Formal Modelling and Analysis of Distributed Storage Systems

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

[1] IBISC, Univ. Évry, IBGBI, 23 bd de France, 91000 Évry, France
jordan.delahoussaye@ibisc.fr, franck.pommereau@ibisc.fr
[2] CEA, DAM, DIF, F-91297, Arpajon, France
philippe.deniel@cea.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, with the innovating feature to separate 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 the latter aspect on a simple yet realistic 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 is caches. They can be found 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 the global 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 exists 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 and innovative part of our contribution (hence its relative complexity) is to provide a clear separation of various concerns with an explicit link between them, as summarised on Figure 1:

- a generic *data model* defines states (see Section 2.1) and allows to consider a variety of operations applicable to them (Sec. 2.2);

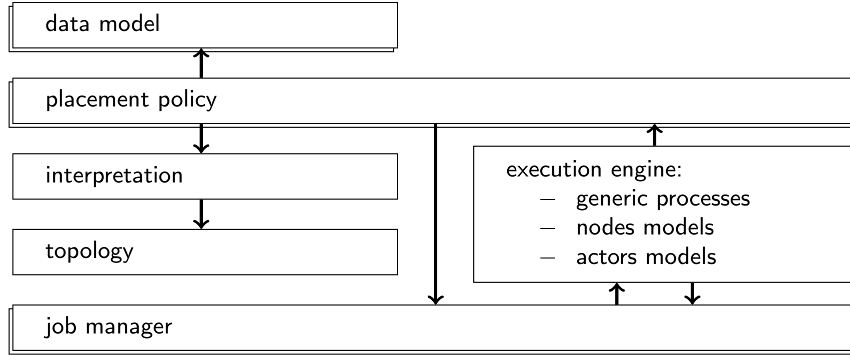

Fig. 1. Overview of the framework where: stacked boxes are replicated on every node of the distributed storage, while the other are globally defined; arrow indicate which aspect relies on which others; boxes on the left-most column (resp. right) relate to static (resp. dynamic) aspects, with two boxes covering both aspects.

- a *topology* is defined independently, describing how states are arranged in the distributed system and how its nodes communicate (Sec. 2.3);
- this leads to a notion of *interpretation* of a distributed state into a global state (Sec. 2.3);
- a *placement policy* decides how to manage the storage on nodes and where each piece of state has to be located (Sec. 2.4);
- a *job manager* is used at each node to store the received requests while they are proceeded, taking into account their dependencies;
- finally, an *execution engine* defines *generic processes* to take into account the requests from the job manager (Sec. 2.5) as well as the *specific processes* executed on each node, in particular *actors* that are the nodes that initiate the whole activity (for instance, a CPU in a cache hierarchy—Sec. 2.6).

All together, these processes yield executions that can be analysed (Sec 2.7) so that 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.

Section 3 illustrates this framework on a simple three-level cache hierarchy equipped a widely used algorithm, and it 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. Finally, Appendix A briefly presents the Petri nets that implement the processes presented in Section 2.3.

2 The Modelling Framework

2.1 Data Model

We consider three pairwise disjoint nonempty sets: K is the set of *keys* that can be thought of as addresses; V is the set of *values* stored at these addresses; L is

the set of *labels* used to label or 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 not be used.

Definition 1. *A* reduced state σ^* *is a pair* $(\sigma^*.h, \sigma^*.r)$ *such that* $\sigma^*.h \in 2^{K\times V}$ *and* $\sigma^*.r \in 2^{L\times 2^K \times 2^K}$. *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* $\sigma^*.h$*:* $\bigcup_{(l,K_1,K_2)\in\sigma^*.r} K_1 \cup K_2 \subseteq \mathsf{dom}(\sigma^*.h)$.

We define a partial order $\preceq^*$ *on* $\Sigma^*_{K,V,L}$ *by* ${\sigma^*}_a \preceq^* {\sigma^*}_b$ *iff* ${\sigma^*}_a.h \subseteq {\sigma^*}_b.h \wedge {\sigma^*}_a.r \subseteq {\sigma^*}_b.r$.

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

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

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

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 a crucial feature of a cache. We have chosen to keep just the necessary information of the history in order to have simpler models. Storing the full history of a state would make models much more memory consuming without providing features identified as useful.

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 $\sigma_a \cup \sigma_b \stackrel{\text{df}}{=} (\sigma_a.\mathit{content} \cup \sigma_b.\mathit{content},\ \sigma_a.\mathit{plus} \cup \sigma_b.\mathit{plus},\ \sigma_a.\mathit{minus} \cup \sigma_b.\mathit{minus})$. 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. Finally, we define *projection* $\gg$ as follows:

$$\begin{aligned}\sigma_a \gg \sigma_b \stackrel{\text{df}}{=} \Big(&\big((\sigma_a.\mathit{content} \cup \sigma_b.\mathit{content}) \setminus \sigma_a.\mathit{minus}\big) \cup \sigma_a.\mathit{plus},\\ &\big((\sigma_a.\mathit{plus} \setminus \sigma_b.\mathit{minus}) \cup \sigma_b.\mathit{plus}\big) \setminus \sigma_a.\mathit{minus},\\ &\big((\sigma_a.\mathit{minus} \setminus \sigma_b.\mathit{plus}) \cup \sigma_b.\mathit{minus}\big) \setminus \sigma_a.\mathit{plus}\Big)\end{aligned}$$

Projection is used to compute the effect of some operations on states. Consider for instance our example of a simplified file system presented above, and take an initial state where only "/" and "`/bin`" exist. The creation of "`/usr`" can be computed through a projection as follows:

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

The definition of projection 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 $\sigma_\emptyset \gg \sigma = \sigma$ and $\sigma \gg \sigma_\emptyset = \sigma$;
- 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 if 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 also hides the mutually cancelling operations.

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. Such 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 of* op*;*
- $op.effect$ *is an expression that can 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 both* $op.effect$ *and* $op.guard$ *can be actually evaluated (*i.e.*, both are actually computable simultaneously).*

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 on which 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 $\beta : \mathsf{vars}(op) \setminus op.params \to K \cup V \cup L$.

It should be stressed that we intentionally avoid to define a precise syntax for expression because we do not want to fix K, V and L, nor we want to restrict the scope of our definitions. The last item in definition 3 is sufficient to ensure that a concrete implementation of the framework has to provide a concrete syntax (possibly a typing) for expressions as well as an effective way to evaluate them.

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:

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

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 $\beta + \{\sigma \rightarrow \sigma_{in}\}$, 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 $op(\sigma_{in}, \beta_{in}) \stackrel{\text{df}}{=} \{(\beta_{out}, op.effect(\beta_{in} + \beta_{out}) \gg \sigma_{in}) \mid \beta_{out} \in op.candidates(\sigma_{in}, \beta_{in})\}$.*

The part of the model defined so far can be used on its own to study the data model itself. For instance, one can check the correctness of (sequences of) operations, on a chosen set of states and input bindings. In other words, it becomes possible to check the correctness of operations with respect to a specification, in particular by using model-checking.

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. If one needs to allow several messages transmissions in parallel, it is possible to model more than one bus between two nodes: for instance, to have two buses between a and b, one could add extra "dummy" nodes 1 and 2 and the buses would be the hyper edges $\{a, b, 1\}$ and $\{a, b, 2\}$. Moreover, a receiver is blocked until a message is sent for it, *i.e.*, reading on a bus is a blocking operation. The possible frames are defined in Figure 2. 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:

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

Fig. 2. 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).

`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 triple (`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.

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 a topology* T *is a pair of functions:*

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

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 $\Sigma_{K,V,L} \times \Sigma_{K,V,L} \rightarrow \Sigma_{K,V,L}$. This one takes a pair of states (σ_a, σ_b) and combines them into a single state σ'_a that can be understood as the integration on a 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 that of 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 identity of the node on which these methods are called, then $P^{me}_{I_T}$ is provided as a set of methods:

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

space $(keys \subseteq K, \sigma_{in} \in \Sigma_{K,V,L}) \rightarrow \mathbb{N}$
: 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 $\mathcal{H}$ like we have used for asynchronous requests; indeed, it is used internally by the nodes for their bookkeeping (see below).

purge $() \rightarrow K$
: Deletes and returns a resource, currently stored on node *me*, which should be considered as the least useful when purge is called. For instance, a LRU cache will exactly choose 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 usefulness 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 process 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. Instead, we have this notion of transactions that we can commit or rollback.

$P_{I_T}^{me}$ can be thought of 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 each other. 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* that is fed by process listener shown in Figure 3: 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 iff $\mathsf{keys}(r_1.\beta) \cap \mathsf{keys}(r_2.\beta) = \emptyset$. The job manager handles these dependencies and provides the following methods:

last $(key \in K) \to \mathcal{H} \uplus \{✗\}$
: Returns the handler of the last request added with a domain including *key*, or a dummy value ✗ if no such request exists.

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 is recorded as dependent on the lastly added request for every key in keys(*request*).

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

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

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

Nodes processes are implemented as coloured Petri nets [8] (see appendix A), however, to clarify the presentation, we provide them here as 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 $\mathsf{listener}! \parallel \mathsf{worker}!$, which is executed in a context with the following global variables:

- me is the node on which the process is executed;
- $jobs_{me}$ 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 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 and to match free variables against received values, in particular the sender's identity (see, *e.g.*, the first instruction of listener);
- ret is a communication channel internal to the node that behaves like a bus, *i.e.*, it provides methods $ret.\mathsf{send}\,(m)$ and $ret.\mathsf{receive}\,(m)$;
- I_T and $P_{I_T}^{me}$ are the interpretation and the placement respectively;
- σ_{me} is the current state.

Figure 3 shows process listener that is responsible for receiving a message for the node, add it to the job manager and send back the response as soon as it is available. It is quite a simple process, but it is worth noting how asynchronous requests are handled. Figure 3 also shows 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 forwards 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 4 also 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.

2.6 Actors

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, and so it stimulates the system in its particular way. For instance, a processor at the top of a memory hierarchy could behave in many different ways

process listener:

- T.receive $(bus, src, me, (kind, req))$ — *# receive a message $(kind, req)$*
- $h \leftarrow jobs_{me}$.add (req) — *# add it to the job manager and get its handler h*
- $P^{me}_{I_T} \leftarrow P^{me}_{I_T}.update(keys(req.\beta), h)$ — *# 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*

process worker:

- $req, h \leftarrow jobs$.next $()$ — *# wait until a new job is available*
- **if** $P^{me}_{I_T}$.where (keys$(req.\beta), \bot) = me$:
  - $c \leftarrow req.op.candidates(\sigma, req.\beta)$ — *# search for possible β_{out}*
  - **if** $c \neq \emptyset$:
    - **choose** $\beta_{out} \in c$ — *# make a non-deterministic choice if $|c| > 1$*
    - $resp \leftarrow ($`success`$, \beta_{out})$ — *# build the response*
    - apply $(req, resp, h, me)$ — *# apply the effect to update σ*
  - **elif** $P^{me}_{I_T}$.where (keys$(req.\beta), \top) \neq$ ✗:
    - $resp, pos \leftarrow$ 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*
    - $resp \leftarrow ($`failure`, `"no node to handle request"`$)$
- **else:** — *# this forwards the request to the appropriate node*
  - $resp, pos \leftarrow$ sync (h) — *# recall that we have $resp[0] \in \{$`failure`, `success`$\}$*
- $P^{me}_{I_T}$.close $(h, resp[0])$ — *# tell the placement about the outcome*
- $jobs_{me}$.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 listener and worker processes, where sub-processes apply and sync are provided in Figure 4.

depending on what kind of programs it is supposed to execute. We can however define basic types of actors that may be useful to exercise a model.

- the *serial player* sequentially sends the messages of a series and wait for each answer before sending the next message;
- the *random messenger* repeatedly sends arbitrary messages, generated from a given set of patterns by instantiating parameters in given ranges. In parallel, it reads the answers as they arrive. To avoid flooding the system, it may count the pending messages (sent requests not yet answered) and avoid to overtake a chosen bound;

```
process apply (req, resp, h, pos):
    k ← keys(req.β + resp.β)                  # get the keys involved in the operation
    P_{I_T}^{me}.update (k, h)    # tell placement that keys k 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

process sync (handler):
    for req, h in jobs.deps (handler):                  # the list order is respected!
        pos ← P_{I_T}^{me}.where (keys(req.β), ⊤)   # where to process req 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                # return the latest response that is for handler
```

Fig. 4. The apply and sync processes called from Figure 3.

- the *scenario performer* plays a scenario describing the messages to send depending of the answers already received. It can be seen as an evolution of the serial player with branches and loops;
- the *profiler* generates and plays a scenario that fits a statistical profile defined from the observation of concrete systems. For instance, we could exercise a cache with respect to spatial and temporal locality observed in real-world programs by generating series of read/write requests on consecutive addresses.

Many other kind of actors may probably be considered. Choosing an adequate model of actor is crucial for a correct analysis. Indeed, most distributed storage systems, and cache policies in particular, are based on strong hypotheses about the access patterns of the systems using them.

2.7 Executions and timed analysis

An actor is implemented as a Petri net that is composed with the Petri nets implementing 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, we propose to apply a *cost function* that maps each transition to its duration. For instance, communication costs can be modelled by weighting appropriately the transitions that correspond to message sending and reception. Applying a cost function results in weighted executions, allowing to compute the global cost of an execution. Concurrent threads of transition firings that occur for distinct nodes are physically executed in parallel, so their costs is not summed but the maximum is taken instead.

3 Application Example

To illustrate our framework, and in particular to clarify what a user has to concretely provide in order to model a system, we propose now a model of a simple hierarchical: 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 $\alpha \stackrel{\text{df}}{=} \{k_1 \to v_1, \dots, k_{sz_K} \to v_{sz_K}\}$ is randomly generated such that σ_S is well-formed, and sz_K is a parameter to control the size of the system, *i.e.*, its number of key/value pairs.

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

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

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 maintains an empty local state):

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

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 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_S is such that every key belongs to S, otherwise it behaves like P_A: where constantly returns S; space constantly returns 0; 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 purge always deletes the key positioned in LRU. So, where returns C if *notme* is false, or S otherwise; space $(keys, \sigma_{in}) \stackrel{\text{df}}{=} max(0, |\sigma_{in}.h| + |keys| - sz_C)$, where sz_C is the size of the cache (*i.e.*, the maximum number of keys it can store); update $(keys, h)$ adds $[(k, h) \mid k \in keys]$ 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; close $(h, outcome)$ either drops from ℓ any pair (k, h) if $outcome =$ `failure` or drops elements at the tail of ℓ until its has at most sz_C elements.

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.

Choosing $sz_K = 10$, we have executed 100 runs of this system for every $sz_C \in \{0, \ldots, 12\}$. For each run we measured its costs using a fixed weighting of events as follows: communication events cost 0 on A, 40 on C and 400 on S; other events cost 0 on A, 1 on C and S. Figure 5 shows the mean value of these estimated costs 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 sz_K where we reach the number of available keys. This closely matches the shape of curves one can obtain from exercising a real LRU. Moreover, curves obtained with larger values of sz_K are closely similar as well.

This 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. A more complex case study can be found in [7, chap. 4] where the *demote* distributed cache protocol presented below is analysed. Both these studies are done within a prototype implementation of the framework presented in this paper. Using the SNAKES toolkit [12], it defines all the classes and methods that correspond to the definitions as well as the necessary to build the Petri net actually used to compute runs or state spaces. In particular, the LRU case study presented in this section requires about 120 lines of Python to be implemented. For comparison, the code for demote in [7] requires about 350 lines of Python. Both codes are straightforward to write, the

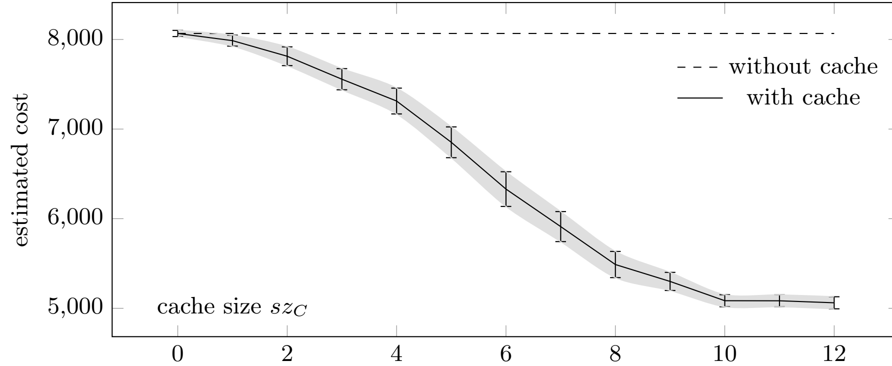

Fig. 5. Estimated cost of a request (lower is better) with respect to cache size; 95% confidence intervals are depicted as vertical segments (and the gray zone).

difficult part being more in the design phase that has to extract the various aspects (in particular the methods for the placement policies) from the original algorithm in which they are intricate.

Note finally that the choice of 100 executions in this section is not due to time limitation. Simulation is fast and we could have run thousands of executions. But 100 is already enough to get smooth curves with good confidence intervals. Time efficiency of simulation based analysis has been observed also on larger instances, as well as on the bigger example of [7].

4 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 has the original feature to allow for a separation of usually intricate concerns. Moreover, it can be applied to qualitative or timed analysis. We have illustrated on a simple yet realistic example how a system can be modelled and its timed performance can be analysed. A more complex case study with a detailed analysis is proposed in [7, chap. 4].

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 [13] that considers a hierarchy of caches. A recent evolution is ARC, defined in [11], that is a sophisticated dynamic eviction algorithm which adapts with respect to frequently or recently used blocks. Regarding analysis aspects, [13] 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 defines *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. [10] explores the idea of exploiting the relations between resources, which are discovered through statistical analysis of accesses. In contrast, our proposal makes 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 be dedicated 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). To cope with this, we shall 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. Combining both is a more challenging problem that we would like to address on the long term. Note however that this is needed for state-space analysis only, indeed, traces are always fast to computed, even with large number of keys as we have experienced using varied parameters of the case study presented in section 3. Moreover, we observed that usually few traces are required in order to obtain smooth curves and good confidence intervals like that of Figure 5.

Another perspective would be to take into account systems with a dynamic topology, in particular, new nodes may appear while other may leave, like in distributed hash tables. This looks like a straightforward evolution of the current setting, but it leads to additional difficulties with respect to state-space analysis. In particular it is likely that we can quickly obtain infinite state systems, which must be avoided to perform analysis other than simulation.

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.: Modèles de stockages distribués appliqués aux caches hiérarchiques. Ph.D. thesis, University of Évry (July 2015)
8. de la Houssaye, J., Pommereau, F., Deniel, P.: Formal modelling and analysis of distributed storage systems. Tech. rep., IBISC, Univ. Évry (2014)
9. Klaudel, H., Koutny, M., Pelz, E., Pommereau, F.: State space reduction for dynamic process creation. Scientific Annals of Computer Science 20, 131–157 (2010)
10. Li, Z., Chen, Z., Srinivasan, S.M., Zhou, Y.: C-miner: Mining block correlations in storage systems. In: FAST'04. USENIX Association (2004)
11. Megiddo, N., Modha, D.S.: ARC: A self-tuning, low overhead replacement cache. In: FAST'03. USENIX Association (2003)
12. Pommereau, F.: SNAKES: A flexible high-level Petri nets library. In: Proc. of PETRI NETS'15. LNCS, vol. 9115. Springer (2015)
13. Wong, T.M., Wilkes, J.: My cache or yours? Making storage more exclusive. In: FAST'02. USENIX Association (2002)

A Petri Nets Implementation

To allow their actual execution, the processes presented in section 2.5 are implemented as colored Petri nets. We present them here to illustrate their complexity but we lack space to give detailed explanations and refer to [7] for such details. We have used the SNAKES toolkit [12] that allows us to split the whole systems into sub-nets with shared places that can be merged latter on. Below, shared places are depicted with thick circles, and are named. Non-shared places are depicted with thin circles and are anonymous. For the sake of readability, we do not draw all the places. Instead we introduce some notations. For instance, the placements are stored in a place *Placement* as pairs (me, P_{me}) for each node me. Access or update to this place are not depicted but an annotation like $n' \leftarrow P_{me}.where(keys(req.\beta), \top)$ in Figure 7 understands a read arc from *Placement* labelled with (me, P_{me}) in order to get the value of P_{me}. Similarly, $P_{me} \leftarrow P_{me}.update(K, h)$ in Figure 10 understands an input arc labelled with (me, P_{me}) together with an output arc labelled by $(me, P_{me}.update(K, h))$that produce the new value for P_{me}. We use similar notations for $jobs_{me}$, I_T and σ_{me}.

Communications are simulated with two shared places used together: *io* and *buses*. Every sent message is produced as a token in place *io*, from where it will be consumed by a transition of the destination node. When a message is sent on a bus b, the token b is also consumed by the sender in place *buses* and it is produced back by the receiver of the message. This way, only one message can transit at a time on a bus. To simplify further the pictures, we also introduce a notation for communications as shown in Figure 6.

In the model, handlers for new requests are created by the job manager. Our implementation makes use a feature of the SNAKES framework: dynamic process identifiers. This feature has initially been created to handle systems that can dynamically start/stop processes, see [9, 5]. We use here the same notations has in these papers to create requests handlers while being able to record by

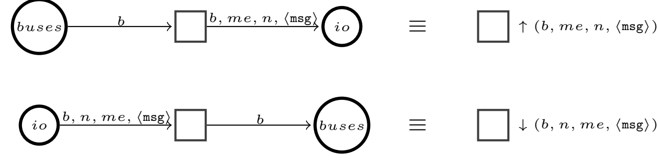

Fig. 6. Notations for communications: the complete nets are depicted on the left and the abbreviated ones on the right.

which node each was created: each node is identified by me that is implemented as a pid (process identifier); given a pid p, $\nu(p)$ creates a new pid that has a parent-child relation with p, and $\chi(p)$ destroys pid p from the system.

A.1 Implementation of the Processes

Process listener (see Section 2.3) is split into two nets for synchronous and asynchronous exchanges depicted respectively in Figures 8 and 9. First, in Figure 8 that depicts the Petri nets implementation of synchronous messages mechanism, place *idle* is used by the sender node to forbid further communications with node n until a response has been received. It is initialised with: $\{(me, n) \in T \mid T[me, n]\}$. Asynchronous messages are handler similarly as depicted in Figure 9, the main difference is that they lead to two successive answers (first a wait, then the corresponding return).

Process sync (see Figure 7) performs a loop over a list of request that need to be sent to another node. It then returns the last response and the node that found it. We use a place *sync* where we put the list, the last response and the last node; then we consume the list request by request until it is empty. Process apply shown in Figure 10 also has an internal loop intended to make some room before integrating a result: it executes $A1$ followed by loops on $B1$ and $B2$, and finally $A2$. Finally, Figure 11 depicts the Petri net implementation of process worker with basically three paths: through $2a$ if the request is not to be handled by the current node; through $2b$ otherwise, then through $3a$ if the request cannot

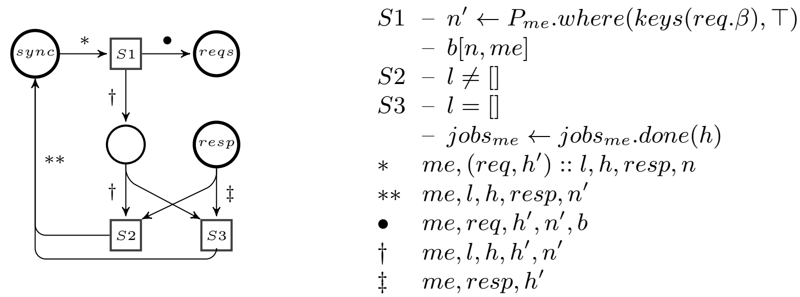

Fig. 7. Petri net implementation of process sync.

be computed locally but can be forwarded, or through $3b$ if an output binding is found and the request can be completed, or finally through $3c$ if no output binding is found and no other node can handle the request.

We refer to [7] for a more detailed description of the processes. However, it is worth noting that these Petri nets have been model-checked on small instances to prove that they always terminate with a correct result. They also have been intensely exercised through simulation of numbers of examples, including large ones, and we have checked that each run was actually a correct execution.

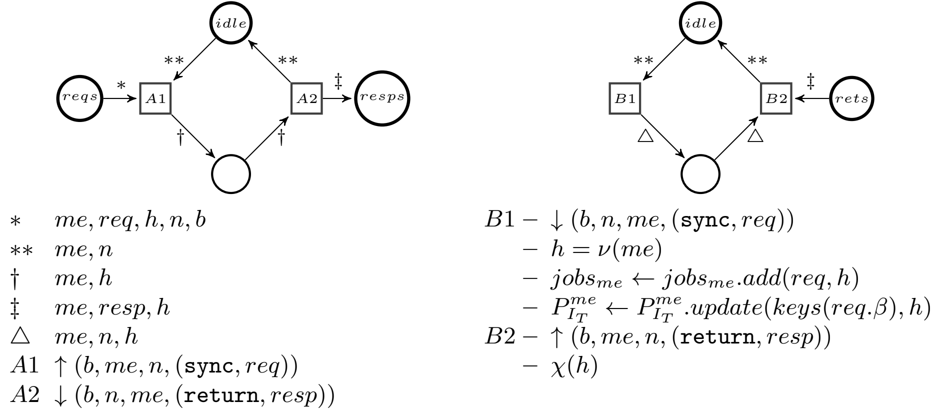

Fig. 8. Petri nets implementation of synchronous exchanges, requests are performed by the left-hand part and answers by the right-hand part of the net.

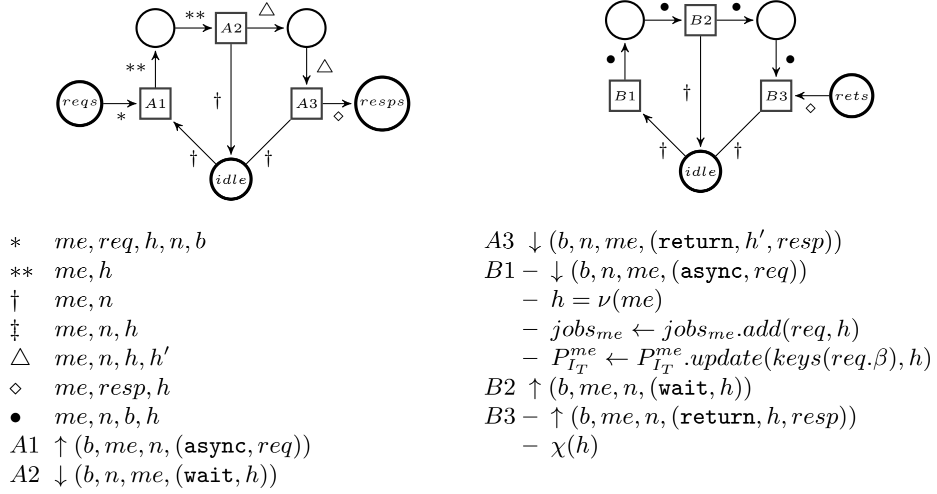

Fig. 9. Petri net implementation of asynchronous exchanges, with requests handled on the left and answers on the right.

$A1$ – $K \leftarrow keys(req.\beta + resp.\beta)$
– $P_{me} \leftarrow P_{me}.update(K, h)$
– $N \leftarrow P_{me}.space(K, \sigma)$
$A2$ – $integrate \leftarrow I_T.integrate(me, n)$
– $\sigma_{me} \leftarrow integrate(\sigma_{me}, req.op.effect(req.\beta + resp.\beta))$
– $jobs_{me} \leftarrow jobs_{me}.done$
$B1$ – $N > 0$
– $P_{me}, least \leftarrow P_{me}.purge()$
– $h' \leftarrow jobs_{me}.last(least)$
– $jobs, l \leftarrow jobs_{me}.deps(h')$
$B2$ – $\sigma_{me} \leftarrow \sigma_{me} \setminus least$

$*$ $me, req, resp, h, n$
$**$ $me, resp, h$
$\dagger$ me, N, h
$\bullet$ $me, N-1, h$
$\ddagger$ $me, 0, h$
$\diamond$ $me, N, least, h', h$
$\triangle$ $me, l, h', ✗, ✗$
∇ $me, [], h', resp, n$

Fig. 10. Petri net implementation of process apply.

1 – $jobs_{me}, req, h \leftarrow jobs_{me}.next()$
– $req \neq ✗$
– $K \leftarrow keys(req.\beta)$
$2a$ – $P_{me}.where(K) \neq me$
– $jobs_{me}, l \leftarrow jobs_{me}.deps(h)$
$2b$ – $P_{me}.where(K) = me$
– $c \leftarrow req.op.candidates(\sigma_{me}, req.\beta)$
$3a$ – $c = []$
– $P_{me}.where(K, \top) \neq ✗$
– $jobs_{me}, l \leftarrow jobs_{me}.deps(h)$
$3b$ – $c \neq []$
– **choose** $\beta_{out} \in c$
– $resp \leftarrow (\text{'}success\text{'}, \beta_{out})$
$3c$ – $c = []$
– $P_{me}.where(K, \top) = ✗$
– $jobs_{me} \leftarrow jobs_{me}.done(h)$
$4a$ – $l = []$
– $resp.type = \text{'}failure\text{'}$
$4b$ – $l = []$
– $resp.type = \text{'}success\text{'}$
5 – $P_{me} \leftarrow P_{me}.close(h, resp.type)$

$*$ me, req, h
$**$ me, req, h, c
$\bullet$ me, h
∇ $me, l, h, ✗, ✗$
$\triangle$ $me, [], h, resp, n$
$\dagger$ $me, req, resp, h, me$
$\ddagger$ $me, req, resp, h, n$
$\diamond$ $me, resp, h$

Fig. 11. Petri net implementation of process worker.