http://github.com/basho/riak_core.

 Basho Technologies Inc.

Riak dt source code repository.

http://github.com/basho/riak_dt.

References II

-  N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier.
Logic and lattices for distributed programming.
Technical Report UCB/EECS-2012-167, EECS Department, University of California, Berkeley, Jun 2012.
-  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.
-  K. Gilles.
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.

References III



Joel Reymont.

[erlang-questions] is there an elephant in the room? mnesia network partition.

<http://erlang.org/pipermail/erlang-questions/2008-November/039537.html>.



G. Kahn and D. MacQueen.

Coroutines and networks of parallel processes.

In *Proc. of the IFIP Congress*, volume 77, pages 994–998, 1977.





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.

References IV

-  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.
-  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.
-  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.

References V



P. Van Roy and S. Haridi.

Concepts, techniques, and models of computer programming.

MIT press, 2004.



D. Wyatt.

Akka concurrency: Building reliable software in a multi-core world.

Artima, 2013.