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.

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 ex-

plained 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.

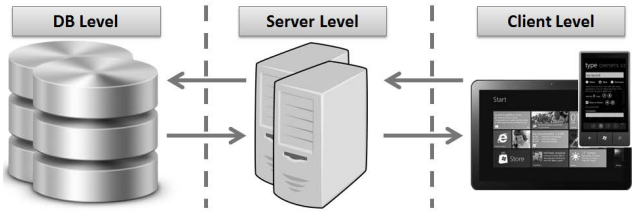

Figure 1. Different implementation layers in an application for eventually consistent systems

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 ap-

plication 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)