An Optimal Speculative Transactional Replication Protocol

Paolo Romano*, Roberto Palmieri†, Francesco Quaglia†, Nuno Carvalho*, Luis Rodrigues*
INESC-ID (*), Sapienza Rome University (†)

***Abstract*—In this paper we investigate the problem of speculative processing in a replicated transactional system layered on top of an optimistic atomic broadcast service. We consider a realistic model in which transactions' read/write sets are not known a-priori, and transactions' data access patterns may vary depending on the observed snapshot. We formalize a set of correctness and optimality properties aimed at ensuring that transactions are not activated on inconsistent snapshots, as well as the minimality and completeness of the set of explored serialization orders. Finally, an optimal speculative transaction replication protocol is presented.**

I. 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 total order for the execution of the transactions. This is typically achieved by relying on some form of non-blocking distributed consensus, such as Atomic Broadcast [2] (AB), before activating a transaction [3], [4].

Given that the latency of AB can seriously penalize the performance of a replicated system, it is important to design strategies to mitigate its impact. One of such strategies is to overlap 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 optimistic message delivery, 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 guess. Otherwise optimistically activated transactions may access inconsistent snapshots and be forced to rollback. 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, sparing from the risks of propagating the results generated by optimistically delivered transactions. On the other hand, it can constrain the achievable degree of parallelism in the processing of optimistically delivered transactions. Furthermore, existing approaches rely on the a-priori knowledge of both read and write sets associated with incoming transactions in order to associate transactions with conflict classes at the time of their optimistic delivery, i.e. prior to their execution. This requirement raises the non-trivial problem of systematically predicting data access patterns, which may force to significant over-estimations of the likelihood of transaction conflicts (compared to actual ones), 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 [8], or in systems where the ratio between the communication delay and the computation granularity is very large, such as Transactional Memories [9]. In this paper, we address these challenges by exploring the use of speculative transaction processing, motivated by the widespread use of multi-core parallel machines whose computational power can be fully unleashed using such an approach.

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 snapshots 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 and optimality 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.

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

This work was partially supported by the ARISTOS (PTDC/EIA-EIA/102496/2008) project and by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

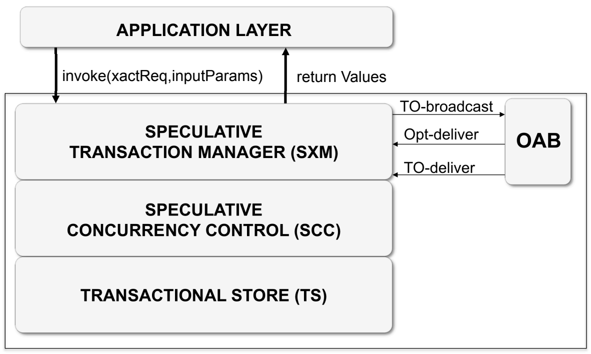
APPLICATION LAYER
invoke(xactReq,inputParams)
return Values
TO-broadcast
Opt-deliver
TO-deliver
OAB
SPECULATIVE TRANSACTION MANAGER (SXM)
SPECULATIVE CONCURRENCY CONTROL (SCC)
TRANSACTIONAL STORE (TS)

Fig. 1. Software Architecture of Each Process

set of explored speculative serialization orders.

The remainder of this paper is structured as follows. In Section II we discuss related work. Section III illustrates the system model. In Section IV we formalize correctness and optimality properties for STR. Section V presents an optimal STR protocol. Finally, Section VI concludes the paper.

II. Related Work

Atomic Broadcast (AB) based replication protocols for transactional systems [3], [4], [5], [11] achieve a global transaction serialization order (across all the replicas), in a non-blocking fashion, without incurring the scalability problems affecting classical eager replication mechanisms based on distributed locking and atomic commit protocols [12]. Our STR protocol builds on Optimistic Atomic Broadcast (OAB) [5], [6] and, compared with the aforementioned results, takes a more aggressive optimistic approach by speculatively exploring the minimum set of serialization orders in which optimistically delivered transactions observe distinct snapshots of the transactional system's state. Further, unlike schemes such as [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 investigated in [13] and [14]. The work in [13] 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 speculative orders. The solution in [14] targets distributed databases relying on distributed locking and atomic commit for transaction validation. Further, it restrains the transaction execution model, making the assumption that each data item can be written by a transaction at most once.

III. System Model

We consider a classical asynchronous distributed system model [15] 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 implement an OAB service ([1]) providing the following interface: *TO-broadcast*(m), which allows broadcasting 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. For convenience of exposition, we also assume that the OAB service provides two additional primitives *TO-DeliveredMsgs()* and *Opt-DeliveredMsgs()*, returning the totally ordered sequences of TO-delivered and Opt-delivered messages, respectively. The OAB service adheres to the following properties [6]:

- **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.

The diagram in Figure 1 shows the software architecture of each process $p_i \in \Pi$. Applications generate transactions by calling the `invoke` method of the local Speculative Transaction Manager (SXM), specifying the business logic to be executed (e.g. the name of a DBMS stored procedure or of a method in a transactional memory) and the corresponding input parameters (if any). The SXM is responsible of (i) propagating (through the OAB service) the transactional request across the set of replicated processes, (ii) executing the transactional logic on the underlying Speculative Transactional Store (STS), and (iii) returning the corresponding result to the user-level application.

We assume that each data item X maintained by the STS is associated with a set of versions $\{X^1, \ldots, X^n\}$, where each version X^i stores (i) the data item value and (ii) the identity of the creating transaction. A single version of X can be committed at any time. Uncommitted versions residing in STS are reflections of speculative computations, and are used to propagate updates along chains of speculated serialization orders. The STS 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 conventional DBMSs).

As depicted in Figure 1, the interactions between the SXM and the STS are mediated by the Speculative Concurrency Control (SCC) layer, which externalizes a classical interface

[1]To this end, for instance, it is enough to assume the availability of an eventually perfect failure detector and the correctness of a majority of processes [15].

to trigger read/write operations on the data items, as well as to commit/abort transactions. The SCC can additionally trigger the re-spawn and the forking of a (not yet committed) transaction T, in order to cope with the speculative construction of alternative serialization orders in which T observes different states of the transactional system. In order to univocally identify multiple, speculatively executed, instances of a same transaction T_i, the SCC relies on an additional identifier referred to as $specId$. For the sake of brevity, we will use the notation T_i^j to refer to the instance of a transaction T_i having $specId = j$, and will refer speculative instances of a same transaction (say T_i^j and T_i^k) to as *sibling*.

In the following we shall consider only update transactions, and delay the discussion on how to extend the proposed STR protocol to account for read-only transactions in Section V-D. Accordingly, we adopt a classical model where a transaction T consists of a sequence of operations, each one being either a read or a write operation 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 a-priori known. Conversely, we assume that the transaction data access pattern can vary depending on the current state of the underlying transactional store. More precisely, we assume that the transactional business logic is *snapshot deterministic* in the sense that if a same transaction is activated multiple times, the sequence of read/write operations it executes does not change unless the return value of any of its reads changes. In other words, if whichever instance of a transaction T always sees a snapshot S, defined as the set of values returned by all its read operations, then it behaves deterministically by always executing a same set of read/write operations. On the other hand, if instances of a transaction T are activated several times on different snapshots, they may generate different sequences of operations.

With no loss of generality, we assume the existence of a method `complete`(), used for explicitly notifying the SCC about the completion of a transaction's execution. Also, we assume that the SXM can retrieve the result generated by a transaction through the method `getResult`().

The manipulation of the versions of a data item occurs via the following methods provided by the STS: `addSVersions(`T_i^j`)`, which makes the write set of a completed transaction T_i^j visible; `removeSVersions(`T_i^j`)`, which removes from the STS the write set of T_i^j; `commitSVersions(`T_i^j`)`, which commits the write set of transaction T_i^j by replacing the corresponding existing committed version of any data item.

IV. Problem Formalization

From the perspective of the replicated transactional system as a whole, our target correctness criteria is classic 1-copy serializability [16], which ensures that a transaction execution history $\mathcal{H}$ across the whole set of replicated processes Π is equivalent to a serial transaction execution history on a non-replicated system. More specifically, we are interested in *view serializability* [16], [17] defined as a property of $\mathcal{H}$ such that, for any prefix $\mathcal{H}'$ of $\mathcal{H}$, its committed projection $C(\mathcal{H}')$ (obtained from $\mathcal{H}'$ by deleting all operations not belonging to transactions committed in $\mathcal{H}'$) is *view equivalent* to some serial history. Roughly speaking, view equivalence of two histories $\mathcal{H}_1$ and $\mathcal{H}_2$ is defined as the property by which, for any data item X: (i) if T_i reads X from T_j in $\mathcal{H}_1$, then T_i reads X from T_j also in $\mathcal{H}_2$ and (ii) for each data item X, if X^w is the final written value of X by T_i in $\mathcal{H}_1$, then it is also the final written value of X by T_i in $\mathcal{H}_2$.

We now introduce the notion of optimality for a speculatively replicated transactional system. This is done by formalizing 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 transactional store. Let $\Sigma = \{T_1, \ldots, T_n\}$ be the set of Opt-delivered, but not yet TO-delivered, transactions, and denote with $\Sigma' = \{T_1^1, \ldots, T_1^k, \ldots, T_n^1, \ldots, T_n^m\}$ the set of the corresponding speculative transactions that have run to completion, namely that have fully executed their sequence of read and write operations but have not been committed yet. We say that a Speculative Transactional Replication (STR) protocol is optimal if it guarantees the following properties:

- **Consistency:** *the history of execution of each speculative transaction is view serializable.*
- **Non-redundancy:** *no two sibling transactions observe the same snapshot.*
- **Completeness:** *if the system is quiescent, i.e. if the OAB service stops Opt-delivering and TO-delivering transactions, then for any possible permutation of Σ, say $\pi(\Sigma)$, there eventually exists a speculative transaction $T_i^j \in \Sigma'$ that has executed on (i.e. observed) the same snapshot that would have been produced by sequentially executing all the transactions T_k preceding T_i in $\pi(\Sigma)$.*

The non-redundancy property of an STR protocol 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 final 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 the corresponding 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 to the application.

V. An Optimal STR Protocol

In our STR protocol, each replica immediately starts processing transactions as soon as these are optimistically delivered by the OAB service. The issue of generating a speculated set of different serialization orders is tackled by the SCC layer, which dynamically tracks the dependencies developed during the execution of the transactions through a novel graph based construct which we name Speculative Polygraph (SP). SPs can be viewed as an extension of Papadimitriou's polygraphs [17], which were introduced to determine the view-serializability of (non-speculative) transaction histories. More in detail, the SCC exploits knowledge on conflict dependencies tracked by a SP associated with each transaction T_i^j in order to determine which subset $V(X)$ of the currently available versions of a data item X can T_i^j return upon its n-th read operation (on data item X) without violating view-serializability and given the history of execution of its former $n-1$ read operations. By forking from T_i^j a number of $|V(X)|-1$ sibling transactions, and delivering to each forked transaction, and to the parent, a different value of X in the set $V(X)$, SCC completely covers all the distinct execution trajectories that T_i^j could undertake by letting the read operation return a different value (though representative of some view-serializable execution history) among those already available for X.

This read-triggered forking mechanism is however insufficient to ensure the complete exploration of the alternative transactions' speculative serialization orders. In fact, new versions of a data item X can become available after the execution of the read on X by transaction T_i^j. This happens if some transaction writes on X after T_i^j carries out its read on X. To tackle these situations, the SCC relies on an additional a-posteriori transaction re-spawning mechanism. The re-spawning of transaction T_i^j, which leads to the restart of a new transaction, say, T_i^k, is triggered as soon as a transaction T completes its execution and makes available a new version of a data item X, which could have been visible by T_i^j at the time of its read on X, in some legal sequential history. The SCC responds to all the reads issued by the re-spawned transaction, T_i^k, up to the read on data item X, by returning the same values already observed by transaction T_i^j. Since we are assuming the snapshot determinism of transactions, this implies that T_i^k will "clone" the execution of T_i^j up to the read on X, which, conversely, will return the version created by transaction T.

As hinted in the system model section, we made the choice to make visible the data item versions written by a transaction only when the transaction completes its execution, rather than as soon as the write operation is completed. In the re-spawning mechanism, this avoids scenarios in which a transaction 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 would in fact lead to violations of the Consistency property.

A. Speculative Transaction Manager

The pseudo-code of the SXM is reported in Figure 2. The SXM relies on three data structures, namely the *ActivatedXacts*, *CompletedXacts* and *CommittedXacts* sets, which contain references to the transactions T_i^j in the corresponding execution stage. To simplify the pseudo-code, we assume that whenever a transaction is started, or forked or respawned, it is inserted in the *ActivatedXacts* set, and that it is inserted in the *CompletedXacts* when its `completed()` method is invoked. Also, whenever a transaction is aborted, it is removed from the *ActivatedXacts* and CompletedXacts set and when it is committed, it is removed from the *CompletedXacts* set and added to the *CommittedXacts* set.

```
Set<Transaction> ActivatedXacts;
Set<Transaction> CompletedXacts; // CompletedXacts ⊆ ActivatedXacts
Set<Transaction> CommittedXacts;

Result invoke(TransactionalLogic T, inputParams p) do
 Transaction $T_i$ = new Transaction(T, p, getNewXactID());
 OAB.TO-broadcast($T_i$);
 wait $\exists T_i^j \in CommittedXacts$
 return $T_i^j$.getResult();

upon Opt-Deliver(Transaction $T_i$) do
 startSXact($T_i$);

upon TO-Deliver(Transaction $T_i$) do
 if ($\exists T_i^j \in$CompletedXacts s.t. $T_i^j$.validateTransaction())
  $T_i^j$.commit(); //the commit method aborts any sibling transaction of $T_i^j$
 else
  $\forall T_i^j \in$ CompletedXacts do
   $T_i^j$.abort(); //any completed $T_i^j$ is invalid
  if ($\nexists T_i^k \in ActivatedXacts$)
   if (($\forall T_r$ s.t. ($T_r \rightarrow T_i$)$\in$OAB.TO-DeliveredMsgs() $\exists T_r^s \in$CommittedXacts))
    startNonSXact($T_i$);
   else
    startSXact($T_i$);
```

Fig. 2. Pseudo-code for the Speculative Transaction Manager

When the application calls the `invoke()` method of the SXM, the latter marshals a message containing the input parameters specified by the application, generates a unique transaction identifier through the `getNewXactID()` primitive (which we denote with i in the pseudo-code) and TO-broadcasts the transaction through the OAB service. Next it waits for a transaction T_i^j to be committed in order to return the associated result (retrieved through the `getResult()` primitive) to the local application.

The activities of the SXM are triggered by two additional events, namely the Opt-Deliver and the TO-Deliver of a transaction T_i. In the former case, the SXM invokes the `startSXact` primitive with T_i as input in order to start a new transaction T_i^j which will be executed in speculative mode (more details on the difference between speculative and non-speculative mode of execution of transactions will be provided in Section V-B) and will be added to the *ActivatedXacts* set.

Upon a TO-deliver for transaction T_i, on the other hand, the SXM checks whether there exists an already completed transaction T_i^j that successfully passes the validation phase, which is meant to verify whether T_i^j accessed a snapshot consistent with the one produced by sequentially executing all the transactions that precede T_i in the final order (see Section V-B for details). If at least a transaction T_i^j is successfully validated, this is simply committed, causing the abort of any of its other sibling transactions, see Figure 5. Otherwise, whichever completed transaction T_i^j is aborted, causing the cascading abort of any other transaction exhibiting (a possibly indirect) read-from dependency (see Section V-B for further details). Next the SXM checks whether there is no other transaction T_i^j currently active. In such a case, a new transaction T_i^k needs to be activated through either the `startNonSXact`, or the `startSXact` primitives depending on whether the transactions that precede T_i according to the final order have all been committed or not. In such a case, in fact, the freshly activated transaction can safely read the current committed snapshot. On the other hand, if there is any transaction preceding T_i in the final order that has not been committed yet, the SXM, rather than waiting for their commitment, activates T_i^k in speculative mode. This choice reflects the optimistic assumption of absence of conflicts among T_i^k and any other not yet committed transactions preceding T_i^k in the final order.

B. Speculative Concurrency Control

Speculative Polygraphs. To determine the set of speculative serialization orders in which transactions need to be executed, the SCC relies on a novel graph-based construct, which we call Speculative Polygraphs (SP). SPs are inspired by Papadimitriou's polygraphs, which, as previously hinted, were introduced in [17] to test the view-serializability of a non-speculative history $\mathcal{H}$ and whose definition we briefly recall in the following.

A polygraph $P = (N, A, B)$ is a direct 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 $< (T'' \rightarrow T'), (T' \rightarrow T) >$ (where $(T \rightarrow T'') \in A$), not necessarily present in A. De-facto, a polygraph is a compact representation of a family of direct 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 some data item X from transaction T', the arc $(T' \rightarrow T)$ is added in A; (ii) if a third transaction T'' also writes X, then the bipath $< (T \rightarrow T''), (T'' \rightarrow T') >$ is added to B. In other words, each arc (T', T) in A keeps track of the direct read-from relation between transactions T and T', whereas a bipath $< (T \rightarrow T''), (T'' \rightarrow T') >$ means that since also T'' writes X, it can not be between T' and T, but must either precede T' or follow T.

Based on the above definition of polygraph, Papadimitriou defines a polygraph as acyclic iff there is at least an 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 [17]. We show in Figure 3.a and 3.b the polygraphs associated with, respectively, the following two sequential transaction histories $\mathcal{H}_1$=$\{T_1, T_2, T_3^0\}$ and $\mathcal{H}_2$=$\{T_2, T_1, T_3^1\}$, where T_1 writes on data items X and Y, T_2 writes on data items Y and Z, whereas T_3^0 and T_3^1 read the same data items, namely X, Y and Z (returning different values for Y given that $\mathcal{H}_1$ and $\mathcal{H}_2$ serialize in a different order transactions T_1 and T_2). As in [17], in Figure 3 a bipath is denoted by adding a circular edge on its common node. It can be easily verified that the only acyclic digraph associated with the polygraph in Figure 3.a is obtained by selecting the edge $(T_1 \rightarrow T_2)$ of the bipath centered on T_1, whereas the only acyclic digraph associated with the polygraph in Figure 3.b is obtained by selecting the edge $(T_2 \rightarrow T_1)$ of the bipath centered on T_2.

The unfeasibility of conventional polygraphs to reason on the the view serializability of a speculative transaction history appears manifest if one considers that the simultaneous coexistence within a polygraph of two sibling transactions, representative of inconciliable serialization orders (such as transactions T_3^0 and T_3^1 in Figure 3), can corrupt the polygraph by introducing cycles that might render it useless. An example of such a problem is shown in Figure 3.c, which shows the polygraph obtained by merging the polygraphs in Figure 3.a and 3.b. It is straightforward to verify that every direct 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_1 and T_2.

Speculative Polygraphs address exactly these problems. Unlike Papadimitriou's polygraphs, which are representative of a *whole*, and non-speculative, history, speculative polygraphs are designed to keep into account the history of execution as perceived by each speculative transaction. Roughly speaking, the SP of a transaction T_i^j is dynamically generated by selectively merging only the polygraphs of those speculative transactions T^* that i) conflict, either directly or indirectly, with T_i^j, and ii) such that exists at least a (non speculative) serialization order which allows both T^* and T_i^j to coexist. More formally, we define the speculative polygraph of transaction T_i^j, denoted as SP(T_i^j), 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 $(T_r^s \circledast \rightarrow T_i^j)$, with $T_r^s, T_i^j \in$N, and where the notation $T_r^s \circledast$ means that we are not just adding an edge between T_r^s and T_i^j, but also merging SP(T_r^s) with SP(T_i^j), and linking them via a (plain) edge from T_r^s to T_i^j.
- B is a set of, so called, *asymmetric bipaths*, denoted as $< (T_u^v \circledast \rightarrow T_r^s), (T_i^j \rightarrow T_u^v) >$, with $T_r^s, T_i^j \in$N, where the first of the two arcs is a merging edge linking SP(T_u^v) with SP(T_i^j) through the plain edge $(T_u^v \rightarrow T_r^s)$, and the

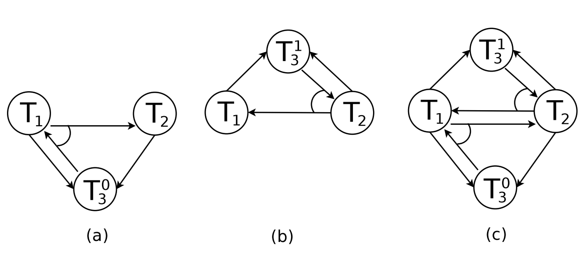

Fig. 3. Polygraphs Associated with Histories $\mathcal{H}_1$ and $\mathcal{H}_2$

second one, namely ($T_i^j \rightarrow T_u^v$), is a plain edge between the nodes T_i^j and T_u^v.

A speculative polygraph SP(T_i^j)=(N, A, B) generates a family of direct graphs $\mathcal{D}(SP(T_i^j))$, where each direct graph $\delta \in \mathcal{D}(SP(T_i^j))$ is obtained by (1) recursively replacing any merging arc, say $(T_r^s \circledast \rightarrow T_t^u)$, of A and B with the speculative polygraph SP(T_r^s)$\cup$($T_r^s \rightarrow T_t^u$), and (2) for each asymmetric bypath $< a_1, b_1 >$ present after the previous "merging phase", selecting either a_1 or b_1.

A speculative polygraph SP(T_i^j) is initialized, at transaction creation time (see Figure 4), by serializing T_i^j after the most recently committed transaction (according to the TO-deliver order), say T_c^d, through a merging edge. This has the effect of setting a barrier, in terms of minimum logical time, for the visibility of data item versions observable by the reads of T_i^j.

The speculative polygraph of a transaction T_i^j is then used to determine whether the k-th read operation of T_i^j can return a given version X^s (possibly created by a not yet committed transaction). Indeed, letting the k-th read of T_i^j return a version X^s, rather than any other available one, corresponds to speculating on a set of possible serialization orders for T_i^j. In order to ensure the consistency of the k-th read of a transaction with its current execution history it is however necessary that at least one of the speculative serialization orders associated with the reading of version X^s results "compatible" with those already determined by having executed the preceding k-1 reads. In this case we say that X^s is speculatively visible to T_i^j. To determine if T_i^j may speculatively view a data item version X^s based on its current execution history, its Speculative Polygraph, SP(T_i^j), is updated by:

(R.1) adding a merging edge from the creator of version X^s to T_i^j, namely $(X^s.creator\circledast \rightarrow T_i^j)$

(R.2) adding, for each other available version $X^{s'}$, an asymmetric bipath: $< (X^{s'}.creator\circledast \rightarrow X^s.creator), (T_i^j \rightarrow X^{s'}.creator) >$.

Version X^s is considered speculatively visible iff there exists at least one direct graph $\delta \in \mathcal{D}(\text{SP}(T_i^j))$ such that:

(C.1) δ is acyclic, and

(C.2) δ does not contain two sibling transactions T_a^r, T_a^q both serialized before T_i^j, or, formally, $\nexists T_a^r, T_a^q \in \delta$ such that $(T_a^r \rightarrow T_i^j) \in \delta^*$ and $(T_a^r \rightarrow T_i^j) \in \delta^*$, where with δ^* we denote the transitive closure of δ.

If for a $\delta \in \mathcal{D}(\text{SP}(T_i^j))$ both conditions (C.1) and (C.2) hold, we also say that δ is *valid*.

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

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

Transaction's Data Structures. Each speculative transaction T$_i^j$ is associated with an instance of the Transaction class which keeps track of the following state information (see Figure 4): i) id and $specId$, which are set, respectively, to the values i and j when the Transaction object associated with T$_i^j$ is created; ii) SP, which stores the speculative polygraph of T$_i^j$; iii) RS, namely T$_i^j$'s read set, which is organized as an array whose n-th entry records the identity X of the data item read during the n-th read operation, the version of X read, and a copy of T$_i^j$'s Speculative Polygraph right *before* the read took place; iv) WS, T$_i^j$'s write set; v) two boolean variables, $speculative$ and $respawned$, reflecting, respectively, the transaction was activated speculatively, and whether it was respawned; vi) $readOpCounter$, namely a counter that keeps track of how many read operations have been issued up to date by the transaction; vii) $generatingRead$, which stores a value different from 0 only if T$_i^j$ has been activated through the fork/spawn of some sibling transaction T$_i^j$, in which case it keeps track of the T$_i^k$'s read operation that has triggered the forking/spawing of T$_i^j$.

Write/Read Operations. The logic for the management of write operations (see the `write()` method in Figure 4) is extremely simple: the transaction simply logs the identity of the target data item, as well as its new value, into its local WS variable, which can be seen as the transaction's "private workspace".

Concerning the read operations (see the `read()` method in Figure 4), the SCC first checks whether the transaction has already written the same data item for which it's issuing the read. In the positive case, it just returns the corresponding value stored in its write set. Next, it verifies whether the transaction has been activated in non-speculative mode. As anticipated in Section V-A, T_i^j is activated in non-speculative mode only if T_i has already been TO-delivered and if all the transactions preceding T_i in the final order have already committed. Hence, any read executed by a non-speculative transaction can safely return the most recently committed version. On the other hand, if the read operation is issued by a speculative transaction, the SCC determines, through the `getAllVisibleVersions()` method, what subset of the available versions is speculatively visible by the reading

```
class Transaction {
 int id, specId;
 Transaction $T_c^d$ =max$\{T_a \in$ OAB.TO-deliveredMsgs() s.t. $\exists T_a^b$ $\in$CommittedXacts$\}$
 SpeculativePolygraph SP = ( $T_c^d \circledast \rightarrow T_{id}^{specId}$ )
 List$<$DataItemID, DataItemVersion, SpeculativePolygraph$>$ RS;
 Set$<$DataItemID,Value$>$ WS;
 boolean speculative, respawned;
 int readOpCounter=0, generatingRead=0;

 void write(DataItemID X,Value v)
  WS.store(X,v); // if X already exists in WS it gets over-ridden

 Value read(DataItemID X)
  readOpCounter++;
  if ($< X, v >$$\in$WS) return $v$;
  if ($\neg$speculative)
   return X.mostRecentCommitted();
  if (respawned $\wedge$ readOpCounter$\leq$generatingRead)
   return RS[readOpCounter].getValue();
  Set$<$DataItemVersion, SpeculativePolygraph$>$ versions = getAllVisibleVersions(X);
  $[X^i, newSP]$ = versions.pop();
  $\forall$ $<$DataItemVersion $X^j$, SpeculativePolygraph SP'$>$$\in$versions do
   forkSibling();
   if (I am the just forked transaction)
    RS[readOpCounter] = $<$X,$X^j$,SP$>$;
    SP = SP';
    generatingRead = readOpCounter;
    return $X^j$;
  RS[readOpCounter] = $<$X,$X^i$,SP$>$;
  SP = newSP;
  return $X^i$;

 void completed() // invoked when the transaction logic terminates its execution
  if ($T_{id} \notin$OAB.TO-DeliveredMsgs())
   STS.addSVersions($T_{id}^{specId}$);
   handleWriteAfterReadConflicts();
  else
   if ($\forall T_r$ s.t. $(T_r \rightarrow T_{id})$$\in$OAB.TO-DeliveredMsgs() : $\exists T_r^s$ $\in$CommittedXacts)
    if ($\neg$speculative $\vee$ validateTransaction())
     commit();
     Transaction $T_j =$ min$\{T_a \in$ OAB.TO-deliveredMsgs() s.t. $(T_{id} \rightarrow T_j)\}$
     do
      if ($\exists T_j^k$ $\in$CompletedXacts s.t. ($\neg T_j^k$.speculative $\vee$ $T_j^k$.validateTransaction()))
       $T_j^k$.commit();
      else
       $\forall T_j^k$ $\in$CompletedXacts do
        $T_j^k$.abort();
        break;
     while($T_j$ = $T_j.next_{OAB.TO-deliveredMsgs()}$)
    else
     abort();
     if ($\nexists T_{id}^{\cdot} \in$ ActivatedXacts)
      startNonSXact($T_{id}^{specId}$);
 . . .
}
```

Fig. 4. Pseudo-code for the Speculative Concurrency Control (1)

transaction based on its execution history. The isVisible() method returns either a $\perp$ value if the target data item version is not speculatively visible. Otherwise, it returns the SP of the reading transaction T_i^j updated to reflect the outcome of the read. Next, the transaction picks one of the selected visible versions, say X^s, to use it as a return value for the on-going read, logs it, together with its current SP, in its read set, and only then replaces its SP with the one updated by the isVisible() method. Finally, before returning X^s, T_i^j forks, for each other visible version X^t, a sibling transaction whose execution will proceed in parallel with T_i^j, after returning X^t for the ongoing read (and after having correspondingly updated its read set and SP).

Transaction's Completion. When T_i^j completes its execution,

```
class Transaction {
. . .
void handleWriteAfterReadConflicts()
 $\forall T_j^k \in$ ActivatedXacts s.t. $(T_j^k.RS \cap WS \neq \emptyset)$ do
  let $r$ be the min index s.t. $T_j^k.RS[r].DataItemId \in WS$;
  $(X, \cdot, SP') = T_j^k.RS[r]$;
  let $X^i$ be the value of $X$ written by the current transaction;
  if (r $>$ generatingRead)
   SpeculativePolygraph newSP = isVisible(X, $X^i$, $SP'$, $T_j^k$);
   if (newSP$\neq\perp$)
    $\forall T_j^k.RS[s] =< \cdot, \cdot, SP'' > s.t. s > r$ do
     $T_j^k.RS[s] =< \cdot, \cdot, SP''$$\cup$ $<$($T_{id}^{specId} \circledast \rightarrow X^s$.creator),
       ($T_j^k \rightarrow T_{id}^{specId}$)$>>$;
    ReadSet newRS;
    $\forall$ $1 < t < r$ do
     newRS[t] = $T_j^k$.RS[t];
    newRS[r] = $<$X,$X^i$,newSP$>$;
    spawnSibling($T_j$,newSP,newRS,r);

Set$<$DataItemVersion,SpeculativePolygraph$>$ getAllVisibleVersions(DataItem X)
 Set$<$DataItemVersion,SpeculativePolygraph$>$ VisibleVersions;
 $\forall X^i \in X$.getAllVersions() do
  SpeculativePolygraph newSP = isVisible(X, $X^i$, SP, $T_{id}^{specId}$);
  if (newSP$\neq\perp$)
   VisibleVersions = VisibleVersions $\cup$ $< X^i, newSG >$;
 return VisibleVersions;

SpeculativePolygraph isVisible(DataItem X,
   DataItemVersion $X^i$, SpeculativePolygraph currentSP, Transaction T)
 SpeculativePolygraph newSP = ($X^i$.creator$\circledast$ $\rightarrow$T)$\cup$currentSP;
 $\forall(X^j \neq X^i) \in$ STS.getAllVersions(X) do
  newSP = newSP $\cup$ $<$ ($X^j$.creator$\circledast$ $\rightarrow$ $X^i$.creator) , (T$\rightarrow$ $X^j$.creator)$>$;
 if ( $\exists\delta \in \mathcal{D}$(newSP) s.t. isValid($\delta$, T))
  return newSP;
 else return $\perp$;

void abort()
 STS.removeSVersions($T_{id}^{specId}$);
 $\forall T_l^m \in ActivatedXacts$ do
  $\forall\delta \in \mathcal{D}(T_l^m.SP)$ s.t. isValid($\delta$,$T_l^m$) do
   if (($T_{id} \rightarrow T_l^m$) $\in \delta^*$)
    $T_l^m$.abort();
 ActivatedXacts = ActivatedXacts $\setminus$ $\{T_{id}^{specId}\}$;
 CompletedXacts = CompletedXacts $\setminus$ $\{T_{id}^{specId}\}$;

void commit()
 STS.commitSVersions($T_{id}^{specId}$);
 $\forall T_{id}^m \in ActivatedXacts$ do
   $T_{id}^m$.abort(); // Abort sibling transactions

boolean isValid(DirectGraph $\delta$, Transaction T)
 return ($\delta$ is acyclic $\wedge$ ($\nexists$ ($T_i^j \rightarrow T$), ($T_i^k \rightarrow T$) $\in \delta^*$ with $j \neq k$));

boolean validateTransaction()
 $\forall$ $<$X,value,$\cdot$ $>$ $\in$ RS do
  if (value $\neq$ X.getCommittedVersion().getValue())
   return false;
  return true;
}
```

Fig. 5. Pseudo-code for the Speculative Concurrency Control (2)

it adds itself to the Completed set. Then, if T_i is already TO-delivered and if exists some transaction T_p that precedes T_i in the final order and has not yet committed, the completed() method simply ends, delegating the task of attempting to commit T_i^j to any T_p^s that will subsequently commit. Conversely, if all the transactions T_p preceding T_i in the final order have already committed, and T_i^j is either non-speculative or successfully passes the validation phase, T_i^j is committed and attempts to commit any completed transaction T_l^m, such that T_l follows T_i in the final order.

On the other hand, in case T_i has not yet been TO-delivered, T_i^j makes available its versions through `addSVersions`. Then it invokes the `handleWriteAfterReadConflicts`() method to determine whether there is any transaction T_a^b that has read some data item also written by T_i^j, and whether the version created by T_i^j was speculatively visible for T_a^b at the time in which it executed the read. To this end, the `isVisible`() method is invoked passing as input parameter the SP (retrieved from T_a^b's read set) that T_a^b was storing at the earliest time in which T_a^b read a data item, say its $r-$th read, also written by T_i^j.

Note that by retrieving the SP of the *first* data item in T_a^b's read set to have also been written by T_i^j, we detect the earliest possible point in the execution trajectory of T_a^b that could have been affected by a write of T_i^j. Recall that in a snapshot deterministic model, if T_a^b had seen the version created by T_i^j, rather than the one returned in its execution, T_a^b might not execute the same set of subsequent reads. Hence, to avoid violating the Non-redundancy property, the SCC respawns only a single sibling of T_a^b, say T_a^c, even if there are multiple data items in $T_i^j.WS \cap T_a^b.RS$. T_a^c will re-execute (see the read() method in Figure 4) the same set of reads already performed by T_a^b, up to the read that caused its spawning. Such a read will return the value created by T_i^j and, henceforth, T_a^c will be free to evolve in its execution trajectory. To enforce such a behavior, the first r-1 entries of the read set of the spawned transaction are set equal to the corresponding ones of T_a^b, and its r-th entry to reflect the read from T_i^j. Note that T_i^j also updates, in the T_a^b's read set, every entry following the read that caused the spawning, reflecting the fact that T_a^b did not observe during the r-th read the data item version generated by T_i^j. This guarantees the completeness of the information stored by SP(T_a^b), which could be again queried in future from within the `handleWriteAfterReadConflicts`() method.

In order not to incur in violations of the Non-redundancy property, we take one additional measure: if T_i^j detects that the conflict affects the $r-th$ read of a transaction T_a^b and T_a^b was activated due to a fork/respawn of some sibling transaction T_a^c occured upon T_a^c's k-th read (i.e. the value of $generatingRead$ for T_a^b is k), where $r < k$, then T_i^j avoids respawning T_a^b. In this case, in fact, since T_a^b and T_a^c exhibit the same behavior up to the $k-1$-th read, the sibling transactions spawned by T_a^b and T_a^c to deal with a conflict occuring at a read $r < k$ would observe the same snapshot. It is therefore sufficient to re-spawn exclusively T_a^c.

Commit/Abort/Validation. A transaction is considered successfully validated iff the values read by the transaction during its execution, and stored in its RS variable, coincide with the values currently stored by the committed version of the corresponding data item (see validateTransaction() method of Figure 5). The commit() method marks the data item versions created by the committing transaction as the currently committed versions, and then triggers the abort of any of its sibling transactions through the abort() method. When this latter method is invoked on transaction T_i^j, it first removes any data items' versions made available by T_i^j, and then triggers the cascading abort of any other transaction, say T_l^m, having a, possibly indirect, read-from dependency with the aborting transaction. This is verified by checking if for every valid $\delta \in \mathcal{D}(\text{SP}(T_l^m))$ there is a path from T_i^j to T_l^m.

Example Scenario. Let us consider an example scenario of execution of the SCC associated with the following history:
$\mathcal{H}_3$= { $B_{T_1^0}$, $R_{T_1^0}(X)$, $W_{T_1^0}(X)$, $C_{T_1^0}$, $B_{T_2^0}$, $R_{T_2^0}(X)$, $F_{T_2^1}$, $W_{T_2^0}(X)$, $R_{T_2^1}(X)$, $W_{T_2^1}(X)$}
where we use the notation $B_{T_i^j}$, $R_{T_i^j}$, $W_{T_i^j}$, $C_{T_i^j}$ and $F_{T_i^j}$, to denote, respectively, the begin, read, write, complete (and, note, not the commit) and forking of a transaction T_i^j.

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 Figure 6.a we show the state of SP(T_1^0) right after the execution of the read operation on X. Note that, in order to refer to a merging edge, and distinguish it from a plain edge, we use a dashed arrow, which in Figure 6.a reflects the read-from dependency from T_1^0 to C. Further, within the speculative graph of SP(T_i^j), we use the convention of drawing a double circle to refer to transaction T_i^j

Figure 6.b shows the state of SP(T_2^0) after the execution of the read on X, assuming that T_2^0 reads the committed version. Since T_1^0 has already completed executing (but has not been committed yet) at the time in which T_2^0 performs the read, T_2^0 finds also available the versions created by T_1^0. Thus, SP(T_2^0) contains also the asymmetric bipath $<(T_1^0\circledast \rightarrow C),(T_2^0 \rightarrow T_1^0)>$, which we denoted by adding a circular arc on the common node. Figure 6.c and 6.d shows the two direct graphs $\delta, \delta' \in \mathcal{D}(\text{SP}(T_2^0))$. As it can be seen, the direct graph in Figure 6.c, associated with the merging edge $(T_1^0\circledast \rightarrow C)$ exhibits a cycle, but since the direct graph in Figure 6.d is acyclic the committed version of X results speculatively visible by T_2^0.

In Figure 6.e we provide the speculative polygraph of T_1^0 after the update performed during the completion phase of T_2^0 (i.e. within the method `handleWriteAfterReadConflicts`()), that adds the asymmetric bipath $b_1 =<(T_2^0\circledast \rightarrow C),(T_1^0 \rightarrow T_2^0)>$ to SP(T_1^0). In Figure 6.g, we show the only valid direct graph in $\mathcal{D}(\text{SP}(T_1^0))$, which serializes T_1^0 before T_2^0. In fact, as shown in Figure 6.f, by considering the merging edge $(T_2^0\circledast \rightarrow C)$ of b_1, and merging SP(T_2^0) with SP(T_1^0), the resulting speculative polygraph is necessarily cyclic.

Finally, we show in Figure 6.h the speculative graph of T_2^1, namely the transaction forked by T_2^0 upon the read of data item X, and which returns the version of X written by T_1^0. Figure 6.i shows the only direct graph in $\mathcal{D}(\text{SP}(T_2^1))$ to be acyclic that permits the speculative visibility of the version written by T_1^0.

C. Correctness

For space constraints, we cannot report the formal proof of protocol correctness. We remaind the interested readers to [20],

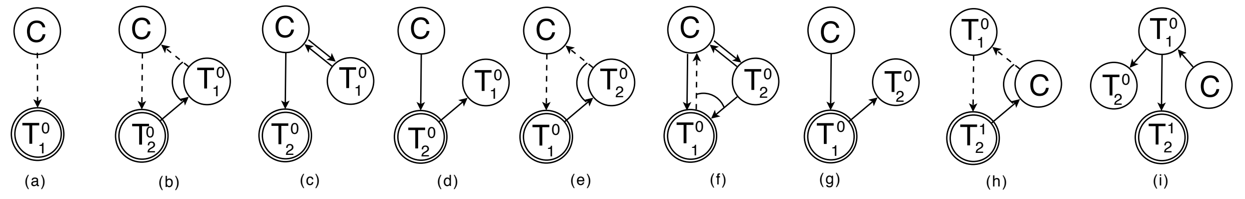

Fig. 6. SCC Execution for History $\mathcal{H}_3$

where we prove how the protocol guarantees all the properties specified by the problem statement in Section IV.

D. 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 commit 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 in aborts of read-only transactions.

An alternative solution, exhibiting specular 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 STS to maintain multiple committed versions of each data item.

VI. Conclusions

In this paper we investigated the issues related with the adoption of a speculative approach for the replication of transactional systems, which we name STR (Speculative Transactional Replication). STR exploits an OAB service and maximizes the overlapping between communication and computation by processing transactions in speculative serialization orders, corresponding to distinct permutations of the set of optimistically delivered messages. The challenge is to avoid the exhaustive exploration of the set Π of permutations of the optimistically delivered messages, something that is both infeasible and non-productive as, in practice, it is highly unlikely that every transaction conflicts with every other one. Hence, a (possibly large) portion of the set of the permutations in Π would actually generate identical, redundant computations.

We formalized the problem in a realistic model where transactions' data access patterns are not known a-priori and may vary depending on the snapshots they read. We defined a set of correctness and optimality criteria that shelter transactions from reading inconsistent snapshots and require the online identification of the minimum set of serialization orders generating every distinct transaction execution trajectories.

Finally, we presented an optimal STR protocol, which relies on a novel graph-based construct, called Speculative Polygraph (SP), to dynamically encode information on the conflict relations developed during the speculative execution of transactions.

References

[1] Schneider, F.B.: Replication management using the state-machine approach. ACM Press/Addison-Wesley Publishing Co. (1993)

[2] D. Powell (ed.): Special Issue on Group Communication. Volume 39. ACM (1996)

[3] Agrawal, D., Alonso, G., Abbadi, A.E., Stanoi, I.: Exploiting atomic broadcast in replicated databases (extended abstract). In: Proc. of International Euro-Par Conference on Parallel Processing, Springer-Verlag (1997) 496–503

[4] Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: Proc. of the International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (1998) 156

[5] Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Proc. of the International Conference on Distributed Computing Systems, IEEE Computer Society (1999) 424

[6] Pedone, F., Schiper, A.: Optimistic atomic broadcast: a pragmatic viewpoint. Theor. Comput. Sci. **291**(1) (2003) 79–101

[7] Patino-Martinez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Middle-r: Consistent database replication at the middleware level. ACM Trans. Comput. Syst. **23**(4) (2005) 375–423

[8] Mocito, J., Respicio, A., Rodrigues, L.: On statistically estimated optimistic delivery in large-scale total order protocols. In: Proc. of the International Symposium on Pacific Rim Dependable Computing, IEEE (2006)

[9] Romano, P., Carvalho, N., Rodrigues, L.: Towards distributed software transactional memory systems. In: Proc. of the Workshop on Large-Scale Distributed Systems and Middleware (2008)

[10] Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. SIGPLAN Not. **41**(10) (2006) 253–262

[11] Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases **14**(1) (2003) 71–98

[12] Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proc. of the Conference on the Management of Data, ACM (1996) 173–182

[13] Bestavros, A., Braoudakis, S.: Value-cognizant speculative concurrency control. In: Proc. of the International Conference on Very Large Data Bases. (1995) 122–133

[14] Reddy, P.K., Kitsuregawa, M.: Speculative locking protocols to improve performance for distributed database systems. IEEE Trans. on Knowl. and Data Eng. **16**(2) (2004) 154–169

[15] Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer (2006)

[16] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)

[17] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4) (1979) 631–653

[18] Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D^{2}STM: Dependable Distributed Software Transactional Memory. In: Proc. Symposium on Dependable Computing (PRDC), IEEE (2009)

[19] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *Proc. of the 15th International Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, IEEE (2010).

[20] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, On Speculative Replication of Transactional Systems. INESC-ID Tech. Rep. 38/2009, July 2009.