On the behaviour of general-purpose applications on cloud storages ⋆

Laura Bocchi[1] and Hernán Melgratti[2]

[1] Imperial College London
[2] University of Buenos Aires

Abstract. Managing data over cloud infrastructures raises novel challenges with respect to existing and well studied approaches such as ACID and long running transactions. One of the main requirements is to provide availability and partition tolerance in a scenario with replicas and distributed control. This comes at the price of a weaker consistency, usually called eventual consistency. These weak memory models have proved to be suitable in a number of scenarios, such as the analysis of large data with Map-Reduce. However, due to the widespread availability of cloud infrastructures, weak storages are used not only by specialized applications but also by general purpose applications. We provide a formal approach, based on process calculi, to reason about the behaviour of programs that rely on cloud stores. For instance, one can check that the composition of a process with a cloud store ensures 'strong' properties through a wise usage of asynchronous message-passing. Also, the compositionality of process calculi enables one to define/verify middleware-services (interfacing the weak data storage with the application) to provide 'strong' properties in a transparent way to the application.

1 Introduction

In the past decade, the emergence of the Service-Oriented paradigm has posed novel requirements in transaction management. For instance, the classic notion of ACID transaction (i.e., providing Atomicity, Consistency, Isolation and Durability) proved to be unsuitable in loosely coupled multi-organizational scenarios with long lasting activities. Both industry and academia found an answer to these novel requirements in a weaker notion of transaction, Long Running Transactions (LRTs).

At present, the increasing availability of resources offered by cloud infrastructures enables small and medium-sized enterprises to benefit from IT technologies on a pay-per-use basis, hence with no need of high up-front investments. The range of available resources, offered as services, include e.g., applications (Software as a Service), development platforms (Platform as a Service), hardware components (Infrastructure as a Service), and data storage. Providing and managing data over cloud infrastructures poses yet novel challenges and requirements to data management.

⋆ This work has been partially sponsored by the project Leverhulme Trust award Tracing Networks, Ocean Observatories Initiative, ANPCyT Project BID-PICT-2008-00319, EU 7FP under grant agreement no. 295261 (MEALS).

Whereas the main issue in LRTs, with respect to the ACID properties, is to minimise resource locks by dropping isolation, the problematic properties in cloud databases are *durability* and *consistency*. Durability is dropped in favour of a *soft state* (i.e., data is not preserved unless its persistence is explicitly 'renewed' by the user), whereas consistency is relaxed in order to guarantee availability. [15] nicely summarises these new requirements with the acronym BASE (Basically Available, Soft State, Eventual Consistency), as opposed to ACID. In this paper we focus on consistency, leaving the consideration of soft state as a future work.

Cloud infrastructures provide data storages that are virtually unlimited, elastic (i.e., scalable at run-time), robust (which is achieved using replicas), highly available and partition tolerant. It is known (CAP theorem [8]) that one system cannot provide availability, partition tolerance and consistency, but has to drop one of these properties. Cloud data stores typically relax consistency, while providing a weaker version called *eventual consistency*. Eventual consistency ensures that, although data consistency can be at time violated, at some point it will be restored.

Although not appropriate in all scenarios, BASE properties have been introduced because they are the most practicable solution in a number of scenarios. In other words, eventual consistency is *suitable sometimes*. In fact, some applications e.g., some banking applications, need consistency, whereas some others can provide a satisfactory functionality also when consistency is relaxed e.g., YouTube file upload.

In this paper we set basis for a formal analysis of what 'suitable' and 'sometimes' means, so that general purpose applications can safely be run using these weaker memory models. In fact, the widespread availability of Cloud infrastructures makes it crucial to support cloud data storages, not only by specialized applications, such as the analysis of large data using Map-Reduce, but also by general purpose applications.

We argue that it is crucial to provide modelling primitives and analysis tools that suit the specific (BASE) properties. The aim of this paper is to offer the basis for a testing theory, based on process calculi, that enables to verify that an application 'works fine' when composed with a weak storage. In particular, we focus on distributed and interoperable applications that can communicate asynchronously, other than using cloud storages. We model applications as processes in CCS [14], a well known formalism for concurrent and distributed systems.

The technical contributions are: one abstract (Section 2.1) and one operational (Section 2.2) characterisation of stores, strong consistency and weak consistency, and an approach (Section 4) that allows us to model and compare applications on stores. This approach is based on the value-passing CCS and on the operational characterisation of stores given in Section 2.2, and uses a behavioural preorder that takes into account both the behaviour of the applications, modelled as processes, and the levels of consistency of the stores they use. In Section 3 we provide two examples of stores with replicas and distributed control and one strong store, and analyse the consistency properties they provide. These stores are also used to illustrate the proposed approach in Section 4.

2 Strong and Weak consistency

This section presents a formal approach to modelling cloud storages, which we will simply refer to as *stores*. Firstly, Section 2.1 presents stores as abstract data types, whose operations come equipped with a denotational semantics. Section 2.2, presents the operational characterisation of stores that will be useful in the remaining of this paper.

2.1 Stores as abstract data types

Let $\mathbb{K}$ be the set of names, ranged over by $o, o', o_1, \ldots$, used to uniquely identify the objects in a store (e.g., URIs). We interpret any state σ of a store as a total function that associates keys $o \in \mathbb{K}$ to values v in some domain $\mathbb{V}$, i.e., $\sigma : \mathbb{K} \to \mathbb{V}$. We write Σ for the set of all possible stores. We assume $\mathbb{V}$ to include $\bot$, which denotes an undefined value. Hereafter, we assume that any store contains a distinguished initial state σ_0.

We characterise stores in terms of their operations (i.e., queries). A store is defined in terms of a set $\mathcal{O} = \mathcal{W} \cup \mathcal{R}$ of operations, where elements in $\mathcal{W}$ denote write operations and those in $\mathcal{R}$ stand for read operations. We require $\mathcal{W} \cap \mathcal{R} = \emptyset$. We assume any operation to be equipped with an interpretation function $\mathbb{I}$. In particular:

1. $\alpha \in \mathcal{W}$ is interpreted as a function from states to states, i.e., $\mathbb{I}(\alpha) : (\mathbb{K} \to \mathbb{V}) \to \mathbb{K} \to \mathbb{V}$.
2. $\alpha \in \mathcal{R}$ is interpreted as a function from states to boolean values, i.e., $\mathbb{I}(\alpha) : (\mathbb{K} \to \mathbb{V}) \to \{\mathbf{true}, \mathbf{false}\}$. We model in this way the fact that an action actually reads the value that is stored for that key.

Example 1. A memory containing values in $\mathbb{V}$ can be modelled as follows, assuming that σ_0 associates $\bot$ to all names.

1. $\mathcal{W} = \{\mathbf{write}(o, \mathrm{v}) \mid o \in \mathbb{K} \wedge \mathrm{v} \in \mathbb{V}\}$
2. $\mathcal{R} = \{\mathbf{read}(o, \mathrm{v}) \mid o \in \mathbb{K} \wedge \mathrm{v} \in \mathbb{V}\}$
3. $\mathbb{I}(\mathbf{write}(o, \mathrm{v}))\sigma = \sigma[o \mapsto \mathrm{v}]$ (with $\_[\_]$ is the usual update operator)
4. $\mathbb{I}(\mathbf{read}(o, \mathrm{v}))\sigma = (\mathrm{v} == \sigma(o))$.

Example 2. A store σ handling data of type set can be defined as follows, assuming that σ_0 is defined such that $\sigma_0(o) = \emptyset$ for all o.

1. $\mathcal{W} = \{\mathbf{add}(o, \mathrm{v}) \mid o \in \mathbb{K} \wedge \mathrm{v} \in \mathbb{V}\}$
2. $\mathcal{R} = \{\mathbf{read}(o, \mathbb{V}) \mid o \in \mathbb{K} \wedge \mathbb{V} \in 2^{\mathbb{V}}\}$
3. $\mathbb{I}(\mathbf{add}(o, \mathrm{v}))\sigma = \sigma[o \mapsto \sigma(o) \cup \{\mathrm{v}\}]$
4. $\mathbb{I}(\mathbf{read}(o, \mathbb{V}))\sigma = (\mathbb{V} == \sigma(o))$.

Example 3. An alternative characterization for sets can obtained by changing the interpretation of action **read** as follows: $\mathbb{I}(\mathbf{read}(o, \mathbb{V}))\sigma = \mathbb{V} \subseteq \sigma(o)$.

Definition 1. *Let $\alpha \in \mathcal{W}$ and $o \in \mathbb{K}$. We say α* modifies *o iff there exists σ such that $\sigma(o) \neq (\mathbb{I}(\alpha)\sigma)(o)$, i.e., when α may modify the value associated with o. Similarly, $\alpha \in \mathcal{R}$* reads *$o \in \mathbb{K}$ iff $\mathbb{I}(\alpha)\sigma \neq \mathbb{I}(\alpha)(\sigma[o \mapsto v])$ for some $v \in \mathbb{V}$, i.e., when α actually depends on the value of o. We let $objects(\alpha)$ to be the objects read by or written by the action α, i.e.,*

$$objects(\alpha) = \begin{cases} \{o \mid \alpha \textit{ modifies } o\} & \textit{if } \alpha \in \mathcal{W} \\ \{o \mid \alpha \textit{ reads } o\} & \textit{if } \alpha \in \mathcal{R} \end{cases}$$

Example 4. Consider $\mathcal{W}$, $\mathcal{R}$ and $\mathbb{I}$ defined in Example 1. Then, $\mathbf{write}(o, \mathrm{v})$ modifies o' iff $o = o'$. Similarly, $\mathbf{read}(o, \mathrm{v})$ reads o' iff $o = o'$. Also, $objects(\mathbf{read}(o, \mathrm{v})) = objects(\mathbf{write}(o, \mathrm{v})) = \{o\}$.

Definition 2 (Valid read). *Given a read action α and a state σ, α is a valid read of σ, written $\sigma \bowtie \alpha$, when $\mathbb{I}(\alpha)\sigma = \mathbf{true}$. We write $\sigma \bowtie$ to denote the set of all valid reads of σ, i.e., $\sigma \bowtie = \{\alpha \mid \sigma \bowtie \alpha\}$.*

The following definition gives a basic criterion of correctness for weak specifications, namely a read action over a weak store may not return the most recently written value. Formally, the specification of a weak store admits any read action to be valid also after store modifications. In other words, the specification of a weak store does not require modifications to have immediate effects.

Definition 3 (Weak consistent specification). *An interpretation $\mathbb{I}$ defines a weak consistent store iff $\forall \alpha \in \mathcal{R}, \beta \in \mathcal{W}, \sigma \in \Sigma : \sigma \bowtie \alpha \implies \mathbb{I}(\beta)\sigma \bowtie \alpha$.*

Example 5. The specification in Example 3 is a weak consistent specification. In fact, we take $\alpha = \mathbf{read}(o, \mathbb{V})$ and $\beta = \mathbf{add}(o', \mathrm{v}')$. If $\sigma \bowtie \alpha$, then $\mathbb{V} \subseteq \sigma(o)$. In addition, $\sigma' = \mathbb{I}(\beta)\sigma = \sigma[o' \mapsto \sigma(o') \cup \{\mathrm{v}'\}]$. It is straightforward to check that $\mathbb{V} \subseteq \sigma(o) \implies \mathbb{V} \subseteq \sigma'(o)$. Hence, $\sigma' \bowtie \alpha$. Differently, the specifications in Examples 1 and 2 are not weak consistent specifications. In fact, the interpretations for the read actions in both specifications require the store to return exactly the last written value in the store. Intuitively, this means that each update needs to be immediate and not deferred.

The following two definitions are instrumental to the formalisation of strong consistent specifications.

Definition 4 (Last written value). *Given a state σ, a write action $\alpha \in \mathcal{W}$ writes o with v, written $\alpha \downarrow_\sigma^{o \mapsto \mathrm{v}}$, whenever $\sigma(o) \neq \mathrm{v}$ and $(\mathbb{I}(\alpha)\sigma)(o) = \mathrm{v}$.*

Definition 5 (Read a particular value). *Given a state σ, a read action $\alpha \in \mathcal{R}$ reads the value v for o in σ, written $\alpha \uparrow_\sigma^{o \mapsto \mathrm{v}}$, whenever $o \in objects(\alpha)$ implies*

$$(\sigma[o \mapsto \mathrm{v}] \bowtie) \neq (\sigma \bowtie) \implies \alpha \in (\sigma[o \mapsto \mathrm{v}] \bowtie) \backslash (\sigma \bowtie)$$

The above definition requires that whenever the update $[o \mapsto \mathrm{v}]$ of σ alters the set of valid reads, then α is valid only with the modification $[o \mapsto \mathrm{v}]$, i.e., α is enabled by the modification $[o \mapsto \mathrm{v}]$.

Definition 6 (Strong consistent specification). *An interpretation $\mathbb{I}$ defines a strong consistent store whenever any read α of an object o corresponds to the last written value of o, namely iff*

$$\forall \alpha \in \mathcal{R}, \beta \in \mathcal{W} : (\beta \downarrow_\sigma^{o \mapsto \mathrm{v}} \quad \wedge \quad \mathbb{I}(\beta)\sigma \bowtie \alpha) \implies \alpha \uparrow_\sigma^{o \mapsto \mathrm{v}}$$

Example 6. It is easy to check that the specification in Example 1 is strong consistent. For any pair of actions $\alpha = \mathbf{read}(o, \mathrm{v})$ and $\beta = \mathbf{write}(o', \ \mathrm{v}')$, if $\beta \downarrow_\sigma^{o' \mapsto \mathrm{v}'}$ then $\mathbb{I}(\beta)\sigma = \sigma[o' \mapsto \mathrm{v}']$. Then, $\sigma[o' \mapsto \mathrm{v}'] \bowtie \alpha$ implies either (1) $o \neq o'$ or (2) $o = o'$ and

$\alpha \in (\sigma[o' \mapsto \mathrm{v}'] \bowtie) \backslash (\sigma \bowtie)$. In both cases, $\alpha \uparrow_{\sigma}^{o' \mapsto \mathrm{v}'}$. Analogously, it can be shown that the store in Example 2 is also a strong consistent specification. Differently, the store in Example 3 is not a strong consistent specification. Consider $\alpha = \mathbf{read}(o, \emptyset)$, $\beta = \mathbf{add}(o, \mathrm{v})$ and $\sigma = \sigma_0[o \mapsto \emptyset]$. It is straightforward to check that $\beta \downarrow_{\sigma}^{o \mapsto \{\mathrm{v}\}}$ and $\mathbb{I}(\beta)\sigma \bowtie \alpha$. However, it is not the case that $\alpha \uparrow_{\sigma}^{o \mapsto \{\mathrm{v}\}}$ because $\alpha \in \sigma \bowtie$.

We remark that Definitions 3 and 6 give an abstract characterization of the consistency provided/expected from a store. Note that such notions are incomparable. Firstly, a strong consistent specification is not a weak consistent specification because the first requires updates to be immediate while the second may defer them. Analogously, a weak consistent specification is not a strong consistent specification.

2.2 Stores as Labelled Transition Systems

In what follows we will find useful to rely on an operational characterisation of stores in terms of labelled transitions systems. An operational implementation of a specification $\langle \mathcal{O}, \mathbb{I} \rangle$ is an LTS with a set of labels $\mathcal{L} = \mathcal{O} \cup \mathcal{S}$: we consider an implementation to exhibit also some internal or silent actions in $\mathcal{S}$. We use $\tau, \tau', \ldots$ to range over $\mathcal{S}$.

Given a sequence of actions $t \in \mathcal{L}^*$ and a set of labels $\mathcal{A} \subseteq \mathcal{L}$, we write $t \downarrow_{\mathcal{A}}$ to denote the projection that removes from t all labels not in $\mathcal{A}$. Moreover, for $t = \alpha_0 \ldots \alpha_j \ldots$ we will use the standard subindex notation, i.e., $t[j] = \alpha_j$, and $t[i..j] = \alpha_i \ldots \alpha_j$. Given a finite sequence of labels $\alpha_0 \ldots \alpha_j$ s.t. $\alpha_i \in \mathcal{W}$ for all $0 \leq i \leq j$, its interpretation is the function accounting for the composition of all α_i, i.e., $\mathbb{I}(\alpha_0 \ldots \alpha_j) = \alpha_j \circ \ldots \circ \alpha_0$.

Definition 7 (Trace). *A sequence $s \in \mathcal{L}^*$ is a trace from σ_0 satisfying $\langle \mathcal{O}, \mathbb{I} \rangle$ when $s[i] \in \mathcal{R}$ implies $\mathbb{I}(s[0..i-1] \downarrow_{\mathcal{W}})\sigma_0 \bowtie s[i]$, i.e., any read is valid with respect to the store obtained after applying (in order) all preceding operations in $\mathcal{W}$. We write $tr(\sigma_0)$ for the set of all traces from σ_0.*

The definition above provides an operational characterisation for the basic consistency criterion given in Definition 2. In order to refine the above notion to deal with strong consistent stores, we need a finer notion that captures the dependencies between read and write operations.

Definition 8 (Read-write dependency). *The read-write dependency relation, written $\leftrightarrow \subseteq \mathcal{R} \times \mathcal{W}$, is defined as follows*

$$\alpha \leftrightarrow \beta \textit{ iff } objects(\alpha) \cap objects(\beta) \neq \emptyset.$$

Example 7. For the stores introduced in Examples 1–3, we have $\alpha \leftrightarrow \beta$ iff $\alpha = \mathbf{read}(o, \mathrm{v})$ and $\beta = \mathbf{write}(o, \mathrm{v}')$.

Definition 9 (Read follows a write). *Given a trace t, a read action $\alpha = t[j] \in \mathcal{R}$ and a write action $\beta = t[i] \in \mathcal{W}$ s.t. $\alpha \leftrightarrow \beta$, we say α follows β, written $\beta \rightsquigarrow \alpha \in t$, iff $i < j$ and $\forall i < k < j . t[k] \in \mathcal{W} \implies \alpha \not\leftrightarrow t[k]$.*

Now, we are ready to formalise the definition of strong consistency given in [22], that is: '*Every read on a data item x returns a value corresponding to the result of the most recent write on x*'.

Definition 10 (Strong consistent trace and store). *A trace t is strong consistent if for all $\alpha \rightsquigarrow \beta \in t$, $\beta \downarrow_{\sigma}^{o \mapsto v}$ implies $\alpha \uparrow_{\sigma}^{o \mapsto v}$. A store is strong consistent if every trace is strong consistent.*

Lemma 1 establishes a correspondence between Definition 6 and the operational definition of strong consistent trace/store in Definition 10.

Lemma 1. *Let $\langle \mathcal{O}, \mathbb{I} \rangle$ be a strong consistent specification (by Definition 6). If t is a trace satisfying $\langle \mathcal{O}, \mathbb{I} \rangle$, then t is a strong consistent trace.*

Proof. Assume $\beta \rightsquigarrow \alpha \in t$. Then $\exists i < j$ s.t. $t[j] = \alpha \in \mathcal{R}$ and $t[i] = \beta \in \mathcal{W}$. Since t satisfies $\langle \mathcal{O}, \mathbb{I} \rangle$, $\mathbb{I}(t[0..j-1] \downarrow_{\mathcal{W}})\sigma_0 \bowtie \alpha$. Moreover, $\beta \rightsquigarrow \alpha \in t$ implies $\forall i < k < j. objects(j) \cap objects(k) = \emptyset$ and hence $\mathbb{I}(t[0..i] \downarrow_{\mathcal{W}})\sigma_0 \bowtie \alpha$. Consequently, $\mathbb{I}(\beta)(\sigma') \bowtie \alpha)$ (1). Additionally, by Definition 6, it holds that

$$(\beta \downarrow_{\sigma'}^{o \mapsto \mathrm{v}} \quad \wedge \quad \mathbb{I}(\beta)(\sigma') \bowtie \alpha) \implies \alpha \uparrow_{\sigma'}^{o \mapsto \mathrm{v}} \qquad (2)$$

Finally, from (1) and (2) we conclude that $\beta \downarrow_{\sigma'}^{o \mapsto \mathrm{v}} \implies \alpha \uparrow_{\sigma'}^{o \mapsto \mathrm{v}}$. □

Now we take advantage of the operational characterisation of stores to state our notion of eventual consistency.

Consider a scenario in which information is replicated, hence an older value may be read for a key due to the temporary lack of synchronisation of some replicas. A widely (e.g., [2, 23]) used formulation of eventual consistency is the one in [23]: '*if no new updates are made (...), eventually all accesses will return the last updated value*'. We consider here a slightly different notion (on the lines of [21]) which does not require absence of outputs, namely: '*given a write operation on a store, eventually all accesses will return that or a more recent value*'. We prefer this notion as it allows a more natural testing of composition of an application with a store (e.g., not imposing that the application stops writing on the store at a certain point in time allows us to test non-terminating applications). In other words, we do not require that the store will reach a stability, but we require a progress in the synchronisation of the replicas. Notably, this property entails that if no writes are done then all replicas will eventually be synchronized.

Definition 11 (Eventual consistency). *A store σ is eventually consistent if $\forall t \in tr(\sigma_0)$ the following holds: Let $t[i] \in \mathcal{W}$ then $\exists k \geq i. \forall j > k. t[j] \leftrightarrow t[i]$ s.t.:*

1. *$t[i] \downarrow_{\sigma}^{o \mapsto v}$ and $t[j] \uparrow_{\sigma}^{o \mapsto v}$, i.e., $t[j]$ reads the value written by $t[i]$, or*
2. *$\exists i < h < j$ s.t. $t[h] \downarrow_{\sigma}^{o \mapsto v}$ and $t[j] \uparrow_{\sigma}^{o \mapsto v}$, i.e., $t[j]$ reads a newer value.*

The above definition of eventual consistency assumes a total order of write actions. For the sake of the simplicity, we avoid the definitions of even weaker notions as the ones in [6], which could also be recast in this framework.

3 Two Weak and One Strong Stores

In this section we consider three examples of stores, two weak stores with replicas and distributed control as, e.g., Amazon S3 [11] (although our model has a quite simplified API), and one strong store. In the weak stores we assume that, after a write, the 'synchronisation' of the replicas happens asynchronously and without locks.

In a replicated store, the ensemble of replicas can store different object versions for the same key. Given a key o and a store σ, $\sigma(o)$ is the set of *versions* of o in σ. We model each version as a pair (v, n) where v is the object content (e.g., blob, record, etc) and n is a version vector. The set of version vectors $\mathbb{W}$ is a poset with lower bound $\bot$, and version vectors range over $n, n', n_1, \ldots$. We write $n > n'$ when n is a successive version of n', and we write $n <> n'$ when neither $n > n'$ nor $n' > n$ (i.e., n and n' are conflicting versions). We say that n is fresh in $\sigma(o)$ if $\forall (v, n') \in \sigma(o),\ n \neq n'$. We write $n + 1$ to denote a fresh version vector such that $n + 1 > n$. Similarly, given a set of version vectors $\{n_1, .., n_m\}$ we use the notation $\{n_1, .., n_m\} + 1$ to denote a fresh version vector n' such that $n' > n_i$ for all $i \in \{1, .., n_m\}$. Note that there is more than one vector $n + 1$ for a given n, and that if two replicas create two (fresh) next version numbers $n_1 = n + 1$ and $n_2 = n + 1$ we have $n_1 <> n_2$.

3.1 Weak Store

We first model a weak store σW in which replicas rely on a synchronisation mechanism that enables them to choose, when handling a write action, a version vector that is certainly greater than all version vectors associated to the same object. Such a mechanism could be implemented, for instance, using timestamps as version vector, assuming that replicas have a global notion of time which is precise enough to always distinguish between the timestamp of a couple of events (i.e., actions are done at different times).

Definition 12 (Weak store). *A (replicated, distributedly controlled) weak store is defined as follows:*

1. $\mathcal{W} = \{\mathbf{write}(o, v) \mid o \in \mathbb{K} \wedge v \in \mathbb{V}\}$
2. $\mathcal{R} = \{\mathbf{read}(o, (v, n)) \mid o \in \mathbb{K} \wedge v \in \mathbb{V} \wedge n \in \mathbb{W}\}$
3. $\mathbb{I}(\mathbf{write}(o, v))\sigma W = \sigma W[o \mapsto \sigma W(o) \cup \{(v, \{(v', m) \in \sigma W(o)\} + 1\}]$
4. $\mathbb{I}(\mathbf{read}(o, (v, n)))\sigma W = (v, n) \in \sigma W(o)$

The semantics of σW is given by the LTS in Fig. 1 using the labels $\mathcal{W} \cup \mathcal{R} \cup \{\tau\}$.

A *read* operation non-deteministically returns one version of an object, not necessarily the most recent update with the greater version vector (rule [WREAD]). A *write* is modelled by rule [WWRITE] that adds a new version associated with a newly created version vector, that is greater than all version vectors of the existing versions of the same object. A store *propagation* [WPRO] models the asynchronous communication among the replicas to achieve a consistent view of the data.

$$\frac{(\mathrm{v}, n) \in \sigma W(o)}{\sigma W \xrightarrow{\mathbf{read}(o,(\mathrm{v},n))} \sigma W} \text{ [WREAD]}$$

$$\frac{\sigma W' = \sigma W[o \mapsto \sigma W(o) \cup \{(\mathrm{v}, n)\}] \quad n = \{m \mid (\mathrm{v}', m) \in \sigma W(o)\} + 1}{\sigma W \xrightarrow{\mathbf{write}(o,\mathrm{v})} \sigma W'} \text{ [WWRITE]}$$

$$\frac{(\mathrm{v}_i, n_i) \in \sigma W(o) \quad i \in \{1,2\} \quad n_1 \geq n_2 \quad \sigma W' = \sigma W[o \mapsto \sigma W(o) \setminus \{(\mathrm{v}_1, n_1)\}]}{\sigma W \xrightarrow{\tau} \sigma W'} \text{ [WPRO]}$$

Fig. 1. Labelled transition relation for the weak store σW

Weak store and strong consistency Proposition 1 shows that σW is not strong consistent as it does not satisfy Definition 10.

Proposition 1. *The weak store σW is not strong consistent.*

Proof. Consider the initial store $\sigma W_0(o) = \{(\bot, n_0)\}$. Then, $\mathit{traces}(\sigma W_0)$ includes

$$t = \mathbf{write}(o, 10), \mathbf{read}(o, (\bot, n_0))$$

obtained by applying, in sequence, rule [WWRITE] and [WREAD] to σW_0, i.e.,

$$\sigma W_0 \xrightarrow{\mathbf{write}(o,10)} \sigma W_1 \xrightarrow{\mathbf{read}(o,(\bot,n_0))} \sigma W_1$$

where $\sigma W_1(o) = \{(\bot, n_0), (10, n_1)\}$ for some $n_1 = n_0 + 1$. Then, $t[0] \rightsquigarrow t[1]$, with $t[0] = \mathbf{write}(o, 10) \in \mathcal{W}$, $t[1] = \mathbf{read}(o, (\bot, n_0)) \in \mathcal{R}$, and $t[0] \downarrow^{o \mapsto \{(\bot,n_0),(10,n_1)\}}_{\sigma W_1}$. However, $t[0] \uparrow^{o \mapsto \{(\bot,n_0),(10,n_1)\}}_{\sigma W_1}$ does not hold since $\sigma W_1 \bowtie \neq \sigma W_0 \bowtie$ but $t[1] \notin ((\sigma W_1 \bowtie) \setminus \sigma W_0 \bowtie)$. Namely, the read action in $t[1]$ does not allow to read *only* the most recent value $(10, n_1)$ when it takes place too early to allow the other replicas to synchronise and restore consistency. □

Weak store and eventual consistency We show below that σW ensures eventual consistency (Definition 11), after a few auxiliary lemmas. First we observe that all version vectors for the same object in a state σW are distinguished.

Lemma 2. $\forall o \in \mathbb{K}.\ (\mathrm{v}_1, n_1), (\mathrm{v}_2, n_2) \in \sigma W(o) \implies n_1 \neq n_2$.

In fact, σ_0 only has one value for each object, and the only transition rule that increments the number of version numbers for the same object is [WWRITE] that choses n fresh.

Lemma 3. $\forall o \in \mathbb{K}.\ (\mathrm{v}_1, n_1), (\mathrm{v}_2, n_2) \in \sigma W(o) \implies \neg(n_1 <> n_2)$.

In fact, the only rule that introduces new version vectors is [WWRITE] which introduces n that is comparable with the existing vectors for the same object.

Next we observe that if there are no write actions on an object o then the number of versions in subsequent states is monotonically non-increasing (Lemma 4) and will eventually be just one (Lemma 5). Below we use the following notation: given a set $O \in \mathbb{K}$ we write $\mathcal{W}_O$ for $\{\alpha \mid \alpha \in \mathcal{W} \wedge \mathit{objects}(\alpha) \cap O \neq \emptyset\}$.

Lemma 4. *Consider the trace* $t[i..\infty[\ \downarrow_{(\mathcal{L}\setminus\mathcal{W}_{\{o\}})}$ *with* $\sigma W_j \xrightarrow{t[j+1]} \sigma W_{j+1}$ *for all* $j \geq i$*. Then,* $\forall j \geq i.\ |\sigma W_i(o)| \geq |\sigma W_{i+1}(o)|$.

Proof. Assume $|\sigma W_j(o)| = m_o$ for some $j \geq i, m_o \geq 1$. Since by hypothesis $t_{j+1} \notin \mathcal{W}_{\{o\}}$, then either

- $t_{j+1} \in \mathcal{W}$ but writes $o' \neq o$, hence by [WWRITE] $|\sigma W_j(o)| = |\sigma W_{j+1}(o)| = m_o$
- $t_{j+1} \in \mathcal{R}$ hence, by [WREAD] $|\sigma W_j(o)| = |\sigma W_{j+1}(o)| = m_o$
- $t_{j+1} \in \mathcal{S}$ hence, by [WPRO] either $|\sigma W_j(o)| = |\sigma W_{j+1}(o)| = m_o$ or, if one version of o is removed, $|\sigma W_{j+1}(o)| - 1 = m_o - 1$.

Lemma 5. *Consider the trace* $t[i..\infty[\ \downarrow_{(\mathcal{L}\setminus\mathcal{W}_{\{o\}})}$ *with* $\sigma W_j \xrightarrow{t[j+1]} \sigma W_{j+1}$ *for all* $j \geq i$*. Then,* $\exists j \geq i.\ |\sigma W_j(o)| = 1$*, assuming fairness in the LTS in Fig. 1.*

Proof. Consider any $j \geq i$, we have two cases: either $|\sigma W_j(o)| = 1$, or $|\sigma W_j(o)| > 1$. In the first case, by Lemma 4, $\forall l \geq j.\ |\sigma W_l(o)| = 1$. In the second case, $\sigma W_j(o) \ni (\mathrm{v}_1, n_1), (\mathrm{v}_2, n_2)$. By Lemma 2 $n_1 \neq n_2$ and by Lemma 3 it is never the case that $n_1 <> n_2$, hence either $n_1 < n_2$ or $n_2 < n_1$. If $n_1 < n_2$ (resp. $n_2 < n_1$) then transition [WPRO] is enabled and hence, by fairness, it will eventually execute so to decrease the number of versions (which, again will not, in the meanwhile, increase by Lemma 4). □

Proposition 2. *The weak store* σW *is eventually consistent (by Definition 11) assuming strong fairness* [3] *in the LTS in Fig. 1.*

Proof. Let t be an infinite trace and let $t[i] \in \mathcal{W}$. By Definition 12, $t[i]$ has the form $\mathbf{write}(o, \mathrm{v})$ for some o, v. By Lemma 5, if we apply the actions in $t[i..\infty[\ \downarrow_{(\mathcal{L}\setminus\mathcal{W}_{\{o\}})}$ to a state σW we obtain a trace t' and there is a k such that $|\sigma W_k(o)| = 1$. Let k' be the position occupied in t by the action $t'[k]$. If there is no write action $\alpha \in t[k'..\infty[$ such that $t[i] \leftrightarrow \alpha$ then the proposition is trivially true. If such α exists we let n be the version vector introduced by α and assume, by contradiction, that (1) $\mathrm{w} \neq \mathrm{v}$ (case 2 of Definition 11), and (2) that there is no write action $\beta = t[h] \in t[i, k']$ such that $\beta \downarrow^{o \mapsto \mathrm{w}}_{\sigma W_h}$. Then either $\mathrm{v} = \perp$ or there is a write action $t[j] \downarrow^{o \mapsto \mathrm{w}}_{\sigma W_j}$ such that $j < i$. In both cases, by rulename [WWRITE] the version vector associated to w is smaller than n. By [WPRO] the smaller version numbers are eliminated, hence the read in k' must return only one value, which is the one written by $t[i]$, namely v. □

3.2 An Asynchronous Weak Store

We define the *weak asynchronous store* σA as a relation from keys to sets of versioned objects. In this case replicas cannot rely on an absolute ordering of versions wrt the global timeline. This is reflected in the rule for writing new versions, [AWRITE]: the version vector of the newly introduced value is be greater of the version vector of one replica but possibly incomparable with the version vectors in the other replicas.

[3] We rely on the following notion of fairness: "If any of the actions $a_1, ..., a_k$ is ever enabled infinitely often, then a step satisfying one of those actions must eventually occur" (For a formal definition see[12].)

$$\frac{(\mathrm{v}', n') \in \sigma A(o) \quad \sigma A' = \sigma A[o \mapsto \sigma A(o) \cup \{(\mathrm{v}, n)\}] \quad n = n' + 1}{\sigma A \xrightarrow{\mathbf{write}(o,\mathrm{v})} \sigma A'} \text{ [AWRITE]}$$

$$\frac{\begin{array}{c}(\mathrm{v}_i, n_i) \in \sigma A(o) \quad i \in \{1,2\} \quad n_1 <> n_2 \\ \sigma A' = \sigma A[o \mapsto (\sigma A(o) \setminus \{(\mathrm{v}_i, n_i) \mid i \in \{1,2\}\} \cup \{(\mathrm{v}_1, n_3)\})] \quad n_3 = \{n_1, n_2\} + 1\end{array}}{\sigma A \xrightarrow{\tau} \sigma A'} \text{ [ASOL]}$$

Fig. 2. Labelled transition relation for the weak asynchronous store σA

The store σA is defined using the interpretation and labels in Definition 12 but changing rule [WWRITE] in Fig. 1 with rule [AWRITE] and adding rule [ASOL], as shown in Fig. 2. Rule [ASOL] is necessary to resolve conflicts arising when two version vectors that are incomparable during replicas synchronisation:

With a similar argument as in Proposition 1 one can show that σA is not strong consistent. Furthermore, σA is not eventually consistent, according to Definition 11.

Proposition 3. *The asynchronous weak store σA is not eventually consistent.*

Consider an initial store σA_0 such that $\sigma A_0(o) = \{(\bot, n_0)\}$, then $\mathit{traces}(\sigma A_0)$ includes the following family of traces

$$t = \mathbf{write}(o, 10), \mathbf{write}(o, 5), \tau, t', \mathbf{read}(o, (10, n_1))$$

with $t' \downarrow_{\mathcal{W}_{\{o\}}} = \emptyset$. Any trace t can be obtained as follows:

1. By [AWRITE], $\sigma A_0 \xrightarrow{\mathbf{write}(o,10)} \sigma A_1$ where $\sigma A_1(o) = \{(\bot, n_0), (10, n_1)\}$.
2. By [AWRITE], $\sigma A_1 \xrightarrow{\mathbf{write}(o,5)} \sigma A_2$ where $\sigma A_2(o) = \{(5, n_2), (10, n_1)\}$ and $n_1 <> n_2$.
3. By [ASOL], $\sigma A_2 \xrightarrow{\tau} \sigma A_3$ where $\sigma A_3(o) = \{(10, n_3)\}$ with $n_3 > n_1, n_2$.

Any read of o after σA_3 returns $(10, n_3)$. This violates Definition 11 as after no k a read will return the value written by $\mathbf{write}(o, 5)$ or a more recently written one.

However, Lemmas 4 and 5 hold also for the σA. In fact, despite not ensuring eventual consistency, the weak asynchronous store still ensures the convergence of the replicas to a consistent value if no new updates are made.

3.3 Strong Store

We now model a strong store σS (with version vectors) that satisfies *strong consistency* (Definition 10); having one version for each key σS always return the last version.

Definition 13 (Strong store). *The family of strong stores σS, containing values that are pairs (v, n) where v is a value and n a version vector is defined as follows:*

1. $\mathcal{W} = \{\mathbf{write}(o, \mathrm{v}) \mid o \in \mathbb{K} \wedge \mathrm{v} \in \mathbb{V}\}$
2. $\mathcal{R} = \{\mathbf{read}(o, (\mathrm{v}, n)) \mid o \in \mathbb{K} \wedge \mathrm{v} \in \mathbb{V} \wedge n \in \mathbb{W}\}$

$$\frac{(\mathrm{v}, n) = \sigma S(o)}{\sigma S \xrightarrow{\mathbf{read}(o,(\mathrm{v},n))} \sigma S} \text{ [SREAD]}$$

$$\frac{(\mathrm{v}', n') \in \sigma S(o) \quad \sigma S' = \sigma S[o \mapsto (\mathrm{v}, n)] \quad n = n' + 1}{\sigma S \xrightarrow{\mathbf{write}(o,\mathrm{v})} \sigma S'} \text{ [SWRITE]}$$

Fig. 3. Labelled transition relation for strong stores

3. $\mathbb{I}(\mathbf{write}(o, \mathrm{v}))\sigma S = \sigma S[o \mapsto (\mathrm{v}, n+1)]$ *with* $\sigma S(o) = (\mathrm{v}', n)$
4. $\mathbb{I}(\mathbf{read}(o, \mathrm{v}))\sigma S = \sigma S(o)$

The semantics of σS is given by the LTS in Fig. 3 using the labels $\mathcal{W} \cup \mathcal{R}$.

Proposition 4. *The strong store σS (from Definition 13) is strong consistent.*

Proof. Let $t \in \mathit{traces}(\sigma S_0)$ with $\sigma S_0(o) = (\bot, n_0)$ for all $o \in \mathbb{K}$. Let $t[i] \in \mathcal{W}$ and $t[j] \in \mathcal{R}$ with $t[i] \rightsquigarrow t[j]$ (hence $t[i] \leftrightarrow t[j]$) with $i < j$. By $t[i] \leftrightarrow t[j]$ and by the form of reads and writes (which are defined on single objects) we have that $\mathit{objects}(t[i]) = \mathit{objects}(t[j]) = \{o\}$ for some $o \in \mathbb{K}$. The proof is by induction on the distance $j - i$ between the write and read actions.

Base case ($i = 1$). The only possible transition from σS_{i-1} to σS_i is by [SWRITE]

$$\sigma S_{i-1} \xrightarrow{\mathbf{write}(o,\mathrm{v})} \sigma S_i = \sigma S_{i-1}[o \mapsto (\mathrm{v}, n)]$$

Hence the only possible read action of o in state σS_i is, by rule [SREAD]:

$$\frac{\sigma S_i(o) = (\mathrm{v}, n)}{\sigma S_i \xrightarrow{\mathbf{read}(o,(\mathrm{v},n))} \sigma S_i}$$

hence $t[i] \downarrow_{\sigma S_i}^{o \mapsto (\mathrm{v},n)}$ and $t[i] \uparrow_{\sigma S_i}^{o \mapsto (\mathrm{v},n)}$.

Inductive case ($i > 1$). We proceed by case analysis on the action $t[j-1]$: (1) if $t[j-1]$ is a read action that it will not affect the store (by [SREAD]) hence $\sigma S_{i-1} = \sigma S_i$. By induction we have that reading o in position $j - 1$ returns (v, n) hence it will return (v, n) also in σS_i. (2)if $t[j-1]$ is a write, then it writes $o' \neq o$ because $t[i] \rightsquigarrow t[j]$. This case is similar to (1) observing that writing an object o' does not affect the values read in σS_j for o. □

4 The Calculus

We now focus on the model of programs that operate over stores. Our programs are distributed applications that are able to perform read and write operations over a shared store and communicate through message passing *à la* value-passing CCS [14].

4.1 Syntax

As usual, we let communication channel names be ranged over by $x, x', y, \ldots$, and variables by $v, v', \ldots$. As in previous section values in $\mathbb{V}$ are ranged over by $\mathrm{v}, \mathrm{v}', \ldots$ and keys in $\mathbb{K}$ by $o, o', \ldots$. We also assume that variables, values and objects can be combined into larger expressions whose syntax is left unspecified. We use $e, e', \ldots$ to range over expressions. The syntax of programs is just an extension of value-passing CCS with prefixes accounting for the ability of programs to interact with the store. We do not restrict such prefixes to be just the actions $\alpha \in \mathcal{W} \cup \mathcal{R}$, because these are ground terms and we would like to be able to write programs such as $x(v).\mathbf{read}(v, w).P$ that read from the store the value w associated with the key v that has been previously received over the channel x. Similarly, we would like to consider programs such as $x(v).y(w).\mathbf{write}(o, v + w)$ that updates the value associated to the key o with the result of evaluating an expression. For this reason, *program actions over stores* are of the following form $\mathbf{operation\_name}(e_1, \ldots, e_n)$. We use $\mathcal{A}$ for the set of program actions over stores, and let ρ, ρ' to range over $\mathcal{A}$.

$$P ::= \overline{x}e \mid x(v).P \mid \rho.P \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P \mid \nu x.P \mid P||P \mid\ !P \mid 0$$

Process $\overline{x}e$ asynchronously sends along channel x the value obtained by evaluating expression e. Dually, $x(v).P$ receives along channel x a value, used to instantiate variable v in the continuation P. Process $\rho.P$ stands for a process that performs a store action ρ over the store and then continues as P. Processes $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P$, $\nu x.P$, $P||P$, $!P$ and 0 are standard conditional statement, name restriction, parallel composition, replicated process and idle process, respectively. We will write $\mathit{fn}(P)$ (resp. $\mathit{bn}(P)$) to denote the set of free (resp. bound) variables of P (their definition is standard). As usual, we will restrict ourself to consider closed programs.

A program P interacting with a store σ is called a *system* and it is denoted by $[P]\sigma$.

4.2 Operational Semantics

The labelled transition relation for systems uses the following labels:

$$\mu ::= \tau \mid \overline{x}\mathrm{v} \mid x(v) \mid \alpha$$

We remark that our labels are the standard ones for CCS ($\tau, \overline{x}\mathrm{v}, x(v)$) and the ground store operations $\alpha \in \mathcal{W} \cup \mathcal{R}$, i.e., stores perform concrete operations over the store. We write $\mathit{obj}(\mu)$ to denote the set of objects of the label μ, which is defined as usual. We rely on a reduction relation on expressions $\longrightarrow^e$ that evaluates expressions to values (omitted). For any program action $\rho = \mathbf{a}(e_1, \ldots, e_n)$ over the store and any ground substitution θ, we will write $\rho \longrightarrow^e_\theta \alpha$ if $\forall i. e_i\theta \longrightarrow^e g_i$ and $\mathbf{a}(g_1, \ldots, g_n) = \alpha$, i.e., it denotes that ρ can be properly instantiated with θ to obtain a ground action α. For instance, $\mathbf{write}(o, v) \longrightarrow^e_{v \mapsto \mathrm{v}} \mathbf{write}(o, \mathrm{v})$. We also rely on the LTS for stores, also denoted with $\xrightarrow{\alpha}$, as defined in previous sections.

The LTS for systems is defined by the inference rules in Fig. 4. All rules for processes but [STORE] are standard. Rule [STORE] models a program P that performs a

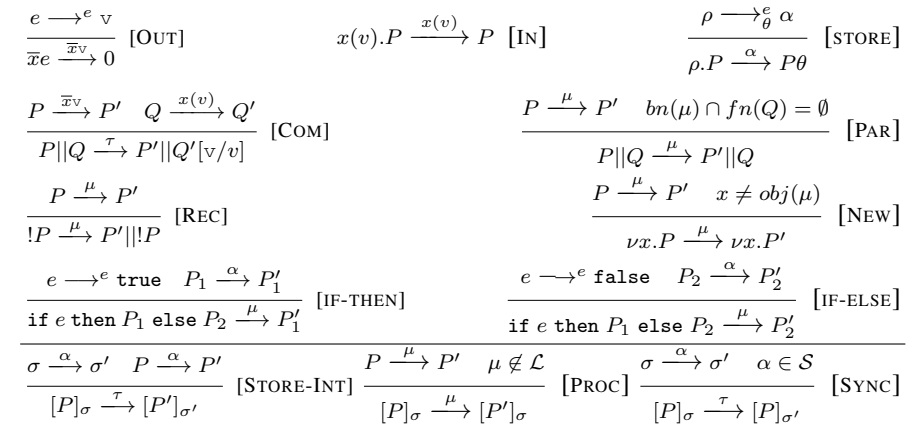

Fig. 4. Labelled Transitions for processes (top) and systems (bottom).

store action. The substitution θ models the values read from the store. Rules for networks model the interactions of the program with the store: [STORE-INT] stands for a program P that writes to or read from the store σ; [PROC] stands for the computational steps of a program that do not involve any interaction with the store, while [SYNC] accounts for the synchronisation steps of the store.

4.3 Reasoning about programs

We rely on some behavioural preorder on systems (e.g., standard weak simulation), denoted by $\precsim$ (and $\sim$ for the associated equivalence). We now characterise the consistency level provided by a store σ relatively to the consistency level of other store σ' by comparing the behaviours of the systems $[P]_\sigma$ and $[P]_{\sigma'}$.

Definition 14. *We say σ is stronger than σ', written $\sigma \prec \sigma'$, whenever $[P]_\sigma \precsim [P]_{\sigma'}$ for all P. We write $\sigma \cong \sigma'$ when $\sigma \prec \sigma'$ and $\sigma' \prec \sigma$.*

Then, the different consistency levels can be thought as the equivalence classes of $\cong$. We write $[\sigma]_\cong$ for the representative of the equivalence class of σ.

Example 8. If we take $\precsim$ as weak simulation $\precsim_{\mathsf{ws}}$, then we can show that the strong consistent store σS (Section 3.3) is stronger than both the weak consistent store σW (Section 3.1) and the asynchronous weak consistent store σA (Section 3.2). This can be done by showing that the following relations are weak simulations

$$S_{\mathcal{S},\mathcal{W}} = \{(\sigma S_i, \sigma W_i) \mid \forall o.\sigma S_i(o) \in \sigma W_i(o)\}$$
$$S_{\mathcal{S},\mathcal{A}} = \{(\sigma S_i, \sigma A_i) \mid \forall o.\sigma S_i(o) \in \sigma A_i(o)\}$$

Analogously, it can be shown that $\sigma W \prec \sigma A$ using the following relation.

$$S_{\mathcal{W},\mathcal{A}} = \{(\sigma W_i, \sigma A_i) \mid \forall o.\sigma W_i(o) \subseteq \sigma A_i(o)\}$$

Additionally, it is easy to check that $\sigma W \not\preceq \sigma S$. It is enough to consider the trace shown in proof of Proposition 1, which cannot be mimicked by σ_S. Similarly, it can be shown that $\sigma A \not\preceq \sigma S$ and $\sigma A \not\preceq \sigma W$.

When considering $\precsim$ as weak simulation $\precsim_{\mathsf{ws}}$, we have $0 \precsim_{\mathsf{ws}} \sigma$ for all σ. Hence, $[P]_0 \precsim_{\mathsf{ws}} [P]_\sigma$ for all P and σ. This mean that 0 (i.e., the store providing no operation) is the strongest consistent store. More generally, all stores weak bisimilar to 0 are the smallest elements in the preorder $\prec$, and hence the ones providing the strongest notion of consistency. However, these stores are non-available services, because they are unable to perform any read or write operation. Hence, in what follows we will focus on available store, i.e., stores in which write and read operations are always enabled.

Definition 15. *σ is available whenever the following two conditions hold:*

- *$\forall \alpha \in \mathcal{W} : \sigma \xrightarrow{\alpha}$, the store allows any write operation at any time.*
- *$\forall \rho \in \mathcal{A}$ s.t. $\rho \longrightarrow^e_\theta \alpha \in \mathcal{R}$, i.e., if the action is a read, then there exists θ' s.t. $\rho \longrightarrow^e_{\theta'} \alpha$ and $\sigma \xrightarrow{\alpha}$, i.e., the store allows any non-ground read at any time.*

Second condition states that a read of some particular values may not be possible in some states, but a read operation is always possible.

Example 9. It is easy to check that the three stores in Section 3 provide availability. From the LTSs, we can conclude that for any pair of object o and value v, the transition $\sigma \xrightarrow{\mathbf{write}(o,\mathrm{v})}$ can be derived. Similarly, for any object o there is always a pair (v, n) such that $\sigma \xrightarrow{\mathbf{read}(o,(\mathrm{v},n))}$ can be performed.

The following result states that the strong store in Section 3.3 is minimal w.r.t. $\prec$ when restricting to available services.

Lemma 6. *Let σS the strong store defined by the transition rules in Fig. 3. Then, $\sigma' \prec \sigma S$ and σ' available imply $\sigma S \prec \sigma'$.*

Proof. By showing that the following relation is a weak simulation

$$S = \{(\sigma S_i, \sigma_i) \mid \sigma_i \precsim_{\mathsf{ws}} \sigma S_i \ \wedge \sigma_i \text{ available}\}$$

We proceed by case analysis on $\sigma S_i \xrightarrow{\alpha} \sigma S_i'$

- $\alpha = \mathbf{read}(o, (\mathrm{v}, n))$: Since σ_i is available, $\exists \alpha' = \mathbf{read}(o, (\mathrm{v}', n'))$, s.t. $\sigma_i \xrightarrow{\alpha'} \sigma_i'$. Since $\sigma_i \precsim_{\mathsf{ws}} \sigma S_i$, it holds that $\sigma S_i \xrightarrow{\alpha'} \sigma S_i''$ with $\sigma' \precsim_{\mathsf{ws}} \sigma S_i''$. By definition of σS_i, it holds that $\forall o$ there exists a unique (v, n) s.t. $\sigma S_i \xrightarrow{\mathbf{read}(o,(\mathrm{v}',n))} \sigma S_i''$. Therefore, $\mathrm{v} = \mathrm{v}$, $n = n'$ and $\sigma S_i' = \sigma S_i$. Hence, $(\sigma S_i', \sigma_i') \in S$.
- $\alpha = \mathbf{write}(o, \mathrm{v})$: Since σ_i is available, $\sigma_i \xrightarrow{\alpha} \sigma_i'$. Since, $\sigma_i \precsim_{\mathsf{ws}} \sigma S_i$, there exists $\sigma S_i''$ s.t. $\sigma S_i \xrightarrow{\alpha} \sigma S_i''$ and $\sigma_i' \precsim_{\mathsf{ws}} \sigma S_i''$. Since σS_i is deterministic, $S_i'' = S_i'$. Therefore, $(\sigma S_i', \sigma_i') \in S$. □

Finally, we can define the consistency level supported by a given program.

Definition 16. *A program P supports the consistency level $[\sigma]_{\cong}$ iff $[P]_{\sigma} \precsim [P]_{\sigma'}$ for all $\sigma' \prec \sigma$.*

The above definition states that a program P supports a particular consistency level $[\sigma]_{\cong}$ when the behaviours of P running against σ are at most the behaviours that can obtained when running P against any store σ' providing stronger consistency property. We remark that the above definition implies to consider just the minimal σ' s.t. $\sigma' \prec \sigma$. In fact, for all other σ'' s.t. $\sigma' \prec \sigma'' \prec \sigma$ it holds that $[P]_{\sigma'} \precsim [P]_{\sigma''}$ for all P and therefore $[P]_{\sigma} \precsim [P]_{\sigma'}$ implies $[P]_{\sigma} \precsim [P]_{\sigma''}$. Thanks to Lemma 6, we need just to compare a particular consistency level σ against the strong store in Section 3.

Example 10. Consider the program $P \triangleq \nu x(P_P \mid P_C)$ where

$$P_P \triangleq \mathbf{write}(o, \mathrm{v}); \overline{x} \qquad P_C \triangleq !x.\mathbf{read}(o, (v, v')); \overline{show}(o, (v, v'))$$

Running P over the strong store σS defined in 3.3, the only possible execution trace is the following (regardless of the initial state of the store)

$$[P]_{\sigma S} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\overline{show}(o,(\mathrm{v},\mathrm{n}))}$$

whereas the weak store defined in 3.2 produces two additional traces (assume $\sigma W_0(o) = (\mathrm{v}_0, \mathrm{n}_0)$)

$$[P]_{\sigma W} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\overline{show}(o,(\mathrm{v},\mathrm{n}))}$$
$$[P]_{\sigma W} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\overline{show}(o,(\mathrm{v}_0,\mathrm{n}_0))}$$

The second trace models the case in which the process is reading a data that has not yet been propagated to all replicas. In Example 11 this problem is solved by adding extra communication in the processes.

Example 11. Consider the following variant of the program P in Example 10: $P' \triangleq \nu x(P_P \mid P'_C)$

$$P_P \triangleq \mathbf{read}(o, (\mathrm{v}', n)).\mathbf{write}(o, \mathrm{v}); \overline{x}n$$
$$P'_C \triangleq \overline{y} \mid !y.x(v).\mathbf{read}(o, (v', v''));\ \mathtt{if}\ v < v''\ \mathtt{then}\ \overline{show}(o, (v', v''))\ \mathtt{else}\ \overline{y}$$

Now, it is easy to check that $[P']_{\sigma W} \precsim_{\mathsf{ws}} [P']_{\sigma S}$ (note that all traces have the shape $\xrightarrow{\tau}{}^{*} \xrightarrow{\overline{show}(o,(\mathrm{v},\mathrm{n}))}$). Thanks to the synchronization between producer and consumer on x and about the last versioning number n, process P has the same behaviour in both weak and strong memory model.

5 Related Work

There is an extensive literature on the properties of hardware weak memory models (e.g., TSO-x86 [20] and POWER [13]), whose properties derive for instance from the reordering of memory operations made by the processor. Within this research thread, [18, 19] focus on the correctness of x86 assembly code via verified compilers, [16] presents a proof system for x86 assembly code based on a rely-guarantee assertion method, [13] proposes an axiomatic model for POWER memory model, equivalent to the operational specification given in [17], and illustrates how it can be used for verification using a SAT constraint solver. A general account of weak memory models is given in [1], together with a mechanism of memory fences (i.e., barriers located in the code) that preserve properties such as sequential consistency, and an automatic generation of tests for processors implementations. Existing work on memory models also addresses higher levels of abstraction. For instance, [9] provides a static typing framework to reason on weak memory models in Java, and in particular on the property *happens-before*. The work in [6] studies a similar notion of store to the one we consider; it defines sufficient rules for the correct implementation of eventually consistent transactions in a server using revision diagrams. [5] proposes data types for ensuring eventually consistency over cloud systems.

With respect to the work on hardware memory models, we consider a higher level of abstraction, focusing on consistency of replicated cloud stores. The main difference with existing work is in the aim of our paper: albeit we provide a characterisation of weak and strong storages, our aim is not to ensure specific properties (e.g., strong consistency) but to check that a general purpose application will execute correctly when composed with a store offering a given level of consistency. Our notion of correctness depends from the specific applications.

In [3] a similar approach is followed: a characterisation of weak stores is given together with a parametric operational semantics for a functional language. The focus is on ensuring properties such as race-freedom on shared memory with buffers. In our scenario, replicas differ from buffers as they can be accessed by any process/thread immediately after any update, and the focus is put on the interplay of asynchronous communication of processes and usage of cloud storages.

6 Conclusion

In this paper we address the formal study of database consistency levels. We start by proposing a general declarative way for specifying stores in terms of the operations they provide. We show that we can characterise some basic properties about stores at this abstract level. We also show that we can relate such specifications with some concrete operational implementations in terms of LTS. We also illustrate how to provide a more fine grained specification of properties, e.g., eventual delivery. For simplicity, we just consider eventual delivery under the assumption of total ordering of events. We leave as future work to extend our approach to consider even weaker models, as the one studied in [6, 24], which uses Revision Diagrams. We also analyse some concrete implementations of stores with different consistency levels, by using idealised operational

models. The study of concrete real implementations such as Amazon key-value store called Dynamo [7], Amazon SimpleDB or Apache CouchDB (which solves conflicts non-deterministically, incremental Map Reduce) is left as future work.

Finally, we propose an approach for analysing the interaction between programs and stores, in particular, to understand the consistency requirements of programs. Firstly, we defined a preorder relation $\prec$ over stores in terms of their behaviour when interacting with applications. The equivalence classes induced by the equivalence relation associated with $\prec$ corresponds exactly to the consistency levels. Then, we classify programs in terms of the consistency levels they may allow. We use this classification to reason about program correctness running over a particular store. Basically, it can be used to show that an application supports weaker consistency levels by showing that it behaviour corresponds to the one observed when interacting with a strong consistent store. In addition, it may be useful for understanding how to "fix" an application that is supposed to run over a weak store but need stronger properties. A practical example of this is given in [11] where architectures are built to interface applications with weak stores to provide properties not originally satisfied by the store. Our framework is aimed at allowing the formal analysis of such proposals by exploiting the compositional reasoning enabled by process calculi standard notions.

Acknowledgments We thank the anonymous reviewers of WS-FM 2013 for their insightful and helpful comments on the previous version of this paper.

References

1. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2):170–205, 2012.
2. P. Bailis and A. Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013.
3. G. Boudol and G. Petri. Relaxed memory models: an operational approach. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 392–403. ACM, 2009.
4. G. Boudol, G. Petri, and B. P. Serpette. Relaxed operational semantics of concurrent programming languages. In B. Luttik and M. A. Reniers, editors, *DCM*, volume 89 of *EPTCS*, pages 19–33, 2012.
5. S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag.
6. S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In H. Seidl, editor, *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 67–86. Springer, 2012.
7. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.
8. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
9. M. Goto, R. Jagadeesan, C. Ptcher, and J. Riely. Types for relaxed memory models. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, pages 25–38, New York, NY, USA, 2012. ACM.

10. R. Jagadeesan, G. Petri, C. Pitcher, and J. Riely. Quarantining weakness - compositional reasoning under relaxed memory models (extended abstract). In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 492–511. Springer, 2013.
11. D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 579–590. ACM, 2010.
12. L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.
13. P. Madhusudan and S. A. Seshia, editors. *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*. Springer, 2012.
14. R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
15. D. Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.
16. T. Ridge. A rely-guarantee proof system for x86-tso. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE'10, pages 55–70, Berlin, Heidelberg, 2010. Springer-Verlag.
17. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. *SIGPLAN Not.*, 47(6):175–186, June 2011.
18. J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In T. Ball and M. Sagiv, editors, *POPL*, pages 43–54. ACM, 2011.
19. J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.
20. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
21. M. Shapiro and B. Kemme. Eventual consistency. In M. T. Özsu and L. Liu, editors, *Encyclopedia of Database Systems (online and print)*. Springer, Oct. 2009.
22. A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education, 2007.
23. W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.
24. K. von Gleissenthall and A. Rybalchenko. An epistemic perspective on consistency of concurrent computations. *CoRR*, abs/1305.2295, 2013.