TROM: A Transactional Replicated Object Memory

Jörg Domaschka
E-mail: joerg.domaschka@uni-ulm.de

Abstract

This paper presents transactional replication, a novel approach for replicating distributed objects. It uses non-blocking synchronisation techniques to realise software transactional memory for ensuring replica consistency. Unlike traditional passive object replication, which executes requests on a single primary, our approach allows using an arbitrary replica to execute a client request. Fine-grained synchronisation mechanisms ensure consistency between the replicas. This way, the computational load can be equally distributed on all replicas. Unlike active object replication, our approach does not offer redundant execution of requests. Nevertheless, it enables a fast reaction to replica failures in most situations.

1 INTRODUCTION

The two main replication strategies for distributed objects are active and passive replication. In this position paper we present transactional replication, a novel approach to object replication that combines the support for non-deterministic object implementations known from passive replication with the non-centralistic interaction and fast reaction to failures which both are features of active replication. We also claim that our approach, if applied in the right environment, increases concurrency and multithreading abilities. However, it comes at the cost of increased network traffic. Our approach uses techniques and concepts from software transactional memory and database replication. The first is applied to provide a transactional API and journaling mechanisms, the latter to come to an agreement about the commitment of a transaction. In addition to that we sketch how to map software transactional memory to replicated environments leading to *transactional replicated object memory (TROM).*

The main contribution of this position paper is to propose a way how to use transactional APIs as known from software transactional memory to replicate distributed objects. This enables a very fine-grained synchronisation between processes running on different hosts and is supposed to increase concurrency and throughput at the cost of additional network communication. This paper is structured as follows. The next section contains a state of the art survey of object replication, database replication, deterministic scheduling and lock-free serialisation. Section 3 discusses the implementation of TROM. Finally, Section 4 concludes.

2 STATE OF THE ART

2.1 Styles of Object Replication

For replicating distributed systems two main strategies are widely known: active and passive replication. In the following we use the definition of *Défago and Schiper* [1], who define passive replication as a replication technique with parsimonious processing, i.e., a request is processed by exactly one leader replica that in turn spreads the update information to all other follower replicas. Active replication is defined as a replication technique with redundant processing, i.e., all replicas execute a request.

Active replication offers fast reaction to node failures and is able to guarantee a constant response time, but requires a deterministic object implementation, as all replicas have to achieve the same result for each request. If the replicated object interacts with the system environment, the environment also has to provide deterministic responses. The determinism of the replicated object ensures that a single request causes identical changes to the state of all replicas. If all requests are distributed to all replicas using totally ordered broadcast, the execution of all requests in an identical order together with the determinism ensures consistency of the replica state over the sequence of all requests. Passive replication - in general - less depends on a deterministic object implementation.

In the past multiple systems supporting the replication of distributed objects have been published. Many of them were developed in the context of CORBA (see [2] for a survey) such as Electra [3], DOORS [4], and OGS [5]. Some systems have also been developed for Java RMI namely Filterfresh [6], Jgroup [7], Aroma [8], and one published by *Cazzola et al.* [9]. Other object-oriented systems with replication support are Globe [10] and FT*flex* [11], which both follow a non-monolithic object model.

The systems differ in the way replication is integrated in existing environments, the kinds of replication techniques they support, the restrictions they pose on the object implementation, and the obligations they impose on the programmer.

• *J. Domaschka is with the Institute of Distributed Systems, University of Ulm, Germany.*

originally published January 1, 2009

2.2 Deterministic Multithreading

In order to ensure determinism, most existing replication frameworks process incoming requests in a strictly sequential order without any concurrency. Such a scheduling strategy is inefficient in terms of throughput, as it does not allow full CPU utilisation (for example, if the current thread is idle due to I/O operations, no other request can utilize the CPU), and—even worse—it does not obtain any benefit on multi-core or multi-CPU machines. To circumvent the restrictions of sequential execution and to keep up determinism at the same time, recently, *Basile et al.* [12], *Napper et al.* [13], and *Reiser et al.* [14] have presented algorithms realising deterministic multithreading. To the best of our knowledge those are the only algorithms for that purpose published so far. All of them use locking operations as synchronisation points, i.e. they depend on the use of mutexes. The middleware run-time system intercepts the requests for locking and releasing mutexes and informs the middleware scheduler which takes the appropriate steps ensuring that threads processing conflicting requests are never scheduled in parallel by the operating system scheduler. Such a proceeding allows concurrency for non-conflicting threads and those that are outside a critical region. However, as it is hard to predict which mutexes a thread will use during the execution of a request [15], the decisions the schedulers take do have to be conservative so that they bear the risk of being too restrictive for certain application scenarios.

2.3 Software Transactional Memory

Transactional memory (TM) is a memory abstraction that allows grouping a sequence of `read` and `write` operations to the memory in one atomic operation called *transaction*. *Software transactional memory* (STM) is TM that does not rely on specific hardware support, but realises all transactional functionality in software (see [16] for a survey). Typically those systems are implemented in a non-blocking way. Mutexes are not used, because they bear the risk of deadlocks. STM systems can be divided into three categories according to the guarantees they make. *Wait-free* [17] is the strongest guarantee, as all processes will make progress in a finite number of steps. *Lock-free* systems guarantee [18] that the system as a whole will make progress, whereas the weakest category, *obstruction-free* systems [19], does so only in the absence of contention. Lock-free systems allow starvation, while obstruction-free systems allow starvation as well as livelocks. However, those are worst-case scenarios which occur with low probability in practice.

Shavit and Touitou [20] have presented the first STM. It provides linerizablity and requires pre-knowledge of all locations involved in the transaction. *Herlihy et al.* [19] proposed DSTM an obstruction-free, lineralizeable, and dynamic STM implementation for objects that uses a pluggable *contention-manager* to ensure progress and to rule out livelocks. DSTM uses early conflict resolution, i.e., the system can abort transactions before the final commit. *Fraser and Harris* [21] present OSTM, an object-based and lock-free STM similar to DSTM, but without early conflict resolution. Instead it lets all transactions run privately until they commit. Such an approach bears the risk of wasting resource due to unnecessary computations, but also enables threads helping one another committing their transactions. Finally, *Napper and Alvisi* [18] present an STM that is similar to OSTM, but realises serializability instead of linearizability.

To enable a programmer to use such STM *Herlihy et al.* [22] provide the DSTM2 library. In contrast *Harris et al.* [23] present extensions to the syntax of the Java programming language. Similarly, *Welc et al.* [24] propose transactional monitors for Java. In contrast to [23] they also support condition variables inside the monitors.

2.4 Replication in Databases

Besides their use in STM, transactions are a key concept in databases. Here, transactions are a set of logically linked `read` and `write` operations. Once started, transactions are eventually committed or aborted either by the user or the runtime system. Either the system or the user can trigger an abort; only the user can request a commit, and the system has to verify the commit before accepting it. A single transaction can consist of multiple client requests to the database and is terminated by a request for either a *commit* or an *abort*.

Wiesmann et al. [25] define a formal framework for replication techniques. Within the framework, they discuss multiple interaction patterns, which all reflect existing replication protocols. Besides other approaches for replication in distributed systems, they cover database replication strategies. In all of these strategies, a client interacts with exactly one database. First, there exists, similar to passive replication, an *eager primary copy replication* strategy: a request is executed only at the primary database. After each client interaction, the necessary changes are applied and the other databases are informed. *Eager update everywhere replication* can be realised based on distributed locking or based on certification. The client can chose an arbitrary replica. When distributed locking is used each client interaction is followed by a locking phase in which the replica tries to acquire the necessary locks. After the last client operation, a commit phase follows. When certification is used, locks are not required. Instead, all operations are executed at a single replica using shadow copies. For the commit all changes are broadcast with total order to all other replicas. Now, the replicas have to agree on the whether the transaction shall be committed or aborted. This is called the certification step. *Pedone et al.* [26] presented the *database state machine* that implements such a replication strategy. As this is the concept that is most similar to what we propose in this paper, we always refer to an eager update everywhere replication strategy based on certification when we talk about database replication.

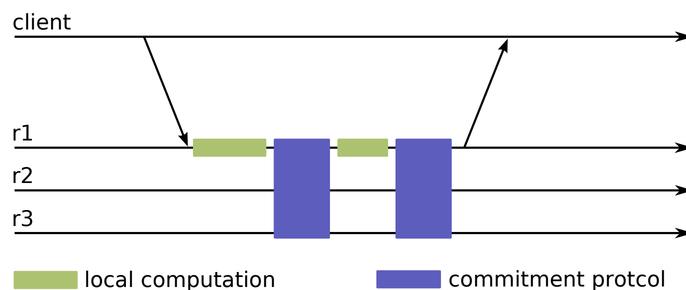

Fig. 1. Processing of a Remote Method Invocation

3 TRANSACTION-BASED OBJECT REPLICATION

In this section we present the main contribution of this paper. We propose to use STM mechanisms in order to realise object replication with transactions. This section is structured as follows. In the next subsection, we define our system model, followed by the basic approach on an abstract level. Afterwards, we compare the new approach to existing approaches in object and database replication. In Section 3.5 we analyse how such an approach can be implemented. Finally, we present open questions and sketch future work.

3.1 System Model

For the discussion throughout the rest of this paper, we assume the following system model. At the server-side there is a set of replicas with identical interfaces. The replicas can communicate by a group communication system with total order. Replicas may crash arbitrarily, and once they have crashed, they never recover, i.e., we assume a crash-stop failure model. A client can invoke methods at the remote interface of the object group using a local proxy, which contains the client-side replication logic. For a remote method invocation the stub contacts a single replica no matter which replication strategy is used. Each method consists of computational and synchronisation blocks. During the execution of a computational block the method exclusively operates on private data. In contrast while processing a synchronisation block it operates on data other methods might also access, i.e., the shared object state. We assume that the remote invocation infrastructure provides at-most-once semantics. This means that resending a request after a communication problem between client and servers does never causes a duplicated execution of this request. We further assume that the proxy at client side helps to ensure the server-side semantics by tagging a repeated request with the message ID of the original one. The client re-issues a request whenever it gets aware that the contacted replica has crashed.

3.2 Basic Approach

On an abstract level, we try to replace blocking solutions for concurrency control in object replication based on mutexes and deterministic scheduling by a non-blocking approach using a transactional API. In contrast to existing approaches, we protect critical sections by transactions instead of mutexes. Consequently, the execution of a remote method invocation may trigger multiple transactions. Figure 1 shows a high-level sequence diagram that illustrates the steps to be taken while processing one request. A client contacts exactly one replica. In contrast to passive replication, this replica is not fixed, but can be chosen arbitrarily resulting in a balanced load among the replicas. The replica processes the request locally. When the processing of a request requires the execution of a transaction all transaction-related operations are executed locally and journaled just like in STM without early conflict detection. At the commit, however, the transaction journal is broadcast to all other replicas. Finally, the replica group comes to a decision about whether to commit or abort the transaction.

Note that none of *Wiesmann et al.'s* [25] interaction patterns fits to our approach, though it is possible to map transactional replication to their general framework.

3.3 Comparison to Existing Object Replication Strategies

In comparison to passive replication our approach offers a much higher computational power, as all available replicas may run transactions concurrently. Furthermore, the reaction to failures is faster as any replica may execute the computations. However, in the face of objects with a small state that is also nearly completely accessed by every method invocation in combination with a high request frequency the advantage of high parallelism may turn out to be counter-productive, as probably many transactions will have to be aborted due to congestion. Compared to both other replication strategies transaction-based replication can become very expensive in terms of message numbers, as there is not only one totally ordered message per client request as in active replication, but at least one per transaction. If methods use a high number of transactions this may lead to a decreased execution speed because of frequent network access. Moreover, using mutex-based synchronisation with

deterministic multithreading algorithms will eventually finish processing a request assumed a deadlock-free implementation and an appropriate interaction pattern. Our approach is based on existing STM systems. Most of these systems STM systems are either obstruction- or lock-free, starvation becomes possible even in the face of a correct implementation and regular interaction patterns.

There exist also many scenarios in which transactional replication might turn out to be applicable and useful. First of all, processing a request on a single host weakens the necessity for a totally deterministic object implementation that is a hard requirement in active replication. Furthermore, for huge remote objects whose state consists of largely nested object structures that are manipulated independently from each other, and that require high availability, transactional replication might be useful. An example for such a data structure are object graphs in multi-player games.

3.4 Comparison to Database Systems

Although the use of transactions for object replication might resemble database replication, it is not possible to neglect some differences in the assumptions both kinds of systems make. The first and most fundamental difference is the fact that databases do not come with their own application logic, but are just a data-storage. Consequently, they usually do not trigger the execution of transactions at other databases nor do they invoke other services. These assumptions do not hold anymore for replicated objects, no matter if they use transactions for synchronisation or not. In the context of transactional replication, however, this brings up the questions how to handle transactions that span objects that are services on their own. Nested transactions may be one way to answer the question.

Our system model from Section 3.1 assumes that clients invoke single methods, which are containers for the transactions. In databases the client-side programme logic results in the fact that a transaction might consist of a sequence of multiple client requests finished by either a *commit* or *abort* request. This leads to one multicast per multiple client requests in the database state machine approach. The transactional replication scheme, however, leads to zero, one, or multiple transactions and with it multiple multicasts per single client-interaction.

Another difference concerns the execution state. In the object-based model, the client is not aware of the execution state of a method invocation. It is only aware whether a method has been completed successfully or requires re-execution. For databases, the logic is at client-side, so that the client is always in control of the transaction it is currently executing. Consequently, in the transactional model, when re-issuing the request in case of a replica crash, the client has to rely on server-side mechanisms to recognise already executed transactions and handle them appropriately.

Finally, a database implementation operates on external data; data that is not part of the implementation. Due to that, operations on the data are easily manageable by the database run-time. In an object implementation, however, the access to object state is executed directly by object code and thus not controllable by the run-time. This fact complicates handling transactions in object implementations and introduces overhead.

3.5 Realisation Sketch

In order to enable transactions in a object-oriented and replicated environment two prerequisites are required. Firstly, the run-time needs to be transaction-aware. For that purpose both approaches presented in Section 2.3 are appropriate: libraries and languages extensions. In a first run we aim using and extending a library, as this does not require changes to compiler and run-time. In addition instrumentation for evaluation purposes is added more easily. Furthermore, we plan to leave the decision on how to handle aborted transactions to the developer, i.e., we do not automatically repeat failed transactions.

Secondly, it has to be possible to describe transactions on an abstract level due to the following reasons. STM, which we plan to use for executing and journaling transactions, generate their records based on memory locations. This is fine as long as the information does not have to cross address spaces. Applying this scheme for a replicated scenario, however, requires that other replicas can interpret the information contained in the transaction journal, which is not the case if information relies on memory locations. As an approach to satisfy this requirement, we propose to describe the state of an object at the interface level by an interface definition language as for instance CORBA IDL. Furthermore, this enables the automatic generation of code which maps between addresses, object names and objects so that the programmer does not need to handle it manually. In addition such an approach ensures that sufficient information about the object state is available to serialise it automatically. Yet, as a consequence no objects may be used in transactions that are not part of the transitive closure of the object state. However, this is acceptable, as all other objects are considered request- or method-local so that they are not subject to race conditions and thus do not have to be protected against concurrent access. Furthermore, compared to current solutions using deterministic multithreading this does not even constitute a limitation, as those imply that the mutexes in use have to be part of the object state.

For achieving fault-tolerance special care has to be taken. Unlike active or passive replication that are designed with focus on fault-tolerance, the goal of transactions is mainly an increased concurrency. Thus, transactional replication does not offer fault-tolerance *per se*. In active replication the crash of one replica does not lead to any problems as all of them behave identically. Similarly, in passive replication the crash of a replica cannot cause any harm. It is either a follower replica that is not needed for achieving progress or the primary replica that, if crashed during processing a request, will not have left other replicas in

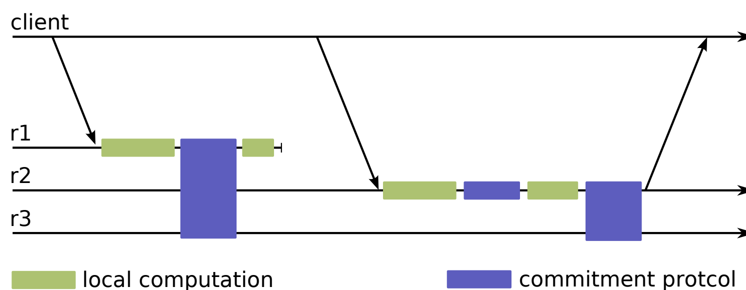

Fig. 2. Processing of a Remote Method Invocation in Face of Replica Crashes

an inconsistent state, as update messages are only sent after a successful execution. In transactional replication, however, the crash of a replica during the execution of a request may leave other replicas in an inconsistent state. To put it another way, it might lead to a partial re-execution of some transactions what has to be avoided in order to achieve an at-most-once semantics. In particular this requires some bookkeeping about transactions that have already been executed. Figure 2 shows such a scenario. The client sends a request to replica r_1. After having partially processed the request including the commitment of one transaction, r_1 crashes. The client notices the crash and re-issues the request to replica r_2. r_2 starts processing the request and recognises the previously committed transaction. All read and write operations during that transactions are performed according to the transaction logs. Finally, the commitment step is skipped. The processing continues with succeeding transactions being committed regularly. To realise such behaviour we plan to make use of the message IDs the client uses. Furthermore, we enumerate the transactions per remote message and encode both IDs in the transaction journal. With the help of a cache, all replicas keep track of the transactions they received during the commitment phase. In case they receive the same transaction more than once, i.e., two transactions with identical IDs, the others are ignored. Cache entries can either be deleted after a certain period of time or at the next request of the same client. How the transaction log can be built in an efficient way is subject to further investigations.

3.6 Open Questions - Future Work

Goal of this position paper is to stress the possibility of transactional object replication based on STM mechanisms. It is for sure not a one-size-fits-all solution to all problems of object replication. However, we think that it offers an interesting alternative to existing replication approaches and is worth further investigation. As always in distributed computing the benefit of this approach will heavily depend on whether its usage is appropriate to environmental conditions including object implementation and load pattern. That is why in a next step we are targeting a proof of concept implementation that can evaluate scenarios in which transactional-replication outperforms the other replication strategies. Besides that, there are multiple open questions that found the basis of our future research.

An elementary question for us is whether it is possible to realise our replication scheme in a completely non-blocking manner. Currently, we are just facing a framework that uses STM mechanisms to enable concurrent execution. For updating the replicas during the commit phase we are not insisting on non-blocking synchronisation.

In the long run we are targeting a highly adaptive system that allows switching between all three replication strategies, i.e. active replication, passive replication, and transactional replication, even at run-time. An absolutely necessary condition to realise this is to allow the programmer to specify critical regions without taking care of the later strategy to be used to handle concurrency. The benefit would be enormous as the system could not only be adapted manually, but would also be ready for self-adaptation. One approach to enable strategy switching could be to use mappings between existing solutions, e.g. between the Java `synchronized` statement and an STM library. How such mappings could look like is subject to further investigations. Another interesting question is whether it is possible to use both lock-based synchronisation with a deterministic scheduler and non-blocking synchronisation in parallel. Reasons for this requirement is to enable the programmer to use the solution that fits best for his problem. In a self adapting system this might also enable the system to optimise on a per critical region basis, instead of object-basis.

Replication systems require a state transfer mechanism when new replicas join the replica group. State transfer in the presence of multiple threads has turned out to be a major challenge in systems using deterministic multithreading based on mutexes and condition variables. In the general case it is not possible to give any guarantees about the point of time when the object implementation will be thread-free what is mandatory during state transfer. The fact that also for transactional replication eventual termination of all requests cannot be guaranteed entails the same problem. However, as in transactional replication a state change can always be seen as the commit of a transaction, storing the latest set of transactions that covers the entire state could be a first approach to handle state transfer on a generic level. Furthermore, such an approach in combination with mappings between different synchronisation schemes might even help to solve the state transfer problem coming along with deterministic multithreading.

CORBA IDL allows to express only limited dynamic data structures namely arrays and sequences. It is not clear up to what extend this supports arbitrary dynamic data structures. Thus, further investigations are required. Of course the introduction of dynamic states also affects the state transfer problem, as it becomes harder to determine the latest set of transactions that covers the entire current state.

Finally, we have to evaluate how the programming model of TROM changes compared to STM. For instance we expect that transactions in an replicated environment have to cover larger parts of the implementation in order to achieve fault-tolerance.

4 Conclusions

In this position paper we have presented transactional replication with TROM, an alternative strategy for replicating distributed objects. We use mechanisms known from STM systems to group critical regions in transactions and apply them in a replicated environment. A remote method invocation consists of zero, one or more transactions. Methods are executed at a single, but arbitrary replica. Transactions of different processes run in parallel until the commit. By then information about the transaction is broadcast to all other replicas which find a consensus whether to commit or abort the transaction. Finding out about which transaction to commit or abort can be achieved by different algorithms that are employed for instance in database systems.

References

[1] X. Défago and A. Schiper, "Specification of Replication Techniques, Semi-Passive Replication, and Lazy Consensus*," EPFL, Tech. Rep., 2002.

[2] P. Felber and P. Narasimhan, "Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems," *IEEE Trans. Comput.*, vol. 53, no. 5, pp. 497–511, 2004.

[3] S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," in *COOTS '95*, June 1995, pp. 135–146. [Online]. Available: citeseer.ist.psu.edu/maffeis95adding.html

[4] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt, "DOORS: Towards High-Performance Fault Tolerant CORBA," in *DOA '00*, 2000, pp. 39–48.

[5] P. Felber, X. Défago, P. T. Eugster, and A. Schiper, "Replicating CORBA Objects: A Marriage Between Active and Passive Replication," in *DAIS '99*, 1999, pp. 375–388.

[6] A. Baratloo, P. E. Chung, Y. Huang, S. Rangarajan, and S. Yajnik, "Filterfresh: Hot Replication of Java RMI Server Objects," in *COOTS '98*, 1998, pp. 65–78. [Online]. Available: citeseer.ist.psu.edu/baratloo98filterfresh.html

[7] A. Montresor, "The Jgroup Distributed Object Model," in *DAIS '99*, 1999, pp. 389–402.

[8] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Transparent Consistent Replication of Java RMI Objects," in *DOA '00*, 2000, pp. 17–26.

[9] W. Cazzola, M. Ancona, F. Canepa, M. Mancini, and V. Siccardi, "Enhancing Java to Support Object Groups," in *ROOTS '02*, 2002.

[10] P. Homburg, M. van Steen, and A. S. Tanenbaum, "Unifying Internet Services Using Distributed Shared Objects," Vrije Universiteit Amsterdam, Tech. Rep. IR-409, 1996. [Online]. Available: citeseer.ist.psu.edu/homburg96unifying.html

[11] H. P. Reiser, R. Kapitza, J. Domaschka, and F. J. Hauck, "Fault-Tolerant Replication Based on Fragmented Objects," in *DAIS '06*, 2006, pp. 256–271.

[12] C. Basile, Z. Kalbarczyk, and R. K. Iyer, "Active replication of multithreaded applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 5, pp. 448–465, 2006.

[13] J. Napper, L. Alvisi, and H. Vin, "A Fault-Tolerant Java Virtual Machine," in *DSN '03*, 2003. [Online]. Available: http://www.cs.utexas.edu/users/jmn/papers/napper03fault.pdf

[14] H. P. Reiser, F. J. Hauck, J. Domaschka, R. Kapitza, and W. Schröder-Preikschat, "Consistent Replication of Multithreaded Distributed Objects," in *SRDS '06*, 2006, pp. 257–266.

[15] J. Domaschka, A. I. Schmied, H. P. Reiser, and F. J. Hauck, "Revisiting Deterministic Multithreading Strategies," in *IWJPDC '07*, 2007.

[16] V. J. Marathe and M. L. Scott, "A Qualitative Survey of Modern Software Transactional Memory Systems," University of Rochester, Tech. Rep. TR 839, 2004.

[17] M. Herlihy, "Wait-Free Synchronization," *ACM TOPLAS*, vol. 11, no. 1, pp. 124–149, January 1991.

[18] J. Napper and L. Alvisi, "Lock-Free Serializable Transactions," The University of Texas at Austin, Department of Computer Sciences, Tech. Rep. TR-05-04, 2005. [Online]. Available: http://www.cs.utexas.edu/ftp/pub/techreports/tr05-04.pdf

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, "Software Transactional Memory for Dynamic-Sized Data Structures," in *PODC '03*, 2003, pp. 92–101.

[20] N. Shavit and D. Touitou, "Software Transactional Memory," in *PODC '95*, 1995.

[21] K. Fraser and T. Harris, "Concurrent Programming Without Locks," *ACM TOCS*, vol. 25, no. 2, p. 5, 2007.

[22] M. Herlihy, V. Luchangco, and M. Moir, "A Flexible Framework for Implementing Software Transactional Memory," *SIGPLAN Not.*, vol. 41, no. 10, pp. 253–262, 2006.

[23] T. Harris and K. Fraser, "Language Support for Lightweight Transactions," in *OOPSLA '03*, 2003, pp. 388–402.

[24] A. .Welc, S. Jagannathan, and A. L. Hosking, "Transactional Monitors for Concurrent Objects," in *ECOOP '04*, 2004, pp. 519–542.

[25] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding Replication in Databases and Distributed Systems," in *ICDCS 2000*, 2000, pp. 464–474.

[26] F. Pedone, R. Guerraoui, and A. Schiper, "The Database State Machine Approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.