Correctness of a New Distributed Database Synthesis Protocol Using SMT Solvers

Kiarash Rahmani

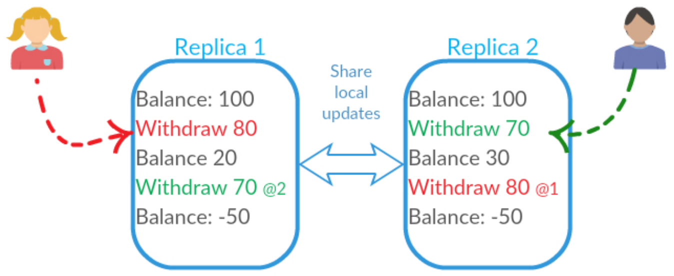
Computer-Aided Program Reasoning Course Project - Dr. Samanta `rahmank@purdue.edu`
`https://github.com/Kiarahmani/CAPR2016`

Abstract. As a result of the CAP theorem, today's distributed databases offer various levels of *weak consistency*. Application developers must pick the correct consistency level for each operation to preserve the correctness of their application. Recent works[1] successfully take this burden off the developer's shoulders. They correctly choose the weakest store offered consistency, that preserves application-level requirements. We are now planning to extend these works by actually **implementing** required consistency guarantees.

Keywords: Eventual Consistency, SMT solvers, Synthesis, Weak Consistency, Quelea, Cassandra

1 Introduction

Replication is a must in the real-world web services such as those maintained by Facebook, Google or Twitter. These systems replicate data and logic across geo-distributed data centers (and also within data centers) to reduce user-perceived latency and to tolerate network partitioning and node failures.
As the running example, let's consider a replicated bank account software, that serves the clients via the host data center. In an ideal world, clients would submit request to the local replica, updates are created locally and *would become available at all other replicas immediately.* However, we know this is not the case in reality: physical network partitions are unavoidable and delay in update propagation is non-negligible.
It is easy to see why this can become problematic, if replicas do synchronize operation execution:

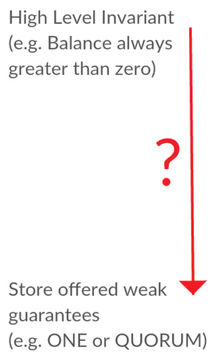


If two users concurrently submit updates on the same object (for example, in the above figure the green transaction was submitted at replica 2 before the updates from red transaction became available at replica 2), the system might become inconsistent. This is due to the *dependency of operations to the ordered history of updates available at that replica.* For example, the Withdraw operation in the bank account, reads the current history of updates and decides if the money should be deducted or not.

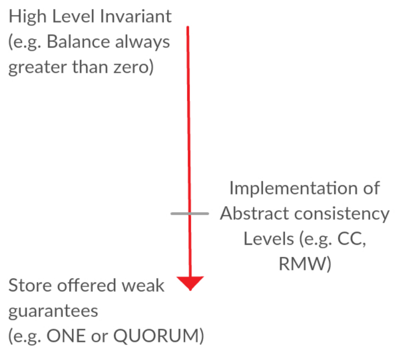
Weak Consistency Levels: Above examples demonstrate the need for synchronization between replicas. However, always keeping the replicated data consistent, requires global consensus and massive communication between data stores, which reduces the overall performance. One might ask, is it even necessary in all cases to guarantee **S**trong **C**onsistency for all operations? For example, in the bank account example, we can permit the Deposit operations to perform and commit locally and the application still would be safe. In other words concurrent deposits cannot falsify the correctness of the bank account application.
In the above setting, the least that we must be sure of is that *all updates become available at all replicas eventually*. This is a called **E**ventual **C**onsistency and is usually offered by the underlying system. **Weak Abstract Consistency** levels are those that fill the gap between EC and SC guarantees. For example, **C**ausal **C**onsistency, guarantees that if an update is present in the replica, all updates that it witnessed at the time of execution -possibly at some other replica- are also present. Or **R**ead **M**y **W**rites, guarantees that an operation always witnesses all the previous updates from its own session.

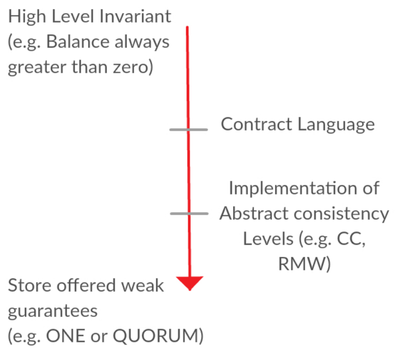
2 Problem Definition

Real world data-stores, do not offer any of the conceptually defined weak guarantees such as RMW or CC. They only let the users work with low-level notions like TWO (A write must be written on at least two node). As we can see there is a large conceptual gap here: the application writer only cares about the high-level invariant of the program (non-negative balance), but has to pick store-offered guarantees for each operation to preserve the invariant.



As explained earlier this gap is filled by conceptual guarantees like CC, that offer one level of abstraction for the application writers: now users can pick one of these more abstract guarantees that have been previously implemented on top of the store. We call them more abstract, because they consider updates and relationships among them, as opposed to low-level inter replica agreements.





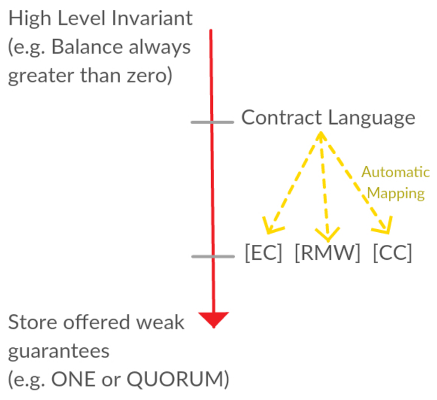
Recent developments, offer another level of abstraction: using a language of **contracts**, user can specify the *application level requirements* for each operations, and operations are automatically mapped to the correct available consistency implementation. For example following contract is mapped to CC:

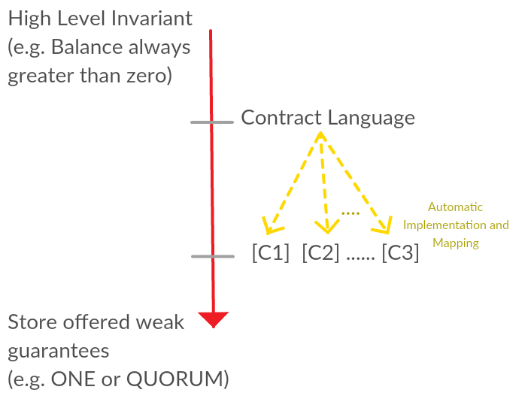
$$\forall(a : deposit)(b : withdraw).((vis(a, b)) \wedge (vis(b, \eta)) \Rightarrow (vis(a, \eta)))$$

It simply says that if the current operation η witnesses an update b, and b has witnessed another update a, it must be the case that η also witnesses a.

This obviously makes the developers job easier, since he or she does not have to deal with the low-level consistency guarantees anymore.
However, there are still some problems remained...

Proposed Extension Current system, based on the given contract *maps* each operation to some *pre-implemented* consistency level. Since the mapped level must preserve the contract's requirements, the system tends to be conservative: If $L1$ does not preserve the contract for the operation op but $L2$ does, op must be mapped to $L2$. However, $L2$ might be too strong for op and we lose performance because of unnecessary synchronization.
What if we could first analyze the code and the contracts and automatically synthesis the consistency levels that offer the exact guarantees required for each operation.




High Level Invariant
(e.g. Balance always greater than zero)
Contract Language
....
Automatic Implementation and Mapping
[C1] [C2] ...... [C3]
Store offered weak guarantees
(e.g. ONE or QUORUM)

3 Proposed Approach

Well-Formed Contracts: Our first observation is that we can limit the structure of the contracts, without losing any generality. All interesting contracts can in fact, be written in the following form:

$$\forall a, \Phi_{pre} \Rightarrow vis(a, \eta)$$

which specifies that, all updates that satisfy the pre-condition Φ_{pre} must be visible to the current operation η. This intuitively makes sense, since developer is only interested in specifying what updates must be witnessed by the operation that is being executed.

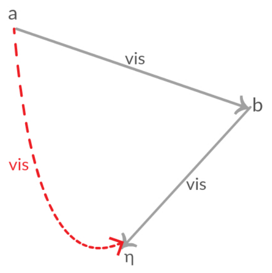
Cache Synthesis: Our idea is to first analyze the contracts and create multiple caches on top of the store, each of which is guaranteed to show the specified behavior by the contract. For example if the contract of an operation is: $\forall a, so(a, \eta) \Rightarrow vis(a, \eta)$, a cache is created for this contract that makes sure that the operation is executed only if $so^{-1}(\eta)$ is present in the cache (can be done by blocking the execution, until the required updates become available)

3.1 First Steps

We observed that contracts can be divided into two major groups and we deal with each separately:

Wait-Free Contracts: are those that never require blocking the operation: the cache can pick some subset of the available update and make only them visible to the current operation. For example, consider the following contract and its associated graph:

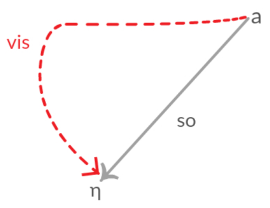
$$\forall a.vis(a, b) \wedge vis(b, \eta) \Rightarrow vis(a, \eta)$$



The contract specifies that if an effect b is visible to the current operation, all updates a that were visible to b, must also be visible to the current operation. However, this does not mean the current operation must be blocked until all such updates are received: we can filter those bs that does not satisfy this property and then execute the current operation. In other words, we by filtering some present updates at the replica, the current operation can see a *safe environment* based on its contract.
The important property that our filter must have, is called MAX-VIS and is defined as follows:
if the cache filters out some updates and returns S as the safe environment, there cannot be an update u in the store, that $S \cup \{u\}$ is safe.

Waiting Contracts: Similarly, there are some contracts, that when implemented in the cache, waiting for some updates would be unavoidable:

$$\forall a.so(a, \eta) \Rightarrow vis(a, \eta)$$



This specifies that every previous update in the same session with the current operation, must be visible to this operation. This means that the current operation must be blocked until those updates become available at the local replica. Similar to the previous section, we define MIN-WAIT property for the blocking, as follows:
if cache blockes the current operation, until a set s of updates are arrived, there cannot be a smaller set s', that $R \cup \{s'\}$ is safe (R is the effects available at the replica).

3.2 next steps?

Now we have a well defined problem: How can cache decide the following?

1. if the contract is wait-free or not.
2. what updates must become visible to the current operation for wait-free contracts, satisfying MAX-VIS
3. what updates must be waited for while the operation is blocked that satisfies MIN-WAIT.

For the first question, by analyzing the associated graph for each contract, we realized that for all the wait-free contracts, the pre-condition part (gray section in above figures) ends with a *vis* edge pointing to the current operation. This intuitively makes sense, because a *vis* edge brings flexibility to the system: we can manipulate the pre-condition of the contract, by modifying the set of effects that are visible to the current operation. On the other hand, Waiting contracts always have an *so* edge pointing to the current operation: *so* is the session order relation, which cannot be modified by the system; as a result pre-condition cannot be changed and wait is unavoidable.

For the next two problems, we believe they can be translated into MAX-SAT problems and fed into an automatic solver for the result. If the results are promising, we are going to pack everything into a tool, that by analyzing the contracts and with the help of SAT-solvers, can guarantee the *minimum required synchronization among replicated data stores.*

References

1. KC Sivaramakrishnan, G. Kaki, S. Jagannathan: Declarative Programming over Eventually Consistent Data Stores. PLDI '15.
2. S. Burckhardt et. al: Replicated Data Types: Specification, Verification, Optimality. POPL'14.
3. S. Gilbert, N. Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51-59, June 2002.
4. M. Lesani, C. Bell, A. Chlipala: Chapar: Certified Causally Consistent Distributed Key-Value Stores. POPL'16.
5. G.DeCandia et.al. : Dynamo: Amazon's Highly Available Key-value Store. SOSP'07.