Project no.	609551
Project acronym:	SyncFree
Project title:	*Large-scale computation without synchronisation*

European Seventh Framework Programme
ICT call 10

Deliverable reference number and title:	D.4.1 Programming principles and tools
Due date of deliverable:	October 1, 2014
Actual submission date:	17th November 2014
Start date of project:	October 1, 2013
Duration:	36 months
Organisation name of lead contractor for this deliverable:	UCL
Revision:	1
Dissemination level:	PU

Contents

1 Executive summary

The purpose of WP4 is to design the programming principles and tools that we need to program large-scale applications based on CRDTs. This workpackage is done in tight collaboration with WP1, WP2 and WP5. WP1 is documenting the use cases of large-scale applications, which are formalized in WP4. WP2 is building the SyncFree reference platform, called Antidote, and WP4 is building the programming models and tools for using this platform. WP5 has also been responsible for building a benchmarking and testing platform for use with the WP4 programming model. WP4 has made progress in three tracks during the first year for building applications based on CRDTs: the programming model, specification, and verification. We have also explored a possible integration of these tracks based on explicit consistency.

In the second year of the project, we will explore connections between the tracks and how to make them converge. The key concept in all tracks is the *invariants* that are used both to specify eventually consistent applications and to enforce correct behavior during execution. That is, the invariants cover both application intent and execution behavior. The programming model track explores how a suitable programming model can enforce invariants by using language properties. The specification track explores the invariants themselves for eventually consistent applications, exploring how to relax some and maintain others. The verification track checks invariants at various levels: for CRDTs and applications, at development time and runtime.

Programming model The programming model track's main goal is to provide the concepts and software needed to write large-scale distributed applications based on the SyncFree approach (see Section 4). The aim is to provide for both easy programmability and scalability of applications that use as little synchronization as possible. Using CRDTs has the potential to greatly simplify the writing of large-scale distributed applications, because it reduces both the synchronization and nondeterminism inherent in distributed computing. But there are two caveats: first, how to combine individual CRDTs while keeping the good properties, and second, how to add synchronization where needed while keeping the good properties elsewhere.

In the first year, we have addressed the first caveat. We designed and built a deterministic dataflow model called Derflow that is built on top of riak-core. We have generalized this to a new model called Derflow$_L$ that uses CRDTs and allows programming pure CRDT applications. It also enables running our code on the cluster, directly at the replicas. We have started implementation of three application scenarios in this model to explore our models expressiveness: specifically the ad counter, education platform, and score ranking. As we implement these application scenarios, we will learn what, if any, further extensions should be made to the model to support them.

In the second year, the major goal is to extend Derflow$_L$ to become a general framework in the Antidote platform for computing with eventually consistent materialized views that have strong convergence properties. We plan to make a clean interface between Derflow$_L$ and Antidote, to reduce as much as possible the implementation dependencies between the two. We plan to extend Derflow$_L$ with explicit causality and transactions. We will also investigate non-monotonic operations, memory management, and nonfunctional properties such as divergence monitoring and control.

Specification The goal of the specification track is to specify large-scale distributed programs that use eventual consistency and are typically based on CRDTs (see Section 5). Specification of eventually consistent programs using CRDTs is a largely unexplored territory. We started this work by specifying single CRDTs and we then generalized this to allow specifying eventually consistent programs. Our approach is to start from sequential specifications, which are already well understood, and adapt and relax them to concurrent specifications. For concurrent specifications we explored axiomatic and operational specification techniques. In order to explore the different kind of specifications and their relation, we have written formal specifications in Isabelle/HOL for variations of the wallet and ad counter use cases.

In the second year we will continue this work towards a formal development process for eventually consistent applications. We will investigate how to simplify the writing of specifications for eventually consistent programs. We will work on a language to define whole systems and their interfaces and we will investigate the relationships with the work on Derflow_L . We also intend to work on tools for working with specifications, for example to generate part of the implementation automatically from a specification and to generate proof obligations from the specification and the system description.

Verification In the verification track, we have worked on verification of single CRDTs and or high-level formal models of applications, using TLA+ and TLC (see Section 6). We have made low-level formal specifications in TLA+ for several state-based and operation-based CRDTs, including convergence properties and invariants on the convergent states. For applications, we worked on the wallet use case, the ad counter, the FMK healthcare use case, and the score ranking (leaderboard) use case. So far we have used model checking as a form of dynamic verification to verify invariants for small configurations and catch invariant violations.

In the second year we intend to extend this to use static verification (rather than model checking) of application properties, which has the potential to verify much stronger properties such as configurations with arbitrary numbers of replicas and programs of arbitrary length. We intend to build a tool for dynamic execution exploration with multiple instances of Riak.

Ensuring invariants with explicit consistency In addition to the above tracks, we have explored an approach to reduce synchronization for large-scale distributed applications that combines elements of specification and verification and applies them to the programming model (see Section 7). The idea is to specify the application invariants and enforce them explicitly during execution. We have built a prototype platform called Indigo that realizes this approach, and showed its usefulness with a small application similar to the education use case. Application invariants are specified in first-order logic and we use a SAT solver to identify before execution potential conflicts between operations. These conflicts are then handled during execution using a reservation mechanism. This two-phase approach lets us use reservations during execution in exactly the right situations and no more.

2 Milestones in the Deliverable

Milestone S1 (M12) on CRDT consolidation in a static environment. This concerns work packages WP1, WP2, WP3, WP4, WP5. Task 4.1 has contributed to this milestone by focusing on the following goals, as stated in the description of work:

> *T4.1: Basic programming model (feasibility)*
> This task will define the basic programming model for building applications with CRDTs. To achieve a simple and expressive model that supports programming with composition and abstraction of CRDTs, causality determination of CRDT operations, and object purging, we will base the model on a simple process calculus. Programmers do not have to know this calculus, but its existence makes reasoning possible and ensures the absence of unexpected behavior. The programming model will be accessible both through APIs in existing languages (e.g., Java libraries, for direct industrial usage) and through direct language support (e.g., extensions of the Mozart system, which supports language extensibility and transparent distribution). This two-pronged approach will ensure the general applicability of our research results.

3 Contractors contributing to the Deliverable

3.1 UCLouvain

Peter Van Roy, Manuel Bravo, Zhongmiao Li.

3.2 Basho

Christopher Meiklejohn.

3.3 Koç

Serdar Tasiran, Burcu Kulacioglu Ozkan, Erdal Mutlu, Suha Orhun Mutluergil.

3.4 KL

Peter Zeller, Arnd Poetzsch-Heffter.

3.5 Nova

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça.

3.6 INRIA

Mahsa Najafzadeh, Marc Shapiro.

4 Programming model

4.1 Overview

Writing a distributed application is difficult because it introduces significant new non-functional properties over a centralized application, namely geographic distribution and partial failure. This leads to complex issues of fault tolerance, consistency, resource management, latency, and throughput. Building applications based on CRDTs has the potential to greatly simplify the management of these nonfunctional properties, because they support applications that are by default deterministic and require no synchronization. For an application built with CRDTs, the nondeterminism inherent in distributed programming is not visible at the application level. This achieves both consistency and fault tolerance without complicating the application code. However, these advantages come with two strong caveats:

1. They hold for operations on single CRDTs. It is not clear how to write programs that combine CRDTs to build abstractions while still keeping their good properties.
2. They hold for applications that require zero synchronization. In realistic applications, some synchronization is occasionally needed, so the model should keep the good properties when no synchronization is used and give these properties up only when absolutely necessary.

During the first year of SyncFree, we focused on the first caveat. We first designed and implemented a deterministic dataflow model called Derflow in Erlang on top of *riak_core* [3], the same foundation as Antidote, the SyncFree reference platform built in WP2. The Derflow library provides replication and fault tolerance for deterministic dataflow programs using dataflow variables. It allows writing deterministic fault-tolerant programs in Erlang. The semantic foundation of Derflow is concurrent constraint programming (CCP), a process calculus for concurrent programming that uses logical properties to define operational behavior.

In previous work we have used CCP to define a deterministic dataflow language, namely a subset of the Oz language (see chapter 4 of [13]). We observe that the basic data structures of Oz, namely logic variables bound with unification, are actually CRDTs when implemented in a distributed setting. The update operation on logic variables is distributed rational tree unification, which in error-free programs satisfies the strong eventual consistency property [6].

We then generalized Derflow in three ways to make it suitable for general CRDT programming: first by replacing dataflow variables by general CRDTs, second by allowing programs to run at the replicas, and third by separating the language layer from the persistence layer. This extension is called Derflow$_L$ and it implements pure synchronization-free CRDT programming. In the second year of SyncFree, we will integrate Derflow$_L$ with Antidote and extend it for explicit causality, transactions, and various operations related to synchronization and non-monotonicity, according to the needs of our use cases.

Structure The following sections elaborate the approach given above:

- Section 4.2 gives a brief introduction to the CCP process calculus that underlies our work and explains how we have modified and extended it for deterministic dataflow programming.

- Section 4.3 introduces Derflow, an implementation of the deterministic dataflow model in Erlang on top of *riak_core*. We give the Derflow API and illustrate its abilities with programming examples.
- Section 4.4 introduces an extension to Derflow called Derflow$_{\mathrm{L}}$ that provides primitives for programming with CRDTs.
- Section 4.5 gives a nontrivial programming example, namely the ad counter from WP1, to illustrate how Derflow$_{\mathrm{L}}$ works.
- Section 4.6 explains how to install the Derflow and Derflow$_{\mathrm{L}}$ software, which are both available on GitHub under an open source license.

4.2 Deterministic dataflow programming

Deterministic dataflow is a form of concurrent functional programming that preserves all the good properties traditionally part of functional programming, e.g., confluence and referential transparency. Deterministic dataflow has two useful properties for concurrent programming:

- It avoids race conditions by design.
- All higher-order programming patterns become concurrency patterns.

Deterministic dataflow programming is a subset of the Oz language, as explained in chapter 4 of [13]. This subset has been implemented in Scala, giving the Ozma language [4], and in Erlang, giving the Derflow library (explained in this report). The semantic foundation of Derflow and Derflow$_{\mathrm{L}}$ is concurrent constraint programming. We give a brief introduction to the CCP process calculus and we explain how it is adapted for Derflow.

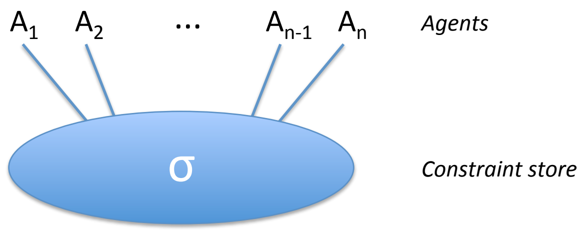

Figure 1: Agents observing a shared store in concurrent constraint programming

4.2.1 Concurrent constraint programming

Concurrent constraint programming (CCP) is a process calculus for concurrent programming introduced by Vijay Saraswat in 1993 [9]. It is based on Michael Maher's logic semantics for concurrent programming, which introduces the key idea of using logical conditions to define a program's control flow [7]. In the core calculus of CCP, concurrent processes A_i (called *agents*) communicate through a shared constraint store σ (see Figure 1). The store consists of a conjunction of basic constraints: $\sigma = \wedge_i c_i$. The agents can do two basic operations with respect to the store, called *tell* and *ask*. The *tell*(c) operation adds the constraint c to the store. The *ask*(c) operation causes the agent to wait until the constraint c is logically entailed by the store. The syntax of core CCP is as follows:

$$\begin{array}{lll} \textit{Declarations} & D ::= & p(\overline{x}) \leftarrow A \mid D, D \\ \textit{Agent} & A ::= & \textit{true} \mid \textit{tell}(c) \mid \sum_{1\leq i\leq n} \textit{ask}(c_i) \rightarrow A_i \mid A \parallel A \mid \exists x.p(\overline{x}) \mid p(\overline{x}) \end{array}$$

The operational semantics of core CCP is given as a transition system $< A, \sigma >$ with a set of agents A and a constraint store σ:

$$\begin{array}{ll} \textit{Tell} & < \textit{tell}(c), \sigma > \rightarrow < \textit{true}, c \wedge \sigma > \\ \textit{Ask} & < \sum_{1\leq i\leq n} \textit{ask}(c_i) \rightarrow A_i, \sigma > \rightarrow < A_j, \sigma > \text{ if } \sigma \models c_j \; (1 \leq j \leq n) \\ \textit{Composition} & < (A \parallel B), \sigma > \rightarrow < (A' \parallel B), \sigma' > \text{ if } < A, \sigma > \rightarrow < A', \sigma' > \\ & < (A \parallel B), \sigma > \rightarrow < (A \parallel B'), \sigma' > \text{ if } < B, \sigma > \rightarrow < B', \sigma' > \\ \textit{Unfold} & < p(\overline{x}), \sigma > \rightarrow < A \parallel \textit{tell}(\wedge_i x_i = y_i), \sigma > \text{ if } (p(\overline{y}) \leftarrow A) \text{ in } D \end{array}$$

Here x_i and y_i are variables, $\overline{x}$ means $x_1, ..., x_n$, and the constraint system is assumed to have an equality operation. This syntax and operational semantics follows chapter 13 of the Handbook of Constraint Programming [8]. The full Oz language is based on an extension of this core CCP calculus with additional concepts whose semantics is given in chapter 13 of [13].

4.2.2 Deterministic dataflow model

We give the simple process calculus semantics of deterministic dataflow, using the core CCP calculus of the previous section as the basis. We modify and extend it with three design choices, of which the first two are:

- First, we restrict the nondeterministic choice $\sum_{1\leq i\leq n} \textit{ask}(c_i) \rightarrow A_i$ to one choice. This guarantees that the resulting concurrent language is deterministic, i.e., by design it cannot have race conditions.
- Second, we define the constraint system underlying σ as containing constraints c of just two forms, namely $x = y$ and $x = a(y_1, ..., y_n)$, where x and y are variables and $a(y_1, \cdots, y_n)$ is a record. This represents the standard record data structures of the Erlang language.

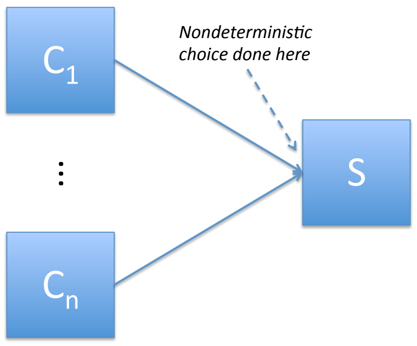

Figure 2: A simple client/server application

It is very important to start with a base language that cannot express nondeterminism. This is a crucial property since we will be computing with CRDTs, which have a deterministic property, namely strong eventual consistency (SEC). This is not a limitation in practice since we will add a nondeterministic operation to the full language. Our experience shows that any realistic program will only need nondeterministic operations in a very few places. For example, consider a client/server application (see Figure 2). A single server can serve a large number of clients. In the simplest case, only one nondeterministic operation is needed in this program, namely the point where the server accepts commands from the next client. This point does a nondeterministic choice of the next client to be served. The server and client programs can however be completely deterministic.

To obtain our final deterministic dataflow language, we make a final design choice:

- Third, we add the ability to create *closures*, i.e., references to declarations $p(\overline{x}) \leftarrow A$ from within a program. We do this in a simple way: we annotate each declaration with a constant value and we allow constraints to contain these constants. We denote these new constants by Greek letters ξ. We also allow agents A to contain declarations. Declarations and procedure calls then take the following forms:

  Named declarations $D ::= \xi : (p(\overline{x}) \leftarrow A) \mid D, D$
  Named unfold $< x(\overline{x}), \sigma > \rightarrow < A \parallel \textit{tell}(\wedge_i x_i = y_i), \sigma >$
  if $\sigma \models x = \xi$ and $\xi : (p(\overline{y}) \leftarrow A)$ in D

In the following section we present the Derflow programming model, which implements this semantics in Erlang using the riak-core library as a base. Derflow extends the pure CCP semantics with a well-defined distribution behavior. In particular, it replicates single-assignment variables to provide fault tolerance.

4.3 Derflow: distributed deterministic dataflow in Erlang

Erlang implements a message-passing execution model, which is inherently nondeterministic. Nondeterminism drastically increases the difficulty to debug programs or verify program behavior, especially in a distributed setting. We propose Derflow, a new execution model for Erlang that implements deterministic dataflow programming. It provides concurrency yet eliminates all observable nondeterminism. Lazy execution and streams are provided as useful programming techniques. Last but not least, nondeterminism can be added when necessary.

Derflow is based on a highly available, scalable single-assignment data store that is implemented on top of the $riak\_core$ open source distributed systems framework.

4.3.1 Semantics of Derflow

Deterministic dataflow The deterministic dataflow model uses a single-assignment data store, which is shared among all processes in the deterministic dataflow computation. It stores dataflow variables. Each dataflow variable is identified by a key and can have three different states: unbound, bound to an Erlang term or bound to another dataflow variable which is unbound, denoted as partially bound. When a dataflow variable is bound, all dataflow variables that are bound to it will also be bound to its value.

As the name 'single-assignment' suggests, dataflow variables can be bound once, or be bound several times to the same value only. When binding a dataflow variable that is already bound to a different value, a program error is generated. Concurrency can be added with Erlang primitive *spawn*.

The following operations are supported for dataflow variables:

- $declare()$: create a dataflow variable and return the key of the dataflow variable.
- $bind(x_i, v_i)$: bind the dataflow variable x_i to v_i. v_i can be an Erlang term that represents a value, or can be another dataflow variable. In case x_i is already bound to v_j while v_i does not match v_j, the execution of the deterministic program will terminate. The *bind* operation corresponds to a *tell* in the concurrent constraint calculus.
- $bind(x_i, m, f, args)$: bind the dataflow variable x_i to the result of executing *m:f(args)* with *module* m and *function* f in the module.
- $read(x_i)$: returns the term bound to x_i. In case x_i is unbound or partially bound, the execution is suspended until the x_i is bound. The *read* operation corresponds to an *ask* in the concurrent constraint calculus.

Streams Streams are a useful technique that allows processes to communicate and synchronize in concurrent programming. A stream is implemented as a list of dataflow variables, with an unbound variable as the tail of the list. In terms of constraints, a stream's tail is represented by a variable x, the stream is extended by a tell of the constraint $x = cons(y, x')$ where x' is the new tail, and the stream is read by an ask of the constraint $\exists y \exists x'. x = cons(y, x')$. Multiple processes can read a stream concurrently without compromising determinism. Nevertheless, there can be only one producer for a stream, in order to ensure determinism.

The following operations are supported for streams:

- $extend(x_i)$: extend the stream by creating a pointer to a new unbound tail, x_{i+1}.
- $produce(x_i, v_i)$: extend the stream by binding its tail x_i to v_i and creating a new unbound tail, x_{i+1}.
- $produce(x_i, m, f, args)$: extend the stream by binding its tail x_i to the result of executing *m:f(args)* with *module* m and *function* f in the module, and creating a new unbound tail, x_{i+1}.
- $consume(x_i)$: read the dataflow variable x_i and returns a pair of the next element in the list and the next tail. In terms of constraints, this waits until x_i is bound to a constraint of the form $x = cons(y, x')$, and returns y and x'.

Laziness Lazy execution delays the evaluation of an expression until the valued is needed somewhere else in the program. It can potentially improve the performance of programs by avoiding unnecessary computation. Moreover, it enables writing programs that create infinite data structures since elements will only be created when it is needed.

We provide one operation to enable lazy execution:

- $wait\_needed(x_i)$: suspend the caller process until x_i is needed, i.e. some process is executing $read(x_i)$. Then the caller continues execution and can bind x_i.

Nondeterminism, when necessary Although deterministic dataflow is a powerful concurrent programming paradigm, not all programs can be written in a purely deterministic manner. For example, a simple client-server application needs nondeterminism since the server should handle requests from any client, but it can not know the order of requests in advance.

The client-server program can be written by using the original Erlang message passing primitive (*A ! B*). Besides that, we propose one primitive that also introduces nondeterminism:

- $is\_det(x)$: check whether a dataflow variable x is bound or not. This introduces nondeterminism, as during different execution the result of this call can vary due to process scheduling and network delay.

Failure handling Failures also introduce nondeterminism. Therefore, a deterministic program can become nondeterministic if care is not taken to handle failures in a deterministic manner.

We identify two kinds of possible failures and proposes methods for handling these failures:

- **Computing process failures:** an Erlang process that uses dataflow variables failed. If the failed process is written in a deterministic manner, it can be safely restarted without introducing any accidental nondeterminism.
- **Dataflow variable failures:** a dataflow variable in the single-assignment store is not reachable. It will block processes trying to access the failed dataflow variable forever. Simply restarting the process does not guarantee progress. In case the required dataflow variables were created by other processes (i.e. the restarted process will not recreated these dataflow variables) and all processes storing them have failed, the restarted process will be blocked immediately. This case can be handled by using the Erlang primitives *monitor/2* and *link/1* for building custom supervision trees, in which a process will be re-executed if any of its created dataflow variables have failed.

4.3.2 Distribution model

The distribution model of Derflow centers around Dynamo-style [11] partitioning and replication of the single-assignment variable store. In this model, each variable is stored on a set of replicas for a given partition, where its placement in a particular partition is determined through consistent hashing and hash-space partitioning. This distribution model is illustrated in Figure 3.

When *declare*, *bind* or *read* operations are performed, they are performed against a quorum of the replicas for a given object – with a default replication factor of 3, this gives the system the ability to tolerate 1 failure without sacrificing progress – therefore, our system remains available as long as $(N/2) + 1$ nodes are online.

However, the system must still deal with partial writes – where a request fails because the request did not successfully complete at a majority of replicas but may have completed at some. Given that all of our variables in the single-assignment store are immutable, as long as our applications do not introduce any accidental nondeterminism, we can reissue the request without sacrificing determinism.

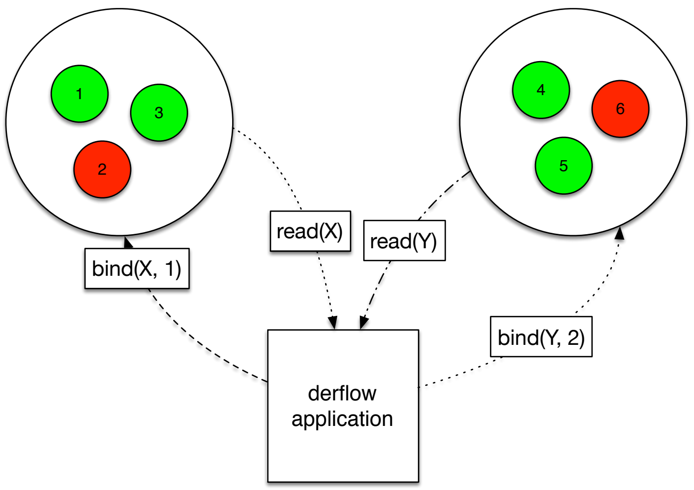

Figure 3: Two replication groups, where a quorum of nodes are available for writes; in thix example, variable X is partitioned into the first replica set, and variable Y is partitioned into the second. The application sits outside the cluster and makes round trips to the data store for each operation.

The Derflow paper (see appendix) goes into futher detail about how our distribution model is both highly-available and fault-tolerant.

4.3.3 Programming examples in Derflow

In this section we describe some important programming patterns in Derflow.

Concurrency transparency Programs written in Derflow can be re-organized into a series of independent processes performing parts of the computation without having to worry about introducing accidental nondeterminism. This effectively allows programs to add arbitrary concurrency to their programs – in the form of distribution or parallel computation – without having to worry about data races or bugs due to nondeterministic process inter-leavings.

One example is a sequential map function that receives a stream of inputs and applies a function to each element resulting in 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 introducing nondeterminism. 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, M, 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* should be performed concurrently. This allows the map function to read the next element from the input stream without waiting for the function evaluation to complete. One possible application of this pattern is parallelism: if processes are executed on separate cores then the concurrent map will be faster than its sequential counterpart.

Concurrent deployment In concurrent deployment, we can 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 a process 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.
```

Laziness The following examples show how the *wait_needed* primitive can be used to implement lazy functions. The example below shows a lazy version of a sorting algorithm for a list of numbers.

```
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.
```

We achieve laziness in this computation by breaking down the computation into a series of processes; each of which we immediately call *wait_needed* on. By performing this call, we suspend its execution until the result is required by another process; at this point, we awaken the suspended process.

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:

- **Consumer:** Asks for items eagerly.
- **Producer:** Only produces items when it is needed. This is achieved by calling *wait_needed* for the next element after it has produced an item.
- **Bounded buffer:** Takes the output stream from the **producer** and the input stream of the **consumer**, initially allocating the buffer size for the producer, and extending the stream as values are consumed.

Process supervision Since redundant computation does not change the result of Derflow programs – this follows from both the concurrency transparency and concurrent deployment properties – programs that do not use the nondeterministic or lazy operators are idempotent. Given this, we can devise a simple supervisor which restarts failing processes when a problem is detected.

```
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* message; the former enables monitoring of a particular process; the latter is received when processes fail. In both cases, the supervisor restarts the processes with the original arguments, similar to the supervision strategies used by Erlang/OTP.

4.4 Derflow$_L$: an extension of Derflow for CRDT programming

Derflow$_L$ extends the programming model in several ways. Specifically, it:

- Generalizes single-assignment variables to bounded join-semilattice variables.
- Provides the ability to run computations in the Derflow cluster at the replicas.
- Separates the persistence layer from the language semantics.

4.4.1 Generalization to lattice variables

Derflow$_L$ extends our programming model to state-based CRDTs, as described by Shapiro et al. in [22]. We start by generalizing our single-assignment model to a bounded join-semilattice (referred to herein as lattices) with two possible states, unbound and bound, as shown in Figure 4. When performing this generalization, we are able to represent our previously discussed *bind* semantics in a different way: we allow *bind* operations to succeed when the value we are binding triggers an inflation of the lattice. We use the top state of the lattice to represent the error state which is reached when we attempt to rebind a variable to a different value.

As we begin to extend our model to a larger collection of state-based CRDTs, or lattices, the size of possible states for each type grows as complexity of the data type grows. While previously we have bound opaque values to each dataflow variable, when operating with CRDTs we bind the internal state of the CRDT, which contains a *query* function for determining its current value.

To explore how this complexity grows in size, examine the following:

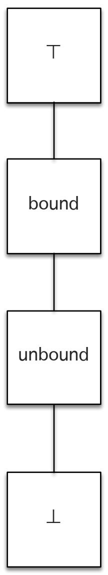

Figure 4: Single-assignment variables generalized to a bounded join-semilattice.

- Last writer wins registers (LWW-Register) only track the current value and a timestamp – the least upper bound, or join, of two registers always returns the latest timestamp and value.
- Grow-only sets (G-Set) track only the number of elements added by each actor – the least upper bound, or join, of two sets simply unions the products of the actors and elements added.
- Remove-once sets (2P-Set) track two grow-only sets – the least upper bound operation of two sets simply performs the join operation across both sets.
- Observed-remove sets (OR-Set) provide the ability for arbitrary additions and removals of elements – this requires uniquely tracking additions and removals performed by each actor in two sets, unioning the sets on join.

The four types above are only a sampling of the CRDTs which have been written about [22], each of which can also have different implementations while retaining similar semantics. However, two things remain true:

- As updates are performed against a given CRDT, the internal state increases; this is an inflationary update of the lattice.
- The *query* function for a given type will always return the current value of the CRDT, of which observations over time are not necessarily monotonic (monotonic for CRDTs such as the G-Set, but not for the 2P-Set or OR-Set).

In extending this model to CRDTs, we allow variables to rebind as long as the change is always inflationary in relation to the lattice – to explore a similistic example: if x_i is initially bound to the grow-only set containing $[a, b, c]$, we allow a subsequent bind to $[a, b, c, d]$, while ignoring a bind to $[a, b]$.

Additionally, we supply a threshold read primitive, similar to the threhold read primitive discussed by Kuper and Newton in [1]. This provides a blocking read operation until the lattice reaches a particular state, in which the threshold is returned regardless of state. This supports ”deterministic by construction” applications. It is still unclear how this operation will grow in usability as we move to more complicated CRDTs given the growth in state as the complexity of CRDTs increases.

To support this new behavior, we extend our API like so: [1]

- $declare(t_i)$: create a dataflow variable of a given type t_i and return the key of the dataflow variable.
- $bind(x_i, v_i)$: bind the dataflow variable x_i to v_i. v_i can be an Erlang term that represents a value, or can be another dataflow variable. In the case that x_i is already bound to v_j, we compute the *join* of both v_i and v_j and bind if the result of the *join* is an inflation.
- $read(x_i)$: returns the value of x_i once it has been bound (its value is different from $\perp$); until this occurs, execution is suspended.
- $read(x_i, v_i)$: returns the value of x_i once the value bound to x_i is an inflation of v_i; until this occurs, execution is suspended.

4.4.2 Running applications at the replicas

While the distribution model of Derflow is both fault-tolerant and highly-available, the communication overhead for small programs is high. The reason for this is that each Derflow operation requires a full round-trip to the single-assignment store.

As an optimization, we can move the computation of our applications to the server. We provide two additional primitives *register* and *execute* for registering applications with the cluster and executing them. With this change, we no longer distribute variables across the partitions as shown in Figure 3, but rather entire programs. Each of these programs operate on their local state, and once have observed all updates in the system, it is suffient to contact only one replica in the system for the result of the program. To increase recency in a system where messages are constantly being delivered, more replicas can be contacted and their states merged using the normal CRDT *join* operation. This is an extension of the strong eventual consistency model discussed by Shapiro et al. in [22] from data types to programs. The updated distribution model is illustrated in Figure 5.

4.4.3 Separation of layers

As previously discussed, both Derflow and Derflow$_L$ rely on a highly-available, distributed data store for storage of dataflow variables. In the default implementation, the persistence layer used is the Erlang Term Storage (ETS) system, which provides ephemeral term storage.

In order to keep Derflow and Derflow$_L$ as modular as possible, and to enable testing of Derflow independently, and on top of both the SyncFree reference platform and Riak, we have separated the persistence layer from the distribution and language layers.

Additionally, this allows us to tests the language semantics completely independently of the persistence layer as well as support backend specific testing. As a first step towards

[1] It is important to note that when discussing *inflations* we are referring to inflations of internal state of the CRDTs, and not necessarily the observed value – for instance, in a set where elements can be arbitrarily added and removed, the internal state may grow, where the observed value may appear to shink.

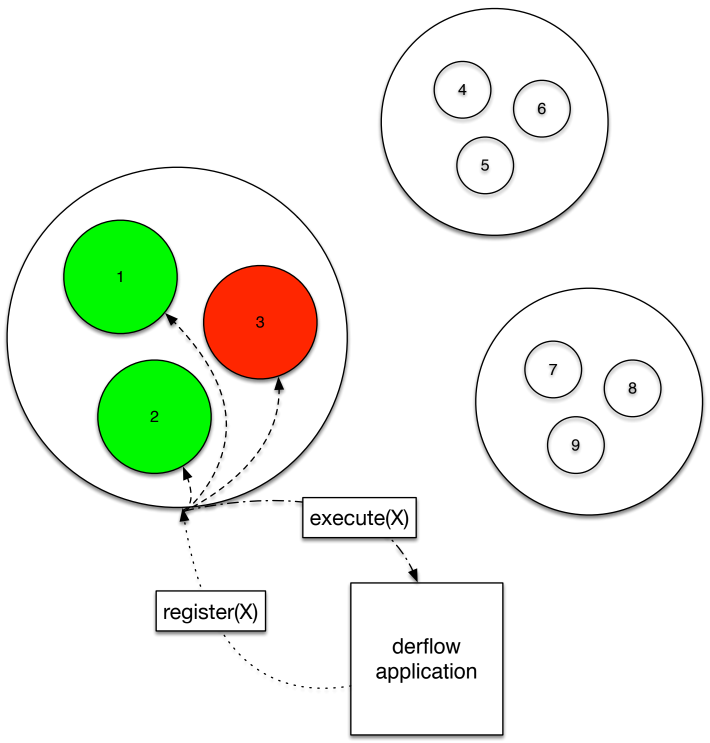

Figure 5: Updated Derflow$_L$ distribution model where applications are pushed to the replicas; each program is rewritten to operate over local state. *execute* contacts a quorum of replicas for the result, which is merged.

this, we've implemented an Erlang QuickCheck model for our ETS backend, the only backend we've currently implemented. This model could be easily extended for other backends.

Here's an example of how we verify invariants for read operations.

```
read(Id, Threshold) ->
   derflow_ets:read(Id, Threshold, ?ETS).

read_pre(S) ->
   has_variables(S).

read_pre(S, [Id, Threshold]) ->
   is_read_valid(S, Id, Threshold).

read_args(#state{store=Store}) ->
   Variables = dict:fetch_keys(Store),
   ?LET(Variable, elements(Variables),
            begin
               Threshold = threshold(Variable, Store),
               [Variable, Threshold]
            end).

read_post(#state{store=Store}, [Id, Threshold], {ok, V, _}) ->
   case dict:find(Id, Store) of
      {ok, #variable{value=Value}} ->
```

```
            Value == V;
        _ ->
            false
    end.
```

Erlang QuickCheck takes a state-machine approach to testing stateful applications. The QuickCheck model is responsible for modeling application state locally and for each command, in this example *read*, a series of pre- and post-conditions are used to validate that a randomly generated command should be executed, and that the resulting state after the call is correct. Digging into the example above, each function here corresponds to one aspect of the state-machine:

- **read:** Responsible for mapping the *read* operation in the model, to the persistence layer specific call which performs the read operation.
- **read_pre:** Responsible for validating a randomly generated *read* operation should be performed or ignored; this will verify that it does not generate a read operations with a threshold which is not valid during this execution.
- **read_args:** Given the current state of our model, when generating a random *read* operation, determine how to generate the argument list for the call. If the variable is a lattice, generate the read with a threshold; otherwise, generate a read operation.
- **read_post:** Verify that for each successful call to the persistence layer the result returned from the call is correct.

4.5 The ad counter use case in Derflow$_L$

To illustrate Derflow$_L$ we have selected the ad counter use case from WP1. This use case shows well how a large number of nodes (mobile nodes and ad server nodes) can coordinate their activities through shared CRDTs (see Figure 6). It is important to note that this example does not need transactions or locks; all coordination is done through shared CRDTs. The example runs using the current API of Derflow$_L$ (see code in Figure 7), but the code may change in the future since the API is work-in-progress and is rapidly changing as we figure out the best ways to express computations.

Nowadays, mobile apps usually contain advertisements. Accurately counting the number of impressions is crucial for an advertisement platform, as advertising companies pay per impression. However, this is challenging to achieve in the mobile environment, because it is a highly dynamic, error-prone, and large-scale environment. CRDT counters are a reasonable solution to this problem, as they can scale to extreme numbers of concurrent clients. To illustrate this example we use the Derflow$_L$ API to write a simple advertisement counter that consists of a server program and a client program. The server program stores a list of ads in a grow-only set (G-Set), with each advertisement containing its own grow-only counter (G-Counter) of impressions. Each server performs a threshold read operation on the growing counter, and once a threshold of impressions has been met, notifies the clients to stop displaying the ad.

We conclude this example with two remarks about the counters. First, splitting the increment into separate read and bind operations is correct here, since grow-only counters are implemented as lists of pairs of client identity and count value. Another possibility would be to add an increment operation to the Derflow$_L$ API. Second, this implementation uses counters that suffer from the CAP theorem: counter values may skew in either

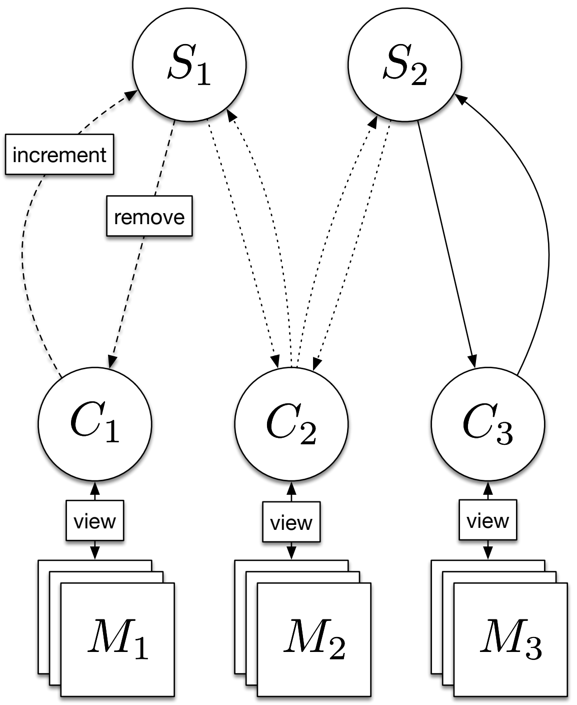

Figure 6: Example advertisement counter written in Derflow$_L$. Mobile clients M_i talk to a local Derflow client C_i to determine which advertisements to display. Clients issue increment operations back to an ad server S_i for each advertisement i they view, while the server sends messages back to the clients when an advertisement has been displayed the maximum number of allowed times.

direction in the event of failures. Fixing this problem requires a counter that correctly collects increments but that does not provide sequencing of counter values, such as idempotent counters or Handoff Counters [10].

4.6 Installation and configuration of Derflow and Derflow$_L$

Derflow and Derflow$_L$ are open source and available on GitHub at `https://github.com/SyncFree/derflow`. Both Derflow and Derflow$_L$ share the same code base; Derflow$_L$ provides additional language extensions to Derflow and leverages the existing codebase. For the following sections, we will use Derflow to refer to the implementation, which contains both the Derflow and Derflow$_L$ primitives.

The following section details how to obtain the Derflow source code, build, run, and test the programming model.

4.6.1 Prerequisites

Prerequisites for obtaining and building Derflow are the following: git, any of the Erlang R16 releases (we recommend R16B02, which is the runtime we have used to perform tests), and a C compiler which can create executables.

```
%% @doc Server functions for the advertisement counter. After 5 views,
%% disable the advertisement.
server(Ad, Clients) ->
   lager:info("Server launched for ad: ~p", [Ad]),
   %% Wait until number of ad impressions reaches threshold.
   {ok, _, _} = derflow:read(Ad, 5),
   lager:info("Threshold reached; disable ad ~p for all clients!", [Ad]),
   lists:map(fun(Client) ->
            %% Tell clients to remove the advertisement.
            Client ! {remove_ad, Ad}
      end, Clients),
   io:format("Advertisement ~p reached display limit!", [Ad]).

%% @doc Client process; standard recursive looping server.
client(Id, Ads) ->
   lager:info("Client ~p running; ads: ~p~n", [Id, Ads]),
   receive
      view_ad ->
         %% Choose an advertisement to display and increment counter.
         Ad = hd(Ads),
         lager:info("Displaying ad: ~p from client: ~p~n", [Ad, Id]),
         %% Update ad by incrementing value.
         {ok, Value, _} = derflow:read(Ad),
         {ok, Updated} = riak_dt_gcounter:update(increment, Id, Value),
         {ok, _} = derflow:bind(Ad, Updated),
         client(Id, Ads);

      {remove_ad, Ad} ->
         lager:info("Removing ad: ~p from client: ~p~n", [Ad, Id]),
         client(Id, Ads -- [Ad])
   end.
```

Figure 7: This code shows both the server and client processes, which are each composed of a single function.

4.6.2 Obtaining and Building Derflow

Obtaining and building Derflow is as easy as cloning the code from GitHub and compiling using the provided make targets.

```
$ git clone git@github.com:SyncFree/derflow.git
$ cd derflow
$ make
```

Two additional development make targets are also provided which build stand alone executable Erlang releases: **make stage** and **make stagedevrel**. Each of these targets build a development release for local testing, with the latter providing six unique copies of the development release to be used in cluster simulation.

4.6.3 Testing and Program Execution

In the process of developing Derflow, we also adapted the open source testing framework for Riak, Riak Test, to assist in reproducible executions of our programs. Obtaining, building, and configuring our test harness is extremely similar to the steps needed to build Derflow itself.

```
$ cd $PATH_TO_DERFLOW/..
$ git clone git@github.com:basho/riak_test.git
$ cd riak_test
$ make
```

Once done, the provided **riak_test.config** file must be moved to your home directory (as **.riak_test.config**), and the paths in the file updated to reflect paths on your local machine. When completed, the entire Derflow test suite can be executed using the following commands:

```
$ cd ../derflow
$ make riak-test
```

The **make riak-test** target will configure a cluster of nodes, running on the local machine, remotely load example Derflow applications, and ensure that they complete with the correct result.

5 Specification

5.1 Overview

A specification of a system is a set of properties that the system should satisfy [18]. The properties can be described in various ways. Usually development starts with an informal description of what the system is supposed to do. Then this specification is refined and adjusted, until a working system is implemented. This process can be improved by using formal specifications and methods, which means that the specifications have to be expressed in some formal language. Using a formal language enables the use of tools to work with specifications, so problems can potentially be discovered earlier, before the whole system is implemented and tested. Additionally a formal specification is a prerequisite for verifying that a system is implemented correctly and it can be used to (automatically) derive tests.

However, the lack of a global state makes it difficult to write specifications for eventually consistent applications. It is not possible to simply define invariants, post-conditions, or other properties about a program in the same way as for sequential programs, especially if there is no central or primary database. In this section we discuss several approaches to specify applications with weak consistency guarantees. The remainder of this Section is structured as follows:

- Section 5.2 discusses known techniques for formally specifying sequential applications.
- Section 5.3 introduces different ways to specify the concurrent behavior of applications.
- Section 5.4 shows how sequential and concurrent specifications can be related.
- Section 5.5 discusses the problems of formulating assertions and invariants in the setting of eventually consistent distributed state.
- Section 5.6 gives the current status of our work related to specifications and explains how we plan to use specifications in our upcoming work, and what role specifications play as part of the programming model.

Running example: Virtual wallet. As a running example we will use a simple virtual wallet application, which is a simplified variation of a use case from WP1. Users of the wallet can deposit money into the wallet and then use it to buy items, for example additional items in a game. In order to buy an item, a user needs to have enough money in his wallet and he must not have bought the same item already. Of course, users are also provided with a function to check their current balance.

We formalize the operations provided by an application using an algebraic datatype. The datatype has a constructor for every operation of the application and the constructor has parameters for the input and output parameters of the operation. As a convention we prefix output parameters with *res_*. For the virtual wallet example, we have the following operations:

$$\begin{array}{rcl} \mathit{call} & = & \mathit{Deposit}(\mathit{amount} : \mathit{int}) \\ & \mid & \mathit{BuyItem}(\mathit{item} : \mathit{string},\ \mathit{cost} : \mathit{int},\ \mathit{res\_success} : \mathit{bool}) \\ & \mid & \mathit{GetItems}(\mathit{res\_items} : \mathit{string\ set}) \\ & \mid & \mathit{GetBalance}(\mathit{res\_balance} : int) \end{array}$$

5.2 Sequential specifications

For many applications it makes sense to start from a sequential specification. This is especially true for applications, which are usually implemented using a central database with serializability or a similar consistency model in mind. But also for other kinds of applications it can help to start from a sequential specification, because it is much easier to understand and it makes the main intention of each operation clear. Of course there are also some applications, where distributed manipulation of data is a core concept, for example version control systems. Then it is better to take a different approach to specification.

We start this chapter by formalizing sequential specifications and then continue to discuss how they can be adapted in concurrent specifications. We represent a sequential execution by a sequence of calls. For our running example of the virtual wallet, such a sequence could be:

$$\begin{aligned}\textit{sequentialExample} \equiv [\ & \textit{Deposit}(50),\\ & \textit{BuyItem}(\textit{``Sword''}, 35, \textit{True}),\\ & \textit{BuyItem}(\textit{``Boots''}, 20, \textit{False}),\\ & \textit{BuyItem}(\textit{``Hat''}, 10, \textit{True}),\\ & \textit{GetItems}(\{\textit{``Sword''}, \textit{``Hat''}\}),\\ & \textit{GetBalance}(5)\ ]\end{aligned}$$

First the player deposits 50 credits to his account. Then he buys a sword, tries to buy some boots, but this operation fails because he has insufficient founds. So then he decides to buy a hat, which is a bit cheaper. The player then checks his items and his balance and finds it to be as expected.

We now want to formally specify for each such sequence of calls, whether it is allowed or not. A specification could be just a predicate which takes a sequence of calls and assigns it a true or false value. Often sequential specifications are given in a state-based form, which describes how every call changes the state of the system, what the valid result values are, and also what the possible initial states of the system are.

For our virtual wallet example the state consists of the balance and the set of bought items:

$$\textit{state} = (\textit{balance} : \textit{int},\ \textit{items} : \textit{string set})$$

The effects and result values of single calls are specified using a simple function, which uses pattern matching to distinguish the different types of calls:

$$\begin{aligned}
& seqSpec : (call, state, state) \rightarrow bool\\
& seqSpec(Deposit(a), s, s') \equiv (s' = s[balance := balance(s) + a])\\
& seqSpec(BuyItem(i, c, True), s, s') \equiv (balance(s) \geq c \wedge i \notin items(s)\\
& \qquad\qquad \wedge (s' = s[balance := balance(s) - c, items := items(s) \cup \{i\}]))\\
& seqSpec(BuyItem(i, c, False), s, s') \equiv (s = s')\\
& seqSpec(GetItems(is), s, s') \equiv (s = s' \wedge is = items(s))\\
& seqSpec(GetBalance(b), s, s') \equiv (s = s' \wedge b = balance(s))
\end{aligned}$$

The most interesting case in this example is the call to *BuyItem*, since it involves a change of state and a result value. If the result value is true, then the item must be bought successfully. That means that in the pre-state the balance was high enough to be able to afford the item, and the same item was not bought already. Then in the post-state the balance and the set of owned items must have been adjusted accordingly. When the result value is false, we only specify that the state must be unchanged. Note that this leaves some freedom for the implementation: Buying an item may fail even when the two pre-conditions are met. Specifying that this should not happen too often can be denoted separately when addressing the quality of service (QoS).

In general, a sequential state-based specification consists of two parts: A set of possible initial states and a specification of the calls. The specification of calls is a predicate, which states whether a call is valid for given pre- and post-states.

$$\begin{aligned} \textit{sequentialSpec} = (&\textit{initialStates} : \textit{state set}, \\ &\textit{callSpec} : (\textit{call} \times \textit{state} \times \textit{state}) \rightarrow \textit{bool}) \end{aligned}$$

We can then define when a given sequence $c_0, \ldots, c_n$ of calls is valid with respect to a specification $spec$. To justify that a sequence is valid, there must be a sequence of states, such that the first state is in the set of possible initial states defined in the spec, and each state in the sequence of states must be linked to the next state in the sequence by the respective call and its specification.

$$\begin{aligned} c_0, \ldots, c_n \models spec \quad \Leftrightarrow \quad & \exists s_0, \ldots, s_{n+1} : s_0 \in \textit{initialStates}_{spec} \\ & \wedge \forall_{0 \leq i \leq n} : \textit{callSpec}_{spec}(c_i, s_i, s_{i+1}) \end{aligned}$$

5.2.1 Adapting sequential specifications to concurrent settings

Classical concepts to adapt a sequential specification to a concurrent setting are sequential consistency [17] or linearizability [16]. In these notions, a concurrent execution must behave equivalently to some sequential execution, where the sequential execution is subject to some additional constraints. This is quite a strong property to demand of concurrent system, and thus it usually inhibits some implementations with better nonfunctional properties. Often the specification can be relaxed a bit in order to gain better performance or higher availability in a distributed system. Still, the relaxed behavior should fulfill the original intention of the system and not be completely different. Also it would be nice, if we could keep the understandable sequential specification and just adapt and relax it at certain points, to get a concurrent specification.

In the virtual wallet example we would like to allow the calls to *GetItems* and to *GetBalance* to return slightly out of date values, as this would allow for a more efficient implementation.

In order to formalize these ideas, we first need a notion of what precisely a concurrent execution is, and how a concurrent specification can be formalized. This is what we discuss in the next section. In Section 5.4 we then come back to the relation of sequential and concurrent specifications.

5.3 Concurrent specifications

A concurrent execution can be described as a set of sequential processes communicating with each other. Depending on the architecture of the system, a sequential process can be a session, a transaction, or a whole machine. Also the communication can have very different forms depending on the system. Therefore, a general framework has to abstract from those details in executions.

For modeling concurrent executions we use a notation very close to the one used by Burckhardt, Gotsman, and Yang [14], which is based on more abstract relations between operations. For every kind of system, these abstract executions then have to be mapped to concrete executions.

A concurrent execution is described by an *executionContext* consisting of the following components:

$$\begin{aligned}\textit{executionContext} = (&\textit{actions} : int \rightharpoonup call,\\ &\textit{sessionOrder} : (int \times int)\ set,\\ &\textit{visibilityOrder} : (int \times int)\ set,\\ &\textit{arbitrationOrder} : (int \times int)\ set)\end{aligned}$$

To simplify the notation of examples, we use integers to identify actions. The partial function *actions* gives the call belonging to each action. In addition to these actions there are three relations on the actions: *sessionOrder*, *visibilityOrder*, and *arbitrationOrder*. In a valid *executionContext* we demand that all these relations are acyclic. Additional constraints could be added to describe the consistency model of the used system (see [14]).

The intuition behind the three relations is as follows: The session order captures the idea of sequential processes. Within a process, the actions are usually ordered in a sequence. The visibility order captures the message passing between processes by stating which actions have been made visible to other actions via messages. The arbitration order, as the name suggests, is usually used to break up ties between concurrent operations. For example this could be a lexical ordering on pairs of timestamps and replica identifiers, i.e. the events are ordered by timestamps first and an order on replica identifiers is used to break ties between identical timestamps.

Figure 8 shows an example execution of the virtual wallet example. The graph consists of two time lines for two replicas *R1* and *R2*. The different actions are denoted by the numbered nodes and the arrows denote relations between actions. The specific parameters of a call are written next to the node. The dashed line denotes a visibility relation, the plain lines denote visibility and session order relation. In the formal notation we can

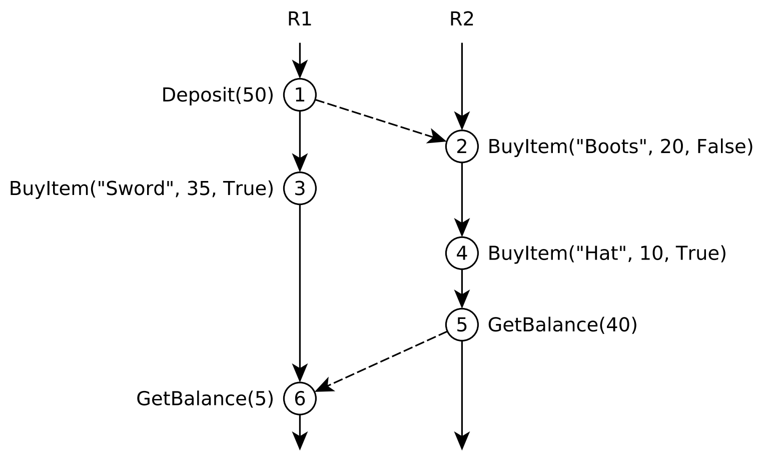
R1
R2
Deposit(50) 1
2 BuyItem("Boots", 20, False)
BuyItem("Sword", 35, True) 3
4 BuyItem("Hat", 10, True)
5 GetBalance(40)
GetBalance(5) 6

Figure 8: Concurrent execution in virtual wallet example

write this example as follows:

$$
\begin{aligned}
&\textit{concurrentExample} \equiv (\\
&\quad \textit{actions} = [\\
&\qquad 1 \mapsto \textit{Deposit}(50),\\
&\qquad 2 \mapsto \textit{BuyItem}(\textit{``Boots''}, 20, \textit{False}),\\
&\qquad 3 \mapsto \textit{BuyItem}(\textit{``Sword''}, 35, \textit{True}),\\
&\qquad 4 \mapsto \textit{BuyItem}(\textit{``Hat''}, 10, \textit{True}),\\
&\qquad 5 \mapsto \textit{GetBalance}(40),\\
&\qquad 6 \mapsto \textit{GetBalance}(5)],\\
&\quad \textit{sessionOrder} = \{(1,3),(3,6),(2,4),(4,5)\}^+,\\
&\quad \textit{visibilityOrder} = \{(1,2),(5,6),(1,3),(3,6),(2,4),(4,5)\}^+,\\
&\quad \textit{arbitrationOrder} = \{(x,y) : x < y\})
\end{aligned}
$$

Here the notation $^+$ denotes the transitive closure of the relation, in general the framework also allows a non-transitive visibility order. Note that the call to *GetBalance* in action 5 returns a slightly stale value, because one call to *BuyItem* is not yet visible at the second replica. We want to allow behavior like this to enable highly available and efficient implementations. Although we relax the specification, we still want the system to maintain some invariants and properties from the sequential specifications: The balance should never fall below zero and the restriction that each item can only be bought once should be considered. As it is not possible to enforce the uniqueness of bought items without synchronization, we resolve conflicts after the fact. When we see that an item has been bought twice by a customer, we only charge him once. In the unlikely case, that the price differs for the two purchases, we choose the customer friendly alternative and only charge for the lower price.

In the next two subsections, we now discuss how the informal description above can be expressed in a formal way, suitable for verification and testing. We describe two different techniques which we found suitable for the specification of concurrent applications:

An axiomatic technique which describes the system in a more mathematical way, and an operational, state-based technique, which is closer to the implementation and the used architecture, and tries to describe the single steps of a system.

5.3.1 Axiomatic specifications

An axiomatic specification is simply a predicate, which checks whether a given execution context is valid. So there is a lot of freedom in writing an axiomatic specification, it is even possible to write down specifications for which no implementation exists. However, there are some patterns which can be used in many specifications and we can collect these patterns to simplify the specification.

The basis of a specification is usually a quantification over all actions in the context, checking that for each action certain properties hold true. For our virtual wallet example, this can be written as follows:

$$\begin{aligned}
&\textit{axiomaticSpec}(\textit{ctxt}) \equiv (\forall a \in dom(\textit{actions}_{\textit{ctxt}}) : \\
&\quad \textbf{case } \textit{actions}_{\textit{ctxt}}(a) \textbf{ of} \\
&\qquad \textit{GetBalance}(x) \Rightarrow x \geq 0 \land x = \textit{specBalance}(\textit{ctxt}, a) \\
&\quad\ \ \mid \textit{GetItems}(is) \Rightarrow is = \textit{specItems}(\textit{ctxt}, a) \\
&\quad\ \ \mid \textit{BuyItem}(i, c, \textit{True}) \Rightarrow i \notin \textit{specItems}(\textit{ctxt}, a) \\
&\quad\ \ \mid \textit{BuyItem}(i, c, \textit{False}) \Rightarrow \textit{True} \\
&\quad\ \ \mid \textit{Deposit}(x) \Rightarrow \textit{True})
\end{aligned}$$

For the verification we use two auxiliary functions: *specBalance* calculates the balance for a given execution context at the time of a given action, *specItems* similarly calculates the set of bought items. Using those functions, we can specify that each call to *GetBalance* should return the value as calculated by *specBalance* and that additionally the returned value should not be negative.

The important part of the specification is in the two auxiliary functions. We first describe *specItems*, the easier of the two functions. This function simply takes the union of all bought items in all calls visible to the given action identifier *aId*:

$$\begin{aligned}
&\textit{specItem} : (\textit{executionContext} \times int) \rightarrow \textit{string set} \\
&\textit{specItem}(\textit{ctxt}, \textit{aId}) \equiv \{i : \exists a, c : (a, \textit{BuyItem}(i, c, \textit{True})) \in \ \textit{visibleCalls}_{\textit{ctxt}}(\textit{aId})\}
\end{aligned}$$

By taking only the visible calls into account, we ensure that no synchronization is necessary to implement the specification.

The calculation of the balance is more complicated, because it has to describe the conflict resolution strategy for the case of concurrent buy-actions. The strategy is described by three steps. First the set *as* of successful buy-actions is calculated. Then from this the set $\textit{as}_{min}$ is computed, by only keeping the entry with the lowest cost for each item. Finally the *spendings* are calculated by summing up the costs. The total balance of the account is then just the total deposits minus the total spendings.

$$
\begin{aligned}
&\mathit{specBalance} : (\mathit{executionContext} \times int) \rightarrow int\\
&\mathit{specBalance}(\mathit{ctxt}, \mathit{aId}) \equiv (\\
&\quad \textbf{let } \mathit{deposits} = \Sigma(i, \mathit{Deposit}(x)) \in \mathit{visibleCalls}_{\mathit{ctxt}}(\mathit{aId}) : x\\
&\qquad \mathit{as} = \bigcup(i, \mathit{BuyItem}(\mathit{item}', \mathit{cost}', \mathit{True})) \in \mathit{visibleCalls}_{\mathit{ctxt}}(\mathit{aId}) : \{(i, \mathit{item}', \mathit{cost}')\}\\
&\qquad \mathit{as}_{min} = \{(i, \mathit{item}, \mathit{cost}) \in \mathit{as} : \nexists(i', \mathit{item}, \mathit{cost}') \in \mathit{buyActions} : \mathit{cost}' < \mathit{cost}\}\\
&\qquad \mathit{spendings} = \Sigma(i, \mathit{item}, \mathit{cost}) \in \mathit{as}_{min} : \mathit{cost}\\
&\quad \textbf{in } \mathit{deposits} - \mathit{spendings})
\end{aligned}
$$

Now we have a complete specification of the admissible execution contexts of the example system. However, execution contexts use the visibility relation, which is not observable from the point of view of a user or a client application. To handle this gap between the definition and the actual system, we define an execution to be correct if there is some visibility relation which explains the outcome of the execution. In the literature this relation is called a visibility witness [15]. For most database systems used in practice the visibility relation is strongly related to the mechanism used for communication between replicas. For example in many systems a message sent from one replica to an other adds a new arc to the visibility relation.

5.3.2 Operational specifications

The axiomatic specifications described above are relatively abstract, which makes it hard to relate them with a concrete implementation. In contrast to this, operational specifications are closer to the implementation by describing the system in terms of states and the effects which operations have on the state. This leads to more modular specifications since each operation can be described solely by its interaction with the state, without taking other operations into account. Furthermore, operational specifications are more suitable for most tools. This is why we use operational specifications for our verification work with *TLA+* which we describe in Section 6.

However, a description of the state naturally requires to fix some aspects of the architecture and thus mixes aspects of implementation and specification. In particular if different replicas can have different states, this has to be reflected in the model. Additionally there has to be a mechanism in the model to synchronize the states of the different replicas. For example there are techniques based on transmitting commutative downstream effects [22], also called shadow operations [19, 20] and there are techniques based on merge functions on states [22].

In the following we only describe a technique based on downstream operations. Other techniques can be modeled in a similar way. The basic idea of this technique is to split an operation into two parts: The first part uses the current local state to calculate a message which is sent to every replica. Then the second part is to process this message at every replica. The processing of messages should be implemented such that the calculation commutes with the processing of other messages. When the system maintains this rule and messages are delivered exactly once, this ensures that two replicas which observed the same set of operations also have equivalent states.

We first describe the general framework for this architecture and then the concrete parts required to describe the virtual wallet example. In the general framework the distributed state consists of two parts, namely the local state for each replica and a message inbox for each replica:

$$\begin{aligned}\mathit{distributedState} = (&\mathit{localState} : \mathit{replica} \to \mathit{localState},\\ &\mathit{messages} : \mathit{replica} \to \mathit{message\ list})\end{aligned}$$

On this state a system can perform two kinds of actions. It can process a call from an external source to one replica, or a replica can process one of the internal, commutative messages from its inbox:

$$\begin{aligned}\mathit{action} = &\,\mathit{Call}(r : \mathit{replica}, c : call)\\ &\mid \mathit{ProcessMessage}(r : \mathit{replica},\ \mathit{index} : nat)\end{aligned}$$

We describe the handling of these actions parameterized on two application-specific functions: *concSpec* is a predicate describing the possible effects of a single call on the distributed state and *processMessage* is a function, which processes an internal message and calculates an effect on the local state of a replica. Although we model the inbox as a list of messages, the order is not important as an action can process any message from the inbox at any time. The effect of a call is described by a predicate to allow for incomplete or nondeterministic specifications. In contrast to this *processMessage* is a function and thus must yield a deterministic effect for every message. It would be hard to model the aspect of processing messages as a predicate, because the principle of this kind of architecture states, that the effect of different messages should commute. Now if the effects would be nondeterministic, it would no longer make sense to demand that $f(g(s)) = g(f(s))$ for two effects f and g.

Formally we define the transition relation for a single step and for several steps as follows:

$$\frac{\mathit{concSpec}(r, c, s, s')}{\mathit{step}(\mathit{Call}(r, c), s, s')}$$

$$\frac{\begin{array}{c} i < \mathit{length}(\mathit{messages}(s, r)) \quad \mathit{messages}(s', r) = \mathit{removeAt}(i, \mathit{messages}(s, r)) \\ \mathit{localState}(s', r) = \mathit{processMessage}(\mathit{messages}(s, r)_i, \mathit{localState}(s, r)) \\ \forall r' \in R : (\mathit{localState}(s', r') = \mathit{localState}(s, r') \land \mathit{messages}(s', r') = \mathit{messages}(s, r')) \end{array}}{step(\mathit{ProcessMessage}(r, i), s, s')}$$

$$\frac{}{steps([], s)} \qquad\qquad \frac{step(a, s, s') \qquad steps(as, s')}{steps(a \cdot as, s)}$$

Having defined the general framework for this particular architecture, we can now give the specification for the virtual wallet by defining how the structure of the local replica states and of the messages, and by giving a concrete *concSpec* predicate and a *processMessage* function.

As one would expect, the local state only contains the current balance of the account and a map of all bought items with the corresponding price:

$$\mathit{localState} = (\mathit{balance} : int,\ \mathit{items} : \mathit{string} \rightharpoonup int)$$

The internal messages used are very similar to the signature of the external calls. There is one message for deposits and one message for buying items:

$$\begin{aligned} \mathit{message} = {} & \mathit{MsgDeposit}(\mathit{change} : int) \\ & \mid \mathit{MsgBuyItem}(\mathit{item} : \mathit{string},\ \mathit{cost} : int) \end{aligned}$$

The code in the *concSpec* predicate describes, how the downstream message is calculated from the current state and the given parameters. This message is then used in the helper function *downStream*, which simply applies the effect of the message locally and sends the message to all other replicas. For a call to *Deposit* we simply produce an equivalent downstream message. For calls to *BuyItem* we have to add some checks. The balance of the account has to be high enough and the item must not be bought already. Note that we only check these conditions on the local state, so here we allow some inconsistencies. In particular the specification allows the balance to become negative. One could add additional invariants as described in Section 5.5 to handle this problem. The specification is also nondeterministic or incomplete, leaving the implementation a choice about when to accept *BuyItem* requests.

$$\begin{aligned}
&\mathit{concSpec} : (\mathit{replica} \times \mathit{call} \times \mathit{distributedState} \times \mathit{distributedState}) \to bool \\
&\mathit{concSpec}(r, \mathit{Deposit}(a), s, s') \equiv \mathit{downStream}(r, \mathit{MsgDeposit}(a), s, s') \\
&\mathit{concSpec}(r, \mathit{BuyItem}(i, c, suc), s, s') \equiv \\
&\quad (\textbf{if } suc \textbf{ then} \\
&\qquad \mathit{balance}(\mathit{localState}(s, r)) \geq c \\
&\qquad \wedge\ \mathit{items}(\mathit{localState}(s, r))i = \mathit{None} \\
&\qquad \wedge\ \mathit{downStream}(r, \mathit{MsgBuyItem}(i, c), s, s') \\
&\quad \textbf{else } s' = s) \\
&\mathit{concSpec}(r, \mathit{GetItems}(is), s, s') \equiv (s' = s \wedge is = dom(\mathit{items}(\mathit{localState}(s, r)))) \\
&\mathit{concSpec}(r, \mathit{GetBalance}(b), s, s') \equiv (s' = s \wedge b = \mathit{balance}(\mathit{localState}(s, r)))
\end{aligned}$$

$$\begin{aligned}
&\mathit{downStream} : (\mathit{replica} \times \mathit{message} \times \mathit{distributedState} \times \mathit{distributedState}) \to bool \\
&\mathit{downStream}(r, msg, s, s') \equiv (\\
&\qquad \mathit{localState}(s', r) = \mathit{processMessage}(msg, \mathit{localState}(s, r)) \\
&\qquad \wedge\ \mathit{messages}(s', r) = \mathit{messages}(s, r) \\
&\qquad \wedge\ (\forall r'. r' \neq r \longrightarrow \mathit{messages}(s', r') = \mathit{messages}(s, r') \cdot msg \\
&\qquad\qquad\qquad \wedge\ \mathit{localState}(s', r') = \mathit{localState}(s, r')))
\end{aligned}$$

The processing of a *Deposit* message is straight forward. The balance is just changed by the amount given in the message. Processing a *BuyItem* message is more involved because conflicts have to be handled. In this specification we handle conflicts by checking if the item was already bought previously. If the previous purchase was more expensive, we refund the old price and subtract the cheaper price. If the previous purchase was cheaper, we do not change the state. And if there is no previous purchase, we just update the balance and the map of items accordingly.

$$\begin{aligned}
&\mathit{processMessage} : (\mathit{message} \times \mathit{localState}) \rightarrow \mathit{localState}\\
&\mathit{processMessage}(\mathit{MsgDeposit}(x), s) = s[\mathit{balance} := \mathit{balances} + x]\\
&\mathit{processMessage}(\mathit{MsgBuyItem}(i, c), s) = (\textbf{case } \mathit{items}(s, i) \textbf{ of}\\
&\quad \mathit{None} \Rightarrow s[\mathit{balance} := \mathit{balance}(s) - c, \mathit{items} := \mathit{items}(s)[i \mapsto c]]\\
&\quad | \ \mathit{Somec}' \Rightarrow \textbf{if } c' < c \textbf{ then } s\\
&\qquad \textbf{else } s[\mathit{balance} := \mathit{balances} - c' + c, \mathit{items} := \mathit{items}(s)[i \mapsto c]])
\end{aligned}$$

5.4 Relating sequential and concurrent specifications

In Section 5.2 we showed techniques for sequential specifications and in Section 5.3 we have seen two kinds of concurrent specifications. The obvious question now is how these two specifications are related. This question also arises in practice. Often the sequential behavior is kind of clear, but the behavior in the case of concurrent updates is not. In this case the question also is how to get from a sequential specification to a concurrent one. In this section we list some known relations between the two.

Sequential consistency. Well known principles for defining the allowed behaviors of concurrent data types are sequential consistency [17] or linearizability [16]. These principles demand that each concurrent execution must be explainable by some sequential ordering of events to which the sequential specification can be applied. The allowed sequential orderings are usually restricted by some rules. For example the ordering must be consistent with the program order.

However, the notion of sequential consistency is usually too strong. For example it would not allow the behavior in our virtual wallet example, where clients can see an outdated view of their balance.

Sequential delayed consistency. One way to relax the above notion of sequential consistency is to allow some staleness, meaning that each action must be explainable by some sequential ordering of only a subset of previous actions. This is similar to the criterion of update consistency [21] and more flexible than sequential consistency. In the virtual wallet example it permits, that the queries return outdated values and that the same item can be bought more than once in certain situations. There are two interesting variations of this principle, which differ in how the subset of previous actions can be selected. One variation is to always use the set of visible updates as the subset, i.e. updates which have already been sent to the replica performing the action in question. In particular this implies that no updates can be ignored forever, and thus our behavior of the virtual wallet would not be valid, since it ignores purchases when a cheaper purchase of the same item has happened in parallel. So an alternative is to allow the selection of a subset according to an arbitrary algorithm. In the case of the virtual wallet example this algorithm would be to take all the visible updates and then remove all purchases for which a cheaper purchase exists.

The more flexible variant of this principle can cover many scenarios from the real world, but not all reasonable concurrent behaviors are covered by it. For example consider a version control system like git. In such systems a merge conflict can never be explained by a sequential ordering of actions. Merge conflicts can only happen when there are concurrent updates to the same data. But even in a system like git, the behavior should be somehow related to how a comparable sequential system would behave. This leads us to the next principle.

Permutation equivalence. The principle of permutation equivalence is a consistency criterion which only affects a subset of the possible concurrent executions: If all possible sequentialized executions of a given concurrent execution result in equivalent states, then the concurrent execution should result in the same state. For executions where a different order of operations produces a different result, the principle of permutation equivalence does not give any guidance. In the example of the version control system the principle demands that concurrent changes to different files will behave as if they are done in some sequential order. For the case of conflicting updates to the same file the order of operations is important, and so the principle of permutation equivalence leaves it to developer to choose a strategy to handle conflicts. Systems like git try to merge changes automatically and if this is not possible it lets the user resolve the conflicts.

5.5 Assertions and invariants

Invariants and assertions can be viewed from two different perspectives, namely users and developers. A user has certain requirements that she can express via invariants or assertions. For example in the virtual wallet application a user expects the invariant, that his balance is never negative. And if a user successfully buys an item, he asserts that the item is in his inventory afterwards.

Developers have to write code which maintains the user expectations, but additionally developers also use invariants and assertions to reason about the correctness of code. This can be done explicitly by writing down the assertions, or just implicitly by thinking about the code. Furthermore it can be done formally with dynamic or static checking supported by a tool, or just informally.

For all cases matters get more difficult when the state is distributed, shared and only eventually consistent. Invariants and assertions are based on states, so if there is no central state the question is which state the assertions and invariants refer to.

One possibility to handle the distributed state with invariants is the notion of invariant confluence ($\mathcal{I}$-confluence) [23]. Intuitively, this notion requires that there is a merge function on local states, such that merging two states again results in a state satisfying the invariant, when the two states can be reached via sequences of invariant preserving operations. A definition which is easier to handle is the notion of $\mathcal{I}$-confluent global states. We use this notion in our work on explicitly consistency, which we present in Section 7. A global state S is defined to be $\mathcal{I}$-confluent if the invariant holds for every possible combination of merged local states S_r:

$$\mathcal{I}\text{-}\mathit{confluent}(S) \Leftrightarrow \forall R \subseteq \mathit{Replias} : \mathit{Inv}(\bigsqcup_{r \in R} S_r)$$

With this definition it is possible to define meaningful invariants referring to the global state. However, this notion cannot be easily transferred to other kinds of assertions like pre- and post-conditions of operations or assertions about intermediate states inside a method. Therefore in assertions we are currently restricted to refer to the local state or to explicitly mention replicas as we did for the operational specifications in Section 5.3.2.

5.6 Future work

Our current work with respect to specifications was focused on specifying and verifying replicated data types [24]. The techniques for specifying data types are strongly related to the techniques for describing whole applications, since both have to tackle the same problem of concurrent updates. With respect to the specification of applications, we have specified some variations of use cases from WP1 to Isabelle. We plan to use these specifications as the basis for future work on verification, tools, and as examples to evaluate the programming model. Formal specifications are especially important when it comes to tool support, because the intent of code can in general not be extracted from source code.

In an ideal world, programmers would just write down a specification of a program and hit one button to compile it to executable code. In practice developers hardly write down formal specifications, because it is hard to do so, the benefits are not worth it, and there is a certain gap between specifications and implementations. In future work we plan to improve on the first two points by developing tools and techniques, which help programmers to write specification, and which make specifications more useful. For example this can be tools for verification or for dynamic testing. Regarding the third point, which is to close the gap between specifications and implementations, we think that the approach of deterministic dataflow programming will make this gap narrower. The deterministic aspect of the language makes it a lot easier to reason about the relation between specification and implementation, since the number of possible executions on the implementation side is reduced to a minimum and common patterns can be expressed in the language in a more natural way. Today, programmers already use programming patterns like monotonicity, immutability, and idempotence but with little support from a programming language or framework. Often there are a lot of patterns used in the code, but they are not explicitly visible. This makes it hard to build tools which work on code written like this and to see the relation to a specification. In our future work we plan to improve on this situation by bringing specification and implementation closer together. In particular we want to investigate, how patterns in natural language descriptions map to patterns in formal specifications and then how these map to patterns in the implementation.

6 Verification

6.1 Overview

Verification research during the first year of SyncFree had as its focus verifying and ensuring application-level properties for applications built using CRDTs. We believe this to be the foremost priority, as the key research question being explored by the SyncFree project is the determination of the extent to which CRDTs can enable the building of practical systems. As such, much effort was dedicated to exploration of specifying (as detailed in the previous section), verifying and enforcing properties desired at the application level. Below, we provide overviews on these two lines of work: model-checking based verification of applications, and ensuring application invariants using reservations and escrow.

For specifying and verifying programs built on CRDTs, we focused on high-level mathematical modeling and verification of our industrial case studies. We carried out a formal modeling of the applications in our case studies, as well as the state- and operation-based CRDTs they make use of, using the TLA+ specification language. This approach had the significant benefit of decoupling the issues around programming languages and the CRDT and geo-replication platforms from the fundamental issue pursued in this study – whether CRDT-based application designs can accomplish the desired application-level properties, and whether one can determine application correctness and carry out iterative design with adequate verification tool support. The joint experience of industrial and academic partners (Trifork, Universidade Nova de Lisboa, and Koç University) is affirmative. Using the TLA+ language and the TLC model checker that supports TLA+ specifications, we are able to precisely model state- and operation-based CRDTs, applications that build on them, natural specifications regarding convergence and application invariants. We are further able to detect violations of application invariants, explore error traces, modify the applications and iterate the verification process. So far, this effort has concentrated on modeling applications that use transactions and CRDTs, but not more elaborate mechanisms for ensuring invariants such as reservations and escrow, and not more elaborate, more efficient replication mechanisms such as partial or adaptive replication. Since the initial exploration of the verification and tooling issue has yielded encouraging results, we plan to investigate these more elaborate mechanisms, their correctness and how they enable programmers to guarantee application-wide invariants.

As alluded to above, ensuring invariants for CRDT-based applications without sacrificing availability and responsiveness, and avoiding synchronization when possible is a challenging problem. Eventually consistent transactions may not be sufficient for applications such as e-wallets, where rather strong application-level invariants are desired. The approach proposed can be viewed as an alternative consistency model, explicit consistency, that strengthens eventual consistency with a guarantee to preserve specific invariants defined by the applications. Given these application-specific invariants, we identify which operations are potentially unsafe under concurrent execution by different replicas. The programmer then has the options of avoiding invariant violations or restoring ("repairing") the invariant. If avoiding invariant violations is targeted, we provide techniqued for allowing most of operations to still complete locally and remain responsive by relying on a reservation system that moves replica coordination off the critical path of operation execution. If, instead, invariant repair is targeted, operations are allowed to execute without restriction, and later repair operations restore invariants fixing the database state. The product of this line of research is the Indigo system, a middleware that provides explicit consistency on top of a causally-consistent data store. Indigo guarantees strong application invariants while providing latency similar to an eventually consistent system. This is explained further in Section 7 of this report.

In the rest of this section, we provide a more detailed account of the research along the two lines outlined above.

6.2 Describing CRDTs in TLA+

To write high-level models for our industrial use cases, we used the TLA+ specification language [29]. TLA+ is a formal language that enables description and reasoning about distributed and concurrent systems. It is based on mathematical logic, set theory and temporal logic. A system is described in TLA+ using quantified first-order formulae and set theoretical constructs. Properties or the invariants are described in terms of first-order and temporal logic formulae. By using the tools of TLA+, one can reason about the system by either trying to prove that specified properties are satisfied by the formal model (by using the proof manager TLAPS) or by trying to find a counter-example in the model that violates a certain property (by using the model checker TLC). Since the TLA+ language is purely mathematical and is not based on any programming language, systems can be described at the desired abstraction level in TLA+ separately from implementation of the system in a particular programming language and on a particular platform. For these reasons above, TLA+ is a widely known and used language in both academia ([25–27]) and industry ([28]). We modelled and analysed CRDTs and industrial applications using TLA+ for the very same reasons.

In our industrial use cases, both operation-based (op-based) and state-based (st-based) CRDTs are utilized. Thanks to the modular structure of TLA+, we can define CRDTs as a separate module and use them in the application specifications. In this section, we highlight parts of the specification efforts for one st-based and one op-based CRDT in order to highlight the simplicity and implementation-independence of TLA+ specifications.

An Op-based G-Counter: Following Shapiro et al. ([31]) we model an op-based CRDT using four components: a payload, initialization, query and update operations. We follow this definition to describe an op-based G-counter in TLA+:

```
GCounter == Nat
Message == [src: Replicas, op: {"inc"}]
MQueue == [Replicas -> Seq(Message)]

InitGC == 0
InitMQ == [r \in Replicas |-> << >> ]

EvalGC(gc) == gc

IncGC(mq,gc,rep) ==
LET new_msg == [src |-> rep, op |-> "inc"]
IN  <<gc+1,
13   [r \in Replicas |-> IF r = rep THEN mq[rep]
14     ELSE Append(new_msg, mq[r])]>>
15
16 ProcessMsg(mq,gc,rep) ==
17  IF mq[rep] = << >> THEN << >> ELSE
18   <<gc+1,[mq EXCEPT ! [rep] = Tail(mq[rep]) ] >>
```

A G-Counter keeps a natural number as the payload (line 1). In addition to the payload, we model messages and input message queues of replicas. Message objects contain two fields `src` and `op` (line 2). `src` field shows the source replica that sends the message. `op` field is kept for the type of operation and increment is the only operation for the G-Counter. `MQueue` keeps a sequence of messages for each replica.

Initial values for G-Counter and message queues are assigned in lines 5,6. G-Counter's payload is initialized to zero and input message queue of each replica is initialized to the empty sequence. The query action in the G-Counter contains only evaluation of the G-counter – `EvalGC` returns the value of payload (line 8). The only state update action is the increment operation (lines 10-14). `IncGC` takes three inputs `mq, gc, rep` that are supposed to be a message queue, a G-counter and a replica, respectively. `IncGC` forms new message named `new_msg`. It returns a tuple of two elements. The first element is the incremented value of the G-Counter `gc` and the second element contains the updated version of the message queues `mq` so that `new_msg` is appended to the message queue of every replica except `rep`'s which stays unchanged.

These definitions complete the description of G-Counter. However, we define one more helper operation for processing messages in the queue (lines 16-18). If the message queue of `rep` is empty `ProcessMsg` returns an empty sequence. Otherwise, it processes a message by removing it from the `rep`'s queue and incrementing `gc` by one.

State-based PN-Counter A st-based CRDT can be defined in terms of six components ([31]): payload, initial, query, update, compare and merge. We follow this definition to describe a st-based G-counter in TLA+. We define this data structure in TLA+ as follows:

```
1  PNCounter == [p: [Replicas -> Nat],
2        n: [Replicas -> Nat] ]
3
4  InitPN == [p |-> [r \in Replicas |-> 0],
5   n |-> [r \in Replicas |-> 0] ]
6
7  EvalPNCounter(pnc) == SumAll(pnc.p) - SumAll(pnc.n)
8
9  IncrementPN(pnc,rep) == [pnc EXCEPT !.p[rep] = pnc[rep]+1]
10 DecrementPN(pnc,rep) == [pnc EXCEPT !.n[rep] = pnc[rep]+1]
11
12 ComparePN(pnc1,pnc2) ==
13 IF (\A r \in Replicas: pnc1.p[r] <= pnc2.p[r] /\ pnc1.n[r] <= pnc2.n[r])
14 THEN TRUE ELSE FALSE
15
16 MergePN(pnc1, pnc2) ==
17 [p |-> [r \in Replicas |-> Max(pnc1.p[r], pnc2.p[r])],
18  n|-> [r \in Replicas |-> Max(pnc1.n[r], pnc2.n[r])]]
```

p and n keep the positive and negative updates, respectively. In TLA+, p and n are defined as maps from the Replicas set to the natural numbers (lines $1, 2$). They can be conceived as arrays indices of which are replicas. The initialization of a PN-Counter is done by a predicate which assigns 0 to all elements in both fields (lines 4,5). The query operation consists of evaluation of a PN-Counter. The value of a PN-Counter is the difference between sum of all the elements in the positive part and the negative part (line 7). The update operation has two operations: increment and decrement (lines $9, 10$). IncrementPN (DecrementPN) operation increments (decrements) value at the index rep of the p (n) map of the PN-Counter. Comparison of two PN-Counters is based on the underlying natural partial order defined on them (lines $12 - 14$). Two PN-Counters are merged by taking pairwise maximum of elements for all replicas in both p and n maps (lines 16-18).

6.3 Describing applications in TLA+

We formalized the Leader Board, Advertisement Counter and Virtual Wallet industrial use cases in TLA+. In this section, we give two examples of applications: Advertisement Counter and Virtual Wallet. The first example utilizes the op-based G-Counter and the second one utilizes the st-based PN-Counter.

6.3.1 The Advertisement Counter application

Formal Model

```
---------------- MODULE adCounterOp ------------------
EXTENDS Naturals, Sequences, TLC, GCounter
CONSTANTS Replicas, Ads, ...
VARIABLE adViews
-------------------------------------------------------

State == [mq: [Ads->MQueue],
         views: [Ads-> [Replicas-> GCounter]],
          ]

TypeInv == adViews \in State

Init == ...

Increment(ad,rep) == ...
Process(ad,rep) == ...

Next == \E a \in Ads, r \in Replicas:
(Increment(a,r) \/ Process(a,r))

Spec == Init /\ [] [Next]_<<adViews>>
=======================================================
```

In the first line, we give the external modules that are utilized inside `AdCounter`. All modules are the built-in TLA+ modules except the `GCounter`. In the second line, we introduce `Replicas` and `Ads` as the rigid variables which are initially fixed to some value and can not be modified during the state transitions. The third line introduces the flexible variable `adViews` which will be modified by the state transitions. TLA+ does not put any type restriction on the flexible variables if there is no invariant defined on them explicitly. Hence, we introduce the type invariant (line 10) that enforces `AdViews` to be an element of `State` set. A state (lines $6-9$) contains a message queue for every ad (`mq`) and a G-Counter for every ad in each replica (`views`). The module contains `Init` predicate (line 12) to initialize each field of `AdViews` to a default value by utilizing the `InitGC` and `InitMQ` operations.

Lines $14-15$ introduce two operators that change the state of the `adViews` variable. `Increment` operation increments the value of the G-Counter in the replica `rep` for advertisement `ad` by one. In addition, it appends a new message to the message queues of the other replicas for advertisement `ad` by modifying `adViews.mq[ad]` field. For those updates, `IncGC` operation provided by `GCounter` module is utilized. `Process` also takes the same parameters as `Increment`. This operation removes a message from the message queue `adViews.mq[ad][rep]` for advertisement `ad` in replica `rep`. To process the message, it increments its local G-Counter `adViews.views[ad][rep]` by one. `Process` operation utilizes `ProcessMsg` operation provided by the `GCounter` module for these updates. The next state predicate (lines 17,18) states that the next state of the system is obtained by applying either an `Increment` or a `Process` operation with arbitrary valid arguments. Next state operation is non-deterministic, i.e., next state can be obtained by picking one of these operations and its arguments arbitrarily.

Program specification predicate (line 20) states that initially the `Init` predicate holds and the system takes a step every time by applying `Next` predicate on `adViews` variable.

Invariants We introduced three invariants to hold in every state. First invariant to check is the `TypeInv` defined in line 10 of `adCounterOp` module. Other invariants are:

- **Convergence** Eventual Consistency guarantees that each update is eventually delivered to each replica. Hence, after the updates finish and each replica processes messages in its message queue, replicas must converge to the same state. `Convergence` predicate states this constraint as a first-order formula.
- **No Clicks Lost** invariant states that total value of number of local increments, processed messages and unprocessed messages in the input queue must be equal for any two replicas.

6.3.2 The Virtual Wallet application

Formal Model

```
------------ MODULE walletv4 ------------
1  EXTENDS Naturals, Sequences, TLC, PNCounter
2  CONSTANTS Replicas, V1Cost, InitBal, Qtylim, ...
3  VARIABLES wallets
```

```
4  -----------------------------------------
5  Wallet == [balance: PNCounter,
6            v1cnt: PNCounter,
7            vecclc: [Replicas ->Nat] ]
8
9  TypeInv == /\ wallets \in [Replicas->Seq(Wallet)]
10
11 Init == ...
12
13 BuyVoucherOne(rep,qty) ==
14 LET wr == Head(wallets[rep])
15 ...
16 IN /\ EvalPNCounter(wr.balance)+ InitBal >= qty*V1Cost
17 ....
18 MergeLastStates(rep1,rep2) ==
19 LET wr1 == Head(wallets[rep1])
20 wr2 == Head(wallets[rep2])
21 ...
22 IN  \E r \in Replicas: wr1.vecclc[r] < wr2.vecclc[r]
23 ...
24
25 Next == \E r1 \in Replicas, r2 \in Replicas, qty \in 1..Qtylim:
26   (BuyVoucherOne(r1,qty) \/ MergeLastStates(r1,r2))
27
28 Spec == Init /\ [] [Next]_<<wallets>>
29 ==========================================
```

For simplicity, we assume that there is only one client that has the virtual wallet and there is only one type of voucher that could be purchased. This module utilizes `PNCounter` module (line 1). `Replicas` is the set of replicas, `V1Cost` is the cost of purchasing one amount of voucher one,`InitBal` is the initial balance of the client and `Qtylim` is the maximum amount of voucher one that can be bought at once (line 2).

The only flexible variable in this description is `wallets`. A type invariant on `wallets` (line 9) checks `wallets` is a map from `Replicas` to a sequence of `Wallet` objects. This invariant informally states that `wallets` keeps the history of the client's wallet for each replica. A member of `Wallet` contains three fields (lines $5-7$): `balance` is a PN-Counter keeping the changes in the client's balance, `v1cnt` is a PN-Counter keeping the available amount of voucher one in the wallet and `vecclc` is a ghost map vector clock that is used for specifying some invariants among and inside replicas.

There are two operations that change the state of the `wallets` variable. `BuyVoucherOne` operator (lines 13-17) has a precondition (line 16) stating that the client's latest balance must be enough for purchasing `qty` amount of voucher 1. If this condition is satisfied, `BuyVoucherOne` operator decrements the `balance` PN-Counter of the client's wallet at replica `rep` by `qty`×`V1Cost` amount utilizing `PNCounter` module's decrement operator, increments `v1cnt` PN-Counter of the client's wallet at replica `rep` by `qty` amount utilizing `PNCounter` module's increment operator and increments value of the `wallets[rep].vecclc[rep]` by one. Then, it appends the newly formed object into `wallets[rep]` sequence.

`MergeLastStates` operator (lines $18 - 23$) models the communication between replicas. More specifically, it simulates the situation in which `rep2` sends its entire state to `rep1` and `rep1` merges its state with `rep2`. The precondition (line 22) informally states that `rep2` must contain more recent information about at least one of the other replicas than `rep1`. If this precondition is satisfied, a new `Wallet` is formed by merging the `balance` and `v1cnt` fields of `rep1` and `rep2` by utilizing `PNCounter` module's `mergePN` operators and similarly merging the vector clocks. Then, this new wallet is appended into `wallets[rep1]` sequence. `Next` and `Spec` predicates are defined similar to the Advertisement Counter module. `Next` predicate states that the next state can be obtained non-deterministically by calling `BuyVoucherOne` or `MergeLastStates` operators with arbitrary valid inputs.

Invariants We introduced five invariants to be checked in every state for the formal virtual wallet specification. The first one is the `TypeInv` invariant (line 9). Apart from that we introduced two invariants about CRDT properties and two application specific invariants.

- **Convergence** We implicitly limit the number of `BuyVoucherOne` operations for one replica by bounding the natural numbers by `Natlim` from above and defining elements of vector clocks as natural numbers. Then, local updates of replica `rep` must stop after `Natlim` number of `BuyVoucherOne` operations and we expect last states of the wallets in two replicas two converge eventually in `Convergence`.
- **Monotonicity** states that PN-Counters and vector clocks in each replica are non-decreasing.
- **Positive Balance** states that the current balance of the client is always non-negative in each replica.
- **Conservation of Money** states that amount of money spent for buying voucher according to a replica must be equal to the change in balance.

6.4 Verification using TLA+ model checking

We analyzed both application specifications using the TLA+ model checker tool TLC [30]. The configuration settings we used to make models finite and the results of model checking for each example are presented in this section.

Advertisement Counter Results We used the following configuration parameters in order to make the set of states finite:

- `Replicas = {r1,r2,r3}`,
- `Ads = {ad1,ad2}`, and
- `Natlim = 2`. We also restricted natural numbers to be set between 0 and $3\times$`Natlim`.

TLC generated 208283 distinct states and performed a search of depth 19. In 12 seconds it verified that no state violates the provided 3 invariants.

Virtual Wallet Results We specified constants as: `Replicas = {r1,r2,r3}`, `Qtylim = 2`, `Natlim = 2`, `V1Cost = 1` and `InitBal = 2`. We also restricted natural numbers to be set between 0 and `Natlim`. TLC generated 66649 distinct states and performed a search of depth 16 in 12 seconds. TLC found an example violating the `posBalance` invariant although `BuyVoucherOne` never allows balance to reduce into negative amounts. The counter example found can be summarized as follows:

```
st1: [r1: [balance: 2,...], r2: [balance: 2,...], ... ]
BuyVoucherOne(r1,2)
st2: [r1: [balance: 0,...], r2: [balance: 2,...], ... ]
BuyVoucherOne(r2,2)
st3: [r1: [balance: 0,...], r2: [balance: 0,...], ... ]
MergeLastStates(r1,r2)
st4: [r1: [balance: -2,...], r2: [balance: 0,...], ... ]
```

Variant scenario 1: In addition to the existing model, we implemented two alternative scenarios to analyze effects of CRDTs and atomic transactions for the wallet application. In the first scenario, we replaced `balance` and `v1cnt` PN-Counters with integers. We modified `MergeLastStates` operation so that merged state contains the maximum of the `balance` fields of two replicas as the `balance`and the maximum of the `v1cnt` fields of two replicas as the `v1cnt`. For this scenario, `posBalance` was not violated due to the merging policy but `ConservationOfMoney` is violated with the following error trace:

```
st1: [r1: [balance: 2, v1cnt: 0], r2: [balance: 2, v1cnt: 0], ...]
BuyVoucherOne(r1,1)
st2: [r1: [balance: 1, v1cnt: 1], r2: [balance: 2, v1cnt: 0], ... ]
BuyVoucherOne(r2,2)
st3: [r1: [balance: 1, v1cnt: 1], r2: [balance: 0, v1cnt: 2], ... ]
MergeLastStates(rep1,rep2)
st4: [r1: [balance: 1, v1cnt: 2], r2: [balance: 0, v1cnt: 2], ... ]
```

Variant scenario 2: In the second scenario, we analysed the effect of atomic updates. Since `MergeLastStates` is the only operation providing communication among replicas, we divided it into two decoupled merges to remove atomicity. In each step, `Next` predicate was allowed to choose `BuyVoucherOne`, `MergeBalance` or `MergeV1Cnt` operations non-deterministically. For this scenario, both `posBalance` and `ConservationOfMoney` invariants were violated. TLC produced example for `posBalance`:

```
st1: [r1: [balance: 2,...], r2: [balance: 2,...], ... ]
BuyVoucherOne(r1,2)
st2: [r1: [balance: 0,...], r2: [balance: 2,...], ... ]
BuyVoucherOne(r2,2)
st3: [r1: [balance: 0,...], r2: [balance: 0,...], ... ]
MergeBalance(r1,r2)
st4: [r1: [balance: -2,...], r2: [balance: 0,...], ... ]
```

and for `ConservationOfMoney`:

```
st1: [r1: [balance: 2, v1cnt: 0], r2: [balance: 2, v1cnt: 0],... ]
BuyVoucherOne(r1,1)
st2: [r1: [balance: 1, v1cnt: 1], r2: [balance: 2, v1cnt: 0],... ]
MergeV1Cnt(r2,r1)
st3: [r1: [balance: 1, v1cnt: 1], r2: [balance: 1, v1cnt: 0],... ]
```

6.5 Conclusions

Our experience with specifying our industrial use cases using TLA+ and verifying small configurations using the TLC model checker has led us to believe that formal specifications that can be parsed and automatically analyzed are very valuable. They constitute the next level of formal specification after the text-based specifications we have devised in WP1 and are useful means for capturing mathematically the underpinnings of both the applications and the operational semantics of the particular CRDTs they are built on.

Furthermore, TLA+ specifications at this level allow efficient exploration of the application state space at a higher level of abstraction. This is because we do not model actual, optimized and therefore complicated state spaces of CRDTs as would have been the case had we taken CRDTs implemented in a programming language as the basis for our formal model. In the next year of SyncFree, we will investigate the formal refinement (correct implementation) relation between a high-level CRDT specification and a lower-level implementation.

6.6 Future work

In the next year of SyncFree, we plan to pursue verification research along the following lines:

- On the high-level modeling and verification using TLA+ front, we plan to pursue static verification (rather than model checking) of application properties. While more intensive in terms of human effort, this approach will have the conceptual benefit of verifying properties for arbitrary configurations, i.e., number of replicas, replicated objects, and arbitrary number of operations and transactions performed during an execution. This will necessitate us to devise and verify global invariants that hold in non-convergent states as well. We also plan to verify application correctness when using explicit consistency, and partial and adaptive replication policies. We intend to explore whether correct-by-construction application code can be obtained from these verified TLA+-level descriptions.
- We have started the design of a dynamic behavior exploration tool for applicaitons built on top of the execution platforms developed in SyncFree, using Riak. We are working on building a tool that orchestrates multiple Riak instances running on a single machine in order to systematically explore the states different replicas can get into, and how this affects application-level properties.

7 Ensuring invariants with explicit consistency

7.1 Overview

Applications built using CRDTs have the potential of allowing low latency replies to user requests that are routed to a closeby datacenter. In return, they have to deal with concurrent operations executing without being aware of each other, which leads to potentially unintuitive and undesirable user-perceived semantics.

As part of the SyncFree work, we have proposed and investigated explicit consistency as an alternative consistency model, in which applications define the consistency rules that the system must maintain as a set of invariants. Unlike other consistency models where consistency is defined solely based on the enforced execution order for operations, we define explicit consistency in terms of application properties or invariants that the system must enforce. These invariants can encode not only rules over the database state but also over state transitions. Furthermore, we show how we can enforce explicit consistency for any type of invariant written in first-order logic, as is the case in our TLA+ descriptions for our industrial use cases. In most cases, we avoid any cross-datacenter coordination to execute user operations, even for many of the critical operations that potentially break invariants.

We investigated a methodology that, starting from the set of application invariants, defines suitable invariant violation avoidance solutions specific to different types of invariants (or, alternatively, a set of invariant repair solutions that recover the service to a desired state). These solutions are then deployed on top of a geo-replicated storage system. First, based on static analysis, we infer which operations can be safely executed without coordination. Second, for those operations that cannot be safely executed concurrently, we provide the choice of addressing the problems that can occur by relying on automatic repair or avoidance techniques. Our avoidance techniques combine and extend solutions adopted in escrow and reservation-based systems. Such solutions minimize coordination by moving the required coordination outside the critical path of operation execution and by amortizing the cost of coordination over multiple requests. We additionally run an optimization process that minimizes the required coordination operations based on the frequency of operation execution.

We present the design of Indigo, a middleware that enforces explicit consistency on top of a geo-replicated key-value store. Our solution requires only properties that have been shown to be efficient to implement, such as per-key linearizability for replicas in each datacenter, causal consistency and transactions with weak semantics.

7.2 Formalization approach

In this section we define precisely the consistency guarantees that Indigo provides. To explain these, we start by defining the system model, and then how explicit consistency restricts the set of behaviors allowed by the model.

To illustrate the concepts, we use as running example the management of tournaments in a distributed multi-player game. The game maintains information about players and tournaments. Players can register and de-register from the game. Players compete in tournaments, for which they can enroll and disenroll. A set of matches occurs for each tournament. A tournament has a maximum capacity. In some cases – e.g., when there are not enough participants – a tournament can be canceled before it starts. Otherwise a tournament's lifecycle is creation, start, and end.

We consider a database composed of a set of objects in a typical cloud deployment, where data is fully replicated in multiple datacenters, and partitioned inside each datacenter. For simplicity we assume that the goal of replication is performance, and not fault tolerance. As such, we can assume that replicas do not fail. However, it would be straightforward to handle faults by replacing each machine at a given datacenter with a replica group running a protocol like Paxos [32].

Applications access and modify the database by issuing high-level operations. These operations include a sequence of *read* and *write* operations enclosed in transactions. We define a database snapshot as the value of the database after executing the writes of a sequence of transactions in the initial database state. The state of a replica is the database snapshot that results from executing all committed transactions received in the replica - both local and remote. An application submits a transaction in a replica, with reads and writes executing in a private copy of the replica state. The application may decide to commit or abort the transaction. In the former case, writes are immediately applied in the local replica and asynchronously propagated to remote replicas. In the latter case, the transaction has no side-effect. The snapshot set of a database snapshot is the set of transactions used for computing it. The happens-before relation defines a partial order among transactions. We say a serialization is valid if it is a linear extension of this order.

Our approach allows transactions to execute concurrently. Each replica can execute transactions according to a different valid serialization. We assume the system guarantees state convergence, i.e., for a given set of transactions, all of its valid serializations lead to the same database state. Different techniques can be used to this end, from a simple *last-writer-wins* strategy to more complex approaches based on conflict-free replicated data types (CRDTs) [33, 34].

We now define *explicit consistency*, a novel consistency semantics for replicated systems. The high level idea is to let programmers define the application-specific correctness rules that should be met at all times. These rules are defined as invariants over the database state.

7.3 Illustrating example: a tournament application

In our tournament application, one invariant states that the cardinality of the set of enrolled players in a tournament cannot exceed its capacity. Another invariant is that the enrollment relation must bind players and tournaments that exist - this type of invariant is known as referential integrity in databases. Even if invariants are checked when an operation is executed, in the presence of concurrent operations these invariants can be broken – e.g., if two replicas concurrently enroll players to the same tournament, and the merge function takes the union of the two sets of participants, the capacity of the tournament can be exceeded.

Specifying restrictions over the state: To define explicit consistency, we use first-order logic for specifying invariants as conditions over the state of database. For example, for specifying that the enrollment relation must bind players and tournaments that exist, we could define three predicates: $player(P)$, $tournament(T)$ and $enrolled(P, T)$ to specify that a player P exists, a tournament T exists and that player P is enrolled in tournament T respectively. The condition would then be specified by the following formula:

$\forall P, T, enrolled(P, T) \Rightarrow player(P) \wedge tournament(T)$.

Specifying rules over state transitions: In addition to conditions over the current state, we support some forms of temporal specifications by specifying restrictions over state transitions. In our example, we can specify, for instance, that players cannot enroll or drop from a tournament between the start and the end of the tournament.

Such temporal specification can be turned into an invariant defined over the state of the database, by having the application store information that allows for such verification. In our example, when a tournament starts the application can store the list of participants for later checking against the list of enrollments. The rules that forbids enrollment/disenrollment of players can then be specified as $\forall P, T, participant(P, T) \Leftrightarrow enrolled(P, T)$, with the new predicate $participant(P, T)$ specifying that player P participates in active tournament T.

The alternative to this approach would have been to use temporal logics that can specify rules over time [35, 36]. Such approaches would require more complex specification for programmers and a more complex analysis. As our experience has shown that this simpler approach was sufficient for specifying most common application invariants, we have decided to rely on this approach.

Correctness conditions We can now formally define explicit consistency, starting with the helper definition of an invariant I as a logical condition applied over the state of the database.

For a given set of transactions we say that a linear order on these transactions is a I-valid serialization of iff it is valid serialization and I holds in the state that results from executing any prefix of the linear order. A system is correct, providing explicit consistency, iff all serializations of its executions are I-valid serializations.

7.4 Enforcing explicit concurrency

Given the invariants expressed by the programmer, our approach for enforcing explicit consistency has three steps: (i) detect the sets of operations that may lead to invariant violation when executed concurrently (we call these sets *I-offender sets*); (ii) select an efficient mechanism for handling *I-offender sets*; (iii) instrument the application code to use the selected mechanism in a weakly consistent database system.

The first step consists of discovering *I-offender sets*. For this analysis, it is necessary to model the effects of operations. This information should be provided by programmers, in the form of annotations specifying how predicates are affected by each operation [2]. Using this information and the invariants, a static analysis process infers the minimal sets of operation invocations that may lead to invariant violation when executed concurrently (*I-offender sets*), and the reason for such violation. Conceptually, the analysis considers all valid database states and, for each valid database state, all sets of operation invocations that can execute in that state, and checks if executing all these sets in the same state is valid or not. Obviously, exhaustively considering all database states and operation sets would be impossible in practice, which required the use of the efficient verification techniques..

The second step consists in deciding which approach will be used to handle *I-offender sets*. The programmer must select from the two alternative approaches supported: *invariant-*

[2]This step could be automated using program analysis techniques, as done for example in [37, 38].

repair, in which operations are allowed to execute concurrently and invariants are enforced by automatic conflict resolution rules; *violation-avoidance*, in which the system restricts the concurrent execution of operations that can lead to invariant violation.

In the *invariant-repair* approach, the system automatically guarantees that invariants hold when merging operations executed concurrently, by including the necessary code for restoring invariants in the operations. This is achieved by relying on CRDTs, such as sets, trees and graphs. For example, concurrent changes to a tree can lead to cycles that can be broken using different repair strategies [39].

In the *violation-avoidance* approach, the system uses a set of techniques to control when it is possible and impossible to execute an operation in a datacenter without coordinating with others. For example, to guarantee that an enrollment can only bind a player and a tournament that exist, enrollments can execute in any replica without coordination by forbidding the deletion of players and tournaments. A datacenter can reserve the right to forbid the deletion for a subset of players and tournaments, which gives it the ability to execute enrollments for those players and tournaments without coordinating with other datacenters. Our reservation mechanisms supports such functionality with reservations tailored to the different types of invariants.

Third, the application code is instrumented to use the conflict-repair and conflict-avoidance mechanisms selected by the programmer. This involves extending operations to call the appropriate API functions defined in Indigo.

7.5 Indigo middleware

We have built a prototype of Indigo on top of a geo-replicated data store with the following properties: (i) causal consistency; (ii) support for transactions that access a database snapshot and merge concurrent updates using CRDTs [33]; (iii) linearizable execution of operations for each object in each datacenter. It has been shown that all these properties can be implemented efficiently in geo-replicated stores and at least two systems support all these functionalities: SwiftCloud [40] and Walter [34]. Given that SwiftCloud has a more extensive support for CRDTs, which are fundamental for invariant-repair, we decided to build Indigo prototype on top of SwiftCloud.

Reservation objects are stored in the underlying storage system and they are replicated in all datacenters. Reservation rights are assigned to datacenters individually, which keeps the information small. As discussed in the previous section, the execution of operations in reservation objects must be linearizable (to guarantee that two concurrent transactions do not consume the same rights).

The execution of an operation in the replica where it is submitted has three phases: i) the reservation rights needed for executing the operation are obtained - if not all rights can be obtained, the operation fails; ii) the operation executes, reading and writing the objects of the database; iii) the used rights are released. For escrow reservations, rights consumed are not released; new rights are created in this phase. The side-effects of the operation in the data and reservation objects are propagated and executed in other replicas asynchronously and atomically.

Reservations guarantee that operations that can lead to invariant violation do not execute concurrently. However, operations need to check if the preconditions for operation execution hold before execution[3]. In our tournament example, an operation to remove a tournament cannot execute before removing all enrolled players. Reservations do not guarantee that this is the case, but only that a remove tournament will not execute concurrently with an enrollment.

An operation needs to access a database snapshot compatible with the used reservation rights, i.e., a snapshot that reflects the updates executed before the replica has acquired the rights being used. In our example, for removing a tournament it is necessary to obtain the right that allows such operation. This precludes the execution of concurrent enroll operations for that tournament. After the tournament has been deleted, an enroll operation can obtain a forbid right on tournament removal. For correctness, it is necessary that the operation observes the tournament as deleted, which is achieved by enforcing that updates of an operation are atomic and that the read snapshot is causally consistent (obtaining the forbid right necessarily happens after revoking the allow right, which happens after deleting the tournament). These properties are guaranteed in Indigo directly by the underlying storage system.

Obtaining reservation rights The first and last phases of operation execution obtain and free the rights needed for operation execution. Indigo provides API functions for obtaining and releasing a list of rights. Indigo tries to obtain the necessary rights locally using ordered locking to avoid deadlocks. If other datacenters need to be contacted for obtaining some reservation rights, this process is executed before start obtaining rights locally. Unlike the process for obtaining rights in the local datacenter, Indigo tries to obtain the needed rights from remote datacenters in parallel for minimizing latency. This approach is prone to deadlocks - if some remote right cannot be obtained, we use an exponential backoff approach that frees all rights and tries to obtain them again after an increasing amount of time.

When it is necessary to contact other datacenters to obtain some right, latency of operation execution is severely affected. In Indigo, reservation rights are obtained pro-actively using the following strategy. Escrow lock rights are divided among datacenters, with a datacenter asking for additional rights to the datacenter it believes has more rights (based on local information). Multi-level lock and multi-level mask rights are pre-allocated to allow executing the most common operations (based on the expected frequency of operations), with shared allow and forbid rights being shared among all datacenters. In the tournament example, *shared forbid* for removing tournaments and players can be owned in all datacenters, allowing the most frequent enroll to execute locally.

The middleware maintains a cache of reservation objects and allows concurrent operations to use the same shared (allow or forbid) right. While some ongoing operation is using a shared or exclusive right, the right cannot be revoked.

7.6 Fault tolerance

Indigo builds on the fault-tolerance of the underlying storage system. In a typical geo-replicated store, data is replicated inside a datacenter using quorums or relying on a state-machine replication algorithm. Thus, the failure of a machine inside a datacenter does

[3]This step could be automated by inferring preconditions from invariants and operation side-effects, given that the programmer specifies the code for computing the value of predicates

not lead to any data loss. If a datacenter (fails or) gets partitioned from other datacenters, it is impossible to transfer rights from and to the partitioned datacenter. In each partition, operations that only require rights available in the partition can execute normally. Operations requiring rights not available in the partition will fail. When the partition is repaired (or the datacenter recovers with its state intact), normal operation is resumed.

In the event that a datacenter fails losing its internal state, the rights held by that datacenter are lost. As reservation objects maintain the rights held by all replicas, the procedure to recover the rights lost by the datacenter failure is greatly simplified - it is only necessary to guarantee that recovery is executed only once with a state that reflects all updates received from the failed datacenter.

7.7 Experimental evaluation

This section presents our experimental evaluation of Indigo. The main question our evaluation tries to answer is how does explicit consistency compares against *causal consistency* and *strong consistency* in terms of latency and throughput with different workloads. Additionally, we tried to answer the following questions:

- Can the algorithm for detecting *I-offender sets* be used with realistic applications?
- What is the impact of an increasing the amount of contention in objects and reservations?
- What is the impact of using an increasing number of reservations in each operation?
- What is the behavior when coordination is necessary for obtaining reservations?

To evaluate Indigo, we used the two following applications.

Ad counter We provide, for the reader's convenience, a brief overview of our ad counter industrial use case. The ad counter application models the information maintained by a system that manages the displaying of ads in online applications. This information needs to be geo-replicated for allowing fast delivery of ads. For maximizing revenue, an ad should be displayed exactly the number of times the advertiser is willing to pay for, a requirement that we express as an application invariant. In a real system, when a client application asks for a new ad to be impressed, some complex logic will decide which ad should be impressed. In our application, an operation we only execute a set of reads to counters followed by an operation to increment the number of times a randomly selected ad has been displayed.

Advertisers will typically require ads to be impressed a minimum number of times in some countries - e.g. ad A should be impressed 10.000 times, including 4.000 times in US and 4.000 times in EU. This example is modeled by having the following additional invariants for specifying the limits on the number of impressions (impressions in excess in Europe and US can be accounted in $nrImpressionsOther$).

We modeled this application by having independent counters for each ad and region. Invariants were defined with the limits stored in database objects. A single update operation that increments the ad tally was defined - this operation updates the predicate $nrImpressions$. Our analysis shows that the increment operation conflicts with itself for any given counter, but increments on different counters are independent. Invariants can be enforced by relying on escrow lock reservations for each ad.

Our experiments used workloads with a mix of: a read only operation that returns the value of a set of counters selected randomly; an operation that reads and increments a randomly selected counter. Our default workload included only increment operations.

Tournament management This a version of the application for managing tournaments described in section 7 (and used throughout the paper as our running example), extended with read operations for browsing tournaments. The operations defined in this application are similar to operations that one would find in other management applications such as courseware management.

This application has a rich set of invariants, including uniqueness rules for assigning ids; generic referential integrity rules for enrollments; and order relations for specifying the capacity of each tournament. This leads to a reservation system that uses both escrow lock and multi-level lock reservation objects. Three operations do not require any right to execute - add player, add tournament and disenroll tournament - although the latter access the escrow lock object associated with the capacity of the tournament. The other update operations involve acquiring rights before they can execute.

In our experiments we have run a workload with 82% of read operations (a value similar to the TPC-W shopping workload), 4% of update operations requiring no right for executing, and 14% of update operations requiring rights (8% of the operations are enrollment and disenrollments).

Performance of the Analysis We have implemented the algorithm for detecting *I-offender sets* in Java, relying on the satisfiability modulo theory (SMT) solver Z3 [41] for verifying invariants. The algorithm was able to find the existing *I-offender sets* in the applications. The average running time of this process in a recent MacBook Pro laptop was 19 ms for the ad counter applications and 2892 ms for the more complex tournament application.

We have also modeled TPC-W - the invariants in this benchmark are a subset of those of the tournament application. The average running time for detecting *I-offender sets* was 937 ms. These results show that the running time increases with the number of invariants and operations, but that our algorithm can process realistic applications.

We compare Indigo against three alternative approaches:

Causal Consistency (Causal) As our system was built on top of the causally consistent SwiftCloud system[40], we have used unmodified SwiftCloud as representative of a system providing causal consistency. We note that this system cannot enforce invariants. This comparison allows us to measure the overhead introduced by Indigo.

Strong Consistency (Strong) We have emulated a strongly consistent system by running Indigo in a single DC and forwarding all operations to that DC. We note that this approach allows more concurrency than a typical strong consistency system as it allows updates on the same objects to proceed concurrently and be merged if they do not violate invariants.

Red-Blue consistency (RedBlue) We have emulated a system with Red-Blue consistency [20] by running Indigo in all DCs and having red operations (those that may violate invariants and require reservations) execute in a master DC, while blue operations execute in the closest DC respecting causal dependencies.

Our experiments comprised 3 Amazon EC2 datacenters - US-East, US-West and EU - with inter-datacenter latency presented in Table 1. In each DC, Indigo servers run in a single m3.xlarge virtual machine with 4 vCPUs and 8 ECUs of computational power, and 15GB of memory available. Clients that issue transactions run in up to three m3.xlarge machines. Where appropriate, we placed the master DC in US-East datacenter to minimize the communication latency and have those configurations perform optimally.

RTT (ms)	US-E	US-W
US-West	81	-
EU	93	161

Table 1: RTT Latency among Datacenters in Amazon EC2

Latency and throughput We start by comparing the latency and throughput of Indigo with alternative deployments for both applications. We have run the ad counter application with 1000 ads and a single invariant for each ad. The limit on the number of impressions was set sufficiently high to guarantee that the limit is not reached. The workload included only update operations for incrementing the counter. This allows us to measure the peak throughput when operations are able to obtain reservations in advance. The results show that Indigo achieves throughput and latency similar to a causally consistent system. Strong and RedBlue results are similar, as all update operations are red and execute in the master DC in both configurations.

Results show that Indigo achieves throughput and latency similar to a causally consistent system. In this case, as most operations are read-only or can be classified as blue and execute in the local datacenter, RedBlue throughput is only slightly worse than that of Indigo. The results also show that Indigo exhibits lower latency than RedBlue for red operations. These operations can execute in the local DC in Indigo, as they require either no reservation or reservations that can be shared and are typically locally available.

Two other results deserve some discussion. *Remove tournament* requires canceling shared forbid rights acquired by other DCs before being able to acquire the shared allow right for removing the tournament, which explain the high latency. Sometimes latency is extremely high (as shown by the line with the maximum value) - this is a result of the asynchronous algorithms implemented and the approach for requesting remote DCs to cancel their rights, which can fail when a right is being used. This could be improved by running a more elaborate protocol based on Paxos. *Add player* has a surprisingly high latency in all configurations. Analyzing the situation, we found out that the reason for this lies in the fact that this operation manipulates very large objects used to maintain indexes - all configurations have a fix overhead due to this manipulation.

The throughput of Indigo decreases when contention increases as several steps require executing operations sequentially. Our middleware introduces additional contention when accessing the cache. As the underlying storage system also implements linearizability per-object, it is also possible to observe its throughput also decreases with increased contention, although more slowly. The results also show that the peak throughput with Indigo decreases while latency keeps constant. The reason for this is that for escrow locks, each invariant has an associated reservation object - thus, when increasing the number of invariants the number of updated objects also increases, with impact on the operations that each datacenter needs to execute. To verify our explanation, we have run a workload with operations that access the same number of counters in the weak consistency configuration - the presented results show the same pattern for decreased throughput. When rights do not exist locally, Indigo cannot mask the latency imposed by coordination - in this case, for obtaining additional rights from the remote datacenters.

A big impact in latency is only experienced when it is necessary to revoke shared forbid rights in all replicas before acquiring the needed shared allow right. The positive consequence of this approach is that enroll operations requiring the shared forbid right that was shared by all replicas execute with latency close to zero. The maximum latency line in enroll operation shows the maximum latency experienced when a replica acquires a shared forbid right from a replica already holding such right.

Conclusions from Experiments The results show that the modified applications have performance similar to weak consistency for most operations, while being able to enforce application invariants. Some rare operations that require intricate rights transfers exhibit high latency. As future work, we intend to improve the algorithms for exchanging reservation rights on those situations.

7.8 Conclusions

We investigated an application-centric consistency model for geo-replicated services - explicit consistency - where programmers specify the consistency rules that the system must maintain as a set of invariants. We described a methodology that helps programmers decide which invariant-repair and violation-avoidance techniques to use to enforce explicit consistency, extending existing applications. We also present the design of Indigo, a middleware that can enforce explicit consistency on top of a causally consistent store. The results show that the modified applications have performance similar to weak consistency for most operations, while being able to enforce application invariants. Some rare operations that require intricate rights transfers exhibit high latency. As future work, we intend to improve the algorithms for exchanging reservation rights on those situations.

8 Papers and publications

- Manuel Bravo, Zhongmiao Li, Peter Van Roy, and Christopher Meiklejohn. *Derflow: Distributed Deterministic Dataflow Programming for Erlang*, 13th ACM SIGPLAN Erlang Workshop, Gothenburg, Sweden, Sep. 5, 2014.
- Christopher Meiklejohn. *Eventual Consistency and Deterministic Dataflow Programming*, 8th Workshop on Large-Scale Distributed Systems and Middleware (LADIS '14), Cambridge, UK, Oct. 23-24, 2014.
- Peter Zeller, Annette Bieniusa and Arnd Poetzsch-Heffter. *Formal Specification and Verification of CRDTs*, Formal Techniques for Distributed Objects, FORTE 2014, Berlin, Germany, June 3-5, 2014.
- Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. *Putting Consistency Back into Eventual Consistency*. Submitted to EuroSys 2015.
- Burcu Kulahcioglu Ozkan, Erdal Mutlu and Serdar Tasiran. *Towards Verifying Eventually Consistent Applications*, Workshop on the Principles and Practice of Eventual Consistency, PaPEC '14. Amsterdam, The Netherlands, April 13, 2014.

We apologize for the last publication, authored by Koç University researchers, which fails to acknowledge SyncFree funding. This is Prof. Tasiran's group's first EU project involvement and he and his students were unaware of this requirement. They will be sure to acknowledge SyncFree funding in upcoming publications.

References

[1] Kuper, Lindsey, and Ryan R. Newton. LVars: lattice-based data structures for deterministic parallelism. Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing. ACM, 2013.

[2] Dean, Jeffrey, and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. Communications of the ACM 51.1 (2008): 107-113.

[3] Rusty Klophaus. Riak Core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*, New York, NY, USA, , Article 14 , 1 pages.

[4] Sébastien Doeraene and Peter Van Roy. A New Concurrency Model for Scala Based on a Deterministic Dataflow Core. Fourth Annual Scala Workshop, Montpellier, France, July 1-2, 2013.

[5] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming Languages for Distributed Applications. Journal of New Generation Computing, May 1998, Vol. 16, No. 3, pp. 223-261.

[6] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient Logic Variables for Distributed Computing. ACM Transactions on Programming Languages and Systems (TOPLAS), May 1999, pp. 569-626.

[7] Michael Maher. Logic Semantics for a Class of Committed-choice Programs. In *International Conference on Logic Programming (ICLP 87)*, Melbourne, Australia, May 1987, pp. 856-876.

[8] F. Rossi, P. van Beek, T. Walsh (eds.). *Handbook of Constraint Programming*, Elsevier, 2006.

[9] Vijay A. Saraswat. *Concurrent Constraint Programming*, MIT Press, 1993.

[10] Paulo Sérgio Almeida and Carlos Baquero. Scalable Eventually Consistent Counters over Unreliable Networks. arXiv:1307.3207 [cs.DC], July 2013.

[11] DeCandia, Giuseppe, et al. Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review. Vol. 41. No. 6. ACM, 2007.

[12] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile Objects in Distributed Oz. ACM Transactions on Programming Languages and Systems (TOPLAS), Sep. 1997, pp. 804-851.

[13] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004.

[14] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, March 2013. This document is work in progress. Feel free to cite, but note that we will update the contents

without warning (the first page contains a timestamp), and that we are likely going to publish the content in some future venue, at which point we will update this paragraph.

[15] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 271–284. ACM, 2014.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[17] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[18] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. ACM, 2000.

[19] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 281–292. USENIX Association, 2014.

[20] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012.

[21] Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Brief announcement: Update consistency in partitionable systems. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, page 546, 2014.

[22] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.

[23] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination-avoiding database systems. *CoRR*, abs/1402.2237, 2014.

[24] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*, volume 8461 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014.

[25] Martín Abadi, Leslie Lamport, and Stephan Merz. A tla solution to the rpc-memory specification problem. In Manfred Broy, Stephan Merz, and Katharina Spies, editors, *Formal Systems Specification*, volume 1169 of *Lecture Notes in Computer Science*, pages 21–66. Springer Berlin Heidelberg, 1996.

[26] Tianxiang Lu, Stephan Merz, Christoph Weidenbach, et al. Model checking the pastry routing protocol. In *10th International Workshop Automated Verification of Critical Systems*, pages 19–21, 2010.

[27] Stephan Merz, Martin Wirsing, and Júlia Zappe. A spatio-temporal logic for the specification and refinement of mobile systems. In *Fundamental Approaches to Software Engineering*, pages 87–101. Springer, 2003.

[28] Chris Newcombe. Why amazon chose tla + . In Yamine Ait Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477 of *Lecture Notes in Computer Science*, pages 25–39. Springer Berlin Heidelberg, 2014.

[29] Leslie Lamport. Specifying systems. Addison-Wesley Reading, 2002.

[30] Yuan Yu, Panagiotis Manolios and Leslie Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66, Springer, 1999.

[31] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. 2011.

[32] Jim Gray and Leslie Lamport. Consensus on transaction commit. In *ACM Trans. Database Syst.*, Volume 31, pages 133–160, ACM, New York, NY, USA, March 2006.

[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Springer-Verlag, 2011.

[34] Yair Sovran, Russell Power, Marcos K. Aguilera and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 385–400, ACM, 2011.

[35] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS '77)*, pages 46–57, IEEE, 1977.

[36] Leslie Lamport. The temporal logic of actions. In *ACM Trans. Program. Lang. Syst.*, Volume 16, pages 872–923, ACM, May 1994.

[37] Cheng Li, J. Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*, pages 281–292, USENIX Association, 2014.

[38] Sudip Roy, Lucja Kot, Nate Foster, Johannes Gehrke, Hossein Hojjat and Christoph Koch. Writes that fall in the forest and make no sound: semantics-based adaptive data consistency. In *CoRR '14*, 2014.

[39] Stéphane Martin, Mehdi Ahmed-Nacer and Pascal Urso. Abstract unordered and ordered trees CRDT. In *CoRR '12*, 2012.

[40] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro and Nuno M. Preguiça SwiftCloud: fault-tolerant geo-replication integrated all the way to the client machine. In CoRR '13, 2013.

[41] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, pages 337–340, Springer, 2008.

A Derflow: Distributed Deterministic Dataflow Programming for Erlang

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:

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

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:

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

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).

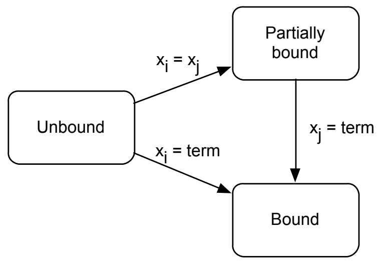

Figure 1. Dataflow variable state diagram from x_i perspective

- 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
  - $\forall x \in x_i.bound\_variables,$ $bind(x, v_i)$
  - $x_i.value = v_i$
- 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: $\sigma = \{x_1, \dots, x_i, \dots, x_n\}$
- $y = monitor(x_i)$
  - $x_i.monitors \cup \{self()\}$
  - $y = declare()$
- After: $\sigma = \{x_1, \dots, x_i, \dots, x_n, y\}$

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

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.

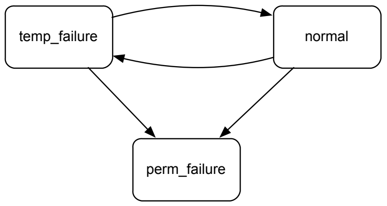

Figure 2. Dataflow variable reachability state diagram.

- *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.

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

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.

B Eventual Consistency and Deterministic Dataflow Programming

Eventual Consistency and Deterministic Dataflow Programming

A Case Study of Integrating Derflow with the Riak Data Store

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

Abstract

Even with the upcoming 2.0 release, queryability of Riak [1], an open source distributed database from Basho Technologies, remains an area for improvement. As of this release, Riak provides three main mechanisms for executing queries across values stored in the database: secondary indexing (2i), a MapReduce-like [5] framework, and Yokozuna. However, all three have significant drawbacks in terms of scalability and flexibility.

Secondary indexing offers the ability to tag objects as they are written into the database with key-value pairs that can be used as the basis for queries. However, the entire set of tags needs to be specified every time the object is written, and tags are restricted to range and equalities over strings and integers. In addition, there is no mechanism for providing ad-hoc conjunctions or disjunctions.

Riak's MapReduce-like framework, provided through an application called $riak\_pipe$, provides the ability to do on-demand, scatter-gather queries, but requires re-evaluation of the whole input set even though there may be no changes for a phase, causing increased cluster load.

Finally, Yokozuna provides an abstraction over distributed Solr, but relies on a glue layer on top of Riak to interface with a JVM running on each node, executing Solr queries across the cluster. As of writing, it is still unclear how far this mechanism can be scaled, and what penalty exists at scale when moving data between the Java Virtual Machine and the Erlang runtime system.

Given these drawbacks, we have identified a series of desirable properties for a future query mechanism for Riak. Specifically, these are:

- The ability for a user to submit a computation to the Riak cluster, and have it performed in a highly-available, fault-tolerant manner across the entire cluster.
- An execution mechanism that can re-use and incrementally update partial results, thereby alleviating the need to re-execute the entire query across the cluster on repeated executions.
- A query mechanism that reduces harvest while maintaining yield [3] during failure conditions.

Recently, there has been a series of research efforts surrounding the use of bounded-join semilattices, a generalization of state-based conflict-free replicated data types (CRDTs) [9], as data structures in new programming models to provide deterministic execution in distributed scenarios. Two examples of this are LVars [8], providing deterministic execution across multiple threads in Haskell, and Bloom [4], which provides deterministic execution across multiple instances of the Ruby virtual machine. In both these cases, properties of the bounded-join semilattice, combined with monotone functions, assist in ensuring determinism, specifically handling cases of repeated updates and out-of-order updates.

More recently, as part of the SyncFree project in the European Seventh Framework Programme of which Basho is a participant, there has been further addition to these distributed deterministic programming models named Derflow [10]. Derflow provides a similar programming model, but is built using the Erlang-based, Dynamo-inspired [6], distributed systems toolkit, $riak\_core$. [2]

We explore the process of developing the reference implementation of Derflow while simultaneously integrating the reference implementation with Riak to provide a new prototype query mechanism. We expose the ability for users to submit deterministic computations to the Riak data store, which are executed as values are written, providing the user the ability to retrieve the computed results through a query API, similar to a materialized view mechanism as exposed by other commercial databases, such as CouchDB. [7]

Our integration exploits the following properties of Riak and Derflow:

- Computations and their results are partitioned and replicated along with their input data in the data store. This allows us to provide highly-available results, which sacrifice harvest during failure conditions. [1]
- The partial results of computations can be combined deterministically, given the merge properties of state-based conflict-free replicated data types.

The main contribution of this talk is an experience report from the Basho engineering team that details the following:

- Assisting in the design and development of the Derflow library on top of $riak\_core$.
- Adapting the research concept and reference implementation of Derflow into Riak.
- Contributing changes made to Derflow for use inside of Riak back to the reference implementation of Derflow.

Acknowledgments

This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n^o 609551.

References

[1] Basho Technologies Inc. Riak source code repository. `http://github.com/basho/riak`, .

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

[1] Specifically, as input values to the computation become unavailable, the computations that resulted in those inputs also become unavailable.

[3] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001. ISSN 1089-7801. . URL `http://dx.doi.org/10.1109/4236.939450`.

[4] 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. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-167.html`.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/1327452.1327492`.

[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] B. Holt. *Writing and Querying MapReduce Views in CouchDB*. O'Reilly Media, Inc., 1st edition, 2011. ISBN 1449303129, 9781449303129.

[8] 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`.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011. URL `http://hal.inria.fr/inria-00555588`.

[10] SyncFree. Derflow source code repository. `http://github.com/syncfree/derflow`.

C Formal Specification and Verification of CRDTs

Formal Specification and Verification of CRDTs

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{p_zeller,bieniusa,poetzsch}@cs.uni-kl.de

Abstract. *Convergent Replicated Data Types* (CRDTs) can be used as basic building blocks for storing and managing replicated data in a distributed system. They provide high availability and performance, and they guarantee eventual consistency. In this paper, we develop a formal framework for the analysis and verification of CRDTs. We investigate and compare the three currently used specification techniques for CRDTs and formalize them based on an abstract model for managing replicated data in distributed systems. We show how CRDT implementations can be expressed in our framework and present a general strategy for verifying CRDTs. Finally, we report on our experiences in using the framework for the verification of important existing CRDT implementations. The framework and the proofs were developed within the interactive theorem prover Isabelle/HOL.

Keywords: CRDT, formal verification, eventual consistency

1 Introduction

Global computing systems and worldwide applications often require *data replication*, that is, the data is not only stored at one computing node, but at several nodes. There are a number of reasons for replication. To realize reliable services, high availability and fault-tolerance might be important. A centralized storage system might not provide enough throughput for allowing millions of users to access the data. Furthermore, systems often serve clients from different regions in the world; Geo-replication helps to keep the latency low. Last but not least, mobile clients might not be connected all the time, but should provide certain offline functionality using a local store that is synchronized with the global state when the connection is reestablished.

The CAP theorem[6] tells us that distributed systems cannot guarantee high availability, partition tolerance, and strong consistency at the same time. In this paper, we investigate techniques that aim at high availability and partition tolerance by providing a weaker form of consistency, called *eventual consistency*[14,15,11,5].

In an eventually consistent system, the different replicas do not have to provide the same view of the data at all times. Operations manipulating the data are first applied to a subset of the replicas. At some suitable later point in time, the updates are communicated to the other replicas. Only after all updates are delivered to all replicas the data has to be consistent again. This means in particular that the replicas have to be able to merge concurrent updates into a consistent view. It usually depends on the application how a merge operation should behave. The idea of replicated data types is to package a behavior given by operations and a way to handle concurrent updates, so that they can be reused in many situations.

Convergent replicated data types (CRDTs)[12] are a special class of replicated data types where concurrent updates are handled by merging the resulting states. The basic setting is that there is a number of replicas of the data type. Operations are executed on a single replica and the different replicas conceptually exchange their whole states with each other according to some protocol. The state of a replica is also called the *payload*. To guarantee convergence, there needs to be a partial order on the payloads and a merge operation computing the least upper bound, such that the payloads form a semilattice. All operations which change the payload have to increase the payload with respect to the partial order. This setup ensures that replicas which have seen the same set of updates also have the same state and thus return the same value. This property is called Strong Eventual Consistency (SEC)[13].

A simple example of a CRDT is a counter. A counter provides update operations to increment or decrement the value and a query function to read the value. It is usually implemented by keeping the number of increments and the number of decrements for each replica in a map from replica-IDs to positive integers. Each replica only increments its own increment- and decrement-counts and states are merged by taking the component-wise maximum. It is easy to prove that this computes a least upper bound. The value of the counter can be determined by summing up all increment counts in the map and subtracting the sum of the decrement counts. Separating increment- and decrement-counts ensures, that the updates are increasing operations. Having a count for every replica instead of a single counter ensures that no updates are lost due to concurrent writes.

Contributions. In general, CRDT implementations can be quite sophisticated and complex, in particular in the way they handle concurrent updates. Therefore it is important to provide high-level behavioral specifications for the users and techniques to verify CRDT implementations w.r.t. their specifications. Towards this challenge, the paper makes the following contributions:

- A framework in Isabelle/HOL [10] that supports the formal verification of CRDTs. The framework in particular provides a system model that is parametric w.r.t. the CRDT to be analyzed and support for verification.
- We analyzed, clarified, and formalized different specification and abstract implementation techniques for CRDTs (Sections 3, 4 and 6).
- We successfully verified a number of CRDTs described in the literature with our framework.

We present the system model in Section 2; describe our specification technique in Section 3, the implementation technique in Section 4 and the verification as-

System state:

$\mathit{version} :: \mathit{replicaId} \rightarrow \mathbb{N}$

$\mathit{payloadHistory} :: (\mathit{version} \times \mathit{payload})\ \mathit{set}$

$\mathit{systemState} :: (\mathit{replicaId} \rightarrow \mathit{payload}) \times (\mathit{replicaId} \rightarrow \mathit{version}) \times \mathit{payloadHistory}$

Operations and traces:

$\mathit{Operation} := \mathit{Update}(\mathit{replicaId}, \mathit{args}) \quad | \quad \mathit{Merge}(\mathit{replicaId}, \mathit{version}, \mathit{payload})$

$\mathit{Trace} := \mathit{Trace};\ \mathit{Operation} \quad | \quad []$

Operational semantics:

$$s_{init} = (\lambda r.\ \mathit{init}_{crdt},\ \lambda r.\ v_0,\ \emptyset) \qquad \frac{}{s \xrightarrow{[]} s} \qquad \frac{s \xrightarrow{as} s' \qquad s' \xrightarrow{a} s''}{s \xrightarrow{as;a} s''}$$

$$(\mathit{update}) \quad \frac{v' = vs(r)(r{+}{=}1) \qquad pl' = \mathit{update}_{crdt}(a, r, pls(r))}{(pls,vs,ph) \xrightarrow{Update(r,a)} (pls(r := pl'),\ vs(r := v'),\ ph \cup \{(v', pl')\})}$$

$$(\mathit{merge}) \quad \frac{(v, pl) \in ph \qquad v' = vs(r) \sqcup v \qquad pl' = \mathit{merge}_{crdt}(pls(r), pl)}{(pls,vs,ph) \xrightarrow{Merge(r,v,pl)} (pls(r := pl'),\ vs(r := v'),\ ph \cup \{(v', pl')\})}$$

Table 1. System Model

pects in Section 5. In Section 6 we discuss an alternative specification technique. Finally, we consider related work, summarize the results, and give an outlook on future work in Sections 7 and 8

2 System model

We developed a formal system model that supports an arbitrary but fixed number of replicas and is parametric in the CRDT to be analyzed, i.e., it can be instantiated with different CRDTs. A CRDT is parameterized by four type parameters and described by four components:

– **pl**, the type of the payload (i.e. the state at each replica)
– **ua**, the type for the update arguments (a sum type of all update operations)
– **qa**, the type for the query arguments (a sum type of all query operations)
– **r**, the sum type for the possible return values of the queries

– **init** :: pl , the initial payload for all replicas
– **update** :: $ua \Rightarrow \mathit{replicaId} \Rightarrow pl \Rightarrow pl$, the function expressing how an update operation modifies the payload at a given replica, where *replicaId* denotes the node on which the update is performed first
– **merge** :: $pl \Rightarrow pl \Rightarrow pl$, the function merging payloads

– **query** :: $qa \Rightarrow pl \Rightarrow r$, the function expressing the results of querying a payload

Given a CRDT $(init_{crdt}, update_{crdt}, merge_{crdt}, query_{crdt})$, the system model describes the labeled transition relation $s \xrightarrow{tr} s'$ expressing that executing trace tr in state s leads to state s' (cf. Table 1) where a trace is a sequence of operations and an operation is either the application of an update or a merge (queries need not be considered, as they do not modify the state). The state of the system consists of three components:

– For each replica r, its current payload.
– For each replica r, its current version vector[8]. The *version vector* or short *version* is a mapping from replica-IDs to natural numbers. If the version of r has m as entry for key k, the payload of r has merged the first m operations applied to replica k into its payload.
– The set of all version-payload pairs that have been seen during execution so far. This set is called the *payload history* and is used to make sure that a merge operation can only be applied to version-payload pairs that appeared earlier in the execution.

Initially, the payload of each replica is $init_{crdt}$, the version vector of each replica is the all-zero vector, and the payload history is the empty set. There are two kind of transition steps:

– An *update* operation $Update(r, a)$ applies an update function of the CRDT (determined by the arguments a) to the current payload of r and modifies the state accordingly; in particular, the rth component of the version vector of r is incremented by one.
– A *merge* operation $Merge(r, v, pl)$ is executed as follows: The new version v' is calculated by taking the least upper bound of the old version vector $vs(r)$ and the merged version vector v. Similarly, the new payload pl' is the least upper bound operation on payloads, which is specific to the respective CRDT, and implemented by the $merge_{crdt}$ function.

Discussion. The system model focuses on simple operational behavior. Other aspects, such as timestamps, were intentionally left out because they would make the system model more complicated than needed for most CRDTs. Only a few CRDTs like the Last-Writer-Wins-Register depend on timestamps. Also, often timestamps are only used to provide some total order on operations, and lexicographic ordering of the version vectors also suffices to provide such an order.

3 Specification

In this section we present and formalize a technique for specifying the behavior of CRDTs based on the system model presented in the previous section.

A specification should tell users the result value of any query operation, when a trace of operations performed in the system is given. A trace gives a total order on the operations performed in the system, but for operations performed independently on different replicas this order does not influence the result. Therefore, we would like to abstract from this total order. Furthermore, the traces include explicit merge operations, but from a user's perspective this merge operations are implicit. Thus, it should not be important, how updates were delivered to a replica. The result of an operation should only depend on those operations that are visible when the operation is about to be performed. Hence, the trace can be actually deconstructed into a partially ordered set of update operations, where the partial order is the visibility relation, which we denote by $\prec$ in the following.

The specification technique that we formalize here supports the sketched abstractions and explains the result of an operation only depending on the visible update history. It follows the ideas from Bouajjani et al.[4], and Burckhardt et al.[5], but is specifically tailored to CRDTs, allowing for some simplifications. In the case of CRDTs, the visibility relation is a partial order. All operations at one replica are ordered by time and a merge makes all operations, which are visible to the source of the merge, visible at the destination of the merge.

Formalization. In our formalization we represent the visibility relation using version vectors. The advantage of this is that they are easy to handle in the operational semantics and also when working with Isabelle/HOL. The properties of a partial order like transitivity and antisymmetry are already given by the structure and do not have to be specified additionally. The version vector at each replica can be directly derived from a given trace. It also uniquely identifies every operation, and we can encode the whole history of updates with the visibility relation as a set of update operations, represented by $(\mathit{version}, \mathit{replicaId}, \mathit{args})$ triples. We call this structure the **update history** and denote it by H in the following. The visibility relation $\prec$ on update operations is simply derived from the order on the version vectors.

A specification is formalized as a function *spec*, which takes the update history visible at a given replica and the arguments of a query and returns the result of the query. A specification is **valid** if for every reachable state and all queries a, the specification yields the same result as the application of the query to the current state:

$$\forall_{tr,pls,vs,a,r}.\ s_{init} \xrightarrow{tr} (pls, vs, \_) \ \Rightarrow\ \mathit{spec}(H(tr, vs(r)), a) = \mathit{query}_{crdt}(a, pls(r))$$

Here, the term $H(tr, vs(r))$ calculates the update history from the trace tr while only taking the operations before the version $vs(r)$ into account.

Examples. Table 2 shows specifications for several CRDTs from the literature[12]. The **Counter** is a data type providing an update-operation to increment or decrement the value of the Counter and a query-operation $\mathit{Get}()$ which returns the current value of the counter. The argument to the update is a single integer value. We specify the return value of a $\mathit{Get}()$ operation on a counter by

Counter:	$spec(H, Get()) = \sum_{e \in H.} args(e)$
Grow-Set:	$spec(H, Contains(x)) = \exists_{e \in H.} args(e) = Add(x)$
Two-Phase-Set:	$spec(H, Contains(x)) = \exists_{e \in H.} args(e) = Add(x) \land \neg(\exists e \in H.\ args(e) = Remove(x))$
Two-Phase-Set (guarded remove):	$spec(H, Contains(x)) = \exists_{e \in H.} args(e) = Add(x) \land \neg(\exists_{e \in H.} args(e) = Remove(x) \land (\exists_{f \in H.} args(f) = Add(x) \land f \prec e))$
Observed-Remove-Set:	$spec(H, Contains(x)) = \exists_{a \in H.} args(a) = Add(x) \land \neg(\exists_{r \in H.} a \prec r \land args(r) = Remove(x))$
Multi-Value-Register:	$spec(H, Get()) = \{x \mid \exists_{e \in H.} args(e) = Set(x) \land \neg(\exists_{f \in H.} e \prec f)\}$

Table 2. Specifications of CRDTs

taking all update operations e from the update history H and then summing up their update arguments. The **Grow-Set** is a set which only allows adding elements. An element is in the set, when there exists an operation adding the element. The **Two-Phase-Set** also allows to remove elements from the set, with the limitation that an element cannot be added again once it was removed. An element is in the set, if there is an operation adding the element and no operation removing it. This specification allows removing an element before it was added to the set, which might not be desired. The **Two-Phase-Set with the guarded remove operation**, ignores remove operations, when the respective element is not yet in the set. For this data type, an element is in the set, when there exists an operation adding the element, and there is no operation which removes the element and which happened after an add operation of the same element. The **Observed-Remove-Set** is a set, where an element can be added and removed arbitrarily often. A remove operation only affects the add operations which have been observed, i.e. which happened before the remove operation. We specify that the query $Contains(x)$ returns true, if and only if there exists an update operation a adding x to the set and there exists no update operation r which happened after a and removes x from the set. The final example is the **Multi-Value-Register**. It has a $Set(x)$ operation to set the register to a single value. The $Get()$ query returns a set containing the values of the last concurrent Set operations. More precisely, it returns all values x so that there exists an operation $Set(x)$, for which no later update operation exists.

Properties and Discussion. It is not possible to describe non-converging data types with this technique. Since the specified return value of a query only depends on the visible update history and the arguments, two replicas which have seen the same set of updates will also return the same result.

One problem with this specification technique is that in general a specification can reference all operations from the past. The state of a replica is basically determined by the complete update history, which can be quite large and not very abstract. Therefore it is hard to reason about the effects of operations when programming with the data types or when verifying properties about systems using them, where one usually wants to reason about the effect of a single method in a modular way.

Also, the example of the Two-Phase-Set with a guarded remove operation shows that small changes to the behavior of one update operation can make the whole specification more complex. We would like such a change to only affect the specification of the remove operation. Thus, the question is whether we can specify CRDTs avoiding these problems. We present and discuss an alternative specification technique in Section 6. It is also possible to use abstract implementations as a form of specification, as detailed in the next section.

4 Implementations

To implement a CRDT in our framework one has to define the type of the payload and the four fields of the CRDT record ($init_{crdt}$, $update_{crdt}$, $merge_{crdt}$, $query_{crdt}$) as defined in the system model. Technically, the implementation can be any Isabelle function with a matching type.

To keep the examples short, we introduced a *uid*-function, which generates a new unique identifier. This can easily be implemented in our system model by adding a counter to the payload of the data type. A unique identifier can then be obtained by taking a pair (*replicaId, counter*) and incrementing the counter. It is also possible to use the version vector as a unique identifier for an update operation, which can make the verification easier, as the payload is then directly related to the update history.

Abstract Counter:	$s :: (id \times int)set = \{\}$ $\mathbf{update}(x, r, s) = s \cup \{(uid(), x)\}$ $\mathbf{merge}(s, s') = s \cup s'$ $\mathbf{query}(Get(), s) = \sum_{(id,x) \in s} x$
Optimized Counter:	$s :: (replicaId \to int) \times (replicaId \to int) = (\lambda r.\ 0, \lambda r.\ 0)$ $\mathbf{update}(x, r, (p, n)) = \mathbf{if}\ x \geq 0\ \mathbf{then}\ (p(r := p(r) + x), n)$ $\mathbf{else}\ (p, n(r := n(r) - x))$ $\mathbf{merge}((p, n), (p', n')) = (\lambda r.\ max(p(r), p'(r)), \lambda r.\ max(n(r), n'(r)))$ $\mathbf{query}(Get(), (p, n)) = \sum_r p(r) - \sum_r n(r)$

Table 3. Abstract and optimized implementation of a Counter CRDT

Table 3 shows two implementations of the Counter CRDT. The first implementation is an abstract one, in which the payload is a set of all update arguments tagged with a unique identifier. The query can then be answered by summing up all the update arguments in the set. This implementation is very inefficient, but easy to understand. The second implementation is closer to Counter implementations found in real systems. Here, the payload consists of two mappings from replicaIds to integers. The first map (p) sums up all the positive update operations per replica and the second map (n) sums up all the negative ones. While this is still one of the easier CRDTs, it is not trivial to see, that the optimized implementation is valid with respect to its specification. We will come back to this example in Section 5 and show how the correctness can be proven using our framework.

Grow-Set:	$s ::' a\ set = \{\}$ $\mathbf{update}(Add(x), r, s) = s \cup \{x\}$ $\mathbf{merge}(s, s') = s \cup s'$ $\mathbf{query}(Contains(x), s) = x \in s$
Two-Phase-Set:	$s ::' a \Rightarrow \{init = 0, in = 1, out = 2\} = (\lambda x.\ init)$ $\mathbf{update}(Add(x), r, s) = (\mathbf{if}\ s(x) = init\ \mathbf{then}\ s(x := in)\ \mathbf{else}\ s)$ $\mathbf{update}(Rem(x), r, s) = s(x := out)$ $\mathbf{merge}(s, s') = (\lambda x.\ max(s(x), s'(x)))$ $\mathbf{query}(Contains(x), s) = (s(x) = in)$
Two-Phase-Set (guarded remove):	Same as above, but with different remove operation: $\mathbf{update}(Rem(x), r, s) = (\mathbf{if}\ s(x) = in\ \mathbf{then}\ s(x := out)\ \mathbf{else}\ s)$
Observed-Remove-Set:	$s.e :: (id \times' a)set = \{\}, s.t :: id\, set = \{\}$ $\mathbf{update}(Add(x), r, s) = s(e := s.e \cup \{(uid(), x)\})$ $\mathbf{update}(Rem(x), r, s) = s(t := s.t \cup \{id \mid \exists_x.(id, x) \in s.e\})$ $\mathbf{merge}(s, s') = (e = s.e \cup s'.e,\ t = s.t \cup s'.t)$ $\mathbf{query}(Contains(x), s) = \exists_{id}.(x, id) \in s.e \wedge id \notin s.t$
Multi-Value-Register:	$s.e :: (id \times' a)set = \{\}, s.t :: id\, set = \{\}$ $\mathbf{update}(Set(x), r, s) = s(e := \{(uid(), x)\},$ $t := s.t \cup \{id \mid \exists_x.(id, x) \in s.e\})$ $\mathbf{merge}(s, s') = (e = s.e \cup s'.e,\ t = s.t \cup s'.t)$ $\mathbf{query}(Get(), s) = \{x \mid \exists_{id}.(x, id) \in s.e \wedge id \notin s.t\}$

Table 4. State-based specifications of CRDTs

Table 4 shows abstract implementations of the other CRDTs introduced in Section 3. The Grow-Set can be implemented using a normal set where the merge is simply the union of two sets. The payload of the Two-Phase-Set can be described by assigning one out of three possible states to each element. In a new, empty set all elements are in the *init* state. Once an element is added, it goes to the *in* state, and when it is removed it goes to the *out* state. The merge simply takes the maximum state for each element with respect to the order $init < in < out$. The last two CRDTs in Table 4 have a very similar implementation. This is not very surprising, as the *Set* operation of the register is basically an operation, that first removes all elements from the set and then adds a single new element. In both cases the payload consists of a set of elements tagged with an unique identifier and a set of tombstones, that contains all unique identifiers of the removed elements. The unique identifier makes sure, that the remove operation only affects previous add-operations, as it is demanded by the specification.

Relation to Specifications. All CRDT implementations can be specified by the specification technique described in Section 3, when the merge operation computes a least upper bound with respect to a semilattice and the update operations are increasing with respect to the order on the semilattice. This is possible, as the state can be reconstructed from a given update history, when the implementation is known. Because the merge operation of a CRDT computes a least upper bound, it is straight-forward to extend it to a merge function, which merges a set of payloads. Then a function to calculate the state can be defined recursively in terms of previous versions: If there is an update at the top of the history, apply the update operation to the merge of all previous versions. If there is no update operation at the top, just take the merge of all previous versions. This terminates, when the set of all previous versions only consists of the initial state.

The converse is also true: each specification given in this form describes a CRDT. A specification can be turned into an inefficient implementation by storing the visible update history in the payload of the data type. The update history is just a growing set which can be merged using the union of sets, thus forming a semilattice.

5 Verification

In our work on verification of CRDTs we considered two properties. The first property is the convergence of a CRDT, meaning that two replicas, which have received the same set of updates, should return the same results for any given query. This property is common to all CRDTs and does not require any further specification. The second property is the behavior of a CRDT, i.e. we want to prove, that a specification as presented in Section 3 is valid for a given implementation. As we discussed earlier, this is a strictly stronger property, but it requires a specification for each data type.

Section 5.1 covers the verification of the convergence property, in Section 5.2 we present a technique for verifying the behavior, and in Section 5.3 we evaluate our experience in using Isabelle/HOL for the verification of CRDTs with the presented techniques.

5.1 Verification of Convergence

The convergence property can be verified by proving that the payload of the CRDT forms a semilattice, such that the merge-operation computes a least upper bound and the update-operations increase the payload with respect to the order on the semilattice.[12]

(refl)	$Inv(H, pl) \Rightarrow pl \leq_{crdt} pl$
(trans)	$Inv(H_1, pl_1) \wedge Inv(H_2, pl_2) \wedge Inv(H_3, pl_3) \wedge$ $pl_1 \leq_{crdt} pl_2 \wedge pl_2 \leq_{crdt} pl_3 \Rightarrow pl_1 \leq_{crdt} pl_3$
(antisym)	$Inv(H_1, pl_1) \wedge Inv(H_2, pl_2) \wedge pl_1 \leq_{crdt} pl_2 \leq_{crdt} pl1 \Rightarrow pl_1 = pl_2$
(commute)	$Inv(H_1, pl_1) \wedge Inv(H_2, pl_2) \Rightarrow merge_{crdt}(pl_1, pl_2) = merge_{crdt}(pl_2, pl_1)$
(upper bound)	$Inv(H_1, pl_1) \wedge Inv(H_2, pl_2) \Rightarrow pl_1 \leq_{crdt} merge_{crdt}(pl_1, pl_2)$
(least upper bound)	$Inv(H_1, pl_1) \wedge Inv(H_2, pl_2) \wedge Inv(H_3, pl_3) \wedge$ $pl_1 \leq_{crdt} pl_3 \wedge pl_2 \leq_{crdt} pl_3 \Rightarrow merge_{crdt}(pl_1, pl_2) \leq pl_3$
(monotonic updates)	$Inv(H, pl) \Rightarrow pl \leq_{crdt} update_{crdt}(args, r, pl)$

Table 5. Verifying convergence of CRDTs

However, only very simple data types form a semilattice in the classical mathematical sense. Often the semilattice properties only hold for a subset of the payloads. For some states which are theoretically representable by the payload type, but are never reached in an actual execution, the semilattice properties sometimes do not hold. In theory it could even be the case that there are two reachable states for which the merge operation does not yield the correct result, but where the two states can never be reached in the same execution. However, for the examples we considered it was always sufficient to restrict the payload to exclude some of the unreachable states. Technically, this was done by giving an invariant *Inv* over the update history H and the payload *pl*. The same type of invariant will also be used for the verification of behavioral properties in the next section. In the examples we considered, it was not necessary to use the update history H in the invariant. An overview of the sufficient conditions for convergence, which we used, is given in Table 5. The order on the payloads is denoted by $\leq_{crdt}$.

In order to verify these conditions for the Counter CRDT, we have to define the order on the payloads. Here we can simply compare the mappings for each replicaId: $(p, n) \leq (p', n') \leftrightarrow \forall_r \ p(r) \leq p'(r) \wedge n(r) \leq n'(r)$. An invariant is not required for this example and the proof of the semilattice conditions can be done mainly automatically by Isabelle/HOL. In fact, for all the easier examples, it was possible to do the majority of the proofs with the automated methods provided by Isabelle/HOL (sledgehammer, auto, ...).

5.2 Verification of Behavior

For the verification of behavioral properties we have developed a small framework, which simplifies the verification and provides two general strategies for verifying a CRDT. The first strategy basically is an induction over the traces. The idea of the second strategy is to show that a CRDT behaves equivalently to

The invariant must hold initially:
$Inv(\{\}, initial_{crdt})$
Merges must preserve the invariant:
$\forall_{H_1,H_2,pl_1,pl_2}\ valid(H_1) \land valid(H_2) \land Inv(H_1, pl_1) \land Inv(H_2, pl_2) \land consistent(H_1, H_2) \Rightarrow Inv(H_1 \cup H_2, merge_{crdt}(pl_1, pl_2))$
Updates must preserve the invariant:
$\forall_{H,pl,r,v,args}\ valid(H) \land Inv(H, pl) \land v = sup_v(H) \Rightarrow Inv(H \cup \{(v(r := v(r) + 1), r, args)\}, update_{crdt}(args, r, pl))$
The invariant must imply the specification:
$\forall_{H,pl,qa}\ valid(H) \land Inv(H, pl) \Rightarrow query_{crdt}(qa, pl) = spec(H, qa)$

Table 6. Verifying behavior of CRDTs

another CRDT which has already been verified. In this paper we only present the first strategy.

When using this strategy, one has to provide an invariant between the payloads and the visible update history. It then has to be shown that the invariant implies the specification, that the invariant holds for the initial payload with the empty update history, and that the invariant is maintained by update- and merge-operations. Table 6 shows the four subgoals.

For both operations our framework provides basic properties about **valid update histories** (predicate *valid*), which hold for all CRDTs. Because we used version vectors for representing the visibility relation, it is not necessary to specify the partial order properties of the relation, but instead it is necessary to specify constraints for the version vectors. The most important property is that the updates on one replica form a total order where the local component of the version vector is always increased by one and the other components increase monotonically. Other properties describe the causality between version vectors in more detail and can be found in [16].

In the case of an update operation one has to show that the invariant is maintained when adding a new update to the top of the update history, meaning that all other updates are visible to the new update. In a real execution this is usually not the case, but the framework can still do this abstraction step, because updates which are not visible do not influence the new update.

In the case of a merge operation one can assume that the invariant holds for two *compatible* update histories with two corresponding payloads, and then has to show that the invariant also holds for the union of the two update histories with the merged payload. Two update histories are *compatible*, when for each replica, the sequence of updates on that replica in one update history is a prefix of the sequence of updates in the other update history.

To verify the counter example we used the following invariant: $Inv(H, (p, n)) \leftrightarrow \forall_r\ p(r) = \sum\{x | \exists_v\ (v, r, x) \in H \land x \geq 0\} \land n(r) = \sum\{-x | \exists_v\ (v, r, x) \in H \land x < 0\}$.

For proving, that a merge-operation preserves the invariant, we have to use the property of *compatible* histories. From this property we get, that for any replica r, we either have $\{x | \exists_v\ (v, r, x) \in H \wedge x \geq 0\} \subseteq \{x | \exists_v\ (v, r, x) \in H' \wedge x \geq 0\}$ or the other way around. This combined with the fact, that all elements are positive, ensures that calculating the maximum yields the correct result. The other parts of the verification, namely update-operations, the initial state and the connection between the invariant and the specification, are rather trivial on paper, whereas in Isabelle the latter requires some work in transforming the sums.

5.3 Evaluation

We used the interactive theorem prover Isabelle/HOL[10] for the verification of several important CRDTs. To this end, we manually translated the pseudo-code implementations from the literature[12,2] into Isabelle functions, and then verified those implementations. The verified CRDTs are the Increment-Only-Counter, PN-Counter, Grow-Set, Two-Phase-Set, a simple and an optimized OR-Set implementation, and a Multi-Value-Register. The theory files are available on GitHub[1].

For the simple data types, the semilattice properties were mostly automatically proved by Isabelle/HOL. For the more complicated data types, like the optimized OR-Set or the similarly implemented MV-register, a suitable invariant had to be found and verified first, which required more manual work in the proofs.

Verifying the behavior of the data types was a more difficult task. Finding a suitable invariant has to be done manually, and the invariant has to be chosen such that it is easy to work with it in Isabelle/HOL. Proving that the invariant is maintained also requires many manual steps, as it usually requires some data transformations which can not be handled automatically by Isabelle/HOL.

We found two small problems, while verifying the CRDTs mentioned above:

– When trying to verify an implementation of the OR-set based on figure 2 in [3], we found a small problem in our translation of this implementation to Isabelle. In the original description the remove-operation computes the set R of entries to be removed with the formula $R = \{(e, n) | \exists n : (e, n) \in E\}$. When this expression is translated to Isabelle code in a direct way, one obtains an equation like $R = \{(e, n).\ \exists n.(e, n) \in E\}$. Then R will always contain all possible entries, because in Isabelle e and n are new variables, and e does not reference the parameter of the function as intended. This problem can be easily fixed, and was not a real issue in the original description, but rather a mistake made in the translation to Isabelle, which happened because of the different semantics of the pseudo-code used in the original description and Isabelle.

[1] `https://github.com/SyncFree/isabelle_crdt_verification`

– We discovered another small problem with the MV-Register presented in specification 10 from [12]. This MV-register is slightly different from the one described in the previous sections, as its assign operation allows to assign multiple values to the register in one step. The problem is in the assign function. When the assigned list of elements is empty, the payload will also be empty after the operation. This is a problem, because all information about the current version is lost. It thus violates the requirement that updates monotonically increase the payload and it can lead to inconsistent replicas. As an example consider the following sequence of operations executed on replica 1: $\{(\bot, [0,0])\} \xrightarrow{Assign(\{a\})} \{(a, [1,0])\} \xrightarrow{Assign(\{b\})} \{(b, [2,0])\} \xrightarrow{Assign(\{\})} \{\} \xrightarrow{Assign(\{c\})} \{(c, [1,0])\}$. Furthermore assume that replica 2 first merges the payload $\{(b, [2,0])\}$ and then the payload $\{(c, [1,0])\}$. Then all updates have been delivered to both replicas, but the payload of replica 1 is $\{(c, [1,0])\}$ and the payload of replica 2 is $\{(b, [2,0])\}$. This problem can be easily fixed by disallowing the assignment of an empty set or by storing the current version in an extra field of the payload.

6 Alternative Specifications

We have already seen two specification techniques: specifications based on the complete update history, and abstract implementations, which are a kind of state-based specifications. Another specification technique was sketched in [1]. In this section we discuss and formalize the technique.

The technique is a state-based one, and uses the notation of pre- and post-conditions to specify the effect of operations on the state. Using the technique of pre- and post-conditions, a sequential specification can be given as a set of Hoare-triples. The Hoare-triple $\{P\}op\{Q\}$ requires that Q should hold after operation *op* whenever P was true before executing the operation.

For example, the increment operation of a counter can be specified by the triple $\{val() = i\}\ inc(x)\ \{val() = i + x\}$ and similarly a set is specified using triples like $\{true\}\ add(e)\ \{contains(e)\}$. Such a sequential specification is applicable to replicated data types if there is no interaction with other replicas between the pre- and post-condition.

In such cases, the replicated counter and the Observed-Remove-Set behave exactly as their corresponding sequential data type. For the Two-Phase-Set this is not true, since an add-operation does not guarantee that the element is in the set afterwards. There are examples like the Multi-Value-Register, where no corresponding and meaningful sequential data type exists, but for replicated data types which try to mimic sequential data types, it is a desirable property to maintain the sequential semantics in time spans where there are no concurrent operations, i.e. where the visibility relation describes a sequence.

In [1] those sequential specifications are combined with concurrent specifications, that describe the effect of executing operations concurrently. The concurrent specification is written in a similar style as the sequential specification.

Instead of only a single operation it considers several operations of the following form executed in parallel: $\{P\} op_1 \parallel op_2 \parallel \cdots \parallel op_n \{Q\}$. The informal meaning is that if P holds in a state s, then executing all operations on s independently and then merging the resulting states should yield a state where Q holds.

Formally, we define a triple $\{P\} op_1 \parallel op_2 \parallel \cdots \parallel op_n \{Q\}$ to be valid, if the following condition is met:

$$
\begin{aligned}
&\forall_{tr,pls,vs,ph,pls',vs',ph',r1,\ldots,r_n,op_1,op_n} : s_{init} \xrightarrow{tr} (pls, vs, ph) \\
&\wedge (pls, vs, ph) \xrightarrow{Update(r_1,op_1);\ldots;Update(r_n,op_n)} (pls', vs', ph') \\
&\wedge \forall_{i \in \{1,\ldots,n\}}\ pls(r_i) = pls(r_1) \\
&\wedge P(pls(r_1)) \Rightarrow Q(merge_{crdt}(pls'(r_1), \ldots, pls'(r_n)))
\end{aligned}
$$

If we reach a state (pls, vs, ph) where the payload on the replicas r_1 to r_n are equal and satisfy the pre-condition P, then executing each operation op_i on replica r_i yields a state (pls', vs', ph') where the post-condition Q holds for the merged payload of replicas r_1 to r_n.

Obviously, one can only specify a subset of all possible executions using this specification techniques. The advantages of this technique is that it is more modular and thus better composable than the technique introduced in Section 3, and that it is easier to see the sequential semantics of the data type. Also, there is the principle of permutation equivalence[1], which can be applied to this technique very easily, and is a good design guideline for designing new CRDTs.

7 Related Work

Burckhardt et al.[5] worked on verifying the behavioral properties of CRDTs. Their techniques are very similar to ours, but they have not used a tool to check their proofs. Their formal system model is more general than ours, as it supports timestamps and visibility relations which are not partial orders.

Bouajjani et al.[4] present a specification technique which is based on the history of updates with the visibility relation. They obtain a more flexible system model by allowing the partial order to be completely different for different operations. This allows them to cover a wide selection of optimistic replication systems, in particular ones that use speculative execution. In contrast to our work, they use an algorithmic approach to reduce the verification problem to model-checking and reachability.

The only other work we are aware of which uses a theorem prover to verify CRDTs is by Christopher Meiklejohn. Using the interactive theorem prover Coq, the semilattice properties of the increase-only-counter and the PN-counter CRDTs[9] are verified. Unlike our work, the behavioral properties of CRDTs are not considered and the verification is not based on a formal system model.

8 Conclusion and Future work

In this paper, we have presented a formal framework for the analysis and verification of CRDTs. As the case studies have shown, it is feasible to verify CRDTs with Isabelle/HOL. The problem found in the MV-register during verification shows that it is easy to miss some corner case when designing a CRDT. The verified CRDTs were given in pseudo-code and then translated to Isabelle, which is a very high level language. Real implementations of the same CRDTs will probably be more complex, and thus the chance of introducing bugs might be even higher. But also the amount of work required for verifying a real implementation is higher.

It is an open question if more research into the automated verification and testing of CRDTs is required. This depends on how applications will use CRDTs in the future. For sequential data types, it is often sufficient to have lists, sets, and maps for managing data, as can be seen in commonly used data formats like XML or JSON. In the case of CRDTs, more data types are required, because different applications require different conflict resolution behavior. This could be very application specific. For example, an application could require a set where add-operations win over remove-operations, but when a remove-operation is performed by an administrator of the system, then that operation should win. If every application needs its own CRDTs, then automatic tools to auto-generate correct code might be a good idea.

In future work, we want to extend the specification techniques presented in this paper for reasoning about applications using CRDTs. Such large-scale distributed applications are usually long-running and should be stable, but are difficult to maintain. It is therefore of special interest to have a stable and correct code base.

Acknowledgement. This research is supported in part by European FP7 project 609 551 SyncFree

References

1. Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In Marcos K. Aguilera, editor, *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 441–442. Springer, 2012.
2. Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.
3. Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.
4. Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. In Jagannathan and Sewell [7], pages 285–296.

5. Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Jagannathan and Sewell [7], pages 271–284.
6. Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
7. Suresh Jagannathan and Peter Sewell, editors. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014.
8. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
9. Christopher Meiklejohn. Distributed data structures with Coq. `http://christophermeiklejohn.com/coq/2013/06/11/distributed-data-structures.html`, June 2013.
10. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
11. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.
12. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.
13. Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.
14. Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183, 1995.
15. Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.
16. Peter Zeller. Specification and Verification of Convergent Replicated Data Types. Master's thesis, TU Kaiserslautern, Germany, 2013.

D Putting Consistency Back into Eventual Consistency

Putting Consistency Back into Eventual Consistency

Valter Balegas
CITI/FCT/Universidade Nova de Lisboa

Sérgio Duarte
CITI/FCT/Universidade Nova de Lisboa

Rodrigo Rodrigues
CITI/FCT/Universidade Nova de Lisboa

Carla Ferreira
CITI/FCT/Universidade Nova de Lisboa

Nuno Preguiça
CITI/FCT/Universidade Nova de Lisboa

Marc Shapiro
INRIA / LIP6

Mahsa Najafzadeh
INRIA / LIP6
[*]

Abstract

Geo-replicated storage systems are at the core of current Internet services. The designers of the replication protocols for these systems have to choose between either supporting low latency, eventually consistent operations, or supporting strong consistency for ensuring application correctness. We propose an alternative consistency model, *explicit consistency*, that strengthens eventual consistency with a guarantee to preserve specific invariants defined by the applications. Given these application-specific invariants, a system that supports explicit consistency must identify which operations are unsafe under concurrent execution, and help programmers to select either violation-avoidance or invariant-repair techniques. We show how to achieve the former while allowing most of operations to complete locally, by relying on a reservation system that moves replica coordination off the critical path of operation execution. The latter, in turn, allow operations to execute without restriction, and restore invariants by applying a repair operation to the database state. We present the design and evaluation of Indigo, a middleware that provides Explicit Consistency on top of a causally-consistent data store. Indigo guarantees strong application invariants while providing latency similar to an eventually consistent system.

[*] This research is supported in part by European FP7 project 609 551 SyncFree (2013—2016), Fundação para a Ciência e Tecnologia SFRH/BD/87540/2012 and PEst-OE/EEI/UI0527/2014.

1. Introduction

To improve the user experience in services that operate on a global scale, from social networks and multi-player online games to e-commerce applications, the infrastructure that supports these services often resorts to geo-replication [8, 9, 11, 21, 23, 24, 37], i.e., maintains copies of application data and logic in multiple datacenters scattered across the globe. This ensures low latency, by routing requests to the closest datacenter, but only when the request does not require cross-datacenter synchronization. Executing update operations without cross-datacenter synchronization is normally achieved through weak consistency. The downside of weak consistency models is that applications have to deal with concurrent operations not seeing the effects of each other, which can lead to non-intuitive and undesirable semantics.

Semantic anomalies do not occur in systems that offer strong consistency guarantees, namely those that serialize all updates [9, 21, 39]. However, these consistency models require coordination among replicas, which increases latency and decreases availability. A promising alternative is to try to combine the strengths of both approaches by supporting both weak and strong consistency for different operations [21, 37, 39]. However, operations requiring strong consistency still incur in high latency. Additionally, these systems make it harder to design applications, as operations need to be correctly classified to guarantee the correctness of the application.

In this paper, we propose *explicit consistency* as an alternative consistency model, in which applications define the consistency rules that the system must maintain as a set of invariants. Unlike models defined in terms of execution orders, explicit consistency is defined in terms of application properties: the system is free to reorder the execution of operations at different replicas, provided that application invariants are maintained.

In addition to proposing explicit consistency, we show that it is possible to implement it while mostly avoid cross-datacenter coordination, even for critical operations that potentially break invariants. To this end, we propose a methodology that, starting from the set of application invariants helps in the deployment of a modified version of the application that includes a set of techniques for precluding invariant violation under concurrency (or, alternatively, use a set of invariant repair actions that recover the service to a desired state). The methodology we propose is composed of the following three steps.

First, based on static analysis, we infer which operations can be safely executed without coordination. Second, for the remaining operations, we provide the programmer with a choice of automatic repair [34] or avoidance techniques. The latter extend escrow and reservation approaches [14, 29, 31, 35], in which a replica reserves the permission to execute a number of operations without coordinating with other replicas. This way we amortize the cost of coordination over multiple requests and move it outside the critical path. Third, after the potentially conflicting operations are identified and the strategy to handle them is chosen, the application code is instrumented with the appropriate calls to our middleware library.

Finally, we present the design of Indigo, a middleware for explicit consistency built on top of a geo-replicated key-value store. Indigo requires the underlying store to provide only properties that have been shown to be efficient to implement, namely per-key linearizability for replicas in each datacenter, causal consistency, and transactions with weak semantics [1, 23, 24].

In summary, this paper makes the following contributions:

- We propose explicit consistency as a new consistency model for application correctness, centered on the application behavior instead of the the order of the execution of operations;
- A methodology that, starting with an application and a set of associated invariants, derives an efficient reservation system to enforce explicit consistency;
- Indigo, a middleware system ensuring explicit consistency on top of a weakly consistent geo-replicated key-value store.

The remaining of the paper is organized as follows: Section 2 introduces explicit consistency; Section 3 gives an overview on the proposed approach to enforce explicit consistency; Section 4 details the analysis for detecting unsafe concurrent operations and Section 5 details the techniques for handling these operations; Section 6 discusses the implementation of Indigo and Section 7 presents an evaluation of the system; related work is discussed in Section 8 and Section 9 concludes the paper with some final remarks.

2. Explicit Consistency

In this section we define precisely the consistency guarantees that Indigo provides. To explain these, we start by defining the system model, and then how explicit consistency restricts the set of behaviors allowed by the model.

To illustrate the concepts, we use as running example the management of tournaments in a distributed multi-player game. The game maintains information about players and tournaments. Players can register and de-register from the game. Players compete in tournaments, for which they can enroll and disenroll. A set of matches occurs for each tournament. A tournament has a maximum capacity. In some cases – e.g., when there are not enough participants – a tournament can be canceled before it starts. Otherwise a tournament's lifecycle is creation, start, and end.

2.1 System model and definitions

We consider a database composed of a set of objects in a typical cloud deployment, where data is fully replicated in multiple datacenters, and partitioned inside each datacenter. For simplicity we assume that the goal of replication is performance, and not fault tolerance. As such, we can assume that replicas do not fail. However, it would be straightforward to handle faults by replacing each machine at a given datacenter with a replica group running a protocol like Paxos [16].

Applications access and modify the database by issuing high-level operations. These operations include a sequence of *read* and *write* operations enclosed in transactions.

We define a database snapshot, S_n, as the value of the database after executing the writes of a sequence of transactions $t_1, \dots, t_n$ in the initial database state, S_{init}, i.e., $S_n = t_n(\dots(t_1(S_{init})))$, with $t_i(S)$ the state after applying the write operations of t_i to S. The state of a replica is the database snapshot that results from executing all committed transactions received in the replica - both local and remote. An application submits a transaction in a replica, with reads and writes executing in a private copy of the replica state. The application may decide to commit or abort the transaction. In the former case, writes are immediately applied in the local replica and asynchronously propagated to remote replicas. In the latter case, the transaction has no side-effect.

The snapshot set $T(S)$ of a database snapshot S is the set of transactions used for computing S - e.g. $T(S_n) = \{t_1, \dots, t_n\}$. We say a transaction t_{i+1} executing in a database snapshot S_i happened-before t_{j+1} executing in S_j, $t_{i+1} \prec t_{j+1}$, iff $T(S_i) \subsetneq T(S_j)$. Two transactions t_{i+1} and t_{j+1} are concurrent, $t_i \parallel t_j$, iff $t_{i+1} \not\prec t_{j+1} \wedge t_{j+1} \not\prec t_{i+1}$ [20].

Happens-before relation defines a partial order among transactions, $O = (T, \prec)$. We say $O_i = (T, <)$ is a valid serialization of $O = (T, \prec)$ if O_i is a linear extension of O, i.e., $<$ is a total order compatible with $\prec$.

Our approach allows transactions to execute concurrently. Each replica can execute transactions according to a different valid serialization. We assume the system guarantees state convergence, i.e., for a given set of transactions T, all valid serializations of $(T, \prec)$ lead to the same database state. Different techniques can be used to this end, from a simple *last-writer-wins* strategy to more complex approaches based on conflict-free replicated data types (CRDTs) [34, 37].

2.2 Explicit consistency

We now define *explicit consistency*, a novel consistency semantics for replicated systems. The high level idea is to let programmers define the application-specific correctness rules that should be met at all times. These rules are defined as invariants over the database state.

In our tournament application, one invariant states that the cardinality of the set of enrolled players in a tournament cannot exceed its capacity. Another invariant is that the enrollment relation must bind players and tournaments that exist - this type of invariant is known as referential integrity in databases. Even if invariants are checked when an operation is executed, in the presence of concurrent operations these invariants can be broken – e.g., if two replicas concurrently enroll players to the same tournament, and the merge function takes the union of the two sets of participants, the capacity of the tournament can be exceeded.

Specifying restrictions over the state: To define explicit consistency, we use first-order logic for specifying invariants as conditions over the state of database. For example, for specifying that the enrollment relation must bind players and tournaments that exist, we could define three predicates: $player(P)$, $tournament(T)$ and $enrolled(P, T)$ to specify that a player P exists, a tournament T exists and that player P is enrolled in tournament T respectively. The condition would then be specified by the following formula: $\forall P, T, enrolled(P, T) \Rightarrow player(P) \wedge tournament(T)$.

Specifying rules over state transitions: In addition to conditions over the current state, we support some forms of temporal specifications by specifying restrictions over state transitions. In our example, we can specify, for instance, that players cannot enroll or drop from a tournament between the start and the end of the tournament.

Such temporal specification can be turned into an invariant defined over the state of the database, by having the application store information that allows for such verification. In our example, when a tournament starts the application can store the list of participants for later checking against the list of enrollments. The rules that forbids enrollment/disenrollment of players can then be specified as $\forall P, T, participant(P, T) \Leftrightarrow enrolled(P, T)$, with the new predicate $participant(P, T)$ specifying that player P participates in active tournament T.

The alternative to this approach would have been to use temporal logics that can specify rules over time [20, 30]. Such approaches would require more complex specification for programmers and a more complex analysis. As our experience has shown that this simpler approach was sufficient for specifying most common application invariants, we have decided to rely on this approach.

Correctness conditions We can now formally define explicit consistency, starting with the helper definition of an invariant I as a logical condition applied over the state of the database. We say that I holds in state S iff $I(S) = true$.

Definition 2.1 (I-valid serialization)**.** For a given set of transactions T, we say that $O_i = (T, <)$ is a I-valid serialization of $O = (T, \prec)$ iff O_i is a valid serialization of O and I holds in the state that results from executing any prefix of O_i.

A system is correct, providing explicit consistency, iff all serializations of $O = (T, \prec)$ are I-valid serializations.

3. Overview

Given the invariants expressed by the programmer, our approach for enforcing explicit consistency has three steps: (i) detect the sets of operations that may lead to invariant violation when executed concurrently (we call these sets *I-offender sets*); (ii) select an efficient mechanism for handling *I-offender sets*; (iii) instrument the application code to use the selected mechanism in a weakly consistent database system.

The first step consists of discovering *I-offender sets*. For this analysis, it is necessary to model the effects of operations. This information should be provided by programmers, in the form of annotations specifying how predicates are affected by each operation [1]. Using this information and the invariants, a static analysis process infers the minimal sets of operation invocations that may lead to invariant violation when executed concurrently (*I-offender sets*), and the reason for such violation. Conceptually, the analysis considers all valid database states and, for each valid database state, all sets of operation invocations that can execute in that state, and checks if executing all these sets in the same state is valid or not. Obviously, exhaustively considering all database states and operation sets would be impossible in practice, which required the use of the efficient verification techniques detailed in section 4.

The second step consists in deciding which approach will be used to handle *I-offender sets*. The programmer

[1] This step could be automated using program analysis techniques, as done for example in [22, 33].

must select from the two alternative approaches supported: *invariant-repair*, in which operations are allowed to execute concurrently and invariants are enforced by automatic conflict resolution rules; *violation-avoidance*, in which the system restricts the concurrent execution of operations that can lead to invariant violation.

In the *invariant-repair* approach, the system automatically guarantees that invariants hold when merging operations executed concurrently, by including the necessary code for restoring invariants in the operations. This is achieved by relying on CRDTs, such as sets, trees and graphs. For example, concurrent changes to a tree can lead to cycles that can be broken using different repair strategies [27].

In the *violation-avoidance* approach, the system uses a set of techniques to control when it is possible and impossible to execute an operation in a datacenter without coordinating with others. For example, to guarantee that an enrollment can only bind a player and a tournament that exist, enrollments can execute in any replica without coordination by forbidding the deletion of players and tournaments. A datacenter can reserve the right to forbid the deletion for a subset of players and tournaments, which gives it the ability to execute enrollments for those players and tournaments without coordinating with other datacenters. Our reservation mechanisms supports such functionality with reservations tailored to the different types of invariants, as detailed in section 5.

Third, the application code is instrumented to use the conflict-repair and conflict-avoidance mechanisms selected by the programmer. This involves extending operations to call the appropriate API functions defined in Indigo.

4. Detecting *I-offender sets*

The language for specifying application invariants is first-order logic formulas containing user-defined predicates and numeric functions. More formally, we assume the invariant is an universally quantified formula in prenex normal form[2]

$$\forall x_1, \cdots, x_n, \varphi(x_1, \cdots, x_n).$$

First-order logic formulas can express a wide variety of consistency constraints, as we exemplify in Section 4.1.

We have already seen that an invariant can use predicates, such as $player(P)$ or $enrolled(P,T)$. Numeric restrictions can be expressed through the use of functions. For example, function $nrPlayers(T)$ that returns the number of players in tournament T, can be used to express that tournaments must have at most five players enrolled: $\forall T, nrPlayers(T) \leq 5$. Invariants can be combined to define the global invariant of an application. For instance, we can have:

$$\begin{aligned} I = \forall P, T, \; & enrolled(P,T) \Rightarrow player(P) \wedge tournament(T) \\ & \wedge \\ & nrPlayers(T) \leq 5 \end{aligned}$$

The programmer does not need to provide an interpretation for the predicates and functions used in the invariant - she just has to write the application invariant and the effects of each operation over the terms of the invariant.

[2] Formula $\forall x, \varphi(x)$ is in prenex normal form if clause φ is quantifier-free. Every first-order logic formula has an equivalent prenex normal form.

Defining operation postconditions To express the effects of operations we use its side-effects, or postconditions, stating what properties are ensured after execution of the operation. Moreover, we take the postcondition to be the conjunction of all side-effects. There are two types of side-effect clauses: predicate clauses, which describe a truth assignment for a predicate (stating whether the predicate is true or false after execution of the operation); and function clauses, which define the relation between the initial and final function values. For example, operation $remPlayer(P)$, which removes player P, has a postcondition with predicate clause $\neg player(P)$, stating that predicate $player$ is false for player P. Operation $enroll(P,T)$, which enrolls player P into tournament T, has a postcondition with two clauses, $enrolled(P,T) \wedge nrPlayers(T) = nrPlayers(T) + 1$. The second clause can be interpreted as a variable assignment, where $nrPlayers(T)$ is increased by one.

The syntax for postconditions is given by the grammar:

$$\begin{array}{lll} post & ::= & clause_1 \wedge clause_2 \wedge \cdots \wedge clause_k \\ clause & ::= & pclause \mid fclause \\ pclause & ::= & p(o_1, o_2, \cdots, o_n) \mid \neg p(o_1, o_2, \cdots, o_n) \\ fclause & ::= & f(o_1, o_2, \cdots, o_n) = exp \oplus exp \\ exp & ::= & n \mid f(o_1, o_2, \cdots, o_n) \\ \oplus & ::= & + \mid - \mid * \end{array}$$

where p and f are predicates and functions respectively, over objects $o_1, o_2, \cdots, o_n$.

Although we imposed that a postcondition is a conjunction, it is possible to deal with operations that have alternative side-effects, by splitting the alternatives between multiple dummy operations. For example, an operation φ with postcondition $\varphi_1 \vee \varphi_2$ could be replaced by operations op_1 and op_2 with postconditions φ_1 and φ_2, respectively.

The fact that postconditions are conjunctions of simple expressions and that predicates and functions are uninterpreted (no interpretation is given), imposes limits on the properties that can be expressed in this setting. For example, it not possible to express reachability properties and other properties over recursive data structures. Nevertheless, the next section shows it is possible to express a wide variety of database consistency properties.

Existential quantifiers So far, the invariants have been formulated as universally quantified formulas. However, some properties require existential quantifiers. For example, to state that tournaments must have at least one player enrolled: $\forall T, tournament(T) \Rightarrow (\exists P, enrolled(P,T))$. In practice the existential quantifier can be replaced by a function, using a technique called skolemization. For this example at hand, we may use function $nrPlayers$ as such: $\forall T, tournament(T) \Rightarrow nrPlayers(T) \geq 1$.

4.1 Expressing Application Invariants

The intrinsic complexity of general invariants makes it difficult to build a comprehensive invariant model. We decided to use a simple model for defining invariants and predicates that still can express significant classes of invariants. This models allows programmers to express invariants in a rather straightforward way, as we exemplify for the following types of invariants.

Uniqueness The uniqueness constraint can be used to express different correctness properties required by applications - e.g. uniqueness of identifiers within a collection. This invariant can be defined using a function that counts the number of elements with a given identifier. For example, the formula $\forall P, \mathit{player}(P) \Rightarrow \mathit{nrPlayerId}(P) = 1$, states that P must have a unique player identifier. A different example of an uniqueness constraint is the existence of a single leader in a collection: $\forall T, \mathit{tournament}(T) \Rightarrow \mathit{nrLeaders}(T) = 1$.

Numeric constraints Numeric constraints refer to numeric properties of the application and set lower or upper-bounds to data values (equality and inequality are special cases). Usually these constraints control the use or access to a limited resource, such as the limited capacity of a tournament exemplified before. Ensuring that a player does not overspend its (virtual) budget can be expressed as: $\forall P, \mathit{player}(P) \Rightarrow \mathit{budget}(P) \geq 0$. Ensuring experienced players cannot participate in beginner's tournaments can be expressed as: $\forall T, P, \mathit{enrolled}(P, T) \wedge \mathit{beginners}(T) \Rightarrow \mathit{score}(P) \leq 30$.

Integrity constraints This type of constraints describes relationships between different objects, known as foreign keys constraints in databases, such as the fact that the enrollment must refer to existing players and tournaments, as exemplified in the beginning of this section. If the tournament application had a score table for players, another integrity constraint would be that every table entry must belong to an existing player: $\forall P, \mathit{hasScore}(P) \Rightarrow \mathit{player}(P)$.

4.2 Determining *I-offender sets*

To detect the sets of concurrent operation invocations that may lead to an invariant violation, we perform a static analysis of the operation's postconditions against invariants. Starting from a valid state, where the invariant is true, if the preconditions hold, the sequential execution of operations always preserve the invariant. However, concurrently executing operations in different replicas may cause a conflict, leading to an invariant violation.

We start by intuitively explaining the process of detecting *I-offender sets*. The process starts by checking operations with opposing postconditions (e.g. $p(x)$ and $\neg p(x)$). Take operations $\mathit{addPlayer}(P)$ with effect $\mathit{player}(P)$ and $\mathit{remPlayer}(P)$ with effect $\neg\mathit{player}(P)$. If these two operations are concurrently executed it is unclear whether player P exists or not in the database. This is an implicit invariant and can be usually addressed choosing a resolution policy (as add-wins).

The process continues by considering, for each invariant, the effects of concurrent executions of multiple operations that affect the invariant: first pairs, then triples, and so forth until all operations are considered or a conflict arises.

To illustrate this process, we use our tournament application and the invariant I presented in the beginning of section 4. For simplicity of presentation, we consider each of the conditions defined in invariant I independently. The first invariant is a numeric restrictions: $\forall T, \mathit{nrPlayers}(T) \leq 5$. In this case, we have to take into account operation $\mathit{enroll}(P, T)$ that affects function $\mathit{nrPlayers}$ and determine if concurrent executions of $\mathit{enroll}(P, T)$ may break the invariant. For that, we substitute in invariant I the operation's effects over function $\mathit{nrPlayers}$. Under the assumption that $\mathit{nrPlayers}(T) < 5$, the weakest precondition ensuring the invariant is not locally broken, we substitute and check whether this results in a valid formula (notation $I\{f\}$ describes the application of formula f in invariant I):

$$\begin{array}{r} I\,\{\mathit{nrPlayers}(T) \leftarrow \mathit{nrPlayers}(T) + 1\} \\ \{\mathit{nrPlayers}(T) \leftarrow \mathit{nrPlayers}(T) + 1\} \\ \mathit{nrPlayers}(T) \leq 5\,\{\mathit{nrPlayers}(T) \leftarrow \mathit{nrPlayers}(T) + 1\} \\ \{\mathit{nrPlayers}(T) \leftarrow \mathit{nrPlayers}(T) + 1\} \\ \mathit{nrPlayers}(T) + 1 \leq 5\,\{\mathit{nrPlayers}(T) \leftarrow \mathit{nrPlayers}(T) + 1\} \\ \mathit{nrPlayers}(T) + 1 + 1 \leq 5 \end{array}$$

The assumption $\mathit{nrPlayers}(T) < 5$ does not ensure the resulting inequality. So, it can be concluded that concurrent executions of operation $\mathit{enroll}(P, T)$ can lead to an invariant violation. For this operation, ensuring locally the (weakest) preconditions does not ensure the invariant will hold globally.

The second invariant of I is $\forall P, T, \mathit{enrolled}(P, T) \Rightarrow \mathit{player}(P)$. In this case we need to detect whether $\mathit{enroll}(P, T)$ and $\mathit{remPlayer}(P)$ lead to an invariant violation. To this end, we substitute the effects of these operations in the invariant and check whether the resulting formula is valid.

$$\begin{array}{r} I\,\{\mathit{enrolled}(P, T) \leftarrow \mathit{true}\}\,\{\mathit{player}(P) \leftarrow \mathit{false}\} \\ \mathit{true} \Rightarrow \mathit{false} \wedge \mathit{Tournament}(T) \\ \mathit{false} \end{array}$$

As the resulting formula is not valid, a set of *I-offenders* is identified: $\{\mathit{enroll}, \mathit{remPlayer}(P)\}$.

We now systematically present the algorithm used to detected *I-offender sets*.

Lemma 4.1 (Conflicting operations)**.** Operations op_1, op_2, $\cdots$, op_n *conflict* with respect to invariant I iff, assuming that I is initially true and preconditions of op_i are initially true ($1 \leq i \leq n$), the result of substituting the postconditions into the invariant is not a valid formula.

Algorithm 1 statically determines the minimal subsets of conflicting (or unsafe) operations. The core of the algorithm is function $\mathit{conflict}(I, s)$ which determines whether the set of operations s break invariant I. This function uses the satisfiability modulo theory (SMT) solver Z3 [10] to verify

Algorithm 1 Algorithm for detecting unsafe operations.

```
Require: $I$ : invariant; $O$ : operations.
 1: $C \leftarrow \emptyset$ {subsets of unsafe operations}
 2: $N \leftarrow \emptyset$ {set of non-idempotent unsafe operations}
 3: $S \leftarrow \emptyset$ {subsets of non-conflicting operations}
 4: for $op \in O$ do
 5:     if $conflict(I, \{op\})$ then
 6:         $N \leftarrow N \cup \{\{op\}\}$
 7:     $S \leftarrow S \cup \{\{op\}\}$
 8: $i \leftarrow 1$
 9: for $s \in S$ and $\#s = i$ and $i < \#O$ do
10:     for $op \in O - s$ and $s \cup \{op\} \notin C$ do
11:         if $conflict(I, s \cup \{op\})$ then
12:             $C \leftarrow C \cup \{s \cup \{op\}\}$
13:         else
14:             $S \leftarrow S \cup \{s \cup \{op\}\}$
15:         $i \leftarrow i + 1$
16: return $C \cup N$
```

the validity of the logical formulas used in Definition 4.1. The function checks first if the operations in s have opposing postconditions (as $addPlayer$ and $remPlayer$). If that check fails, the next step is to submit to the solver a formula obtained by substituting all operations post-conditions in the invariant, and determine its validity.

Algorithm 1 has an initial loop (line 4) to determine which non-idempotent operations cause conflicts over numeric restrictions. The main loop (line 10) iteratively checks if adding a new operation into every possible subset of non-conflicting operations raises a conflict. Each step of the iteration increases the numbers of operations in the subset considered. It starts by determining which pairs of operations conflict. If a conflict is detected, it adds a new operation into the set of unsafe operations. Otherwise, in the next step, it checks whether joining another operation raises any conflict, and so forth. Although not expressed in the algorithm, the operation to be added should affect predicates still not instantiated in the invariant (line 10). The overall complexity of the algorithm is exponential on the number of operations, but this could be improved. Each *I-offender set* determined by the algorithm can be seen as an assignment to the predicates in the invariant that results in a non-valid (invariant) formula. Therefore, we could adapt an (efficient) algorithm for satisfiability module theories, as the ones overviewed in [28].

5. Handling *I-offender sets*

The previous step identifies *I-offender sets*. These sets are reported to the programmer that decides how each situation should be addressed. We now discuss the techniques that are available to the programmer in Indigo.

5.1 Invariant repairing

The first approach that can be used is to allow operations to execute concurrently and repair invariant violation after operations are executed. Indigo has limited support for this approach, which can only address invariants defined in the context of a single database object (which can be as complex as a tree or a graph). To this end, Indigo provides a library of objects that repair invariants automatically with techniques proposed in literature - e.g. sets, maps, graphs, trees with different conflict resolution policies [27, 34].

The programmer still has the opportunity to extend these objects for supporting additional invariants - e.g. it is possible to extend a general set to implement a set with limited capacity n by modifying queries to consider that only n elements exist selected deterministically from all elements in the underlying set [26].

5.2 Invariant-violation avoidance

The alternative approach is to avoid the concurrent execution of operations that would lead to an invariant violation when combining their effects. Indigo provides a set of basic techniques for achieving this.

5.2.1 Reservations

We now discuss the high-level semantics of techniques used to restrict concurrent execution of updates - implementation in weakly consistent stores is addressed in the next section.

UID generator: One important source of potential invariant violations come from the concurrent creation of the same identifier in situations where these identifiers must be unique - e.g. identifier of objects in sets [3, 21]. This problem can be easily solved by splitting the space of identifiers that can be created in each replica. Indigo provides a service that generates unique identifiers by appending to a locally generated identifier a replica-specific suffix. Applications must use this service to generate unique identifiers that are used in operations.

Escrow reservation: For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing decrements to be executed without coordination. Given an initial value for $x = x_0$, there are initially $x_0 - k$ rights to execute decrements. These rights can be split by different replicas. For executing $x.decrement(n)$, the operation must acquire and consume n rights to decrement x in the replica it is submitted. If not enough rights exist in the replica, the system will try to obtain additional rights from other replicas. If this is not possible, the operation will fail. Executing $x.increment(n)$ creates n rights to decrement n initially assigned to the replica in which the operation that executes the increment is submitted.

A similar approach is used for invariants of the form $x \leq k$, with increments consuming rights and decrements creating new rights. For invariants of the form $x+y+\ldots+z \geq k$, a single escrow reservation is used, with decrements to any of the involved variables consuming rights and increments creating rights. If a variable x is involved in more than one invariant, several escrow reservations will be affected by a single increment/decrement operation on x.

Multi-level lock reservation: When the invariant in risk is not numeric, we use a multi-level lock reservation (or simply multi-level lock) to restrict the concurrent execution of operations that can break invariants. A multi-level lock can provide the following rights: (i) *shared forbid*, giving the shared right to forbid some action to occur; (ii) *shared allow*, giving the shared right to allow some action to occur; (iii) *exclusive allow*, giving the exclusive right to execute some action.

When a replica holds some right, it knows no other replica holds rights of a different type - e.g. if a replica holds a *shared forbid*, it knows no replica has any form of *allow*. We now show how to use this knowledge to control the execution of *I-offender sets*.

In the tournament example, $\{enroll(P,T), remPlayer(P)\}$ is an *I-offender set*. We can associate a multi-level lock to one of the operations, for specific values of the parameters. For example, we can have a multi-level lock associated with $remPlayer(P)$, for each value of P. For executing $remPlayer(P)$, it is necessary to obtain the right *shared allow* on the reservation for $remPlayer(P)$. For executing $enroll(P,T)$, it is necessary to obtain the right *shared forbid* on the reservation for $remPlayer(P)$. This guarantees that enrolling some player will not execute concurrently with deleting the player, but concurrent enrolls or concurrent deletes can occur. If all replicas hold the *shared forbid* right on removing players, the most frequent enroll operation can execute in any replica without coordination with other replicas.

The *exclusive allow* right is necessary when an operation is incompatible with itself, i.e., when executing concurrently the same operation may lead to an invariant violation.

Multi-level mask reservation: For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates false may lead to an invariant violation if all other predicates were originally false. In our analysis, each of these pairs is an *I-offender set*.

Using simple multi-level locks for each pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of the other operation in all pairs. In this case, for executing one operation is suffices to guarantee that a single other operation is forbidden (assuming that the predicate associated with the forbidden operation is true).

To this end, Indigo includes a multi-level mask reservation that maintains the same rights as multi-level lock regarding a set of K operations. With multi-level mask, when obtaining a *shared allow* right for some operation, it is necessary to obtain (if it does not exist already) a *shared forbid* right on some other operation. These operations are executed atomically by our system.

5.2.2 Using Reservations

Our analysis outputs *I-offender sets* and the invariant that can be broken if operations execute concurrently. For each *I-offender set*, the programmer must select the type of reservation to be used - based on the invariant type that can be broken, a suggested reservation type is generated.

Even when using the same type of reservations for each *I-offender set*, it is possible to prevent the concurrent execution of *I-offender sets* using different sets of reservations - we call this a reservation system. For example, consider our tournament example with the following two *I-offender sets*:

$$\{enroll(P,T), remPlayer(P)\}$$
$$\{enroll(P,T), remTournament(P)\}$$

Given these *I-offender sets*, two different reservation systems can be used. The first system includes a single multi-level lock associated with $enroll(P,T)$, with $enroll(P,T)$ having to obtain a *shared allow* right to execute, while both $remPlayer(P)$ and $remTournament(T)$ would have to obtain the *shared forbid* right to execute. The second system includes two multi-level lock associated with $remPlayer(P)$ and $remTournament(T)$, with enroll having to obtain the *shared forbid* right in both to execute.

Indigo runs a simple optimization process to decide which reservation system to use. As generating all possible systems may take too long, this process starts by generating a small number of systems using the following heuristic algorithm: (i) select a random *I-offender set*; (ii) decide the reservation to control the concurrent execution of operations in the set, and associate the reservation with the operation: if a reservation already exists for some of the operations, use the same reservation; otherwise, generate a new reservation from the type previously selected by the user; (iii) select the remaining *I-offender set*, if any, that has more operations controlled by existing reservations and repeat the previous step.

For each generated reservations system, Indigo computes the expected frequency of reservation operations needed using as input the expected frequency of operations. The optimization process tries to minimize this expected frequency of reservation operations.

After deciding which reservation system will be used, each operation is extended to acquire and release the necessary rights before and after executing the code of the operation. For escrow locks, an operation that consumes rights will acquire rights before its execution and these rights will not be released in the end. Conversely, an operation that creates rights will create these rights after its execution.

6. Implementation

In this section, we discuss the implementation of Indigo as a middleware running on top of a causally consistent store. We first explain the implementation of reservations and how they are used to enforce explicit consistency. We conclude by explaining how Indigo is implemented on top of an existing geo-replicated store.

6.1 Reservations

Indigo maintains information about reservations as objects stored in the underlying causally consistent storage system. For each type of reservation, a specific object class exists. Each reservation instance maintains information about the rights assigned to each of the replicas - in Indigo, each datacenter is considered a single replica, as explained later.

The escrow lock object maintains the rights currently assigned to each replica. The following operations can be submitted to modify the state of the object: *escrow_consume* depletes rights assigned to the local replica; *escrow_generate* generates new rights in the local replica; *escrow_transfer* transfers rights from the local replica to some given replica. For example, for an invariant $x \geq K$, *escrow_consume* must be used by an operation that decrements x and *escrow_generate* by operations that increment x.

When an operation executes in the replica where it is submitted, if insufficient rights are assigned to the local replica, the operation fails and has no side-effects. Otherwise, the state of the replica is updated accordingly and the side-effects are asynchronously propagated to the other replicas, using the normal replication mechanisms of the underlying storage system. As operations only deplete rights of the replica where they are submitted, it is guaranteed that every replica has a conservative view of the rights assigned to it - all operations that have consumed rights are known, but any operations that transferred new rights from some other replica may still have to be received. Given that the execution of operations is linearizable in a replica, this approach guarantees the correctness of the system in the presence of any number of concurrent updates in different replicas and asynchronous replication, as no replica will ever consume more rights than those assigned to it.

The multi-level lock object maintains which right (exclusive allow, shared allow, shared forbid) is assigned to each replica, if any. Rights are obtained for executing operations with some given parameters - e.g. in the tournament example, for removing player P the replica needs a *shadow allow* right for player P. Thus, a multi-level lock object manages the rights for the different parameters independently - a replica can have a given right for a specific value of the parameters or a subset of the parameter values. For simplicity, in our description, we assume that a single parameter exists.

The following operations can be submitted to modify the state of the multi-level lock object: *mll_giveRight* gives a right to some other replica - a replica with a shared right can give the same right to some other replica; a replica that is the only one with some right can change the right type and give it to itself or to some other replica; *mll_freeRight* revokes a right assigned to the local replica. As a replica can have been given rights by multiple concurrent *mll_giveRight* operations executed in different replicas, *mll_freeRight* internally encodes which *mll_giveRight* operations are being revoked. This is necessary to guarantee that all replicas converge to the same state.

As with escrow lock objects, each replica has a conservative view of the rights assigned to it, as all operations that revoke the local rights are always executed initially in the local replica. Additionally, assuming causal consistency, if the local replica shows that it is the only replica with some right, that information is correct system-wide. This condition holds despite concurrent operations and asynchronous propagation of updates, as any *mll_giveRight* executed in some replica is always propagated before a *mll_freeRight* in that replica. Thus, if the local replica shows that no other replica holds any right that is because no *mll_giveRight* has been executed (without being revoked).

The multi-level mask object maintains the information needed for a multi-level mask reservation by combining several multi-level lock objects. The operation *mlm_giveRight* allows to give rights for one of the specified multi-level locks.

6.2 Indigo middleware

We have built a prototype of Indigo on top of a geo-replicated data store with the following properties: (i) causal consistency; (ii) support for transactions that access a database snapshot and merge concurrent updates using CRDTs [34]; (iii) linearizable execution of operations for each object in each datacenter. It has been shown that all these properties can be implemented efficiently in geo-replicated stores and at least two systems support all these functionalities: SwiftCloud [42] and Walter [37]. Given that SwiftCloud has a more extensive support for CRDTs, which are fundamental for invariant-repair, we decided to build Indigo prototype on top of SwiftCloud.

Reservation objects are stored in the underlying storage system and they are replicated in all datacenters. Reservation rights are assigned to datacenters individually, which keeps the information small. As discussed in the previous section, the execution of operations in reservation objects must be linearizable (to guarantee that two concurrent transactions do not consume the same rights).

The execution of an operation in the replica where it is submitted has three phases: i) the reservation rights needed for executing the operation are obtained - if not all rights can be obtained, the operation fails; ii) the operation executes, reading and writing the objects of the database; iii) the used rights are released. For escrow reservations, rights consumed are not released; new rights are created in this phase. The side-effects of the operation in the data and reservation objects are propagated and executed in other replicas asynchronously and atomically.

Reservations guarantee that operations that can lead to invariant violation do not execute concurrently. However, operations need to check if the preconditions for operation execution hold before execution[3]. In our tournament example, an operation to remove a tournament cannot execute before removing all enrolled players. Reservations do not guarantee that this is the case, but only that a remove tournament will not execute concurrently with an enrollment.

An operation needs to access a database snapshot compatible with the used reservation rights, i.e., a snapshot that reflects the updates executed before the replica has acquired the rights being used. In our example, for removing a tournament it is necessary to obtain the right that allows such operation. This precludes the execution of concurrent enroll operations for that tournament. After the tournament has been deleted, an enroll operation can obtain a forbid right on tournament removal. For correctness, it is necessary that the operation observes the tournament as deleted, which is achieved by enforcing that updates of an operation are atomic and that the read snapshot is causally consistent (obtaining the forbid right necessarily happens after revoking the allow right, which happens after deleting the tournament). These properties are guaranteed in Indigo directly by the underlying storage system.

Obtaining reservation rights The first and last phases of operation execution obtain and free the rights needed for operation execution. Indigo provides API functions for obtaining and releasing a list of rights. Indigo tries to obtain the necessary rights locally using ordered locking to avoid deadlocks. If other datacenters need to be contacted for obtaining some reservation rights, this process is executed before start obtaining rights locally. Unlike the process for obtaining rights in the local datacenter, Indigo tries to obtain the needed rights from remote datacenters in parallel for minimizing latency. This approach is prone to deadlocks - if some remote right cannot be obtained, we use an exponential backoff approach that frees all rights and tries to obtain them again after an increasing amount of time.

When it is necessary to contact other datacenters to obtain some right, latency of operation execution is severely affected. In Indigo, reservation rights are obtained pro-actively using the following strategy. Escrow lock rights are divided among datacenters, with a datacenter asking for additional rights to the datacenter it believes has more rights (based on local information). Multi-level lock and multi-level mask rights are pre-allocated to allow executing the most common operations (based on the expected frequency of operations), with shared allow and forbid rights being shared among all datacenters. In the tournament example, *shared forbid* for removing tournaments and players can be owned in all datacenters, allowing the most frequent enroll to execute locally.

The middleware maintains a cache of reservation objects and allows concurrent operations to use the same shared (allow or forbid) right. While some ongoing operation is using a shared or exclusive right, the right cannot be revoked.

[3] This step could be automated by inferring preconditions from invariants and operation side-effects, given that the programmer specifies the code for computing the value of predicates

6.3 Fault-tolerance

Indigo builds on the fault-tolerance of the underlying storage system. In a typical geo-replicated store, data is replicated inside a datacenter using quorums or relying on a state-machine replication algorithm. Thus, the failure of a machine inside a datacenter does not lead to any data loss.

If a datacenter (fails or) gets partitioned from other datacenters, it is impossible to transfer rights from and to the partitioned datacenter. In each partition, operations that only require rights available in the partition can execute normally. Operations requiring rights not available in the partition will fail. When the partition is repaired (or the datacenter recovers with its state intact), normal operation is resumed.

In the event that a datacenter fails losing its internal state, the rights held by that datacenter are lost. As reservation objects maintain the rights held by all replicas, the procedure to recover the rights lost by the datacenter failure is greatly simplified - it is only necessary to guarantee that recovery is executed only once with a state that reflects all updates received from the failed datacenter.

7. Evaluation

This section presents an evaluation of Indigo. The main question our evaluation tries to answer is how does explicit consistency compares against *causal consistency* and *strong consistency* in terms of latency and throughput with different workloads. Additionally, we try to answer the following questions:

- Can the algorithm for detecting *I-offender sets* be used with realistic applications?
- What is the impact of an increasing the amount of contention in objects and reservations?
- What is the impact of using an increasing number of reservations in each operation?
- What is the behavior when coordination is necessary for obtaining reservations?

7.1 Applications

To evaluate Indigo, we used the two following applications.

Ad counter The ad counter application models the information maintained by a system that manages ad impressions in online applications. This information needs to be geo-replicated for allowing fast delivery of ads. For maximizing revenue, an ad should be impressed exactly the number of times the advertiser is willing to pay for. This invariant can be easily expressed as $nrImpressions(A_i) \leq K_i$, with K_i the maximum number of times ad A_i should be impressed and the predicate $nrImpressions(A_i)$ returning the number of times it has been impressed. In a real system, when a client application asks for a new ad to be impressed, some complex logic will decide which ad should be impressed.

Advertisers will typically require ads to be impressed a minimum number of times in some countries - e.g. ad A should be impressed 10.000 times, including 4.000 times in US and 4.000 times in EU. This example is modeled by having the following additional invariants for specifying the limits on the number of impressions (impressions in excess in Europe and US can be accounted in $nrImpressionsOther$):

$$nrImpressionsEU(A) \leq 4000$$
$$nrImpressionsUS(A) \leq 4000$$
$$nrImpressionsOther(A) \leq 2000$$

We modeled this application by having independent counters for each ad and region. Invariants were defined with the limits stored in database objects:

$$nrImpressions((region, ad)) \leq targetImpressions((region, ad))$$

A single update operation that increments the ad tally was defined - this operation updates the predicate $nrImpressions$. Our analysis shows that the increment operation conflicts with itself for any given counter, but increments on different counters are independent. Invariants can be enforced by relying on escrow lock reservations for each ad.

Our experiments used workloads with a mix of: a read only operation that returns the value of a set of counters selected randomly; an operation that reads and increments a randomly selected counter. Our default workload included only increment operations.

Tournament management This a version of the application for managing tournaments described in section 2 (and used throughout the paper as our running example), extended with read operations for browsing tournaments. The operations defined in this application are similar to operations that one would find in other management applications such as courseware management.

As detailed throughout the paper, this application has a rich set of invariants, including uniqueness rules for assigning ids; generic referential integrity rules for enrollments; and order relations for specifying the capacity of each tournament. This leads to a reservation system that uses both escrow lock and multi-level lock reservation objects. Three operations do not require any right to execute - add player, add tournament and disenroll tournament - although the latter access the escrow lock object associated with the capacity of the tournament. The other update operations involve acquiring rights before they can execute.

In our experiments we have run a workload with 82% of read operations (a value similar to the TPC-W shopping workload), 4% of update operations requiring no right for executing, and 14% of update operations requiring rights (8% of the operations are enrollment and disenrollments).

7.1.1 Performance of the Analysis

We have implemented the algorithm described in Section 4 for detecting *I-offender sets* in Java, relying on the satisfiability modulo theory (SMT) solver Z3 [10] for verifying invariants. The algorithm was able to find the existing *I-offender sets* in the applications. The average running time of this process in a recent MacBook Pro laptop was 19 ms for the ad counter applications and 2892 ms for the more complex tournament application.

We have also modeled TPC-W - the invariants in this benchmark are a subset of those of the tournament application. The average running time for detecting *I-offender sets* was 937 ms. These results show that the running time increases with the number of invariants and operations, but that our algorithm can process realistic applications.

7.2 Experimental Setup

We compare Indigo against three alternative approaches:

Causal Consistency (Causal) As our system was built on top of causally consistent SwiftCloud system[42], we have used unmodified SwiftCloud as representative of a system providing causal consistency. We note that this system cannot enforce invariants. This comparison allows us to measure the overhead introduced by Indigo.

Strong Consistency (Strong) We have emulated a strongly consistent system by running Indigo in a single DC and forwarding all operations to that DC. We note that this approach allows more concurrency than a typical strong consistency system as it allows updates on the same objects to proceed concurrently and be merged if they do not violate invariants.

Red-Blue consistency (RedBlue) We have emulated a system with Red-Blue consistency [21] by running Indigo in all DCs and having red operations (those that may violate invariants and require reservations) execute in a master DC, while blue operations execute in the closest DC respecting causal dependencies.

Our experiments comprised 3 Amazon EC2 datacenters - US-East, US-West and EU - with inter-datacenter latency presented in Table 1. In each DC, Indigo servers run in a single m3.xlarge virtual machine with 4 vCPUs and 8 ECUs of computational power, and 15GB of memory available. Clients that issue transactions run in up to three m3.xlarge machines. Where appropriate, we placed the master DC in US-East datacenter to minimize the communication latency and have those configurations perform optimally.

RTT (ms)	US-E	US-W
US-West	81	-
EU	93	161

Table 1. RTT Latency among Datacenters in Amazon EC2

7.3 Latency and throughput

We start by comparing the latency and throughput of Indigo with alternative deployments for both applications.

We have run the ad counter application with 1000 ads and a single invariant for each ad. The limit on the number of impressions was set sufficiently high to guarantee that the limit is not reached. The workload included only update operations for incrementing the counter. This allows us to

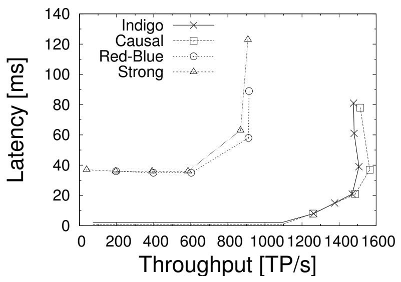

Figure 1. Peak throughput (ad counter application).

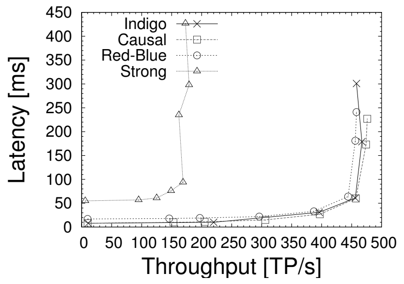

Figure 2. Peak throughput (tournament application).

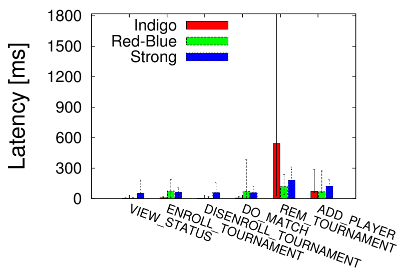

Figure 3. Average latency per op. type - Indigo (tournament app.).

measure the peak throughput when operations are able to obtain reservations in advance. The results are presented in Figure 1, and show that Indigo achieves throughput and latency similar to a causally consistent system. Strong and RedBlue results are similar, as all update operations are red and execute in the master DC in both configurations.

Figure 2 presents the results when running the tournament application with the default workload. As before, results show that Indigo achieves throughput and latency similar to a causally consistent system. In this case, as most operations are read-only or can be classified as blue and execute in the local datacenter, RedBlue throughput is only slightly worse than that of Indigo.

Figure 3 details these results presenting latency per operation type (for selected operations) in a run with throughput close to the peak value. The results show that Indigo exhibits lower latency than RedBlue for red operations. These operation can execute in the local DC in Indigo, as they require either no reservation or reservations that can be shared and are typically locally available.

Two other results deserve some discussion. *Remove tournament* requires canceling shared forbid rights acquired by other DCs before being able to acquire the shared allow right for removing the tournament, which explain the high latency. Sometimes latency is extremely high (as shown by the line with the maximum value) - this is a result of the asynchronous algorithms implemented and the approach for requesting remote DCs to cancel their rights, which can fail when a right is being used. This could be improved by running a more elaborate protocol based on Paxos. *Add player* has a surprisingly high latency in all configurations. Analyzing the situation, we found out that the reason for this lies in the fact that this operation manipulates very large objects used to maintain indexes - all configurations have a fix overhead due to this manipulation.

7.4 Micro-benchmarks

Next, we examine the impact of key parameters.

Increasing contention Figure 4 shows the throughput of the system with increasing contention in the ad counter application, by varying the number of counters in the experiment. As expected, the throughput of Indigo decreases when contention increases as several steps require executing operations sequentially. Our middleware introduces additional contention when accessing the cache. As the underlying storage system also implements linearizability per-object, it is also possible to observe its throughput also decreases with increased contention, although more slowly.

Increasing number of invariants Figure 5 presents the results of ad counter application with an increasing number of invariants - from one to three. In this case, the results show that the peak throughput with Indigo decreases while latency keeps constant. The reason for this is that for escrow locks, each invariant has an associated reservation object - thus, when increasing the number of invariants the number of updated objects also increases, with impact on the operations that each datacenter needs to execute. To verify our explanation, we have run a workload with operations that access the same number of counters in the weak consistency configuration - the presented results show the same pattern for decreased throughput.

Behaviour when transferring reservations Figure 6 shows the latency of individual operations executed in US-W datacenter in the ad counter application for a workload where increments reach the invariant limit for multiple counters. When rights do not exist locally, Indigo cannot mask the latency imposed by coordination - in this case, for obtaining additional rights from the remote datacenters.

In Figure 3 we have shown the impact of obtaining a multi-level lock shared right that requires revoking rights present in all other replicas. We have discussed this problem and a possible solution in section 7.3. Nevertheless, it is important to note that such big impact in latency is only experienced when it is necessary to revoke shared forbid rights in all replicas before acquiring the needed shared allow right. The positive consequence of this approach is that enroll operations requiring the shared forbid right that was shared by all replicas execute with latency close to zero. The maximum latency line in enroll operation shows the maximum latency experienced when a replica acquires a shared forbid right from a replica already holding such right.

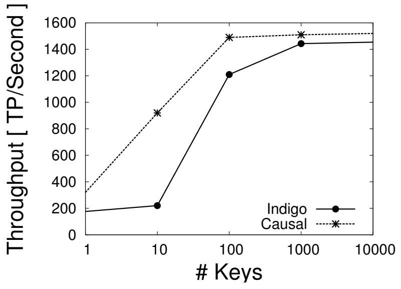

Figure 4. Peak throughput with increasing contention (ad counter application).

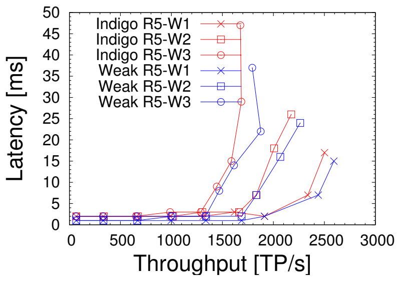

Figure 5. Peak throughput with an increasing number of invariants (ad counter application).

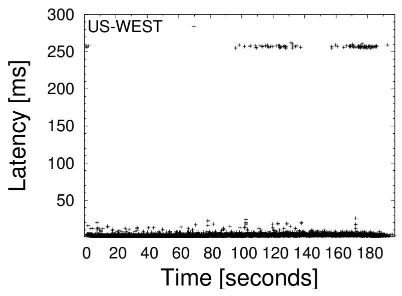

Figure 6. Latency of individual operations of US-W datacenter (ad counter application).

8. Related work

Geo-replicated storage systems Many cloud storage systems supporting geo-replication emerged in recent years. Some [1, 11, 19, 23, 24] offer variants of eventual consistency, where operations return right after being executed in a single datacenter, usually the closest one to the end-user to improve response times. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [1, 2, 13, 23]; supporting limited transactions where a set of updates are made visible atomically [4, 24]; supporting application-specific or type-specific reconciliation with no lost updates [6, 11, 23, 37], etc. Indigo is built on top of a geo-replicated store supporting causal consistency, a restricted form of transactions and automatic reconciliation; it extends those properties by enforcing application invariants.

Eventual consistency is insufficient for some applications that require (some operations to execute under) strong consistency for correctness. Spanner [9] provides strong consistency for the whole database, at the cost of incurring coordination overhead for all updates. Transaction chains [43] support transaction serializability with latency proportional to the latency to the first replica accessed. MDCC [18] and Replicated Commit [25] propose optimized approaches for executing transactions but still incur in intra-datacenter latency for committing transactions.

Some systems tried to combine the benefits of weak and strong consistency models by supporting both. In Walter [37] and Gemini [21], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [38] and Pileus [39] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [8] and DynamoDB [36] also combine weak consistency with per-object strong consistency relying on conditional writes, where a write fails in the presence of concurrent writes. Indigo enforces explicit consistency rules, exploring application semantics to let (most) operations execute in a single datacenter.

Exploring application semantics Several works have explored the semantics of applications (and data types) for improving concurrent execution. Semantic types [15] have been used for building non serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [34] explore commutativity for enabling the automatic merge of concurrent updates, which Walter [37], Gemini [21] and SwiftCloud [42] use as the basis for providing eventual consistency. Indigo goes further by exploring application semantics to enforce application invariants that can span multiple objects.

Escrow transactions [29] offer a mechanism for enforcing numeric invariants under concurrent execution of transactions. By enforcing local invariants in each transaction, they can guarantee that a global invariant is not broken. This idea can be applied to other data types, and it has been explored for supporting disconnected operation in mobile computing [31, 35, 40]. The demarcation protocol [5] is aimed at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [14] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages.

Indigo builds on these works, but it is the first to provide an approach that, starting from application invariants expressed in first-order logic leads to the deployment of the appropriate techniques for enforcing such invariants in a geo-replicated weakly consistent data store.

Other related work Bailis et al. [3] studied the possibility of avoiding coordination in database systems and still maintain application invariants. Our work complements that, addressing the cases that cannot entirely avoid coordination, yet allow operations to execute immediately by obtaining the required reservations in bulk and anticipation.

Others have tried to reduce the need for coordination by bounding the degree of divergence among replicas. Epsilon-serializability [32] and TACT [41] use deterministic algorithms for bounding the amount of divergence observed by an application using different metrics - numerical error, order error and staleness. Consistency rationing [17] uses a statistical model to predict the evolution of replicas state and allows applications to switch from weak to strong consistency on the likelihood of invariant violation. In contrast to these works, Indigo focuses on enforcing invariants efficiently.

The static analysis of code is a standard technique used extensively for various purposes [7, 12**?**], including in a context similar to ours. Sieve [22] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a Red-Blue system [21]. In [33], the authors present an analysis algorithm that describes the semantics of transactions. These works are complementary to ours, and the proposed techniques could be used to automatically infer application side-effects. The latter work also proposes an algorithm to allow replicas to execute transactions independently by defining conditions that must be met in each replica. Whenever an operation cannot commit locally, a new set of conditions is computed and installed in all replicas using two-phase commit. In Indigo, replicas can exchange rights peer-to-peer.

9. Conclusions

This paper proposes an application-centric consistency model for geo-replicated services - explicit consistency - where programmers specify the consistency rules that the system must maintain as a set of invariants. We describe a methodology that helps programmers decide which invariant-repair and violation-avoidance techniques to use to enforce explicit consistency, extending existing applications. We also present the design of Indigo, a middleware that can enforce explicit consistency on top of a causally consistent store. The results show that the modified applications have performance similar to weak consistency for most operations, while being able to enforce application invariants. Some rare operations that require intricate rights transfers exhibit high latency. As future work, we intend to improve the algorithms for exchanging reservation rights on those situations.

References

[1] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. . URL `http://doi.acm.org/10.1145/2465351.2465361`.

[2] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. . URL `http://doi.acm.org/10.1145/2463676.2465279`.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination-avoiding database systems. *CoRR*, abs/1402.2237, 2014.

[4] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *ACM SIGMOD Conference*, 2014.

[5] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994. ISSN 1066-8888. . URL `http://dx.doi.org/10.1007/BF01232643`.

[6] Basho. Riak. `http://basho.com/riak/`, 2014. Accessed Oct/2014.

[7] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=1454159.1454167`.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387905`.

[10] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08, pages 337–340. Springer, 2008.

[11] 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`.

[12] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 12 1998.

[13] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. . URL `http://doi.acm.org/10.1145/2523616.2523628`.

[14] ed Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, nsdi'14, Berkeley, CA, USA, 2014. USENIX Association.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983. ISSN 0362-5915. . URL `http://doi.acm.org/10.1145/319983.319985`.

[16] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006. ISSN 0362-5915. . URL `http://doi.acm.org/10.1145/1132863.1132867`.

[17] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=1687627.1687657`.

[18] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. . URL `http://doi.acm.org/10.1145/2465351.2465363`.

[19] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010. ISSN 0163-5980. . URL `http://doi.acm.org/10.1145/1773912.1773922`.

[20] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/177492.177726`.

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387906`.

[22] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `http://dl.acm.org/citation.cfm?id=2643634.2643664`.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. . URL `http://doi.acm.org/10.1145/2043556.2043593`.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2482626.2482657`.

[25] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=2536360.2536366`.

[26] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, pages 471–480, Oct 2012.

[27] S. Martin, M. Ahmed-Nacer, and P. Urso. Abstract unordered and ordered trees crdt. *CoRR*, abs/1201.1784, 2012.

[28] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[29] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986. ISSN 0362-5915. . URL `http://doi.acm.org/10.1145/7239.7265`.

[30] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, FOCS, pages 46–57. IEEE, 1977.

[31] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM. . URL `http://doi.acm.org/10.1145/1066116.1189038`.

[32] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, Dec. 1995. ISSN 1041-4347. . URL `http://dx.doi.org/10.1109/69.476504`.

[33] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014. URL `http://arxiv.org/abs/1403.2307`.

[34] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24549-7. URL `http://dl.acm.org/citation.cfm?id=2050613.2050642`.

[35] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc. ISBN 3-540-89855-7. URL `http://dl.acm.org/citation.cfm?id=1496950.1496954`.

[36] S. Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. . URL `http://doi.acm.org/10.1145/2213836.2213945`.

[37] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA,

2011. ACM. ISBN 978-1-4503-0977-6. . URL http://doi.acm.org/10.1145/2043556.2043592.

[38] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. . URL http://doi.acm.org/10.1145/224056.224070.

[39] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. . URL http://doi.acm.org/10.1145/2517349.2522731.

[40] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7153-X. URL http://dl.acm.org/citation.cfm?id=829520.830874.

[41] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251229.1251250.

[42] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR*, abs/1310.3107, 2013.

[43] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8.

E Towards Verifying Eventually Consistent Applications

Towards Verifying Eventually Consistent Applications

Burcu Kulahcioglu Ozkan Erdal Mutlu Serdar Tasiran
Koç University
{bkulahcioglu,ermutlu,stasiran}@ku.edu.tr

1. Introduction

Modern cloud and distributed systems depend heavily on replication of large-scale databases to guarantee properties like high availability, scalability and fault tolerance. These replicas are maintained in geographically distant locations to be able to serve clients from different regions without any loss of performance. Ideally, these systems require to achieve immediate availability while preserving strong consistency in the presence of network partitions. But unfortunately, the CAP theorem [1] proves that it is impossible to have all these properties together in a distributed system. For this reason, architects of current distributed systems frequently omit strong consistency guarantees in favor of weaker forms of consistency, commonly called eventual consistency[2].

The basic guarantee that eventual consistency model provides, is that: "if all update requests stop, after a period of time all replicas of the database will converge to be logically equivalent"[3]. Today "eventual consistency" became a common term for proposed different forms of weak consistency models [4–7]. Each of these work proposes consistency models that provides different weak guarantees and features. With the absence of a uniform specification formalism on the eventual consistency guarantees, the development and usage of eventually consistent systems became very challenging for the programmers. There are different solutions proposed for making weak consistency model more programmer-friendly. While solutions like [7, 8] try to define new replicated data types for programming weak consistency, other solutions [4, 9] try to solve the same problem by defining new programming languages and programming models. These solutions try to solve the programmability issues by hiding some of the non-determinism exposed by the eventual consistency models. The main sources of the non-determinism in such systems are the asynchronous message passing and the weak guarantees. Depending on the guarantees given by the weak consistency models, there can be inconsistencies among replicas. These different sources of non-determinism make the problem more challenging than shared-memory concurrent programs and general message-passing programs. In the presence of such different sources of non-determinism, it is important to provide good debugging and verification tool support to the programmers.

In this proposal, we aim to investigate application level specifications and develop verification techniques (both static and dynamic) for applications running on eventually consistent systems and using replicated data types.

PAPEC'14, April 13-16, 2014, Amsterdam, Netherlands.
Copyright © 2014 ACM 978-1-4503-2716-9/14/04…$15.00.
http://dx.doi.org/10.1145/2596631.2596638

2. Motivation

Due to relaxed guarantees of eventual consistency, applications running on such systems allow for a wider set of program behaviors than a traditional application running on strong consistency. Depending on the nondeterminism induced by consistency level, an operation can/cannot see the updates of its own session, may read and operate on stale data or may receive concurrent updates in different orderings. Thus, application programmers should take into account different possible execution scenarios that can happen in eventually consistent systems.

Consider a business-to-business e-commerce application which allows its clients (stores) to view catalogs of products and place orders. The application can be accessed using computers or mobile devices and it is built on top of an eventually consistent system. In addition to high availability, supporting eventual consistency enables store employees to place orders even if his device is in offline mode. The application aims to guarantee that: (i) If the product is out of stock, the client will be informed and no shipping will be processed (ii) The budget of a client is updated after each shipment (iii) If an order is cancelled, neither shipment nor budget update is performed. (iv) Every order submitted to the system is eventually processed (v) The quantity of a product in stock is always non-negative. (vi) The budget of a client is always non-negative.

Writing specification (i.e. invariants, assertions etc.) for applications running on eventually consistent distributed systems is non-trival. An application such as the one explained above, will contain different implementation levels (i.e client-side, server-side) which will make it challenging to write specifications. Figure 1 shows a general model for the application interactions between different implementation layers. Depending on the specification of the application, different properties can be checked on different implementation levels and on different variables. For instance, properties related to single client can be checked with local variables on client level whereas application wide properties have to be defined over the eventual global variables on database or server level.

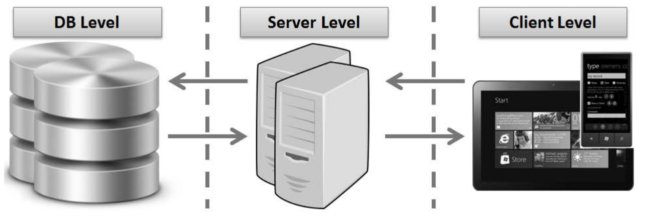

Figure 1. Different implementation layers in an application for eventually consistent systems

Using the specification language provided by the state-of-the-art programs verification tools, we can attempt to write application specifications as follows (assuming each client and order has unique ids):

(i) If the product is out of stock, the client will be informed and no shipping will be processed. This can be stated as a post-condition to processOrder:

$(ensures\ \ checkStock(o.qty) \leq 0 \Longrightarrow clientInformed[o]\ \&\&\ !orderShipped[o])$

(ii) The budget of a client is updated after each shipment. This can be stated as a post-condition to makeShipment:

$(ensures\ \ orderShipped[o] \Longrightarrow clientBudget[o.client] == (old(clientBudget[o.client]) - o.totalCost))$

(iii) If an order is cancelled, neither shipment nor budget update is performed. As a post-condition to cancelOrder:

$(ensures\ \ orderCancelled[o] \Longrightarrow (clientBudget[o.client] == old(clientBudget[o.client]))\&\&!orderShipped[o])$

(iv) Every order submitted to the system is eventually processed. (Note that the following statement cannot express *eventuality*.)

$(invariant\ \ \forall\, Order\ o;\ o.id \Longrightarrow processed[o])$

(v) The quantity of a product in stock is always non-negative.

$(invariant\ \ \forall\, Product\ p;\ stockQty[p] \geq 0)$

(vi) The budget of a client is always non-negative.

$(invariant\ \ \forall\, Client\ c;\ clientBudget[c] \geq 0)$

However, these specifications are not expressive enough to state the necessary information on the properties that an eventually consistent application must satisfy. It is not clear in these specifications whether they are stated on the replica-local state or eventual global state. For instance, some applications may keep some data in the client layer (see Figure 1) and its operations can specify restrictions on client state. On the other side, some specifications must hold on to the eventual global state reached in the server layer. Similarly, an application may have seperate invariants for local and global states, that need to hold in specific cases. Besides, it might be helpful for a specification to make use of some *ghost variables* that does not exist in the programmer's code but appear in the program annotations. As given in the specifications (ii) and (iii), the interpretation of $old(variable)$ indicating the value of a data object before an operation is also not clear in the eventually consistent setting. Another point is that the requirements specifying a liveness property of eventual processing such as (iv) cannot be represented in such program verification specification languages. A spefication language of a verification tool for eventually consistent systems needs to be extended so that it is capable of stating expressions vital for eventually consistent applications.

While developing such an eventually consistent application, the programmer should consider many non-trivial cases that may arise from the eventual transmission of the updates. For instance, he should define how the system behaves (i) when a client who has not received the shipment information yet, cancels an order or (ii) when a store employee makes an order in offline mode and the order is duplicated by another employee or (iii) two concurrent orders processed successfully in the replicas they are submitted to, but their sum exceeds the number of available products in stock. Some sequence of operations might require the use of (eventually consistent) transactions to provide isolation and atomicity. Furthermore, there might be critical operations (such as updating the store budget) that might need stronger guarantees (main copy for updating some replicated data objects or providing stronger consistency levels for some specific operations, etc.).

The nondeterminism in eventual consistency models makes it harder to reason whether a program behaves as intended. Moreover, eventual consistency is a novel concept and programmers are not used to think in this relaxed semantics. Therefore, there is a need for verification of application-level guarantees of such programs.

3. Our Approach

The example system in Section 2, shows that there are many questions to consider in the design of an eventually consistent application and hence a wide range of properties that an application must satisfy. Yet, the concept of eventual consistency is too broad and it is not explicit how to specify these properties.

Based on this example system and the discussion of its specifications, we aim to investigate (i) how to state these application specifications(including assertions, invariants, and temporal specifications) and how to interpret their semantics (ii) how we can build static and dynamic techniques for verifying given application specifications.

We plan to represent the eventual consistency model presented in [13] in a formal specification written using a programming-language-like formalism (such as PlusCal (+CAL) [14], TLA+ [15], Boogie [16], VCC's input language of annotated C [20], or Spec# [18]). This formal representation will then set a common basis for the development of dynamic and static tools for the eventually consistent applications.

Dynamic verification tools aim to verify the correctness of a program under test with respect to its specifications by exploring the execution space and observing its behaviors. Different than testing, dynamic verification techniques employ efficient model checking algorithms over the possible execution space. There are different approaches proposed to systematically [10, 11] or randomly [12] explore this execution space efficiently.

In this proposal, we aim to build a dynamic verification tool for characterizing and exploring possible behaviors of programs written for eventually consistent systems using replicated data types. We plan to build our dynamic verification technique upon a formal mathematical representation of the formalization and specifications defined for eventually consistent systems and replicated data types in [13]. With such a mathematical representation, we can make the assumption of having a correct implementation of the eventually consistent system and replicate data types. Than, we will employ a systematical testing algorithm (i.e. CHESS [10]) for exploring all possible execution order of an input program.

Static verification tools analyze the source code of the program without compiling or executing it. The state-of-the-art powerful verification tools [17–20], use deductive methods that take a program together with its specification, generate verification conditions (in the form of first order logical statements) and prove given specifications using theorem provers.

Our proposal to build a static verification tool for eventually consistent applications will base on constructing a mechanism on top of an existing tool (which is unaware of eventual consistency) so that it can reason on and detect problems in eventually consistent programs with replicated data types. This requires to encode the semantics of the system (formalized by a mathematical language) so that the abstracted system models all possible executions with respect to the system's consistency guarantees. Then, the tool will verify whether a user program satisfies the necessary specifications considering all possible concurrent operations. This approach is successfully applied for the transactional systems running under relaxed semantics such as snapshot isolation [21].

References

[1] Brewer, E. A.: Towards robust distributed systems. In: Proc. PODC '00. ACM, New York, USA. (2000)

[2] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing update conflicts in bayou, a weakly connected replicated storage system. SIGOPS Oper. Syst. Rev. 29. (1995)

[3] Kawell Jr., L., Beckhardt, S., Halvorsen, T., Ozzie, R., Greif, I.: Replicated document management in a group communication system. In: Proc. the 1988 ACM Conference on Computer-supported Cooperative Work: 395. (1988)

[4] Conway, N., Marczak, W.R., Alvaro, P., Hellerstein, J.M., Maier, D.: Logic and lattices for distributed programming. In: Proc. SoCC'12. ACM, NY, USA (2012)

[5] Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazons highly available key-value store. In: Symposium on Operating Systems Principles. (2007)

[6] Lloyd, W., Freedman, M. J., Kaminsky, M. and Andersen, D. G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: Proc. SOSP '11. ACM, New York, NY, USA. (2011)

[7] Shapiro, M., Preguia, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proc. SSS'11. Springer-Verlag, Berlin, Heidelberg. (2011)

[8] Burckhardt, S., Fhndrich, M., Leijen, D., and Wood, B. P.: Cloud types for eventual consistency. In: Proc. ECOOP'12. Springer-Verlag, Berlin, Heidelberg. (2012)

[9] Alvaro, P., Conway, N., Hellerstein, J., Marczak, W.: Consistency analysis in Bloom: a CALM and collected approach. In CIDR'11. Asilomar, CA, USA.(2011)

[10] Musuvathi, M., Qadeer, S., and Ball, T.: CHESS: A Systematic Testing Tool for Concurrent Software. Tech. Rep. MSR-TR-2007-149, (2007).

[11] Godefroid, P.: Model Checking for Programming Languages Using VeriSoft. In: Proc. POPL '97, ACM, New York, USA. (1997)

[12] Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. ASPLOS XV, ACM (2010)

[13] Burckhardt, S., Gotsman, A., Yang, H.: Understanding Eventual Consistency. Tech. Rep. MSR-TR-2013-39. (2013)

[14] L. Lamport. The +cal algorithm language. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, pages 5–5, July 2006. .

[15] Leslie Lamport. Tla in pictures. *IEEE Trans. Software Eng.*, 21(9):768–775, 1995.

[16] Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Formal Methods for Components and Objects, 4th International Symposium, FMCO (2005)

[17] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 02, New York, NY, USA, ACM Press. (2002)

[18] Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. CASSIS04, Berlin, Heidelberg, Springer-Verlag. (2005)

[19] Fahndrich, M.: Static verification for code contracts. In: Proceedings of the 17th international conference on Static analysis. SAS10, Berlin, Heidelberg, Springer-Verlag. (2010)

[20] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: Vcc: Contract-based modular verification of concurrent C. In: ICSECompanion 2009. (2009)

[21] Kuru, I., Kulahcioglu Ozkan, B., Mutluergil, S.O., Tasiran, S., Elmas, T., Cohen, E.: Verifying Programs under Snapshot Isolation and Similar Relaxed Consistency Models In: 9th ACM SIGPLAN Workshop on Transactional Computing. (2014)