Simulation of Partial Replication in Distributed Transactional Memory

Diogo Lima[1,2]
[1] Escola Superior de Hotelaria e Turismo do Estoril, Portugal
Email: dlima@lasige.di.fc.ul.pt

Hugo Miranda[2]
[2] LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
Email: hamiranda@ciencias.ulisboa.pt

François Taïani[3]
[3] Université de Rennes 1, IRISA, ESIR, Rennes, France
Email: francois.taiani@irisa.fr

***Abstract*—Distributed Transactional Memory (DTM) is a concurrency mechanism aimed at simplifying distributed programming by allowing operations to execute atomically, mirroring the well-known transaction model of relational databases. DTM can play a fundamental role in the coordination of participants in mobile distributed applications.**

Most DTM solutions follow a full replication scheme, in spite of recent studies showing that partial replication approaches can present gains in scalability by reducing the amount of data stored at each node. This paper investigates the role of replica location in DTMs. The goal is to understand the effect of latency on the DTM's system performance in face of judicious replica distribution, taking into consideration the locations where data is more frequently accessed.

I. Introduction

To mitigate mobile devices' computing power limitations, applications can delegate their most computing intensive tasks on external servers, creating mobile distributed applications. Such applications need to mediate the interaction between a variety of actors, efficiently managing and exchanging state. On the other hand, the system can benefit from a replica deployment in proximity to the mobile devices, which allows reducing latency from data access. This implies the usage of concurrency control mechanisms that guarantee a safe access by different actors to the same dataset.

Concurrent programming is a challenging task. Developers need to guarantee consistent access to information by multiple threads or processes. The traditional solution to ensure such safe access relies on using locks or semaphores, which carry a number of well-know software engineering challenges. *Transactional Memory* is a concurrency mechanism aimed at simplifying the development of concurrent applications by allowing operations to execute in an atomic way. Analogous to database transactions, transactional memory defines a specific sequence of tasks that are considered a transaction, that either commits (and all changes produced by the tasks become visible) or aborts (and no change is visible). Originally proposed as a hardware architecture [1] for shared memory access, Transactional Memory was later extended to address parallelism in multiprocessor systems, being named as *Software Transactional Memory* (STM) [2], [3].

Distributed Transactional Memory (DTM) [4]–[11] has been motivated as an extension of STM to distributed systems, addressing new issues such as data replication and node failure. It provides an additional abstraction level to the programmer, where traditional distributed programming details (e.g. socket management or data serialization) become transparent and integrated with concurrency in a unique and coherent approach. Most of the existing DTM solutions follow a full replication scheme where all nodes in the system keep a replica of every object. However, recent studies show that the reduction of data stored at each node provides to the partial replication approach additional scalability gains [9].

Motivated by recent commercial applications, in particular, large scale, augmented reality games (i.e Ingress and Pokemon Go), this paper assumes mobile distributed applications running over multiple servers and client mobile devices, sharing data. In such scenario, servers and the client mobile devices are expected to be geographically dispersed among different regions. Intuition suggests that, as participants may be more geographically dispersed, the latency and message concurrency to validate and commit transactions between a group of nodes increase, hampering system performance.

This paper evaluates the role of latency in partial replication in the context of DTM. In particular, it shows that the system's performance is affected by the node partitioning policies applied.

II. Related Work

Implementing a transactional memory framework requires a considerable software engineering effort, for example in defining how applications can declare the scope of transactions. In most systems (e.g. [8], [9], [12]–[14]) byte-code is rewritten, changing the underlying virtual machine or compiler to interpret the "@Atomic" method annotation that delimits the transactions. Transactional code usually provides methods to *start*, *abort* and *commit* transactions, maintaining isolation, as found in relational databases. Libraries are extended to include a validation algorithm, needed to decide which concurrent transactions to commit and which to abort. This algorithm must provide an appropriately consistent view of the shared memory to all participants, either static or mobile. There are several alternatives to guarantee such consistency.

A. Single-copy Model

In single-copy model [5], [11], there is only one writable copy of each object in the system and transactions are serial-

ized so that there is at most one manipulating those objects in each moment. This can be achieved by preventing other transactions to access data through distributed queuing.

Unfortunately the latency to acquire the objects of each transaction may be an issue. Moreover, the single-copy model is inherently non fault-tolerant: the failure of an object's owner means the object becomes unreachable to the rest of the system, at least until some root node assigns a new owner to the object and recovers its state from some previous version.

B. Multiversioning Models

Multiversioning models trade off the complexity of detecting collisions with the latency in the execution of read-only operations. Multiversioning consists in maintaining different versions of each data item and can be achieved by either having different versions of the same object on different nodes (replica-based), or by keeping a history of each object's updates (history-based).

1) Replica-based multiversioning: To maintain a consistent order over committed transactions, the *Distributed Multiversioning* (DMV) system [7] requires each transaction to obtain a unique system-wide token at the beginning of its execution. At commit time, a writing transaction follows a voting protocol where all nodes agree before the transaction is able to successfully commit. Different versions of the same data item arise from the fact that the voting nodes explicitly delay the application of remote transactions updates to decrease the chance of invalidating the snapshot of the currently active local read-only transactions.

However, due to the system wide token acquisition at the beginning of each transaction execution, transaction serialization still exists in DMV. The token allows a consistent transaction order, but represents a considerable overhead and transaction serialization hampers concurrency of the DTM system.

2) History-based multiversioning: History-based multiversioning was originally proposed in the context of the JVSTM framework [12] to provide concurrency control for multiprocessor computers. JVSTM follows a multi-version concurrency control scheme using the versioned box (VBox) abstraction that keeps a history of values for each object. The VBox is a container that keeps a sufficient number of versions of each transactional data item so that read-only transactions are never aborted. Each version contains the changes made by successfully committed transactions and the timestamp of the corresponding transaction.

This scheme was extended to the context of DTMs in the *Dependable Distributed Software Transactional Memory* (D^2STM) system [4], built on top of JVSTM. D^2STM was motivated as a fault-tolerant DTM, following a fully replicated scheme. All data items are replicated to all nodes and updating transactions are validated through a non-voting certification scheme, where both the write-set and read-set of a transaction need to be atomically broadcast to all other nodes. Since the authors expect the read-set to be larger than the write-set, the D^2STM protocol encodes the transaction's read-set using Bloom Filters to reduce the communication overhead. The read-set is validated against transactions that have committed since the beginning of the committing transaction, and update transactions are validated once their broadcasts are delivered.

C. Clock Validation

The DTM framework *HyFlow* [8] introduced a new validation algorithm, called *Transactional Forwarding Algorithm* (or TFA) [15], based on the *happened before* ordering through the use of Lamport' logical clocks [16].

In TFA each node maintains a local clock which is incremented whenever a local transaction successfully commits. This solution also relies on object versioning. However, in contrast with other approaches, objects' versions are based on the value of the node's local clock at the time of the last update of that object instead of a globally defined clock. When an object is accessed in the scope of a transaction, the object's version is compared to the transaction's starting time. If the object's version is newer than the transaction's starting time, the transaction is aborted and restarted as it indicates that some other transaction using the object has committed.

Clock values are included in all messages sent by a node. If a remote node's clock value is older than the received value in a transaction request, the remote node advances its clock to the newer value. Instead, if the local node observes in the reply message that the remote node's clock is newer, the local node must execute an early validation, called the *transactional forwarding* operation. This operation evaluates if none of the objects in the transaction's read-set have been updated after the transaction's starting time. If true, the operation advances the transaction's starting time and the latter can proceed. Otherwise the transaction is aborted and restarted.

III. Replication in DTM

Replication of objects in DTMs can serve two purposes: to improve availability in the presence of faults and to improve performance by making the data locally available at the interested nodes.

A. Full Replication

All the solutions discussed in Section II-B follow a full certification-based replication scheme: all nodes in the system keep a replica of every object and transactions are locally executed, synchronizing objects state with the other replicas at commit time. This synchronization can be achieved either through a voting or non-voting certification. As observed in the DMV system [7], in the voting certification approach, a committing transaction needs to broadcast its updates to the other nodes and will only commit if they vote favorably.

On non-voting approaches however, a communication round is saved as the decision can be taken locally and therefore, replicas do not need to reply to the transaction's issuer. The need to vote is exchanged in a trade-off with the amount of data provided in the first round [17]. In particular, the transaction owner must provide both the transaction's write and read sets. The D^2STM [4] follows a variant of this non-voting certification-based replication, where bloom filters are used to reduce the size of broadcasted messages.

Since the non-voting certification approach allows replicas to independently validate transactions and every data item is replicated among all nodes, the failure of nodes in the system does not harm consistency. However, coordination of all nodes imposes a considerable communication overhead. Namely, broadcasting transactional read/write sets is inherently non-scalable, as messages broadcasted grow quadratically with the number of nodes present in the system [6].

B. Partial Replication

In partial replication, the full application's dataset is subdivided into *n* partitions and each partition is replicated in a group of *m* nodes. Partial replication is more scalable as committing transactions only need to reach the groups storing data items accessed in the transaction.

To the best of our knowledge, SCORe [18] is the only partial replication protocol developed for DTM systems. SCORe combines the Two Phase Commit (2PC) algorithm [19] with Skeen's total order multicast [20] to form a commit protocol that ensures that only the replicas that maintain data accessed by a transaction participate in its outcome. SCORe relies on logical clocks where each node keeps two scalar timestamps: the *commitId* which is the timestamp of the last update transaction committed on that node, and the *nextId* which indicates the next timestamp the node will propose for a remote commit request.

At commit time, the transaction issuer triggers a 2PC instance by total order multicasting a validation message to all involved replicas. Every replica that receives this message validates the transaction by attempting to acquire exclusive and shared locks for the transaction's write and read sets, respectively. If the validation is confirmed, the *nextId* is piggybacked on the reliably unicasted *vote* message and the transaction is locally stored in a pending buffer. The transaction issuer then collects all *vote* messages (aborting the transaction in case one of the contacted node does not respond within a predefined timeout), sets the transaction's final commit timestamp as the maximum of the proposed *nextId* and multicasts back the *decide* message with the transaction's outcome and the *commitId*. If the outcome is positive, the receiving replicas buffer the transaction in a queue of stable transactions. Otherwise, the transaction is aborted and the previously acquired locks are released. A transaction T is finally committed only if there are no other transactions in both pending and stable buffer with a timestamp less that T's *commitId*.

However, the distribution of the data items by replication groups follows a pseudo-random algorithm and therefore does not exploit data and node partitioning to its full extent. For example, as participants are more geographically disperse within a group, the increased latency and message concurrency induced to validate and commit transactions may hamper the system's performance. By exploring more judicious distribution of replicas such as the locations where data are more frequently accessed, partial replication can more actively contribute to improve DTMs' performance.

Geographical distribution has already been addressed in distributed transactional SQL databases such as CockroachDB [1], where the user is able to define replica locations. CockroachDB builds its SQL database on top of a transactional and strongly consistent sorted monolithic key value store map. Each record on that key value store represents a column value in a row of a SQL table, and consists on a triplet *<key;commit timestamp; value>*. The key value store is then partitioned into continuous *ranges* that are distributed among replicas, guaranteeing that each range is replicated in at least 3 replicas.

However, replica location in CockroachDB is based on the types of failures a user wants to tolerate, e.g. replication in different servers within a datacenter to tolerate power failures, or different servers in different datacenters to tolerate large scale network or power outages. In opposition, we aim at using replica location to reduce latency and improve system performance by storing data in partial replication groups formed in proximity to their users.

IV. Evaluation

To understand the impact of replica geographical distribution in partial replication DTMs, we prepared a simple scenario where group replicas are either kept in the same server or interconnected via an Ethernet network.

For the experiments we assume a vehicle traffic scenario, where two adjacent regions of a city are managed by a pair of traffic controllers, thus creating a distributed application with 4 processes. Each region is further divided in 3, 30, 50 or 500 locations. The (simplified) role of these controllers is to receive location advertisements from vehicles and to consistently move them from one location to another, ensuring that the number of vehicles in the system is constant.

In the experiments, each process simulates 10000 vehicle location transfers. This scenario is modeled in a DTM application with partial replication by associating the data of each region to an integers array, managed by a partial replication group. Each element of the array represents the number of vehicles in one of the locations of the region associated to the array. Therefore, DTM transactions consist in transferring one unit between array elements, knowing that a transfer between elements of the same array represents a transfer between two locations of the same region (i.e. partial replication group). The implementation does not repeat aborted transactions. This is in contrast with the expected behaviour of a realistic system but provides relevant evaluation metrics.

Experiments were conducted on two servers. One (hereafter named Nonio) has a Dual-Core Intel Xeon 3060 CPU at 2.4GHz and 2GB of RAM. The other (Pati) is a Quad-Core Intel Xeon X3370 CPU at 3.0GHz and 8GB of RAM. Hardware heterogeneity is not considered to negatively impact the results given that results only consider the overall system performance. Servers run a Debian GNU/Linux v6.0.10 operating system and a Java Virtual Machine v. 1.6.0_26. Evaluation was conducted using the ReDstm framework [9]

[1] https://www.cockroachlabs.com/

with the SCORe partial replication multi-version algorithm. JGroups [21] provides the underlying group communication service. All simulations equally divided the 4 processes by the 2 servers. Two distinct combinations can be devised. These experiments are named in the form (G_G), where G will list the name of the servers hosting processes for partial replication group G. Following this rationale, in NP_NP experiments, each partial replication group process is run on a distinct server. Correspondingly, in NN_PP experiments, each server will host the two processes of the same group. NN_PP experiments are expected to reflect a lower latency given that the processes of each partial replication group can communicate using in-memory networking features of the Linux kernel.

Throughout this evaluation, we are interested in comparing the system's performance in the presence of distinct probabilities of a transaction staying in the issuer's partial replication group. Two configurations were simulated:

- 70-10 With 70% probability, a transfer will be between two locations managed by the issuer's partial group controller, 20% of the transfers are made between locations of different regions and 10% between locations not managed by the controller.
- 10-70 There is only 10% probability of a transfer being between two locations managed by the issuer's partial group controller, and 70% between locations managed by the other partial group controller. The value of 20% is kept for transactions made between locations of different regions.

All results presented are the average of 10 simulations, performed in comparable conditions. Plot's error bars show the distance of the average to the standard deviation. It should be noted that the uneven distribution of transfers performed by the controllers will force each process to interact more frequently with one of the partial replication groups (regions).

A. Commit Ratio

The proportion of transactions that successfully committed is depicted in Fig. 1. The simulations confirm that the throughput improves as the number of available location increases. From 30% of committed transaction with 6 possible locations to 98% with 1000 locations. Having more locations means that there is a higher probability that transactions do not randomly select the same objects, avoiding transaction conflicts. Moreover, all simulations converge to a similar throughput in respect to the worst and best conflicting scenarios (with 6 and 1000 locations, respectively) meaning that, at those extremes, is the system's concurrency level the major factor impacting performance. However, in the more balanced scenarios of 60 and 100 locations results are more dispersed. The commit ratio of both 10-70 tests are near identical, however they remain consistently above the *NN_PP* 70-10 test with around 2,5% more committed transactions and more 4% than the *NP_NP* 70-10 configuration. This result looks, at first glance, surprising as suggestion would indicate that the system would benefit from keeping most of its transactions in the same partial replication group with lower latency.

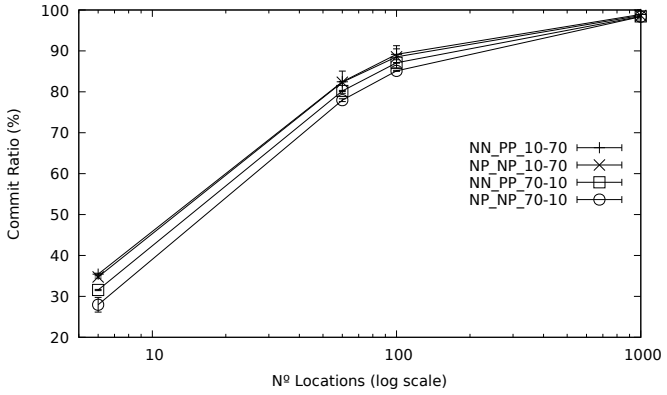

Fig. 1. Ratio of committed transactions upon the number of locations available.

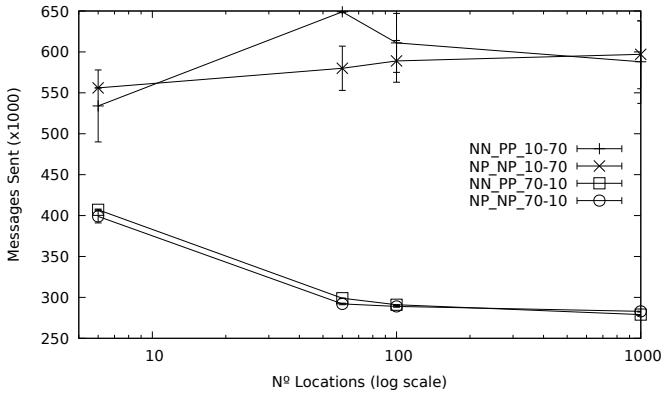

Fig. 2. Total number of messages sent.

B. Network Traffic

In order to understand the previous result, we further investigated the total number of messages sent by the system and the average waiting time for a node to collect all validation responses for a committed transaction.

Fig. 2 depicts the volume of messages sent by the system. Nodes can send the following type of messages: data read request, data read return, transaction validation request, transaction validation vote, and transaction final decision. We observe that, for the 70-10 configurations, as the concurrency level drops by having more locations available for transactions, the volume of messages sent drops as well. This can be explained by the fact that, with higher concurrency levels, multiple transactions access the same object. Once one commits, all the remaining abort and extra read requests and responses are created in the system. In opposition, as more locations are available, it is more likely to have non concurrent transactions that already have the most up-to-date data version locally, thus preventing the need to send additional requests.

On the other, the 10-70 configurations show the exact opposite behaviour. In fact, as the number of locations increase the volume of messages sent also increases. This can be explained by the fact the majority of the transactions issued manipulate data that the originating node does not have, so

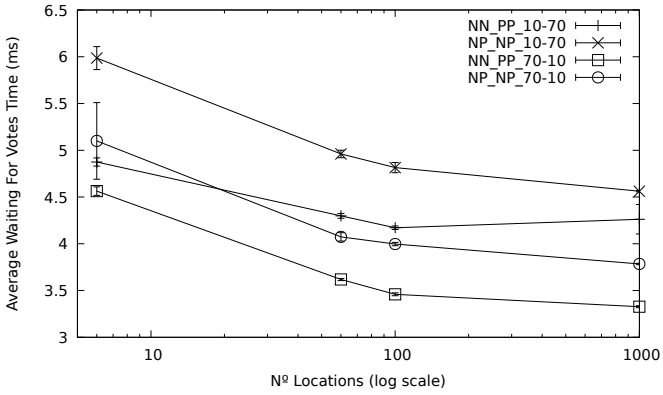

Fig. 3. Average waiting time to receive all validation votes for committed transactions.

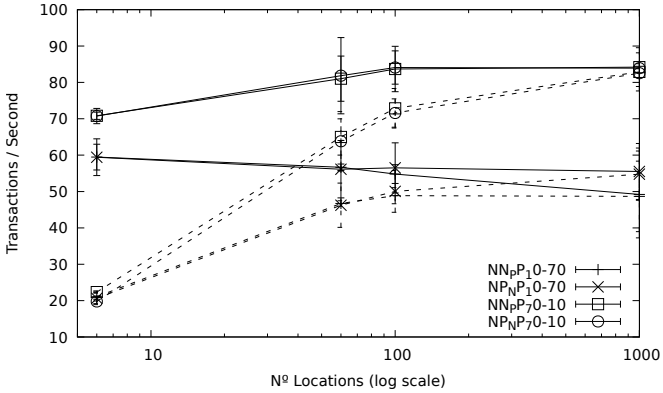

Fig. 4. Average test duration. Continuous lines represent transaction throughput and the dashed lines represent their respective commit throughput.

read requests are necessary in the majority of the transaction to obtain that data from the other partial replication group. This is confirmed by the 2/1 ratio of messages sent from the 10-70 to the 70-10 tests: each transaction manipulates two objects, thus the 10-70 scenario need to read request those two objects before all other transaction related messages. In each configuration, the volume of messages sent remains similar for the 60, 100 and 1000 location scenarios, which is expected, as the workload (i.e., the number of transactions) remains similar for all simulations. The exception is the worst concurrency level scenario (6 locations available) where the number of messages increase 100 000 (i.e. 2.5 messages per transaction) for the 70-10 tests and decreases around 75 000 messages for the 10-70 tests (i.e. 1.9 messages per transaction). This can be explained by having more conflicts, which lead to a greater number of aborts. In the case of the 70-10 tests, this results on requiring more messages to obtain the most up-to-date versions of two conflicting objects manipulated in a transaction. For the 10-70 tests, the volume of read requests remains constant, however the volume of messages sent is reduced by having fewer validation and commit phases since nodes are able to detect early that their objects are already outdated and that the respective transaction must be aborted. Nevertheless, Fig. 2 proves that the volume of message sent is not directly influencing the system's throughput.

Fig. 3 depicts the average waiting time for a node to collect all validation responses for a committed transaction. Recall that the SCORe algorithm relies on a voting certification to validate transactions. When taking in consideration the group composition, we can observe that *NN_PP* configurations are on average 0.5 ms faster than the *NP_NP* configurations. This result evidences the penalty of latency in partial replication groups. On the other hand, the waiting vote period for the 70-10 configurations remain consistently below the 10-70 tests. This result emerges as the consequence of the increasing network traffic induced in the 10-70 scenarios.

However, Fig. 3 also does not explain the counter intuitive commit ratios obtained in Fig. 1 as waiting for a longer period of time to validate a transaction would theoretically increase the probability of having concurrent transactions modifying the accessed data items and leading the transaction to abort due to an inconsistent data view.

C. Transaction and Commit Throughputs

The transaction throughput is defined by the ratio of transactions issued divided by test duration. The commit throughput is the ratio of committed transactions per test duration. Results are depicted in Fig. 4. Each scenario is represented by an unique symbol, while the continuous lines represent their transaction throughput and the dashed lines their respective commit throughput.

Fig. 4 shows that transaction throughputs tend to stabilize as the number of available location increases, indicating the presence of an upper bound that can not be surpassed by the simulations in respect of the number of transaction issued per second. On the other, commit throughputs increase with more locations available, as expected, until converging to the transaction throughput upper bond. As observed in Fig. 1, this is explained by the 98% commit ratio for the more concurrency favourable 1000 location test case, where almost every transaction issued is a committed transaction.

However, Fig. 4 clearly distinguishes 70-10 from 10-70 configurations. The 70-10 configuration is able to respectively issue and commit 20 more transactions per second than the 10-70 configuration on average. This means that the same workload (40 000 transactions) takes approximately an additional 150 seconds to be completed in the 10-70 than in the 70-10 configuration, indicating that the latter is preventing concurrency in the system. In fact, as shown in Fig. 1, the 10-70 configuration is able to commit more transactions in absolute number, however the system's performance is hampered since the same workload takes 30% more time to executed in such configuration. Thus, the system does benefit from locality, i.e. keeping group replicas close, since configurations that heavily rely on the network have smaller transaction and commit throughputs.

V. Future Work

The evaluation showed that replication location directly influences partial replication DTM's performance. Both the

number of transaction and successfully committed transaction throughput improve when members of the same partial replication group have lower latency. However, such model brings additional research challenges. A first research challenge to be addressed in the scope of this work consists in extending the DTM interfaces to efficiently and dynamically map resources on partial replication groups.

A second research challenge consists in optimizing the underlying group communication service, benefiting from the partial groups defined at the DTM level. A technique that will be pursued consists in having different groups sharing common resources, a technique previously named *Light-Weight Groups (LWGs)* [22]. The idea is to create a light-weight group abstraction where many groups that share common characteristics are mapped to the same underlying virtually synchronous group. LWGs would lead to a two-tier group membership, where full and partial membership groups can share resources, reducing the number of messages delivered to each member.

Moreover, this two-tier architecture would also be helpful for highly dynamic applications where the objects manipulated and the membership suffer frequent changes, such as in mobile applications. Membership change is a time and resource consuming operation, likely to impact performance due to its mandatory definition of synchronization points. LWGs are expected to play an important role as well as they are able to mitigate the impact of changes in the upper tier of group membership, hiding them from the lower tier, which would include all the participants and where membership changes negatively impact the overall system performance.

VI. Conclusion

This paper discusses the role of replica location in DTMs, in order to understand if a judicious distribution of the replicas, that takes into consideration the locations where data is more frequently accessed, can contribute to improve DTMs' performance. The paper evaluated the role of node and data distribution in partial replication. In particular, we observed that the system's performance is affected by the replica location, where configurations that heavily rely on the network have smaller transaction and commit throughputs.

Results show that locality plays a major role in a transactional systems, and that it should be exploited at group configuration level. A possible solution that we intend to explore is the possibility of implementing the concept of *Light-Weight Groups* in the underlying communication service in order to optimize groups membership operation costs.

Acknowledgment

Work described in this paper was partially supported by Fundação para a Ciência e Tecnologia, Portugal, under project PTDC/EEI–ESS/5863/2014 - doit.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[2] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *Proc. of the 22nd Annual Symposium on Principles of Distributed Computing (PODC'03)*, 2003, pp. 92–101.

[3] M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory," *SIGPLAN*, vol. 41, no. 10, pp. 253–262, Oct. 2006.

[4] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Proc. of the 2009 15th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC'09)*, 2009, pp. 307–313.

[5] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," in *Proc. of the 19th Int'l Conference on Distributed Computing (DISC'05)*, 2005, pp. 324–338.

[6] J. Kim and B. Ravindran, "Scheduling transactions in replicated distributed software transactional memory," in *Proc. of the 13th IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2013, pp. 227–234.

[7] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster," in *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, 2006, pp. 198–208.

[8] M. M. Saad and B. Ravindran, "Hyflow: A high performance distributed software transactional memory framework," in *Proc. of the 20th Int'l Symposium on High Performance Distributed Computing (HPDC'11)*, 2011, pp. 265–266.

[9] J. a. A. Silva, T. M. Vale, R. J. Dias, H. Paulino, and J. a. M. Lourenço, "Supporting multiple data replication models in distributed transactional memory," in *Proc. of the 2015 Int'l Conf. on Distributed Computing and Networking (ICDCN'15)*, 2015, pp. 11:1–11:10.

[10] A. Turcu, B. Ravindran, and R. Palmieri, "Hyflow2: A high performance distributed transactional memory framework in scala," in *Proc. of the 2013 Int'l Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'13)*, 2013, pp. 79–88.

[11] B. Zhang and B. Ravindran, "Relay: A cache-coherence protocol for distributed transactional memory," *Principles of Distributed Systems*, pp. 48–53, 2009.

[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] G. Korl, N. Shavit, and P. Felber, "Noninvasive concurrency with java stm," in *Workshop on Programmability Issues for Heterogeneous Multicores*, Jan. 2010.

[14] M. M. Saad and B. Ravindran, "Distributed hybrid-flow stm," , Tech. Rep., Dec. 2010.

[15] M. Saad and B. Ravindran, "Transactional forwarding: Supporting highly-concurrent stm in asynchronous distributed systems," in *24th Int'l Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2012, pp. 219–226.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[17] B. Kemme and G. Alonso, "A suite of database replication protocols based on group communication primitives," in *Proc. of the 18th Int'l Conference on Distributed Computing Systems*, May 1998, pp. 156–163.

[18] S. Peluso, P. Romano, and F. Quaglia, "Score: A scalable one-copy serializable partial replication protocol," in *Proc. of the 13th Int'l Middleware Conference (Middleware'12)*, 2012, pp. 456–475.

[19] J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*, 1978, pp. 393–481.

[20] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[21] B. Ban, "Design and implementation of a reliable group communication toolkit for java," Tech. Rep., 1998.

[22] L. Rodrigues, K. Guo, P. Verssimo, and K. P. Birman, "A dynamic light-weight group service," *Journal of Parallel and Distributed Computing*, vol. 60, no. 12, pp. 1449 – 1479, 2000.