Formal modelling and analysis of distributed storage systems

Jordan de La Houssaye, Franck Pommereau, Philippe Deniel

Formal modelling and analysis of distributed storage systems

Jordan de la Houssaye[1,2], Franck Pommereau[2], and Philippe Deniel[1]

[1] CEA, DAM, DIF, F-91297, Arpajon, France
jordan.delahoussaye@cea.fr, philippe.deniel@cea.fr
[2] IBISC, Univ. Paris-Saclay/Évry, IBGBI, 23 bd de France, 91037 Évry, France
jordan.delahoussaye@ibisc.fr, franck.pommereau@ibisc.fr

Abstract. Distributed storage systems are nowadays ubiquitous, often under the form of multiple caches forming a hierarchy. A large amount of work has been dedicated to design, implement and optimise such systems. However, there exists to the best of our knowledge no attempt to use formal modelling and analysis in this field. This paper proposes a formal modelling framework to design distributed storage systems while separating the various concerns they involve like data-model, operations, placement, consistency, topology, etc. A system modelled in such a way can be analysed through model-checking to prove correctness properties, or through simulation to measure timed performance. In this paper, we define the modelling framework and then focus on timing analysis. We illustrate these two aspects on a simple example showing that our proposal has the potential to be used to make design decisions before the real system is implemented.

1 Introduction

Nowadays technologies make intensive use of distributed storage systems. A particular and prominent form of such systems are caches. They can be found in many places embedded in almost every piece of hardware or software system that involves information storage at some point. This results in overwhelmingly complex systems in which we cannot even be sure that caches actually improve performance. One reason for this situation is the lack of tools to analyse such systems during their designing stages; in particular, to the best of our knowledge, there exist no attempt to apply formal modelling and analysis to such systems. Our main contribution in this paper is thus to propose a modelling framework that can be applied to design distributed storage systems. Moreover, the overall performance depends on a very large number of intricate aspects that cannot easily be considered separately from each other. An important part of our contribution is to separate various concerns and to link them explicitly:

- a generic data model is defined, allowing to consider a variety of operations applicable to data;
- a topology is defined independently, describing how states are arranged in the distributed system and how its nodes (see section 2.3) communicate;

- policy-related questions like placement strategy, hierarchical organisation of nodes, collaboration between nodes, interpretation of the distributed state as a global state, etc., can be considered separately;
- properties like data consistency (*e.g.*, cache coherence), correctness and termination of operations, deadlock-freeness, worst/best/mean-time execution, etc., can be studied separately on the modelled systems.

In the next section, we define the static aspects of the framework, including: data model, operations on data, topology model, communication between nodes, interpretation of the distributed data as a global state, placement policy and consistency properties. This is enriched in section 3 with a notion of actors that generate activity yielding executions described by so called timelines; the rest of the section describes the various kind of analysis that can be performed on modelled systems. Section 4 illustrates these aspects on a toy three-level cache hierarchy and shows how it can be modelled and how its performance can be analysed. The last section concludes and gives perspectives, together with a survey of related work. Additional details can be found in a technical report [7].

2 Static aspects of the modelling framework

From a static point of view, a model consists of three aspects:

- a data model that defines states and operations on them;
- a topology model that defines a notion of nodes storing data, together with a communication model between nodes. This leads to a notion of interpretation of a distributed state into a global state;
- a placement policy that decides how to manage the storage on nodes and where each piece of state has to be located.

2.1 Data model

We consider three pairwise disjoint nonempty sets: K is the set of *keys* that can be thought as addresses; V is the set of *values* stored at the addresses; L is the set of *labels* used to relate keys. For instance, for a Unix file system, K would be the inodes addresses, V their content and L could model relations like directory membership. For a memory model, V would hold all the possible memory blocks whose addresses would be in K, and L would be unused.

Definition 1. *Let* $H_{K,V} \stackrel{\text{df}}{=} 2^{K\times V}$ *and* $R_{K,L} \stackrel{\text{df}}{=} 2^{L\times 2^K\times 2^K}$. *A* reduced state σ *is a pair* $(\sigma.h, \sigma.r)$ *such that* $\sigma.h \in H_{K,V}$ *and* $\sigma.r \in R_{K,L}$. *We note by* $\Sigma^*_{K,V,L}$ *the set of all reduced states, and define* $\mathsf{dom}(\sigma.h) \stackrel{\text{df}}{=} \{k \mid \exists v \in V, (k,v) \in \sigma.k\}$. *A reduced state* $\sigma \in \Sigma^*_{K,V,L}$ *is* well-formed *iff it satisfies the following conditions:*

- $\sigma.h$ *is a map:* $\forall k \in \mathsf{dom}(\sigma.h)$, $|\{(k,v) \in \sigma.h\}| = 1$;
- *all the keys in* $\sigma.r$ *are mapped by* $\sigma.h$*:* $\bigcup_{(l,K1,K2)\in\sigma.r} K1 \cup K2 \subseteq \mathsf{dom}(\sigma.h)$.

We define a partial order $\preceq^*$ *on* $\Sigma^*_{K,V,L}$ *by* $\sigma_a \preceq^* \sigma_b \Leftrightarrow \sigma_a.h \subseteq \sigma_b.h \wedge \sigma_a.r \subseteq \sigma_b.r$.

Intuitively, a reduced state is a map from related keys to the corresponding data. For instance, consider an extremely simplified file-system containing the following objects: the root directory "/", sub-directories "`/bin`", "`/usr`" with nested sub-directory "`/usr/bin`", and files "`/bin/sh`" and "`/usr/bin/sh`". These objects could be represented in a state σ as follows:

$$\sigma.h \stackrel{\text{df}}{=} \{(0,/),\ (1,\texttt{bin}),\ (2,\texttt{usr}),\ (3,\texttt{sh}),\ (4,\texttt{bin}),\ (5,\texttt{sh})\ \}$$
$$\sigma.r \stackrel{\text{df}}{=} \{(\mathsf{root},\{0\},\emptyset),(\mathsf{dir},\{0\},\{1\}),\ (\mathsf{dir},\{0\},\{2\}),\ (\mathsf{dir},\{2\},\{4\}),\ (\mathsf{file},\{1\},\{3\}),\ (\mathsf{file},\{4\},\{5\})\}$$

where $\sigma.h$ stores identifiers of the file-system objects associating them to their names, and $\sigma.r$ stores links between the objects, allowing to identify a root directory (root label) and the children of each directory, which may be directories themselves (dir label) or files (file label).

Definition 2. *A* (complete) state *is a triple* $\sigma \stackrel{\text{df}}{=} (\sigma.base, \sigma.plus, \sigma.minus)$ *in* $\Sigma_{K,V,L} \stackrel{\text{df}}{=} (\Sigma^*_{K,V,L})^3$. *Such a state is* well-formed *iff* $\sigma.base$ *is well-formed,* $\sigma.plus \preceq^* \sigma.base$ *and* $\sigma.minus \cap \sigma.base = \emptyset$. *We define a partial order* $\preceq$ *on* $\Sigma_{K,V,L}$ *as the component-wise extension of* $\preceq^*$, i.e., $\sigma_a \preceq \sigma_b \Leftrightarrow \sigma_a.base \preceq^* \sigma_b.base \wedge \sigma_a.plus \preceq^* \sigma_b.plus \wedge \sigma_a.minus \preceq^* \sigma_b.minus$.

A state $(\sigma.base, \sigma.plus, \sigma.minus)$ can be understood as a reduced state with a history, *i.e.*, $\sigma.base$ is the result of adding $\sigma.plus$ to and removing $\sigma.minus$ from an original state $\sigma_0.base$. This shall be depicted as:

$$\begin{pmatrix} \{\,\sigma.base.h\,\} \ ^+\{\,\sigma.plus.h\,\} \ ^-\{\,\sigma.minus.h\,\} \\ \{\,\sigma.base.r\,\} \ ^+\{\,\sigma.plus.r\,\} \ ^-\{\,\sigma.minus.r\,\} \end{pmatrix}$$

Historicized states are needed to model distributed storage. Consider indeed a simple case where a cache lies between a process and a storage. If the process requests to delete the resource associated to a key k, this may be made in the cache only. Just dropping the information associated to k in the cache is not correct. Indeed, by definition, the cache holds only a subset of the information that the storage holds. The absence of k in the cache is thus not a sufficient information to know that k has to be deleted in the storage too, it may as well mean that k has never been stored in the cache. Moreover, if later k is allocated again, the cache may store the new value associated to k and forget about the fact that k has been deleted previously. So, a history enables a cache for actually hiding operations to the storage, which is crucial for a cache.

Reduced and complete states are equipped with various compositions and operations. For $\sigma^*_a, \sigma^*_b \in \Sigma^*_{K,V,L}$, we define union ($\cup$), intersection ($\cap$) and difference ($\setminus$) as simple component-wise extensions of their sets counterparts. For instance, we have $\sigma^*_a \cup \sigma^*_b \stackrel{\text{df}}{=} (\sigma^*_a.h \cup \sigma^*_b.h,\ \sigma^*_a.r \cup \sigma^*_b.r)$. Moreover, for $\sigma_a, \sigma_b \in \Sigma_{K,V,L}$, these operations are further extended component-wise. For instance, we have $\sigma_a \cup \sigma_b \stackrel{\text{df}}{=} (\sigma_a.base \cup \sigma_b.base,\ \sigma_a.plus \cup \sigma_b.plus,\ \sigma_a.minus \cup \sigma_b.minus)$. For $k \in K$, $\sigma^* \in \Sigma^*_{K,V,L}$, we note by $\sigma^* \setminus k$ the restriction of σ^* in which we removed any element involving k; this notation is extended component-wise to a complete state. We finally define projection ($\gg$) as follows:

$$\sigma_a \gg \sigma_b \stackrel{\text{df}}{=} \Big(\big((\sigma_a.base \cup \sigma_b.base) \setminus \sigma_a.minus \big) \cup \sigma_a.plus,\\ \big((\sigma_a.plus \setminus \sigma_b.minus) \cup \sigma_b.plus \big) \setminus \sigma_a.minus,\\ \big((\sigma_a.minus \setminus \sigma_b.plus) \cup \sigma_b.minus \big) \setminus \sigma_a.plus \Big)$$

This definition is specially designed to provide with the following properties:

- $\sigma_\emptyset \stackrel{\text{df}}{=} ((\emptyset,\emptyset), (\emptyset,\emptyset), (\emptyset,\emptyset))$ is neutral: for any $\sigma \in \Sigma_{K,V,L}$ that is well defined we have $\sigma_0 \gg \sigma = \sigma$ and $\sigma \gg \sigma_0 = \sigma$;
- intermediate changes that cancel each other are hidden: consider for example an initially empty state $\sigma_\emptyset$ as above on which we perform a series of updates
  - add a: $((\emptyset,\emptyset), a, (\emptyset,\emptyset)) \gg \sigma_\emptyset = (a, a, (\emptyset,\emptyset)) \stackrel{\text{df}}{=} \sigma_1$,
  - drop a to add b instead: $((\emptyset,\emptyset), b, a) \gg \sigma_1 = (b, b, a) \stackrel{\text{df}}{=} \sigma_2$,
  - finally drop b to add c instead: $((\emptyset,\emptyset), c, b) \gg \sigma_2 = (c, c, a)$ in which b has disappeared like we had dropped a to add c directly from σ_1;
- similarly, if as above we start from σ_1 then drop a to add b instead, we get σ_2; then if we now drop b to add a back, we compute $(a, a, b) \gg (b, b, a) = (a, (\emptyset,\emptyset), (\emptyset,\emptyset))$ which hides the mutually cancelling operations.

Now, consider again our example of a simplified file system and consider an initial state where only "`/`" and "`/bin`" exist. The creation of "`/usr`" can be represented a projection as follows:

$$\begin{pmatrix} \{\ (0,/), (1,\texttt{bin})\ \} & {}^{+}\{\} & {}^{-}\{\} \\ \{\ (\mathsf{root},\{0\},\emptyset), (\mathsf{dir},\{0\},\{1\})\ \} & {}^{+}\{\} & {}^{-}\{\} \end{pmatrix} \ll \begin{pmatrix} \{\} & {}^{+}\{\ (2,\texttt{usr})\ \} & {}^{-}\{\} \\ \{\} & {}^{+}\{\ (\mathsf{dir},\{0\},\{2\})\ \} & {}^{-}\{\} \end{pmatrix}$$
$$= \begin{pmatrix} \{\ (0,/), (1,\texttt{bin}), (2,\texttt{usr})\ \} & {}^{+}\{\ (2,\texttt{usr})\ \} & {}^{-}\{\} \\ \{\ (\mathsf{root},\{0\},\emptyset), (\mathsf{dir},\{0\},\{1\}), (\mathsf{dir},\{0\},\{2\})\ \} & {}^{+}\{\ (\mathsf{dir},\{0\},\{2\})\ \} & {}^{-}\{\} \end{pmatrix}$$

2.2 Operations

An operation is a request a user of the storage system might perform and is part of a system definition. We assume that any operation has an effect (possibly neutral), provided as a parametrised complete state, and a result. To apply an operation, one provides a valuation of the input parameters, then the result is a valuation of the output parameters. If no output parameters can be found, the operation fails. Otherwise, the effect is computed from the parametrised state evaluated using to the input and output parameters values. A mapping from variables to values is called a *binding* and usually noted by β, possibly with subscripts or superscripts. We note by $\mathsf{keys}(\beta) \stackrel{\text{df}}{=} \mathsf{img}(\beta) \cap K$ the set of keys referenced in β, where img is the image (or codomain) of the binding.

Definition 3. *Let* $\mathsf{vars}(e)$ *be the set of variables involved in an expression* e*. An operation is a 4-tuple* $op \stackrel{\text{df}}{=} (op.name, op.guard, op.effect, op.params)$ *such that:*

- *$op.name$ is a name used to refer to the operation (any string);*

- *op.guard is a Boolean expression that guards the application;*
- *op.effect is a an expression that may be evaluated to a complete state;*
- *op.params is a set of variables such that* $op.params \subseteq \mathsf{vars}(op.guard) \cup \mathsf{vars}(op.effect) \stackrel{\text{df}}{=} \mathsf{vars}(op)$*;*
- *we have* $\mathsf{vars}(op.effect) \subseteq op.params \cup \mathsf{vars}(op.guard)$*;*
- *there exists at least one binding such that op.effect and op.guard can be actually evaluated (*i.e.*, both are actually computable).*

We note by $\mathcal{OPS}$ *the set of all defined operations.*

The role of the guard is to prevent operations to be applied on incompatible states (*e.g.*, one cannot read from an unallocated address). Thus the guard is always evaluated on the state the operation is meant to be applied for a given valuation of the input parameters. Then, if the guard is true and output parameters can be computed, the effect is evaluated and projected onto the state. Given an operation *op*, we note by:

- $\mathcal{B}_{op,K,V,L}$ the set of all bindings $\beta : \mathsf{vars}(op) \to K \cup V \cup L$;
- $\mathcal{B}^{in}_{op,K,V,L}$ the set of all bindings $\beta : op.params \to K \cup V \cup L$;
- $\mathcal{B}^{out}_{op,K,V,L}$ the set of all bindings $\beta : \mathsf{vars}(op) \setminus op.params \to K \cup V \cup L$.

For two bindings $\beta_a, \beta_b \in \mathcal{B}_{op,K,V,L}$ such that $\mathsf{dom}(\beta_a) \cap \mathsf{dom}(\beta_b) = \emptyset$, we define their composition $\beta \stackrel{\text{df}}{=} \beta_a + \beta_b : \mathsf{dom}(\beta_a) \cup \mathsf{dom}(\beta_b) \to K \cup V \cup L$ as follows:

$$\forall x \in \mathsf{dom}(\beta), \beta(x) \stackrel{\text{df}}{=} \begin{cases} \beta_a(x) & \text{if } x \in \mathsf{dom}(\beta_a), \\ \beta_b(x) & \text{otherwise, } \textit{i.e.}\text{, if } x \in \mathsf{dom}(\beta_b) \end{cases}$$

For convenience, we introduce some more notations. Let $\sigma_{in} \in \Sigma_{K,V,L}$, $op \in \mathcal{OPS}$, and $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$, we define:

- $op.guard(\sigma_{in}, \beta)$ is the evaluation of *op.guard* through $\beta + \{\sigma \to \sigma_{in}\}$, where σ refers to the input state and can be used to access it from the guard;
- $op.effect(\beta)$ is the evaluation of *op.effect* through a binding β;
- $op.candidates(\sigma_{in}, \beta_{in}) \stackrel{\text{df}}{=} \{\beta_{out} \in \mathcal{B}^{out}_{op,K,V,L} \mid op.guard(\sigma_{in}, \beta_{in} + \beta_{out}) \wedge op.effect(\beta_{in} + \beta_{out}) \in \Sigma_{K,V,L}\}$ is the set of possible output bindings that, combined with β_{in}, allow to validate the guard and to evaluate the effect to an actual complete state;
- *op* is called *elligible* for σ_{in} and β_{in} iff $op.candidates(\sigma_{in}, \beta_{in}) \neq \emptyset$;
- *op* is called *deterministic* iff for all $\sigma_{in} \in \Sigma_{K,V,L}$ and all $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$ we have $|op.candidates(\sigma_{in}, \beta_{in})| \leq 1$.

Then, when *op* is elligible for some input state and input binding, the set of output states and output bindings is computed by applying *op* with every possible candidate binding, which is made using a projection as follows.

Definition 4. *The* application *of operation* $op \in \mathcal{OPS}$ *onto input state* $\sigma_{in} \in \Sigma_{K,V,L}$ *given an input binding* $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$ *results in the subset of* $\mathcal{B}^{out}_{op,K,V,L} \times \Sigma_{K,V,L}$ *defined by* $op(\sigma_{in}, \beta_{in}) \stackrel{\text{df}}{=} \{(\beta_{out}, op.effect(\beta_{in} + \beta_{out}) \gg \sigma_{in}) \mid \beta_{out} \in op.candidates(\sigma_{in}, \beta_{in})\}$.

2.3 Topology

A distributed storage consists of a set of *nodes* that store (local) states and communicate through *buses*. This is formalised as an hypergraph as follows.

Definition 5. *Let N be a set of* nodes*, a* topology *T on N is a pair $T \stackrel{\text{df}}{=} (T.nodes, T.buses)$ where $T.nodes \stackrel{\text{df}}{=} N$ is the set of nodes and $T.buses \subseteq 2^N \setminus \emptyset$ is the set of hyperedges. For $i, j \in T.nodes$, we note by $T[i,j]$ the fact that there exists $b \in T.buses$ such that $\{i,j\} \subseteq b$.*

Given a topology T, nodes in $T.nodes$ are allowed to communicate by exchanging *frames* over the buses in $T.buses$. We assume that a bus can transmit only one message at a time, *i.e.*, a sender is blocked until a previously sent message has been received. Moreover, a receiver is blocked until a message is sent for it. The possible frames are defined in figure 1. Each frame is a 4-tuple holding the bus on which the communication is made, the sender and recipient nodes identities, and the message itself. Message can be of four types:

`sync` this type of message transmits a ⟨`request`⟩. It is synchronous in that there can be no further message between source and destination until the destination has responded with a `return` message holding the expected ⟨`response`⟩;

`async` this type of message transmits a ⟨`request`⟩. It is asynchronous in that it only blocks the sender until the destination has responded with a `wait` message, but the actual ⟨`response`⟩ will come later;

`wait` this is a response to an `async` message, which comes with a *handler* (a unique identifier) so that the receiver will be able to link its request with the response that will be provided later. We assume that $\mathcal{H}$ is a set that is large enough (*e.g.*, infinite) to assign a unique handler for every `wait` message;

`return` this type of message transmits a ⟨`response`⟩ to a ⟨`request`⟩. A response to a `sync` message comes as a pair (`return`, ⟨`response`⟩); a response to an `async` message comes as a triplet (`return`, *handler*, ⟨`response`⟩), where *handler* is the identifier that was provided with the `wait` response.

There is currently only one type of ⟨`request`⟩, but this may change if needed. A request $req \stackrel{\text{df}}{=} (\texttt{operate}, op, \beta)$ is parametrised by an operation $req.op$ and an input binding $req.\beta$ for this operation. The corresponding answer, sent synchronously or asynchronously, is a ⟨`response`⟩ that can be a `success` or a `failure`. In the former case, it comes with the output binding (noted $resp.\beta$) chosen by the system; in the latter case, it comes with a failure message.

$$
\begin{aligned}
\langle\texttt{frame}\rangle_T &::= (bus, source, destination, \langle\texttt{message}\rangle)\\
\langle\texttt{message}\rangle &::= (\texttt{sync}, \langle\texttt{request}\rangle) \mid (\texttt{async}, \langle\texttt{request}\rangle)\\
&\quad \mid (\texttt{wait}, handler) \mid (\texttt{return}, \langle\texttt{response}\rangle) \mid (\texttt{return}, handler, \langle\texttt{response}\rangle)\\
\langle\texttt{request}\rangle &::= (\texttt{operate}, op, \beta_{in})\\
\langle\texttt{response}\rangle &::= (\texttt{success}, \beta_{out}) \mid (\texttt{failure}, text)
\end{aligned}
$$

Fig. 1. The frames exchanged between the nodes of a topology T, where $bus \in T.buses$, $source, destination \in T.nodes$, $handler \in \mathcal{H}$ ($\mathcal{H}$ is a set of identifiers), $op \in \mathcal{OPS}$, $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$, $\beta_{out} \in \mathcal{B}^{out}_{op,K,V,L}$ and *text* is a text string. Special typesetting denotes ⟨`non terminals`⟩ and `symbols` (*i.e.*, constants).

Interpretations and integration. As soon as states are distributed over a topology, we need to define how to compose these local states into a unique global state. This must be user-defined together with the topology. Moreover we must define how a node integrates the information about states it can deduce from its exchanges with other nodes. For instance, consider a memory hierarchy with a cache that receives a request to read a block a. If it forwards the request to the next level in the hierarchy and eventually receives the value v in the response, it knows that (a, v) could be added to its local state. More generally, because of the way operations are defined, knowing the operation together with the input and output bindings is enough to evaluate *op.effect*. The latter is a state that may be composed with the local state. How this composition must be made (or avoided) is dependent on how the distributed state is interpreted and must be user-defined as well.

Definition 6. *An* interpretation I_T *of topology* T *is a pair of functions:*

$$I_T \stackrel{\text{df}}{=} \big(\ \mathit{globalview} : T.\mathit{nodes} \times \Sigma_{K,V,L} \ \to \Sigma_{K,V,L}\ ,\\ \mathit{integrate} :\ \ T.\mathit{nodes} \times T.\mathit{nodes} \to \Sigma_{K,V,L} \times \Sigma_{K,V,L} \to \Sigma_{K,V,L}\ \big)$$

In this definition, *globalview* is responsible for computing a single global state from the collection of states located on $T.nodes$. Function *integrate* is more complex: it takes a pair of nodes (a, b) and returns another function $\Sigma_{K,V,L} \times \Sigma_{K,V,L} \to \Sigma_{K,V,L}$. This one takes a pair of states (σ_a, σ_b) and combine them into a single state σ'_a that can be understood as the integration of the effect σ_b on the state σ_a, for an operation that was actually computed on node b.

When considering a hierarchy, where a process accesses a storage through a chain of caches, function *globalview* can be computed as: $\sigma_0 \gg \sigma_1 \gg \cdots \gg \sigma_n$ where the σ_i's are the locals states ordered from the one closest to the process (*i.e.*, σ_0) to the state of the storage itself (*i.e.*, σ_n).

2.4 Placement policy

The question of placement is complementary to interpretation: a node has to know on which other node the value associated with a key is located. This way it knows how to retrieve this value or to whom it has to forward a request it cannot handle itself (or does not want to). This information is provided by a *placement policy* $P^{me}_{I_T}$ that is provided by the user for an interpretation I_T. Let us assume a global variable *me* that is the node on which these methods are called, then $P^{me}_{I_T}$ is provided as a set of methods:

where $(\mathit{keys} \subseteq K, \mathit{notme} \in \{\bot, \top\}) \to T.\mathit{nodes} \cup \{\text{✘}\}$
: Returns a node where the resources referenced by *keys* should be stored, or a dummy value ✘ if no such node can be identified. If $\mathit{notme} = \top$, the return value cannot be *me*.

space $(\mathit{keys} \subseteq K, \sigma_{in} \in \Sigma_{K,V,L}) \to \mathbb{N}$
: Returns the number of resources currently stored on node *me* that need to be deleted in order to be able to store locally the values associated to *keys*.

update $(keys \subseteq K, handler \in \mathcal{H})$
: This method does not return any value but is called on node *me* whenever a request identified by *handler* has just been received. It is used to update the current knowledge about the situation that may be maintained by the policy. For instance it may update the MRU (*most recently used*) keys in a LRU (*least recently used*) cache. Notice that we see here a handler in $\mathcal{H}$ as for asynchronous requests; indeed, we will see later on that they are also used internally to the nodes for their bookkeeping.

purge $() \to K$
: Deletes and returns a resource currently stored on node *me*, which should be chosen as the one which has the least value when purge is called. For instance, a LRU cache will exactly chose the least recently used key.

close $(handler \in \mathcal{H}, outcome \in \{\texttt{success}, \texttt{failure}\})$
: This method is called to commit (on a `success`) or cancel (on a `failure`) the changes that occurred when update has been called.

Methods update and close work together: calling update allows to increase the importance of a set of keys, then calling close allows to commit or cancel the update. The reason for such a mechanism is that most operations on a node cannot be realised atomically and may require to communicate with other nodes. During this process, the node may receive and proceed other requests that can be completed locally, so we cannot rely on a mechanism that would lock the whole node during the processing of a request.

$P^{me}_{I_T}$ can be thought as a class of which each node *me* holds an instance and the above definitions are its methods. Note that update, close and purge are thus expected to have side effects on the instance.

2.5 Nodes management processes

We now describe how the nodes manage their states and communicate with others. It should be stressed that these algorithms are completely generic: the user just has to provide the elements specified above to get a working model.

At the core of each node is the *job manager*: when a $\langle\texttt{request}\rangle$ is received by a node, it is first stored in a job manager and associated to a handler in $\mathcal{H}$; it is kept here until it is fully processed. Dependencies can occur between requests: two requests r_1 and r_2 are independent if $\mathsf{keys}(r_1.\beta) \cap \mathsf{keys}(r_2.\beta) = \emptyset$. The job manager handles these dependencies and provides the following methods:

last $(key \in K) \to \mathcal{H} \cup \{\text{✘}\}$
: Returns the handler of the last request added with a domain including *key* if any, or a dummy value ✘ if no request is associated to *key*.

add $(request \in \langle\texttt{request}\rangle) \to \mathcal{H}$
: Adds *request* identified by *handler* into the manager and returns a fresh handler for it. The added request depends on the lastly added request for every key in keys(*request*).

$\mathsf{next}\,() \to \langle \texttt{request} \rangle \times \mathcal{H}$
: Returns a pair $(\mathit{request}, \mathit{handler})$ that is ready to be proceeded (no pending dependencies). The caller is blocked until such a job is actually available.

$\mathsf{deps}\,(\mathit{handler} \in \mathcal{H}) \to (\langle \texttt{request} \rangle \times \mathcal{H})^*$
: Returns the list of pairs (r, h) corresponding to all requests r and handlers h the request r_{handler} associated with *handlers* depends on. This list is ordered consistently with dependencies, the last item being $(r_{\mathit{handler}}, \mathit{handler})$, and is deterministically computed.

$\mathsf{done}\,(\mathit{handler} \in \mathcal{H})$
: Marks every information associated to *handler* as disposable and clears any disposable information that is not needed anymore.

Nodes processes are implemented as coloured Petri nets [7], however, we present them here using simpler pseudo-code. Noting by $p!$ the infinite replication of a process p, each node runs a simple process consisting of two such replications composed in parallel: $\mathsf{listener}! \parallel \mathsf{worker}!$. These processes are executed in a context with the following global variables:

- me is the node on which the process is executed;
- $\mathit{jobs}_{\mathit{me}}$ is the job manager for node me;
- T is the topology and we note by $T.\mathsf{send}\,(b, s, d, m)$ the sending of a message m on bus b from a source node s to a destination node d; the corresponding reception is noted by $T.\mathsf{receive}\,(b, s, d, m)$. Recall that $T.\mathsf{send}\,(b, \dots)$ is blocking if a message is already in transit on b and $T.\mathsf{receive}\,(b, s, d, \dots)$ is blocking until a message is actually sent on b, from s to d. Moreover, pattern matching may be used to filter the format of received messages;
- ret is a communication channel internal to the node that behaves like a bus, *i.e.*, it provides methods $\mathit{ret}.\mathsf{send}\,(m)$ and $\mathit{ret}.\mathsf{receive}\,(m)$;
- I_T and $P^{\mathit{me}}_{I_T}$ are the interpretation and the placement respectively;
- σ_{me} is the current state.

Figure 2 shows process listener that is responsible for receiving a message for the node, add it to the job manager and send back the response when it is available. It is quite a simple process, but it is worth noting how asynchronous requests are handled.

Figure 3 describes process worker that is responsible for actually executing the jobs. Essentially, it uses the placement to know if node me is responsible for the keys associated to the request and if so, it computes the effect locally if possible or forward the request to the appropriate node otherwise.

Figure 4 shows process apply that is responsible for applying on the local state σ_{me} the effect of an operation for which we have obtained the output binding. To do so, it possibly makes room in the local state if needed. For instance, a cache may drop a block if it has to store one more block but is already full.

Finally, figure 5 shows process sync that applies all the pending requests a given handler depends on. It should be stressed that a call to $\mathsf{sync}\,(h)$ also proceeds the request for h itself, as the last one. So sync returns the response for this request together with the identity of the node that actually answered it.

```
process listener is
    T.receive (bus, src, me, (kind, req))    /* receive a message (kind, req) */
    h ← jobs_me.add (req)    /* add it to the job manager and get its handler h */
    P^me_{I_T} ← P^me_{I_T}.update(keys(req.β), h)    /* notify the placement policy */
    if kind = async:    /* this is an asynchronous request */
        T.send (bus, me, src, (wait, h))    /* immediately send a wait answer */
    ret.receive (resp, h)    /* wait for the worker process to respond */
    if kind = async:
        T.send (bus, me, src, (return, h, resp))    /* send asynchronous answer */
    else:
        T.send (bus, me, src, (return, resp))    /* send synchronous answer */
```

Fig. 2. The listener process.

```
process worker is
    req, h ← jobs.next ()    /* wait until a new job is available */
    if P^me_{I_T}.where (keys(req.β), ⊥) = me:
        c ← req.op.candidates(σ, req.β)    /* search for possible β_out */
        if c ≠ ∅:
            choose β_out ∈ c    /* make a non-deterministic choice if |c| > 1 */
            resp ← (success, β_out)    /* build the response */
            apply (req, resp, h, me)    /* apply the effect to update σ */
        elif P^me_{I_T}.where (keys(req.β), ⊤) ≠ ✘:
            resp, pos ← sync (h)    /* complete all the dependencies on h and get a
            response from node pos that performed the latest operation in sync */
            if resp[0] = success:
                apply (req, resp, h, pos)    /* apply the effect to update σ */
        else:    /* we do not know how to process the request */
            resp ← (failure, "no node to handle request")
    else:    /* this forwards the request to the appropriate node */
        resp, pos ← sync (h)
    P^me_{I_T}.close (h, resp[0])    /* tell the placement about the outcome, recall that
    we have resp[0] ∈ {failure, success} */
    jobs_me.done (h)    /* tell the job manager that the request for h is done */
    ret.send (resp, h)    /* send the response back to the listener */
```

Fig. 3. The worker process.

3 Dynamic aspects of the modelling framework

To produce activity, we need to introduce dedicated nodes, called *actors*, whose only role is to send messages and receive the corresponding answers. For instance, a processor is the actor in a memory hierarchy. It is not possible to define a generic model of an actor because each one corresponds to a particular profile of activity, that will stimulate the system in its particular way. For instance, our processor at the top of a memory hierarchy could behave in many different ways depending on what kind of program it is supposed to execute. Choosing an adequate model of actor is crucial for a correct analysis. Indeed, most distributed

```
process apply (req, resp, h, pos) is
    k ← keys(req.β + resp.β)    /* get the keys involved in the operation */
    P_{I_T}^{me}.update (k, h)    /* tell the placement that these keys are currently under
    interest */
    for 1 ≤ i ≤ P_{I_T}^{me}.space (k, σ):
        least ← P_{I_T}^{me}.purge ()    /* get and drop the least value element */
        h' ← jobs_{me}.last (least)    /* get the last added request for least */
        if h' ≠ ✘:
            r, p ← sync (h')    /* flush operations h' depends on */
        σ_{me} ← σ_{me} \ least    /* restrict σ_{me} to remove least */
    integrate = I_T.integrate (me, pos)    /* get the method to integrate the effect
    in the local state */
    σ_{me} ← integrate(σ_{me}, req.op.effect(req.β + resp.β))    /* do it actually */
```

Fig. 4. The apply process.

```
process sync (handler) is
    foreach req, h ∈ jobs.deps (handler)do    /* the list order is respected! */
        pos ← P_{I_T}^{me}.where (keys(req.β), ⊤)    /* search where req should be
        processes, excluding me */
        choose b ∈ T.buses such that T[pos, me]    /* get a bus to reach pos */
        if no such b:    /* this is a bug in the placement or the topology! */
            return (failure, "no path to pos"), me
        T.send (b, me, pos, (sync, req))    /* forward req to pos */
        T.receive (b, pos, me, (return, resp))    /* wait for the response */
    return resp, pos    /* returns the latest response that is for handler */
```

Fig. 5. The sync process.

storage systems, and cache policies in particular, are based on strong hypotheses about the access patterns of the systems using them.

An actor is implemented as a Petri net that is composed with the Petri nets for the nodes processes to obtain a full system from which we can get executions of two kinds. On the one hand, the *state space*, consists of the reachable states of the Petri net linked by the transitions from one state to another. This is usually a huge object that is suitable for qualitative analysis, in particular through model-checking. On the other hand, a *trace* is a sequence of alternating states and transitions that corresponds to a path in the state space. As such, it is usually used to exhibit a faulty execution discovered using model-checking.

To enable for timed analysis of the modelled systems, and in particular performance analysis, traces are extended by applying a *cost function* that maps each transition to its duration. The resulting weighted traces can be rendered on *timelines* that represent the activity of each node, exhibiting both its busy and idle phases. Figure 6 shows a graphical representation of a timeline: time is passing from the left to the right and each horizontal line represents the activity of one node. Vertical segments depict communications. In this picture, the topology is displayed on the left. Events, *i.e.*, transitions occurrences, are depicted on the lines and ordered respecting causality, event costs are displayed within the

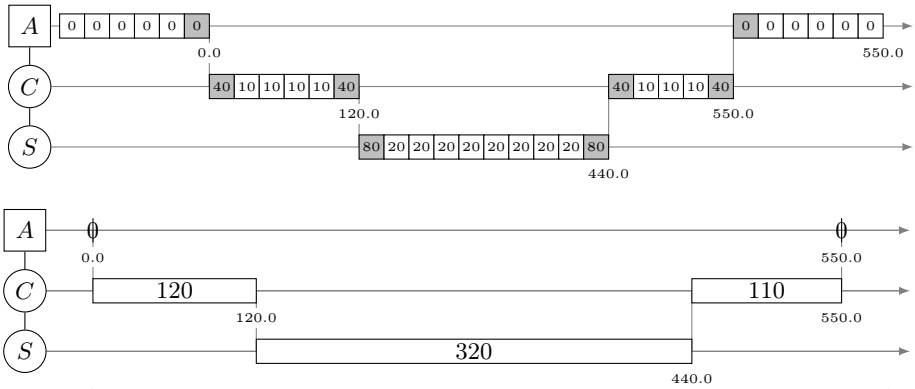

Fig. 6. At the top, a complete timeline for a system with three nodes: an actor A, a cache C and a storage S. Below, the aggregated version.

events. We model communication costs by weighting more the transitions that correspond to message sending and reception (the grey nodes in the top-most figure). If successive events between two communications are aggregated, we get a simpler but still accurate view of the nodes activity as shown at the bottom of the figure. More generally, such a graphical representation may become quickly unreadable on large systems or when complex topologies are involved. So a timeline should better be thought as a timed trace distributed onto the nodes rather than a visual representation. This representation assumes that each node is sequential and interleaves its concurrent activities. It is easy to generalise this to concurrent nodes by allowing multiple lines for each node, as many as the node can handle concurrent tasks.

4 Application example

To illustrate our framework, we propose now a model of the simple hierarchical system of figure 6: an actor A requests memory blocks to a storage S through a LRU cache C. These nodes are arranged on topology $T \stackrel{\text{df}}{=} (\{A, C, S\}, \{\{A, C\}, \{C, S\}\})$ and their initial states are:

$$\sigma_A \stackrel{\text{df}}{=} \left(\begin{smallmatrix} \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \end{smallmatrix}\right), \qquad \sigma_C \stackrel{\text{df}}{=} \left(\begin{smallmatrix} \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \end{smallmatrix}\right), \qquad \text{and} \qquad \sigma_S \stackrel{\text{df}}{=} \left(\begin{smallmatrix} \alpha & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \end{smallmatrix}\right),$$

where $\alpha \stackrel{\text{df}}{=} \{k_1 \to v_1, \ldots, k_{10} \to v_{10}\}$ is randomly generated such that σ_S is well-formed.

This system uses two operations, *read* and *write* defined as follows:

$$\mathit{read} \stackrel{\text{df}}{=} \begin{cases} \mathit{name} & \stackrel{\text{df}}{=} \texttt{``read''} \\ \mathit{guard} & \stackrel{\text{df}}{=} (k, v) \in \sigma.\mathit{base}.h \\ \mathit{effect} & \stackrel{\text{df}}{=} \left(\begin{smallmatrix} (k,v) & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \end{smallmatrix}\right) \\ \mathit{params} & \stackrel{\text{df}}{=} \{k\} \end{cases} \qquad \mathit{write} \stackrel{\text{df}}{=} \begin{cases} \mathit{name} & \stackrel{\text{df}}{=} \texttt{``write''} \\ \mathit{guard} & \stackrel{\text{df}}{=} (k, v_1) \in \sigma.\mathit{base}.h \\ \mathit{effect} & \stackrel{\text{df}}{=} \left(\begin{smallmatrix} \emptyset & (k,v2) & (k,v_1) \\ \emptyset & \emptyset & \emptyset \end{smallmatrix}\right) \\ \mathit{params} & \stackrel{\text{df}}{=} \{k, v_2\} \end{cases}$$

Operation *read* gets the value v associated to a given key k. Operation *write* replaces the value v_1 associated to key k with value v_2 also passed as argument.

We have here a hierarchical system in which state interpretation is straightforward: the global state is obtained by projecting states top-down and integration projects an observed state onto the local state (except for A that has no local state):

$$I_T \stackrel{\text{df}}{=} \Big(\textit{globalview} : \{(A,\sigma_A),(C,\sigma_C),(S,\sigma_S)\} \mapsto (\sigma_A \gg \sigma_C) \gg \sigma_S \ ,$$
$$\textit{integrate} : \ \ \textit{me}, \textit{pos} \mapsto \begin{cases} \sigma_{me}, \sigma_{pos} \mapsto \sigma_{me} & \text{if } \textit{me} = A, \\ \sigma_{me}, \sigma_{pos} \mapsto \sigma_{pos} \gg \sigma_{me} & \text{otherwise.} \end{cases}$$

The placements P_A, P_C and P_S respectively associated to the nodes A, C and S are defined as follows:

- P_A is such that every key belongs to the node C because A does not store any data. So, where constantly returns C; space constantly returns 0 (and thus, purge is never called); update and close are no-ops;
- P_C is such that every key belongs to C, moreover, it maintains a list ℓ in which new keys are positioned in MRU and the purge always deletes the key positioned in LRU. So, where constantly returns C or S if $\textit{notme} = \top$; $\mathsf{space}\,(\textit{keys}, \sigma_{in}) \stackrel{\text{df}}{=} \textit{max}(0, |\sigma_{in}.h| + |\textit{keys}| - \textit{size})$, where *size* is the size of the cache (*i.e.*, the maximum number of keys it can hold); $\mathsf{update}\,(\textit{keys}, h)$ adds $[(k,h) \mid k \in \textit{keys}]$ at the head of ℓ (MRU position); purge returns k such that (k,h) is the tail of ℓ (LRU position), which is dropped from ℓ; $\mathsf{close}\,(h, \textit{outcome})$ either drops from ℓ any pair (k,h) if $\textit{outcome} = \texttt{failure}$ or drops elements at the tail of ℓ until its has at most *size* elements;
- P_S is such that every key belongs to S, apart for this, it behaves like P_A: where constantly returns S; space constantly returns 0 (and thus, purge is never called); update and close are no-ops.

To perform a timed analysis of this system, we have considered a LRU friendly actor that sequentially sends requests (waiting for each answer before to send the next request) as follows:

- it maintains a MRU-to-LRU ordered list L of keys already sent in a request;
- a read or write is chosen with 50% probability each;
- with probability $1/a$, a key k is chosen in L, otherwise, it is chosen in $K \setminus L$;
- with probability $1/b$, the LRU key is dropped from L;
- k is added to L in MRU position.

We have run 100 executions of this system for every $\textit{size} \in \{0, \ldots, 12\}$. For each run we obtained a timeline and measured its duration weighting events as follows: communication events cost 0 on A, 40 on C and 400 on S; other event cost 0 on A, 1 on C and S. Figure 7 shows the mean value of these timeline durations (estimated cost) with respect to the size of the cache. Because the actor is LRU friendly (with $a = 2$ and $b = 100$), costs decrease with the cache size, until 10 where we reach the number of available keys.

This simple example shows how it is easy to use simulations of modelled systems to analyse the impact of various parameters on the timed performance of the system. We have considered here a simple system with a simple analysis, but it is easy to see that we could have considered many other analyses of the already numerous parameters of this system.

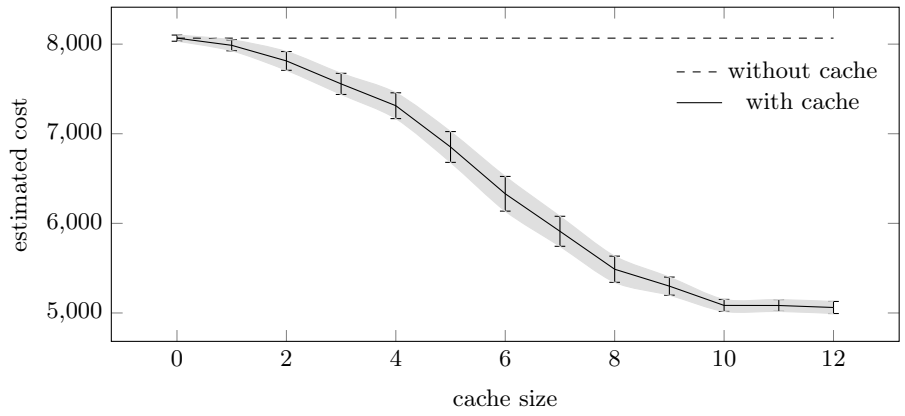

Fig. 7. Estimated performance of our system (lower is better) with respect to cache size. 95% confidence intervals are depicted as vertical segments (and the greyed zone).

5 Conclusion, related work and perspectives

We have presented what is, to the best of our knowledge, the first attempt to provide a generic modelling framework for distributed storage systems, and in particular cache systems. Our proposal allows a separation of usually intricate concerns and can be applied to qualitative or timed analysis. We have illustrated on a simple, yet reasonably realistic, example how a system is modelled and how we can analyse its timed performance.

We have surveyed about 60 papers about caches and distributed storage systems and found no work directly related to ours. However, among others, several papers are worth citing. [1] is probably the first paper to introduce the notion of caches (not yet named this way) using a FIFO eviction algorithm. Later, in [4], LRU (*least recently used*) is introduced, which is further generalised in [10] that considers a hierarchy of caches. A recent evolution is ARC, defined in [9], that is a sophisticated dynamic eviction algorithm which adapts itself with respect to frequently or recently used blocks. Regarding analysis aspects, [10] presents a simulation driven design of an efficient cache algorithm (called *demote*). However, it is not implemented because it involves extensions of existing low-level APIs of storage. This work also introduces the idea of distributed storage by partitioning the key domain across the caches in a hierarchy. Another proposal is [6] that proposes *promote* to fix costs problems of *demote*. An interesting contribution is to introduce a notion of optimality of a cache algorithm, showing that *promote* approaches it. Moreover, this work introduces ideas to address multi-path hierarchies. [8] explores the idea of exploiting the relations between resources, which are discovered through statistical analysis of accesses. In contrast, our proposal make these relations explicit in $\sigma.r$. Finally, an interesting paper is [3] that surveys majors multi-level cache systems, with a classification with respect to collaboration between levels, eviction algorithm and local optimisation strategies. It also shows an analysis of the algorithm through simulation

and actual implementation of widely used benchmarks. These benchmarks could be rendered as dedicated actors in our proposal.

Future work will include the modelling of a realistic system in order to analyse it thoroughly, proving in particular correctness properties through model-checking and comparing the performances of various settings of its parameters. Multi-level variants of LRU, ARC, demote or promote could be good candidates for this case study. We also intend to explore performance analysis directly on the state space, instead of resorting to simulated traces. It may be more accurate than our current simulation-based method, but probably also less efficient if non trivial actors are considered (leading to larger state spaces). Finally, we will consider symbolic techniques to reduce the cost of model-checking on models in our framework. In particular, symmetries reductions on keys like in [5] and finite abstraction of values on infinite domain like in [2] should be easy to adapt to our case and would allow to consider realistic storage sizes (contrasting with the ten keys/values we have considered here). Combining both is however a more challenging problem that we would like to address on the long term.

References

1. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Syst. J. 5 (1966)
2. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of agent-based artifact systems. ArXiv:1301.2678 [cs.MA] (2013)
3. Chen, Z., Zhang, Y., Zhou, Y., Scott, H., Schiefer, B.: Empirical evaluation of multi-level buffer cache collaboration for storage systems. In: SIGMETRICS'05. ACM (2005)
4. Denning, P.J.: The working set model for program behavior. Commun. ACM 11 (1968)
5. Fronc, Ł.: Effective marking equivalence checking in systems with dynamic process creation. In: INFINITY'12. ENTCS, Elsevier (2012)
6. Gill, B.S.: On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In: FAST'08. USENIX Association (2008)
7. de la Houssaye, J., Pommereau, F., Deniel, P.: Formal modelling and analysis of distributed storage systems. Tech. rep., IBISC, Univ. Évry/Paris-Saclay (2014)
8. Li, Z., Chen, Z., Srinivasan, S.M., Zhou, Y.: C-miner: Mining block correlations in storage systems. In: FAST'04. USENIX Association (2004)
9. Megiddo, N., Modha, D.S.: ARC: A self-tuning, low overhead replacement cache. In: FAST'03. USENIX Association (2003)
10. Wong, T.M., Wilkes, J.: My cache or yours? Making storage more exclusive. In: FAST'02. USENIX Association (2002)