SyncFree Technology White Paper
Just-Right Consistency, or How to tailor consistency to application requirements

Nuno Preguiça, U. Nova de Lisboa
Marc Shapiro, UPMC-LIP6 & Inria

25 December 2016

Why application developers care about consistency

Companies that operate over the Internet need to provide services to millions of users scattered across the globe. It is imperative to have services that are available, scalable and fast, so that the expectations of their users are met. Failing to address these requirements may harm revenue and ultimately lead to the shut down of the Internet service [4].

Replication is a key technique for achieving these properties. When data is replicated at multiple machines, multiple copies of the same data exist. Some systems adopt a strong consistency model, enforcing a single view of the data across the different replicas. Several designs have been proposed for scaling strong consistency systems, but these designs provide low latency only within the boundaries of a data center, because replicas need to coordinate before replying to a request. Furthermore, these designs fail to offer high availability in the presence of network partitions and server failures and are much harder to scale.

This is why many companies adopt so-called AP (Available under network Partition) models, particularly for geo-replication scenarios. Under an AP model, a replica will always accept an update, without checking other replicas first, since this would violate availability. Thus replicas may diverge temporarily; updates are propagated in the background (asynchronously) to other replicas. The advantage is low latency, high availability, and efficient use of CPU resources, but it makes writing applications more difficult, as concurrent updates may conflict and compromise application correctness.

A well-known AP model is Eventual Consistency, which is particularly efficient and scalable, because it enables unlimited parallelism. However, Eventual Consistency is also highly anomalous, since even two closely related updates may be observed in any order and mixed with other updates.

The SyncFree consistency model

The SyncFree geo-replicated database Antidote offers the strongest AP consistency model. It is called Transactional Causal Consistency (TCC), because it supports both highly-available transactions and causal visibility. Causal visibility guarantees that dependent updates are applied in the correct order: for example, in a train reservation system, a reservation for a new car is only applied after the operation that added that car to the train. Transactions guarantee that a set of operations happen consistently and in an all-or-nothing fashion. For example, two reservations inside a transaction are both made visible at the same time.

TCC can be as available, scalable and performant as eventual consistency, but developers using TCC enjoy the guarantees of transactions and causal visibility, and related updates will not suffer anomalies. In this sense, TCC partially bridges the gap with strong-consistency models. However, TCC does allow concurrent updates to the same data item; these are merged (instead of leading to aborts as in traditional SQL-style transactions).

Co-designing the application and the model

Since concurrent updates are allowed, and even though its guarantees are relatively strong, the TCC model is not sufficient to guarantee the correct execution of some applications. This is because some kinds of application invariants require concurrency control, which conflicts with availability. However, switching to a strongly-consistent (and non-available) model, which orders all updates, is overkill in many cases.

Consider, for instance, an e-commerce application that sells some articles in stock. To avoid that stock becomes negative, one way would be to run the application above a strongly-consistent database. However, it should be clear replenishing the stock cannot make stock negative, and therefore does not need to be synchronised.

The Just-Right Consistency approach consists of *heuristics* and *tools* to ensure that the system has sufficient concurrency control to ensure the application is correct, but no more, and that minimises its availability and performance impact.

The heuristics are encapsulated in specialised data types. For instance, the e-commerce stock example can be solved by using the *Bounded Counter CRDT* provided with Antidote. This data type will stop stock-depleting operations from going under zero but will allow stock-increasing operations at all times. Furthermore it enables logically dividing the stock, so that most stock-depleting operations are available too. The Bounded Counter CRDT is described in the companion white paper, "Bounded Counters: maintaining numeric invariants with high availability."

SyncFree also offers *automated tools* to carefully analyse the application and detect which operations require concurrency control. For instance, the tools would analyse the e-commerce application and automatically determine that stock-increasing operations do not need synchronisation, but that stock-depleting ones may violate the non-negative-stock invariant. How to repair this issue is the developer's design decision. One alternative is to *strengthen the consistency* by controlling the concurrency of the stock-depleting operations, e.g., disallowing concurrent sales of the same item. The (only) other alternative is to *weaken the specification*, e.g., accept that stock becomes negative and automatically ordering new stock to compensate. The former approach decreases availability but ensures the invariant is always true; the latter remains available but may be more dangerous.

We have developed two tools to this effect.

CISE The CISE tool statically verifies that a given application will maintain a given invariant under a given consistency model. If not, the CISE tool will provide a counter-example that helps the developer diagnose and add the missing concurrency control. Then the developer runs the tool again, and so on until the verification succeeds. Once the verification is successful, this is a proof that the invariant will be satisfied in all executions.

IPA The IPA tool does a similar analysis to CISE. When verification fails, IPA will automatically propose appropriate weakenings of the specification.

They are described in more detail hereafter.

The CISE Tool

While some applications can run correctly under weak consistency, others require some degree of synchronization for, at least, some operations. CISE tool allows to verify if an application will run correctly under weak consistency or whether it requires some degree of synchronization.

As such, CISE tool will be used by application programmers to verify their applications before deployment. CISE tool needs to be used whenever the application specification is changed, i.e., new operations are added or the behavior of operations is changed.

How to use CISE

For using the CISE tool, the application programmer must specify the application correctness criteria as a set of invariants over the applications state. Additionally, she must specify the pre-conditions and post-conditions of the operations defined in the application.

Given these input, the CISE tool will verify if the program will run correctly under weak consistency. When this is not the case, the tool will present a counter-example with a run that leads to the violation of an application invariant when executing a pair of concurrent operations.

The tools is additionally able to identify a minimal set of synchronisation points in application operations that, when enforced, guarantee the correct executions of the application. These synchronisation points are implemented by a set of tokens (a.k.a., reservations) and escrowable data types (e.g., the Bounded Counter) that restrict the concurrent execution of some operations.

The modified application adopts a hybrid approach where some operations remain unsynchronised but selected ones are synchronised. For instance, consider a train-reservation example with the invariant "no overbooking." The tool would allow concurrently adding a car, but would flag concurrent reservations as unsafe and would propose the synchronisation to be added. The developer has the option of changing the specification (e.g., allowing overbooking), or adding the suggested synchronisation – in this case, using the Bounded Counter would allow different replicas to concurrently accept reservations to the same car that do not exceed some threshold, while tokens would allow a single replica to accept reservations for a given car at a given moment. The tool should be used again to validate the modified application.

The CISE tool is instrumental to help programmers achieve correct behavior and good performance. Too much synchronisation degrades performance and availability; too little might corrupt data.

The CISE tool is generic, and can verify the correct maintenance of application invariants for many weak consistency models, including Antidote, Lasp and Legion. However, the analysis requires at least causal visibility of updates.

What's behind the CISE tool

The CISE tool builds upon a sound verification logic [1]. It can be summarised by the following three rules: *(i)* Each update must be individually correct, i.e., that before making any changes, it checks a *precondition* establishing that the update will maintain the invariant. *(ii)* Any two concurrent updates must commute. *(iii)* If an update u is concurrent with some update v, then the precondition of u (from Rule *(i)*) must not be made false by applying v.

If any update violates Rule *(i)*, then the application is incorrect even in a sequential execution. To fix it, the developer must either strengthen the precondition or weaken the invariant. If any pair of updates violates Rule *(ii)*, then the developer must, either redesign the updates (weakening the specification), or insert concurrency control, so that they do not execute concurrently. Similarly, if any pair violates Rule *(iii)*, the developer must either weaken the specification or insert concurrency control.

Check out our description of the tool [2] and our demo video [3]. The source code of the CISE tool is available on the SyncFree github [5].

The IPA tool

The previous tools allow to verify that an application can run correctly under weak consistency. When this is not possible, if the programmer does not want to change the correctness rules of the application, the previous tools propose restricting the concurrent execution of operation by adding synchronisation points, thus guaranteeing that the modified applications will execute correctly. This synchronisation imposes overhead to the operation execution and reduces availability and fault tolerance.

It has been shown that in some situations, it is possible to enforce the same correctness rules under weak consistency by slightly changing the specification of operations and using per-object conflict resolution rules. However, identifying such situations is not trivial. The IPA tool addresses this problem by identifying which operations and how these operations must be modified to execute correctly under weak consistency.

As such, the IPA tool will be be used by application programmers to modify their applications before deployment. For operations that cannot be modified, the application programmer must add some form of synchronisation. To this end, she can use the CISE tool for helping her identifying the synchronisation point required.

For using the IPA tool, the application programmer must specify the application correctness criteria as a set of invariants over the applications state. Additionally, she must specify the effects (post-conditions) of the operations defined in the application.

Given these inputs, the IPA tool detects which pairs of operations can lead to incorrect behavior when executed concurrently under weak consistency, and suggests modifications to the application to make them execute correctly. These modifications consist in defining appropriate rules for merging concurrent updates in each object and identifying the set of additional effects that are necessary in each operation in order to make the operation compatible with all other concurrent operations.

The IPA tool is complementary to the CISE tool. While the IPA tool helps programmers extend the set of operations that can run correctly under weak consistency by modifying the operations, it will be impossible to modify all operations. The CISE tool can be used to help a programmer adding the necessary synchronisation for her application to run correctly under weak consistency.

As the CISE tool, the IPA tools can be used with applications that use Antidote, it can also be used with application that run other weakly consistent databases.

The IPA tool systematically explores all pairs of updates that may execute concurrently, and verifies if the concurrent execution of such operations would lead to an invariant violation. When this is the case, it explores if the operations can execute concurrently if some effects are added to each of the operations.

Related tools

We already mentioned Bounded Counters. The SyncFree project has developed two further related tools, Commander and Verifico.

Commander Commander is a tool for finding application bugs in an application running above a weak consistency model. Commander systematically explores operation reorderings in order to detect cases that will violate application invariants. Compared with CISE and IPA, Commander is focused on run-time checks and debugging and does not assume causal visibility. We refer the interested reader to the companion White Paper: “Commander: Runtime Verification of Programs Running on Weakly Consistent Platforms.”

Verifico Verifico is a static verification framework, supporting reasoning about application correctness when running under eventual consistency and using CRDTs. When compared with CISE, Verifico provides less automation but more flexibility in the properties that are verified. We refer the interested reader to the companion White Paper: “Verifico: CRDT-App Verification Framework for Isabelle.”

References

[1] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. ’Cause I’m strong enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 371–384, St. Petersburg, FL, USA, 2016.

[2] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE tool: Proving weakly-consistent applications correct. In *W. on Principles and Practice of Consistency for Distr. Data (PaPoC)*, EuroSys 2016 workshops, London, UK, April 2016. ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery.

[3] Mahsa Najafzadeh and Marc Shapiro. Demo of the CISE tool, November 2015.

[4] Eric Schurman and Jake Brutlag. Performance Related Changes and their User Impact. Presented at Velocity Web Performance and Operations Conference, 2009.

[5] SyncFree Project. SyncFree @github. GitHub open-source repository. https://github.com/SyncFree.