An interactive tool for demonstrating CRDTs

October 24, 2017

1 Introduction

Conflict-free Replicated Data Types (also Convergent and Commutative Replicated Data Types, short CRDT) are data types for managing replicated data. They can be updated on multiple servers in parallel and independently from other replicas. Updates can be sent to other replicas asynchronously and each CRDT has a different built-in method to handle (potentially conflicting) updates when they are received.

Example: MV-Register One example of a CRDT is the Multi-Value register (MV-Register). It has only one update operation, which is called `assign` which assigns a value to the register. Reading a Multi-value Register returns a list of all concurrently assigned values, which have not been overridden by other assignments. We assume that the returned list is ordered by the lexicographic order on the byte representation of the values. In the initial state reading the register returns the empty list.

Visualization To explain the behavior of CRDTs, it is useful to consider concrete examples like the one shown in Figure 1. Here, each of the horizontal lines represents one replica and the grey arrows between the replicas denote delivery of messages. The visualization shows the assignments performed and the values returned by reads.

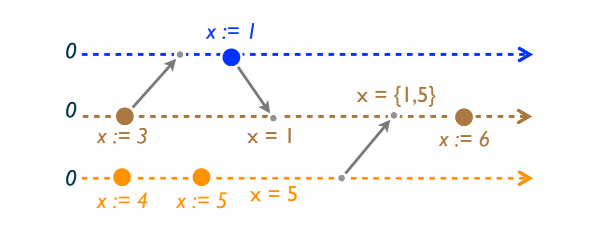

Figure 1: Execution of a MV-Register, drawn in the Shapiro style.

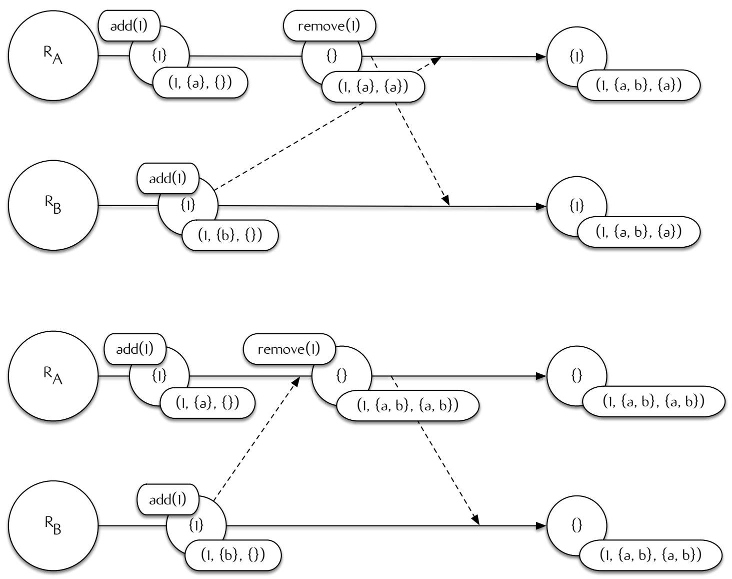

Figure 2: Executions of an Add-Wins Set, drawn in the Meiklejohn style.

Another form of visualization is shown in Figure 2. This visualization shows the operations performed, the current value of the data type, and the internal state of the implementation.

2 Task

Your task in this project is to build an interactive tool for demonstrating the semantics of CRDTs with visualizations. The tool will be used to improve the current documentation of the data types used in Antidote (`http://syncfree.github.io/antidote/crdts.html`). Therefore, it should be possible to embed the tool in a web page and it should run completely client side. Also, it must be possible to embed the visualizer into the Antidote documentation, which is written using Markdown.

Users of the tool should be able to specify an execution, run their own executions and see how results are changing. For users interested in the internals of the implementation, it should be possible to see the internal states and downstream messages.

The tool should support all data types currently supported by Antidote and it should be easy to integrate additional data types at a later point in time.

Antidote Jupyter Notebook

October 24, 2017

The Jupyter Notebook (`http://jupyter.org/`) is a web application for creating interactive documents. The document contains code examples which can be executed in the notebook. The result of the computation is displayed directly in the document. A Notebook is driven by a server-side component, which is called a Kernel. There already are many Kernels for different languages, which can be used or adapted for this project.

Task

Your task in this project is to develop an interactive tutorial for Antidote as an interactive Jupyter notebook. Readers of the document should be able to experiment with the Antidote API by changing the code examples.

In the project you also have to address the following points:

Controlling Datacenters Users should see, how Antidote behaves in the case of partitions and how concurrent updates are resolved by Antidote. Therefore users should be able to artificially disconnect data centers and reconnect them again with commands in the Jupyter notebook.

Deployment It should be easy to run a notebook locally. Ideally users just have to run `docker-compose up` to get the notebook running in Docker containers.

Managing a causally consistent materializer

October 24, 2017

1 Introduction

Antidote[1] is a geo-replicated key-value database that allows non-blocking concurrent updates. The persistent storage in Antidote is an ‘append-only log’ that records each update operation that is executed. A materialized view, called *snapshot*, of a key can be generated by applying all updates to that key, starting from an initial snapshot. The materialized view is generated by a component called *Materializer*.

For example, consider a key K representing a counter. The persistent log has 3 records with $\{K$, increment$\}$, each representing an increment to key K. The materialized view of K is generated by applying the increment operation 3 times, starting with a value 0, resulting in value 3.

A persistent log may have several updates to different keys. Hence, to generate a snapshot for a particular key K, the materializer must filter the relevant updates and throw away other updates while reading from the log. Reading the entire log and filter the operations for each read request is obviously not efficient.

2 Task

The task of this project is to implement a Materializer for Antidote that can generate snapshots efficiently, by caching operations and snapshots in-memory. The implementation must be done in Java, with the interface and constraints given in the following section. The implementation must be well tested with unit tests and evaluated with performance tests. To this end, you will be provided with realistic execution traces from Antidote which can be used for testing.

3 Interface

We expect you to implement the follwoing interfaces:

[1] `http://antidotedb.org`

```
interface Materializer {
  /** Retrieves the materialized snapshot of object under key that is prior to and including vc */
  Object getSnapshot(Key key, Vectorclock vc);

  /**  adds operations that were committed at time vc */
  void putOps(Map<Key, List<Object>> operations, Vectorclock vc);

  /** confirms that versions with timestamp older than vc will never be requested
    * (and can be garbage-collected) */
  void setVcLowerBound(Vectorclock vc);
}

interface Vectorclock extends Map<ByteString, Integer>, Comparable<Vectorclock> {
}

interface Key {
  ByteString getBucket();
  ByteString getKey();
  CrdtType getCrdtType();
}

interface CrdtType<State, Operation> {
  /** returns the initial state of the CRDT */
  State initialState();

  /** applies an operation to the CRDT state and returns the new state.
    * The old state might be changed and should no longer be used. */
  State applyOperation(State state, Operation op);
}
```

3.1 Constraints

1. To construct a snapshot, the operations have to be applied in a causally consistent order:
   - If `vc(op1) < vc(op2)`, then `op1` has to be applied before `op2`.
   - If `vc(op1) = vc(op2)` then the operations were added in the same `put_ops` call and should be executed in the order they appeared in the list of operations.
2. To construct a snapshot the **`new`** and `update` functions from the CRDT module can be used.
3. `get_snapshot(key, vc)` will produce a snapshot using all operations with clock $\leq$ `vc`.
4. There can be concurrent calls to the materializer, and it should provide a linearizable behavior.
5. When the function call `put_ops` returns, the operations must be persistently stored (i.e. they must be available after a crash and restart).

3.2 Assumptions

1. When `get_snapshot` is called with clock `vc`, then no consequent call to `put_ops` with the same key will have a vector clock less than or equal to `vc`.

2. The function `put_ops` takes a list of key-operations pairs, where the operations are a list of downstream-effects as produced by the CRDT library. The keys are distinct.

   `put_ops` is never called twice for the same vector clock.

3. `put_ops` is called in a causally consistent order: When a call of method `put_ops(... op1 ..., vc1)` is executed before `put_ops(... op2 ..., vc2)`, then executing `op1` before `op2` is always correct. However, it can happen that `vc2 > vc1`. In this case, the operations are actually concurrent, so they can be executed in any order with the same result. The vector clocks do not capture concurrency on a single data center and order concurrent operations arbitrarily.

3.3 Nonfunctional Requirements

Performance goals:

- Low latency of < 10 ms for reads and writes (write acknowledgement guarantees persistence)
- High throughput of > 10.000 reads/s

A Performance Benchmark for Antidote

October 24, 2017

1 Introduction

During the development of the Antidote database, changes applied to the code have an impact on the correctness of the data store (e.g. by fixing bugs), but also on the performance. The ultimate goal is to have a correct system that is at the same time as performant as possible. But the impact of code modifications on the overall performance of the system is not easy to measure using regression tests. These tests mainly check the correctness of the modifications. Non-functional properties, such as access latencies or throughput, have to be evaluated by running benchmarks.

Existing solutions There are already some standardized benchmarks (e.g. the Yahoo! Cloud Serving Benchmark[1]) and systems for developing custom benchmarks (e.g. Basho Bench[2]). The main goal of the Yahoo! Cloud Serving Benchmark is to compare the performance different data stores. But for comparing the performance of two versions of the Antidote data store (e.g. one with and one without some new optimizations), the measures taken might not help in diagnosing the speed-up of specific parts of the system.

Basho Bench allows to create custom benchmarks. A driver needs to be implemented that allows the benchmark to communicate with the system under test (in this case Antidote). The load generated by the benchmark can be specified in a configuration file and there are some parameters to tune, which operations are executed by the benchmark. This configuration needs to be given by the person executing the benchmark and the quality of benchmark configurations influences the comparability of the results. Additionally, Basho Bench does not support natively to execute a benchmark on different versions of the same system. This has to be done manually by building and starting each version and then executing the benchmark on both versions and compare the results manually.

[1]`https://github.com/brianfrankcooper/YCSB/wiki`

[2]`https://github.com/basho/basho_bench`

2 Task

The task of this project is to develop a benchmarking solution for Antidote that allows to compare different versions of Antidote. The versions will be given in form of git commits/branches in the Antidote Github repository. The work-load generated should be suitable to compare the performance of different parts of Antidote (e.g. write-operation-focused, read-operation-focused, cross-data-center-performance). You will need to find suitable measures to compare different versions/branches. When comparing several versions of Antidote, the tool should generate a result file visualizing the comparison in a suitable way. The benchmark tool should take as input a commit-id/branch name of the Antidote GitHub repository and should otherwise be fully automated. The tool must be extensible with new functionality resulting in additional tools, for example a tool for comparing two branches and showing an overview of the performance differences. Another use case can be performance monitoring over time as done by the Firefox project under `https://arewefastyet.com/`.