FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 38 (2013) No. 2

DOI: 10.2478/fcds-2013-0003 ISSN 0867-6356
e-ISSN 2300-3405

DESIGNING A SOFTWARE TRANSACTIONAL MEMORY FOR PEER-TO-PEER SYSTEMS

Aurel PAULOVIČ *, Peter LACKO †

Abstract. Transactional memory is a rather novel approach to concurrency control in parallel computing, that has just recently found its way into distributed systems research. However, the research concentrates mainly on single processor solutions or cluster environment. In this paper we argue, that peer-to-peer systems would require a different design of transactional memory because of the increased failure-rate of nodes, slower network and possibility of network splits. We also present a few of our design ideas, namely increased performance and fault tolerance through the use of higher-level conflict detection and resolution via abstract data types and eventually consistency, that as we think could be important to a successful implementation of a scalable and resilient transactional memory.

Keywords: distributed transactional memory, STM, peer-to-peer, eventual consistency, abstract data types

1 Introduction

With the rise of parallel and distributed applications, concurrency becomes developer's daily bread and butter. Traditionally, the concurrency problem has been solved by using explicit locking and critical sections. While this might be straightforward in smaller applications running on a single computer, it quickly becomes complicated and error-prone as systems grow larger and more complex and the programmers have

This article is an extended version of a paper from the ADBIS PhD Consortium 2012 that was published by Springer [17]

*Slovak University of Technology, Faculty of Informatics and Information Technologies, Ilkovičova 3, 842 16 Bratislava, Slovakia, paulovic@fiit.stuba.sk

†Slovak University of Technology, Faculty of Informatics and Information Technologies, Ilkovičova 3, 842 16 Bratislava, Slovakia, lacko@fiit.stuba.sk

to reason about hard-to-debug problems like deadlock, livelock, priority inversion or lock convoying.

When diving into distributed systems, explicit locking gets even harder. Managing and tracking locks in the presence of node failures, latency and network splits often requires carefully crafted locking solutions using dedicated locking services and policies. In addition, locking highly contended resources across a large distributed system can easily become a bottleneck due to the introduced blocking.

Peer-to-peer (P2P) systems, as a class of distributed systems, try to provide scalability and effective sharing of computational resources. In contrast to many systems developed for cluster computing, they tend to be decentralized, geographically dispersed and inherently fault-tolerant. However, many of P2P's desired features become issues, when trying to concurrently manage and work with shared data. High node volatility (joining and leaving the network), latency, asymmetric connection speeds and network partitions, that are all much more frequent and severe than in typical cluster systems running in a data center, make concurrency control using explicit locks complicated.

Researchers try to alleviate these problems with new concurrency mechanisms and models. One of such novel approaches is transactional memory (TM). So far, TM research has mainly focused on parallelism on a single chip. In recent years attention has slightly shifted towards TM for multiprocessors and cluster computing. However, to our knowledge only a little work has been done on the topic of TM for P2P systems.

In the rest of this paper we first briefly introduce TM as a concurrency control mechanism and the notion of higher-level transaction conflict detection and resolution. We follow with a description of distributed TM and the differences between TM for cluster computing and P2P systems. Finally, we present our preliminary ideas and design proposals on a software transactional memory for peer-to-peer systems.

2 Transactional Memory

Transactional memory is a relatively new approach to managing concurrency in parallel and distributed systems, that was first practically demonstrated by Herlihy and Moss [7] as an extension to multiprocessor cache-coherence protocol, and later in software implementation by Shavit and Touitou [19]. TM uses the notion of transactions as a concurrency primitive for memory operations and is often implemented as a form of optimistic concurrency control. Generally, a transaction in TM is a finite sequence of operations that satisfies the failure atomicity and isolation properties, and allows the operations of different transactions to be executed concurrently in isolation. It either completely successfully commits, if no conflict occurred, or automatically aborts and rolls back all its changes to the state before the start of the transaction.

The goal of transactions is to free the developer from the need to use explicit locking to denote and protect critical sections that access shared variables, which could be the subject of race conditions. By removing the explicit locks, we can develop a general concurrency management that is more efficient than coarse grained

locking, yet does not expose the complications of fine-grained locking and its issues (e.g. deadlock). Also, since transactions can generally be managed at higher levels of concurrency at runtime and not at the level of locks in code, they should provide easier composability of operations that need synchronization.

2.1 Conflict detection and resolution

Traditionally TM implementations detect conflicts between transactions (read-write, write-write conflicts) using the individual memory addresses that the transactions access. In general the TM runtime records every address read by a transaction in a read-set and every write in a write-set and compares these with the read- and write-sets of other concurrently running transactions. If there is some nonempty intersection between the write-set of one transaction and the read- or write-set of another transaction, a conflict is detected and one of the transactions has to be aborted. While this is relatively straightforward and does not require the TM implementation to understand the operations performed, it can also result in false positives and unnecessary transaction aborts. For example, if one transaction were to read the last element of a singly-linked list it would have to traverse the entire list and the transaction would end up with a read-set containing all of the elements in the list. Such a transaction would then be in a conflict with every transaction that would wish to insert or remove an element anywhere in the list despite the fact that the operations might actually logically not interfere.

In order to alleviate this issue, the TM can use the notion of *Higher-Level Conflict Detection and Resolution* (HLCDR). In HLCDR the TM does not detect conflicts using the accessed memory addresses, but uses the higher semantics of the operations performed on the data structures instead. In the previous example the TM could, using the semantics of used operations, allow both transactions to succeed despite the fact that read- and write-sets of the transactions might overlap. For the TM to be able to support this, the data structures used in transactions have to declare semantics of the operations that can be applied to them. Such data types are called *Abstract Data Types* (ADT). ADTs completely encapsulate their state and expose higher-level operations together with information on their mutual commutativity and its constraints as well as their inverses, which can be used in the case of a transaction rollback. ADT and HLCDR have been explored in multiple works [11, 16, 8, 9] with focus on their performance benefits.

3 Distributed Transactional Memory

The research of transactional memory for distributed systems has been mainly driven by the rise of cluster computing, data centers and cloud. However, the design and implementation of TM that can be used in distributed environment differs from the more traditional TM for single processor systems in many aspects, from which some of the most important are:

- Nodes in a distributed system can fail and a failure of a node should not stop the whole system (failure transparency).
- Distributed systems often use replication and data items handled by a TM might be replicated on multiple nodes. Distributed TM has to manage coherency between all item replicas (replication transparency). On the other hand, replication could be exploited for the purpose of data versioning.
- Communication latency between processes running on separate nodes is many orders of magnitude higher than inter-process communication on a single node. The concurrency control overhead of TM can be diminished by the overhead of communication between remote distributed processes and TM can perform more complex algorithms to avoid conflicts.
- While overhead of TM on a single processor is usually considerably higher than the overhead of explicit locking, in a distributed system, the overhead of locking raises significantly due to potentially higher contention and can be outperformed by optimistic concurrency control performed by TM.
- Distributed applications tend to be more complex than single processor applications and concurrency control via transactions may offer easier interface than using explicit locks, while still providing better protection against deadlock, etc.

The ever-growing body of research of distributed transactional memory implementations is characteristic by its variety. Some of the TM designs focus on relatively small number of nodes while others show good performance even for large clusters.

Kotselidis *et al.* [10] created a distributed STM called DiSTM using which they analysed 3 different prototypes of distributed coherence protocols: *a*) a decentralized Transactional Coherence and Consistency (TCC) protocol, that broadcasts the read- and write-sets of a committing transaction to all nodes in the system, where they are checked for conflicts; *b*) a serialization lease protocol effectively used as a lock on the master; and *c*) a scheme using multiple leases. Their TM design, however, always uses a single master that effectively limits the scalability and reliability of their solution as a single point of failure.

The Cluster-STM presented by Bocchino *et al.* [2] was designed for large scale clusters and showed good performance for systems comprising up to 512 processors. Their implementation guarantees serializability and imposes a programming restriction, that each memory location can be accessed between two global synchronization points either only transactionally or only non-transactionally. To be able to perform well and scale to large clusters, the STM uses several key techniques. It offers a remote evaluation of transactional operations via the *on* construct in order to avoid transferring large amounts of data to the node, that is initiating the transaction; it supports multiword data movement, that allows for bulk transfers of data between transactional and local memory; and employs a novel strategy that uses a transaction descriptor distributed across multiple processors. However, their solution focuses on HPC workloads and does not provide fault tolerance.

Manassiev *et al.* [13] exploited in their Distributed Multiversioning (DMV) algorithm the natural presence of multiple data replica versions across a distributed

transactional system. They use the different replica versions to run conflicting read-only and update transactions on different nodes without having to keep different versions of data locally, which allows them to avoid many conflict waits and transaction roll-backs. Their implementation broadcasts all modifications performed during a transaction to all the nodes in the system and has to acquire their acknowledgment prior to local commit, thus is likely to be latency sensitive and not scalable to larger number of nodes.

In Sinfonia [1] Aguilera *et al.* proposed a service built on the concept of mini-transactions. A minitransaction trades programmability and general expressiveness for better scalability and performance. It does not allow to perform a sequential set of operations that can both read transactional data and then write it in the same transaction to a different location; instead the operations merely consists from a set of compare items, a set of read items and a set of write items, which are all chosen before the minitransaction starts executing. Upon commit, the minitransaction is then performed on the nodes holding the data, the compare items are compared to the actual values and if the comparison succeeds, the read- and write-items are treated as input and output of the whole transaction. The communication of the system is very efficient, but it is debatable whether the programming model is not too restrictive.

Fault tolerance in distributed STM was the focus of D^2STM [5] and the Fault-Tolerant DTM [12]. The former processes each transaction autonomously in isolation on each node and then uses non-blocking distributed certification scheme based on atomic broadcast and a novel Bloom Filter Certification (BFC) read-set encoding to validate the transaction. The later uses combination of primary, backup, cached and transactional local copies of objects and prefetching to optimize the performance and mask the latency in the networked system. While fault-tolerant, both designs have been tested only with a small number of nodes and relatively good network and it is therefore unclear how scalable they are.

Transactional memory for P2P systems, has been studied rather sparsely. Pratt-Szeliga and Fawcet [18] used Hilbert Space Filling Curves to search the P2P overlay for data without flooding. Müller *et al.* [15] presented the implementation of multiple distributed commit protocols:*a*) a peer-to-peer based approach using a forward validation with a first-wins strategy; *b*) a both forward and backward validation algorithm based on ultra-peers; and *c*) a combined solution using super-peers and an ultra-peer node. Mesaros *et al.* [14] designed a transactional system for structured overlay networks using decentralized lock-based management.

4 Differences between Cluster and P2P STM

Peer-to-peer networks typically consist of hundreds or thousands of highly unreliable, geographically dispersed peer nodes that can almost arbitrarily leave and join the network at any time. The latencies can be one or two orders of magnitude grater than those in a high-end data center and some nodes might have asymmetric connections. Based on that, we have identified a set of differences between cluster and P2P environments that could prove to be important in the design of a TM.

Nodes in a P2P system are much more likely to fail or to get split from the other nodes by a network partition. TM for P2P networks should probably focus on optimistic concurrency since detecting node failures is relatively complicated and pessimistic concurrency could introduce blocking. Also in the case that a node running a transaction fails, it should not hold any locks or otherwise block the rest of the system.

Since P2P nodes are volatile, data needs to be replicated. This can be true for cluster environment as well, however, there might be a difference between the possibilities of data replication and migration. The peer nodes are often not dedicated solely to the purpose of running the computation or application and could enforce strict limits on the memory used (and would typically have much less memory than a server in a data center) or might be unwilling to host replicated data from other peers because of privacy or performance issues.

Nodes in a P2P network could be potentially malicious. To allow more secure work with data, some resources might be limited to a privileged group of peers that have direct access to them, while other nodes would have to access the resources remotely. TM could support executing transactional operations on remote peers to allow for this scenario.

Distance and latency between nodes in a P2P system is much greater than in a cluster, also the connection speed and throughput are typically considerably lower than between nodes in a data center. The conflict detection and resolution algorithms of a TM should focus on minimizing transaction abort-rate and communication overhead. In addition, a peer network connection might be asymmetric and could prohibit transferring large amounts of data from the node to the rest of the system. TM for P2P should be able to exploit data locality and communication batching.

Because of the nature of P2P systems and their often decentralized architecture, that provides the individual peers only with a partial view of the entire network, locating data items and their replicas is much more complicated. Structured and unstructured networks would potentially require different data search and discovery services and a potential TM implementation should be able to support them.

The workloads of typical P2P applications are often quite different from those of clusters. Distributed systems in data centers often focus on heavy computation, data analysis, supporting large number of client requests or performance in highly-parallelizable tasks. On the other hand P2P systems are more used in collaborative or sharing scenario. This has impact on the type and frequency of potential transaction conflicts. We think, that a more *Create, Read, Update* and *Delete* (CRUD) -like scenario is appropriate and benchmarks modelling such workload should be used to test a P2P TM design.

5 Design Proposals for P2P STM

Generally, we build distributed systems because we need to scale the size of data that is processed by the system, increase the overall performance of the system, or achieve reliability and fault tolerance. However, most of the distributed transactional

memory research so far focused only on providing better performance and scaling and can be thus seen as an extension of the previous TM systems for a single processing node. The proposed solutions are therefore often optimized for high-speed close-range networks in a typical data center or a group of servers in a rack. Some of the designs also use transaction managers that form a single point of failure.

We aim to explore the properties and possible use of TM in highly unreliable distributed environment with emphasis on fault tolerance. However, retrofitting existing cluster TM designs for this purpose would be, at the very least, complicated if not entirely impossible and the overhead introduced with such mechanisms could severely hamper their performance. Also since the latency, network partitions and nodes volatility of a P2P system tend to make the communication between peers slower and generally even harder than in a cluster system, we believe, that lowering the abort rate of transactions and the communication overhead is essential. For this reasons, we want to focus on computationally more expensive conflict resolution and avoidance algorithms, of which cost is diminished by the network I/O, and the relaxation of consistency semantics. In compliance with our observations, we try to draft a few design ideas, that could shape our future work and help us with the implementation of an effective solution for P2P TM.

5.1 Higher-level conflict detection and resolution

Since there is no way to prevent node failures, the only way for a distributed system to remain resilient to failures is to replicate the data across the system, so that if a part of the network gets disconnected by a network partition or a node hosting some piece of data fails, we still have a copy of the data residing somewhere on the remaining processing nodes. This in turn requires the TM to keep the data replicas consistent. It has to propagate and validate all update operations, that are executed on the data, which raises the amount of inter-node communication needed for the validation and commit phases of TM transactions. This communication often needs to be totally ordered across the whole system and can require relatively large data transfers (read- and write-sets), that could in the case of a grater network latency or timeouts severely degrade performance of the system. Existing distributed fault-tolerant TM designs, which we introduced in section 3, employ different strategies to tackle this problem. One of the common features of these strategies is the minimization of the communication needed to synchronize data using e.g. prefetching schemes to hide network latency, efficient read-set encoding or trading transaction expressiveness for reduced communication overhead.

In our research, we try to explore a different kind of synchronization optimization. Our goal is to minimize possible conflicts and therefore reduce the required information needed to validate and propagate transactions. To do this, we plan to use abstract data types (ADT) and the notion of higher-level conflict detection and resolution (HLCDR).

By rising the level of conflict detection from individual memory addresses or, depending on the TM granularity, the shared words or objects, to the higher semantics

of operations on the used data structures, we believe we can dramatically reduce the size of read- and write-sets of individual transactions in distributed environment. Such reduction decreases the change of a data access race between concurrent transactions as it has been shown by the respective studies [8, 9, 16]. Interestingly enough, these studies have analysed the effect of higher-level conflict semantics only on the performance and transaction abort-rate of single node systems and have not examined its impact on replica synchronization in distributed environment. Also, to our knowledge, HLCDR in TM has not yet been studied in the context of fault tolerance.

The use of HLCDR in combination with abstract data types (ATD) could allow us to employ a form of lazy replication. As an example, imagine two concurrent transactions that both try to insert distinct items into a replicated unordered set. If this set was implemented as an ADT, both transactions could insert the item without a conflict. If the inserted items happened to be the same, depending on the definition of equivalence for the item, the ADT could possibly merge the concurrent inserts into a single insert. This idea could be possibly even extended to employ deferred consistency control. If e.g. the transaction would not need to explicitly check for the presence of an item in the set, we could defer the replica synchronization to some later time and piggy-back it with some other communication, which could increase the performance of the TM.

5.2 Eventual Consistency

Strong consistency and isolation semantics of transactions might be easier to reason about and therefore more desirable, but their practical implementation is difficult and often requires excessive communication to validate the consistency of transactional data, which can prohibit TM scalability. Also distributed TM systems are, as every other distributed system, affected by the consequences of the CAP theorem [6]. While existing distributed TM implementations favour consistency over availability or partition tolerance, we would argue that the characteristics of P2P system require a different approach.

Following the use of relaxed consistency models such as eventual consistency (EC) in recent NoSQL data stores to cope with node failures, we think that EC could be used to improve TM scalability, performance as well as fault tolerance in P2P STM. In 2012 Burckhardt *et al.* [4] proposed and formally specified a novel consistency model for concurrent revisions based on eventually consistent transactions. Although their model was not developed for TM systems, but for concurrent programming with revisions and isolation types [3], it bears several significant similarities with snapshot isolation and ADTs.

In their consistency model, each new concurrent transaction (revision) starts by forking with a stable snapshot of shared data items represented as the isolation types from some existing revision. The forked transaction then executes all it operations on the local copies of the data from the snapshot in complete isolation without synchronizing with parallel processes. The forked transactions are then explicitly joined together (according to defined rules) and the data is merged using the isolation types.

The difference from snapshot isolation used in databases and some TM implementations lies in the merging phase, where the EC isolation types can merge even write conflicts. In contrast, the traditional snapshot isolation cannot merge write-write conflicts and one of transactions has to perform an abort and a rollback.

We believe, the consistency model developed by Burckhardt *et al.* could be adapted for use in distributed TM systems using ADT and HLCDR as a replacement for isolation types and merging. This model could then further minimize the synchronization needed in a P2P STM and allow for better fault- and partition-tolerance.

5.3 Control-Flow

Control-flow (used in several TMs) is a model in which transactional operations are not performed locally on downloaded copies of data but are sent to the remote node, on which the data item resides. This is the opposite of data-flow model, in which data items are temporarily copied from their respective remote nodes to the node that is running the transaction. While control-flow might lower the total data-transfer in the network, it increases the performance requirements on the remote node that has to perform the computation.

However, control-flow in P2P system could solve the issues of data security, asymmetric connections and also exploit the raw processing power of selected super-peers. The P2P application could for example use small client peers, that have limited resources (e.g. battery powered devices) and that would perform only lightweight computation, and server super-peers, that would do the heavy lifting. Other examples would include data with restricted access, like an account with protected secret balance located on a secure server, but can be transactionally charged or increased from remote peers through a secure validating interface, etc.

6 Conclusions

Although mainly explored in academia, transactional memory and its concepts have been slowly finding their way into some programming languages. The simplification of concurrency control provides a great incentive to use TM in new parallel and distributed systems. However, so far it has been studied mainly in the environment of highly reliable data centers and cluster computing and only a little work has been devoted to TM in the context of P2P systems and fault tolerance.

In our work, we have focused on designing a software transactional memory system for P2P networks. We have identified several key differences between cluster and P2P systems that, as we think, could have a large impact on the architectural requirements and performance properties of a TM. We believe, that the best way to combat high latency and severe node volatility is to minimize the overall replica synchronization and conflict-rate of the TM runtime. To do that, we believe that abstract data types, higher-level conflict detection and resolution and the relaxation of consistency model should be used. We also think, that the nature of P2P systems could benefit from

control flow, that could exploit the processing power and security characteristics of dedicated super-peers.

Ultimately, we think that TM could simplify the development of highly available and reliable distributed systems and would abstract the issues of node failures, replication and network partitions from the developer, while still providing good parallelism and concurrency. In this sense, we see TM as an enabling technology that could give birth to a whole new breed of distributed applications.

Acknowledgements

This article is an extended version of a paper from the ADBIS PhD Consortium 2012 that was published by Springer [17]. This work was partially supported by the Slovak Research and Development Agency under the contract No. APVV-0233-10 and by the Scientific Grant Agency of Slovak Republic, grant No. VG1/0971/11.

References

[1] Aguilera, M. K.; Merchant, A.; Shah, M.; et al.: Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, New York, NY, USA: ACM, 2007, pp. 159–174.

[2] Bocchino, R. L.; Adve, V. S.; Chamberlain, B. L.: Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, New York, NY, USA: ACM, 2008, pp. 247–258.

[3] Burckhardt, S.; Baldassin, A.; Leijen, D.: Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, New York, NY, USA: ACM, 2010, pp. 691–707.

[4] Burckhardt, S.; Fahndrich, M.; Leijen, D.; et al.: Eventually Consistent Transactions. In *Proceedings of the 22n European Symposium on Programming (ESOP)*, Springer, March 2012.

[5] Couceiro, M.; Romano, P.; Carvalho, N.; et al.: D2STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the 2009 15th IEEE pacific Rim International Symposium on Dependable Computing*, PRDC '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 307–313.

[6] Gilbert, S.; Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, volume 33, num. 2, June 2002: pp. 51–59.

[7] Herlihy, M.; Moss, J. E. B.: Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, volume 21, num. 2, May 1993: pp. 289–300.

[8] Herlihy, M.; Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, New York, NY, USA: ACM, 2008, pp. 207–216.

[9] Koskinen, E.; Parkinson, M.; Herlihy, M.: Coarse-grained transactions. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, New York, NY, USA: ACM, 2010, pp. 19–30.

[10] Kotselidis, C.; Ansari, M.; Jarvis, K.; et al.: DiSTM: A Software Transactional Memory Framework for Clusters. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 51–58.

[11] Kulkarni, M.; Pingali, K.; Walter, B.; et al.: Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, New York, NY, USA: ACM, 2007, pp. 211–222.

[12] Lee, J.; Dash, A.; Tucker, S.; et al.: Fault-Tolerant Distributed Transactional Memory. Under review for Transactions on Programming Languages and Systems.

[13] Manassiev, K.; Mihailescu, M.; Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, New York, NY, USA: ACM, 2006, pp. 198–208.

[14] Mesaros, V.; Collet, R.; Glynn, K.; et al.: A Transactional System for Structured Overlay Networks. March 2005.

[15] Müller, M.-F.; Möller, K.-T.; Schöttner, M.: Commit Protocols for a Distributed Transactional Memory. In *Proc. of the 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[16] Ni, Y.; Menon, V. S.; Adl-Tabatabai, A.-R.; et al.: Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP 07, New York, NY, USA: ACM, 2007, pp. 68–78.

[17] Paulovič, A; Návrat, P.: Designing a Software Transactional Memory for Peer-to-Peer Systems. In *New Trends in Databases and Information Systems*, Advances in Intelligent Systems and Computing, volume 185, Springer, 2013, pp.395–401.

[18] Pratt-Szeliga, P.; Fawcet, J.: p2pstm: A Peer-to-Peer Software Transactional Memory. Technical report, Syracuse University, Syracuse, NY, 2010.

[19] Shavit, N.; Touitou, D.: Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, New York, NY, USA: ACM, 1995, pp. 204–213.

Presented at the 16th East-European Conference on Advances in Databases and Information Systems September, 17-20, 2012, Poznań, Poland