Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

On speculative replication of transactional systems

Paolo Romano [a,*], Roberto Palmieri [b], Francesco Quaglia [b], Nuno Carvalho [a], Luis Rodrigues [a]

[a] INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal
[b] Sapienza Università di Roma, Italy

ARTICLE INFO

Article history:
Received 17 December 2010
Received in revised form 28 November 2012
Accepted 22 July 2013
Available online 6 August 2013

Keywords:
Fault tolerance
Distributed protocols
Software transactional memories

ABSTRACT

In this paper we investigate, from a theoretical perspective, the problem of how to build speculative replication protocols for transactional systems layered on top of an Optimistic Atomic Broadcast (OAB) service. The OAB service provides an early, possibly erroneous, guess on transaction's final serialization order. This can be exploited to speculatively execute transactions in parallel with the algorithm used to determine their final total delivery (and serialization) order. To maximize the chances of guessing their final serialization order, transactions are executed multiple times, speculating on the possible orderings eventually determined by the OAB service. We formalize the Speculative Transactional Replication (STR) problem by means of a set of properties ensuring that transactions are never activated on inconsistent snapshots, as well as the minimality and completeness of the set of speculatively explored serialization orders. Finally, we present a protocol solving the STR problem, along with simulation results assessing its effectiveness.

1. Introduction

Active Replication (AR) is a fundamental approach for achieving fault tolerance and high availability [1]. When applied to transactional systems, it requires that replicas agree on a common order for the execution of transactions. This is typically achieved by relying on some form of non-blocking distributed consensus, such as Atomic Broadcast [2] (AB), before transaction processing takes place [3,4].

Given that AB may exhibit non-negligible latency, it is important to design strategies to mitigate its impact. One of such strategies is based on the overlap of local processing and replica coordination [5]. This can be achieved using an Optimistic Atomic Broadcast (OAB) service, that provides an "early" (though potentially erroneous) guessing of the final outcome of the coordination phase [6]. Exploiting the optimistic message delivery order, each site may immediately start the (optimistic) processing of transactional requests without waiting for the completion of the coordination phase.

Clearly, this strategy pays off only if the final total order does not contradict the initial optimistic guess. Further, existing OAB-based solutions only permit the parallel activation of optimistically delivered transactions that are known not to conflict with each other [5,7]. Such a choice simplifies the management of local processing activities, avoiding the risks of propagating the results generated by optimistically delivered transactions. However, it can seriously constrain the achievable degree of parallelism [8]. Additionally, existing approaches require *a priori* knowledge of both read and write sets associated with incoming transactions in order to frame transactions within conflict classes as soon as they are delivered optimistically, i.e., prior to their execution. This requirement raises the non-trivial problem of systematically predicting data access pat-

* Corresponding author.
E-mail addresses: romanop@gsd.inesc-id.pt (P. Romano), palmieri@dis.uniroma1.it (R. Palmieri), quaglia@dis.uniroma1.it (F. Quaglia), nonius@gsd.inesc-id.pt (N. Carvalho), ler@ist.utl.pt (L. Rodrigues).

0022-0000/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jcss.2013.07.006

terns, which may entail significant overestimation of the likelihood of transaction conflicts, with an obvious negative impact on concurrency, especially in case of transactions exhibiting non-deterministic data access patterns.

The above drawbacks are exacerbated in a number of realistic scenarios such as large scale geographical replication, where guessing the final order can be very challenging [9], or in systems where the ratio between the communication delay and the computation granularity is very large, such as Transactional Memories (as discussed, for instance, in [10]).

In this paper, we address these challenges by exploring the use of speculative transaction processing. Particularly, we investigate, from a theoretical perspective, the issues related to the adoption of a speculative approach to replication of transactional systems, which we call Speculative Transactional Replication (STR). The idea underlying STR is rather simple: exploring multiple serialization orders for the optimistically delivered transactions, letting them observe the data item versions generated by conflicting transactions, rather than pessimistically blocking them waiting for the outcome of the coordination phase.

We frame the problem in a (desirable) model in which we do not assume the availability of any *a priori* information on the set of data items to be accessed by the transactions (in either read or write mode), and in which we allow data access patterns to be influenced by the state observed during the execution. Next, we formalize a set of correctness criteria for the speculative exploration of the permutations of the optimistically delivered transactions, demanding the *on-line* identification of all and only the transaction serialization orders that would cause the optimistically executed transactions to exhibit distinct outcomes.

Also, we present an STR protocol relying on a novel graph-based construct, named Speculative Polygraph (SP), which encodes information on the conflict relations materialized during the speculative execution of transactions. SPs are designed to identify what subsets of the speculatively available data item versions would be visible in any view-serializable execution, thus ensuring *completeness* and *non-redundancy* of the set of explored speculative serialization orders.

To assess the viability and practical relevance of our proposal, we also report simulation results based on data access patterns produced by a well-known benchmark for Software Transactional Memories [11].

The remainder of this paper is structured as follows. In Section 2 we discuss related work. Section 3 illustrates the system model. In Section 4 we formalize the properties characterizing STR. In Section 5 we introduce the Speculative Polygraph construct. Section 6 presents the STR protocol, along with the proof of its correctness. Finally, Section 7 provides the results of the simulation study.

2. Related work

Atomic Broadcast (AB) based replication protocols for transactional systems [3–5,12,13] achieve global transaction serialization order (across all the replicas) without incurring the scalability problems affecting classical eager replication mechanisms based on distributed locking and atomic commit protocols [14]. Our STR protocol builds on Optimistic Atomic Broadcast (OAB) [5,6] and, compared to the above results, takes a more aggressive optimistic approach by speculatively exploring the minimum set of serialization orders in which optimistically delivered transactions observe every distinct snapshot of the transactional system's state. Further, unlike other OAB-based schemes [5], STR does not rely on the assumption of *a priori* knowledge of the data items to be accessed by transactions, and makes use of optimistic transaction scheduling to favor concurrency.

To the best of our knowledge the idea of exploiting speculation in transaction processing environments has been first investigated in [15] and [16]. The work by Bestavros and Braoudakis [15] targets *non-replicated* real-time databases and shows the benefits, in terms of transaction timeliness, by speculatively forking, upon detection of a conflict, a copy of the current transaction that remains idle and serves as a save-point to reduce the rollback cost. On the other hand, STR addresses speculation in the context of OAB-based replication of transactional systems, and permits to concurrently process multiple instances of a same transaction in different serialization orders. The solution by Reddy and Kitsuregawa [16] targets distributed databases relying on distributed locking and atomic commit for transaction validation. Further, it constrains the transaction execution model, making the assumption that each data item can be written by a transaction at most once.

3. System architecture

3.1. Distributed system model

We consider a classical distributed system model [17] consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ communicating via message passing, which can fail according to the fail-stop (crash) model.

We assume that the number of correct processes (i.e., processes that do not fail) and the system synchrony level suffice to support an OAB service[1] providing the following interface: `TO-broadcast`(m), which broadcasts messages to all the processes in Π; `Opt-deliver`(m), which delivers message m to a process in Π in a tentative, also called optimistic, order; `TO-deliver`(m), which delivers a message m to a process in Π in a so-called *final order* that is the same for all the processes in Π. The OAB service is characterized by the following properties [6]:

[1] To this end, for instance, it is enough to assume an asynchronous distributed system model, augmented with an eventually perfect failure detector, and the correctness of a majority of processes [17].

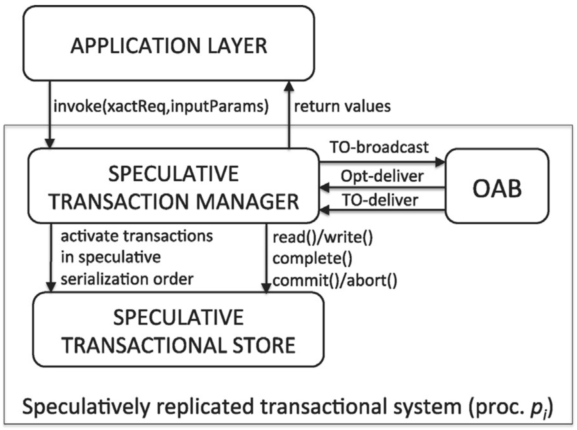

APPLICATION LAYER
invoke(xactReq,inputParams)
return values
TO-broadcast
SPECULATIVE TRANSACTION MANAGER
Opt-deliver
TO-deliver
OAB
activate transactions in speculative serialization order
read()/write()
complete()
commit()/abort()
SPECULATIVE TRANSACTIONAL STORE
Speculatively replicated transactional system (proc. p_i)

Fig. 1. Reference architecture.

- **Termination:** If a correct process TO-broadcasts m, it eventually Opt-delivers m;
- **Global Agreement:** If a process Opt-delivers m, every correct process eventually Opt-delivers m;
- **Local Agreement:** If a correct process Opt-delivers m, it eventually TO-delivers m;
- **Global Order:** If two processes p_i and p_j TO-deliver messages m and m', they do so in the same order;
- **Local Order:** If a process TO-delivers m, it does this only after it Opt-delivers m.

3.2. Software architecture

The diagram in Fig. 1 shows the software architecture of any process $p_i \in \Pi$. The process maintains a (full) copy of a replicated transactional system, such as, for instance, a database or a transactional memory.

Applications generate transactions by calling the `invoke` method of the local Speculative Transaction Manager (SXM), specifying the logic to be executed and the corresponding input parameters (if any). SXM is responsible for (i) propagating to the processes in Π (through the OAB service) the transactions submitted by the application layer, (ii) executing the transactional logic on the underlying Speculative Transactional Store (STS), and (iii) returning the corresponding result to the user-level application.

In this model, the messages disseminated via OAB encode a reference to the transactional logic to be executed by each replica, (e.g., the name of a method containing transactional code, in software transactional memory contexts, or of a stored procedure, in DBMS settings), as well as any of its input parameters. Given that a one-to-one relationship exists between transactions submitted to the replicated system via the `invoke` method and messages disseminated via the OAB layer, for the sake of brevity, in the following, we will use the expressions TO-sent/Opt-delivered/TO-delivered transactions in order to refer to transactions whose corresponding messages have been TO-sent/Opt-delivered/TO-delivered.

STS is the component that maintains the state of a replica and provides APIs for executing transactions in a speculative fashion. Specifically, in addition to classical transaction processing APIs, such as those associated with read, write, commit and abort operations, STS provides programmatic support for activating multiple instances of a transaction, which execute along alternative serialization orders. We intentionally abstract over the mechanisms employed by STS to maintain and regulate concurrent accesses to the multiple data item versions produced by speculatively executing the same transaction in various serialization orders; the definition of these mechanisms is delegated to the protocols solving the STR problem.

3.3. Transaction execution model

In the following we use the term *transaction instance*, or *speculative transaction*, to refer to an incarnation of a transaction generated by executing it in a given serialization order. Each transaction is assigned a unique identifier, referred to as *txId*, and may have multiple speculative instances, each with its own unique sub-identifier, referred to as *specId*. We adopt the notation T^i_X to refer to the instance of a transaction with *txId* $= X$ and *specId* $= i$. Further, we refer to transaction instances associated with the same transaction (say T^i_X and T^j_X) as *siblings*.

We adopt a classical model where a transaction T consists of a sequence of operations, denoted with $\mathcal{O}(T) = \{o_1, \ldots, o_n\}$, where each operation is either a read or a write on a data item. We assume that neither the sequence of operations to be executed within a transaction, nor the data items to be accessed are known *a priori*. Conversely, we assume that the transaction data access pattern can vary depending on the current state of the system. For example, the value returned by a read operation could be used in a control operation and influence the set of operations subsequently executed by that transaction. More precisely, we assume that the transactional business logic is *snapshot deterministic* in the sense that if two instances of a transaction T^i_X and T^j_X observe the same snapshot S (defined by the set of values returned by their read operations), then they execute exactly the same sequence of read/write operations. In this case we denote that $T^i_X = T^j_X$.

On the other hand, if two instances of a transaction T_X^i and T_X^j observe different snapshots, then they may generate different sequences of operations. More formally, consider two executions of the transaction T_X, producing respectively the two sequences of operations $\mathcal{O}(T'_X) = \{o'_1, \ldots, o'_n\}$ and $\mathcal{O}(T''_X) = \{o''_1, \ldots, o''_k\}$ and assume, with no loss of generality, that $k < n$. Let $j \in [2 \ldots k]$ be the index of the first operation in $\mathcal{O}(T'_X)$ and $\mathcal{O}(T''_X)$ for which $o'_j \neq o''_j$ (note that, since transactions are snapshot deterministic, we always have $o'_1 = o''_1$). This implies that, in both $\mathcal{O}(T'_X)$ and $\mathcal{O}(T''_X)$ there exists a read operation on a data item I that is executed before o'_j/o''_j and that returns a different value of I in each of the two transaction instances.

We say that a transaction T that has fully executed its sequence of operations $\mathcal{O}(T)$ is a *completed* transaction. We assume that transactions signal the completion of their sequence of operations, and their intention to commit, by generating a *complete* event. Note that our model does not consider aborts explicitly triggered by the application logic. This is done exclusively for simplifying presentation, as extending the model (and the proposed algorithm) in order to cope with abort operations explicitly triggered by the application logic is straightforward.

A transaction history $\mathcal{H}(S, <_{\mathcal{H}})$ is a partial order defined over a set S of transactions. Analogously to classic transaction history definitions [18], we require that the partial order $<_{\mathcal{H}}$ honors the operations ordering specified within each transaction. A transaction history $\mathcal{H}(S, <_{\mathcal{H}})$ is sequential if $<_{\mathcal{H}}$ is a total order. For the sake of brevity, we write more succinctly $\mathcal{H}$ instead of $\mathcal{H}(S, <_{\mathcal{H}})$, when the set of transactions and their partial order are clear from the context.

4. The Speculative Transactional Replication problem

In this section we introduce the *Speculative Transactional Replication (STR) problem*. This problem is formalized by a set of properties jointly ensuring the consistency of speculative transactions, as well as the exploration of *all and only* the speculative serialization orders in which the transactions observe *distinct* states of the speculative transactional store.

Let $\Sigma = \{T_A, \ldots, T_O\}$ be the set of Opt-delivered but not yet TO-delivered transactions on process p_i, and let $\Sigma' = \{T_A^1, \ldots, T_A^k, \ldots, T_O^1, \ldots, T_O^m\}$ be the set of the corresponding, not yet committed, instances on p_i. We say that the system is *quiescent* if, on any process p_i, the OAB service stops Opt-delivering and TO-delivering transactions, which ensures that Σ does not change over time on that process. Finally, let us denote with $\pi(\Sigma)$ the set of all possible permutations of Σ.

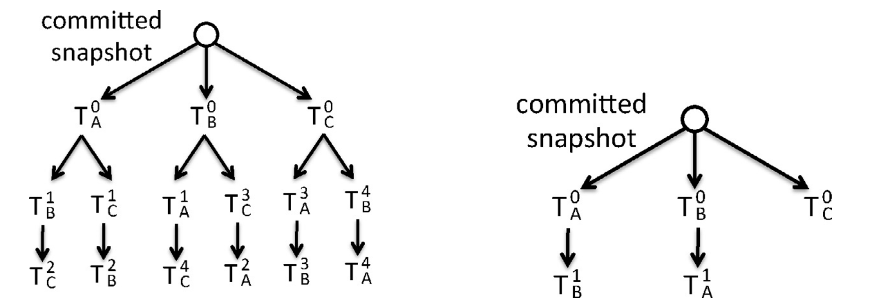
We say that a (distributed) algorithm solves the STR problem if it guarantees, in all of its executions, the following properties:

- **Global-Consistency:** *the history of committed instances is one-copy serializable.*
- **Local-Consistency:** *every instance $T_X^i \in \Sigma'$ sees a snapshot producible by the serial execution of a subset of the transactions in Σ.*
- **Non-Redundancy:** *no two sibling instances in Σ' observe the same snapshot.* (*Specifically,* $\forall T_X^i, T_X^j \in \Sigma'$ *s.t.* $i \neq j$: $T_X^i \neq T_X^j$).
- **Completeness:** *if the system is quiescent then for every permutation $\sigma \in \pi(\Sigma)$ and for every transaction $T_X \in \Sigma$, there eventually exists an instance $T_X^i \in \Sigma'$ that executes on (i.e., observes) the same snapshot that would have been produced by sequentially executing instances associated with all the transactions preceding T_X in σ.*

The global-consistency property focuses exclusively on committed transactions, requesting their compliance with classic one-copy serializability criterion. On the other hand, the local-consistency property defines a safety property encompassing not only committed transactions, but also live transactions and transactions that have to be eventually aborted, e.g., because executed in a speculative serialization order that is eventually found out not to be compliant with the one established by the total order service. In order to filter out anomalous executions that may result from the observation of arbitrary snapshots (e.g., non-serializable snapshots or snapshots produced by the partial execution of transactions), the local-consistency property requires that the snapshot speculatively observed by an instance must be producible by the sequential execution of instances associated with some subset of the Opt-delivered but not yet TO-delivered transactions. Note that this correctness criterion is similar in spirit to virtual world consistency [19] and opacity [20], which, roughly speaking, ensure serializability even for the snapshots observed by active transactions. However, these two properties cannot be rigorously applied to the case of STR, as they do not allow the observation of snapshots generated by not yet committed transactions (in other words they do not encompass speculative propagation of transactions' write sets).

The non-redundancy property filters out trivial solutions based on the exhaustive enumeration of every possible permutation of the Opt-delivered transactions for the construction of plausible serialization orders. Such an approach would certainly enumerate the permutation that will be eventually established by the TO-deliver order, thus providing completeness. On the other hand, denoting with n the number of Opt-delivered but not yet TO-delivered messages, this approach would *always* require executing $\sum_{i=1 \ldots n} \frac{n!}{(n-i)!} = \Theta(n!)$ speculative transactions (i.e., the number of nodes of a permutation tree for a set of cardinality n), independently of the conflict relations actually developed by those transactions. This would likely cause the useless exploration of a (possibly very large) number of redundant serialization orders in which transactions execute along identical trajectories, thus observing the same snapshots and externalizing the same results.

Finally, the completeness property ensures that, if on a process p_i the STR protocol is given sufficient time, the entire set of serialization orders giving rise to different states of the system will be explored. Note that, given that we are assuming an asynchronous system, the assumption on the system's quiescence (ensuring that Σ does not change over time on any process p_i) is a necessary requirement in order to exclude scenarios where processes are never given sufficient time to



(a) History satisfying completeness, but not satisfying non-redundancy.

(b) History satisfying completeness and non-redundancy.

Fig. 2. Illustrating the completeness and non-redundancy properties.

complete the speculative exploration of the set of alternative transaction serialization orders. It is worth highlighting that the completeness property is related to the permissiveness property [21]. The latter property demands that it is possible to identify every possible transaction history that satisfies a given safety property (e.g., serializability), avoiding any unnecessary abort. The completeness property, on the other hand, requires that, given a set of transactions S, it is possible to generate a set of transaction histories, all defined over S, which, roughly speaking, "covers" all the execution trajectories obtained by considering any serialization order of the transactions in S.

In order to provide further hints on the definition of the STR problem, and in particular the completeness and non-redundancy properties, we report in Fig. 2 an example execution in which the set Σ contains three transactions $\{T_A, T_B, T_C\}$. Let us assume that the set of operations issued by these transactions is deterministic (i.e., it is not influenced by the actual values observed by the execution of their read operations), and that these are: $T_A = \{R(X), W(Y)\}$, $T_B = \{R(Y), W(X)\}$, $T_C = \{R(Z), W(Z)\}$

The history shown in Fig. 2(a) achieves completeness by blindly exploring all possible permutations of Σ. The consequence of such a trivial exploration policy is high redundancy, as all the instances of T_C will read and write the same value of z; the same snapshot will be seen and generated also by each of the following pairs of instances: (T^1_B, T^2_B), (T^0_B, T^4_B), (T^1_A, T^2_A), (T^0_A, T^4_A). Conversely, the history shown in Fig. 2(b) ensures both non-redundancy and completeness, activating the minimum number of speculative transactions to "cover" every distinct execution trajectory producible by the transactions in Σ.

5. Speculative polygraphs

In order to determine the set of speculative serialization orders according to which transactions need to be executed to guarantee the completeness property, we introduce a novel graph-based construct, which we call Speculative Polygraph (SP). SPs are a key building block for our protocol to be presented in Section 6. SPs are inspired by Papadimitriou's polygraphs [22], which were introduced to test view-serializability of a non-speculative history $\mathcal{H}$ and whose definition we briefly recall in the following.

Polygraphs. A polygraph $P = (N, A, B)$ is a directed graph (N, A), whose nodes are defined by the set N and whose arcs are defined by the set A, augmented with a set B of so-called *bipaths*. Each bipath is a pair of arcs $\langle(T \rightarrow T''), (T'' \rightarrow T')\rangle$. As we discuss in the following, the existence of bipath $\langle(T \rightarrow T''), (T'' \rightarrow T')\rangle$ implies that $(T' \rightarrow T) \in A$; however, the arcs $(T \rightarrow T'')$ and $(T'' \rightarrow T')$ may not be present in A. De-facto, a polygraph is a compact representation of a family of directed graphs (digraphs) $\mathcal{D}(N, A, B)$. A digraph (N, A') is in $\mathcal{D}(N, A, B)$ if and only if $A \subseteq A'$, and, for each bipath $(a_1, a_2) \in B$, A' contains at least one of the arcs a_1 and a_2.

Polygraphs capture partial order relations in a history of transactions, and the polygraph $P(\mathcal{H})$ associated with a history $\mathcal{H}$ is constructed according to the following two rules: (i) whenever a transaction T reads a version of data item I from transaction T', the arc $(T' \rightarrow T)$ is added in A; (ii) if a third transaction T'' also writes I, then the bipath $\langle(T \rightarrow T''), (T'' \rightarrow T')\rangle$ is added to B. In other words, the arc (T', T) in A keeps track of the direct read-from relation between transactions T and T', whereas the bipath $\langle(T \rightarrow T''), (T'' \rightarrow T')\rangle$ captures the fact that since also T'' writes I, it cannot be serialized between T' and T, but must either follow T or precede T'.

Based on the above definition of polygraph, Papadimitriou defines a polygraph as acyclic iff there is at least one acyclic digraph in $\mathcal{D}(N, A, B)$ and proves that a history $\mathcal{H}$ is view-serializable iff its polygraph $P(\mathcal{H})$ is acyclic [22]. We show in Fig. 3(a) and 3(b) the polygraphs associated with, respectively, the following two sequential transaction histories $\mathcal{H}_1 = \{T_A \rightarrow T_B \rightarrow T_C\}$ and $\mathcal{H}_2 = \{T_B \rightarrow T_A \rightarrow T_C\}$, where T_A writes on data items X and Y, T_B writes on data items Y and Z, and T_C reads X, Y and Z (returning different values for Y in the two histories given that $\mathcal{H}_1$ and $\mathcal{H}_2$ serialize T_A and T_B in different orders). According to the notation adopted in [22], a bipath is identified by drawing an arc segment between alternative edges, around their common node. It can be easily verified that the only acyclic digraph associated with the

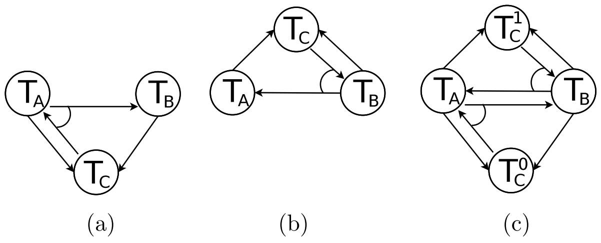


Fig. 3. Polygraphs associated with histories $\mathcal{H}_1$ and $\mathcal{H}_2$.

polygraph in Fig. 3(a) is obtained by selecting the edge $(T_A \to T_B)$ of the bipath centered on T_A, whereas the only acyclic digraph associated with the polygraph in Fig. 3(b) is obtained by selecting the edge $(T_B \to T_A)$ of the bipath centered on T_B.

Speculative polygraphs. Conventional polygraphs are not appropriate for reasoning about speculative transactional histories. If instances associated with incompatible serialization orders are mixed in a same polygraph, cycles can appear that are irrelevant. An example of such a problem is shown in Fig. 3(c), which shows the polygraph obtained by merging the polygraphs in Fig. 3(a) and 3(b), when considering that two different instances of transaction T_C, namely T_C^0 and T_C^1, are speculatively serialized according to different histories (in this example the *specId* values associated with instances of transactions T_A and T_B are irrelevant). It is straightforward to verify that every directed graph associated with this polygraph is cyclic. This is of no surprise considering that the polygraph keeps track of every partial order relation in a history that contains transactions that assume opposite serialization orders for T_A and T_B.

Speculative polygraphs address exactly these problems. Unlike Papadimitriou's polygraphs, which are representative of a *whole*, and non-speculative, history, SPs are designed to take into account the execution history as perceived by each speculative transaction. Roughly speaking, the SP of a transaction T_X^i is dynamically generated by selectively merging only the polygraphs of those speculative instances T_*^* that i) conflict, either directly or indirectly, with T_X^i, and ii) such that there exists at least one (non-speculative) serialization order which allows both T_*^* and T_X^i to coexist.

More formally, we define the speculative polygraph associated with transaction T_X^i, denoted as $\mathrm{SP}(T_X^i)$, as a triple (N, A, B) where:

- N is a set of nodes, each one representative of some speculative transaction.
- A is a set of, so-called, *merging edges* denoted as $(T_R^j \circledast \to T_X^i)$, with $T_R^j, T_X^i \in N$, and where the notation $T_R^j \circledast$ means that we are not just adding an edge linking T_R^j and T_X^i, instead we are creating a new SP obtained by performing the union[2] of $\mathrm{SP}(T_R^j)$ and $\mathrm{SP}(T_X^i)$, and adding to the resulting SP a (plain) edge from T_R^j to T_X^i.
- B is a set of, so-called, *asymmetric bipaths*, which we denote as $\langle (T_U^k \circledast \to T_R^j), (T_X^i \to T_U^k)\rangle$, with $T_R^j, T_X^i \in N$, where the first of the two arcs is a merging edge linking $\mathrm{SP}(T_U^k)$ with $\mathrm{SP}(T_R^j)$ through the plain edge $(T_U^k \to T_R^j)$, and the second one, namely $(T_X^i \to T_U^k)$, is a plain edge between the nodes T_X^i and T_U^k.

A speculative polygraph $\mathrm{SP}(T_X^i) = (N, A, B)$ generates a family of directed graphs $\mathcal{D}(SP(T_X^i))$, where each directed graph $\delta \in \mathcal{D}(SP(T_X^i))$ is obtained by (1) recursively replacing any merging arc, say $(T_R^j \circledast \to T_V^l)$, of A and B with the speculative polygraph $\mathrm{SP}(T_R^j) \cup (T_R^j \to T_V^l)$, and (2) for each asymmetric bipath $\langle a_1, b_1\rangle$ present after the previous "merging phase", selecting either a_1 or b_1.

Upon activation of an instance T_X^i, its speculative polygraph $\mathrm{SP}(T_X^i)$ is initialized by serializing T_X^i after the most recently committed instance, say T_C^h, via a merging edge, namely $(T_C^h \circledast \to T_X^i)$. As a result, the speculative polygraph of T_X^i is initialized with a copy of the speculative polygraph $\mathrm{SP}(T_C^h)$, which is augmented by adding a plain edge from T_C^h to T_X^i. In terms of logical time, this corresponds to setting a barrier for the visibility of data item versions observable via read operations by T_X^i.

The speculative polygraph associated with transaction T_X^i is then used to determine whether the k-th read operation of T_X^i can return a given version I^v (possibly created by a not yet committed transaction). Indeed, letting the k-th read of T_X^i return a specific version I^v, rather than any other available version, corresponds to speculating on a set of possible serialization orders for T_X^i. In order to ensure the consistency of the k-th read by a transaction within its current execution history, it is however necessary that at least one of the speculative serialization orders associated with the reading of version I^v is "compatible" with those already determined after the execution of the preceding k-1 reads. In this case we say that

[2] Given two SPs, $\mathrm{SP}^1 = (N^1, A^1, B^1)$ and $\mathrm{SP}^2 = (N^2, A^2, B^2)$, we formally define the union operation between SP^1 and SP^2 as follows: $\mathrm{SP}^1 \cup \mathrm{SP}^2 = (N^1 \cup N^2, A^1 \cup A^2, B^1 \cup B^2)$.

```
// Given an instance $T^i_X$, its current speculative polygraph, and a data item version $I^v$, return $\perp$ if $I^v$
// is not visible; otherwise returns SP($T^i_X$), updated to reflect the read of $I^v$ by $T^i_X$
method SpeculativePolygraph isVisible(DataItemVersion $I^v$, SpeculativePolygraph currSP, TxInstance $T^i_X$)
    SpeculativePolygraph newSP = $(I^v.\text{creator}\circledast \to T^i_A) \cup$ currSP;
    forall $(I^v \neq I^w) \in$ TS.getAllVersions(I) do
        newSP = newSP $\cup\{\langle (I^w.\text{creator}\circledast \to I^v.\text{creator}),\ (T^i_X \to I^w.\text{creator})\rangle\}$;
    if $(\exists\delta \in \mathcal{D}(\text{newSP})$ s.t. isValid($\delta$, $T^i_A$))
        return newSP;
    else  return $\perp$;

method boolean isValid(DirectGraph $\delta$, TxInstance $T^i_A$)
    return ($\delta$ is acyclic $\wedge$ ($\nexists$ $(T^q_X \to T^i_A), (T^r_X \to T^i_A) \in \delta^*$ with $q \neq r$));
```

Fig. 4. Pseudo-code of the version visibility logic.

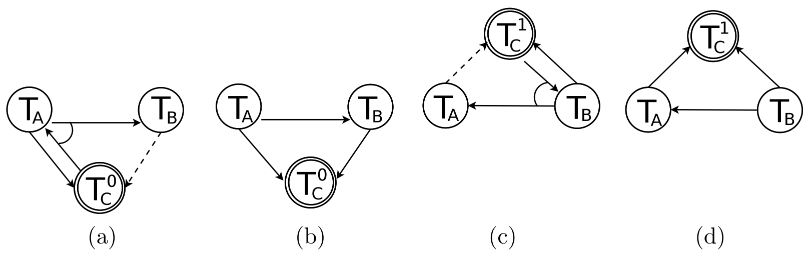


Fig. 5. SPs associated with histories $\mathcal{H}_1$ and $\mathcal{H}_2$.

I^v is speculatively visible to T^i_X. To determine if T^i_X may speculatively view a data item version I^v based on its current execution history, its speculative polygraph SP(T^i_X) is updated by:

(R.1) adding a merging edge from the creator of version I^v to T^i_X, namely $(I^v.creator\circledast \to T^i_X)$;
(R.2) adding, for each other available version $I^{v'}$, an asymmetric bipath: $\langle (I^{v'}.creator\circledast \to I^v.creator), (T^i_X \to I^{v'}.creator)\rangle$.

Version I^v is considered speculatively visible iff there exists at least one directed graph $\delta \in \mathcal{D}(\text{SP}(T^i_X))$ such that:

(C.1) δ is acyclic, and
(C.2) δ does not contain two sibling instances T^r_Y, T^q_Y both serialized before T^i_X, or, formally, $\nexists T^r_Y, T^q_Y \in \delta$ such that $(T^r_Y \to T^i_X) \in \delta^*$ and $(T^q_Y \to T^i_X) \in \delta^*$, where with δ^* we denote the transitive closure of δ.

If for a $\delta \in \mathcal{D}(\text{SP}(T^i_X))$ both conditions C.1 and C.2 hold, we also say that δ is *valid*. The pseudo-code describing the logic used to determine speculative visibility of data item versions is reported in Fig. 4.

The rationale underlying the above rules is to ensure that, whenever a transaction T^q_Y is serialized before T^i_X, the speculative polygraphs of both transactions are recursively merged. This ensures that the resulting SP(T^i_X) keeps a complete track of any conflict relation among the transactions that generated the snapshot seen by T^i_X. On the other hand, whenever T^i_X issues a read operation on a data item for which there exists a version created by a transaction T^u_Y, whose SP(T^u_Y) cannot be merged with SP(T^i_X) without generating cycles in any $\delta \in \mathcal{D}(\text{SP}(T^i_X))$,[3] rule R.2 allows serializing T^u_Y after T^i_X through a plain edge *without* requiring to merge the polygraph of T^u_X. This prevents corrupting SP(T^i_X) by blindly incorporating into it the history of transactions associated with incompatible speculative serialization orders, and whose writes shall never be visible to T^i_X. On the other hand, by serializing T^u_Y after T^i_X through the plain edge of an asymmetric bipath, we can still detect cyclic dependencies involving transactions not serializable before T^i_X in SP(T^i_X). Finally, condition (C.2) avoids reading inconsistent snapshots generated by an execution history which serializes (at least) a pair of different sibling transactions T^r_Y, T^q_Y before T^i_X, as in any serial history a transaction T_Y can be committed in a single serialization order.

Example scenarios. Let us first reconsider $\mathcal{H}_1$ and $\mathcal{H}_2$, which were previously used to highlight the inability of classic polygraphs to manage speculative histories. In Fig. 5(a) and Fig. 5(c) we show, respectively, the SP of T^0_C and T^1_C. Note that, in order to refer to a merging edge, and distinguish it from a plain edge, we use a dashed arrow. Further, within the speculative polygraph SP(T^i_A), we use the convention of drawing a double circle to refer to transaction T^i_A. In Fig. 5(b), resp.

[3] This may happen for instance if the polygraphs have two transactions in common, but ordered in an opposite manner.

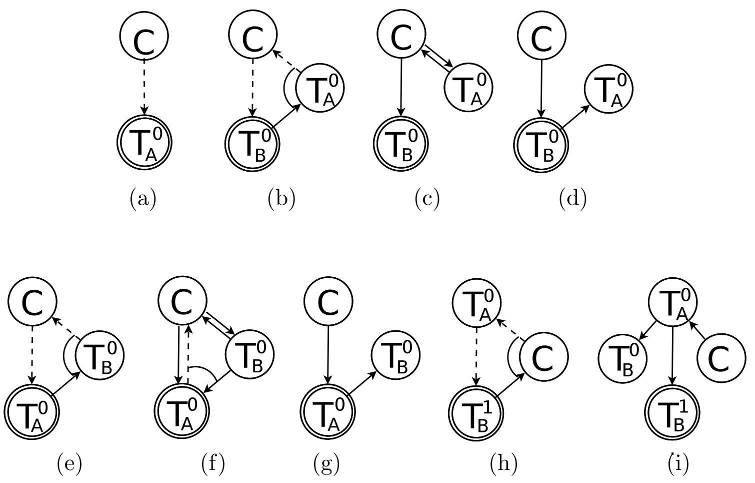


Fig. 6. SPs associated with history $\mathcal{H}_3$.

Fig. 5(d), we report the only acyclic directed graphs that can be obtained by the corresponding SP shown in Fig. 5(a), resp. Fig. 5(c).

Let us now consider a slightly more complex example, associated with the following history:

$$\mathcal{H}_3 = \left\{B_{T_A^0}, R_{T_A^0}(I), W_{T_A^0}(I), C_{T_A^0}, B_{T_B^0}, R_{T_B^0}(I), F_{T_B^1}, W_{T_B^0}(I), R_{T_B^1}(I), W_{T_B^1}(I)\right\}$$

where we use the notation $B_{T_X^i}$, $R_{T_X^i}$, $W_{T_X^i}$, $C_{T_X^i}$ and $F_{T_X^i}$, to denote, respectively, begin, read, write, complete (hence not commit) and fork of a transaction T_X^i.

We shall assume that the only version for each data item present in memory is the committed version, and denote with C the identifier of the last committed transaction. In Fig. 6(a) we show the state of SP(T_A^0) right after the execution of the read operation on X.

Fig. 6(b) shows the state of SP(T_B^0) after the execution of the read on X, assuming that T_B^0 reads the committed version. Since T_A^0 has already completed executing (but has not been committed yet) at the time in which T_B^0 performs the read, T_B^0 finds available also the versions created by T_A^0. Thus, SP(T_B^0) also contains the asymmetric bipath $\langle (T_A^0 \circledast \to C), (T_B^0 \to T_A^0) \rangle$, which we denoted by adding a circular arc on the common node. Fig. 6(c) and Fig. 6(d) show the two directed graphs $\delta, \delta' \in \mathcal{D}(\text{SP}(T_B^0))$. As it can be seen, the directed graph in Fig. 6(c), associated with the merging edge $(T_A^0 \circledast \to C)$, exhibits a cycle. However, since the directed graph in Fig. 6(d) is acyclic the committed version of I is speculatively visible to T_B^0.

In Fig. 6(e) we provide the speculative polygraph of T_A^0 after the update performed during the completion phase of T_B^0. As mentioned in Section 6.1, when T_B^0 completes its execution, it makes available the version of X it generated and updates the SPs of the transactions that already issued a read on I. This ensures that the SPs of these latter transactions reflect that they did not observe the version that T_A^0 is externalizing. In the considered example, this causes the addition of the asymmetric bipath $b_1 = \langle (T_B^0 \circledast \to C), (T_A^0 \to T_B^0) \rangle$ to SP(T_A^0). In Fig. 6(g), we show the only valid directed graph in $\mathcal{D}(\text{SP}(T_A^0))$, which serializes T_A^0 before T_B^0. In fact, as shown in Fig. 6(f), by considering the merging edge $(T_B^0 \circledast \to C)$ of b_1, and merging SP(T_B^0) with SP(T_A^0), the resulting speculative polygraph is necessarily cyclic.

Finally, we show in Fig. 6(h) the speculative polygraph of T_B^1, namely the transaction forked by T_B^0 upon the read of data item I, and which returns the version of I written by T_A^0. Fig. 6(i) shows the only directed graph in $\mathcal{D}(\text{SP}(T_B^1))$ to be acyclic, which permits the speculative visibility of the version written by T_A^0.

6. A protocol solving the STR problem

This section describes a protocol that solves the STR problem. We start by providing a global overview of the proposed STR protocol in Section 6.1. In Section 6.2, we specify the used primitives. The pseudo-code for the algorithms of the Speculative Transaction Manager (SXM) and the Speculative Concurrency Control (SCC) is presented, respectively, in Section 6.3 and Section 6.4. In Section 6.5, we prove that the proposed protocol satisfies the STR properties. Finally, the special case of read-only transactions is discussed in Section 6.6.

6.1. Protocol overview

The architecture of the proposed STR protocol is shown in Fig. 7. The diagram represents a refinement of the higher-level architecture illustrated in Fig. 1, in which the Speculative Transactional Store has been subdivided in two modules: the

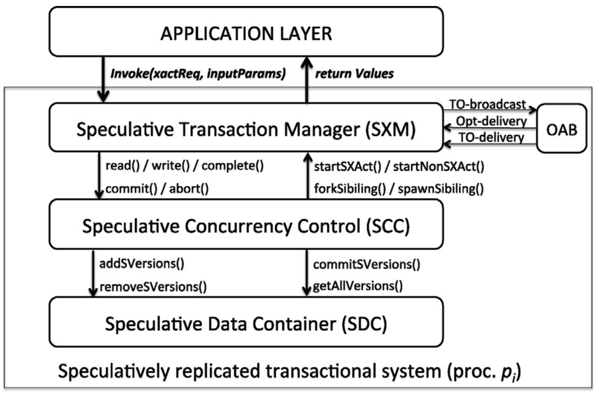

APPLICATION LAYER
Invoke(xactReq, inputParams)
return Values
Speculative Transaction Manager (SXM)
TO-broadcast
Opt-delivery
TO-delivery
OAB
read() / write() / complete()
commit() / abort()
startSXAct() / startNonSXAct()
forkSibling() / spawnSibling()
Speculative Concurrency Control (SCC)
addSVersions()
removeSVersions()
commitSVersions()
getAllVersions()
Speculative Data Container (SDC)
Speculatively replicated transactional system (proc. p_i)

Fig. 7. Architecture of each process for the proposed STR protocol.

Speculative Concurrency Control (SCC) and the Speculative Data Container (SDC). The SDC layer abstracts low level storage mechanisms, which may encompass RAM-only memory accesses (as for the case of transactional memories) and logging on persistent storage to ensure transaction durability (as for the case of a conventional DBMS). We assume that each data item I maintained by SDC is associated with a set of versions $\{I^1, \ldots, I^n\}$, where each version I^v stores (i) the corresponding data item value and (ii) the identity of the creating transaction. SDC exposes primitives for retrieving the set of versions associated with a data item, and for making atomically (all-or-nothing) visible the new versions of the set of data items updated by a transaction. At any time, SDC maintains a single committed version of each data item, plus possibly several, not yet committed versions, created by speculative instances that have been fully executed but whose final serialization order has not yet been determined.

The interactions between SXM and SDC are mediated by the SCC layer, which exposes a classical interface to issue read/write operations and to commit/abort transactions. SCC can trigger the activation of additional instances associated with a given transaction, in order to explore, in a speculative fashion, alternative serialization orders for that transaction. As we will explain more in detail, SCC relies on two mechanisms to activate a transaction T^i_X: i) by creating a new thread that executes the transactional logic of T_X from the first operation, or ii) by forking a thread executing an already active instance T^i_X; the new spawned thread will continue (without restarting) to execute the transactional logic of T_X from the point where T^i_X was forked (naturally, if multiple instances are forked from the same parent instance T^i_X, each sibling will observe a different snapshot to ensure non-redundancy).

Transactions are submitted by the application via the `invoke` primitive, and are disseminated among the replicas via the OAB service. Each replica immediately starts processing transactions as soon as these are optimistically delivered by the OAB service. The issue of generating a speculative set of different serialization orders is tackled by the SCC layer. SCC dynamically tracks the dependencies developed during the execution of transactions via Speculative Polygraphs (SP).

Consider that the n-th operation of instance T^i_X is the first read operation by T_X on data item I. Denote $V(I)$ the available versions of a data item I at that point. SCC exploits knowledge on conflict dependencies tracked by the SP associated with transaction T^i_X to determine which versions can be returned without violating view-serializability (given the history of the former $n-1$ read operations of T^i_X). We denote these versions as $\mathit{SAFE}^i_X(I)$.

SCC lets T^i_X observe one version in $\mathit{SAFE}^i_X(I)$, and forks $|\mathit{SAFE}^i_X(I)|-1$ threads that will read a different version of I in the set $\mathit{SAFE}^i_X(I)$. In this way, the SCC covers all the distinct execution trajectories that T^i_X could undertake by letting the read operation return a different value (each representative of some view-serializable execution history) among those available for I.

Further, T^i_X forks another thread, which will run in background till T^i_X is either committed or aborted. This thread will just wait for new versions of the data item I to become available (because created by some committed or speculatively completed instance) and, when any new version becomes available, it will check if new values may be added to $\mathit{SAFE}^i_X(I)$. Every time a new value is added $\mathit{SAFE}^i_X(I)$, a new sibling is spawned using the new value as part of its snapshot.

The reason for the behavior above is that forking T^i_X when I is first read is insufficient to ensure the complete exploration of speculative serialization orders. In fact, new versions of a data item I can become available after the execution of the first read on I by instance T^i_X, if some other instance T^j_Y later writes on I. By spawning additional siblings that observe these snapshots, when they become available, we ensure completeness.

We opt to make new versions of a data item visible only when an instance completes its execution (rather than as soon as the write operation is completed). This avoids scenarios in which an instance that writes multiple times the same data item causes the activation of new speculative transactions that observe “intermediate” values that would have never been visible in any view-serializable history. Such a phenomenon could lead to violations of the local-consistency property and to suboptimal usage of computing resources (wasting time executing doomed speculative transactions). Furthermore, making “intermediate” values visible may also allow the transactional logic to generate other anomalies [20] in

```
Set of TxInstance: activatedXacts;
Set of TxInstance: completedXacts; // CompletedXacts ⊆ ActivatedXacts
Set of TxInstance: committedXacts;
Sequence of TxId: delivered;

1: method Result invoke(TransactionalLogic T, inputParams p) do
2:   int: id = getNewXactID();
3:   OAB.TO-broadcast([T, p, id]);
4:   wait until $\exists i : T^i_{id} \in$ committedXacts;
5:   return $T^i_{id}$.getResult();

6: upon Opt-Deliver([Transaction T, inputParams p, TxID X]) do
7:   startSXact(T, p, X);

8: upon TO-Deliver([Transaction T, inputParams p, TxID X]) do
9:   delivered.add (X);
10:  if ($\exists T^i_X \in$ completedXacts s.t. $T^i_X$.isValid())
11:    $T^i_X$.commit(); //the commit method aborts any sibling transaction of $T^i_X$
12:  else
13:    forall $T^i_X \in$ completedXacts do
14:      $T^i_X$.abort(); //any completed $T^i_X$ is invalid
15:    if ($\nexists T^i_X \in$ activatedXacts)
16:      if ($\forall R \in$ delivered $\exists T^j_R \in$ committedXacts)
17:        startNonSXact(T, p, X);
18:      else
19:        startSXact(T, p, X);
```

Fig. 8. Pseudo-code for the Speculative Transaction Manager.

contexts where the transactional system is not fully isolated by the external environment (such as transactional memory systems [23]).

6.2. Primitives and notations used in the pseudo-code

We assume that the logic associated with a transaction T_X is activated via the `startSXact` and `startNonSXact` primitives. These take as input parameter a transaction T_X and activate a new thread which starts a new instance T^i_X (where i is a freshly generated *specId* value) in a speculative and non-speculative mode, respectively. As it will be further discussed in the following, an instance of transaction T_X is activated in non-speculative mode only if its TO-deliver order has already been established and all the transactions that precede T_X in the TO-deliver order have already been committed. Otherwise, the instance is activated in speculative mode.

We associate each instance with an object of the TxInstance class, and abstract over the implementation of the application logic by assuming that, whenever the thread executing an instance issues a read/write operation, the read/write method of the corresponding transaction's object is invoked. We also assume that the TxInstance class provides a method `complete`, which is invoked by the thread executing a transaction whenever it completes the execution of the whole set of operations associated with a transaction.

Finally, the manipulation of the versions of the data items stored by SDC occurs via the following primitives: `addSVersions(`T^i_X`)`, which makes the write set of a completed instance T^i_X visible; `removeSVersions(`T^i_X`)`, which removes from SDC the write set of T^i_X; `commitSVersions(`T^i_X`)`, which commits the write set of instance T^i_X by replacing the corresponding existing committed version of any data item.

6.3. Speculative transaction manager

The Speculative Transaction Manager (SXM) intercepts the application level's transactional requests, and interacts with the Optimistic Atomic Broadcast (OAB) service and with the Speculative Concurrency Control (SCC) layer to consistently orchestrate the set of speculatively activated instances.

The pseudo-code for SXM is reported in Fig. 8. SXM uses three sets, namely the *activatedXacts*, *completedXacts* and *committedXacts*, which contain references to T^i_X instances within the corresponding execution stages. To simplify the pseudo-code, we assume that whenever an instance is started or forked it is inserted into the *activatedXacts* set. Further, when an instance is committed, it is removed from the *completedXacts* and *activatedXacts* set and is added to the *committedXacts* set. Furthermore, SXM also keeps track of the sequence of (final) totally ordered transactions in the variable *delivered*.

When the application calls the `invoke` method, SXM marshals a message containing the input parameters specified by the application, generates a unique transaction identifier through the `getNewXactID` primitive (which we denote with *id* in the pseudo-code) and TO-broadcasts the transaction through the OAB service. Next it waits for the commitment of an instance T^i_X in order to return the associated result (retrieved through `getResult`) to the overlying application.

```
class TxInstance {
  int id, specId;
  TxInstance: $T_Y^j = \max\{Y \in$ delivered $\wedge T_Y^j \in$ committedXacts$\}$
  SpeculativePolygraph: SP $= (T_Y^j \circledast \rightarrow T_{id}^{specId})$
  Set of ⟨DataItemID, DataItemVersion, SpeculativePolygraph⟩: RS;
  Set of ⟨DataItemID, Value⟩: WS;
  boolean: speculative;

  method void write(DataItemID I,Value v)
    WS.store(⟨$I, v$⟩); // if I already exists in WS it gets overridden

  method Value read(DataItemID I)
    if (⟨$I, v$⟩ ∈ WS) return $v$;            // if previously written, fetch from local context
    if (⟨$I, v, SP$⟩ ∈ RS) return $v$;        // if previously read, fetch from local context
    if (¬speculative)
      return I.mostRecentCommitted(); // return the latest committed version
    Set of ⟨DataItemVersion, SpeculativePolygraph⟩: versions = getAllVisibleVersions(I);
    fork a new thread running the method shadow(versions);
    ⟨$I^v$, $SP^v$⟩ = versions.removeFirst();    // determine version observed by curr. thread
    exploreNewSerOrder(versions); // observe the other visible versions in alternative threads
    RS.store (⟨$I, I^v, SP^v$⟩);
    SP $= SP^v$;
    return $I^v$;

  method void exploreNewSerOrder(Set of ⟨DataItemVersion, SpeculativePolygraph⟩: versions)
    for each ⟨$I^v$, $SP^v$⟩ ∈ versions do
      fork a new thread running the following code: // actual spawning of sibling tx instances
        RS.store (⟨$I, I^v, SP^v$⟩);
        SP = $SP^v$;
        return $I^v$;

  method void shadow(Set of⟨DataItemVersion, SpeculativePolygraph⟩: exploredVersions)
    Set of⟨DataItemVersion, SpeculativePolygraph⟩: curVers;
    while (true) do
      wait ((curVers=getAllVisibleVersions(I))≠exploredVersions) or ($T_Y^j$ is aborted or committed);
      if ($T_Y^j$ is aborted or committed) exit(); // terminate this thread
      // else fork new threads observing the freshly available versions
      exploreNewSerOrder(curVers \ exploredVersions);
      exploredVersions = exploredVersions ∪ curVers;
...
}
```

Fig. 9. Pseudo-code for the Speculative Concurrency Control (1).

The activities of SXM are also triggered by two other events, namely Opt-deliver and TO-deliver of any transaction T_X. In the former case, SXM invokes the `startSXact` primitive in order to start a new instance T_X^i that will be executed in speculative mode and will be added to *activatedXacts*.

Upon TO-deliver of transaction T_X, SXM checks whether there exists an already completed instance T_X^i that successfully passes the validation phase. An instance is validated by checking whether it accessed a snapshot consistent with the one produced by sequentially executing all the transactions that precede it in the final order (see Section 6.4 for details). If at least one instance T_X^i is successfully validated, SCC commits it and aborts all of its sibling instances. Otherwise, every completed instance, say T_X^j, is aborted. This triggers the cascading abort of any instance exhibiting a (possibly indirect) read-from dependency from T_X^j (see Section 6.4 for further details).

Next, SXM checks whether some instance of T_X is currently active. If not, a new instance T_X^k needs to be activated through either `startNonSXact` or `startSXact`. If instances associated with all the transactions that precede T_X according to the final order have been committed, the new instance can observe the current committed snapshot and does not need to be activated in speculative mode. Otherwise, if some of the preceding transactions has not committed yet, T_X^k is activated in speculative mode.

6.4. Speculative Concurrency Control

SCC is responsible for ensuring the consistent execution of speculative instances, as well as for triggering the re-execution of a transaction according to alternative serialization orders. To this end, SCC associates each instance T_X^i with an object of the TxInstance class that keeps track of the following state information (see Fig. 9): i) *txId* and *specId*, which are set, respectively, to the values X and i when the object associated with T_X^i is created; ii) SP, which stores the speculative polygraph of T_X^i; iii) RS, namely T_X^i's read set, which is organized as an array whose n-th entry records the identity I of the data item read during the n-th read operation, the read version of I, and a copy of T_X^i's speculative polygraph right

```
class TxInstance {
...
method void complete() // invoked when the transaction logic terminates its execution
  completedXacts=completedXacts ∪ $T_{id}^{specId}$;
  if ($id \notin$ delivered)            // not yet final delivered transaction
    TS.addSVersions($T_{id}^{specId}$);   //make new versions visible to any $T_Y^k$ s.t. $T_Y^k.RS \cap WS \neq \emptyset$
  else                                  // already final delivered transaction
    // check if prev. final delivered transactions have been already all committed
    if ($\forall R$ s.t. ($R \in$ delivered $\wedge T_R \rightarrow T_{id}$): $\exists T_R^j \in$ committedXacts)
      if ($\neg$speculative $\vee$ isValid()) // check if this TxInstance can be safely committed
        commit();
        // for each tx final del. after $T_{id}$, check if a committable tx inst. already exists
        TxID: Y = next(delivered, id)
        while (Y $\neq \perp$) do
          if ($\exists T_Y^k \in$ completedXacts s.t. ($\neg T_Y^k$.speculative $\vee$ $T_Y^k$.isValid()))
            $T_Y^k$.commit();
          else
            for each $T_Y^k \in$ completedXacts do $T_Y^k$.abort();
            break;
          Y = next(delivered, Y)
        end while;
      else
        abort();
        // If all tx instances of $T_{id}$ were aborted, activate a new non-speculative one
        if ($\nexists j : T_{id}^j \in$ activatedXacts) startNonSXact(T, p, id);
...
}
```

Fig. 10. Pseudo-code for the Speculative Concurrency Control (2).

before the read operation took place; iv) WS, T_X^i's write set; v) a boolean variable, *speculative*, reflecting if the instance was activated in speculative or non-speculative mode.

Write/read operations. Upon a write operation (see the `write` method in Fig. 9) the instance simply logs the identity of the target data item, as well as its new value, into its local write set.

Concerning read operations (see the `read` method in Fig. 9), if the instance already wrote the same data item for which it is issuing the read, the corresponding value stored in the transaction's write set is returned. Similarly, if the data item has been previously read, the version already observed by that instance is extracted from the read set.

If the data item was not previously written or read by the instance, the "right" visible version needs to be fetched. This depends on whether T_X^i has been activated in speculative or non-speculative mode. As anticipated in Section 6.3, T_X^i is activated in non-speculative mode only if T_X has already been TO-delivered and instances associated with all the transactions preceding T_X in the final order have already been committed. Hence, any read executed by a non-speculative instance can safely return the most recently committed version.

On the other hand, if the read operation is issued by a speculative instance, SCC determines what subset of the available versions is speculatively visible to the reading transaction based on its execution history. This is done through the `getAllVisibleVersions` method (see Fig. 11). One of the versions will be part of the snapshot observed by the current instance. On the other hand, for any other visible version say I^v, T_X^i will fork itself, creating a new child that takes I^v as part of its snapshot and that returns the value corresponding to that version (see the `exploreNewSerOrder` method in Fig. 9). Thus, a number of children may be created, each sibling using a different version of data item I as input. T_X^i also spawns a thread that just remains waiting for additional visible versions of data item I to be subsequently created (see the `shadow` method in Fig. 9). For each new version that is made visible, this thread will fork an additional child. This process gets repeated until the originally reading instance T_X^i is aborted or committed.

Transaction's completion. When T_X^i completes its execution, the method `Complete` (see Fig. 10) is invoked, which immediately adds the instance to the *completedXacts* set. Then, if T_X is already TO-delivered, but there exists some transaction T_Y, that precedes T_X in the final order, for which no instance has been committed yet, the `completed` method simply ends. In this case, the task of attempting to commit T_X^i will be delegated to any T_Y^j that will subsequently enter the complete/commit phase. Conversely, if all transactions T_Y preceding T_X in the final order have instances already committed, and T_X^i is either non-speculative or successfully passes the validation phase, T_X^i is committed. In this case, SCC also attempts to commit any completed instance T_Z^m, whose corresponding transaction T_Z follows T_X in the final order. On the other hand, if T_X has not yet been TO-delivered, T_X^i makes speculatively available its versions through `addSVersions`. These versions will be made visible to instances that have read those data items and are waiting for new versions to appear in order to spawn new siblings.

```
class TxInstance {
...
1:  method Set of ⟨DataItemVersion,SpeculativePolygraph⟩: getAllVisibleVersions(DataItem I)
2:      Set of ⟨DataItemVersion,SpeculativePolygraph⟩: visibleVersions;

        // Returns the set of versions of I that $T_{id}^{specId}$ can observe given its former read operations.
        // For each visible version $I^v$, it returns SP($T_{id}^{specId}$) updated to track the read of $I^v$
3:      forall $I^v \in I$.getAllVersions() do
4:          SpeculativePolygraph: updatedSP = isVisible($I^v$, SP, $T_{id}^{specId}$);
5:          if (updatedSP$\neq\perp$) visibleVersions = visibleVersions $\cup$ {⟨$I^v$, updatedSP⟩};
6:      return visibleVersions;

7:  method void abort()
8:      TS.removeSVersions($T_{id}^{specId}$);
        // Cascading abort of any transaction that depends, possibly transitively, on $T_{id}^{specId}$
9:      forall $T_Y^i \in$ activatedXacts do
10:         forall $\delta \in \mathcal{D}(T_Y^i.SP)$ s.t. isValid($\delta$,$T_Y^i$) do
11:             if (($T_{id}^{specd} \rightarrow T_Y^i$) $\in \delta^*$)
12:                 $T_Y^i$.abort();
13:     activatedXacts = activatedXacts \ { $T_{id}^{specId}$};
14:     completedXacts = completedXacts \ { $T_{id}^{specId}$};

15: method void commit()
16:     TS.commitSVersions($T_{id}^{specId}$);
17:     forall $T_{id}^j \in$ activatedXacts do $T_{id}^j$.abort(); // Abort sibling transactions

18: method boolean isValid()
19:     forall ⟨I, value, ·⟩ $\in$ RS do
20:         if (value $\neq$ I.getCommittedVersion().getValue()) return false;
21:         else return true;
}
```

Fig. 11. Pseudo-code for the Speculative Concurrency Control (3).

Commit/abort/validation. An instance is considered successfully validated iff the values read during its execution, and stored within its RS variable, coincide with the values currently stored by the committed version of the corresponding data items (see the `isValid` method in Fig. 11). The `commit` method marks the data item versions created by the committing instance as the currently committed versions, and then triggers the abort of any of its siblings through the `abort` method. When the latter method is invoked on instance T_X^i, it first removes any data items' versions made available by T_X^i, and then triggers the cascading abort of any other instance, say T_Y^j, that has a (possibly indirect) read-from dependency from the aborting instance (see lines 15-18 in Fig. 11).

6.5. Correctness proof

Theorem 1 *(Global-Consistency). The history of committed instances is one-copy serializable.*

Proof. A process $p_k \in \Pi$ commits an instance T_X^i only if the corresponding transaction T_X has been already TO-delivered and, for any of the preceding transactions within the TO-deliver order, one instance has been committed (see lines 7–8 in Fig. 10 and lines 8–10 in Fig. 8). Hence, by the global agreement and global order properties of Optimistic Atomic Broadcast, it follows that every correct process $p_k \in \Pi$ commits instances according to the same ordered sequence of transaction identifiers $\mathrm{S}_k = \{T_A^*, \ldots, T_L^*, \ldots, T_N^*\}$, where with the notation T_L^* we denote that the l-th instance along the sequence of two distinct processes $p_k, p_{k'} \in \Pi$, must have the same id, namely L, but may have been attributed different *specId* values by p_k and $p_{k'}$. Also, any process that crashes can only commit a prefix of S_k. Hence, one-copy serializability follows from the fact that any instance T_L^i committed by a process $p_k \in \Pi$ is deterministically validated (taking into account the same set of transactions preceding T_L). This ensures that every read that T_L^i issued returned the most recently committed version with respect to the execution of a prefix of length l of the sequence S_k. □

Lemma 1. *Let $\mathcal{R}_{T_X^i} = \{r_1(I_1), \ldots, r_{k-1}(I_{k-1})\}$ be the set of $k-1$ read operations performed by an instance T_X^i at any given point of its execution. The `isVisible` method determines that a version I_k^v of data item I_k is visible by the k-th read of T_X^i if and only if there exists a sequential history $\mathcal{H} = (\sigma, <_{\mathcal{H}})$, where $\sigma \subseteq \Sigma$, in which T_X executes the same sequence of reads in $\mathcal{R}_{T_X^i}$, returning the same sequence of values seen by T_X^i, and I_k^v at its k-th read.*

Proof. We proceed by showing that the `isVisible` method returns all and only the versions that would have been visible by the considered transaction if this had been executed in a speculative, but view-serializable, history. To this end we prove the equivalence between the speculative polygraph of T_X^i, namely $\mathrm{SP}(T_X^i)$, and the polygraph $P(\mathcal{H})$, where $\mathcal{H}$ is a non-speculative history that generates the snapshot read by T_X^i up to its current execution phase. A speculative polygraph $\mathrm{SP}(T_X^i)$ is equivalent to a polygraph $P(\mathcal{H})$, iff for each valid directed graph $\delta = (N, A) \in \mathcal{D}(\mathrm{SP}(T_X^i))$ there exists an acyclic directed graph $\delta' = (N', A') \in \mathcal{D}(P(\mathcal{H}))$.

The rules R.1 and R.2 used to construct $\mathrm{SP}(T_X^i)$ ensure that any instance that is serialized before T_X^i (either directly, or indirectly, e.g., through the merging edge of a speculative bipath) must have merged its speculative polygraph with the one of T_X^i. This implies that for any $\delta \in \mathcal{D}(\mathrm{SP}(T_X^i))$ there exists a one-to-one correspondence with a directed graph $\delta' \in \mathcal{D}(P(\mathcal{H}))$ where P is the polygraph associated with the history $\mathcal{H}$.

Also, since the `isVisible` method considers a data item version I^v speculatively visible to T_X^i only if, by condition C.2, there are no two sibling transactions T_Y^r, T_Y^q preceding T_X^i, the above history $\mathcal{H}$ is guaranteed to contain at most one instance of a given transaction. Hence for any δ for which condition C.2 holds, the corresponding history $\mathcal{H}$ constructed as discussed above, is representative of a non-speculative history.

Note, however, that the directed graph $\delta \in \mathcal{D}(\mathrm{SP}(T_X^i))$ may contain a set S of vertexes not present in its corresponding $\delta' \in \mathcal{D}(P(\mathcal{H}))$, each vertex being associated with an instance T_Y^j, such that:

- it created a version I^v of some data item I for which a read $r(X)$ operation was issued either by T_X^i, or by any transaction T_Z^r preceding $T_X^i \in \delta$, and such that $r(I)$ returned a different version I^b, and
- its speculative polygraph $\mathrm{SP}(T_Y^j)$ prevents to serialize T_Y^j before T_X^i (via a plain edge or through the merging edge of a speculative bipath) because, in the resulting speculative polygraph SP*, every direct graph $\delta \in \mathcal{D}(SP^*)$ would not be valid.

Recalling that, by rule R.2, each time we serialize an instance that creates a data item version I^b after one that reads a different data item version of I, we do so via a simple (i.e., non-expanding) edge of a speculative bipath, it follows that all the vertexes in S must be sink nodes (i.e., they have no outgoing edges). However, since S is composed exclusively of sink vertexes, their presence in $\mathrm{SP}(T_X^i)$ is not influential for the generation of cycles in any directed graph $\delta \in \mathcal{D}(\mathrm{SP}(T_X^i))$. This ensures the equivalence between $\mathrm{SP}(T_X^i)$ and the polygraph $P(\mathcal{H})$ representative of the non-speculative history that generates the snapshot read by T_X^i up to the current execution phase.

Finally, recalling that a history $\mathcal{H}$ is view-serializable if and only if $P(\mathcal{H})$ is acyclic, it follows that the `isVisible` method returns all and only the versions that would have been visible by the considered instance if this had been executed in a speculative, but view-serializable, history. □

Theorem 2 *(Local-Consistency). Every instance $T_X^i \in \Sigma'$ sees a snapshot producible by a serial execution of instances associated with a subset of the transactions in Σ.*

Proof. Derives directly from Lemma 1 and from the fact that a read operation of a speculative transaction returns a data item's version only if this is deemed visible by the `isVisible` method. □

Theorem 3 *(Non-Redundancy). No two sibling instances in Σ' observe the same snapshot.*

Proof. In this proof we denote with generatorOf(T_X^j) the instance T_X^i, if any, that caused the activation (via fork) of the instance T_X^j, and with generatorOf* the transitive closure of the generatorOf relation.

Let us now consider any two sibling instances T_X^j and T_X^k. The following two cases are possible:

1. generatorOf(T_X^k) $= T_X^j$. In this case, T_X^j forked T_X^k following a read operation on some data item X. In this case, T_X^j and T_X^k necessarily returned two different versions of data item X, therefore the two instances necessarily observe distinct snapshots. Clearly, the same considerations apply if generatorOf(T_X^j) $= T_X^k$. Hence the claim follows.
2. Neither generatorOf(T_X^k) $= T_X^j$, nor generatorOf(T_X^j) $= T_X^k$, but $T_X^j \in$ generatorOf*(T_X^k) (or, alternatively, $T_X^k \in$ generatorOf*(T_X^j)).

   In this case it is possible to determine a sequence of sibling instances $(T_X^j, T_X^{j^1}, \ldots, T_X^{j^n}, T_X^k)$, where:

   $$T_X^j = \mathrm{generatorOf}\big(T_X^{j^1}\big), \qquad \ldots, \qquad T_X^{j^{n-1}} = \mathrm{generatorOf}\big(T_X^{j^n}\big), \qquad T_X^{j^n} = \mathrm{generatorOf}\big(T_X^k\big)$$

   such that each instance in the n-th position of the sequence forks the $(n+1)$-th sibling transaction of the sequence. It follows that the snapshots seen by T_X^k and T_X^j differ at least for the result of one read operation, namely the one which caused T_X^j to fork $T_X^{j^1}$. The same considerations apply by switching the roles of T_i^k and T_i^j. Hence the claim follows.

3. $T^k_X \notin \text{generatorOf}^*(T^j_X)$ and $T^j_X \notin \text{generatorOf}^*(T^k_X)$. In this case, the two siblings must have a set of "common ancestors", denoted as $\text{Anc}(T^j_X, T^k_X)$, such that:

$$\forall T^i_X \in \text{Anc}\big(T^j_X, T^k_X\big) : T^i_X \in \big(\text{generatorOf}^*\big(T^j_X\big) \cap \text{generatorOf}^*\big(T^k_X\big)\big).$$

In this case it is possible to order the instances $T^i_X \in \text{Anc}(T^j_X, T^k_X)$ according to the sequence $(T^{i^0}_X, T^{i^1}_X, \ldots, T^{i^o}_X, \ldots, T^{i^n}_X)$, where:

$$T^{i^o}_X = \text{generatorOf}\big(T^{i^{o+1}}_X\big)$$

such that each instance in the n-th position of the sequence forks the $(n+1)$-th sibling transaction of the sequence. $T^{i^n}_X$ is therefore the common ancestor of T^k_X and T^j_X that shares the longest common sequence of operations with T^k_X and T^j_X. It follows that the snapshots seen by T^k_X and T^j_X differ at least for the result of one read operation, namely the one which caused $T^{i^n}_X$ to fork $T^{i^{j'}}_X$ and $T^{i^{k'}}_X$, where $T^{i^{j'}}_X \in \text{generatorOf}^*(T^j_X)$ and $T^{i^{k'}}_X \in \text{generatorOf}^*(T^k_X)$. The same considerations apply by switching the roles of T^k_i and T^j_i. Hence the claim follows. □

Theorem 4 *(Completeness). If the system is quiescent then for every permutation $\sigma \in \pi(\Sigma)$ and for every transaction $T_X \in \Sigma$, there eventually exists an instance $T^i_X \in \Sigma'$ that executes on (i.e., observes) the same snapshot that would have been produced by sequentially executing instances associated with all the transactions preceding T_X in σ.*

Proof. Let us denote with T_Y the last committed transaction by the STR protocol at given point in time on any pocess p_k. Given that by the protocol structure no transaction can ever be committed before it is TO-delivered, and given that the system is quiescent, eventually, T_Y does not vary over time. Also, by the definition of Σ and by the system's quiescence, $T_Y \notin \Sigma$.

Let us now consider whichever permutation σ of the transactions belonging to Σ. The proof is carried out by induction on the number n of transactions preceding T_X in σ.

$(n = 0)$ In this case no transaction belonging to Σ precedes T_X according to a sequential execution, and T_X sees the snapshot produced up to the commit of transaction T_Y. Hence, we have to show that, eventually, an instance T^i_X completes its execution by reading the snapshot produced by T_Y and by the transactions that committed before T_Y. By the structure of the STR protocol, such a snapshot corresponds to the snapshot upon the completion of T_Y. Recall that, always by the structure of the STR protocol, at least one instance T^i_X eventually exists since T_X has been Opt-delivered (given that it belongs to Σ). There are two cases:

1. The instance T^i_X that is activated upon the Opt-deliver of T_X performs its first read after the commit of T_Y and of every transaction T_* preceding T_Y in the TO-deliver order. At this time there exists for each data item a version created by T_Y or by any transaction that committed before T_Y. Given that the history in which T^i_X reads exclusively the committed versions of the data items is serializable, upon the first read operation by T^i_X on whichever data item I, by Theorem 2 and by Lemma 1, the `isVisible` method makes the committed version of I visible. Assume, without loss of generality, that T^i_X exactly returns the committed version of I, while other uncommitted versions of I, if any, are returned by sibling transactions T^j_X forked upon such a read operation. Since the committed version of a data item always exists, we can apply the same reasoning upon the second, and any other subsequent read operation on whichever data item by T^i_X. Hence, T^i_X actually sees the committed snapshot while performing all its read operations. This corresponds to the snapshot produced be executing sequentially all the transactions preceding $T_X \in \sigma$, hence the claim follows.
2. The transaction T^i_X that is activated upon the Opt-deliver of T_X performs its first read before T_Y, or before some transaction T_* preceding T_Y in the final TO-deliver order has been committed. In this case, if T^i_X does not read any of the data items written by T_X or by T_*, by the same reasoning of the above case, T^i_X will complete observing a committed snapshot equivalent to the one generated by executing sequentially all the transactions up to T_Y according to the TO-deliver order. If, conversely, T^i_X reads any of the data items written by T_Y or by T_*, by the STR protocol structure, T^i_X will be eventually aborted upon T_Y's or T_*'s commit, and a new instance T^i_X will be restarted. Hence, eventually an instance T^j_X will be activated after the commit of instances associated with all the transactions that precede T_X in the TO-deliver order, thus leading us to fall in case 1 above. Hence, in any of the discussed cases, the claim follows.

$(n = m)$ Let us consider the subsequence of m transactions that immediately precede T_X in σ. Let us denote with $T_A, T_B, \ldots, T_M$ such a subsequence of transactions. By the inductive assumption, it follows that there is a time t after which, $\forall L \in [A \ldots M]$, a completed instance T^*_L exists on node p_k that has executed observing the snapshot generated by the execution of the sequential transaction history $T_Y, T_A, T_B, \ldots, T_{prev(L)}$. We prove that eventually an instance T^i_X

completes its execution after observing the snapshot generated by the sequential execution of the transaction history $\mathcal{H} = \{T_Y, T_A^*, \ldots, T_M^*\}$.

Let t be the earliest time in which an instance T_X^i performs its first read operation on process p_k and assume, without loss of generality, that the read operation is on data item I. If at time t, all the transactions belonging to the history $\mathcal{H}$ are already completed, then, by Lemma 1, T_X^i will be able to see the version of I created by the last transaction in $\mathcal{H}$ that wrote I. On the other hand, if at time t on process p_k not all the instances in $\mathcal{H}$ are completed, it might happen that an instance $T_K^* \in \mathcal{H}$ generates a new version I^k of the data item I, and makes it available with SDC only after T_X^i issues the read on I. In this case, the shadow thread that T_X^i forked when issuing the read on I will fork a new sibling instance of T_X^i that will read I^k. In both cases, there exists an instance T_X^j that eventually reads the data item version created by the most recent transaction in $\mathcal{H}$ that wrote I. By re-applying the same reasoning to the subsequent reads of T_X^q, we have that eventually there is an instance T_X^w on process p_k that completes by observing the snapshot produced by the execution of the sequential history $\mathcal{H}$. Hence the claim follows. $\square$

6.6. *Dealing with read-only transactions*

As in other OAB-based transactional replication solutions, when employing the STR protocol presented in this paper, read-only transactions can be processed locally by each replica. To this end, a simple approach would consist in running read-only transactions in a purely optimistic fashion, letting them always read the most recently committed data items' versions and relying either on an a-posteriori (at commit time), rather than eager (at each read) validation. This solution requires maintaining at any time only a single committed version of the data items, albeit at the cost of incurring aborts of read-only transactions.

An alternative solution, exhibiting opposite benefits and drawbacks, would consist in ensuring that a read-only transaction is always able to observe the snapshot created by the transactions already committed when it started. This would shelter read-only transactions from the chance of aborting, though forcing the SDC to maintain multiple committed versions of each data item.

7. Simulation results

The formulation of the STR problem, as expressed in Section 4, ensures the ability to explore the minimum set of distinct serialization orders leading to different outcomes, which guarantees at the same time the complete "coverage" of any arbitrary serialization order eventually defined by the final delivery sequence established by the OAB service.

The completeness property has been defined in such a way to capture the behavior of STR protocols when the system is quiescent, namely when the OAB service stops Opt-delivering and TO-delivering transactions. This has been done to provide a clear cut specification, which rules out any dependence on timing/schedule factors such as:

– the time interval between Opt-deliver and TO-deliver indications for each transaction;
– the transaction execution latency and the schedule of its operations;
– the transaction data access pattern.

In real settings, these factors have an influence on the amount of wall-clock-time available for speculation, as well as on the timing of conflict materialization among concurrent transactions, which is expected to influence any STR protocol while determining if/when to spawn speculative instances. As an example, the access to a data item by an instance as its first operation would allow for anticipating conflict detection with already completed instances, when compared to accessing the same data item in its final execution phase. This would lead to anticipate or delay the spawning of speculative instances depending on the specific access pattern, and hence the associated exploration of the different serialization orders. On the other hand, the schedule of transactions' operations will determine whether, upon accessing that data item, those conflicting transactions have already reached the complete stage. This further influences the time instant when conflicts would be materialized, thus determining again the actual speculative transactions to be spawned (and their activation instant) in order to follow different serialization orders.

Overall, depending on the timing of conflict detections, a protocol would be (or not be) allowed to speculate along a specific serialization path prior to the finalization of the OAB service. More generally, the combination of the above three factors could limit the number of (distinct) serialization orders actually covered by the protocol when considering realistic (i.e., non-quiescent) settings.

To cope with the above considerations, we performed an extensive simulation study with a twofold aim:

a) Quantifying the additional work required by an STR protocol in order to ensure the completeness property, namely the work to be done to produce all and only the transaction serialization orders generating different snapshots, in the context of system quiescence;
b) Showing performance results for a realistic scenario characterized by the absence of quiescence, namely where the OAB service works by delivering messages according to delays and timing that are typical of an operating Group Communication Service (GCS).

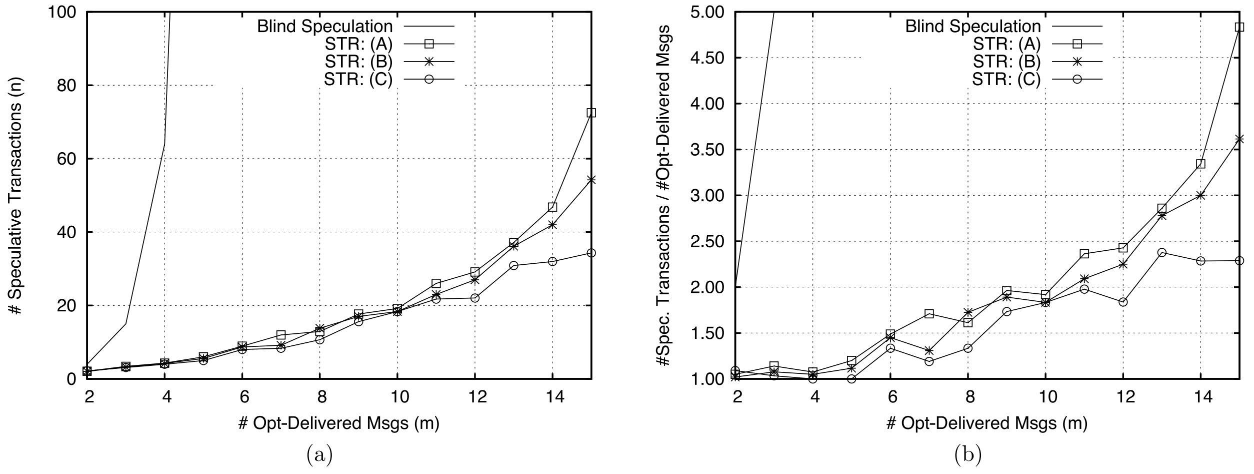


Fig. 12. #Speculative transactions vs #Opt-delivered messages.

In order to reproduce representative transaction execution dynamics we developed a trace-based simulator that we fed with a trace containing the log of data accessed by transactions in a real operating environment. The simulation study is framed within the context of Transactional Memory (TM) systems, which represent an architectural support aimed at simplifying parallel programming by replacing conventional lock-based synchronization mechanisms with the abstraction of atomic transaction.

The choice towards TMs is also motivated by the fact that, as already hinted in Section 1, these systems are natural candidates for the adoption of speculative transactional replication schemes since they exhibit reduced ratio between transaction processing time and coordination (i.e., OAB) latency.

The actual data access pattern of the transactions running within the simulation is determined by using traces of the popular RedBlack Tree benchmark for TMs introduced in [11].

Since read-only transactions can be handled within the replicated system via solutions that are aside of the proposed speculative replication protocol (see Section 6.6), in our simulation experiments we consider a mix containing exclusively update transactions performing, with equal probability, either an insertion or a delete of an entry of the RedBlack Tree.

7.1. Part-A

In this part of the simulation study our goal is to quantify the number of speculative instances of transactions generated by the STR protocol in case of system quiescence. This is an idealized scenario that helps reasoning on protocol properties from a theoretically oriented perspective.

Clearly, given a number m of optimistically delivered transactions, the minimum number n of speculative transactions that have to be spawned to ensure complete coverage is a function of the actual conflict patterns among transactions. Moreover, the value of n can be considered as an indicator of the protocol complexity because it has a direct impact on the computational resources that need to be employed both to carry out the speculative execution and to handle any housekeeping data structure, whose size may vary as a function of n.

In this part of the evaluation, we fed the simulator with: (i) a number m of optimistically delivered transactions (that are not eventually finally delivered due to system quiescence), which is treated as an independent parameter, and (ii) traces of those m transactions accessing a specific data set. Once started up, the simulator runs our STR protocol by activating the whole (and minimum) set of n instances necessary to ensure the complete coverage of the (conflict dependent) speculative serialization orders related to the different delivery orders of the m optimistically delivered messages.

Given that the aim of the this part of the evaluation is to determine the rate of growth of n vs m by observing the protocol evolution in a phase which mimics quiescence, the timing of actions from the traces becomes not relevant. Hence, each read/write operation is modeled as having a fixed fictitious cost of one simulation time unit.

Regarding the configuration of the accessed data set, the RedBlack Tree benchmark is characterized by two parameters, namely the initial size, *init*, and the maximum size, *max*, of the tree, whose tuning allows to vary the probability of conflict among transactions (the larger the size of the tree, the lower the conflict probability among transactions). To compare the number of speculative transactions generated for reaching completeness in scenarios representative of different conflict probabilities, we consider three different settings for these two parameters: (A) $init = 32$, $max = 1024$, (B) $init = 128$, $max = 2048$, and (C) $init = 1280$, $max = 20\,480$.

The plots in Fig. 12(a) contrast the number n of speculative transactions activated by our STR protocol, with the number of transactions that would have been activated by a naive speculative approach that does not take into account the actual transaction conflict relations developed at run-time. As recalled in Section 4, such a number of speculative transactions grows factorially with m. In the plots, we report the value of n while varying the number m of optimistically delivered

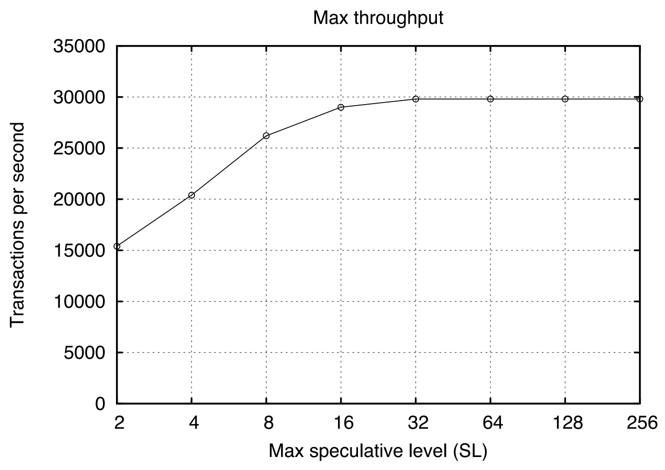


Fig. 13. Max throughput.

messages in the interval between 2 and 15. For the maximum selected value of m, the observed conflict probabilities associated with the above configurations are on the order of 45% (case A), 30% (case B) and 20% (case C).

The plots clearly highlight the effectiveness of STR protocols in reducing the number of serialization orders that need to be speculatively explored in every considered scenario, which maps onto non-redundancy of the speculative exploration. When m is equal to 10, for example, n is equal to around 10 millions for the blind speculative approach, whereas it varies between 15.6 and 17.7 for our STR protocol. Fig. 12(b) allows us to visualize these data from a different perspective. By reporting the ratio between n and m, we can quantify the amount of speculative instances that the system should process in order to ensure completeness, with respect to a non-speculative system which exactly processes a single transaction for every Opt-delivered message (e.g., by processing it only after the corresponding TO-deliver). The plots show that, as long as m is less than 5, such a ratio is flat around the value 1 for the STR protocol. On the other hand, if m grows up to 10, with the STR protocol it suffices, on average, to execute each transaction in no more than two different serialization orders. Finally, with $m = 15$, depending on the considered contention scenario, it would be necessary to explore, on average, between 2.5 and 5 different serialization orders per transaction.

As previously noted, non-speculatively replicated transactional systems wait for the completion of the distributed coordination scheme before activating transaction processing activities. Hence, beyond impacting the latency of transaction completion (and related result delivery) they may also suffer from CPU under-utilization [8], especially in application exposing fine-grained transactions or in massively parallel architectures (whose popularity has increased significantly of late). By the above simulation results, the proposed speculative replication approach can lead to exploit such idle computational resources in a fruitful manner since the reduced amount of speculatively explored serialization orders (compared to any blind scheme not accounting for actual conflicts among transactions) would allow avoiding thrashing.

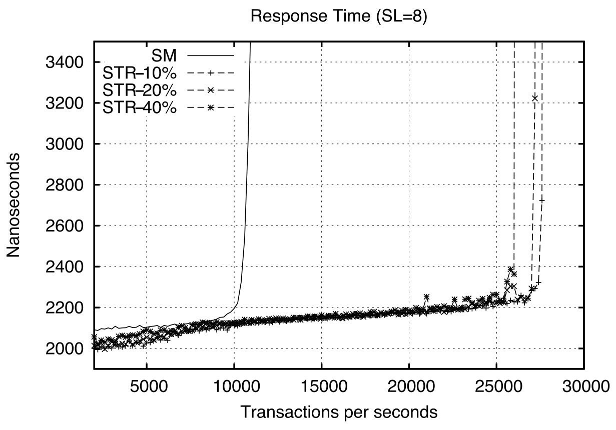
7.2. Part-B

In this second part of the evaluation, we show performance results for the proposed STR protocol in absence of quiescence, namely when both Opt-deliveries and TO-deliveries occur according to a realistic timing, as provided by typical OAB implementations. Clearly, the currently analyzed scenario differs from the one in Part-A due to the fact that, being TO-deliveries not suspended, the time available for speculating on the execution of already Opt-delivered transactions is not unbounded. Furthermore, the materialization of conflicts among concurrent transactions, which is the factor driving speculation, depends on the actual interleaving of transactional operations. Hence, in this settings the simulator explicitly considers transactions' duration and the timing of the accesses to data as specified by the execution traces of the used benchmark.

We also introduced a new independent parameter SL (Speculative Level) in the simulation study, defined as the maximum number of speculative instances of the same transaction that are allowed to be generated by the STR protocol.

Our first aim is to show the maximum throughput obtained by the protocol, while varying SL, when running simulations on the RB-Tree benchmark configured in a way to generate the highest level of conflict ($init = 32$, $max = 1024$). In order to perform the experiment, we changed the simulator configuration by activating the TO-deliver handler and setting the expected latency of the Group Communication System at 500 microseconds for the Opt-deliver, and at 2 milliseconds for the TO-deliver. These values have been selected on the basis of experimental measures obtained running the Appia GCS Toolkit [24] on a cluster of 4 quad-core machines (2.40GHz – 8GB RAM) connected via a switched Gigabit Ethernet.

The results shown in Fig. 13 highlight how the throughput by the STR protocol, in absence of quiescence, tends to stall while increasing SL. This result is not obvious given that, when SL increases, the probability to spawn the speculative transactions actually running on the serialization order compliant with the TO-deliver order should be higher. However,



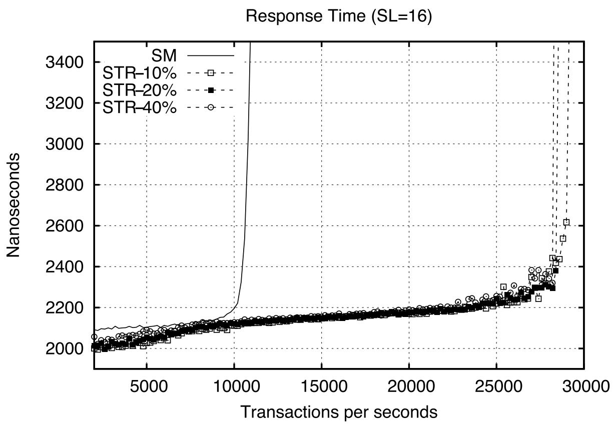


Fig. 14. Response time with $SL = 8$ (left) and $SL = 16$ (right).

in absence of quiescence, the performance bottleneck is essentially related to the limited time for speculating and to the timing according to which the conflicts among transactions drive the generation of alternative serialization orders.

By the plot, the throughput values corresponding to $SL = 8$ and $SL = 16$ represent, respectively, the 75% and the 95% of the maximum throughput reachable by the STR protocol. Leveraging on this result, in the rest of the study we show a detailed performance evaluation of STR when bounding SL at the values of 8 and 16. In pragmatical settings, these values can be expected to give rise to good trade-offs between the actual performance and the housekeeping cost of the protocol, by likely leading to scenarios where such a cost results limited.[4] Based on the above observations, in the following we focus on configurations with $SL = 8$ and $SL = 16$, and choose not to explicitly model any protocol's housekeeping costs (which are assumed to be negligible for small values of SL).

We simulated the STR protocol under various configurations of the OAB service, which have been achieved by changing the percentage of reordering between optimistic and final delivery events. This allows testing the effectiveness of the STR protocol in heterogeneous scenarios, in terms of distributed coordination dynamics across the replicas. In particular, three reordering percentages have been selected: 10%, 20% and 40%.

Fig. 14 reports the response time of the proposed STR protocol with $SL = 8$ and $SL = 16$, as a function of the transaction arrival rate. We use as baseline for this evaluation, a classic, i.e., non-speculative, state machine replication algorithm, which we denote as SM. For both settings of the SL parameter, the performance achievable by STR is represented via three different curves, each one capturing a different reordering percentage in the OAB service.

The gain of STR with respect to SM is higher when $SL = 16$, since, an increase of this parameter leads to an increase of the probability of speculatively processing transactions that are serialized correctly according to the eventually established TO-deliver order. Specifically, the speed-up of STR over SM when $SL = 16$ is about 3x (see Fig. 14 – right side), while it reduces to about 2.5x when setting $SL = 8$ (see Fig. 14 – left side). As the last note, the performances gains achievable by STR tend to slightly decrease in scenarios characterized by larger percentages of message reordering. In low message reordering scenarios, in fact, upon activation of the first speculative transaction associated with a specific Opt-deliver event, there is a higher chance that previously (final and optimistically) delivered conflicting transactions have already reached the complete stage, thus giving rise to a speculative execution characterized by serialization order compliant with the optimistic delivery order. This favors STR performance in scenarios with reduced reordering probability within OAB.

Acknowledgments

This work extends earlier results published in [25,26], and was partially supported by the ARISTOS (PTDC/EIA-EIA/102496/2008) project, by the COST Action IC1001 (Euro-TM), by the FP7 Project Cloud-TM (257784), and by national funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013.

References

[1] F.B. Schneider, Replication Management Using the State-Machine Approach, ACM Press, Addison–Wesley Publishing Co., 1993.
[2] D. Powell (Ed.), Special Issue on Group Communication, vol. 39, ACM, 1996.
[3] D. Agrawal, G. Alonso, A.E. Abbadi, I. Stanoi, Exploiting atomic broadcast in replicated databases (extended abstract), in: Proc. 3rd International Euro-Par Conference on Parallel Processing (Euro-Par), Springer-Verlag, 1997, pp. 496–503.
[4] B. Kemme, G. Alonso, A suite of database replication protocols based on group communication primitives, in: Proc. 18th International Conference on Distributed Computing Systems, ICDCS, IEEE Computer Society, 1998, p. 156.

[4] This should be true also when considering that housekeeping data structures and related access/manipulation tasks could even exploit parallelization techniques, such as querying the polygraphs in parallel.

[5] B. Kemme, F. Pedone, G. Alonso, A. Schiper, Processing transactions over optimistic atomic broadcast protocols, in: Proc. of the 19th IEEE International Conference on Distributed Computing Systems, ICDCS, IEEE Computer Society, 1999, p. 424.
[6] F. Pedone, A. Schiper, Optimistic atomic broadcast: a pragmatic viewpoint, Theor. Comput. Sci. 291 (1) (2003) 79–101, http://dx.doi.org/10.1016/S0304-3975(01)00397-8.
[7] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, G. Alonso, Middle-r: Consistent database replication at the middleware level, ACM Trans. Comput. Syst. 23 (4) (2005) 375–423, http://dx.doi.org/10.1145/1113574.1113576.
[8] R. Palmieri, F. Quaglia, P. Romano, N. Carvalho, Evaluating database-oriented replication schemes in software transactional memory systems, in: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium, 2010, pp. 1–8.
[9] J. Mocito, A. Respício, L. Rodrigues, On statistically estimated optimistic delivery in wide-area total order protocols, in: Proc. of the 12th Pacific Rim International Symposium on Dependable Computing, PRDC, 2006, pp. 202–209.
[10] P. Romano, N. Carvalho, L. Rodrigues, Towards distributed software transactional memory systems, in: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS, ACM, New York, NY, USA, 2008, pp. 4:1–4:4.
[11] M. Herlihy, V. Luchangco, M. Moir, A flexible framework for implementing software transactional memory, SIGPLAN Not. 41 (10) (2006) 253–262, http://dx.doi.org/10.1145/1167515.1167495.
[12] F. Pedone, R. Guerraoui, A. Schiper, The database state machine approach, Distributed and Parallel Databases 14 (1) (2003) 71–98, http://dx.doi.org/10.1023/A:1022887812188.
[13] A. Brito, C. Fetzer, P. Felber, Multithreading-enabled active replication for event stream processing operators, in: Proc. of the 28th IEEE International Symposium on Reliable Distributed Systems, SRDS, IEEE Computer Society, Washington, DC, USA, 2009, pp. 22–31.
[14] J. Gray, P. Helland, P. O'Neil, D. Shasha, The dangers of replication and a solution, in: Proc. of the Conference on the Management of Data, SIGMOD, ACM, 1996, pp. 173–182.
[15] A. Bestavros, S. Braoudakis, Value-cognizant speculative concurrency control, in: Proc. of the 21st International Conference on Very Large Data Bases, VLDB, Morgan Kaufmann, 1995, pp. 122–133.
[16] P.K. Reddy, M. Kitsuregawa, Speculative locking protocols to improve performance for distributed database systems, IEEE Trans. on Knowl. and Data Eng. 16 (2) (2004) 154–169, http://dx.doi.org/10.1109/TKDE.2004.1269595.
[17] R. Guerraoui, L. Rodrigues, Introduction to Reliable Distributed Programming, Springer, 2006.
[18] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison–Wesley, 1987.
[19] D. Imbs, J.R. de Mendivil, M. Raynal, Brief announcement: virtual world consistency: a new condition for stm systems, in: Proc. 28th ACM Symposium on Principles of Distributed Computing, PODC, ACM, New York, NY, USA, 2009, pp. 280–281.
[20] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, ACM, New York, NY, USA, 2008, pp. 175–184.
[21] R. Guerraoui, T.A. Henzinger, V. Singh, Permissiveness in transactional memories, in: Proc. 22nd International Symposium on Distributed Computing, DISC, 2008, pp. 305–319.
[22] C.H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653, http://dx.doi.org/10.1145/322154.322158.
[23] M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, Software transactional memory for dynamic-sized data structures, in: Proc. of 22nd ACM Symposium on Principles of Distributed Computing (PODC), ACM, 2003, pp. 92–101.
[24] H. Miranda, A. Pinto, L. Rodrigues, Appia, a flexible protocol kernel supporting multiple coordinated channels, in: Proc. 21st International Conference on Distributed Computing Systems, ICDCS, IEEE, 2001, pp. 707–710.
[25] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues, Brief announcement: on speculative replication of transactional systems, in: F.M. auf der Heide, C.A. Phillips (Eds.), Proc. 25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA, ACM, 2010, pp. 69–71.
[26] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues, An optimal speculative transactional replication protocol, in: Proc. 8th International Symposium on Parallel and Distributed Processing with Applications, ISPA, IEEE, 2010, pp. 449–457.