Eventual Consistency Formalized

Edel Sherratt[1] and Andreas Prinz[2]

[1] Department of Computer Science, Aberystwyth University,
Aberystwyth SY23 3DB, Wales, UK
eds@aber.ac.uk
[2] Department of ICT, University of Agder, Grimstad, Norway
Andreas.Prinz@UIA.no

Abstract. Distribution of computation is well-known, and there are several frameworks, including some formal frameworks, that capture distributed computation. As yet, however, models of distributed computation are based on the idea that data is conceptually centralized. That is, they assume that data, even if it is distributed, is consistent. This assumption is not valid for many of the database systems in use today, where consistency is compromised to ensure availability and partition tolerance. Starting with an informal definition of eventual consistency, this paper explores several measures of inconsistency that quantify how far from consistency a system is. These measures capture key aspects of eventual consistency in terms of distributed abstract state machines. The definitions move from the traditional binary definition of consistency to more quantitative definitions, where the classical consistency is given by the highest possible level of consistency. Expressing eventual consistency in terms of abstract state machines allows models to be developed that capture distributed computation and highly available distributed data within a single framework.

Keywords: Distributed state · Eventual consistency · Formality · Abstract State Machine.

1 Introduction

Over time, and particularly since the invention of the Internet, computation has more and more become distributed. Today, almost all computation is achieved by cooperating computing entities communicating over a network. There are numerous frameworks to support distributed computing, and formal methods have provided the means to study distributed computation in depth.

In the public Internet, data is central to computation. This represents a change from the earliest days, where computers were thought of primarily as computing engines. Nowadays, persistent data forms the heart of almost all computation. In terms of distribution, the leading idea is still that we want to distribute the computation. This means both distribution of processing and distribution of data.

Current formal models of distributed computation [2,10,11,17,18,22,23,24] rely on data given by a centralized state, i.e. data that is *not* distributed. However, reality is different [1,26]. This is also captured in the Java memory model [20]. Data really is distributed and it is possible to have several copies of the same data that are not consistent.

This means we need to come to an agreement about distributed state in addition to distributed computation, see also [25]. Both of them are closely connected. When we talk about distributed state, we want to do this on the right level of abstraction. This is needed in order to be able to handle the complexity involved. The abstraction cannot mean that we have to look into all the existing copies of data entities and their updates by connected servers. This is obviously too detailed. It is also not enough to look at a centralized data model - that would be too coarse. The right level of abstraction lets us see possible states of the data without going too deep into detail of how these come about.

In noSQL databases, such an understanding is evolving and it is revolving around the term of *eventual consistency*. This means that data can be stored in the system with a certain amount of inconsistency, but that this is resolved over time and finally, the system is consistent. This paper aims at making this idea more formal, based on an understanding of state changes in the presence of distributed data.

We base our discussion on the model of Abstract State Machines (ASM) [8], because they provide a high-level and abstract view of computation. ASM can be considered formalized pseudo-code, such that ASM programs are readable even without much introduction. ASM are using a centralized state model, and we will combine this model with data that is distributed over several locations.

This paper starts with an introduction of eventual consistency and abstract state machines in Section 2. In Section 3, we define distributed state in ASM. After than, we look into ways to quantify inconsistency in Section 4. Afterwards, we define eventual consistency in Section 5. We discuss related work in Section 6 before we conclude in Section 7.

2 Background

2.1 Distributed Data and Eventual Consistency

A first idea of handling data for distributed computation is to store it in one place, and allowing multiple agents at different locations to access this data. This kind of data is called centralized data and is how data is managed in classical relational databases. Coordination of access to the data and timely fulfilment of client requests relies on effective transaction management. Enlarging the database is costly, as it normally requires migrating to a larger, expensive database server.

An alternative to the single-server, centralized relational database is a distributed database, where as well as splitting the data across servers, some data items are replicated. Distribution is designed to optimize performance against

expected client requirements, and extra processing is needed to ensure that replicated data remains consistent. In this way, we still keep the conceptual idea of centralized data, and leave detailed management of distributed, possibly replicated, data to the implementation. This is the place where the classical SQL databases are. From a users point of view, such a database should work as if the data was stored in one place. The famous ACID (atomicity, consistency, isolation, durability) rules for transaction processing ensure that this is the case.

The implementation itself will typically manage transactions so that a change is not committed before enough information is put into the system. This way, conflicting changes are avoided. Here, it is possible to choose whether the complexity should be on the read or on the write or shared between the two. It is important to notice that the read and write activities are finished when the database is in a consistent state and the returned value is correct. However, in case of much access and poor connectivity, client applications may find it slow or impossible to use the data.

Finally, based on the kind of application, we can relinquish the demand for absolute consistency. Now we are working with a truly distributed system, which is sometimes not connected, and which handles the data such that the user is not aware of connectivity problems. Of course, in this model, conflicting changes are possible, and they have to be sorted out at some point in time. The typical method for sorting out conflicting changes is timestamps, i.e. the later one of two changes is the more current one. Now the complexity of handling updates is moved partly out of the agents, and dealt with by their environment[3]. This allows for quick access to the database, but it means that there has to be an underlying process of cleaning up the database while the agents are doing something else.

The term *eventual consistency* is often used to describe how consistency is compromised in NoSQL databases, but its roots go back to the creation of the internet domain name system (DNS) created by Paul Mockapetris[4] in 1983.

Here is one definition of eventual consistency.

> "Eventual consistency is a characteristic of distributed computing systems such that the value for a specific data item will, given enough time without updates, be consistent across all nodes[5]."

Eventual consistency compromises consistency in a distributed data store for availability and network partition tolerance. The need for compromise was famously articulated as the CAP Theorem or Brewer's Theorem [15] which states that you can have at most two of

- Consistency
- Availability
- Partition tolerance

[3] In the conceptually centralized model, the read and write handling is also moved out of the agent code, but it is still inside the agent activity, which is not completed until everything is sorted out.

[4] `https://internethalloffame.org/official-biography-paul-mockapetris`

[5] `https://whatis.techtarget.com/definition/eventual-consistency`

in a shared-data system. Conventional, relational database systems apply the ACID (atomicity, consistency, isolation, durability) rules for transaction processing to ensure consistency. They sacrifice partition tolerance or availability to ensure consistency.

Availability is not negotiable in the internet DNS, and distribution is also essential to ensure scalability, and so consistency is compromised in the DNS in order to ensure availability and network partition tolerance. Brewer [15] coined the term BASE (basic availability, soft state, eventual consistency) to describe the rules for transaction processing applied in the DNS and more recently in several NoSQL database systems.

This way, eventual consistency is tightly coupled with the concept of a soft state, i.e. a state that is changed even without user agent interactions. The soft state repairs consistency problems until the state becomes consistent.

2.2 Abstract State Machines

The basic definitions of locations and updates in abstract state machines (ASMs) are as follows. Variations of these definitions can be found in many sources, including, but not limited to [3,4,6,7,8,9].

At its most basic, an abstract state machine (ASM) consists of abstract states with a transition rule, or ASM program, that specifies how the ASM transitions through its states.

An ASM has a *signature* of symbols and a base set of values. The symbols of the signature are function symbols. Each function symbol has an arity. The symbols are interpreted over the base set so that a symbol with arity zero is interpreted as a single element of the base set, and a symbol with arity n is interpreted as an n-ary function over the base set. Expressions (terms) of the signature are constructed in the usual way, and are interpreted recursively over the base set.

Names can be classified with respect to change. Names like *True*, *False* and *undef*, whose interpretation is the same in all the states of an abstract state machine, are called *static names*, while all other names (called *dynamic* names) are subject to updates. Dynamic names can again be classified with respect to which agents are allowed to change them, which will be explained later. To support readability, new symbols can be defined as abbreviations for complex terms. In SDL such symbols are called *derived names* [19].

The signature includes the predefined names *True*, *False* and *Undefined*, and three distinct values of the base set serve as interpretations for these. The interpretations of *True* and *False* are called truth values. Function symbols whose interpretations deliver truth values are called predicate names.

Unary predicate names can serve as sort names, whose interpretations classify base set elements as belonging to the sort in question.

An interpretation of the symbols over the base set defines a *state* of the ASM.

An ASM program is composed of assignments, if statements, forall statements, and several more statement kinds. We will not formally introduce all

the kinds, but rather refer to [8]. ASM is designed to look like pseudo-code and normally ASM programs can be read without further explanation.

The main means of change in a program is an assignment, taking the form $exp := e$, meaning that the value (object) represented in a given state by the expression exp is changed to the value (object) represented in the given state by the expression e. More generally, expressions at the left-hand side of an update can take the form $exp = f(e_1, \ldots, e_n)$, where f is an n-ary function symbol and e_i are expressions.

The ASM model is a dynamic model. Starting from an *initial state*, an abstract state machine repeatedly produces new states from existing states by *updating* the interpretation of its symbols. Such a sequence of states is called a *run* of the ASM. The *transition* (or *move* or *step*) from one state to the next is specified as a set of *updates* to *locations*, where an update is the change to the current state imposed by an assignment.

More precisely, a function symbol f with a tuple of elements $\overline{a}$ that serves as an argument of f identifies a *location*. The term $f(\overline{a})$ identifies a location and evaluates to a value in a state. In a subsequent state, the value of that location may have changed, and $f(\overline{a})$ may evaluate to a new value. In that case, an *update* indicates what the new value will be, and is expressed using the values of terms in the current state. Updates are written as triples $(f, \overline{a}, b)$, to indicate that $f(\overline{a}) = b$ will be true in the new state.

2.3 Distributed ASM

The abstract state machine model is very flexible and only asserts that state changes are given based on the current state. From here, a natural extension is to look into several ASM agents, each with an ASM program providing state changes. The agents share the (global) state and they start in a common initial state. Because the agents only use part of the state, they would normally not see the complete state, but only the part that is visible based on their signature. Therefore, we can distinguish different kinds of functions names for a distributed agent: *monitored* functions are only read by the agent and updated by other agents or the environment, *controlled* functions are only visible to the agent itself and can be considered private, while *shared* functions are joint between different agents for reading and writing.

The execution model of ASM ensures that the agents do not clash in their updates of shared locations. The important idea here is that the underlying memory model is a global model with all locations being in principle available to all agents.

For the sake of the discussion in Sections 3 and 5, we also assume the availability of a synchronized global time, which is accessed using the monitored function NOW. In reality, it is not possible to completely synchronize time in a distributed system, but here it is enough when time drift between two agents is smaller than their communication delay. This can be achieved using the NTP protocol [21]. There is also an assumption that time-stamps are never accidentally the same. Although that seems like a strong assumption, it is easily

implemented by either sorting the servers issuing time-stamps and using this to sort out the time-stamp order, or to just use an ordering on the values to do the same. In practice, both these solutions, and others, are used [1]. This means we can safely assume that the time-stamps used by different agents are disjoint.

3 Abstract State Machine Model of Distributed State

3.1 Distributed, Duplicated Persistent Data

We start by looking at the user or client view of the database. The following definitions use the ASM method to model multiple agents that read, write, update and delete data that is duplicated, distributed and persistent. The definitions of communicating ASMs provided by Börger and Raschke [9], are modified to take account of duplicated data. This provides a basis for defining soft state as meaning that an update to a location is propagated to all copies. Based on this, several definitions of inconsistency are explored in Section 4.

Useful concepts that have already been developed, and that are relevant to consistency in a persistent data store with multiple ASM clients, include:

- persistent queries [5,6]
- independent concurrent ASMs [28]
- communicating ASMs [9]
- communicating concurrent ASMs with shared memory [27]

Consider a distributed algorithm with several ASM agents, where the agents' persistent data is stored in a distributed database. The database management system (DBMS) can be a classical distributed SQL or a distributed NoSQL DBMS. The DBMS distributes its data across a number of servers, which can be viewed as nodes in a network. Data and functionality is replicated across server nodes, and storage is increased by adding more nodes, an approach called *horizontal scaling*. To optimise availability and partition tolerance, the requirement for consistency is relaxed to a requirement for eventual consistency.

Actions are performed by ASM agents following their ASM programs. Client agents issue requests to the DBMS, and DBMS server agents retrieve or update some of the replicated data in response to the client request. DBMS server agents also generate requests to other servers to propagate updates so as to make updated values available on those other servers.

Each location has multiple copies (replicas) on different DBMS servers. We consider servers to be agents themselves.

domain *Location*
domain *Server* $\subseteq$ *Agent*
static *replicas: Location* $\rightarrow \mathcal{P}$ *Server*
shared *value: Server, Location* $\rightarrow$ *Value*
shared *timestamp: Server, Location* $\rightarrow$ *Time*

We also define the latest timestamp amongst all the replicas of a location.

derived *maxTime(loc)* $\equiv max_{r \in replicas(loc)}$ *timestamp(r,loc)*

We assume that different values for a location have different timestamps. This is obviously true when the different values come from different agents, as timestamps of different agents are disjoint. If the different values come from the same agent, then they have to come from different steps, as it is an inconsistency to assign two different values to the same location in the same step. However, because time advances, the timestamps of different steps of the same agent are different.

As discussed above, each location has a value that was allocated at a time that is universally comparable. Within a set of replicas, there will be a most recent update, defined as an update with the latest timestamp.

Server agents run a soft state update program which is detailed in the next subsection. The client programs run their code, which includes reads and writes of locations. This is the usual ASM handling, where for a given agent with its ASM program, an update set is determined. The update set is the set of writes, while the reading of values is given by the reads of locations. The replicas of shared variables are handled in the definition of read and write.

Connectivity between servers is of interest insofar as an update of a replica on one server is visible to another server. This is modelled using a monitored function *connected*.

monitored *connected : Agent, Agent* $\rightarrow$ *Boolean*

From the perspective of a client agent, database handling is an activity conducted by the environment, and the actions of DBMS agents are perceived as updates to shared or monitored locations.

Client agents can read values, where reading is a function providing a value. In reality, reading a value might be more than a function with an immediate outcome, and will rather have several steps that might or might not provide a result. For the discussion in this paper, it is sufficient to consider immediate results and keep a refinement into action sequences for later.

Reading in the presence of replicas means to read selected replicas and to use the value with the most recent time stamp among them. We abstract the possible replicas to be read with a predicate *ReadPolicyOK*, which represents an unspecified database policy that limits the subsets of replicas that need to be consulted for reading by a client agent.

static *ReadPolicyOK: $\mathcal{P}$Server, Location* $\rightarrow$ *Boolean*

Read(loc) $\equiv$
choose $S \subseteq$ *replicas(loc)*
with *ReadPolicyOK(S,loc)* $\wedge \forall s \in S$: *connected(SELF,s)*
in
choose $s_0 \in S$
with $max_{s \in S}$(*timestamp(loc,s)*) = *timestamp*(loc, s_0)

in
$value(s_0, loc)$

Client agents can also write a value, which means a state is changed. This way, writing is an activity. Again, writing will normally be an activity with many steps, which we abstract here to just one step. These updates bring all the replicas to the latest value. Like *ReadPolicyOK*, the function *WritePolicyOK* checks a subset of the replicas for validity to be updated, and allows write operations to be specified independently of the underlying database activity.

static *WritePolicyOK* : $\mathcal{P}$ *Server, Location* $\rightarrow$ *Boolean*

Write(loc,val) $\equiv$
choose $S \subseteq$ *replicas(loc)*
with *WritePolicyOK(S,loc)* $\wedge \forall s \in S$: *ConnectedReplica(SELF,s)*
in
forall $s \in S$ **do**
value(s,loc):= val
timestamp(s,loc):= NOW

As an example, in a client program there might be an assignment *x:= y+z*. This means, y and z are handled with *Read(y)* and *Read(z)*, respectively. The variable x gets a new value (lets assume 42) and this is handled with *Write(x,42)*.

An example for read and write policies could be that reading requires two replicas while writing requires all replicas. This would mean the following.

ReadPolicyOK_example(S,l) $\equiv S \subseteq$ *replicas(l)*$\wedge \mid S \mid \geq 2$
WritePolicyOK_example(S,l) $\equiv S =$ *replicas(l)*

3.2 Replicas and Updates

The DBMS has also an internal view on the data. DBMS agents handle the soft state and update locations in the background. The duplicates of the different locations are considered to be one from the client perspective, but the DBMS handles them individually and keeps consistency high. The state of the system and the different values present are later used to define eventual consistency.

We define a background process that improves consistency in the system by updating the values to newer versions. The abstract program that the DBMS agents are running in parallel to the client agents is as follows.

SoftStateUpdate $\equiv$
choose $l \in$ *Location* **with** *SELF* $\in$ *replicas(l)* **do**
choose $r \in$ *replicas*(l) **with** *SELF* $\neq a \wedge$ *connected*$(SELF, r)$
$\wedge$*timestamp(r,l)* $>$ *timestamp(SELF,l)*
do
value(SELF,l):= value(r,l)
timestamp(SELF,l):= timestamp(r,l)

4 Defining Inconsistency Formally

Eventual consistency means that from any starting state, in the absence of client-initiated updates, the system will reach a consistent state. Classical consistency is a qualitative measure, which can be true or false.

To model eventual consistency, we need a more quantitative measure of consistency as a way to express how far our system is from consistency. This will allow to express how a mechanism like *SoftStateUpdate* will, given sufficient time without client-initiated updates, cause the system to become consistent. Therefore, we do not measure consistency, but inconsistency.

In the following, we present alternative measures of inconsistency, some based on the count of outdated values, and others on the age of outdated values. To serve as a meaningful model of eventual consistency, a measure of inconsistency should have the following properties.

- P1: Inconsistency should decrease when values become less outdated. This could happen, for example, by a DBMS agent executing *SoftStateUpdate*.
- P2: Inconsistency should increase when a new value is introduced incompletely as a consequence of a client's request. In this case, the value is updated only in some replicas, which outdates the remaining values. Inconsistency should not change when consistent locations are updated consistently.
- P3: Changes to the network partitioning should not influence inconsistency. When replicas of a location become disconnected from one another, DBMS activities will not be able to reverse increases in inconsistency caused by client activities, such that the inconsistency does not decrease.

4.1 Total and Sufficient Consistency

Before defining measures of consistency, we first define total consistency, the ideal state which, given sufficient time, the system as a whole will reach in the absence of client-initiated updates. We then define sufficient consistency, a condition that means that each client read will yield the most recent value even though there might be inconsistencies in the data.

Total consistency is a state in which all the members in a set of replicas have equal values, and those equal values all have the latest timestamp. Please remember that the same timestamp implies the same value.

Definition 1 (Total consistency). *All replicas of a location have the same time stamp and the same value.*

$$\begin{aligned} &\mathit{TotallyConsistent}(\mathit{loc}) \equiv \\ &\quad \forall s_0 \in \mathit{replicas}(\mathit{loc}) \bullet \mathit{maxTime}(\mathit{loc}) = \mathit{timestamp}(s_0, \mathit{loc}) \end{aligned}$$

Sufficient consistency is then defined as stating that there might be different values for a location in the system, but these are not visible to clients due to the read policy.

Definition 2 ((Sufficient) consistency). *All possible reads of a location lead to the same, most recent, available value.*

$$\begin{aligned}&Consistent(loc) \equiv \\ &\quad \forall S \subseteq replicas(loc) \bullet ReadPolicyOK(S,loc) \rightarrow \\ &\qquad maxTime(loc) = max_{s \in S}(timestamp(s,loc))\end{aligned}$$

4.2 Consistency as a Count of Inconsistent Replicas

In a distributed database system, consistency means having the same value for each location regardless of which server provides the value. For such a system, the number of extra, inconsistent replicated values in the system is a good measure of inconsistency. These extra values are the extra values that might be externally available to client agents. For distributed databases, there might be other hidden values in the system that are not yet presented to the users before their update operation is finished, but such values are at the level of the DBMS implementation and will not be considered further here.

Definition 3 (Outdated Values). *The measure of consistency in a distributed system is the total number of outdated values in the system. These are values that do not have the latest timestamp.*

$$\begin{aligned}&OutdatedValueCount(loc) \equiv \\ &\quad |\, \{r \in replicas(loc) : timestamp(r,loc) \neq maxTime(loc)\} \,|\end{aligned}$$

$$OutdatedValues \equiv \sum_{l \in Location} OutdatedValueCount(l)$$

When an old value is updated to the latest value, $OutdatedValues$ decreases, as needed for P1. It does not decrease when the update goes to a new, but not the latest value. P2 is true as an incomplete client-initiated update will cause $OutdatedValues$ to increase. The measure is independent of the network thus making P3 trivially true.

A second measure of consistency is presented below that takes account of the connectivity of the servers. It is not meaningful to expect nodes to be updated as long as they are disconnected from the current value.

Definition 4 (Outdated Reachable Values). *The measure of consistency in a distributed system is the number of outdated values that are reachable from the most up-to-date replicas but that are not (yet) up to date.*

$$\begin{aligned}&OutdatedReachableValueCount(loc) \equiv \\ &\quad |\, \{r \in replicas(loc) : timestamp(r,loc) \neq maxTime(loc) \wedge \\ &\qquad \exists\, r' \in replicas(loc) \bullet timestamp(r', loc) = maxTime(loc) \wedge \\ &\qquad connected(r', r)\} \,|\end{aligned}$$

$$\begin{aligned}&OutdatedReachableValues \equiv \\ &\quad \sum_{l \in Location} OutdatedReachableValueCount(l)\end{aligned}$$

Corollary 1.
$OutdatedReachableValues \leq OutdatedValues$

When everything is connected, this measure is the same as the previous one. When some parts are disconnected, then $OutdatedReachableValues$ captures the connected part of the network. Again, P1 and P2 are true as long as they are in the connected parts, as we can assume that values cannot be updated in the unconnected parts. Still, $OutdatedReachableValues$ is not a satisfactory measure of inconsistency because partitioning the network actually leads to increased consistency according to this measure, making P3 invalid.

Therefore we consider a measure that takes account of network partitioning.

Definition 5 (Outdated Isolated Values). *The measure of inconsistency in a distributed system is the number of outdated values that are isolated from the most recent update of a location, and so cannot be made consistent by the DBMS propagation mechanism characterized by $SoftStateUpdate$.*

$$
\begin{aligned}
&OutdatedIsolatedValueCount(loc) \equiv \\
&\quad | \{r \in replicas(loc) : timestamp(r,loc) \neq maxTime(loc) \wedge \\
&\qquad \exists r' \in replicas(loc) : timestamp(r', loc) = maxTime(loc) \wedge \\
&\qquad \textbf{not } connected(r', r)\} |
\end{aligned}
$$

$$OutdatedIsolatedValues \equiv \sum_{l \in Location} OutdatedIsolatedValueCount(l)$$

Corollary 2.
$OutdatedIsolatedValues \leq OutdatedValues$

Corollary 3.
$OutdatedIsolatedValues + OutdatedReachableValues = OutdatedValues$

Inconsistency as measured by $OutdatedIsolatedValues$ will not increase as a consequence of DBMS activities. However, the measure does not capture the fact that DBMS activities will increase consistency within connected parts of the system. Thus, P1 is not true. In fact, there are cases where also P2 and P3 are invalid for $OutdatedIsolatedValues$.

Combining $OutdatedIsolatedValues$ with $OutdatedReachableValues$ provides a measure that fulfils all three of the required properties of a meaningful measure of inconsistency. $OutdatedValues$ is such a combination and therefore provides the most meaningful of the measures of inconsistency explored above.

4.3 Time-based Measures of Inconsistency

Instead of counting inconsistent replicas of locations, the measures explored below describe inconsistency in terms of the time delay of the inconsistent replica values.

Definition 6 (Least Consistency Time). *For a location l, the consistency time is NOW in case the location is consistent. Otherwise, it is the latest timestamp with a value for this location. This leads to a measure of consistency that is the longest distance between NOW and the consistency times of all the locations in the system.*

ConsistencyTime(loc) ≡
  if *Consistent(loc)* **then** *NOW* **else** *maxTime(loc)*

$LeastConsistencyTime \equiv min_{l \in Location}(NOW - ConsistencyTime(l))$

When an inconsistent location is made consistent, *LeastConsistencyTime* decreases, as needed for P1. It does not decrease when the update goes to a new, but not the latest value. It also does not decrease when there are still other outdated values around for the location. P2 is not true, because applying an incomplete client-initiated update to an inconsistent state will cause *LeastConsistencyTime* to decrease. As the measure is independent of the network, P3 is valid.

As this definition uses only the oldest update to a location, it does not take into account changes to other outdated values. An alternative time-based measure of inconsistency is the distance in time to the latest timestamp for all the replicas of a location. This is a measure of how out of date the replicas are.

Definition 7 (Delta Consistency Time). *The timestamp differences for all the replicas of a location.*

ConsistencyDelta(loc) ≡
  $\sum_{r \in replicas(loc)} maxTime(loc) - timestamp(r,loc)$

$DeltaConsistencyTime \equiv \sum_{l \in Location} ConsistencyDelta(l)$

DeltaConsistencyTime is a measure of the delay in propagating updates across all the replicas of a location. It improves with each improvement for any outdated value, such that P1 is true for all cases. It also fulfils P2 in all cases. Finally, as it is not considering the network, it also fulfils P3. *DeltaConsistencyTime* is more detailed than the other inconsistency measures.

5 Formal Definition of Eventual Consistency

5.1 Eventual Consistency

The previous definitions allow a formalization of eventual consistency as follows.

Definition 8 (Eventual Consistency). *A DBMS, in particular its soft state update functionality, is eventually consistent when its* DeltaConsistencyTime *is decreasing if there are no client-initiated updates and* OutdatedReachableValues *is not zero.*

This definition of eventual consistency demands that in the absence of other updates, the predicate *TotallyConsistent* will eventually hold for all locations in the system. In reality, the data is already consistent when the predicate *Consistent* holds, such that users will experience consistency earlier than that.

This is related to the other parts of the DBMS who provide as much as possible consistency already without the soft state.

The definition of eventual consistency and the use of *DeltaConsistencyTime* in particular also allows a comparison of different DBMS mechanisms and database configurations with respect to consistency.

Please note that this definition of eventual consistency does not define the consistency of a state, but the possible behaviours of a DBMS.

5.2 *SoftStateUpdate* Implies Eventual Consistency

Theorem 1. *The abstract soft state update functionality given in Section 3 provides eventual consistency. This means, in the absence of client-initiated updates,* DeltaConsistencyTime *decreases due to DBMS propagation of updated values across replicas.*

Proof. Let $OutdatedReachableValues > 0$ [6]. This means that there is at least one location l_0 that has an outdated value. Let r_0 be a replica with the up-to-date value, and r_1 be the outdated replica. We can choose r_0 and r_1 such that $connected(r_0, r_1)$ because $OutdatedReachableValues > 0$.

Now the conditions for l, r, and $SELF$ in *SoftStateUpdate* are fulfilled by l_0, r_0, and r_1. *SoftStateUpdate* will run on r_1, because it is a DB server and keeps replicas. This means that *SoftStateUpdate* for r_1 does not produce an empty update set, but changes the value of at least one location. This will decrease *DeltaConsistencyTime* by the time difference between the old and the new value. □

5.3 Example for Eventual Consistency

We consider the independent read - independent write (IRIW) algorithm A [10] with four agents $a_1, \ldots, a_4$ as follows.

a_1 : $x := 1$
a_2 : $y := 1$
a_3 : *Read(x); Read(y)*
a_4 : *Read(y); Read(x)*
initially $x = y = 0$

We consider three database servers db_1, db_2, db_3, keeping a replica of x and y each and running the *SoftStateUpdate* program. The user agents use the *Read* for reading values (a_3 and a_4) and the *Write* for storing the assignments of values

[6] Obviously, this also implies that $DeltaConsistencyTime > 0$

(a_1 and a_2). We visualize the system state by the three replicated values for x and y, such that the initial state is $(x = (0, 0, 0), y = (0, 0, 0))$.

First, we look at a database policy where writing and reading are allowed already on just one replica. We run first a_1 and a_2 in parallel at time 1. They store the values for x and y in db_1 and db_2, respectively. This leads to the system state $(x = (1, 0, 0), y = (0, 1, 0))$ with $DeltaConsistencyTime = 4$. Now a_3 and a_4 read in parallel, where a_3 consults db_1 and a_4 consults db_2. This gives the result $x = 1, y = 0$ for a_3 and $x = 0, y = 1$ for a_4. As [10] argues, this is not sequentially consistent. However, still db_1, db_2, and db_3 are active with *SoftStateUpdate*. All of them can update at least one location, leading to $(x = (1, 1, 1), y = (1, 1, 0))$ and $DeltaConsistencyTime = 1$. The server db_3 can do one more update step, before everything is consistent.

Now we look at the case where two replicas are needed with reading and for writing. The previous scenario is not possible now, as now the system is sequentially consistent. Still, we run first a_1 and a_2 in parallel. They store the values for x and y as follows: $(x = (1, 1, 0), y = (0, 1, 1))$ with *DeltaConsistencyTime = 2*. Now a_3 and a_4 read in parallel, and independent of their choice of servers, they both get the result $x = 1$, $y = 1$. The server db_2 is already up-to-date, but db_1 and db_3 run *SoftStateUpdate*, leading to $(x = (1, 1, 1),\ y = (1, 1, 1))$ and *DeltaConsistencyTime = 0*.

Please observe that in the second case, the system is consistent all the time, even though *DeltaConsistencyTime* is not 0. However, the soft state functionality does not stop before everything is updated to the latest value.

In the case of partitions, not all needed updates are possible and have to be delayed until the connection is restored.

6 Related Work

Bosneag and Brockmeyer [12] developed a formalism that enabled specification of different forms of consistency for a given data object. Like the work presented here, their approach is rooted in state machine models. Eventual consistency is defined as the fact that in the absence of updates, all replicas of a data item converge towards identical copies of each other. A history reduction operator is defined based on whether or not operations in a history can be reordered without affecting the end state that is reached, and a proof is given that any algorithm that respects the history reduction operator will achieve eventual consistency.

The definitions provided here differ from [12] in that they deal with a measure of distance from consistency rather than on dependencies between operations. We maintain that this more abstract view of eventual consistency, which does not need to refer to execution traces, provides a better basis for reasoning about whether or not a database management system can be said to ensure eventual consistency.

Burckhardt [16] provides ways to reason about the consistency of protocols in terms of consistency guarantees, ordering guarantees and convergence guarantees. Reasoning is in terms of states, where the current state is viewed as a graph

of prior operations. Formal models are presented for protocol definitions and for executions in distributed systems, and proofs are provided to show that implementations meet consistency guarantees. This goes far beyond the definitions presented here, but has the disadvantage that it relies on a specially developed formalism for specifying the observable behaviour of a system. Our work has the advantage that it builds on the established ASM formalism, and so can be used immediately to reason about existing specifications.

Bouajjani, Enea and Hamza [13,14] define eventual consistency as a property over traces observed by an external witness. Eventual consistency is grounded in the notions of safety and liveness, and is defined in terms of finite prefixes of a global interpretation of method calls in a system where the result of a call is well-defined (safety), and where there exists a global interpretation of all the method calls in an infinite trace. This facilitates reasoning about speculative updates and rollbacks, both of which are essential to a practically useful definition of eventual consistency. Again, our definitions do not refer to histories or execution traces, and so provide a more appropriate level of abstraction for reasoning about eventual consistency than was previously available.

In summary, the work presented here differs from previous work in that it builds on an existing ASM formalism and so can be used to reason about existing specifications without the need to translate those into a new formalism. It differs also in that it focuses on measures of distance from the desired state of consistency rather than on execution traces of a distributed system. This enables reasoning about consistency at a more appropriate level of abstraction than was previously possible. Finally, the ASM formalism also provides the semantic foundation for SDL [18], which provides an opportunity to provide automated support for verifying eventual consistency by building on existing tools.

7 Conclusion

Different definitions aimed at quantifying consistency in a distributed database with replicated data were presented above.

Some of the definitions are based on counts of inconsistent replicas of locations in the distributed system. These definitions capture the idea that client-initiated updates will make the database system less consistent, and that DBMS activities to propagate updates across replicas will make the system more consistent.

Other definitions are based on calculating how out-of-date some replicas are. Those definitions also capture the concept of a distributed database system that becomes less consistent with client updates and more consistent as updates are propagated by the DBMS.

However, none of the definitions fully captures the implications of network failure and partitioning. This is not necessarily a problem, but indicates that further metrics are needed to cover those aspects of distributed databases with eventual consistency.

For sure, distributed persistent data is essential to distributed computation, and the ASM formalism supports formal modelling of persistent distributed datastores with replicated values. Moreover, it enables such datastores to be seamlessly modelled alongside independent client and DBMS server agents.

Overall, a foundation has been laid to conduct deeper investigation of eventual consistency than has previously been possible.

References

1. Apache Software Foundation: Apache Cassandra 4.0 – Web Page and Documentation. http://cassandra.apache.org/ (2019)
2. Best, E.: Semantics of sequential and parallel programs. Prentice Hall (1996)
3. Blass, A., Gurevich, Y.: Ordinary interactive small-step algorithms i. ACM Transactions on Computational Logic **7**(2), 363–419 (2006)
4. Blass, A., Gurevich, Y., Rosenzweig, D., Rossman, B.: Interactive small-step algorithms ii: Abstract state machines and the characterization theorem. Logical Methods in Computer Science **3**(4) (2007). https://doi.org/10.2168/LMCS-3(4:4)2007
5. Blass, A., Gurevich, Y.: Persistent queries. CoRR **abs/0811.0819** (2008), `http://arxiv.org/abs/0811.0819`
6. Blass, A., Gurevich, Y.: Persistent queries in the behavioral theory of algorithms. ACM Transactions on Computational Logic (TOCL) **12**(2), 16:1–16:43 (2011). https://doi.org/10.1145/1877714.1877722
7. Börger, E., Cisternino, A. (eds.): Advances in Software Engineering: Lipari Summer School 2007, Lipari Island, Italy, July 8-21, 2007, Revised Tutorial Lectures, vol. 5316. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89762-0
8. Börger, E., Stärk, R.: Abstract State Machines – a Method for High-Level System Design and Analysis. Springer-Verlag Berlin Heidelberg New York (2003)
9. Börger, E., Raschke, A.: Modelling Companion for Software Practitioners. Springer-Verlag GmbH (2018). https://doi.org/10.1007/978-3-662-56641-1
10. Börger, E., Schewe, K.D.: Concurrent abstract state machines. Acta Inf. **53**(5), 469–492 (2016). https://doi.org/10.1007/s00236-015-0249-7
11. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
12. Bosneag, A.M., Brockmeyer, M.: A unified formal specification for a multi-consistency replication system for dhts. In: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05). pp. 33–40. IEEE (2005)
13. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 285–296. POPL '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2535838.2535877
14. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. ACM SIGPLAN Notices **49**(1), 285–296 (2014). https://doi.org/10.1145/2578855.2535877
15. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. pp. 7–. PODC '00, ACM, New York, NY, USA (2000). https://doi.org/10.1145/343477.343502

16. Burkhardt, S.: Principles of eventual consistency. Foundations and Trends in Programming Languages **1**, 1–150 (2014)
17. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (Aug 1978). https://doi.org/10.1145/359576.359585
18. ITU-T: Specification And Description Language SDL (Z.100 Series). International standard, International Telecommunication Union, Telecommunication Standardization Sector (2016–2018)
19. ITU-T: Specification and Description Language Overview of SDL-2010, Annex F1: SDL-2010 formal definition: General overview. International standard, International Telecommunication Union, Telecommunication Standardization Sector (2016–2018)
20. Manson, J., Pugh, W., Adve, S.V.: The java memory model. SIGPLAN Not. **40**(1), 378–391 (2005). https://doi.org/10.1145/1047659.1040336
21. Mills, D.L.: A brief history of ntp time: Memoirs of an internet timekeeper. SIGCOMM Comput. Commun. Rev. **33**(2), 9–21 (2003). https://doi.org/10.1145/956981.956983
22. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1982)
23. Mironov, A.M.: Theory of processes. CoRR **abs/1009.2259** (2010), `http://arxiv.org/abs/1009.2259`
24. Object Management Group (OMG): OMG® Unified Modeling Language® (OMG UML®), Version 2.5.1. OMG Document Number formal/2017-12-05 (`http://www.omg.org/spec/UML/2.5.1`) (2017)
25. Prinz, A.: Distributed computing on distributed memory. In: Khendek, F., Gotzhein, R. (eds.) System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering. pp. 67–84. Springer International Publishing, Cham (2018)
26. Prinz, A., Sherratt, E.: Distributed ASM – pitfalls and solutions. In: Aït-Ameur, Y., Schewe, K.D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM and Z – Proceedings of the 4th International Conference (ABZ 2014). LNCS, vol. 8477, pp. 210–215. Springer (2014)
27. Schewe, K., Prinz, A., Börger, E.: Concurrent computing with shared replicated memory. CoRR **abs/1902.04789** (2019), `http://arxiv.org/abs/1902.04789`
28. Sherratt, E.: Relativity and abstract state machines. In: System Analysis and Modeling: Theory and Practice - 7th International Workshop, SAM 2012, Innsbruck, Austria, October 1-2, 2012. Revised Selected Papers. pp. 105–120 (2012). https://doi.org/10.1007/978-3-642-36757-1_7