Towards a Proof Framework for Information Systems with Weak Consistency

Peter Zeller and Arnd Poetzsch-Heffter

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

Abstract. Weakly consistent data stores are more scalable and can provide a higher availability than classical, strongly consistent data stores. However, it is much harder to reason about and to implement applications, when the underlying infrastructure provides only few guarantees. In this paper, we report on work in progress on a proof framework, which can be used to formally reason about the correctness of such applications. The framework supports the verification of functional properties, which go beyond the guarantees given by the data store and can cover relations between multiple interactions with clients and invariants between several objects. Additionally, we modeled and support modern database features, like causal consistency, snapshot-transactions, and conflict-free replicated data types (CRDTs). The framework and the proofs are developed within the interactive theorem prover Isabelle/HOL.

1 Introduction

Today, many information systems are built without a strongly consistent data store. There is a variety of reasons for this trend: For services which are offered world-wide, the concept of Geo-Replication allows for low latency in all regions, by replicating data at servers, which are geographically close to the users. However, Geo-Replication does not work well with the concepts of strong consistency. In particular, distributed transactions are incompatible with low latency and high availability [6]. Mobile applications have problems comparable to Geo-Replicated systems. Since the network connection is sometimes slow or unavailable, it is not feasible to use strong consistency to synchronize data between mobile devices and cloud services.

Programming applications using weak consistency is inherently complex. Most importantly, convergence must be ensured, meaning that all replicas represent the same abstract state when they have observed the same set of operations, without losing writes. To help programmers handle this problem, conflict-free replicated data types (CRDTs) [14] have been developed. A CRDT is a reusable data type, which embodies a certain strategy to handle concurrent updates. Examples are counters, sets, and maps. When an application is written using CRDTs, the convergence property comes for free and thus the development effort is reduced.

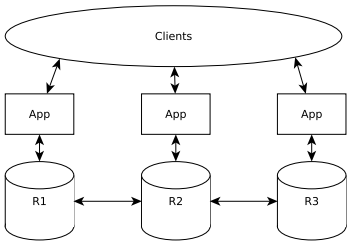

Fig. 1. System architecture

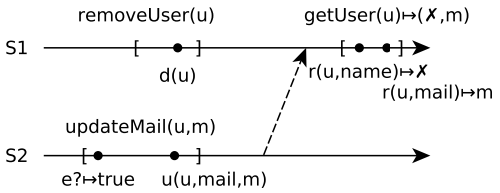

Fig. 2. Problematic Execution

However, convergence is not the only desirable property of an application. It is also important that concurrent updates are handled in a way that makes sense for the application (see Section 2). These correctness properties are often overlooked by developers. One reason for this is that there is no systematic method to reason about the correctness of an implementation. While there are multiple program logics for working with sequential and concurrent programs, there are no frameworks yet, which support reasoning about eventual consistency and CRDTs on a higher level. Thus, it is not feasible to use existing frameworks to reason about nontrivial correctness properties of these kinds of applications.

To make the verification practical, our work aims to considerably reduce the required proof work. We are developing a proof framework in Isabelle/HOL [13], which captures commonalities of applications, which are built on top of weakly consistent data stores with replicated data types. With the common verification tasks lifted to the framework level, the verification of a concrete application can be done on a higher level and focus on the application specific properties and invariants. We discuss our approach to verification in Section 3.

2 Developing applications with weak consistency

To show the need to reason about causal consistency and the choice of data types, we consider a small application to manage user accounts. This application provides the following API to clients:

The function `registerUser(name, email)` creates a new user account with the given data and returns the unique identifier of the newly created user. To update the mail address of a user with a given identifier, there is a function `updateMail(id, newMail)`. To remove a user from the system, `removeUser(id)` can be called. The data of a user can be retrieved via `getUser(id)`, which returns a record with the name and the mail address of a given user, or `not_found` when the user does not exist.

We assume an architecture similar to the one shown in Fig. 1 on which we want to implement our application. At the bottom there are several (full) replicas of the database, which asynchronously synchronize their states. At the top there is a set of clients, for which we do not have any additional assumptions.

```
def registerUser(name, mail) {
  result = newId()
  atomic {
    users[result]['id'].write(result)
    users[result]['name'].write(name)
    users[result]['mail'].write(mail)
  }
  return result
}

def updateMail(id, newMail) {
  atomic {
    val exists = users[id].exists()
    if exists {
      users[id]['mail'].write(newMail)
    }
  }
}

def removeUser(id) {
  users[id].delete()
}

def getUser(id) {
  atomic {
    val exists = users[id].exists()
    if (exists) {
      val name = users[id]['name'].read()
      val mail = users[id]['mail'].read()
      result = {'name': name, 'mail': mail}
    } else {
      result = not_found
    }
  }
  return result
}
```

Fig. 3. Pseudocode implementation of example application to manage user accounts.

In particular clients might be stateful and communicate with each other. The application layer itself is stateless, all data is assumed to be stored in the database or by clients.

In this scenario, an application consists of a set of methods, which can be called by clients. The application can then interact with the database by querying the data and by issuing updates to the database. Queries and updates can be enclosed in a transaction, which works on a causally consistent snapshot and guarantees atomicity of the enclosed operations, but transactions do not provide serializability. In particular, it is possible to read from stale states and there can be concurrent updates on the same object. The data store is parametrized by a data type specification, which defines how concurrent updates are merged. Often, the top-level data type is a map which results in a key-value data store. Furthermore we assume, that the database provides session guarantees as in [4].

Implementation: Fig. 3 shows a pseudocode implementation of the user management example. The variable `users` refers to a map data type in the database and maps user identifiers to another map containing the user data. The inner map contains entries for id, name, and mail which all are last-writer-wins registers [14]. The registers provide `write` and `read` methods. The maps allow to look up a key (squared brackets syntax) and they allow to `delete` entries and to check whether an entry `exists`. We use an add-wins map, which means that a delete-operation will win over concurrent update operations.

One difficulty in implementing this example on a weakly consistent data store is to make `removeUser` work correctly. As an example we consider the following property, which links the invocation of the `removeUser`-method with the expected effect and therefore is more than just an integrity constraint: "A user that is removed should eventually be removed on all replicas, and not reappear because of other operations being called afterwards or concurrently".

When using CRDTs, eventual consistency and high availability come for free. However, a developer still has to reason about the behavior of the application. For

example, consider the scenario in Fig. 2. In this scenario, an update operation on user u is first executed in session 2. Concurrently, user u is removed in session 1. Later, the update from session 2 is received in session 1 and the CRDT handles the concurrent updates by letting the update-operation win over the remove-operation of the user. Thus the getUser-operation in session 1 will return a user-record, although the user has been removed. Even worse, the user record would be inconsistent, since the name was removed, but the mail exists because of the write-operation from the concurrent update.

Choosing a last-writer-wins semantics for the map would lead to similar problems, as the previously explained add-wins semantics. With a remove-wins semantics for the map, the application would work as intended, as we demonstrate in the next section. But there are more pitfalls, into which a developer can run. Even with the remove-wins semantics at the database level, if the update method would not check, whether the user is deleted and just did a blind update, then a user could reappear after being removed. A user could also reappear, if the users identifier was not generated in a way which guarantees uniqueness.

The chosen example of object-deletion often comes up in practice. Riak, a distributed key-value store, uses tombstones for deletion, which means that remove operations win over concurrent updates. However, in the default settings tombstones are purged after 3 seconds[1], which can lead to objects which reappear after they have been deleted.

3 Specification and Verification

Having seen some possible pitfalls of the example, the question arises, how we can assure ourself, that the given implementation is indeed correct, when a remove-wins map is used. We want to describe application properties from the clients perspective. The clients can only observe the procedure calls sent to the system and the responses of the system, which we model as a trace of request and response-events. However, it is hard to specify the system just in terms of this trace, because of the inherent nondeterminism of the system. The response depends on the internal delivery of messages at the database level.

Specification: To handle the problem of nondeterminism, we adapt a technique used for specifying replicated data types [5, 15], where the outcome of operations is expressed using the happens-before relation on update-operations in the history of operations. We lift the happens before relation from the database level to the level of client-calls, by defining that a call c_1 happens before a call c_2 (we write $c_1 \prec c_2$), when all database calls in c_1 happen before every database call in c_2. Using the happens before relation and the set of all client-calls, we can specify invariants about the communication history between the application and its clients. For example, we can formalize the property from Section 2, which states that a removed user should not reappear:

[1] See "Configuring Object Deletion" at `http://docs.basho.com/riak/latest/ops/advanced/deletion/`. The behavior in Cassandra is similar.

$$\forall c_1, c_2 \in \mathit{clientCalls}.\ \forall u.\ \mathit{args}(c_1) = \mathit{removeUser}(u) \wedge\ c_1 \prec c_2 \\ \wedge\ \mathit{args}(c_2) = \mathit{getUser}(u) \quad \longrightarrow \quad \mathit{res}(c_2) = \texttt{not\_found}$$

Verification: For verification we have to express additional invariants about the internals of the application. In particular, we have to relate the client-calls with the corresponding database operations and we have to reason about the internal, local steps done by the application. We explain how our framework supports the verification of the different kind of invariants using a proof sketch for the example property.

Property 1: When *removeUser(id)* has been called, then there must be a corresponding database operation *users[id].delete()*. To support properties like this, our framework provides a mapping function, which maps each database operation to the corresponding client-call. This function can be used in invariants and is automatically updated by the framework in each step.

Property 2: There are no map update operations on a removed user, which happen causally after the remove (except for other removes).

The operations in the *registerUser* procedure cannot come afterwards, because *newUID* never returns an identifier known to clients. Therefore *removeUser(id)* must happen at a point in time after *registerUser* and no happens-before relation can exist which points into the past.

The operations in *updateMail* cannot happen after a remove, because the procedure checks whether the user exists before doing any updates. Because the code is packed in an atomic unit, the check and the map updates see the same set of operations. So if the update operations were executed after a remove, the existence check would have returned false.

The maintenance of property 2 has to be shown for the code of each method. Our framework supports this by annotating the code with assertions, similar to work by Ashcroft [1]. Some proof obligations can already be handled automatically using general properties proven in the framework. In particular, the framework restricts local assertions so that it is not necessary to consider the effect of local steps on other, concurrent procedure invocations. We believe that we can further reduce the effort by automatically generating verification conditions from the code and a few invariants.

Property 3: When *getUser* is called after a *remove*, we get that there is a database operation for deleting the user by property 1. By property 2 we know that no database operation on the same user happened after the remove. There can be concurrent updates, but since we used a remove-wins semantics for the map, we always get the required result, that the user does not exist.

For the reasoning about CRDT semantics, out framework supports high-level specifications of CRDTs, as used in work on verification of CRDTs [3, 5, 7, 15]. Users can write custom CRDT specifications for their applications or reuse and compose the existing specifications of some commonly used CRDTs.

4 Related Work

Gotsman and others have worked on modeling and verifying replicated data types [4, 5, 7]. This work is mostly focused on pen and paper proofs and therefore requires too much effort for realistic applications. Still, the work on Composite Replicated Data Types [7] is the work most similar to ours, since it also uses transactions to build bigger applications from simpler data types.

CISE [8] is a framework which concentrates on the combination of weak consistency with strong consistency and pessimistic concurrency control. The work presents proof rules for this scenario and a tool based on an approximation of these rules, which can automatically check whether enough locks are used to ensure the maintenance of data integrity constraints. However, the tool mostly concentrates on locks and cannot handle more complicated interactions with replicated data types and properties which go beyond integrity constraints and is therefore not applicable to our userbase example.

Chapar [12] is a proof framework for causally consistent databases, which was developed using Coq. The work also includes a simple model checker for applications, which can explore different schedules consistent with causal consistency, but cannot be used to prove the correctness of complex applications. Also the work only considers simple key-value stores without support for replicated data types.

Finally there is a lot of work on general purpose tools like TLA+ [11] or Alloy [9]. While these tools can be applied to check the applications we are interested in, they require to model the complete system, including the database and the data types. Hence the models can be quite big, and the automated checkers become infeasible.

5 Conclusion and Future Work

We aim to reduce the amount of manual work required for performing proofs in the future by capturing more general properties in proof rules and by using more automation. In particular we believe that it will be possible to handle atomic blocks as one single step and to generate verification conditions with more automation, which would significantly reduce the manual effort. The formalization in Isabelle/HOL allows us to improve on this incrementally, since manual proofs are always available, when no automation has been developed yet.

The primitive proof rule we use currently, already helps with informal reasoning about program correctness. We hope that developing more specialized proof rules will also lead to more insights for informal reasoning and thus help developers in writing correct applications.

Acknowledgement. This research is supported in part by European FP7 project 609 551 SyncFree `https://syncfree.lip6.fr/` (2013–2016).

References

1. Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, 1975.
2. Rastislav Bodik and Rupak Majumdar, editors. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016.
3. Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. In Jagannathan and Sewell [10], pages 285–296.
4. 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.
5. Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Jagannathan and Sewell [10], 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. Alexey Gotsman and Hongseok Yang. Composite replicated data types. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 585–609. Springer, 2015.
8. Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In Bodik and Majumdar [2], pages 371–384.
9. Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.
10. 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.
11. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
12. Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In Bodik and Majumdar [2], pages 357–370.
13. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
14. 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.
15. 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.