An abstract distributed middleware for transactions over heterogeneous stores

Luca Geatti, Federico Igne, and Marino Miculan*

Department of Mathematics, Computer Science and Physics, University of Udine
luca.geatti@gmail.com, federico.igne@cs.ox.ac.uk, marino.miculan@uniud.it

Abstract We present an abstract middleware, called Acidify, for the coordination of *transactions* between distributed processes accessing shared heterogeneous (possibly remote) storage services. Processes can specify transactions by means of a specific abstract language; each transaction is then executed atomically and in isolation, following an "optimistic" strategy. To ensure scalability and reliability, Acidify is peer-to-peer, without relying on any centralized service. Moreover, it is *abstract*, in the sense that the transaction language is independent from the underlying storage services, and it can be readily ported to any storage service. We provide a formal model of Acidify as a set of interacting automata; this allows us to prove soundness and termination of the algorithms, and to estimate the overhead in terms of exchanged messages and delays. Finally, we provide an implementation of Acidify as an Erlang behaviour, together with the bindings for Riak KV and Amazon S3.

1 Introduction

Nowadays, many applications keep their data "in the cloud", *i.e.* on remotely accessible storage services; this offers virtually unlimited storage space, high reliability, low maintenance costs, and the possibility to share data between processes running on different platforms. However, concurrent access to shared data easily leads to wrong interactions and race conditions. Since there is no general way to handle these conflicts, most storage services offer little o no support to programmers, who have to implement the right solution for each specific situation. Traditionally, this is done by means of various lock-based abstractions (*e.g.*, semaphores and monitors) but it is well known that these mechanisms are deadlock-prone, inefficient, not composable and not scalable. To overcome these issues, *Software Transactional Memory* (STM) has been proposed as a more effective abstraction for concurrent programming [1, 8, 9, 13]. In this approach, code blocks marked as "atomic" are executed guaranteeing the usual *atomicity*, *consistency*, *isolation* properties. Transactions ensure automatic roll-back on exceptions and timeouts, greater parallelizability and, when implemented with optimistic strategies, absence of deadlocks (although they may suffer from livelocks).

* Partially supported by UniUD project PRID 2017 *ENCASE* and MIUR project PRIN 2017FTXR7S *IT-MaTTerS* (Methods and Tools for Trustworthy Smart Systems).

However, when we try to apply the STM approach in cloud-based applications, new important issues arise. First, these applications often access several *heterogeneous* storage services at once, ranging from RAM to files on local file system, from "buckets in the cloud" (*i.e.* on remote servers) to key-value distributed hash tables in P2P networks. Different services offer different access primitives, which are not easy to integrate and abstract from. Secondly, in a local STM implementation concurrent threads and processes ultimately rely on the same runtime support (or operating system) for the transaction coordination and executions, but this cannot be achieved in a distributed setting: a centralized coordinator of transactions would be a single point of failure and a bottleneck, hindering reliability and scalability of the solution.

In this paper, we propose to solve these issues by means of an *abstract distributed* middleware, called Acidify. This middleware allows the programmer to describe transactions using a specifically designed language, independent from the underlying storage systems where objects are actually saved. These code blocks are then executed by Acidify, ensuring atomicity and isolation. Acidify is *abstract* in the sense that it can be used with virtually any storage service, as long as it allows to implement a very simple set of binding functions (essentially for reading and writing data). Thus, we can operate over different (even heterogeneous) storage services at once; moreover, we can replace a storage service with a different one, without modifying the application. The middleware is distributed since it does not rely on any centralized service for the coordination: nodes will negotiate their right to execute their transaction, peer-to-peer. The absence of central coordinators ensures greater scalability and reliability.

A critical part of each node is the Transaction Engine, which is deputed to correctly execute transactions following an optimistic strategy with backward validation. Transaction Engines at different nodes will coordinate their actions by exchanging messages in a peer-to-peer fashion. In order to prove properties of these Engines, we introduce a formal model of distributed transaction machines based on a version of communicating Mealy automata, called *Acid machines* and *Acid systems*. Using this model we are able to prove the correctness of the Engine and protocols, a liveness property, and also that the delay introduced at each commit is constant and equal to 3 round trip time.

After having proved the soundness of this model, we provide an implementation as an Erlang behaviour[1]. Using this behaviour, a programmer can readily implement abstract transactions over heterogeneous shared memories. As examples of storage services, we provide the bindings for Riak KV and Amazon S3.

The rest of this paper is organized as follows. In Section 2 we give a high level view of the system. In Section 3 we focus on the formal model of Acidify transaction engines, *i.e. Acid machine* and *Acid systems*, and their soundness, liveness and consistency properties. Then, in Section 4 we present the implementation of these automata as an Erlang behaviour, and the API offered to the user such as the language for describing transactions. Finally, some conclusions and directions for future work are given in Section 5.

[1] `https://github.com/lucageatti/Acidify`

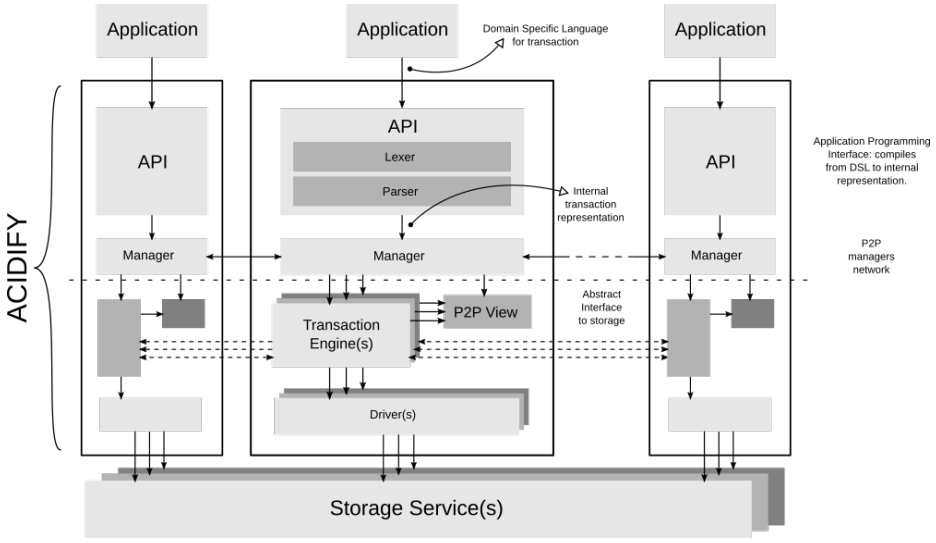

Figure 1. Overview of Acidify architecture.

2 Acidify requirements and architecture

We aim to design a distributed middleware for the execution of atomic transaction over heterogeneous storages. The resulting system will provide *location transparency* (*i.e.* the user does not know where the data is actually stored) and crucially *concurrency transparency* and *consistency*: the user does not know whether other users might be working concurrently over the same data, and if a transaction commits then its changes do not depend on any uncommitted data.

Moreover, Acidify should comply the following main design requirements:

peer-to-peer architecture: the coordination between different concurrent processes has to be carried out in a completely decentralized environment, removing any central authority (*single point of failures*), thus increasing the *fault tolerance* of the system;

modularity: we require our middleware to be modular and agnostic when it comes to the variety of storage services it can access. Extending the middleware to include a new storage service only requires the implementation of a small set of binding functions.

abstraction: the API offered to the middleware user (*i.e.* programmer) should be uniform and independent from the underlying storage services.

In order to achieve the aims above, the Acidify system is designed as in Figure 1. Each *Acidify node* contains the following main logical modules: the *API*, the *Manager*, the *Transaction Engine(s)*, and the *Driver(s)*.

The *API* module implements the interface to the user applications. The user (programmer) describes transactions using a domain specific language, abstract with respect to the storage services (see Section 4.1 for an example language).

These transactions are submitted to the API module, then parsed and translated into a sequence of transactional operations handed to the Manager.

The Manager coordinates the creation (and deletion) of Transaction Engines, and establishes the necessary connections with other nodes working on the same stores. Moreover, the Manager is in charge to keep updated the local view of the peer-to-peer network, discovering new nodes and removing old ones.

The Engines execute the transactions, ensuring atomicity, consistency and isolation. To this end, each Engine keeps a *change log* and communicates with peer Engines in order to coordinate and exchange the needed data (*e.g.* the working sets). Each Engine uses one or more *Driver*s to communicate with storage services. Drivers offer a uniform interface to different storage services, abstracting from their differences. In order to use a storage service with Acidify, it suffices to instantiate a Driver by implementing a small set of callback functions describing how to read and write data on the storage service of interest.

3 Formal model of Transaction Engines

As we have seen above, Transaction Engines have to guarantee the correct execution of transactions. In order to correctly design and implement this module, in this section we briefly discuss the working principles and the algorithms implemented in Acidify, and formalize its behaviour using communicating automata.

3.1 Conflict detection

In order to detect conflicts between transactions, we have adopted the *backward validation* algorithm [6], a kind of "optimistic" concurrency control. Differently from lock-based techniques, optimistic concurrency mechanisms allow transactions to work concurrently, and before committing any change a validation algorithm is executed to check for conflicts. In particular, the transaction execution is divided into three main phases:

Working phase: during this phase, the transaction is executed locally to the Engine; *read* operations are executed against the shared memory, while *write* operations are executed on a local *log*, which is a local partial view of the shared memory. The use of a log allows the Engine to abort without any effect on the memory. In the log we also keep track of *how* the transactional variables have been accessed: *read-only*, *write-only* or *read-write*. For a transaction T, we define its *read set* $RS(T)$ and *write set* $WS(T)$ as the sets of transactional variables it reads and (over)writes, respectively.

Validation phase: the Engine enters this phase at the end of a transaction, when the commit is requested. Before data is committed to the storage, the local log of a transactions is *validated* against any other overlapping[2] transaction log, looking for conflicts. In order to detect conflicts it suffices to compare the read and write sets of the current transaction under validation with those of the overlapping transactions. If a conflict is detected, the transaction under validation is aborted and its log discarded.

[2] Two transactions are *overlapping* if at some point in time both are being executed.

Commit phase: if the validation phase is successful, the transaction enters the commit phase, where all the local changes in the log are made permanent and committed to the shared memory. Since our transactions involve distributed processes, atomicity is achieved with a *two-phase commit protocol* [2]. After committing, the read set is discarded and the write set is kept in order to answer to any request from nodes with overlapping transactions.[3]

We now describe briefly the *backward validation* algorithm adopted in Acidify; we refer to [4] for more details. When it enters its validation phase, each transaction is given a unique integer value called *transaction number* (TN). TNs can be assigned using Birman's total ordering distributed algorithm [3], to compute a total ordering between transactions without a dedicated reliable third party. Given T_i, T_j transactions whose TN are i and j respectively, it holds that:

$$i < j \iff \text{the transaction } T_i \text{ terminated its working phase before } T_j$$

Let $startTN_c$ be the biggest transaction number assigned to some committed transaction at the time when T_c enters its working phase, and let $finishTN_c$ be the biggest transaction number assigned at the time when T_c entered the validation phase. We say that T_i is *overlapping with* T_c if $startTN_c < TN_i < finishTN_c$ and that T_c and T_i are *in conflict* if $(WS(T_i) \cap RS(T_c)) \cup (RS(T_i) \cap WS(T_c)) \neq \emptyset$. The following pseudocode describes the backward validation algorithm for transaction T_c, detecting the transactions in conflict with T_c among all the overlapping ones:

```
for (int i = startTn+1; i < finishTn; i++)
  if (RS(Tc) intersects WS(Ti))
      return false;
return true;
```

3.2 Finite state machine model

To correctly design and implement the algorithms in Acidify, the Engine of a node can be formally defined as a particular type of Mealy machine, with a fixed set of states and transitions. This will allows us to model an Acidify system as a *network* of automata, and to prove the relevant properties. Let us define:

- $\Sigma_{user} = \{\mathsf{atomicRequest(T)}\}$ as the set of messages the user can communicate to the Acidify manager, where the message atomicRequest is parametrized by the transaction T;
- $\Sigma_{net} = \{\mathsf{committedTNrequest}, \mathsf{committedTN(n)}, \mathsf{proposedTN(n)}, \mathsf{newCurrentTN(n)}, \mathsf{requestTNWS}, \mathsf{TNWS(n,S)}, \mathsf{wake}, \mathsf{error(M)}\}$ as the set of messages that different Engines can communicate through the network;

[3] A memory-saving solution would require some mechanism for disposing of useless working sets, *e.g.* by means of a distributed auction system (running asynchronously) to find the most recent transaction which is not overlapping with any other *current* transaction, and discard any transaction with a lower TN.

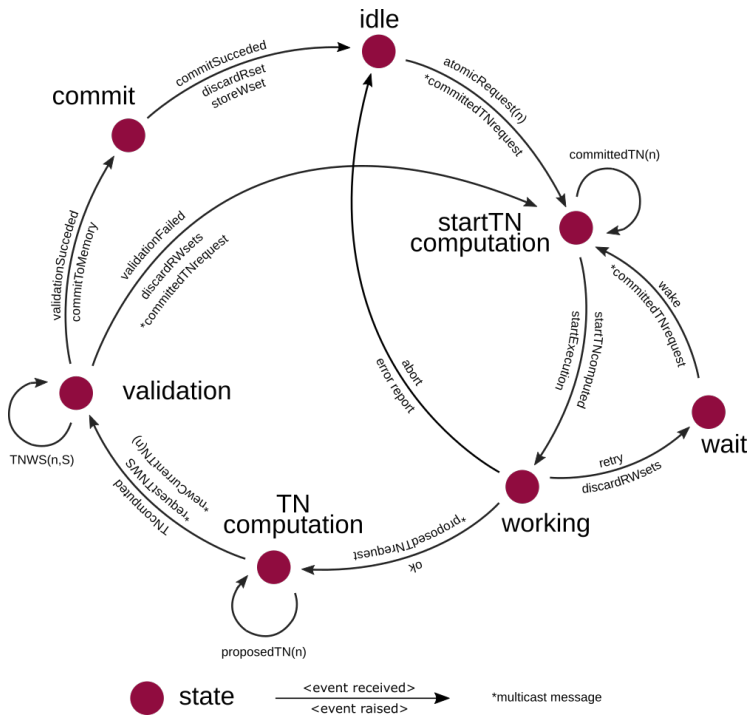

Figure 2. Acid machine. Each transition is labelled with a message it reads (over the arrow) and the corresponding message it issues (under the arrow).

- $\Sigma_{int} = \{$startTNcomputed, startExecution, ok, retry, abort, discardRWsets, discardRset, TNcomputation, validationFailed, validationSucceded, commitToMemory$\}$ as the set of internal messages (*i.e.* "signals") that a machine reads or issues to change its internal phase;
- $\Sigma_{mem} = \{$commitSucceded, error(M)$\}$ as the set of messages that the manager uses to communicate with the shared memory.

Definition 1 (Acid machine). *An Acid machine* $\mathcal{A} = (Q, q_0, \Sigma_I, \Sigma_0, T, G)$ *is a Mealy machine such that:*

- $Q = \{$*idle*, *startTNcomputation*, *working*, *wait*, *TNcomputation*, *validation*, *commit*$\}$ *is the set of states;*
- $q_0 =$ *idle is the initial state;*
- $\Sigma_I = \Sigma_{user} \cup \Sigma_{int} \cup \Sigma_{net}$ *is the input alphabet;*
- $\Sigma_O = \Sigma_{mem} \cup \Sigma_{int} \cup \Sigma_{net}$ *is the output alphabet;*
- $T : Q \times \Sigma_I \to Q$ *is the transition relation depicted in Figure 2;*
- $G : Q \times \Sigma_I \to \Sigma_O$ *is the output function depicted in Figure 2.*

In Figure 2, we depict the transition graph of an Acid machine. We briefly describe the semantics of each of its states:

idle: represents the initial state of the Acid machine. If the user issues the execution of a transaction through the manager by means of a message atomicRequest(T), the engine broadcasts a committedTNrequest message and moves to the next state;

startTNcomputation: represents the initial phase of the backward validation algorithm. It waits for the committedTN(n) message from each engine and computes the maximum, which is the *startTN* value associated with the transaction. When reading the internal message startTNcomputed, the machine proceeds to its next phase, issuing the message startExecution;

working: corresponds to the execution phase of the backward validation algorithm. The engine will handle the local execution of the transaction. This may have three different outcomes: if the execution succeeds (reading ok) the Acid machine moves to the TNcomputation state, broadcasting the message proposedTNrequest; if it stops (reading retry) the Acid machine moves to the wait state; if the execution aborts (reading abort), the Acid machine will report an error and move back to the idle state;

wait: is reached in case of an explicit *retry* from the transaction, issuing the internal message discardRWsets. The log is discarded and the engine will wait until any of the variables involved in the transaction is updated, *i.e.* the event wake. Restarting a computation will restart the backward validation algorithm and compute a new *startTN* value;

TNcomputation: if the transactions ends successful, the engine executes total ordering algorithm to compute its TN (transaction number), reading all the messages proposedTN(n) from the network and broadcasting newCurrentTN(n) to communicate its own new TN. There is no need to compute $finishTN$, since by definition it is equal to $TN - 1$. 1 Once the TN is computed (reading TNcomputed), the machine enters in the validation phase, broadcasting requestTNWS $\in$ over the network, *i.e.* asking for all write sets of transactions associated with a transaction number t such that $startTN < t <$ TN;

validation: corresponds to execution of the backward validation algorithm. It receives the messages TNWS(n,S) from the network and it validates its own read set with respect to the write sets received. If the validation fails (reading validationFailed), the Acid machine restarts the backward validation algorithm, otherwise (it reads validationSucceded) it proceeds to the commit state, issuing the message commitToMemory;

commit is the state when the engine commits all the changes in its log to the shared memory; the backward validation algorithm ensures that it is indeed the only engine in its commit phase[4]. After all changes have committed (reading commitSucceded), the execution is completed: the Acid machine discards its read set (but keeps the write set in order to answer to any request from nodes with overlapping transactions), and moves to the idle state.

[4] This might result in a bottleneck, since only one node at a time is allowed to enter this phase. A possible solution is to preemptively send the writes to the store engine during the validation stage and possibly discard them in case an inconsistent state is reached. However, this approach will generate more network traffic and needs the store engine to implement staging, rollback and commit capabilities.

An Acid machine can also receive asynchronous inputs, *i.e.* committedTNrequest, proposedTNrequest, newCurrentTN(n), requestTNWS, and emit the corresponding outputs committedTN(n), proposedTN(n), TNWS(n,S) in order to take part in the computation of the TN or in the backward validation process of other *engines* in the network. Similarly, it can always asynchronously receive the message error(M), either from the network (*e.g.* the multicast fails) or from the memory (*e.g.* the 2PC process fails), and go a sink state stop. For the sake of simplicity, in the diagram we have omitted these communications and the stop state.

A complete scenario of n engines concurrently accessing the same shared resource can be formalized by the definition of an Acid system.

Definition 2 (Acid system). *An Acid system $\mathcal{S} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ is a network of Acid machines, where each $\mathcal{A}_i$ can asynchronously communicate with any other $\mathcal{A}_j$ either via unicast or multicast, and only sharing messages in Σ_{net}.*

For an Acid system, we can prove the following results.

Proposition 1. *Given an Acid system $\mathcal{S} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, it holds that:*

(soundness) *let $\mathcal{A}_i, \mathcal{A}_j \in \mathcal{S}$ for $i \neq j$; then, it is never the case that both $\mathcal{A}_i$ and $\mathcal{A}_j$ are at the same time into their own commit state, when executing transactions T_i and T_j that are in conflict.*

(liveness) *if a timeout $t < +\infty$ is set for the execution of a transaction on $\mathcal{A}_i$, then $\mathcal{A}_i$ eventually issues either the commitSucceded or the abort message;*

(consistency) *if $\mathcal{A}_i$ issues the commitSucceded message, then the data it has committed does not depend on data which has not been committed.*

Proof. (sketch) (soundness) Follows from the correctness of the total ordering algorithm and backward validation algorithm (we refer to [2, 3] for more details).

(liveness) Follows from the finite state machine behaviour (Figure 2): in particular, it always happens that, if the timeout expires and the algorithm is on a state different from *idle*, then any transaction results into an abort.

(consistency) The data committed by the machine $\mathcal{A}_i$ depend on the read set $RS(T_i)$, which has no intersection with uncommitted write sets, because the backward validation algorithm has concluded successfully. □

It is important to analyse the cost of the algorithm of backward validation, in order to evaluate the overhead it introduces. As usual in distributed algorithms, the cost is given in terms of exchanged messages and communication rounds. For each execution attempt of a transaction, the message traffic introduced by Acidify is linear in the number of transactions it overlaps, and the delay is constant.

Proposition 2 (Cost). *Let $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be an Acid system. When $\mathcal{A}_i$ reaches either the commit or the stop phase or issues the validationFailed message, then*

- *the number of exchanged messages is at most $7n$;*
- *the delay introduced is constant and equal to 3 round trip time.*

Proof. The exchanged messages are $4n$ for the backward validation algorithm (n issued committedTNrequest, n received committedTN(n), n issued requestTNWS and n received TNWS(n,S)), plus $3n$ for the total ordering (Birman's) algorithm (n issued proposedTNrequest, n received proposedTN(n) and n issued newCurrentTN(n)), for a total of $7n$ messages. These messages are exchanged in three rounds: for calculating `startTN`, for calculating `TN` and for collecting the write sets from overlapping transactions. Each round follows a "multicast request/parallel unicast answer" pattern and hence takes a single round trip time. □

It is worth noting that all the exchanged messages but one are fixed-size. The only variable-size message is the representation of the working set, which nevertheless can be encoded efficiently, *e.g.* by means of a map of dirty bits.

4 Implementation

The Acid machine model described in Section 3 has been implemented as an open source Erlang behaviour, called `gen_acid`, and available at `https://github.com/lucageatti/Acidify`. The implementation strictly follows the definition of Acid machine, thanks to Erlang's *finite state machine behaviour*[5]. Using the `gen_acid` behaviour, the user will be able to implement engines for the atomic execution of transactions over a shared memory.

In the rest of this section we will briefly describe the interfaces offered to the programmer for submitting and executing transactions and for adding support for other storage services; finally we give some simple examples.

4.1 A simple language for defining transaction

Acidify's API accepts transactions encoded as strings, akin SQL queries to DBMSs. These strings are actually programs written in a domain specific language for describing transactions. This language should be storage-independent, that is, it should abstract from the differences between access methods offered by different storage services. In this section, we present a simple but expressive language which can be used to this end, whose grammar is given in Figure 3.

We use *transactional variables* (`tvar()`) to refer to data in the shared memory. Transactional variables are identified with an `@` symbol followed by (a tuple of) atoms or strings (*i.e.* `@num42`, `@{foo,bar}` or `@<<"lorem ipsum">>`). `Exprs` are expressions containing integers, booleans and transactional variables, with support for a standard set of arithmetic and logic operators.

Basic operators. The first three operators allow to read and write data on the shared memory, and they involve a direct call to the Driver. The `GET` operator reads the value of a transactional variable and stores it in the local memory of the engine. The `NEW` and `PUT` operators store the value of the expression `Expr` in a transactional variable (respectively a fresh or an existing one).

[5] `http://erlang.org/documentation/doc-8.0-rc1/doc/design_principles/fsm.html`

```
Transaction := Cmds

Block := { } | Cmd | { Cmds }

Cmds := Cmd | Cmd Cmds

ExBlock := { }
         | ExCmd
         | { ExCmds }

ExCmds := ExCmd
        | ExCmd ExCmds
```

```
Cmd := NEW tvar() Expr
     | GET tvar()
     | PUT tvar() Expr
     | RETRY
     | THROW atom()
     | TRY Block CATCH ExBlock
     | IF (Expr) THEN Block
                 ELSE Block
     | WHILE (Expr) Block
     | OR Block ELSE Block

ExCmd := atom() : Block
```

Figure 3. Grammar for the transaction language.

Blocking transactions. Following a lock-free approach, we provide a way for the transaction to explicitly wait for resources. Using the operator `RETRY`, the user forces the transaction to discard local changes and restart the computation. It is worth noting that re-executing the transaction as soon as the *retry* occurs is not efficient; instead, the transaction will wait until at least one variable involved in the computation has been updated by another engine [7].

Control flow and composition. Besides the standard `IF-THEN-ELSE`, `WHILE`, `THROW` and `TRY-CATCH` operators, we introduce an alternative transaction composition in the language by means of a `OR-ELSE` operator as in [7]. The semantics is the following: the first block of operations is executed; if it explicitly `retries`, the block (and its local execution) is discarded and the second block of operations is executed. If this one retries as well, the whole operation retries. In a sense, `OR-ELSE` gives the user the opportunity to choose whether to execute a blocking transaction or to provide an alternative outcome [7].

4.2 Callbacks for Drivers

In order to define the *Driver* used to connect and communicate with the storage service (see Figure 1), the user is only required to provide a small set of callback functions, listed in Figure 4:

- `connect/1` is used by the engine to establish a connection to the remote storage system. It takes a generic term, containing the information for the connection process (*e.g.* IP address and port of a remote server, user credentials, ...), thus allowing the user to adapt the system to virtually any memory. A successful `connect/1` call returns a *connection handle*, keeping track of the information on how to communicate with the memory; this parameter will be passed as input to all the other functions in the module;
- `disconnect/1` is the inverse of `connect/1` and allows the engine to gracefully disconnect from the memory;
- `raw_new`, `raw_get`, `raw_put` are the basic operations used by the engine to declare, read and update values on the storage service. These operations are

```
connect(ConnectArgs :: term())
  -> {ok, ConnectInfo :: term()} | {error, Reason :: term()}.

disconnect(ConnectInfo :: term())
  -> ok | {error, Reason :: term()}.

raw_new(ConnectInfo :: term(), V :: tvar(), Val :: term())
  -> {ok, Val :: term()} | {error, Reason :: term()}.

raw_get(ConnectInfo :: term(), V :: tvar())
  -> {ok, Val :: term()} | {error, Reason :: term()}.

raw_put(ConnectInfo :: term(), V :: tvar(), Val :: term())
  -> {ok, Val :: term()} | {error, Reason :: term()}.
```

Figure 4. Driver callbacks of `gen_acid`.

supposed to be synchronous. Alongside the name of the remote variable and possibly the assigned value, these functions take the connection handle from the `connect/1` call.

The `gen_acid` behaviour has a simple but open interface; the user will be able to write their own clients rather quickly, still having the ability to implement tool-specific features on top of that (*e.g.* type checker). The `gen_acid` repository provides example drivers for the NoSQL distributed database Riak KV and the Object Storage Service Amazon S3, along with some examples of transactions.

4.3 An example transaction

We show how a programmer can use the `gen_acid` behaviour to implement and execute a simple transaction. After having implemented the callbacks described above for the specific store of interest, the programmer can call the function `spawn_engine/3` of the Manager in order to spawn a new engine:

```
spawn_engine(Name :: atom(), Mod :: module(),
             Workspace :: atom(), Args :: term())
  -> ok | {error, Reason :: term()}
```

The function takes a unique `Name` (local to the node) to identify the engine, a module `Mod` implementing the `gen_acid` behaviour, a workspace (which allow transactions to work independently even on the same shared memory) and any argument needed to connect to the storage service.

As an example, we show a simple transaction featuring `new`, `get` and `put` operators, as well as `retry` and `orelse` ones. The pseudocode of this transaction is shown in Figure 5(left), while its formalization in our DSL on the right.

Issuing the execution of a transaction can be achieved via `atomic/3`.

```
atomic(Engine :: atom(), T :: string(), Timeout :: timeout())
  -> {ok, Result :: term()} | {error, Reason :: term()}.
```

The transaction `T` is executed by the engine; it is worth noting that, since `atomic/3` is a blocking call, timeout should be specified.

```
% Lock the mutex
GET @Sem
IF (@Sem > 0)
THEN PUT @Sem @Sem-1
ELSE RETRY

% Release the mutex
GET @Sem
PUT @Sem @Sem+1
```

```
% Lock the mutex
[ gen_acid:get(Sem)
, gen_acid:if_else(
    fun(X) -> X>0 end, [Sem],
    [ gen_acid:put(Sem,
        fun(X) -> X-1 end, [Sem])
    ],
    [ gen_acid:retry() ])
]

% Release the mutex
[ gen_acid:get(Sem)
, gen_acid:put(Sem,
    fun(X)->X+1 end, [Sem])
]
```

Figure 5. Simple transaction implementing a mutex: transaction language (left), internal Erlang representation (right).

5 Conclusions and future work

In this paper we have considered the problem of realizing the *Software Transactional Memory* model in network-centered applications, whose data may be stored on (possibly remote, "in the cloud") storage services and concurrently accessed by processes running on different hosts across the Internet. To address this problem, we have provided the following main contributions:

1. we have introduced Acidify, a modular architecture for the implementation of the STM model in a distributed, peer-to-peer setting (*i.e.* without any centralized coordinator), over heterogeneous storage services;
2. we have defined a formal model of the Transaction Engine of Acidify, based on communicating Finite State Machines (similar to Mealy machines). This model uses optimistic concurrency technique, along with a total ordering among transactions, to ensures the atomic execution of transactions;
3. using this model, we have proved the correctness of the Engine and provided an estimation of the overhead introduced by Acidify;
4. we have also defined a simple language which can be used by applications to submit transactions to the system. These descriptions are translated on-the-fly to an internal representation and then executed by nodes;
5. finally, we have provided an implementation of Acidify as Erlang behaviour, and of the bindings for Amazon S3 and Riak KV. The system is built with a generic interface and can be adapted to virtually any storage service.

A possible future work is to port Acidify to other actor-based languages, such as Scala; in particular we could take advantage of its flexible and powerful type system to guarantee relevant properties of the transaction programs (*e.g.* effect-freeness), like in Concurrent Haskell [7]. Another interesting extension is to allow transactions to communicate during their executions, by means of "retractable messages", along the lines of [10, 12].

Finally, we plan to formally verify the properties of Proposition 1 using suitable formal tools. One possibility could be to adopt *parameterized model checking* techniques [5]; however the problem is far from trivial because it requires to verify a property for an unlimited number of systems, and decidability of the model checking problem may be not even guaranteed. Alternatively, we can resort to a formalization in an interactive proof assistant such as Coq, leveraging existing formalizations of modal-temporal logics [11].

Acknowledgments The authors wish to thank the referees for their useful remarks about the preliminary version of this paper.

Bibliography

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2, 2011.

[2] Y. J. Al-Houmaily and G. Samaras. Two-phase commit. In *Encyclopedia of Database Systems*, pages 3204–3209. Springer, 2009.

[3] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[4] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley, 5th ed., 2011.

[5] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder. Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 122–171. Springer, 2014.

[6] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[7] T. Harris, S. Marlow, S. L. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. PPOPP*, pages 48–60, 2005.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th International Symposium on Computer Architecture*, pages 289–300. ACM, 1993.

[9] M. Herlihy and N. Shavit. Transactional memory: beyond the first two decades. *SIGACT News*, 43(4):101–103, 2012.

[10] M. Lesani and J. Palsberg. Communicating memory transactions. *ACM SIGPLAN Notices*, 46(8):157–168, 2011.

[11] M. Miculan. On the formalization of the modal μ-calculus in the Calculus of Inductive Constructions. *Inf. Comput.*, 164(1):199–231, 2001.

[12] M. Miculan, M. Peressotti, and A. Toneguzzo. Open transactions on shared memory. In *Proc. COORDINATION 2015*, volume 9037 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2015.

[13] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.