Configurable consistency

WPE II

Konstantinos Kallas

Abstract

Geo-replicated data stores are increasingly important for the internet infrastructure. In order to facilitate application development on top of these data stores, consistency notions are used as contracts between the developer and the data store. These consistency notions are often too strict, leading to poor performance, or too weak, leading to unpredictable behaviors. This has led to a recent emergence of hybrid consistency notions that can be configured according to application requirements. These configurable consistency notions come with ranging support to help users configure them depending on their application requirements.

In this report I survey three different configurable consistency notions together with the developer support that they provide for configuring them. The different notions are compared in terms of expressiveness, implementability, and ease of configurability. Finally, I will propose possible directions for future work in this domain.

1 Introduction

Geo-replicated data stores are the foundation of the Internet. In order to improve availability, latency, and network partition tolerance, these data stores perform operations guaranteeing only weak consistency in order to minimize synchronization between replicas. However, weak consistency often leads to unintuitive – and sometimes dangerous – system behaviours. For example, concurrently withdrawing from a bank account under weak consistency semantics could lead to a negative account balance. In order to address that, several configurable consistency notions have recently emerged. These consistency notions are hybrid, performing different operations with different consistency guarantees.

However, supporting configurable consistency is not enough, as it puts the burden on users to choose different consistency levels for each operation. Users then have to choose a configuration that preserves specific invariants on the data without compromising performance achieved by weaker consistency. This is extremely error-prone and challenging, as users have to reason about concurrent executions of different operations and how they could lead to an invariant violation. This motivates the need for both conceptual and automatic tools that support users in their quest of choosing valid consistency levels that both preserve data invariants as well as achieve optimal performance and minimal synchronization.

In this report I will analyze three works that introduce novel configurable consistency notions and provide support to guide developers when choosing appropriate consistency levels for their datastore operations. All three of these works develop a new configurable model of hybrid consistency, and then develop a framework to reason about different configurations of the model.

First, I will describe the work of Li et al. [9]. They propose a new hybrid consistency notion, called *RedBlue consistency*, that allows some operations to be eventually consistent and others to be strongly consistent. They provide conditions that, given integrity invariants on the data objects, dictate when operations must be strongly consistent, or when they can be eventually consistent.

Second, I will review the work of Balegas et al. [1]. They propose a novel consistency notion, called *Explicit Consistency*, that strengthens eventual consistency by preserving given application-specific invariants. They develop a method that, given the invariants, identifies which operations would violate them if executed without strong consistency guarantees, and allows the developer to choose between executing them in a strongly consistent manner, or providing an invariant repair method if the invariant was broken.

Finally, I will describe the work of Gotsman et al. [4]. They define a hybrid consistency model that generalizes several popular hybrid consistency notions (including the work by Li et al. [9]). They propose a proof rule for establishing that a particular choice of consistency levels for a set of operations on a data store preserves a given integrity invariant on the data. They prove their rule to be sound, and develop a tool that automatically establishes the rule using an SMT solver.

2 Background

The underlying system model that all these works build upon is replicated datastores. Replicated datastores are mainly used as the foundation of distributed applications. A replicated datastore contains a set of replicas that hold a copy of the state. The datastore supports a set of operations that can be performed by clients of the system. Each operation is first performed at one replica of the system (which is called primary) and is then propagated to the rest. Operations can be complex, modifying the state of the datastore in a non-trivial way and also returning data to the user as a response. Unfortunately, developing applications on top of replicated datastores is an extremely error-prone process. Since operation propagation is not completed in an instant, the states of different replicas may diverge, becoming inconsistent.

This issue can be addressed using consistency notions, which are essentially contracts that describe the behaviors that a datastore can have. Another way of thinking about them is that they describe the guarantees that can be expected by an application developer. An example is sequential consistency [8], which is one of the strongest consistency notions. Sequential consistency requires that the datastore behaves as if it was a sequential store, performing one operation at a time, preserving the order that each client issues operations with. This guarantee significantly simplifies application development, as it prevents operations from being performed concurrently, leading to an inconsistent state.

However, sequential consistency (and other strong consistency notions) require synchronization, which can be a performance bottleneck, and makes the system unavailable in case of a network partition [3]. Because of that, weaker consistency notions [18, 11] have become increasingly popular. These weaker notions improve performance by offering weaker guarantees, allowing several unexpected behaviors that could not occur if operations were executed with strong consistency guarantees. A common weak consistency notion is eventual consistency, which only guarantees that different replicas will eventually converge to the same state. To better understand the kinds of unexpected behaviors that can be observed from weakly consistent datastores, consider a datastore that contains bank account information and only guarantees eventual consistency. In this setting, two clients can concurrently withdraw from the same account, leading to overdraft.

In order to get the best of both worlds, several hybrid consistency notions have been proposed [7, 14, 15, 17, 16]. They offer better performance when possible, but also predictable behavior that gives similar guarantees with strong consistency. Hybrid consistency notions are usually configurable, either in the operation [9, 1, 10, 4] or the data granularity [6, 19]. This report focuses on configurable consistency in the level of operations. This means that different operations are executed with different consistency guarantees. In the bank account example from above, withdraw operations would be executed with strong consistency guarantees, while other operations, like deposit or view balance, would be executed with weaker consistency to achieve better performance.

An important requirement of hybrid consistency notions is the ease of configurability. Ideally, there should be systematic support that helps users to choose a configuration that offers the weakest possible guarantees, achieving good performance and correct behaviors. More precisely, we are interested in developer support when the data store has to satisfy a set of invariants. In our bank example one of them could be that the balance of bank accounts is non-negative.

3 RedBlue Consistency

In this work Li et al. [9] propose a novel hybrid consistency notion that is called RedBlue consistency. This consistency notion offers configurable consistency at the granularity of individual operations. Operations are

divided in two groups, *Red* which are executed under strong consistency and *Blue* which are executed under weak consistency. They identify adequate conditions that dictate when an operation must be *red* in order for some invariant to be preserved. In addition, they propose a method that allows more operations to be classified as *blue* without violating data invariants.

3.1 Consistency Definition

Their consistency definition assumes a datastore with several replicas, where each replica is modeled as a deterministic state machine that holds a copy of the system state. Clients interact with the replicas by performing specific operations. Replicas apply these on their copy of the state, deterministically transforming it and producing a response that is sent to the client.

Using this system model, they define RedBlue consistency. RedBlue consistency gives different consistency guarantees for different operations. More precisely, it allows dividing operations in two groups: *red* and *blue*, out of which *red* must be executed in the same order in all replicas, while *blue* can be executed in different order in each replica. RedBlue consistency guarantees causal consistency for all operations, i.e. if op_1 was performed before op_2 in op_2's primary replica, then they will be performed with the same order in all other replicas too.

In order to formally define RedBlue consistency, they first define RedBlue order and causal serializations of a RedBlue order. The RedBlue order is a global partial order of all the operations (both red and blue), where red operations are totally ordered. Note that blue operations don't have to be ordered with red operations.

Definition 1 (RedBlue order)**.** *Given a set of operations $U = B \cup R$, where $B \cap R = \emptyset$, a RedBlue order is a partial order $O = (U, \prec)$ with the restriction that $\forall u, v \in R$ such that $u \neq v$, $u \prec v$ or $v \prec u$ (i.e., red operations are totally ordered).*

Based on the global RedBlue order they define causal serializations, which are serializations of the global order for each specific replica that also satisfy causal consistency as described above.

Definition 2 (Causal serialization)**.** *Given a site i, $O_i = (U, <)$ is an i-causal serialization (or short, a causal serialization) of RedBlue order $O = (U, \prec)$ if (a) O_i is a linear extension of O (i.e., $<$ is a total order compatible with the partial order $\prec$), and (b) for any two operations $u, v \in U$, if site$(v) = i$ and $u < v$ in O_i, then $u \prec v$.*

Given these two definitions, RedBlue consistency is defined as follows:

Definition 3 (RedBlue consistency)**.** *A replicated system is O-RedBlue consistent (or short, RedBlue consistent) if each site i applies operations according to an i-causal serialization of RedBlue order O.*

3.2 Improving operation commutativity

RedBlue consistency restricts the possible orderings of operations, therefore also restricting the possible states that the system can reach, allowing invariants that require serialization of red operations to be preserved by the system. However, on its own it doesn't guarantee state convergence, a very important property that ensures that different replicas converge to the same state after performing the same operations. In order to guarantee state convergence, they require that all blue operations are *globally commutative*. This means that all blue operations commute with all operations (both blue and red). This is a strong but necessary requirement, as blue operations can be ordered in different ways in different replicas.

However, this requirement limits the space of blue operations that can be supported by RedBlue consistency. In order to address this, they propose a technique to make more operations globally commutative, by slightly altering their semantics. More concretely, the technique requires splitting each blue operation into two components, a *generator operation*, that is only executed locally and produces a *shadow operation* without altering the replica state. The shadow operation is then executed at every site (as would the original operation be).

It will be easier to understand this with an example. Consider a datastore representing a bank account that has three operations: (i) *deposit*(*amount*), which increases the account balance by *amount*, (ii) *accrueinterest*(), which increases the balance of the account by adding 5% interest, and (iii) *withdraw*(*amount*), which decreases the account balance if the resulting balance is non-negative. Assuming that our invariant is keeping the account balance non-negative, the first two operations can be safely performed concurrently as they only increase the balance. Unfortunately, they do not commute. Starting with an initial balance of 100, performing *deposit*(20) and then *accrueinterest*() would result in a balance of 126, while performing them in the opposite order would result in a balance of 125. Using their technique, one can split *accrueinterest*() in a generator that computes the interest using the current balance, and a shadow operation that just adds the precomputed interest. Revisiting the previous example, performing *accrueinterest*() would add the same interest in all replicas, no matter what their current balance is.

Unfortunately, this technique comes with a drawback; by splitting operations in two parts, and interleaving arbitrarily many operations between them, shadow operations don't always behave as expected. For instance, in the above example, accruing interest won't take into consideration Bob's concurrent deposit. If this introduces an unwanted system behaviour however, the user could classify the operation as red, and have the shadow operation be the same as the original one.

3.3 Conditions for Red/Blue

As we saw in the previous section, blue operations must be globally commutative so that the system state converges. What about integrity invariants? It is clear that if all operations are labeled red, and the integrity invariants are preserved by all serialized executions of operations, then the integrity invariants are preserved.

To determine which operations can be safely labeled blue without violating any invariants, they first define what it means for a shadow operation to be invariant safe. A shadow operation $h_u(S)$ is invariant safe, if for all valid states S and S', the resulting state after applying $h_u(S)$ to S' is also valid. Note that a state is called valid if it satisfies the invariant. Intuitively, because blue operations are performed without any synchronization, the shadow operation that was generated on state S might be applied to an arbitrarily different valid state S' in another replica. Using the definition of invariant safe shadow operations, they state the following theorem.

Theorem 1. *If all shadow operations are correct and all blue shadow operations are invariant safe and globally commutative, then for any execution of that system that is RedBlue consistent, no site is ever in an invalid state.*

A shadow operation is considered to be correct, if for all starting states, the generator operation does not have any effect, and the produced shadow operation has the same effect as the initial operation. Essentially, if shadow operations were executed in sequence, then the system should reach the same state that it would reach when executing the initial operations. Based on the above, they then give two rules that are adequate to classify operations as red. (1) If two shadow operations u and v are non-commutative, label both of them to be red. (2) If a shadow operation u is not invariant safe, label it red. The rest of the operations can be safely labeled as blue.

For the bank example that was introduced above, all three operations (*deposit*, *accrueinterest*, *withdraw*) are pairwise commutative, but *withdraw* is not invariant safe. Consider the instance where $S = 100$ and $S' = 50$, which are both valid. Then $h_u(S) = (\lambda S'.S' - 100)$, but after applying it to S' we would end up with $S'' = -50$, which is clearly not valid.

3.4 Discussion

Technically, the presented definitions do not support splitting operations into generators and shadow operations. In the paper, they revisit the definitions of causal serializations and RedBlue consistency after splitting operations. Since the new definitions are trivial extensions and do not offer any new insights, I have not included them in this report. The authors of this work have also developed a system that implements RedBlue consistency. This system is also a significant contribution of their work and they describe it in detail

in the original paper [9]. Since the goal of this report is to describe the configurable consistency notions and their configurability support, I have decided to not include a description of their system in it.

Finally, note that even though they identify conditions for operations to be labelled red or blue, they do not propose an automatic mechanism to do so. In an extension of this work [10], they develop a technique that uses a combination of static and dynamic analyses to automatically decide if strong consistency is necessary, flagging the necessary operations as red.

4 Explicit Consistency

In this work, Balegas et al. [1] propose a novel hybrid consistency notion called *Explicit Consistency*. Explicit Consistency is defined in terms of application specific invariants, and not execution partial orders (as most other consistency notions are). This allows an implementation to freely reorder operations at replicas, as long as they don't violate these invariants. They use a static analysis to find which operations can safely be executed without coordination. For the unsafe ones, they ask the user to choose between violation avoidance (using several different forms of locks) or invariant repair, after the invariant has been broken.

4.1 Definition

Their system model is a standard replicated datastore that offers a set of high-level operations (that they call transactions) on the datastore. They model executions as growing sets of transactions that have happened on a state $T(S)$. A transaction t_a happened before t_b that executes on S_b, denoted $t_a \prec t_b$ if $t_a \in T(S_b)$. Two transactions are concurrent if none is in the set of the state that the other was executed on for any of the replicas. They represent an execution of the system as the set of transactions T and the $\prec$ partial order. A serialization is a linear extension of the $\prec$ partial order. They implicitly assume that the happens before relation does not conflict in different replicas, effectively assuming that replicas causally propagate events.

Transactions can execute concurrently, with each replica executing transactions according to different serializations. They assume state convergence, that is that all serializations of all executions lead to the same database state. This can be satisfied if all operations are commutative (for example by using replicated data types [13]).

They assume that the datastore comes with an invariant I, i.e. a predicate on a state of the datastore. They define I-valid serializations to be the serializations where the invariant holds in every intermediate state that can be acquired by performing a prefix of the serialization. Based on I-valid serializations, they say that a system provides explicit consistency if all serializations of its execution partial orders are I-valid.

4.2 Identifying unsafe concurrent operations

They have designed a static analysis that identifies operations that are unsafe to execute concurrently. They call a set of unsafe operations to execute concurrently an I-offender set. They support first order logic formulas as the datastore invariant. Formulas can contain predicates and functions. In order to simplify the analysis, the functions and predicates are uninterpreted, and they require that users annotate operations with postconditions to determine the predicate and function values. Disjunctive postconditions are supported by splitting an operation into multiple dummy ones. An example postcondition for the withdraw operation that indicates that the balance of the account under question is decreased by the withdrawal amount.

```
@Decrements(balance($0), $1)
void withdraw(Account acc, int amount){ ... }
```

They show that despite being simple, their invariants can express several interesting conditions. For example they can express lower (and upper bounds on values) $\forall A, account(A) \implies balance(A) \leq 0$. They can also express integrity invariants, e.g. that an entry in the balance table, must also be in the account table $\forall A, hasBalance(A) \implies account(A)$, or existential quantification with the use of counters $\forall A, account(A) \implies nrHolders(A) > 0$. A limitation of the uninterpreted functions and predicates, is that

reachability (or other properties on recursive structures) cannot be expressed, so sound over-approximations of these properties must be encoded instead.

Since they are interested in operations that are unsafe to perform concurrently (they call a set of these operations an I-offender set), they assume that all operations preserve the invariant if executed sequentially. They achieve that by extending operations with preconditions that guarantee that sequential execution would not violate invariants. Their static analysis algorithm can be broken up in two steps. First, they check whether any pair of concurrent operations can result to conflicting postconditions (e.g. a concurrent addition and removal of an account holder). These pairs of operations are added to the set of I-offender sets. They then check for all pairs of operations whether they satisfy the invariant (given that they were performed in an I-valid state) when executed concurrently. The pairs that do not satisfy the invariant, are added as I-offender sets. They argue that checking pairs of operations is sufficient to detect all invariant violations . Internally, the algorithm performs these checks by calling an SMT solver.

4.3 Handling I-offender sets

After identifying pairs of operations that are unsafe to be executed concurrently, the user has to decide how to prevent invariants from being violated. They offer two different approaches to achieve that.

They first allow developers to repair invariant violations after they have happened. To this end, they provide a library of data types with conflict resolution strategies. They also allow users to implement custom data types with conflict resolution, however they do not offer any support for checking that the conflict resolution strategies are correct.

They also offer a system of reservations, which are similar to locks, that can be used for fine-grained coordination between replicas, so that operations are not performed in an unsafe way concurrently. This way they avoid invariant violations. There are several different types of reservations, each of which is fit for a different type of invariant.

Multi-level lock reservation: This is their base mechanism to restrict the concurrent execution of operations that can violate invariants. Multi-level locks can provide three different rights: (i) *shared forbid*, forbidding some operation to occur while this right is held; (ii) *shared allow*, allowing some operation to occur; (iii) *exclusive allow*, allowing some operation to occur only in the replica that holds the right. Essentially the *shared forbid* right on an operation can be held by many replicas, and while it is held, this operation cannot be executed in any replica. Similarly, the *shared allow* right on an operation can be held by many replicas, and this operation can be executed in any of the replicas that holds it. In contrast, the *exclusive allow* right can only be held by one replica, and only that replica can perform this operation while it holds it.

Multi-level locks offer finer control over synchronization as they allow "partial" synchronization, e.g. when many replicas acquire a *shared allow* lock on an operation, they can all execute it without any synchronization. However, it is possible to have even finer control over synchronization by using locks that are specific to the structure of the invariants.

Multi-level mask reservation: For invariants in the form of a conjunction, such as $P_1 \vee P_2 \vee \ldots \vee P_n$, two operations that each make a predicate P_i false, are considered to be an I-offender set by their analysis as all other predicates could have been initially false. However, using simple multi-level locks in this setting could lead to very pessimistic synchronization; getting a lock on one operation that makes a P_i false would prevent the execution of any other operation that makes any P_j false, even though it would suffice to guarantee that some other predicate P_k remains true.

They address that by offering multi-level mask locks, which are essentially vectors of multi-level locks, one for each predicate. Then when a replica acquires a *shared allow* lock, it must also obtain a *shared forbid* for another predicate that is currently true to ensure that the conjunction remains true.

Escrow reservation: For invariants that include upper or lower bounds on numerical values, they extend the escrow technique [12] to improve synchronization when performing operations that increment or decrement some value. The escrow technique is very similar to semaphores that are used in multi-threaded programming. For an invariant of the form $x \geq n$, if the initial value of $x = x_0$ then there are initially $x_0 - k$ rights, associated with variable x and this invariant, that can be shared between replicas. When a replica wants to perform an operation that decrements x by k, it has to consume k rights. If it does not have enough, it has to either acquire them from other replicas, or the operation fails. Whenever a variable is involved in more than one invariant, several escrow reservations are affected by one increment (or decrement).

Partition lock reservation: Sometimes invariants can refer to partitionable resources. For example, in a classroom management system an invariant could be that no two lectures are scheduled to happen in the same class at the same time. To execute an operation that schedules a lecture, a replica would have to acquire an *exclusive allow* lock, preventing any other replica from performing the same operation for a different time slot. To address that, they propose partition reservations, which are locks on an interval of real values. This allows different replicas to concurrently perform operations on non-overlapping ranges.

4.4 Discussion

In this work they propose a coarse-grained static analysis to determine which pairs of operations could invalidate the invariants if performed concurrently. However, the main performance benefits of explicit consistency (in comparison to e.g. RedBlue consistency) are due to the fine-grained reservation mechanisms. Unfortunately, the gap between determining which pairs of operations are unsafe, and determining which reservation to use for these pairs of operation is left unexplored and has to be covered by the users of the system. A related issue is that reservations are specified using their implementation, which makes it difficult to modularly reason about the guarantees of a system of replicas that acquire different kinds of reservations.

An important benefit of the reservation implementation in their system is that they are released lazily. This means that a replica can acquire and keep reservations if no other replica tries to acquire them. This could lead to performance benefits in comparison with implementations that eagerly synchronize every time an unsafe operation has to happen, since operations that happen often at one replica could be performed without any synchronization because the replica already holds the necessary reservations.

Another possible consideration for their work is that it is required that the users describe how each operation affects the state using postconditions. This methodology has two main drawbacks: (i) the postconditions are not checked by the system, so they might not correspond to the real operation, and (ii) it could be difficult to precisely express the effects of an operation on the state using postconditions, which could lead to more pairs of operations being flagged as unsafe from the static analysis.

Finally, as part of their work they also develop a system that implements Explicit Consistency. Similarly to the RedBlue consistency paper I have decided to not include a description of their system in this report as it does not fall in its scope.

5 Reasoning about consistency

In this work, Gotsman et al. [4] introduce a methodology to verify that a given configuration of a hybrid consistency notion satisfies a set of invariants on the data. They define a configurable hybrid consistency notion that is general enough to express several popular notions (such as RedBlue [9]); from now on we will refer to is as Hybrid consistency. Their model of consistency is configurable given a conflict relation, which is a way to guarantee that certain operations will not be performed concurrently. They then propose a proof rule that can establish whether a set of invariants is satisfied by an instantiation of their general hybrid consistency notion. This proof rule is modular, in the sense that it allows reasoning about the guarantees of each operation separately, under some assumptions about the behaviours of other operations.

5.1 Formal Model

They first define the general configurable consistency model. Their model guarantees at least causal consistency, which is the same as the other models do. They assume that the underlying system is a set of replicas that perform operations issued by the clients. Client operations are deterministic and they are performed in a single origin replica (primary), which then sends a message to all other replicas containing the effect of the computation. Messages are causally propagated and are received at most once.

In order to abstract from a particular language, they assume that the semantics of operations are given by a function F of type

$$Op \to (State \to (Val \times (State \to State)))$$

which they denote as

$$\forall o, \sigma. F_o(\sigma) = (F_o^{val}(\sigma), F_o^{eff}(\sigma)).$$

Given a state σ of o's origin replica, $F_o^{val}(\sigma)$ determines the value that will be returned to the client, and $F_o^{eff}(\sigma)$ is a function that will be applied to the state of each replica that will perform this operation.

As an illustrative example, consider a toy banking application where the state is an integer (say the account balance) and the supported operations are deposit an amount α, accrue a 5% interest, and query the balance:

$$\begin{aligned} F_{deposit(\alpha)}(\sigma) &= (\bot, (\lambda\sigma'.\sigma' + \alpha)) \\ F_{interest}(\sigma) &= (\bot, (\lambda\sigma'.\sigma' + 0.05 * \sigma)) \\ F_{query}(\sigma) &= (\sigma, (\lambda\sigma'.\sigma')) \end{aligned}$$

Similarly to the other works, they assume that replicas converge by requiring that all operations commute. Convergence means that two replicas that have performed the same operations must have ended up in the same state.

The execution of an operation is called an event. In their model, the order of events in an execution is represented using a "happens-before" relation. This relation is a *strict partial order*, meaning that it is both transitive and irreflexive. Because of causal consistency, the "happens-before" relation orders at least all the causally dependent events. For example, if o_2 was performed on an origin replica where o_1 was already performed, $o_1 < o_2$, where $<$ represents the "happens-before" relation.

Assume that we added a new operation that withdraws an amount $\alpha > 0$ from the bank account in our toy banking application

$$F_{withdraw(\alpha)}(\sigma) = \text{if } \sigma \geq \alpha \text{ then } (\text{true}, (\lambda\sigma'.\sigma' - \alpha)) \text{ else } (\text{false}, (\lambda\sigma'.\sigma')).$$

Obviously, we do not want to be able to withdraw a higher amount than the account balance, and in general we would like to ensure that the account always has a non-negative balance. We can represent this predicate as the following invariant

$$I = \{\sigma | \sigma \geq 0\}.$$

The withdraw guard is adequate to ensure that the account's balance will never become negative in sequential executions, but it is not clear if it guarantees the satisfaction of the invariant when concurrent withdraws are performed in different replicas. Indeed, two concurrent withdraws can lead to the account's balance becoming negative.

In order to allow for the selective strengthening of their consistency notion, they extend their model with a *token system* $T = (Token, \bowtie)$, where $Token$ is simply a set of tokens, and $\bowtie \subseteq Token \times Token$ is a symmetric *conflict relation* that represents which tokens are conflicting. They also extend operations with a third component $F_o^{tok}(\sigma) \subseteq Token$ that shows which tokens are associated with each operation. Therefore operations are redefined as:

$$\forall o, \sigma. F_o(\sigma) = (F_o^{val}(\sigma), F_o^{eff}(\sigma), F_o^{tok}(\sigma)).$$

Informally, operations that acquire conflicting tokens have to be causally dependent on each other, meaning that they must necessarily be related by the "happens-before" relation. This doesn't allow them to be executed with a different order in different replicas.

Using the above formalisms an execution can be represented by a set of events that correspond to performed actions, and a "happens-before" relation on the set of events. A consistent execution with a token system T and the semantics of operations F is defined to be an execution that could be produced by a datastore with some arbitrary message delivery. They prove that given the operation commutativity assumption, every consistent execution converges to exactly one final state, meaning that all replicas participating in an execution would converge to the same state after performing all the events in the execution.

In order to express the generality of their model, they use it to express different common consistency notions by instantiating the token system accordingly. For example, if the set of tokens is empty $Token = \emptyset$, then the model is equivalent to causal consistency [2, 11]. In order to express sequential consistency [8], they have one token $Token = \{\tau\}$ that conflicts with itself $\bowtie = \{(\tau, \tau)\}$, and require that all operations acquire that token $\forall o, \sigma. F_o^{tok}(\sigma) = \{\tau\}$. Finally, they express RedBlue consistency [9], where all operations are either *red* Op_r or *blue* Op_b, by having one token τ that conflicts with itself and making all red operations acquire τ.

$$(\forall o \in Op_r. \forall \sigma. F_o^{tok}(\sigma) = \{\tau\}) \wedge (\forall o \in Op_b. \forall \sigma. F_o^{tok}(\sigma) = \emptyset)$$

5.2 State-based rule

Given the execution model, they define the verification problem as follows: given a token system $T = (Token, \bowtie)$, prove that operations F maintain an integity invariant $I \subseteq State$ over database states. In other words, all executions that are consistent with T, F must evaluate to a state satisfying I.

The difficulty with solving the aforementioned verification problem as is, is that infinitely many executions have to be taken into account. To address this difficulty they propose a modular proof rule that allows to reason about the behaviour of each operation separately, without considering executions explicitly. They call the rule *state-based* because it refers to replica states and not events in the execution. The rule contains three preconditions:

S1. $\sigma_{init} \in I$

S2. $G_0(I) \subseteq I \wedge \forall \tau. G(\tau)(I) \subseteq I$

S3. $\forall o, \sigma, \sigma'. (\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((F_o^{tok}(\sigma))^{\perp}))^*) \implies (\sigma', F_o^{eff}(\sigma)(\sigma')) \in G_0 \cup G(F_o^{tok}(\sigma))$

The proof rule is based on inductive reasoning, as it requires that the invariant I holds initially (S1), and that it is preserved by any operation that is executed afterwards. If all operations where executed sequentially, on one replica, then we would simply have to show that each operation preserves the invariant if it was performed on a state σ that satisfied the invariant. More formally:

$$\forall \sigma. (\sigma \in I \implies F_o^{eff}(\sigma)(\sigma) \in I).$$

The difficulty arises in the general case with many replicas, where we have to show that the effect of an operation has to preserve the invariant if executed on any state σ' that also satisfies the invariant

$$\forall \sigma, \sigma'. (\sigma, \sigma' \in I \implies F_o^{eff}(\sigma)(\sigma') \in I).$$

However, this is not precise enough, as it doesn't take tokens and their restrictions into account. They address this by introducing a form of rely-guarantee relations [5], where each token τ is associated with a relation $G(\tau)$ on the changes all operations that acquire it can make on the states that they are applied. There is also a relation G_0 about operations that don't acquire any token.

Based on that, condition S2 requires that all guarantee relations preserve the invariant I. Condition S3 is slightly more complicated, so let's break it down to better understand its components. First of all, $T^{\perp} = \{\tau \mid \tau \in Token \wedge \neg \exists \tau' \in T. \tau \bowtie \tau'\}$ and R^* denotes the reflexive transitive closure of a relation R. The first part of S3 restricts σ' to all the states that can be produced by arbitrarily many executions of operations with non-conflicting tokens to $F_o^{tok}(\sigma)$. The conclusion requires that performing the effect on such an σ' preserves satisfies the guarantee relations of the tokens that the operation had acquired.

By satisfying the three conditions, the proof rule establishes that all possible executions preserve the given invariant. Note that the the guarantee relations have to be manually provided, and that might require significant effort. I elaborate on this in Section 5.4.

5.3 Soundness and event-based rule

In order to prove that the state-based proof rule is sound—i.e. if it establishes that a set of operations and a token system preserve the invariant, then all executions indeed preserve the invariant—they first define an *event-based* rule that generalizes the state-based one. They prove that the event-based rule is sound, and consequently that the state-based one is also sound. The event-based rule doesn't refer to specific tokens, but just executions and the "happens-before" partial order on events. Intuitively, the state-based rule is a specialization of the event-based one because tokens constrain the "happens-before" relation.

In order to state the rule, they have to specify the invariant and the guarantee relation on executions instead of states. Intuitively the rule first requires that an empty execution (where no operation has been performed) satisfies the invariant and that the guarantee relation preserves it. The final condition of the rule is that any event preserves the invariant if it is performed "in any of the replicas", that is if it is performed on a valid context of events that could have been performed in other replicas.

Having the event-based rule, they prove that it is sound, and then also prove that the state-based rule is sound by carefully constructing the invariant and guarantee relation on executions given the ones on states. The detailed definition of the event-based rule, as well as the soundness proofs can both be found in the original paper [4].

5.4 Discussion

The rule that is proposed in this work is modular, in the sense that one can reason about each operation in isolation, assuming that the other operations' behaviour is described by the guarantee relations. It also allows reasoning about states that represent a single database copy, and not fragments of many copies. However, the modularity claim has to be taken with a grain of salt. Even though it allows for cleaner reasoning, operations still affect each other. Making one operation acquire one less token might require adding more tokens to other operations, and vice versa. Therefore, complete modularity cannot be achieved as it is unavoidable to reason about the interactions of all pairs of operations in the general case.

In this work together with the proof rule they also develop a tool that automatically proves the necessary conditions establishing that a given token system and a set of operations satisfy the invariant. However, one still has to provide the guarantee relations, which can be somewhat complicated. The examples in their evaluation section all have simple guarantee relations, but one could expect that more complex operations (or stricter synchronization requirements) could require more elaborate guarantee relations that are difficult to come up with.

6 Comparison

Having described the three different consistency notions in detail, the goal of this section is to identify important axes of comparison for the different consistency notions, and compare them in detail. There are several dimensions in which these works differ. The most important ones are (i) formalization, (ii) expressiveness, (iii) automation, and (iv) implementation and performance. A diagram summarizing the comparison results is shown in Figure 1.

6.1 Formalization

This refers to whether the consistency notions are formally defined and whether conditions are identified for the safety of each configuration. Having a formal definition of the consistency notion allows for reasoning

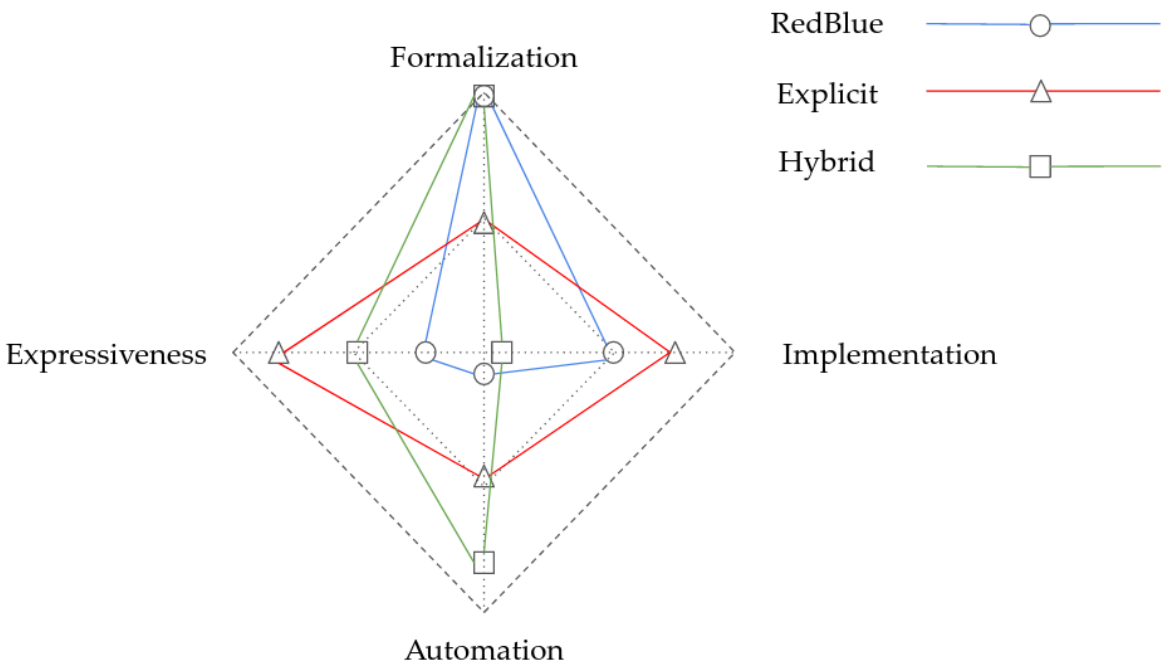

Figure 1: A summary of the qualitative comparison.

about their guarantees, and is a necessary prerequisite for any form of sound automatic support for checking whether a specific configuration is safe.

RedBlue It is formally defined together with conditions that dictate which operations must be labelled as red, and which can labelled blue.

Explicit It is defined formally, but the definition is abstract as it refers to invariants and not executions. More precisely, the full extent of explicit consistency (which includes fine-grained reservations) is not formalized. Because of that, there are no clear conditions that determine when a configuration is safe, i.e. it preserves the given invariants.

Hybrid It is formally defined and includes a proof rule that establishes when a specific configuration is safe. The proof rule is formally proven sound.

6.2 Expressiveness

This refers to the range of behaviors that each consistency notion support. A more expressive consistency notion allows for finer-grain configuration, which can lead to better performance without sacrificing correctness. For example, causal consistency is strictly less expressive than all three notions, since they can all be configured to offer causal consistency. More precisely, causal consistency is the weakest consistency notion that all three of the notions can provide.

RedBlue RedBlue consistency is the least expressive of the three, as it only allows operations to be totally ordered (red), or completely unordered (blue).

Explicit It is strictly more expressive than both RedBlue and Hybrid consistency since it support finer-grain control using the escrow and partition reservations. If restricted to only allow for "exclusive allow" reservations, it supports the same behaviors as RedBlue, and if restricted to only multi-level lock reservations it supports the same behaviours as Hybrid consistency.

Hybrid It is more expressive than RedBlue, but less expressive than Explicit consistency.

6.3 Automation

Automation has to do with the amount of additional effort that is needed from a user to choose a safe configuration. A completely automatic method would only require specifying the invariants on the datastore.

RedBlue They do not propose an automatic technique for determining whether a configuration is safe. In the original paper [9] they just provide conditions to determine whether a configuration is safe. However, in followup work [10] they developed a semi-automatic technique that can help determine whether a configuration preserves given data invariants. More precisely, the technique requires just a small amount of annotations from the user.

Explicit They develop a static analysis that determines whether operations need to be synchronized in order for invariants to be preserved. The static analysis is not completely automatic, in the sense that it requires that the developer annotates all of the operations with postconditions describing their effects. However, their technique is not extended to support the reservation choices, which are the reason that Explicit consistency is more expressive than the other notions.

Hybrid They develop a proof rule that establishes whether a specific configuration satisfies given integrity invariants. The only requirement from the user is to define specific guarantee relations for the operations that acquire each token.

6.4 Implementation

This refers to whether a consistency notion is implemented in a system, and the performance that it offers. The more general a configurable consistency notion is, the more difficult it is to implement it efficiently.

RedBlue RedBlue consistency is implemented in a system called *Gemini*. The system is evaluated in different scenarios and is shown to scale well. The system is not compared with previous solutions in loads that only require strong consistency guarantees (all operations labeled red) or just causal guarantees (all operations labeled as blue).

Explicit Explicit consistency is implemented in a system called *Indigo*. They evaluate the system by comparing its performance with their own versions of strong, causal, and RedBlue consistent systems. Their evaluation shows that in certain cases, the finer-grain control that is offered by reservations can lead to significant performance benefits.

Hybrid The general Hybrid consistency notion is not implemented in a system and it is not clear if it can be efficiently implemented in its full extent. However, the scope of this work is to offer automatic support to developers to choose a consistency notion configuration that preserves a given invariant on the data.

6.5 Discussion

In addition to the main comparison dimensions that were described above, the proposed consistency notions have some subtle differences and assumptions that do not fit in the above axes but are worth exploring.

More precisely, all three notions assume convergence. In the general case, this assumption can be satisfied if all operations commute. In RedBlue and Hybrid consistency, they relax this condition to only hold for operations that are not synchronized; in the case of RedBlue all blue operations, and in the case of Hybrid consistency, all operations that don't acquire conflicting tokens. On the other hand, in Explicit Consistency it is left slightly unclear how to establish convergence in the case of complex combinations of different reservations.

Another interesting point is that the conditions and proof rules for both the RedBlue and the Hybrid consistency notions are modular. This allows for reasoning about a single instance of the data store state and a single operation at a time. In contrast, this is not the case for Explicit consistency, as one has to reason about what reservations each replica holds to ensure that the data invariants are preserved.

7 Future work

There are a lot of interesting research directions that are enabled by the existing work on configurable consistency. I identify and describe some of them below:

Better automation All of the described works offer some form of automation but still require user input. An interesting direction for future work would be to investigate ways of requiring even less user input. For example, automatically discovering the guarantee relations in the work by Gotsman et al. [4] using some form of static analysis.

Dropping the causality assumption All three of the presented consistency notions build on top of causal consistency, not supporting anything weaker. The main reason is that it significantly simplifies the model and reasoning. It would be interesting to weaken the base consistency to eventual, as this could potentially allow for even better performance benefits in cases where causality is not essential.

Extending support for other properties All three of the presented works assume that the correctness specification is an integrity invariant on the data of the datastore. However, this is not the only specification that a user might have. More precisely, one could think of observational properties (i.e. regarding the responses that clients get), or safety properties that refer to a sequence of database states.

Hybrid consistency data types Conflict-free replicated data types (CRDTs) are restricted data types that can be implemented on eventually consistent datastores but offer strong guarantees [13]. It would be interesting to explore extending CRDTs with more complex operations that can be implemented on top of datastores that support hybrid consistency notions.

References

[1] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741972. URL `https://doi.org/10.1145/2741948.2741972`.

[2] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *European Symposium on Programming*, pages 67–86. Springer, 2012.

[3] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[4] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 371–384, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837625. URL `https://doi.org/10.1145/2837614.2837625`.

[5] Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

[6] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

[7] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691, 1979.

[9] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li`.

[10] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2`.

[11] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[12] Patrick O'Neill. The escrow transaction method. *ACM Trans. Database Syst*, 11(4), 1986.

[13] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[14] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, Petros Maniatis, et al. Zeno: Eventually consistent byzantine-fault tolerance. In *NSDI*, volume 9, pages 169–184, 2009.

[15] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[16] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 309–324, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522731. URL `https://doi.org/10.1145/2517349.2522731`.

[17] Robbert Van Renesse, Kenneth P Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[18] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[19] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*, page 21. USENIX Association, 2000.