OB-STM: An Optimistic Approach for Byzantine Fault Tolerance in Software Transactional Memory

Tulio Alberton Ribeiro
Department of Informatics and Statistics
Federal University of Santa Catarina
Santa Catarina, Brazil
tulio.ribeiro@posgrad.ufsc.br

Lau Cheuk Lung
Department of Informatics and Statistics
Federal University of Santa Catarina
Santa Catarina, Brazil
lau.lung@inf.ufsc.br

Hylson Vescovi Netto
Department of Informatics and Statistics
Federal University of Santa Catarina
Santa Catarina, Brazil
hylson.vescovi@posgrad.ufsc.br

***Abstract*—Recently, researchers have shown an increased interest in concurrency control using distributed Software Transactional Memory (STM). However, there has been little discussion about certain types of fault tolerance, such as Byzantine Fault Tolerance (BFT), for kind of systems. The focus of this paper is on tolerating byzantine faults on optimistic processing of transactions using STM. The result is an algorithm named OB-STM. The processing of a transaction runs with an optimistic approach, benefiting from the high probability of messages being delivered in order when using Reliable Multicast on a local network (LAN). The protocol has a better performs when messages are delivered ordered. In case of a malicious replica or out-of-order messages, the Byzantine protocol is initiated. In smaller scenarios and using an optimistic approach, the protocol has a better throughput than Tazio.**

I. Introduction

The Distributed Shared Memory (DSM) systems hides the remote communication mechanism from the application developer [1]. It could be described as a virtual address space that is shared by a number of processors [2]. In DSM systems, the processor accesses every memory address as if it were local. The system allows coherent data sharing through an uniform way of read and write (lock-based synchronization) to shared structures in the common memory. As an alternative to lock-based synchronization, Software Transactional Memory (STM) systems offers a concurrency control mechanism through transactions, which are analogous to the database transactions for controlling access to shared memory in concurrent computing.

Researchers have shown an increased interest in concurrency control using distributed STM [3], [4], [5], [6], [7]. The programmers that use STM take advantage of the fact that it is not necessary to deal explicitly with concurrency control mechanisms, like mutual exclusion algorithms. Instead, they have only to identify which parts of code need to be treated as transactional. This allows programmers to focus on global operation of application, rather than explicit mechanisms of concurrency control. The STM application is responsible for contention management as well.

However, there are few STM architectures that support fault tolerance in the literature [3], [4], [5], [6]. Furthermore, none of the proposed architectures takes into account the high probability of messages being delivered in order. This characteristic allows the construction of optimistic systems, which explore this when using Reliable Multicast on LAN [8], [9].

Only the Zhang's [7] work mentions about Byzantine fault tolerance on the context of STM. The architecture of Zhang is divided in two *clusters*, one for agreement ($3f + 1$, where f means the number of faulty replicas) of messages and the other for transactions execution ($2f+1$). This approach does not use optimistic delivery of messages by the network and does not allow transaction execution without Byzantine protocol.

This paper presents OB-STM, a Byzantine Fault Tolerant protocol using an optimistic approach to Software Transactional Memory. The proposed protocol is based initially on parallel executions of non conflicting transactions, resembling to Kotla [10]. In case of out-of-order messages or malicious replica, the order of transactions is defined with the protocol proposed by Castro [11]. The OB-STM initiates with high probability of messages being delivered in order by the local network using IP-Multicast [8], [9]. While running in local networks, the protocol can explores replica determinism by the use of an optimistic approach. If the messages are delivered into the same order to all replicas and there is no malicious replica among them, then there is no need to begin execution of Byzantine protocol.

The order verification of delivery messages occurs on *committing* transactions stage. On this stage, a verification message is sent to all replicas. If all replicas return affirmative for the request $\langle AskForCommit \rangle$, the transaction can be committed. If there is a malicious replica or some replica does not return positively the request confirmation, the Byzantine protocol is initiated to determine the order and execution of transactions. The order is based only on conflicting and out-of-order transactions. An example of transactions out-of-order is when one replica executes transactions *t1* and *t2*, and another replica executes *t2* and *t1*.

The OB-STM was built using the JVSTM [12], the JVSTM is a library that use *MVCC (MultiVersioning Concurrency Control)* as concurrency control mechanism. The JVSTM library allows for an optimum performance for read-only operations because they do not abort. Also, it uses a deferred update[1] as conflict verification; however, we use a different approach: an eager conflict detection named Parallel Transactional Analyzer (PTA) detailed in section V-D.

The paper has been organized in the following way: Section II presents related works. Section III describes systems and

[1]Deferred update: conflict verification is done only at *commit* phase.

2324-7894/14 $31.00 © 2014 IEEE
DOI 10.1109/SBESC.2013.31

definitions, protocol execution (optimistic and conceptually Byzantine) and execution flows. Section IV displays the algorithm executed in the replicas. Section V presents global and local verification, life cycle of transactions and Parallel Transactional Analyzer. Section VI exhibits the results and evaluation. Finally, section VII concludes the paper.

II. Related Work

Dependable Distributed Software Transactional Memory (D2STM)[3] implements a transactional memory system in distributed fault tolerant software. It tolerates crash faults, uses atomic diffusion for communication between replicas and 1-copy serializable for data distribution. JVSTM [12] works as the base for transactions executing, and read-only transactions are never aborted. Bloom Filter detects conflicts in reading and writing operations. D2STM's architecture uses *Atomic Broadcast* as message passing method, but does not take advantage of probably correct ordering of messages supplied by the use of local networks.

Tazio [5] tolerates crash faults in Software Transactional Memory. A Reliable channel is assumed among replicas for message exchange; but between client and replicas it is not required. The consistency between replicas is ensured by the property 1-copy serializable and the protocols write-update and write-invalidate. To manage conflicts it uses MVCC and version boxes [12]. When a writing transaction begins, it receives the Sequence Number (SN) of the last committed transaction plus one. This transaction can commit if it has a SN greater than the current SN. If the commit transaction succeeds, the current SN is updated. The model does not abort read-only transactions.

RAM-DUR [6] is a mechanism of distributed cache with high performance that uses strong data consistency. In traditional distributed databases, data are partially in memory; if required data is not in memory, disk access needs to be done. In the RAM-DUR database is divided among all replicas. In case of data that is not in memory, network access needs to be done. Only one replica executes the transaction in an optimistic approach, and in case of success it sends the writes and reads sets to the remaining replicas. The model tolerates crash faults.

Zhang [7] uses separated clusters for agreement and execution (STM). The agreement cluster establishes an order for all transactions and forwards them to the execution cluster. After execution, results come back to the agreement cluster where validation occurs, which could be: abort or restart the transaction. It is not possible to run nested transactions, and neither to execute transactions in parallel. There must be $3f + 1$ replicas to provide Byzantine Fault Tolerance. The library LSA-STM (A Lazy Snapshot Algorithm with Eager Validation) [13] is part of implementation; it aborts read-only transactions and uses Compare and Swap in validation phase. Synchronization algorithm in Zhang has the property *lock-freedom* that avoids deadlock, but allows starvation. The Zhang protocol ignores network characteristics.

Table I presents related works and the proposal. The columns refer: related works, uses atomic broadcast, tolerates byzantine faults, number of replicas, can abort a read-only transaction at the end of execution, technique for verifying conflict among transactions and publication year.

The validation phase is done usually on commit time; our approach uses another verification method, we do the analysis before execution. This verification is done initially to reduce transaction re-execution. Atomic Broadcast is used in [3], [5], [6], [7]. Our protocol uses an adaptive approach: the Reliable Multicast is used in optimistic execution, and Atomic Broadcast is used in Byzantine phase if necessary. OB-STM does not abort read-only transactions and uses $3f + 1$ in all protocol. The last line refers to our contribution.

	A-B	Byzantine.	Replica.	A-R	Validation	Year
D2STM	Yes	No	>1	No	Bloom filter	2009
Tazio	Yes	No	>1	No	Seq. number	2010
RAM-DUR	Yes	No	>1	No	Bloom filter	2012
Zhang	Yes	Yes	3f+1[2]; 2f+1[3]	Yes	CAS[4]	2012
OB-STM	**Adaptive**	**Yes**	**3f+1**	**No**	**PTA**	

TABLE I. Related works and the proposal.

III. Model and Definitions

This section describes the basic definitions of system and its architecture.

A. System definitions

We consider a classical asynchronous distributed system [14] consisting of a finite set of non-Byzantine clients C = $\{c1, c2, ..., cn\}$ and a finite set of replicas R = $\{r1, r2, ..., rn\}$. The replica set communicate to each other through message passing and can fail according to the Byzantine failure model [15], and at most f replicas can fail.

A faulty replica or Byzantine, can stop sending messages, send messages out-of-order, omit to send or receive messages, delay messages and corrupt messages. All messages are signed and exchanged through a reliable channel.

The proposed protocol uses pre-declared transactions. The client needs to send all transactional code to the OB-STM replicas. The underlying properties of JVSTM, such as weak atomicity and opacity, are preserved by the OB-STM protocol.

B. System Architecture

Figure 1 provides a high-level overview of the architecture of an OB-STM replica. The OB-STM component receives the request of clients and does conflict analysis. This analysis results in two possible states: *Executing* or *Waiting*. In the *Waiting* state the transaction has a conflict; In the *executing* state the transaction is forwarded to the JVSTM layer, where the transactions are processed and the execution flow is returned to the OB-STM layer.

IV. Algorithm OB-STM

The proposed algorithm has been divided into three parts: Algorithm 1 ***Variables***, the part that contains the variables used

[2]Agreement.
[3]Execution.
[4]Compare and Swap.

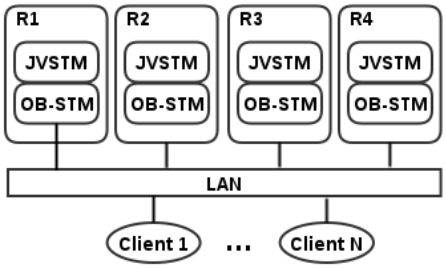

Fig. 1. Replica architecture of OB-STM instance. For $f = 1$.

on the algorithm; Algorithm 2 ***Replica***, which contains optimistic execution code, conflict detection, catch write and read sets of transactions and commit transactions; and Algorithm 3 ***Byzantine Replica***, which determines the execution order and the re-execution of transactions.

The proposed protocol assumes ordered delivery of messages in the network (i.e. IP-Multicast in local networks), due to the high probability that messages can be received in the same order by all replicas [8], [9].

When operating optimistically, presented in Figure 2, the client sends a $\langle REQUEST \rangle$ message to all replicas using reliable multicast (step I). Each client generates an UUID[5] to identify the transaction request. When the message is received (line 1 algorithm 2), the GetWxRx function (line 6 algorithm 2) stores the set of writings and readings, and the PTA function (line 11 algorithm 2) verifies if there are conflicts (step II). If there are no conflicts with transactions that are executing or waiting (line 16 algorithm 2), or if there is only one transaction in the set $\prod^T$, then execution starts (line 22 or 32 algorithm 2). At step III, the replicas individually initiate and execute transactions without exchanging messages. When the execution is finished, an attempt for a commmit of the transactions is issued.

When trying to commit (step IV), the replica sends a message $\langle AskForCommit \rangle$ (line 41 algorithm 2) to its peers, asking permission to confirm the transaction. If the replica receives $(3f)$ confirmations (line 54 algorithm 2), the transaction is confirmed and removed from the set of pending transactions. The set of writings and readings of the transaction are removed as well. As an example, suppose that all replicas receive Tx, Ty and Tz in this order, and all three transactions are conflicting; all replicas should start executing the Tx transaction. When an $\langle AskForCommit \rangle$ message arrives, each replica should look for a transaction Tx that is in the executing or confirmation phase. If a transactions Tx is found, the replica returns *OK!* (line 47 algorithm 2), else returns *NOK!* (line 49 algorithm 2). In step *Va*, the answer is sent to the client (line 66 algorithm 2) and first element of $\prod^T$, if transaction exists, it is forwarded to the PTA (line 67 algorithm 2).

Whenever replicas receive out-of-order messages or there are malicious replicas, *NOK* messages will appear, as presented on Figure 3, step IV. If replicas do not receive at least $3f$ confirmations (line 56 algorithm 2), Byzantine phase starts (steps Vb, VI and VII). The replica that started the confirmation phase sends a $\langle PRE - PREPARE \rangle$ message to the peers

[5]UUID - Universally Unique Identifier

Algorithm 1 Variables

```
1: ∏^R                          ▷ Replica Set
2: ∏^Reply                      ▷ Reply Set
3: ∏^T                          ▷ Transaction Set, Pre-Declared
4: ∏^C                          ▷ UUID-Client Set: unique identifier
5: ∏^WS                         ▷ Write Set of Transactions
6: ∏^RS                         ▷ Read Set of Transactions
7: BCO                          ▷ Buffer Commit Order
8: f                            ▷ Number tolerated faults
9: timeout                      ▷ Timeout of wait replica response
```

Algorithm 2 Replica

```
1: upon: receive(⟨ REQUEST, Ti, Ci ⟩) from client
2:     ∏^T ← ∏^T ∪(Ti, Ci)
3:     Call GetWxRx(Ti, Ci)
4:     Call PTA(Ti, Ci, false)
5:
6: function GETWXRX(Ti, Ci)
7:     ∏^WS ← ∏^WS ∪ (getWriteSet(Ti), Ci)
8:     ∏^RS ← ∏^RS ∪ (getReadSet(Ti), Ci)
9: end function
10:
11: function PTA(Ti, Ci, Analyzed)
12:     if (size(∏^T) >1 ) then
13:         if (Analyzed = false) then
14:             initExec = true
15:             for all Tj ∈ ∏^T minus Ti do
16:                 if (WS(Ti) ∩ RS(Tj) ≠ ∅) ∨ (WS(Tj) ∩ RS(Ti) ≠ ∅) ∨ (WS(Ti) ∩
    WS(Tj) ≠ ∅)  then
17:                     initExec = false
18:                     stop for all
19:                 end if
20:             end for
21:             if (initExec = true) then
22:                 Call EXEC(Ti, Ci)
23:             end if
24:         else
25:             if (size(BCO) >0 ) then
26:                 Call RE-EXEC(getFirstElement(BCO), Ci)
27:             else
28:                 Call EXEC(getFirstElement(∏^T), Ci)
29:             end if
30:         end if
31:     else
32:         Call EXEC(Ti, Ci)
33:     end if
34: end function
35:
36: function EXEC(Ti, Ci)
37:     set Ti as running
38:     begin execution
39:     if  (Ti to try commit) then
40:         set Ti as committing
41:         ReliableMulticast(⟨ASKFORCOMMIT⟩, Ti, ∏^R, Ci)
42:     end if
43: end function
44:
45: upon: receive(⟨ASKFORCOMMIT⟩, Ti, Ri, Ci) from replica
46:     if (Ti can commit) then
47:         ReliableUnicast(⟨R-ASKFORCOMMIT⟩,Ti, Ri, Ci, "OK!")
48:     else
49:         ReliableUnicast(⟨R-ASKFORCOMMIT⟩, Ti, Ri, Ci, "NOK!")
50:     end if
51:
52: upon: receive(⟨R-ASKFORCOMMIT⟩, Ti, Ri, Ci, Reply) from replica
53:     acceptCommit ← WaitForAcceptance(timeout)
54:     if (acceptCommit = 3f) then
55:         Call CommitT (Ti, Ci)
56:     else
57:         //Begin Byzantine protocol.
58:         TO-Deliver(⟨PRE-PREPARE⟩, Ti, Ri, Ci, ORDER)
59:     end if
60:
61: function COMMITT(Ti, Ci)
62:     Commit transaction Ti!
63:     ∏^T ← ∏^T \ Ti
64:     ∏^WS ← ∏^WS \ (getWriteSet(Ti), Ci)
65:     ∏^RS ← ∏^RS \ (getReadSet(Ti), Ci)
66:     SendReliable(⟨REPLY⟩, Ri, Reply, Ci)          ▷ Send reply to client
67:     Call PTA(getFirstElement(∏^T), Ci, true)
68: end function
```

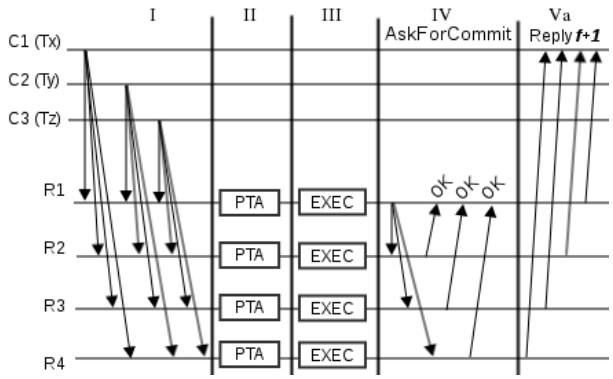

Fig. 2. Optimistic execution.

(line 58 algorithm 2), proposing one execution order for the conflicting and out-of-order transactions. Using the PBFT[11] protocol, $\langle PREPARE \rangle$ and $\langle COMMITORDER \rangle$ messages are exchanged (lines 4 and 11 algorithm 3). At the end of the Byzantine phase, execution order was established (line 16 algorithm 3) and stored in the *buffer* BCO. After the establishment of transaction order, the protocol enters in step VIII for re-execution of transactions if necessary.

Re-execution occurs in RE-EXEC function (line 29 algorithm 3), where transactions are executed and committed without exchanging messages. If a replica has transactions executed in the same order as defined by the Byzantine phase, re-execution will not be necessary for this replica. This comparison is made between transactions that are running and the transaction in the head of buffer BCO (line 22 algorithm 3). In case of differences (line 19 algorithm 3), re-execution is necessary (line 20 algorithm 3). After this procedure, an answer is sent back to the client (step IX).

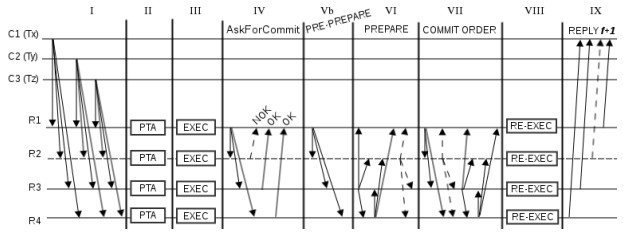

Fig. 3. Non optimistic execution, and replica R2 is malicious.

V. OB-STM TRANSACTIONS

In this section we present the life cycle of transactions, global and local verification, and the Parallel Transactional Analyzer (PTA).

A. Life cycle of transactions

Figure 4 presents the life cycle of transactions, which is explained below.

BEGIN - Every transaction begins by a client request, and when the request is received by the replicas, the transaction is forwarded to the PTA. The PTA performs conflict analysis and

Algorithm 3 Byzantine Replica

```
1: upon: receive(⟨PRE-PREPARE⟩, Ti, Ri, Ci, ORDER) from replicas
2:     if (PRE-PREPARE was accept) then
3:         for all (R ∈ ∏^R minus his own replica) do
4:             TO-Deliver(⟨PREPARE⟩, T, Ri, Ci)
5:         end for
6:     end if
7:
8: upon: receive(⟨PREPARE⟩, Ti, Ri, Ci, ORDER) from replicas
9:     if (PREPARE was accept) then
10:        for all (R ∈ ∏^R minus his own replica) do
11:            TO-Deliver(⟨COMMIT-ORDER⟩, Ti, Ri, Ci)
12:        end for
13:    end if
14:
15: upon: receive(⟨COMMIT-ORDER⟩, Ti, Ri, Ci, ORDER) from replicas
16:    if (COMMIT-ORDER was defined) then
17:        BCO ← ORDER
18:        Tx ← getFirst(BCO)
19:        if ((Tx ≠ Ti) then
20:            Call RE-EXEC (Ti, Ci)
21:        else
22:            if (Ti is running) then
23:                wait Ti is committing
24:            end if
25:            Call CommitT (Ti, Ci)
26:        end if
27:    end if
28:
29: function RE-EXEC(Ti, Ci)
30:    set Ti as running
31:    begin execution
32:    if (Ti to try commit) then
33:        BCO ← BCO \ Ti
34:        Call CommitT (Ti, Ci)
35:    end if
36: end function
```

if there are not conflicts, the transaction is executed. If conflict arises, the transaction is set to *waiting* state. Each transaction is marked with an unique identifier UUID, which is created by the client. The client can send only one request at a time and has to wait for a response to send another request.

WAITING - Transactions on the *waiting* state are transactions that contain data conflicts. After committing transactions in the execution state, the first transaction in the *waiting* buffer is sent to execution. If a transaction arises while there are transactions in execution and if data conflict exists, it will be put in *waiting* state; if not, the transaction will be executed.

EXECUTING - The transactions in this state are transactions that have no data conflicts and are in execution. After its execution, the transaction needs to commit the changes data. When the protocol tries to commit the changed data, the replica will send a message to its peers, asking permission to commit. Afterward, the transaction goes to *committing* state.

COMMITTING - At this phase, replicas wait for (*3f OK!*) reply messages to commit. If the requesting replica receive less than $3f$ replies, or receive a non-permission (*NOK!*), the Byzantine protocol is started. This phase has a timeout (line 53 algorithm 2) for prevent denial of service attack.

BYZANTINE - The Byzantine phase adopts PBFT [11] as Byzantine Fault Tolerance. In case of out-of-order messages or malicious replicas the Byzantine protocol initiates.

RE-EXEC - After Byzantine protocol execution, all replicas receive an ordered set of transactions to be executed and confirmed. Each replica needs to compare the order proposed by the Byzantine protocol with its own list of transactions. In

case of differences, re-execution of transactions in the correct order is necessary. Otherwise, re-execution is not necessary.

COMMITTED - Upon receiving (*3f OK!*) messages, transaction can be confirmed, and a response is sent to the client. Then, the next transaction in the waiting state can start execution.

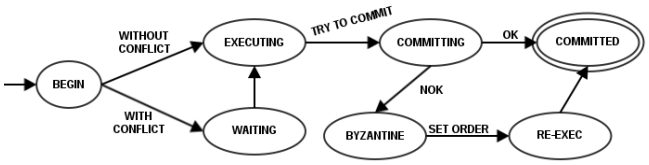

Fig. 4. Life cycle of transaction

B. Local Verification

In previous works [3], [5], [6], local verification phase occurs in commit time. Our approach does not use deferred verification, we use anticipated verification instead. The Parallel Transactions Analyzer (PTA) does this verification before the transaction execution. The verification process is composed by a comparison between transactions from the perspective of the operations requested by it, which are listed on the writings and readings sets (line 16 algorithm 2). Once the verification is done, the transaction will be set to *Executing* or *Waiting* state.

C. Global Verification

The global verification only occurs when the transaction enters on the *committing* phase. In this phase a message $\langle AskForCommit \rangle$ is sent to all peers asking permission to commit. If the requesting replica receives $3f$ confirmations about the solicitation $\langle AskForCommit \rangle$, the commit can be accomplished. Otherwise the Byzantine protocol needs to be initiated for define the order of execution and commit of transactions on all replicas.

The verification happens as the following: suppose that all replicas are running a transaction denominated *Tx*. When the replica *R1* tries to commit the transaction *Tx*, a message *AskForCommit(Tx)* is sent for all peers on the set $\prod^{T}$. Upon receiving the request *AskForCommit(Tx)*, the replica does the following verifications: is the transaction *Tx* being executed? If so, it returns *OK!*. Otherwise, the replica verifies if *Tx* was committed, if confirmed it returns *OK!*. Otherwise, *NOK!* is returned, which means that the transaction *Tx* can not be committed.

D. Parallel Transactional Analyzer

Each new transaction that arrives on replicas is verified with preceding outstanding transactions, and sent to execution if it is not conflicting. The new request is put in a waiting state if it has a conflict.

We say that two transactions are *conflicting* if the write-set of one has at least one state variable in common with the write-set or read-set of the other. More formally, we define conflict as follows: Transaction *Ti* with write-set WS(Ti) and read-set RS(Ti), and transaction *Tj* with write-set WS(Tj) and read-set RS(Tj), are *conflicting transaction* if any of the following conditions is true (1) (WS(Ti) $\cap$ RS(Tj) $\neq \emptyset$), (2) (WS(Tj) $\cap$ RS(Ti) $\neq \emptyset$) or (3) (WS(Ti) $\cap$ WS(Tj) $\neq \emptyset$).

VI. Evaluation and Results

We now show the results of an experiment and we evaluate the performance achieved by the OB-STM. The tests were made using a computer equipped with a Slackware Linux 2.6.37.6 SMP i686 Intel(R) Core(TM)2 Quad CPU Q9650 3.00GHz GenuineIntel GNU/Linux with 3GB memory. We used an unique node with one socket for each replica in both approaches. Digital signatures were used with keys of 512 bits for the communication between clients and replicas.

The figures 5, 6, 7 and 8 represent the results of 20 seconds of execution. The application was a shared counter and the results were obtained of the average of three executions increasing the counter. The clients were simulated using threads, varying from 1 to 32. We do not compare D2STM and Zhang work results with ours because we could not reproduce their work.

Figure 5 compares OB-STM and Tazio, in the optimistic case, considering one client making requests. In the cases with up to seven replicas, OB-STM performs better than Tazio, what changes according to the increasing number of replicas. The decrease of OB-STM performance occurs because whenever the number of replicas increase, more messages are exchanged and the probability of messages being delivered on the same order reduces. When out-of-order messages are delivered, the Byzantine protocols needs to be started.

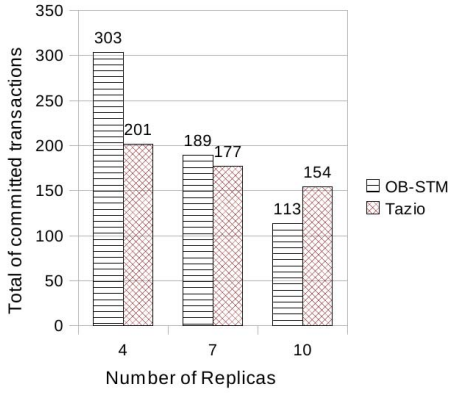

Fig. 5. Number committed transactions

In the analyzed cases, the protocol is tolerating one fault, which requires four replicas.

The scalability is shown in Figure 6, where a high contention scenario is evaluated. In this scenario the amount of committed transactions is increased since the PTA previous analysis allows the best decision to be taken. For example, it is better to do a batch of conflicting transactions instead of submitting each conflicting transaction to the execution, because this reduces the overhead of the agreement protocol [10].

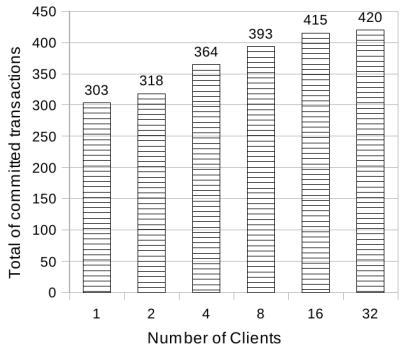

Fig. 6. Scalability

The figure 7 shows the average of $committed - transactions/second$. The average of committed transactions is proportional to the number of clients, achieving a saturation point at 32 clients.

Figure 8 depicts the number of $committed - transactions/client$. Although the curve decreases as the number of clients grow, the final result of committed transactions using more clients is better than with fewer clients. For example, the number of committed transactions with two clients is 318 (average of 159.17 per client); with four clients, 364 (average of 91.08 per client) transactions are committed in total. The total number of committed transactions increases by 46 in this case, despite the increasing contention.

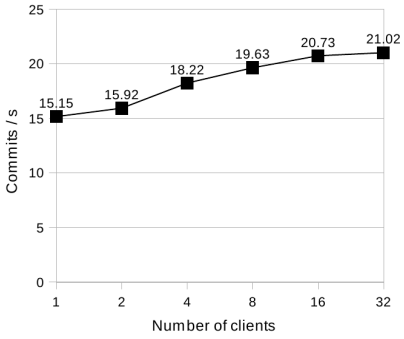

Fig. 7. Transaction committed / second

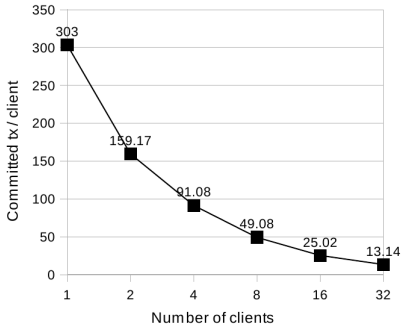

Fig. 8. Transaction committed / client

VII. Conclusions

We present an Optimistic Byzantine Fault Tolerant architecture for Software Transactional Memory. The proposed protocol uses a Parallel Transactional Analyzer (section V-D) to verify conflicts between transactions, which allows transactions that does not conflict to be executed in parallel. As there is a high probability that the messages are delivered ordered, the protocol begins with an optimistic approach. Whenever the messages are not ordered, the Byzantine protocol arranges the transactions and executes them. On smaller scenarios and using an optimistic approach, the protocol has a better throughput then Tazio. When the number of replicas increases, the protocol does not have better performance in regards to Tazio. Because of the number of messages the protocols requires, the performance of the protocol decreases while the number of replicas increases.

Acknowledgments: Supported in part by Brazilian National Research Council (CNPq) through process 560258/2010-0 and Coordination of Improvement of Higher Level Personnel (CAPES) through process 400511/2013-4 PVE A039.

References

[1] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 4, no. 2, pp. 63–71, 1996.

[2] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[3] M. Couceiro and P. Romano, "D2STM: Dependable distributed software transactional memory," *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on. IEEE, 2009.*, 2009.

[4] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "DiSTM: A Software Transactional Memory Framework for Clusters," *37th International Conference on Parallel Processing*, pp. 51–58, Sep. 2008.

[5] A. Reale, E. Savioli, and A. Sorbini, "Tazio: Una Memoria Software Transazionale Distribuita affidabile per Java," 2010. [Online]. Available: http://code.google.com/p/tazio

[6] D. Sciascia and F. Pedone, "RAM-DUR: In-Memory Deferred Update Replication," *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pp. 81–90, Oct. 2012.

[7] H. Zhang and W. Zhao, "Concurrent Byzantine Fault Tolerance for Software-Transaction-Memory Based Applications," *International Journal of Future Computer and Communication*, vol. 1, no. 1, 2012.

[8] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *Knowledge and Data Engineering, IEEE Transactions on*, no. 4, 2003.

[9] F. Pedone and A. Schiper, "Optimistic Atomic Broadcast," *Distributed Computing. Springer Berlin Heidelberg*, no. 95, pp. 318–332, 1998.

[10] R. Kotla and M. Dahlin, "High throughput Byzantine fault tolerance," *International Conference on Dependable Systems and Networks, 2004*, pp. 575–584, 2004.

[11] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[12] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, vol. 63, no. 2, pp. 172–185, 2006.

[13] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Distributed Computing*. Springer, 2006, pp. 284–298.

[14] R. Guerraoui and L. Rodrigues, *Reliable Distributed Programming*. Springer Verlag, Berlin, 2006.

[15] L. Lamport and M. Fischer, "Byzantine generals and transaction commit protocols," Technical Report 62, SRI International, Tech. Rep., 1982.