myjournal manuscript No.
(will be inserted by the editor)

On the behaviour of general-purpose applications on cloud storages

Laura Bocchi · Hernán Melgratti

Received: date / Accepted: date

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 specialised 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, it allows to check that the composition of a process with a cloud store ensures 'strong' properties through a wise usage of asynchronous message-passing; in this case we say that the process supports the consistency level provided by the cloud store. The proposed approach is compositional: the support of a consistency level is preserved by parallel composition when the preorder used to compare process-store ensembles is the weak simulation.

Keywords cloud · weak stores · eventual consistency · process calculi

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 requirements in a weaker notion of transaction, the so called 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 for high up-front investments. The range of available resources, offered as services, includes 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.

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. These requirements are summarised in [19] 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 by using replicas), highly available and partition tolerant. It is known (CAP theorem [13]) that one system cannot provide at the same time 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*. Even-

School of Computing, University of Kent
CT27NF, Canterbury, UK.
E-mail: L.Bocchi@kent.ac.uk

University of Buenos Aires
Pabelln I, Ciudad Universitaria. C1428EGA Buenos Aires, Argentina.
E-mail: hmelgra@dc.uba.ar

tual 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 are the most practicable solution in some 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.

We set the basis for a formal analysis of what 'suitable' and 'sometimes' means, so that general purpose applications can be safely run using these weaker memory models. In fact, the widespread availability of cloud infrastructures is broadening the range of applications using cloud data stores: not only specialised applications, such as those analysing large data using map-reduce, but also general purpose applications. It is therefore crucial to provide modelling primitives and analysis tools that help architects and developers to tackle the possible mismatch between the properties expected/needed and the properties provided by dynamically discovered resources, such as the BASE properties offered by cloud stores.

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 store. We focus on distributed and interoperable applications that can both communicate asynchronously, and use cloud stores. These applications can be naturally modelled as processes in CCS [18], a well known formalism for concurrent and distributed systems.

Existing literature on weak memory models and eventually consistent stores (see § 6 for an overview) focuses on the engineering of weak stores and on the properties they guarantee. The aim of our contribution is somehow orthogonal; that is analysing the observable behaviour of distributed applications using these stores. With respect to the existing work on the properties of systems that use weak stores, instead, we do not focus on a specific set of properties but, rather, we aim at analysing if the observable behaviour of an application can be preserved when using different types of stores. Namely, we address the following questions: which behaviour can we expect from an application, that was developed with a certain type of store in mind, when it uses a store providing weaker properties? When we run a specific application in a system where resources are provided on a pay-per-use basis, which level of consistency should we seek/pay for in order to still observe the desired behaviour?

This work is an extended version of [6]; the main improvements are summarised below:

1. The operational characterisation given in [6], although sufficient for illustrating how stores work, strongly binds the definition of a store to its current state. We now give a more abstract characterisation that differentiates a store from its states. Consequently, we are able to talk about the properties (i.e., the consistency level) satisfied by a store regardless of its current state. For instance, the consistency level of a store is now defined in terms of all its possible states (See Definition 19). Nevertheless, a new result (Proposition 5) allows us to restrict the comparison of the consistency level of two stores just to their initial states. This also shows that the new formalisation conservatively extends the presentation in [6], in which consistency levels are determined by the initial states of the stores.
2. We introduce a notion of compatibility between states (Definition 16 in § 4.3) to enable the comparison of stores supporting different datatypes or representing data differently. This notion is essential to the formulation of the aforementioned Proposition 5. The separation between stores and states, together with the notion of compatibility between states, enables coinductive reasoning on the behaviour of stores (e.g., Theorem 1).
3. Although still leaving our framework parametric with respect to the operations offered by stores as in [6], we have made our assumptions on these operations explicit. Namely, we introduced a notion of well-formedness for store operations (Definition 15), which induces a more realistic behaviour of applications over cloud storage, since read operations cannot block depending on the content of the store.
4. A new section (§ 5) discusses the practical applicability of the framework and states that the consistency level required by an application can be determined compositionally (Theorem 1) when systems are compared by using weak bisimulation.

We proceed as follows: we give one abstract (§ 2.1) and one operational (§ 2.2) characterisation of stores, strong and weak consistency, and an approach (§ 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 § 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 § 3 we provide three examples of stores with replicas, and analyse the consistency properties they provide using weak bisimulation. More precisely, in § 3.1 we present a weak store that guarantees eventual consistency, in § 3.2 we introduce an asynchronous version of the store in § 3.1 that cannot rely on an absolute (time) ordering between versions and does not to guarantee eventual consistency, and in § 3.3 we introduce a strong store. These stores are also used to illustrate the proposed approach in § 4. In § 5 we discuss on the applicability our framework and show that the consistency level required by an application can be checked compositionally (when taking weak bisim-

ulation as the observational equivalence). Related work and conclusion are given in § 6 and § 7, respectively.

2 Strong and Weak consistency

This section presents a formal approach to modelling cloud storages, which we will simply refer to as *stores*. Firstly, § 2.1 presents stores as abstract data types, whose operations come equipped with a denotational semantics, while § 2.2 presents the operational characterisation of stores that will be useful in the rest 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., $\sigma : \mathbb{K} \to \mathbb{V}$. Write Σ for the set of all possible states and assume $\mathbb{V}$ to include $\bot$, which denotes an undefined value. Hereafter, we assume that any store contains a distinguished initial state σ_0.

Stores are characterised in terms of their operations (i.e., queries). A store is defined in terms of a set $\mathcal{O} = \mathcal{W} \cup \mathcal{R}$, where the elements in $\mathcal{W}$ denote write operations and those in $\mathcal{R}$ stand for read operations. We require $\mathcal{W} \cap \mathcal{R} = \emptyset$ and 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.

The following examples illustrate the specification of three different stores. Example 1 introduces a strongly consistent store providing the operations **write** and **read** that atomically change and read the state of the store. Examples 2 and 3 define two weakly consistent stores providing a monotonic set datatype [10] whose values can be read (operation **read**) and modified by inserting a new element (operation **add**).

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. $\mathbb{I}(\mathbf{read}(o, \mathrm{v}))\sigma = \begin{cases} \mathtt{true} & \text{if } \sigma(o) = \mathrm{v} \\ \mathtt{false} & \text{otherwise} \end{cases}$

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. $\mathbb{I}(\mathbf{read}(o, \mathbb{V}))\sigma = \begin{cases} \mathtt{true} & \text{if } \sigma(o) = \mathbb{V} \\ \mathtt{false} & \text{otherwise} \end{cases}$

Example 3 An alternative characterisation for sets can obtained by changing the interpretation of action **read** as follows:

$$\mathbb{I}(\mathbf{read}(o, \mathbb{V}))\sigma = \begin{cases} \mathtt{true} & \text{if } \mathbb{V} \subseteq \sigma(o) \\ \mathtt{false} & \text{otherwise} \end{cases}$$

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 $\exists \mathrm{v} \in \mathbb{V}, \sigma \in \Sigma$ s.t. $\mathbb{I}(\alpha)\sigma \neq \mathbb{I}(\alpha)(\sigma[o \mapsto \mathrm{v}])$, i.e., when α actually depends on the value of o. We let $objects(\alpha)$ to be the objects read or written by the action α, i.e.,

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

Example 4 Consider $\mathcal{W}$, $\mathcal{R}$ and $\mathbb{I}$ defined in Example 1. Then, the operation $\mathbf{write}(o, \mathrm{v})$ modifies o' iff $o = o'$. Similarly, the operation $\mathbf{read}(o, \mathrm{v})$ 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 characterisation for weak specifications, namely an access to 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 some store modifications. In other words, the specification of a weak store does not require modifications to have immediate effects.

Definition 3 (weakly consistent specification) An interpretation $\mathbb{I}$ is weakly consistent 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 weakly 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

weakly 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 cannot be deferred.

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

Definition 4 (Last changed 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 $\sigma = \sigma'[o \mapsto \mathrm{v}]$, a read action $\alpha \in \mathcal{R}$ reads the value v for o in σ, written $\alpha \uparrow_{\sigma}^{o\mapsto \mathrm{v}}$, if $o \in \mathit{objects}(\alpha)$ 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 (Strongly consistent specification) An interpretation $\mathbb{I}$ is strongly consistent if

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

In words, a specification is strongly consistent when it requires every read action α of an object o corresponds to the last changed value of o.

Example 6 It is easy to check that the specification in Example 1 is strongly 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) $o \neq o'$ and hence $o' \notin \mathit{objects}(\alpha)$ or (2) $o = o'$ and $\alpha \in (\sigma[o' \mapsto \mathrm{v}'] \bowtie) \backslash (\sigma \bowtie)$. In both cases, $\alpha \uparrow_{\mathbb{I}(\beta)\sigma}^{o'\mapsto \mathrm{v}'}$. Analogously, it can be shown that the store in Example 2 is also a strongly consistent specification. Differently, the store in Example 3 is not a strongly 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 $\alpha \uparrow_{\mathbb{I}(\beta)\sigma}^{o\mapsto \{\mathrm{v}\}}$ because $\alpha \in \sigma \bowtie$.

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

2.2 Stores as Labelled Transition Systems

In what follows we will find it useful to rely on an operational characterisation of stores in terms of labelled transitions systems. The operational characterisation of a store, or simply *store*, is a triple $\ell = (\mathcal{L}, \mathbb{I}, \rightarrow_{\ell})$ where $\mathcal{L}$ is a set of labels, $\mathbb{I}$ is an interpretation function, and $\rightarrow_{\ell}$ is a labelled transition relation on store states defined on labels $\mathcal{L}$. $\mathcal{L} = \mathcal{O} \cup \mathcal{S}$ is the union of read/write actions $\mathcal{O}$ and internal or silent actions $\mathcal{S}$. We use $\tau, \tau', \ldots$ to range over $\mathcal{S}$.

We can think of a store $(\mathcal{O} \cup \mathcal{S}, \mathbb{I}, \rightarrow_{\ell})$ as an operational implementation of the specification $(\mathcal{O}, \mathbb{I})$. The implementation defines the precise mechanisms of interactions with the store, by transitions with labels in $\mathcal{O}$, as well as mechanisms for preserving or restoring consistency, synchronising distributed replicas, etc., by transitions with labels in $\mathcal{S}$.

Definition 7 (Reachable states) We say that σ is a reachable state of ℓ, or simply a state of ℓ, if $\sigma_0 \longrightarrow_{\ell}^{*} \sigma$, with σ_0 being the initial state of ℓ.

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 $\mathcal{A}$. For a sequence $t = \alpha_0 \ldots \alpha_k ...$, 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 $\alpha_0 \ldots \alpha_j \in \mathcal{W}^*$, its interpretation is the function accounting for the composition of all α_i, i.e., $\mathbb{I}(\alpha_0 \ldots \alpha_j) = \mathbb{I}(\alpha_j) \circ \ldots \circ \mathbb{I}(\alpha_0)$. As usual, for $t = \alpha_0 \ldots \alpha_k$ and σ state of ℓ, we will write $\sigma \stackrel{t}{\Longrightarrow}_{\ell} \sigma'$ if $\exists \sigma_0, \ldots, \sigma_{k+1}$ s.t. $\sigma_i \xrightarrow{\alpha_i}_{\ell} \sigma_{i+1}$, $\sigma_0 = \sigma$ and $\sigma_{k+1} = \sigma'$. Moreover, for $t \in \mathcal{O}^*$ we also write $\sigma \stackrel{t}{\Longrightarrow}_{\ell} \sigma'$ if there exists $t' \in \mathcal{L}^*$ s.t. $\sigma \stackrel{t'}{\Longrightarrow}_{\ell} \sigma'$ and $t = t' \downarrow \mathcal{O}$.

Definition 8 (Trace) Let $\ell = (\mathcal{L}, \mathbb{I}, \rightarrow_{\ell})$ be a store and σ a state of ℓ. A sequence $t \in \mathcal{L}^*$ s.t. $\sigma \stackrel{t}{\Longrightarrow}_{\ell} \sigma'$ is a trace from σ satisfying $\langle \mathcal{O}, \mathbb{I} \rangle$ when $t[i] \in \mathcal{R}$ implies $\mathbb{I}(t[0..i] \downarrow_{\mathcal{W}})\sigma \bowtie t[i]$, i.e., any read is valid with respect to the store obtained after applying (in order) all preceding operations in $\mathcal{W}$. We write $tr(\sigma)$ for the set of all traces from σ.

The above definition provides an operational characterisation of the consistency criterion given by Definition 2. In order to handle strongly consistent stores, we introduce a finer notion that captures the dependencies between read and write operations.

Definition 9 (Read-write dependency) The read-write dependency relation $\leftrightarrow \subseteq \mathcal{R} \times \mathcal{W}$, is defined as follows

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

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 10 (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 strongly consistency given in [27], that is: '*Every read on a data item x returns a value corresponding to the result of the most recent write on x*'.

Definition 11 (strongly consistent trace and store) A trace t is strongly consistent if for all $\beta \rightsquigarrow \alpha \in t$, $\beta \downarrow_{\sigma}^{o \mapsto \mathrm{v}}$ implies $\alpha \uparrow_{\mathbb{I}(\beta)\sigma}^{o \mapsto \mathrm{v}}$. A store is strongly consistent if every trace is strongly consistent.

Lemma 1 establishes a correspondence between Definition 6 and the operational definition of strongly consistent trace/store in Definition 11.

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

Proof Consider $t \in tr(\sigma_0)$ and $\beta \rightsquigarrow \alpha \in t$. Then $\exists i, j$ s.t. $i < j$, $t[j] = \alpha \in \mathcal{R}$ and $t[i] = \beta \in \mathcal{W}$. Let $\sigma = \mathbb{I}(t[0..i-1] \downarrow_{\mathcal{W}})\sigma_0$ and $\sigma' = \mathbb{I}(t[0..j-1] \downarrow_{\mathcal{W}})\sigma_0$. Since t satisfies $\langle \mathcal{O}, \mathbb{I} \rangle$, we have $\sigma' \bowtie \alpha$. Moreover, $\beta \rightsquigarrow \alpha \in t$ implies $\forall i < k < j.t[k] \in \mathcal{W} \Rightarrow objects(t[j]) \cap objects(t[k]) = \emptyset$ and hence $\mathbb{I}(t[0..i] \downarrow_{\mathcal{W}})\sigma_0 \bowtie \alpha$. Consequently,

$$\mathbb{I}(\beta)(\sigma) \bowtie \alpha \tag{1}$$

Additionally, by Definition 6, it holds that

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

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

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., [5, 28]) used formulation of eventual consistency is the one in [28]: '*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 [25]) which does not require absence of updates, 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 stabilise, 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 synchronised.

Definition 12 (Eventual consistency) A store ℓ with initial state σ_0 is eventually consistent if $\forall t \in tr(\sigma_0)$ the following holds: Let $t[i] \in \mathcal{W}$ then $\exists k > i. \forall j > k. t[j] \leftrightarrow t[i]$ implies:

1. $t[i] \downarrow_{\sigma}^{o \mapsto v}$ and $t[j] \uparrow_{\mathbb{I}(t[i])\sigma}^{o \mapsto v}$, 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 $t[j] \uparrow_{\mathbb{I}(t[h])\sigma}^{o \mapsto v}$, 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 simplicity, we avoid the definition of even weaker notions as the ones in which events are not globally ordered [11], which could also be recast into this framework.

3 Two Weak and One Strong Store

In this section we consider three examples of stores, two weak stores with replicas and distributed control as, e.g., Amazon S3 [15] (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 state 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 minimal element n_0. We let 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 $\forall (\mathrm{v}, n') \in \sigma(o),\ n \neq n'$. 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 $i \in \{1, .., m\}$. 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 semantics 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 vectors, 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 happen at different times). The assumption of a global notion of time is reasonable in some scenarios. For instance, as discussed in [3], multiprocessors like Alpha [1] and Sun [26] rely on a global timeline. In cloud systems, an approximated global timeline can be obtained by synchronising the local clocks using, for instance, the Precision Time Protocol (PTP) [2], or by means of an initial synchronisation when one can guarantee that time flows at the same pace for all local clocks. These are approximate solutions allowing some time discrepancy among the local clocks; their applicability depends on whether this discrepancy is negligible with respect to the expected temporal distance between pairs of subsequent write operations on store.

Definition 13 (Weak store) A (replicated, distributedly controlled) weak store is defined as $w = (\mathcal{W} \cup \mathcal{R} \cup \{\tau\}, \mathbb{I}, \rightarrow_w)$ where:

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

and the LTS $\rightarrow_w$ is given by the rules in Fig. 1. The initial state is σ_0 s.t. $\sigma_0(o) = \{(\bot, n_0)\}$ for any $o \in \mathbb{K}$.

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, which is greater than all version vectors of the existing versions of the same object. A *propagation* [WPRO] models the asynchronous communication among the replicas to achieve a consistent view of the data.

3.1.1 Weak store and strong consistency

Proposition 1 shows that the weak store w is not strongly consistent as it does not satisfy Definition 11.

Proposition 1 *The weak store w is not strongly consistent.*

Proof We recall that in the initial state σ_0, $\sigma_0(o) = \{(\bot, n_0)\}$ for any o. Then, $tr(\sigma_0)$ includes

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

obtained by applying, in sequence, rules [WWRITE] and [WREAD] to σ_0, i.e.,

$$\sigma_0 \xrightarrow{\mathbf{write}(o,10)}_w \sigma_1 \xrightarrow{\mathbf{read}(o,(\bot,n_0))}_w \sigma_1$$

where $\sigma_1(o) = \{(\bot, n_0), (10, n_1)\}$ for $n_1 = n_0 + 1$. Then $t[0] = \mathbf{write}(o, 10) \in \mathcal{W}$, $t[1] = \mathbf{read}(o, (\bot, n_0)) \in \mathcal{R}$, $t[0] \rightsquigarrow t[1]$, and $t[0] \downarrow_{\sigma_0}^{o \mapsto \{(\bot,n_0),(10,n_1)\}}$. However, it is not the case that $t[1] \uparrow_{\sigma_1}^{o \mapsto \{(\bot,n_0),(10,n_1)\}}$ because $\sigma_1 \bowtie \neq \sigma_0 \bowtie$ but $t[1] \notin ((\sigma_1 \bowtie) \setminus \sigma_0 \bowtie)$. Namely, the read action $t[1]$ does not ensure to get *only* the most recent written version $(10, n_1)$ if it takes place too early to allow the synchronization that restores the consistency of the replicas. □

3.1.2 Weak store and eventual consistency

We show below that w ensures eventual consistency (Definition 12), after a few auxiliary lemmas and assuming that all $\sigma, \sigma_j, \ldots$ are states of w. First we observe that all version vectors for the same object in a state σ are distinguished.

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

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 $n+1 > m$ (hence fresh) for each stored version vector m.

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

In fact, the only rule that creates new version vectors is [WWRITE] and it introduces a new value n that is comparable with all the existing vectors associated with the same object.

Next we observe that the number of versions associated with an object o in subsequent states is monotonically non-increasing (Lemma 4) and will eventually be just one (Lemma 5) when there are no write actions on o. 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 $t = t'[i..\infty[\, \downarrow_{(\mathcal{L} \setminus \mathcal{W}_{\{o\}})}$ with $\sigma_j \xrightarrow{t[j+1]}_w \sigma_{j+1}$ for all $j \geq i$. Then, $\forall j \geq i.\, |\sigma_j(o)| \geq |\sigma_{j+1}(o)|$.*

Proof Assume $|\sigma_j(o)| = m_o$ 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] $|\sigma_j(o)| = |\sigma_{j+1}(o)| = m_o$
- $t_{j+1} \in \mathcal{R}$ hence, by [WREAD] $|\sigma_j(o)| = |\sigma_{j+1}(o)| = m_o$
- $t_{j+1} \in \mathcal{S}$ hence, by [WPRO] either $|\sigma_j(o)| = |\sigma_{j+1}(o)| = m_o$ or, if one version of o is removed, $|\sigma_{j+1}(o)| = m_o - 1$.

Lemma 5 *Consider the trace $t = t'[i..\infty[\, \downarrow_{(\mathcal{L} \setminus \mathcal{W}_{\{o\}})}$ with $\sigma_j \xrightarrow{t[j+1]}_w \sigma_{j+1}$ for all $j \geq i$. Then, $\exists j \geq i.\, |\sigma_j(o)| = 1$, assuming strong fairness [1] in the LTS in Fig. 1.*

[1] 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[16]).

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

$$\frac{(\mathrm{v}_i, n_i) \in \sigma(o) \quad i \in \{1,2\} \quad n_1 > n_2 \quad \sigma' = \sigma[o \mapsto \sigma(o) \setminus \{(\mathrm{v}_2, n_2)\}]}{\sigma \xrightarrow{\tau}_w \sigma'} \text{ [WPRO]}$$

Fig. 1 Labelled transition relation for weak store w

Proof Consider any $j \geq i$, we have two cases: either $|\sigma_j(o)| = 1$, or $|\sigma_j(o)| > 1$. In the first case, by Lemma 4, $\forall l \geq j.\ |\sigma_l(o)| = 1$. In the second case, $\sigma_j(o) \ni (\mathrm{v}_1, n_1) \neq (\mathrm{v}_2, n_2)$. 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 w is eventually consistent (by Definition 12) assuming strong fairness in the LTS in Fig. 1.*

Proof Let t be an infinite trace and let $t[i] \in \mathcal{W}$. By Definition 13, $t[i]$ 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 σ we obtain a trace t' and there is a k such that $|\sigma_k(o)| = 1$. 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) $\mathtt{w} \neq \mathrm{v}$ (case 2 of Definition 12), and (2) that there is no write action $\beta = t[h] \in t[i, k']$ such that $\beta \downarrow^{o \mapsto \mathtt{w}}_{\sigma_h}$. Then either $\mathrm{v} = \bot$ or there is a write action $t[j] \downarrow^{o \mapsto \mathtt{w}}_{\sigma_j}$ such that $j < i$. In both cases, by rulename [WWRITE] the version vector associated with $\mathtt{w}$ 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 an *asynchronous weak store* where keys are associated with sets of versioned objects. In this case replicas cannot rely on an absolute ordering of versions w.r.t. the global timeline. This is reflected in the rule for writing new versions, [AWRITE] in Figure 2: the version vector of the newly introduced value is greater than the version vector of one replica but possibly incomparable with the version vectors in other replicas.

The asynchronous weak store is defined as $a = (\mathcal{W} \cup \mathcal{R} \cup \{\tau\}, \mathbb{I}, \rightarrow_a)$ using the interpretation and labels in Definition 13. The transition rules are shown in Fig. 2; they are as the rules for weak stores in Fig. 1 except the writing rule [AWRITE] (which is different from [WWRITE]) and a new rule [ASOL] which is necessary to resolve conflicts arising during replicas synchronisation when two version vectors are incomparable.

With a similar argument as in Proposition 1 one can show that the asynchronous weak store a is not strongly consistent. Furthermore, the asynchronous weak store is not eventually consistent, according to Definition 12.

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

Consider the initial state σ_0 for a such that $\sigma_0(o) = \{(\bot, n_0)\}$, then $traces(\sigma_0)$ includes the following family of traces

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

with $t' \downarrow_{\mathcal{W}_{\{o\}}} = \emptyset$. Any trace t in the family of traces above can be obtained as follows:

1. By [AWRITE], $\sigma_0 \xrightarrow{\mathbf{write}(o,10)}_a \sigma_1$ where $\sigma_1(o) = \{(\bot, n_0), (10, n_1)\}$.
2. By [AWRITE], $\sigma_1 \xrightarrow{\mathbf{write}(o,5)}_a \sigma_2$ where $\sigma_2(o) = \{(5, n_2), (10, n_1)\}$ and $n_1 <> n_2$.
3. By [ASOL], $\sigma_2 \xrightarrow{\tau}_a \sigma_3$ where $\sigma_3(o) = \{(10, n_3)\}$ with $n_3 > n_1, n_2$.

Any read of o over σ_3 will return $(10, n_3)$. This violates Definition 12 because there is no k after which every read will return the value written either by $\mathbf{write}(o, 5)$ or by a more recent write operation.

However, the properties expressed by Lemmas 4 and 5 hold also for the asynchronous store a. 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. In fact, in absence of write operations, rule AWRITE is not applied and hence the size of $\sigma(o)$ is not increasing for every o and, by repeatedly applying rule APRO or ASOL, the store will eventually converge to a state σ' such that $|\sigma'(o)| = 1$ for all o, as shown by the following two results.

Lemma 6 *Consider the trace $t = t'[i..\infty[\ \downarrow_{(\mathcal{L} \setminus \mathcal{W}_{\{o\}})}$ with $\sigma_j \xrightarrow{t[j+1]}_a \sigma_{j+1}$ for all $j \geq i$. Then, $\forall j \geq i.\ |\sigma_j(o)| \geq |\sigma_{j+1}(o)|$.*

$$\frac{(\mathrm{v},n)\in\sigma(o)}{\sigma\xrightarrow{\mathbf{read}(o,(\mathrm{v},n))}_a\sigma}\ [\textsc{ARead}] \qquad \frac{(\mathrm{v}',n')\in\sigma(o)\quad \sigma'=\sigma[o\mapsto\sigma(o)\cup\{(\mathrm{v},n)\}]\quad n=n'+1}{\sigma\xrightarrow{\mathbf{write}(o,\mathrm{v})}_a\sigma'}\ [\textsc{AWrite}]$$

$$\frac{(\mathrm{v}_i,n_i)\in\sigma(o)\quad i\in\{1,2\}\quad n_1<>n_2\qquad n_3=\{n_1,n_2\}\quad \sigma'=\sigma[o\mapsto(\sigma(o)\setminus\{(\mathrm{v}_i,n_i)\mid i\in\{1,2\}\}\cup\{(\mathrm{v}_1,n_3)\})]+1}{\sigma\xrightarrow{\tau}_a\sigma'}\ [\textsc{ASol}]$$

$$\frac{(\mathrm{v}_i,n_i)\in\sigma(o)\quad i\in\{1,2\}\quad n_1>n_2\quad \sigma'=\sigma[o\mapsto\sigma(o)\setminus\{(\mathrm{v}_1,n_1)\}]}{\sigma\xrightarrow{\tau}_a\sigma'}\ [\textsc{APro}]$$

Fig. 2 Labelled transition relation for asynchronous weak store a

Proof Assume $|\sigma_j(o)|=m_o$ 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 [AWRITE] $|\sigma_j(o)|=|\sigma_{j+1}(o)|=m_o$
- $t_{j+1}\in\mathcal{R}$ hence, by [AREAD] $|\sigma_j(o)|=|\sigma_{j+1}(o)|=m_o$
- $t_{j+1}\in\mathcal{S}$ hence, by [APRO] or [ASOL] either $|\sigma_j(o)|=|\sigma_{j+1}(o)|=m_o$ or, if one version of o is removed, $|\sigma_{j+1}(o)|=m_o-1$.

Lemma 7 *Consider the trace $t=t'[i..\infty[\ \downarrow_{(\mathcal{L}\setminus\mathcal{W}_{\{o\}})}$ with $\sigma_j\xrightarrow{t[j+1]}_a\sigma_{j+1}$ for all $j\geq i$. Then, $\exists j\geq i.\ |\sigma_j(o)|=1$, assuming strong fairness in the LTS in Fig. 2.*

Proof Consider any $j\geq i$, we have two cases: either $|\sigma_j(o)|=1$, or $|\sigma_j(o)|>1$. In the first case, by Lemma 6, $\forall l\geq j.\ |\sigma_l(o)|=1$. In the second case, $\sigma_j(o)\ni(\mathrm{v}_1,n_1)\neq(\mathrm{v}_2,n_2)$. If $n_1<>n_2$ then transition [ASOL] is enabled and hence, by fairness, it will eventually execute so to decrease the number of versions (which will not, in the meanwhile, increase by Lemma 6). Similarly, if $n_1<n_2$ (resp. $n_2<n_1$) then transition [APRO] is enabled and hence it will eventually decrease the number of versions. □

3.3 Strong Store

We now model a strong store s (with version vectors) that satisfies *strong consistency* (Definition 11). The values associated with objects are pairs (v,n) where v is the value of the object and n is the corresponding version vector.

Definition 14 (Strong store) A strong store is defined as $s=(\mathcal{W}\cup\mathcal{R},\mathbb{I},\rightarrow_s)$ where:

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

and $\rightarrow_s$ is defined by the rules in Fig. 3. The initial state is σ_0 s.t. with $\sigma_0(o)=(\bot,n_0)$ for all $o\in\mathbb{K}$.

$$\frac{(\mathrm{v},n)=\sigma(o)}{\sigma\xrightarrow{\mathbf{read}(o,(\mathrm{v},n))}_s\sigma}\ [\textsc{SRead}] \qquad \frac{(\mathrm{v}',n')\in\sigma(o)\quad \sigma'=\sigma[o\mapsto(\mathrm{v},n)]\quad n=n'+1}{\sigma\xrightarrow{\mathbf{write}(o,\mathrm{v})}_s\sigma'}\ [\textsc{SWrite}]$$

Fig. 3 Labelled transition relation for strong store s

Proposition 4 *The strong store s is strongly consistent.*

Proof Let $t\in traces(\sigma_0)$ with $\sigma_0(o)=(\bot,n_0)$ 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 $objects(t[i])=objects(t[j])=\{o\}$ for some $o\in\mathbb{K}$. The proof is by induction on the distance $j-i$ between the write and read actions.

Base case ($j=i+1$). The only possible transition from σ_{i-1} to σ_i is by [SWRITE]

$$\sigma_{i-1}\xrightarrow{\mathbf{write}(o,\mathrm{v})}_s\sigma_i=\sigma_{i-1}[o\mapsto(\mathrm{v},n)]$$

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

$$\frac{\sigma_i(o)=(\mathrm{v},n)}{\sigma_i\xrightarrow{\mathbf{read}(o,(\mathrm{v},n))}_s\sigma_j=\sigma_i}$$

hence $t[i]\downarrow^{o\mapsto(\mathrm{v},n)}_{\sigma_{i-1}}$ and $t[j]\uparrow^{o\mapsto(\mathrm{v},n)}_{\sigma_i}$.

Inductive case ($j>i+1$). 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 $\sigma_{i-1}=\sigma_i$. By induction we have that reading o in position $j-1$ returns (v,n) hence it will return (v,n) also in σ_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 σ_j for o. □

4 Programs accessing stores

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, as well as communicate through message passing *à la* value-passing CCS [18].

Memory-sharing and message passing are usually considered as alternative programming paradigms; for what concerns expressiveness, we could have provided an equivalent model that uses only one of them. We chose to include both memory-sharing and message passing primitives to provide a realistic model for cloud systems, where memory and network are distinct and possibly competing facilities (e.g., a user can decide when to pay for memory or for network facilities). Featuring both paradigms enables us to reason on the relationship between store properties and the communicating behaviour of processes.

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 sections, 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 and $var(e)$ to denote the set of variables occurring in e. 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 reads from the store the value w associated with the key v, which has been previously received over the channel x. Similarly, we would like to consider programs such as $x(v).y(w).\mathbf{write}(o, v + w)$, which updates the value associated to the key o with the result of evaluating an expression. For this reason, we consider *program actions over stores* with the following shape $\mathbf{operation\_name}(e_1, \ldots, e_n)$. By abusing the notation, we will write $\mathbf{operation\_name}(e_1, \ldots, e_n) \in \mathcal{R}$ (respectively, $\mathbf{operation\_name}(e_1, \ldots, e_n) \in \mathcal{W}$) when there exists a ground substitution over the variables in $e_1, \ldots, e_n$, which applied to $\mathbf{operation\_name}(e_1, \ldots, e_n)$ gives an operation in $\mathcal{R}$ (resp., $\mathcal{W}$). In what follows we assume that programs use the right arities in store operations. Although we do not fix a particular set of store operations, we require read operations to satisfy the well-formedness conditions in Definition 15, e.g., preventing the use of pattern matching over read values.

Definition 15 (Well-formedness of read operations) The following prefix

$$\mathbf{operation\_name}(e_1, \ldots, e_n) \in \mathcal{R}$$

is well-formed if it satisfies the following two conditions:

1. $|\{e_1, \ldots, e_n\} \cap \mathbb{K}| = 1$;
2. $(\{e_1, \ldots, e_n\} \setminus \mathbb{K}) \subseteq \mathbb{V}$;
3. $|\{e_1, \ldots, e_n\} \setminus \mathbb{K}| = n - 1$.

In Definition 15: (1) requires that each read operation accesses just one object, e.g., $\mathbf{read}(o_1, o_2, v_1, v_2)$ is not well-formed whereas $\mathbf{read}(o_1, v_1)$ is well-formed; (2) requires that all parameters except the object read are variables, e.g., $\mathbf{read}(o_1, \mathrm{v})$ is not well-formed whereas $\mathbf{read}(o_1, v)$ is well-formed; by (3) all the variables must be different, e.g., operation $\mathbf{read}(o, v, v)$ is not well-formed whereas operation $\mathbf{read}(o, v_1, v_2)$ is well formed.

We use $\mathcal{A}$ for the set of program actions over stores, and let ρ, ρ' range over $\mathcal{A}$. Then, the syntax of programs acting over stores is given by the grammar.

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

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 an 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 $fn(P)$ to denote the set of free (channel and variable) names of P, which is defined as follows

$$\begin{array}{ll}
fn(\overline{x}e) & = \{x\} \cup var(e) \\
fn(x(v).P) & = \{x\} \cup (fn(P) \setminus \{v\}) \\
fn(\mathbf{op}(e_1, \ldots, e_n).P) & = fn(P) \setminus var(e_1, \ldots, e_n) \\
fn(\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2) & = var(e) \cup fn(P_1) \cup fn(P_2) \\
fn(\nu x.P) & = fn(P) \setminus \{x\} \\
fn(P_1||P_2) & = fn(P_1) \cup fn(P_2) \\
fn(!P) & = fn(P) \\
fn(0) & = \emptyset
\end{array}$$

As usual, we will restrict ourselves to consider closed programs.

A program P interacting with a store ℓ in state σ is called a *system* and it is denoted by $[P]^{\ell}_{\sigma}$.

4.2 Operational Semantics

The labelled transition relation for systems uses the following labels:

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

In addition to the standard labels for CCS ($\tau, \overline{x}\mathrm{v}, x(v)$), we also consider the ground store operations $\alpha \in \mathcal{W} \cup \mathcal{R}$ as labels, i.e., programs perform concrete operations over the store. Given a label μ, $subj(\mu)$ denotes the subject of the action (defined as usual for CCS labels while it is undefined for α).

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 $\xrightarrow{\alpha}_\ell$ with e.g., $\ell \in \{w, a, s\}$ as defined in the 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 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 ℓ which is in state 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 (internal) steps of the store.

4.3 Reasoning about programs

In this section we address the problem of characterising the consistency levels offered by different stores in terms of the behaviour of the programs using them. For instance, we would like to conclude that the consistency level provided by the store w in Definition 13 is weaker than the consistency level provided by the store s in Definition 14, by comparing the behaviour of programs running over w and s. Consider the program

$$P \triangleq \mathbf{write}(o, \mathrm{v}); \mathbf{read}(o, (v', n)); \overline{show}(o, v')$$

and the stores s and w with the initial state σ_w and σ_s such that $\sigma_w(o) = \sigma_s(o) = (\mathtt{v}_0, \mathtt{n})$ with $\mathtt{v} \neq \mathtt{v}_0$. It is easy to observe that P running over w may exhibit more traces than P running over s. In fact,

$$[P]^w_{\sigma_w} \xrightarrow{\tau}_w \xrightarrow{\tau}_w \xrightarrow{\overline{show}(o, \mathtt{v}_0)}_w$$

but this behaviour cannot be mimicked by the system $[P]^s_{\sigma_s}$, which can only show the value $\mathtt{v}$. On the contrary, all traces of P over s can be simulated by P running over w.

In the rest of this section we provide the formal tools that enables the comparison among the consistency levels provided by different stores. We rely on some behavioural preorder on systems (e.g., standard weak simulation), denoted by $\precsim$ (and $\sim$ for the associated equivalence). We characterise the consistency level provided by a store ℓ with semantics $\longrightarrow_\ell$ relatively to the consistency level of other store ℓ' with semantics $\longrightarrow_{\ell'}$ by comparing the traces of the systems $[P]^\ell_\sigma$ and $[P]^{\ell'}_{\sigma'}$, where σ and σ' are required to be 'compatible' states (instead of, for instance, comparing ℓ and ℓ' using the same state).

The notion of compatibility between states is needed for two reasons. The first reason is that ℓ and ℓ' may associate different types of values to the keys, hence it may not be possible to analyse the behaviour ℓ and ℓ' with respect to the same state (e.g., w in § 3.1 associates keys to pairs (v, n) of value and version vector and we may want to compare its behaviour with a store that associates keys to values v). The second reason is that when we compare ℓ and ℓ' it may be too restrictive (e.g., if using a weak simulation) to require that ℓ and ℓ' produce the same states from a σ after each transition. For instance, despite $[P]^\ell_\sigma \xrightarrow{\mu}_\ell [P']^\ell_{\sigma'}$ and $[P]^{\ell'}_\sigma \xrightarrow{\mu}_{\ell'} [P']^{\ell'}_{\sigma''}$ with $\sigma' \neq \sigma''$, P' may have the same observable behaviour in the two stores. Therefore, we allow ℓ and ℓ' to be verified in different states σ and σ' (e.g., $[P]^\ell_\sigma \precsim [P]^{\ell'}_{\sigma'}$) as long as they are compatible, that is obtained by applying the same set of write actions to the initial states of ℓ and ℓ', respectively.

Definition 16 (Compatible states) Consider two stores ℓ_1 and ℓ_2 with initial states σ^1_0 and σ^2_0, respectively. We say that states σ^1 and σ^2 are compatible with respect to ℓ_1 and ℓ_2, written $\sigma^1 \ {}_{\ell_1}\!\sim_{\ell_2} \sigma^2$, if there exists a process P_W such that $[P_W]^{\ell_1}_{\sigma^1_0} \longrightarrow^* [0]^{\ell_1}_{\sigma^1}$ and $[P_W]^{\ell_2}_{\sigma^2_0} \longrightarrow^* [0]^{\ell_2}_{\sigma^2}$.

Definition 17 We say ℓ is stronger than ℓ', written $\ell \prec \ell'$, whenever $[P]^\ell_\sigma \precsim [P]^{\ell'}_{\sigma'}$ for all P and $\sigma \ {}_\ell\!\sim_{\ell'} \sigma'$. We write $\ell \cong \ell'$ when $\ell \prec \ell'$ and $\ell' \prec \ell$.

In practice, it is not necessary to show $[P]^\ell_\sigma \precsim [P]^{\ell'}_{\sigma'}$ for all possible states; by the following result it is sufficient to show the same result by only considering the initial states of ℓ and ℓ', respectively.

Proposition 5 *$[P]^\ell_{\sigma_0} \precsim [P]^{\ell'}_{\sigma'_0}$ for all P implies $[P]^\ell_\sigma \precsim [P]^{\ell'}_{\sigma'}$ for all P and $\sigma \ {}_\ell\!\sim_{\ell'} \sigma'$.*

Proof Let $\sigma \ {}_\ell\!\sim_{\ell'} \sigma'$. Then, there exists $[P_W]^{\ell_1}_{\sigma_0} \longrightarrow^* [0]^{\ell_1}_\sigma$ and $[P_W]^{\ell_2}_{\sigma'_0} \longrightarrow^* [0]^{\ell_2}_{\sigma'}$. Consider the program $P' = P_W; P$ with $\_;\_$ being a suitable encoding of the sequential composition of two programs. By hypothesis, $[P]^\ell_{\sigma_0} \precsim [P]^{\ell'}_{\sigma'_0}$ for all P and, in particular, for P'. Then, $[P_W; P]^\ell_{\sigma_0} \precsim [P_W; P]^{\ell'}_{\sigma'_0}$ (with $P_W; P$). Consequently, $[P]^\ell_\sigma \precsim [P]^{\ell'}_{\sigma'}$.

The different consistency levels can be thought as the equivalence classes of $\cong$. We write $[\ell]_\cong$ for the representative of the equivalence class of ℓ.

$$\frac{e \longrightarrow^{e} \mathrm{v}}{\overline{x}e \xrightarrow{\overline{x}\mathrm{v}} 0}\ [\textsc{Out}] \qquad x(v).P \xrightarrow{x(v)} P\ [\textsc{In}] \qquad \frac{\rho \longrightarrow^{e}_{\theta} \alpha}{\rho.P \xrightarrow{\alpha} P\theta}\ [\textsc{store}] \qquad \frac{P \xrightarrow{\overline{x}\mathrm{v}} P' \quad Q \xrightarrow{x(v)} Q'}{P||Q \xrightarrow{\tau} P'||Q'[\mathrm{v}/v]}\ [\textsc{Com}]$$

$$\frac{P \xrightarrow{\mu} P' \quad bn(\mu) \cap fn(Q) = \emptyset}{P||Q \xrightarrow{\mu} P'||Q}\ [\textsc{Par}] \qquad \frac{P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'||!P}\ [\textsc{Rec}] \qquad \frac{P \xrightarrow{\mu} P' \quad x \neq subj(\mu)}{\nu x.P \xrightarrow{\mu} \nu x.P'}\ [\textsc{New}]$$

$$\frac{e \longrightarrow^{e} \texttt{true} \quad P_1 \xrightarrow{\alpha} P_1'}{\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\mu} P_1'}\ [\textsc{If-then}] \qquad \frac{e \longrightarrow^{e} \texttt{false} \quad P_2 \xrightarrow{\alpha} P_2'}{\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\mu} P_2'}\ [\textsc{If-else}]$$

$$\frac{\sigma \xrightarrow{\alpha}_{\ell} \sigma' \quad P \xrightarrow{\alpha} P'}{[P]^{\ell}_{\sigma} \xrightarrow{\tau} [P']^{\ell}_{\sigma'}}\ [\textsc{Store-Int}] \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \notin \mathcal{L}}{[P]^{\ell}_{\sigma} \xrightarrow{\mu} [P']^{\ell}_{\sigma}}\ [\textsc{Proc}] \qquad \frac{\sigma \xrightarrow{\alpha}_{\ell} \sigma' \quad \alpha \in \mathcal{S}}{[P]^{\ell}_{\sigma} \xrightarrow{\tau} [P]^{\ell}_{\sigma'}}\ [\textsc{Sync}]$$

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

Example 8 If we fix $\precsim$ to be the weak simulation $\precsim_{\mathsf{ws}}$, then we can show that the strongly consistent store s (§ 3.3) is stronger than both the weakly consistent store w (§ 3.1) and the asynchronous weakly consistent store a (§ 3.2). This can be done by showing that the following relations are weak simulations:

$$S_{\mathcal{S},\mathcal{W}} = \{(\sigma s_i, \sigma w_i) \mid \forall o.\sigma s_i(o) \in \sigma w_i(o)\}$$
$$S_{\mathcal{S},\mathcal{A}} = \{(\sigma s_i, \sigma a_i) \mid \forall o.\sigma s_i(o) \in \sigma a_i(o)\}$$

Therefore $\sigma s_0 \precsim_{\mathsf{ws}} \sigma w_0$ and $\sigma s_0 \precsim_{\mathsf{ws}} \sigma a_0$. Since $\precsim_{\mathsf{ws}}$ is a precongruence w.r.t. parallel composition, we conclude that $[P]^{s}_{\sigma s_0} \precsim_{\mathsf{ws}} [P]^{w}_{\sigma w_0}$ and $[P]^{s}_{\sigma s_0} \precsim_{\mathsf{ws}} [P]^{a}_{\sigma a_0}$ for all P. Analogously, it can be shown that $w \prec a$ by 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 $w \not\prec s$. It is enough to consider the trace shown in proof of Proposition 1, which cannot be mimicked by s. Similarly, it can be shown that $a \not\prec s$ and $a \not\prec w$.

Let 0 be the store providing no operation. When considering $\precsim$ as weak simulation $\precsim_{\mathsf{ws}}$, we have $0 \precsim_{\mathsf{ws}} \ell$ for all ℓ. Hence, $[P]^{0}_{\sigma_0} \precsim_{\mathsf{ws}} [P]^{\ell}_{\sigma}$ for all P, σ and ℓ. This means that 0 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 stores, i.e., stores in which write and read operations are enabled in every reachable state.

Definition 18 ℓ is available in state σ whenever the following two conditions hold:

- $\forall \alpha \in \mathcal{W} : \sigma \xrightarrow{\alpha}_{\ell}$, the store in state σ allows any write operation at any time.
- $\forall \rho = \mathbf{op}(e_1, \ldots, e_n) \in \mathcal{R}$ there exists θ s.t. $\rho \longrightarrow^{e}_{\theta} \alpha$ and $\sigma \xrightarrow{\alpha}_{\ell} \sigma'$, i.e., the store allows any well-formed read at any time.

We call ℓ available if it is available in every reachable state of ℓ.

Example 9 The three stores in § 3 provide availability. In fact, from the LTSs, we can conclude that for any pair of object o and value v, the transition $\sigma \xrightarrow{\mathbf{write}(o,\mathrm{v})}_{\ell}$ (with $\ell \in \{w, a, s\}$) can be derived for any state σ. Similarly, for any object o there is always a pair (v, n) s.t. $\sigma \xrightarrow{\mathbf{read}(o,(\mathrm{v},n))}_{\ell}$ can be performed.

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

Lemma 8 *Let s the strong store defined by the transition rules in Fig. 3. Then $\ell \prec s$ and ℓ available imply $s \prec \ell$ when assuming $\preceq$ to be $\precsim_{\mathsf{ws}}$.*

Proof By showing that the following relation is a weak simulation where σs_i denote states of s and σ_i denote states of ℓ:

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

We proceed by case analysis on $\sigma s_i \xrightarrow{\alpha}_{s} \sigma s_i'$

- $\alpha = \mathbf{read}(o, (\mathrm{v}, n))$: Since σ_i is available, there exists $\alpha' = \mathbf{read}(o, (\mathrm{v}', n'))$, s.t. $\sigma_i \xrightarrow{\alpha'}_{\ell} \sigma_i'$. Since $\sigma_i \precsim_{\mathsf{ws}} \sigma s_i$, it holds that $\sigma s_i \xrightarrow{\alpha'}_{s} \sigma s_i''$ 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. $\sigma s_i \xrightarrow{\mathbf{read}(o,(\mathrm{v}',n))}_{s} \sigma s_i''$. 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, $\sigma_i \xrightarrow{\alpha}_{\ell} \sigma_i'$. Since, $\sigma_i \precsim_{\mathsf{ws}} \sigma s_i$, there exists $\sigma s_i''$ s.t. $\sigma s_i \xrightarrow{\alpha}_{s} \sigma s_i''$ and $\sigma_i' \precsim_{\mathsf{ws}} \sigma s_i''$. Since σs_i is deterministic, $\sigma s_i'' = \sigma s_i'$. Therefore, $(\sigma s_i', \sigma_i') \in S$. □

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

Definition 19 A program P supports the consistency level $[\ell]_{\cong}$ iff $[P]^{\ell}_{\sigma} \precsim [P]^{\ell'}_{\sigma'}$ for all $\sigma \ {}_{\ell}\!\sim_{\ell'} \sigma'$ and $\ell' \prec \ell$.

The above definition states that a program P supports a particular consistency level $[\ell]_{\cong}$ when the traces of P running against ℓ are at most the traces that can obtained when running P against any store ℓ' providing stronger consistency property. We remark that the above definition implies to consider just the minimal ℓ' s.t. $\ell' \prec \ell$. In fact, for all other ℓ'' s.t. $\ell' \prec \ell'' \prec \ell$ it holds that $[P]^{\ell'}_{\sigma'} \precsim [P]^{\ell''}_{\sigma''}$ for all P, $\sigma' \ {}_{\ell'}\!\sim_{\ell''} \sigma''$ and therefore $[P]^{\ell}_{\sigma} \precsim [P]^{\ell'}_{\sigma'}$ implies $[P]^{\ell}_{\sigma} \precsim [P]^{\ell''}_{\sigma''}$. Thanks to Lemma 8, we need just to compare a particular consistency level ℓ against the strong store in § 3.

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

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

Consider the state σ s.t. $\sigma(o) = (\mathrm{v}_0, \mathrm{n}_1)$ and $\sigma(o') = (\bot, \mathrm{n}_0)$ for $o' \neq o$. Note that σ can be taken as state of both s and w. Moreover, it is immediate to show that $\sigma \ {}_{s}\!\sim_{w} \sigma$.

When running P over the strong store s defined in § 3.3, the only possible execution trace is the following

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

Differently, the program P produces two additional traces over w

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

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: $P' \triangleq \nu x(P'_P||P'_C)$

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

Now, it is easy to check that $[P']^{w}_{\sigma} \precsim_{\mathsf{ws}} [P']^{s}_{\sigma}$ (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 both in the weak and in strong memory model. This is not the case if we execute P' in the asynchronous weak store a in Figure 2, since $[P']^{a}_{\sigma}$ allows trace $\xrightarrow{\overline{show}(o,(\mathrm{v}_0,\mathrm{n}_1))}$ which is not allowed by $[P']^{s}_{\sigma}$ (i.e., $[P']^{a}_{\sigma} \not\precsim_{\mathsf{ws}} [P']^{s}_{\sigma}$). Example 12 shows a process that, instead, supports the consistency level provided by a.

Example 12 Consider the following variant of the program P' in Example 11 where the keys in the store have values of the form $(\mathrm{v}, \mathtt{T})$, that is pairs of values and their sorts. For instance, the store could be used for video-sharing, with v being a video file and its sort $\mathtt{T}$ being *"cooking a cherry pie"*.

$$P'' \triangleq \nu x(\overline{x}||P''_P||P''_C)$$
$$P''_P \triangleq !\mathbf{write}(o, (\texttt{newvideo}(), \mathtt{T}))$$
$$P''_C \triangleq !x.\mathbf{read}(o, (v', v''));$$
$$\texttt{if } v'' = \mathtt{T} \texttt{ then } \overline{show}(o, v'') \texttt{ else } \overline{x}$$

Process P''_P models a user repeatedly uploading videos of type $\mathtt{T}$ (assuming that $\texttt{newvideo}()$ is a function locally implemented by P''_P that returns the next video of type $\mathtt{T}$ to upload), whereas process P''_C is a user interested in downloading any file of type $\mathtt{T}$. By making only v'' observable, in $\overline{show}(o, v'')$, we model the fact that P''_C is interested in the type of the downloaded video (i.e., *"cooking a cherry pie"*) and not in a specific video. One can easily check that both $[P'']^{a}_{\sigma}$ and $[P'']^{s}_{\sigma}$ exhibit traces with the following shape $\xrightarrow{\tau}{}^{*} \xrightarrow{\overline{show}(o,\mathtt{T})}$. The fact that the asynchronous store eventually reaches consistency ensures that a message of type $\mathtt{T}$ will eventually be read by P''_C, which yields $[P'']^{a}_{\sigma} \precsim_{\mathsf{ws}} [P'']^{s}_{\sigma}$.

In this section we have given a few examples of processes that can be safely run using weak stores. Example 11 illustrates how one can guarantee strong consistency properties by introducing synchronisation actions in the processes. This approach is similar to applying memory fences (i.e., barriers located in the code) when using weak memory models [4]. Example 12 underlines that, instead, in some other scenarios strong consistency can be guaranteed by relaxing or adapting the notion of observation.

5 Compositionality and Applicability of the Framework

Consider again the process $P_P||P_C$ from Example 10, which does not support consistency level w (as shown in § 3.3). Noticeably, whereas P_P alone supports consistency level w as it always produces the traces $\xrightarrow{\tau}\xrightarrow{\overline{x}}$, the process P_C does not support consistency level w. In fact, executing P_C in w will potentially introduce additional traces w.r.t. s as the weak store can return different values (when replicas still need to synchronise in order to resolve inconsistencies). These additional traces are produced even without the presence of

the writer P_P (executing in parallel with P_C over the store) as Definition 19 universally quantifies over all the possible contents of the store (hence, it implicitly includes the store obtained after the write operation of P_P).

Consider now $P'_P||P'_C$ from Example 11, which supports consistency level w. Adding another writer such as P'_P or P_P in parallel to $P'_P||P'_C$ does not change the consistency level of the whole system, e.g., both $P'_P||P'_C||P'_P$ and $P'_P||P'_C||P_P$ support consistency level w.

In general, the consistency level supported by a process P is not affected by other processes running in parallel (if we rely on a behavioural preorder that is a precongruence with respect to parallel composition). This holds *as long as* the added process supports at least the same consistency levels supported by P (e.g., $P'_P||P'_C||P_C$ does not support w as P_C does not).

As a second observation, if we add a parallel process that interferes with P and adds visible actions, this interference would be observable both using store w and s.

Theorem 1 *If P and Q support consistency level ℓ defined in terms of the preorder $\precsim_{\mathsf{ws}}$, then $P||Q$ supports consistency level ℓ.*

Proof The proof follows by showing that the following relation is a simulation

$$\mathcal{R} = \{([P||Q]^{\ell_1}_{\sigma_1}, [P||Q]^{\ell_2}_{\sigma_2}) \mid \ell_2 \prec \ell_1,\ \sigma_1\ {}_{\ell_1}{\sim_{\ell_2}}\ \sigma_2,\ P \textit{ supports } \ell_1, Q \textit{ supports } \ell_1\}$$

We proceed by case analysis on the last applied rule in the derivation $[P||Q]^{\ell_1}_{\sigma_1} \xrightarrow{\mu} S$

- [STORE-INT] Therefore $\sigma_1 \xrightarrow{\alpha}_{\ell_1} \sigma'_1$, $P||Q \xrightarrow{\alpha} R$ and $S = [R]^{\ell_1}_{\sigma'_1}$. By rule analysis, we conclude that either (i) $P \xrightarrow{\alpha} P'$ and $R = P'||Q$ or (ii) $Q \xrightarrow{\alpha} Q'$ and $R = P||Q'$. Case (i) is as follows: Since $\ell_2 \prec \ell_1$, $\sigma_1\ {}_{\ell_1}{\sim_{\ell_2}}\ \sigma_2$, we have $\sigma_1 \xrightarrow{\alpha}_{\ell_1} \sigma'_1$ implies $\sigma_2 \xrightarrow{\alpha}_{\ell_2} \sigma'_2$. Moreover, $\sigma'_1\ {}_{\ell_1}{\sim_{\ell_2}}\ \sigma'_2$
  (because σ'_1 is reachable from σ_1 with action α and σ'_2 is reachable from σ_2 with the same action). From $P \xrightarrow{\alpha} P'$ and $\sigma_2 \xrightarrow{\alpha}_{\ell_2} \sigma'_2$, we conclude

$$[P||Q]^{\ell_2}_{\sigma_2} \xrightarrow{\mu} [P'||Q]^{\ell_2}_{\sigma'_2} = S'$$

  , and $(S, S') \in \mathcal{R}$. Case (ii) follows analogously.
- [PROC] Follows straightforwardly because the store is not involved in the transition.
- [SYNC] Therefore $\sigma_1 \xrightarrow{\alpha}_{\ell_1} \sigma'_1$, $\alpha \in \mathcal{S}$, $S = [P||Q]^{\ell_1}_{\sigma'_1}$. Then, $\sigma_1\ {}_{\ell_1}{\sim_{\ell_2}}\ \sigma_2$ implies $\sigma'_1\ {}_{\ell_1}{\sim_{\ell_2}}\ \sigma_2$. Therefore $(S, [P||Q]^{\ell_1}_{\sigma_2}) \in \mathcal{R}$. □

6 Related Work

There is an extensive literature on hardware weak memory models (e.g., TSO-x86 [24] and POWER [17]) and the properties that can be ensured, for instance, via reordering of memory operations made by the processor. Within this research thread, [22, 23] focus on the correctness of x86 assembly code via verified compilers, [20] presents a proof system for x86 assembly code based on a rely-guarantee assertion method, [17] proposes an axiomatic model for the memory model of POWER multiprocessors, equivalent to the operational specification given in [21], and illustrates how it can be used for verification using a SAT constraint solver. A general account of weak memory models is given in [4], together with a mechanism based on memory fences that preserve properties such as sequential consistency, and an automatic generation of tests for processors implementations. With respect to the work on hardware memory models, we consider a higher level of abstraction, focusing on consistency of replicated cloud stores.

Existing work on memory models also addresses higher levels of abstraction. For instance, [14] provides a static typing framework to reason on weak memory models in Java, and in particular on the property *happens-before*.

A thread of research investigates criteria and data types for the correct implementation of eventually consistent storages. The work in [11] studies a similar notion of store to the one we consider; it defines sufficient rules for the correct implementation of transactions in a server using revision diagrams. [9] proposes data types to ensure eventual consistency over cloud systems. [10] specifies replicated data types using relations over events and proposes a framework for verifying store implementations. On another line of research, the work in [7] focuses on criteria for decidable checking of eventual consistency of systems.

The main difference with this work is in the aim of our paper: albeit we provide a characterisation of weak and strong storages, our aim is not to ensure that a store (or a system) provides 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 on the specific applications.

A similar approach to ours is followed in [8]: a characterisation of weak stores is given together with a parametric operational semantics for Core ML (an imperative call-by-value λ-calculus). In [8], the focus is on ensuring properties such as race-freedom on shared memory with buffers using a precise operational characterisation of weak store. The focus of our work is rather on the interplay of asynchronous communication of processes and usage of cloud storages with several degrees of consistency.

7 Conclusions and future work

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 of stores at this abstract level and that we can relate these specifications with concrete operational implementations in terms of LTS. We also illustrate how to provide a more fine grained specification of properties, e.g., eventual consistency. For simplicity, we just consider eventual consistency under the assumption of total ordering of events. A more general model of computation in which actions are just partially ordered could be more naturally represented by substituting traces with partially ordered sets of actions. We leave this extension, and the formalisation of weaker models of consistency, such as the Revision Diagrams studied in [11, 29], as a future work. 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 [12], 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 correctness or programs running over a particular store. Basically, the classification can be used to show that an application supports weaker consistency levels; this is done by showing that the observable behaviour of the application is the same as when this application interacts with a strongly consistent store.

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. Namely, 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 [15] where architectures are built to interface applications with weak stores to provide properties not originally satisfied by these stores. Our framework is aimed at allowing the formal analysis of such proposals by exploiting the compositional reasoning enabled by process calculi standard notions. More precisely, one can assume having a taxonomy of stores (e.g., ordered according to $\prec$ in 4.3). The proposed framework can be applied to the problem of verifying a specific application P, for which a given known level of consistency ℓ is required. In case level of consistency ℓ is not supported by P but a weaker one is supported, say ℓ', two interesting problems arise. The first one is whether we can automatically derive an interface P' such that $P||P'$ supports level of consistency ℓ. The second one is whether, given a taxonomy of stores, one can create a set of reusable services $P'_{(\ell,\ell')}$ that 'fix' the level of consistency of ℓ' and simulate the level of consistency ℓ, i.e., such that for all P supporting consistency level ℓ', $P||P'_{(\ell,\ell')}$ supports consistency level ℓ.

References

1. Alpha architecture reference manual, 4th edn, 2002.
2. Ieee 1588 precision time protocol (ptp) version 2, 2008.
3. J. Alglave. A formal hierarchy of weak memory models. *Form. Methods Syst. Des.*, 41(2):178–210, Oct. 2012.
4. 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.
5. P. Bailis and A. Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013.
6. L. Bocchi and H. C. Melgratti. On the behaviour of general-purpose applications on cloud storages. In E. Tuosto and C. Ouyang, editors, *WS-FM*, volume 8379 of *Lecture Notes in Computer Science*, pages 29–47. Springer, 2013.
7. A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 285–296. ACM, 2014.
8. 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.
9. 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.
10. S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 271–284. ACM, 2014.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.

17. S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for power multiprocessors. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2012.
18. R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
19. D. Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.
20. 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.
21. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 175–186. ACM, 2011.
22. 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.
23. 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.
24. 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.
25. 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.
26. C. SPARC International, Inc. The sparc architecture manual: Version 8 and 9, 1992.
27. A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education, 2007.
28. W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.
29. K. von Gleissenthall and A. Rybalchenko. An epistemic perspective on consistency of concurrent computations. *CoRR*, abs/1305.2295, 2013.