Self tuning replication in transactional data grids

Hugo Gomes Pimentel

IST – Instituto Superior Técnico
– INESC ID Lisboa
Distributed Systems Group
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
hugo.pimentel@ist.utl.pt

Abstract Partially replicated systems help to improve the scalability of replicated systems by allowing sites to store a subset of the application data and split the load among replicas, to maximize throughput. On the down side, data partitioning leads to the need of dealing with expensive remote data accesses, every time there is a retrieval of data not locally maintained. To cope with this problem several systems introduce an additional caching mechanism aimed at maintaining, at each node, a replica of the remote data items most frequently accessed by local transactions. However, when exposed to different workloads, caching mechanisms can also perform differently.
This report proposes the development of an efficient, self-tuning caching mechanism for partially replicated transactional systems that aims at providing optimal performance when exposed to different workloads.

1 Introduction

The advent of cloud-computing has recently spurred a new generation of distributed data platforms [1, 2, 3, 4, 5, 6], often referred to as NoSQL data grids, designed from the ground up to overcome the limitations of relational DBMSs in terms of scalability and elasticity [7]. This platforms rely on a simpler data model, lightweight application interfaces and efficient mechanisms to achieve data durability.

An important aspect that allows to classify existing systems and that has a significant impact on several of their key design choices is whether data is fully or partially replicated. Fully replicated systems place copies of data items at all replicas to achieve high availability. However, because all the nodes need to be contacted upon commit, fully replicated systems are inherently unscalable, especially in presence of write-dominated workloads. On the other hand, partially replicated systems are highly scalable because they allow sites to store a subset of the application data and split the load among replicas, to maximize throughput. On the down side, data partitioning leads to the need of dealing with expensive remote data accesses, every time there is a retrieval of data not locally maintained. To cope with this problem several systems introduce an additional caching mechanism aimed at maintaining, at each node, a replica of the

remote data items most frequently accessed by local transactions. However, when exposed to different workloads, caching mechanisms can also perform differently.

In this project, I am interested in the development of an efficient, self-tuning caching mechanism for partially replicated transactional systems, which aims to provide optimal performance when faced with different workloads.

The rest of this report is organized as follows. Section 2 motivates and describes the goals of this work. Section 3 provides an introduction to the different technical areas related to it. Section 4 explains in detail the systems involved in this work and what solution will be proposed and Section 5 describes how this solution will be evaluated. The scheduling of future work is presented in Section 6. Finally, Section 7 concludes this report by summarizing its main points.

2 Goals

This work tackles the problem of caching in partially replicated systems. More precisely:

- *Goals*: The main goal of this project is to analyze, design and implement an efficient self-tuning caching mechanism for a NoSQL in-memory transactional data grid.

The work started with a survey of related work (to be presented in Section 3). From this survey it becomes clear that the performance of these systems can be significantly hampered by the need to perform expensive remote accesses to retrieve data not locally maintained. To address this problem, several NoSQL platforms rely on an additional caching mechanism aimed at maintaining, at each node, a replica of the remote data items most frequently accessed by local transactions. However, caching mechanisms can perform differently when exposed to different workloads. Also, by caching very frequently updated objects that can become rapidly stale, the system risks to expose likely stale data to transactions. This may cause a very high abort rate, with detrimental effects on system's throughput.

With this in mind, I propose the implementation of an efficient, self-tuning caching mechanism aimed at autonomously determining whether it is beneficial or not to cache individual object instances. This work will investigate different lightweight heuristics to determine an object's cacheability, and will integrate the proposed caching solution in a well known open source data platform, Infinispan [5] by Red Hat.

Specifically, I will target GMU [8], a highly scalable partial replication protocol that has been recently integrated in Infinispan. The implementation will be evaluated with well established benchmarks for transactional systems.

- *Expected results*: This work will: i) implement the proposed caching mechanism in GMU; ii) provide an experimental comparative evaluation of the use or not of this mechanism and; iii) provide a quantified assessment of its advantages and disadvantages.

3 Related Work

In order to understand the underlying problems that this project poses, we need to review the fundamentals of various areas in distributed systems.

I start by introducing the abstraction of transaction, and overviewing a set of relevant correctness criteria that were introduced to specify acceptable behaviors of (distributed) transactional systems.

Next I overview three types of transactional systems, namely database management systems, software transactional memories, and transactional data grids.

I then move to discuss replication techniques for transactional systems. Before doing so, however, I review some fundamental building blocks that are at the heart of a large number of replication techniques, namely Group Communication Systems, and the group communication primitives they provide.

Finally, I present some of the most recent techniques in the area of self-tuning of transactional data grids.

3.1 Transactions

The transaction abstraction was originally introduced in the context of DBMSs, as a way to batch multiple data manipulation operations into a single unit of work to be treated in a coherent and reliable way by the underlying transactional system.

During its life-cycle, a transaction can pass through four distinct, well-defined states:

- *Executing*: the transaction's operations are executing;
- *Committing*: the transaction has completed the execution of its operations, and therefore, the client requested the transaction's commit;
- *Committed*: the transaction was committed;
- *Aborted*: the transaction was aborted.

Both the executing and committing states are transitory, while both the aborted and committed states are final.

Once started, a transaction must be ended either with commit or rollback. A successfully completed transaction ends with commit to permanently record the transaction results in the database. A failed and then aborted transaction ends with rollback to undo the transaction effects on the database.

Regarding properties, database transactions satisfy atomicity, isolation, consistency and durability. These are commonly referred to as ACID properties. Their description is the following:

- *Atomicity*: ensures that modifications must follow an "all or nothing rule", i.e., either all the modifications made by a committed transaction are made visible or none is;
- *Consistency*: ensures that each transaction changes the database from one consistent state to another consistent state;

- *Isolation*: ensures that individual memory updates within a memory transaction are hidden from concurrent transactions;
- *Durability*: ensures that once a transaction is committed, its updates will survive any subsequent malfunctions.

3.2 Transactional Systems

In the next sections I present three of the most important transactional systems today.

Data Base Management Systems

Data Base Management Systems (DBMS) are a set of programs that enables users to store, modify, and extract information from a database and also provides tools to add, delete, access, modify and analyze data stored in one location.

They can be categorized according to the model they support or the query language or languages that are used to access the database.

The relational model [9] has been the reference model for data storage for decades and consists of three major components:

- the set of relations and set of domains that defines the way data can be represented (data structure);
- integrity rules that define the procedures to protect the data (data integrity);
- the operation that can be operate on the data (data manipulation).

A relational database supports relational algebra, consequently supporting the relational operations of the set theory. In addition to mathematical set operations namely, union, intersection, difference and Cartesian product, relational databases also support select, project, relational join and division operations. These operations are unique to relational databases.

SQL was one of the first commercial languages for this model and became the most widely used.

This model has several advantages relatively to other models, namely the ease of use and design, flexibility and data independence. However, the expressiveness of the relation data model, as well as the richness of the data manipulations supported by SQL, make it extremely difficult to develop systems capable of scaling horizontally to a large number of nodes [10].

Software Transactional Memories

One of the challenges of parallel programming is synchronizing concurrent access to shared memory by multiple threads. Programmers have traditionally used locks for synchronization, but lock-based synchronization has well-known pitfalls. Simplistic coarse-grained locking does not scale well, while more sophisticated fine-grained locking risks introducing deadlocks and data races.

Software Transactional Memory [11] (STM) provides a new concurrency-control construct that avoids the pitfalls of locks and significantly eases concurrent programming. Transactional-language constructs are easy to use and can lead to programs that scale. By avoiding deadlocks and automatically allowing fine-grained concurrency, transactional-language constructs enable the programmer to compose scalable applications safely out of thread-safe libraries. The notions of atomicity, consistency and isolation from database transactions are also provided. STMs can be categorized as either word-based [12, 13] or object-oriented [14, 15].

STMs and DBMSs have some common characteristics, namely:

- both support the abstraction of transaction;
- both control, in a transparent fashion, the concurrent access to data.

However, there are also some key differences between them, namely:

- STMs do not need to ensure data persistence, thus having execution times typically several orders of magnitude smaller than DBMSs;
- DBMSs expose a well defined/restricted SQL API and execute in a sandboxed environment, while STM applications can perform generic memory manipulations directly in the address space of user applications.

This key differences motivated the development of new concurrency control mechanisms specifically tailored for STM environments focused on multicore architectures, such as [12, 16, 17]. Also, they led to the introduction of an important safety property called opacity [18]. While the strongest coherence model of DBMSs is serializability [19], in STMs both serializability and opacity need to be ensured. Opacity can be viewed as an extension of serializability, with the additional requisite that even non-committed transactions are prevented from accessing inconsistent states (DBMSs only guarantee that committed transactions do not see inconsistent states), thus avoiding the occurrence of anomalies due to concurrent data accesses that can lead to arbitrary application behaviors (such as infinite loops or unhandled exceptions) and the probable crash of the entire application.

Transactional Data Grids

As mentioned before, relational databases have represented the indisputable reference solution for transactional data management. However, another broad class of DBMS that is in vogue nowadays is called NoSQL (Not only SQL). NoSQL is identified by the non-adherence to the use of the relational model and its query language SQL. NoSQL data stores are designed to manage large volumes of data that do not necessarily follow a fixed schema. The reduced run time flexibility compared to full SQL systems is compensated by large gains in scalability and performance for this data models.

With the advent of cloud computing, which is the use of computing resources (hardware and software) that are delivered as a service over a network (typically

the Internet), there has been a proliferation of a new generation of NoSQL platforms, often called NoSQL data grids, that rely on a simpler data model, lightweight application interfaces and efficient mechanisms to achieve data durability, because relational databases do not meet the scalability requirements for this type of systems.

The term data grids is used to encompass a wide range of distributed NoSQL systems such as key-value or column-oriented stores. This solutions use a wide range of consistency criteria, and not all of them provide support for transactions. Also those that provide transactions often do not provide the classic serializability criterion. The focus of this section is on Transactional Data Grids, but I briefly overview also non-transactional data grid systems.

Bigtable [1], from Google, is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes. It is built on several other pieces of Google infrastructure such as the distributed Google File System [20] (GFS) to store log and data files and the SST table format that is used to store Bigtable data. It also relies on a highly-available and persistent distributed lock service called Chubby [21] that uses the Paxos algorithm [22] to keep its replicas consistent in the face of failure. Bigtable does not support transactions.

Percolator [2] is an implementation of Bigtable for performing incremental processing at large scale,i.e, it allows the maintenance of a very large repository of documents and updates it efficiently when a new document is crawled. Differently from Bigtable, Percolator supports cross-row, cross-table ACID-compliant transactions that provide the snapshot isolation [23] consistency level.

Spanner [3], also from Google, is a transactional globally-distributed database that has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Unlike similar systems, Spanner relies on timestamps that provide intervals with bounded time uncertainty to implement externally-consistent distributed transactions, lock-free read-only transactions, and atomic schema updates. It makes heavy use of hardware-assisted time synchronization using GPS clocks and atomic clocks to ensure global consistency. This transactions also provide the snapshot isolation consistency level.

Dynamo [4], from Amazon, is a distributed key-value storage system. It uses a synthesis of well known techniques to achieve scalability and availability. The data is partitioned and replicated using consistent hashing, and consistency is facilitated by object versioning. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs a gossip based distributed failure detection and membership protocol, and provides eventual consistency [24], which allows for updates to be propagated to all replicas asynchronously.

Infinispan [5], from Red Hat/JBoss, is a transactional in-memory distributed key-value NoSQL storage system. Like Dynamo, Infinispan also partitions and replicates the data using consistent hashing. At its core Infinispan exposes a Cache interface which extends java.util.Map. It is also optionally backed by a

peer-to-peer network architecture to distribute state efficiently around the data grid. Data is written to stable storage asynchronously. Infinispan's transactions guarantee Read Committed (RC), Repeatable Read (RR) or Repeatable Read with Write Skew Check (RR+WS) consistency levels [23].

Because Infinispan is the system that will be used in this project, it will be explained in more detail in Section 3.2.

Walter [6] is a transactional globally-distributed key-value storage system. Unlike Dynamo, Walter replicates data asynchronously while providing strong guarantees within each site, by resorting to the Parallel Snapshot Isolation [6] consistency level. In order to implement the referred consistency level Walter relies on the notion of preferred sites and on counting sets [25] (csets). Walter uses a multi-version concurrency control within each site, and it can quickly commit transactions that write objects at their preferred sites or that use csets. For other transactions, Walter resorts to two-phase commit to check for conflicts.

3.3 Group Communication Systems

Group communication [26, 27] is a powerful paradigm for performing multi-point to multi-point communication by organizing processes in groups. Typically, a system that implements this paradigm is called a Group Communication System (GCS) and offers membership and reliable broadcast services with different ordering guarantees. GCSs are used at the heart of a large plethora of distributed systems, including transactional systems. They allow programmers to concentrate on what to communicate rather than on how to communicate. Broadcast services ensure all or some of the following properties:

- *Validity*: if a correct process broadcasts/multicasts a message m, then it eventually delivers m;
- *Uniform Agreement*: if a process delivers m, then all correct processes eventually deliver m;
- *Uniform Integrity*: for any message m, every process delivers m at most once, and only if m was previously broadcasted/multicasted by its sender;
- *Uniform Total Order*: if processes p and q both deliver messages m and m', then p delivers m before m' only if q delivers m before m'.

Uniform properties make life easier for application developers, as they apply to both correct and faulty processes. However, enforcing uniformity often has a cost and for this reason it is important to consider whether uniformity is strictly necessary.

Atomic Broadcast [28] (AB), also knowned as Total Order Broadcast, is a communication primitive that ensures that every participant receives all messages by the same order. A set of messages is broadcasted by invoking AB-broadcast and when their final order is known, they are AB-delivered. AB ensures all of the above properties.

Optimistic Atomic Broadcast [29] (OAB) is a variant of AB that exploits the fact that in a LAN, messages normally arrive at different sites exactly in

the same order. This assumption is called optimistic delivery order. A set of messages is broadcasted by invoking AB-broadcast and OAB-delivered as soon as they arrive, but only when their final order is known, they are AB-delivered. The OAB deliver primitive enables applications to overlap computation with communication. OAB ensures all of the above properties and one more:

– *Optimistic Order*: if a process p AB-delivers message m, then p has previously OAB-delivered m.

Reliable Broadcast [30] (RB) is similar to AB but does not guarantee total order in the delivery of messages. RB ensures all of the above properties except one: uniform total order. Because of that, RB skips the coordination phase that determines the total order of messages, so its execution time will be smaller than AB's execution time.

Atomic Multicast [31] (AMcast), also knowned as Total Order Multicast, is also a variant of AB. While AB sends messages to all the processes, AMcast just sends messages to a subset of processes. For every message m, m.dst denotes the groups to which m is multicasted. A message m is multicasted by invoking A-multicast(m) and delivered with A-deliver(m). AMcast ensures all of the above properties for m.dst and one more:

– *Uniform Prefix Order*: for any two messages m and m' and any two sites s and s' such that {s,s'}$\subset \{m.dst, m'.dst\}$, if s A-delivers m and s' A-delivers m', then either s A-delivers m' before m or s' A-delivers m before m';

3.4 Transactional Replication Techniques

Replication is a fundamental building block in the construction of highly available, fault-tolerant systems. As mentioned before, an important aspect that allows to classify existing systems and that has a significant impact on several of their key design choices is whether data is fully or partially replicated.

The key difference of this two techniques is that full replication places copies of data items at all replicas of the system, while partial replication only assigns copies of an individual data item to a set of that replicas. This way, full replication has the advantage of not needing expensive remote data accesses (all data is local) but is inherently unscalable, especially in presence of write-dominated workloads, because all the nodes need to be contacted upon commit. On the other hand, partial replication is highly scalable because a replica only has to execute the updates for data items of which it has local copies but it needs to deal with expensive remote data accesses that can be very detrimental especially in presence of workloads that promote them.

Other important aspect that allows to classify this systems is whether they provide transactional consistency (such as repeatable read [23], 1-copy serializability [32], extented update serializability [33]) or weaker, non-transactional consistency (such as eventual consistency [24], causal consistency [34], or redblue consistency [35]).

The focus of the next sections of this report is systems that provide transactional consistency. Table 1 shows all the algorithms described and their features.

System	Consistency Level	Broadcast Service	Replication Approach
DBSM [36]	1CS	AB	Certification
D2STM [37]	1CS	AB	Certification
ALC [38]	1CS	OAB+RB	Certification
SCert [39]	1CS	OAB	Certification
SPECULA [40]	1CS	AB	Certification
Kemme et al.[41]	N/A	OAB	Active
AGGRO [42]	N/A	OAB	Active
Polycert [43]	1CS	AB	Certification
P-Store [44]	1CS	AMcast	Partial
Ruivo et al.[45]	RR,RC,RR+WS	AMcast	Partial
GMU [8]	EUS	N/A	Partial

Table 1. System's features overview.

Full Replication

There are two main families of full replication techniques: passive and active. Other replication schemes combine aspects of the two previous techniques. An important example is certification-based replication, a scheme that is commonly employed on transactional systems.

In the next sections I explain some of their properties and advantages and disadvantages regarding each other. I also present some of the research made using those techniques.

Primary Backup

Primary backup [46], also known as master-slave or passive replication, is characterized by the existence of a replica, known as the master, that processes all requests and transfers the state updates to the remaining replicas, known as the slaves or backups. In most cases, slaves can also process read-only transactions.

Primary backup often assumes the fail-stop model [47]. When the master fails one of the slaves is elected to replace it, becoming the new master. One of the problems of this technique is that in write intensive workloads the master may become a bottleneck in the system, as it is responsible for all the computation, since the slaves do not share any workload.

State Machine Replication

State machine replication [48], also knowned as active replication, is characterized by having all replicas processing the same sequence of requests, by the exact same order. To ensure the consistency of the replicated data, state machine replication requires all operations to be deterministic, otherwise the state of each replica could diverge.

In transactional systems, requests are processed according to the global serialization order (GSO), which is normally defined by the AB protocol used to disseminate requests. One of the problems of this technique is that AB is a relatively slow communication primitive, as it requires that consensus is reached among all nodes.

Moreover, since all replicas have to execute all update requests, the ability to process them does not increase. On the contrary, it does in fact decrease, as the cost of group communication increases with the number of peers. However, as happens with passive replication, read-only requests can be processed in parallel at different replicas.

In order to improve performance several state machine based schemes normally use OAB instead of AB. As described above, OAB reduces the average delay for message delivery to the application. OAB considers the order messages arrive at each site as a first optimistic guess, and only if a mismatch of messages is detected, further coordination rounds between the sites are executed to agree on a total order. This way the total delivery order notification, which is available only after several communication steps (typically at least three), can be made after a single communication step.

In [41] this idea was further developed. They show how applications can take full advantage of the optimistic assumption by overlapping the coordination phase of the atomic broadcast algorithm with the processing of delivered messages. In particular, it was presented a replicated database architecture that employs this technique to overlap communication and transaction processing, thus providing high performance without relaxing transaction correctness (i.e., serializability).

Aggro [42] is another OAB based replication algorithm that aims at maximizing the overlap between communication and transaction processing. Unlike other OAB-based replication approaches, it does not require information on the transactions data access patterns prior to their actual execution and propagates the updates of yet uncommitted (but complete) transactions to the succeeding transactions. Conversely, it detects any possible discrepancy between the transaction schedule and the optimistic delivery order a posteriori, namely as soon as (and if) conflicts materialize.

Usually, in systems that ensure the spontaneous order property, these cases are uncommon, so the system overall performance will most probably increase.

Certification Based

Certification algorithms allow that the execution of a transactional request can take place at a single node, guaranteeing data coherency in the end, differently from the replication techniques described above. Specifically, these algorithms ensure that every replica agree on the outcome of a transaction at commit time relying on a distributed transaction certification algorithm.

They are usually based on the deferred update model [49] and use group communication primitives. According with this model, transactions are processed locally in one replica and then sent to the other replicas, at confirmation time.

It was first introduced in [36], as a scheme designed to synchronize a cluster of database servers in a multi-master environment

Unlike classic eager replication schemes (based on fine-grained distributed locking and atomic commit), which suffer from large communication overheads and distributed deadlocks, AB based certification schemes do not require any replica coordination during the transaction execution phase. Instead, transactions are executed locally in an optimistic fashion and consistency (typically, 1-Copy serializability) is ensured at commit-time, via a distributed certification phase that uses AB to enforce agreement on a common transaction serialization order, thus providing the following benefits:

- avoids distributed deadlocks, which are known to significantly limit scalability of eagerly replicated transactional systems;
- offers non-blocking guarantees in the presence of failures;
- allows for a modular implementation, where the complexity associated with failure handling is en-capsulated by the AB and group management system layers.

This model has several advantages such as:

- better performance, because several modifications are gathered and propagated together and execute the transaction in one replica, possibly near the client;
- reduced cost of inter node coordination, by imposing a single cluster-wide interaction per commit request;
- better support for fault tolerance, by simplifying the recovering of replicas;
- lower blocking rate, by the elimination of distributed deadlocks.

However, it also presents some disadvantages, like undesirably high abort rates in high conflict scenarios or with heterogeneous workloads that contain mixes of short and long-running transactions, because there is no synchronization during the execution of a transaction.

Existing certification-based replication algorithms can be classified into two main categories:

- *Non-voting schemes*: solutions that disseminate the whole transaction read-set to all replicas, allow each replica to certify transactions locally, by sending both the read-set and write-set via an AB primitive;
- *Voting schemes*: schemes that avoid broadcasting the read-set of transactions by sending (via AB) only the write-set.

Non-voting schemes are optimal in terms of communication steps, but it also makes them prone to generate very large messages and to overload the Group Communication System.

Voting schemes drastically reduce the network bandwidth consumption but they incur into the costs of an additional coordination phase along the critical path of the transaction commit, which can reduce significantly the performance.

The overhead of AB based certification schemes can be particularly detrimental in STM environments because they incur neither in disk access latencies nor in the over-heads of SQL statement parsing and plan optimization. This makes the execution time of typical STM transactions normally much shorter than in database settings and leads to a corresponding amplification of the overhead of inter-replica coordination cost.

Bloom Filter Certification [37] is a variant of the non-voting certification mechanism that relies on the space efficient encoding of Bloom Filters [50] to generate smaller messages. The probabilistic nature of the Bloom filter encoding, however, induces false positives in the certification phase, increasing the transaction abort rate.

One STM system that leverages BFC, at the cost of a user-tunable increase of the transaction abort probability, is D2STM [37]. The goal of D2STM is to leverage replication not only to improve performance, but also to enhance dependability. However, because transactions are only validated at commit time and no bound is provided on the number of times that a transaction will have to be re-executed due to the occurrence of conflicts, D2STM also presents the disadvantages mentioned above.

ALC [38] tackles that issue. In the core of the ALC scheme is the notion of asynchronous lease. Asynchronous leases are used by a replica to establish temporary privileges in the management of a subset of the replicated dataset.

ALC provides two significant advantages with respect to D2STM:

- relies on the cheaper Reliable Broadcast (RB) primitive to disseminate exclusively the write sets;
- shelters transactions from repeated aborts due to remote conflicts.

This leads to a significant reduction of the inter-replica synchronization overhead and to a throughput increase in high conflict scenarios because ALC does not need to atomically broadcast the write set and the (Bloom Filter encoded) read set of a committing transaction and avoids the optimistic certification approach used in D2STM, by using the notion of asynchronous lease.

SCert [39] tackles the same issue that ALC, but in a different manner. The key idea at the core of SCert is to reduce the time to disseminate the updates generated by committing transactions in order to achieve the following two complementary goals:

- provide executing transactions with fresher snapshots;
- detect conflicts earlier during transaction execution.

Those goals are achieved via a speculative approach, which leverages the same OAB service used in [41] and [42] and described above.

SPECULA [40] is other transactional replication protocol that also exploits speculative techniques but to achieve complete overlapping between replica synchronization and transaction processing activities.

The main idea is to execute the commit phase in a non-blocking fashion. This way, rather than waiting until the completion of the replica-wide synchronization phase, the results (i.e. the write set) generated by a transaction that

successfully passes a local validation phase are speculatively committed in a local multi-versioned STM, making them immediately visible to future transactions generated on the same node (either by the same or by a different thread).

In presence of misspeculations, SPECULA detects data and flow dependencies and aborts all the transactions related to that misspeculation.

Partial Replication

Partial replication is a variant of full replication, where one replica does not own all the data. Each replica only saves a determined subset of all the data in that system and that set can have several copies in the other replicas. The idea is that there is no need to all nodes to process a transaction, so that transactions can only be sent to replicas that maintain the data accessed by it. Therefore, the network communication and the replica coordination cost will be less thus improving the scalability of the system. However, by partitioning the data there is the need of dealing with expensive remote data accesses, every time a replica needs a data item that does not own.

Partial replication protocols can be divided in three groups [51]:

- *Genuine*: for each transaction T, the replicas that certificate T are the ones that have the data accessed by T;
- *Quasi-genuine*: for each transaction T the correct replicas that do not have data accessed by T, only store permanently the identifier of T;
- *Non-genuine*: for each transaction T, every replica store information about T, even if they do not have data acessed by T.

Non-genuine protocols [52, 53, 54] go against the goal of partial replication, because all the replicas have to be involved in a transaction, which leads to scalability issues similar to those experienced by full replication protocols.

P-Store [44] is genuine partial replication algorithm that ensures 1-Copy serializability [32]. It assumes a wide area network environment where sites are clustered in groups (e.g., data centers) and seeks to minimize costly and slow inter-group communication. In P-Store, an AMcast service is used to order transactions that operate on the same data items. To execute and certify each transaction, a single message is atomically multicasted.

The work in [45] also exploits the use of the AMcast primitive to ensure agreement on the transaction serialization order but in order to improve 2PC-based algorithms that provide weaker consistency guarantees [23]: Read Committed (RC), Repeatable Read (RR) and Repeatable Read with Write Skew Check (RR+WS).

Results show speed-ups of up to 40x when comparing the proposed algorithms with the previous ones.

GMU [8] is another genuine partial replication protocol for transactional systems, which exploits an highly scalable, distributed multiversioning scheme that relies on a vector clock [55] based synchronization mechanism to track both data and causal dependency relations among transactions, and guarantees

Extended Update serializability [56, 33] (EUS). Unlike existing multiversion-based solutions, GMU does not rely on a global logical clock, which represents a contention point and can limit system scalability. Also, GMU never aborts read-only transactions and spares them from distributed validation schemes, which is a major source of inefficiency on other systems.

GMU is the focus of the work presented in this report and will be explained in more detail in Section 3.4.

Caching in Partial Replication

As it may be impossible to partition the data in a way to ensure that every access performed by applications targets exclusively a single partition, several systems introduce an additional caching mechanism aimed at maintaining, at each node, a replica of the remote data items most frequently accessed by local transactions. If the requested data is contained in the cache (cache hit), this request can be served by simply reading the cache. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which takes more time.

As mentioned before, partially replicated systems need to deal with remote data accesses during processing because they only allow sites to store a subset of the application data, so they can also leverage the same mechanism. However, such mechanism must ensure that accessing cached data does not result in the violation of transaction semantics, so a transactional cache consistency maintenance algorithm is required to ensure that no transactions that access stale (i.e., out-of-date) data are allowed to commit.

Many such algorithms have been proposed in the literature and, as all provide the same functionality, performance is a primary concern in choosing among them. They can be divided into two classes according to whether their approach to preventing stale data access is detection-based or avoidance-based [57]. The key difference between them is that detection-based schemes are lazy, requiring transactions to check the validity of accessed data, while avoidance-based schemes are eager, as they ensure that invalid data is quickly (and atomically) removed from client caches. Both schemes allow data propagation (the newly updated value is installed at the remote site in place of the stale copy) or invalidation (removal of the stale copy from the remote cache so that it will not be accessed by any subsequent transactions).

In the following, I briefly describe three families of caching algorithms (see [58] for a more detailed description).

Server-Based Two-Phase Locking (S2PL) algorithms are detection-based algorithms that validate cached pages synchronously on a transaction's initial access to the page.

Callback Locking (CBL) algorithms are similar to C2PL, in that they are extensions of two-phase locking that support inter-transaction page caching. In contrast to the detection-based C2PL algorithm, however, CBL algorithms are avoidance-based.

Optimistic Two-Phase Locking (O2PL) algorithms are avoidance-based algorithms but, unlike CBL, they are more "optimistic" because they defer write intention declaration until the end of a transaction's execution phase.

There is no such algorithm that fits perfectly all the scenarios possible in these type of environments. There are often complex trade-offs among competing factors such as the amount of generated network traffic or the probability of transaction aborts. The avoidance/detection choice has seen to have a large impact on the number of messages sent, where detection leads to more messages. Regarding the choice of an optimistic or a pessimistic avoidance approach, with no read-write or write-write sharing, both approaches are roughly equal in performance. If sharing is present, using the optimistic approach can save messages. However, if sharing increases to the point were data contention arises, it can lead to significantly higher abort rates.

3.5 Self-tuning in Transactional Systems

With the advent of autonomic computing, that refers to the capability of a system to manage itself in order to provide optimal performance, the research made on transactional data grids also moves towards that direction.

The ultimate goal of this type of systems is to alleviate the developers/administrators from the hard and time-consuming task of profiling the application and selecting the most suitable protocol for each deployment or to avoid a system's misconfiguration that may lead to largely suboptimal performance in presence of heterogeneous workloads.

As already mentioned, distributed transactional platforms are complex systems, which result from the delicate intertwining of a number of systems operating at different levels. In the following I briefly overview self-tuning solutions aimed at optimizing three different layers/aspects of a distributed data grid, namely, i) the replication protocol, ii) the group communication system, iii) the elastic scaling manager (i.e. the system in charge of determining how many replicas should be used to achieve a target performance level).

Replication Protocol

Polycert [43] is a polymorphic certification protocol that allows for the simultaneous coexistence of multiple AB-based certification schemes, relying on machine-learning techniques, namely a regressor based on decision trees [59] and a reinforcement learning technique called UCB [60], to determine the optimal certification scheme on a per transaction basis.

MorphR [61] is a framework that allows to automatically adapt the replication protocol of in-memory transactional platforms according to the current operational conditions, also by relying on machine learning techniques, namely C5.0[1], a state of the art decision-tree classifier. It provides a set of interfaces with

[1] http://www.rulequest.com/see5-info.html.

precisely defined semantics and a generic, protocol-agnostic reconfiguration protocol that guarantees the correct switching between two arbitrary replication protocols, as long as they implement those interfaces. The replication protocols provided by MorphR are Primary-Backup, a 2PC based certification scheme and a total order based one.

Both PolyCert and MorphR are capable of achieving a performance extremely close to that of an optimal non-adaptive protocol in presence of non heterogeneous workloads, and significantly outperform any non-adaptive protocol when used with realistic, complex applications that generate heterogeneous workloads.

Group Communication System

In [62] its proposed an adaptive protocol that is able to dynamically switch between different total order broadcast (TOB) protocols. Unlike similar adaptive protocols, the transition between TOB protocols does not require the traffic to be stopped, allowing a smooth adaptation to changes in the underlying network. However, despite being an adaptive protocol, it does not focus on the conditions that trigger that adaptation, just in providing a generic switching procedure.

The work in [63] show that is also possible to self-tune the batching level of TOB protocols. Batching is a well known technique that allows boosting the throughput of TOB protocols by amortizing the per-message ordering overhead across a set of incoming messages. By combining analytical modeling and reinforcement learning techniques (same as Polycert), it is possible to minimize learning time and accumulate feedback from the operation of the system to enhance the self-tuning accuracy over time thus avoiding blind explorations of inadequately low batching values which would otherwise rapidly lead the system to trashing at high load, and making the self-tuning policy more accurate when using both techniques instead of exclusively the analytical model.

Elastic Scaling

The work in [64] introduces a pro-active scheme based on the K-nearest-neighbors [65] (KNN) machine learning approach for adding database replicas to application allocations in dynamic content web server clusters. This scheme monitors the system and application metrics in order to decide how many databases it should allocate to a given workload and incorporates awareness of system stabilization periods after adaptation in order to improve prediction accuracy and avoid system oscillations.

TAS [66] (Transactional Auto Scaler) is a system that provides the same functionality that the work mentioned above but for in-memory transactional data grids. TAS uses a performance forecasting methodology that exploits the joint usage of analytical modeling and machine-learning thus achieving high extrapolation power and good accuracy even when faced with complex workloads deployed over public cloud infrastructures.

4 Proposed Solution

4.1 Infinispan

As already discussed in Section 4.1, this work will be developed into Infinispan. Infinispan is a highly scalable data grid platform that allows data distribution or replication. It exposes a key-value store data model, and maintains data entirely in-memory relying on replication as its primary mechanism to ensure fault-tolerance and data durability. Both partial and full replication are supported.

As other recent NoSQL platforms, Infinispan opts for weakening consistency in order to maximize performance. It uses the classical 2PC algorithm to maintain data coherence and provides weak consistency guarantees [**?**]: Read Committed (RC), Repeatable Read (RR) and Repeatable Read with Write Skew Check (RR+WS).

In Infinispan architecture, each one of its nodes is composed by the following components:

- *Transaction Manager*: this component is responsible for the execution of transactions, either local or remote;
- *Lock Manager*: this component is responsible for managing the locks acquired by the transactions and is also able to detect distributed blockings. If there is a distributed blocking between two transactions, one of them will be canceled;
- *Replication Manager*: this component is responsible for the maintenance of data coherence between replicas. It certificates transactions through a 2PC algorithm and the Transaction Manager;
- *Persistent Storing*: this component is responsible for guaranteeing that the storing and loading of data in a persistent manner (in a disk or a DB, local or remote). If this component is active, it can work on one of the following execution modes:
  - *Activation/Passivation*: the data is stored persistently or in memory. When a item of that data is needed, it is moved to the memory (and is deleted from the disk/DB). When it is needed no more, that item is deleted from memory and stored again persistently;
  - *Load/Store*: the data is stored in both memory and persistently. When a data item is needed, a copy of that item is sent to memory.
- *Group Communication System*: this component is responsible for the maintenance of group members information (including fault detection) and offers support for the communication between replicas.

A client can initiate the execution of a transaction in any replica of the system. That transaction is executed locally. The information maintained about a transaction data is the key of the accessed item and its value. If there is a modification in some key, the old is also maintained.

When the execution of a transaction ends, its write set (WS) is sent to the other replicas involved in order to be certified, through the 2PC algorithm. The transaction is then confirmed or canceled in those replicas.

As mention, the Group Communication System component provides support for the communication between replicas. Infinispan uses JGroups for this matter.

4.2 GMU

As mentioned before, GMU (Genuine Multiversion Update serializability) is a genuine partial replication algorithm that provides multiversion concurrency control, which relies on a distributed synchronization scheme that exploits vector clocks. Every locally stored versions of a data item is paired with a scalar, monotonically increasing (integer) clock, which is used to totally order the commit events of transactions that update this data items.

In the GMU architecture, each node contains the following data structures:

- *CommitQueue*: an ordered queue whose entries are tuples <T, VC, status> where T is a transaction, VC is its current vector clock, and status is a value in the domain{pending, ready}, that is used to commit transactions using the same commit VC and in the same total order in the nodes that maintain data of that transactions;
- *CommitLog*: a list that maintains, for each committed transaction, the tuple <T, VC, updatedKeys>, where T is the identifier of a committed transaction, VC is its vector clock and updatedKeys is the set of keys locally stored by process p that T has updated during its execution, thus providing the most recent local snapshot that is visible by a transaction.

In the execution phase of a transaction T, GMU stores the following information about it:

- the transaction vector clock, an array of scalar (integer) logical timestamps, having cardinality equal to the number of nodes in the system;
- the transaction read-set, which stores the set of identifiers of the keys read by T;
- the transaction write-set, which stores, as a set of pairs <key, value>, the identifiers and values of the keys written by T;
- an array of booleans, which has an entry for each node in the system and each entry stores the flag indicating whether T has already issued a read operation on a key stored by that node.

Write operations are simply handled by storing the identifier and the key value in the transaction's write-set. On the other hand, read operations require a more complex management as it is necessary to determine which one among the versions maintained by the data platform should be visible to the transaction.

For space constraints, I cannot provide a detailed description of the read handling logic (which, of course, can be found in the GMU paper [8]). However, in the following I will provide a discussion on the three main rules used in GMU to determine versions visibility, as this is necessary to discuss, in Section 4.3, what are the key challenges to be addressed in order to design a caching algorithm for GMU.

Specifically, GMU determines which version of a key K should be returned upon the execution of a read operation on node N by transaction T according to the following rules:

- *Rule 1 - Reading Lower Bound*: as data is replicated among multiple nodes, it is possible that node N may have not yet finalized the commit of a transaction T* whose effects have been already observed by transaction T (during a previous operation) on another node. In order to avoid consistency issues, if the N-th entry in the vector clock of T is larger than the N-th entry of the most recent vector clock in the commit log of N, T is blocked until all dependencies are solved;
- *Rule 2 - Reading Upper Bound*: in order to maximize data freshness, if transaction T is reading for the first time on node N, the vector clock of T is updated in its N-th entry with the N-th entry of the vector clock retrieved from the read;
- *Rule 3 - Data Version Selection*: as in a classic, non-distributed multi-version concurrency control schemes [67], whenever there are multiple entrys of some data item, the entry selected will be the most recent one that has a vector clock smaller or equal than the vector clock of T.

GMU has been recently integrated in Infinispan and results show that it achieves linear scalability while introducing negligible overheads (less than 10%) with respect to solutions ensuring non-serializable semantics in a wide range of workloads. EUS is sufficiently strong to ensure the correctness of complex OLTP workloads (such as TPC-C), but also weak enough to allow for efficient and scalable implementations.

4.3 Caching in GMU

As mentioned before, partially replicated transactional systems need to deal with expensive remote data accesses that can severely hinder their performance, especially when faced with workloads that promote them. To face this problem, this type of systems often resort to a caching mechanism that aims at maintaining, at each node, a replica of the remote data items most frequently accessed by local transactions, thus diminishing the need for remote data accesses. Unfortunately, GMU does not provide such mechanism, so it also suffers from the problem mentioned above.

As already mentioned in Section 2, the key goal of this project is to design an efficient, self-tuning caching algorithm for GMU. In order to achieve this goal, there are two main problems that need to addressed:

- it is necessary to ensure that accessing cached data does not result in the violation of the correctness properties ensured by GMU, i.e., extended update serializability (EUS);
- it is important that the employment of the caching algorithm does not have a negative impact on the freshness of the data observed by transactions, or it may end up hindering performance, rather than improving it.

So far, I have already done some progress in the design of possible solutions, which I briefly overview in the following. Note that these are still preliminary ideas whose correctness and efficiency have to be thoroughly evaluated during the next phase of this project.

Ensuring data consistency

To ensure data consistency, it is necessary to define efficient mechanisms to know, every time an entry is stored in cache, if that entry will be safely read by future transactions originated on that node. The challenge is to define how the entry's owner should inform the other nodes in the system about the occurrence of updates to its keys. A simple solution that would preserve EUS would be to force a node N, which owns a key K that is being updated by a transaction T, to inform all the other nodes in the system that store K in their cache about its update in a synchronous fashion, namely during the execution of 2PC. This is however undesirable because it would significantly slow down the commit phase and violate the genuineness property that is at the basis of the design of GMU and is fundamental to maximize scalability.

The solution I propose as a caching scheme, which I refer as L1 cache, is based on the idea that each entry stored will have the additional meta-data:

- *readVersion*: a vector clock that represents the freshness of the entry, i.e., the most recent vector clock committed that the entry is valid for;
- *creationVersion:* a vector clock that represents the first time the entry was stored in the L1 Cache.

By using these two pieces of information, it is possible to implement a set of rules determining version visibility in GMU without compromising serializability. Specifically:

- *Rule 1*: to determine if the entry in the L1 cache is too obsolete given the causal dependencies developed by the transaction, it is sufficient to check if the transaction's vector clock is not larger than the entry's readVersion;
- *Rule 2*: if the entry in the L1 cache was retrieved from a node from which the transaction has not yet read, the transaction can be updated with the readVersion of the cached version;
- *Rule 3*: this rule requires ensuring that the entry observed by a read is the most recent with respect to transaction's vector clock. Since we are guaranteed to have in the L1 cache the entire set of entries of a data item, we will need to force a cache miss in case the cached entry readVersion is smaller than the transaction's vector clock. In this case, in fact, the entry may have already been overwritten at the remote node, and by reading the cached entry we may violate serializability. In case this check succeeds, it is then safe to select the cached entry having the most recent creationVersion.

Maximizing data freshness

As seen in Section 4.2, GMU maximizes the freshness of the data observed by transactions by updating the transaction's vector clock upon the first read on a remote node N (Rule 2). Specifically, in this phase, GMU verifies what is the freshest commit log entry of N visible for T, considering T's previous read. By interjecting a caching mechanism that may contain obsolete information (i.e. information updated at the time in which the requested data item was put in the cache), we run the risk of letting transactions observe arbitrarily old snapshots. Note that this is not only an issue for applications who have specific constraints on the recency of the snapshots they observe, but can also lead to unacceptable abort rates. In fact, in GMU, update transactions that observe obsolete snapshots are doomed to be aborted at commit time, during the transaction's validation phase.

By analyzing the caching algorithm proposed above, it appears that there is a trade-off between the benefits associated with the possibility of serving a possibly obsolete data from the L1 cache, and the freshness of the data observed by transactions, which could be maximized by bypassing the cache and forcing a remote access to the data owners. Also, some data is more frequently updated than other, so the risk of observing stale data in cache is higher for those data items.

I plan to design a self-tuning algorithm that would automatically identify which entries are beneficial to cache. There is two main challenges that will need to be addressed:

- how to evaluate the benefits of caching or not caching an item;
- how efficient and scalable the self-tuning mechanism will be.

At first sight, to address the first challenge, the update frequency is clearly a useful indicator, but the identification of the thresholds on the update frequency to be used to determine an object's cacheability raises another challenge. Ideally, the caching mechanism should be capable of determining in an automatic fashion the values of these threshold. Hence, I plan to investigate techniques that will allow the self-tuning of this parameter.

Regarding the second challenge, the ideal solution is to determine the cacheability of objects at the granularity of a single key. However, as the number of keys globally maintained by a NoSQL data grid can be extremely large, this raises the problem of how to efficiently gather information on update frequencies and objects cacheability. So I also plan to investigate space-efficient mechanisms that will allow the self-tuning scheme to cope with arbitrary large data sets.

5 Evaluation Method

This work will be integrated into Infinispan. Therefore, the performance of the proposed solution will be determined by experimental evaluations.

5.1 Metrics

The most important metrics to evaluate the performance of the algorithms are:

- the number of transactions delivered at each site;
- the number of transactions processed by the system;
- the number of aborted transactions;
- the number of cache hits/misses;
- the freshness of cached items.

These metrics will allow me to compare the system's performance with and without the caching mechanism and assess the influence of a cache hit in a transaction's execution.

As a result, I expect to get insights on how to configure the caching mechanism for optimal performance and on which scenarios, if any, their use allows significant improvement on the performance of the system.

5.2 Benchmarks

The performance of the proposed solution will be assessed with well established benchmark suites for transactional systems, such as:

- *TPC-C* [68]: is a benchmark that has been ported to work on key-value stores that generates workload representative of OLTP environments and is characterized by complex and heterogeneous transactions, with very skewed access patterns and high conflict probability;
- *Radargun*[2]: is a benchmark created by RedHat specifically for this transactional data grids. With Radargun it is possible to compare the performance of several distributed caches (such as Infinispan, Ehcache[3], Coherence[4], etc.), in different scenarios and, by imposing a fairly high load on the different nodes of the system, it allows us to assess the maximum throughput of each configuration;
- *STAMP* [69]: is a comprehensive benchmark suite for evaluating TM systems that has been ported to work on key-value stores. STAMP includes eight applications and thirty variants of input parameters and data sets in order to represent several application domains and cover a wide range of transactional execution cases;
- *Bank* [17]: is a simple benchmark that has the advantage of providing a fine control on the conflict rate. It simulates the transfer of money amounts between variables representing distinct bank accounts.

[2] https://github.com/radargun/radargun/wiki/.
[3] http://ehcache.org/.
[4] http://www.oracle.com/technetwork/middleware/coherence/overview/index.html.

6 Work Planning

Future work is scheduled as follows:

- January 7 - May 31, 2013: Detailed design and implementation of the proposed architecture, including preliminary tests.
- June 1 - June 31, 2013: Perform the complete experimental evaluation of the results.
- July 1 - July 31, 2013: Write a paper describing the project.
- August 1 - October 15, 2013: Finish the writing of the dissertation.
- October 15, 2013: Deliver the MSc dissertation.

7 Conclusion

This work addresses the implementation of a caching mechanism for partially replicated transactional systems. As seen in the related work presented in this report, partially replicated systems can leverage the implementation of such mechanism. However, caching mechanisms can perform differently when faced with different workloads. To address this problem, an efficient, self-tuning caching mechanism was proposed.

The report is concluded with an explanation of what will be implemented, experimentally compared, and validated in the proposed solution.

Acknowledgments

I am grateful to my advisor, Paolo Romano, for his support and comments during the preparation of this report. This work has been partially supported by the FCT project ARISTOS (PTDC/EIA- EIA/102496/2008).

References

1. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. USENIX Association, 2006, pp. 15–15.
2. D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.
3. J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012, pp. 251–264.

4. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. ACM, 2007, pp. 205–220.
5. F. Marchioni and M. Surtani, *Infinispan Data Grid Platform.* Packt Publishing.
6. Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011, pp. 385–400.
7. M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *Proceedings of the 33rd international conference on Very large data bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1150–1160.
8. S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *ICDCS*. IEEE, 2012, pp. 455–465.
9. E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.
10. B. C. Ooi, "Cloud data management systems: Opportunities and challenges," *Semantics, Knowledge and Grid, International Conference on*, vol. 0, 2009.
11. N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '95. ACM, 1995, pp. 204–213.
12. J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Sci. Comput. Program.*, pp. 172–185.
13. T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '07. ACM, 2007, pp. 221–228.
14. T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proceedings of the 20th international conference on Distributed Computing*, ser. DISC'06. Springer-Verlag, 2006, pp. 284–298.
15. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, ser. PODC '03. ACM, 2003, pp. 92–101.
16. D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th international conference on Distributed Computing*, ser. DISC'06. Springer-Verlag, 2006, pp. 194–208.
17. M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06. ACM, 2006, pp. 253–262.
18. R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proc. of PPOPP*, 2008.
19. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc.
20. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. ACM, 2003, pp. 29–43.
21. M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. USENIX Association, 2006, pp. 335–350.

22. T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, ser. PODC '07. ACM, 2007, pp. 398–407.
23. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '95. ACM, 1995, pp. 1–10.
24. S. Gustavsson and S. F. Andler, "Self-stabilization and eventual consistency in replicated real-time databases," in *Proceedings of the first workshop on Self-healing systems*. ACM, 2002, pp. 105–107.
25. M. Letia, N. Preguiça, and M. Shapiro, "Consistency without concurrency control in large, dynamic systems," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 29–34, Apr. 2010.
26. G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Comput. Surv.*, vol. 33, no. 4, pp. 427–469, Dec. 2001.
27. D. Powell, "Group communication," *Commun. ACM*, vol. 39, no. 4, pp. 50–53, Apr. 1996.
28. V. Hadzilacos and S. Toueg, "Distributed systems (2nd ed.)," S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, ch. Fault-tolerant broadcasts and related problems, pp. 97–145.
29. F. Pedone and A. Schiper, "Optimistic atomic broadcast: a pragmatic viewpoint," *Theor. Comput. Sci.*, vol. 291, no. 1, pp. 79–101, Jan. 2003.
30. J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 251–273, Aug. 1984.
31. 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.
32. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.
33. R. C. Hansdah and L. M. Patnaik, "Update serializability in locking," in *ICDT 86, International Conference on Database Theory, Rome, Italy, September 8-10, 1986, Proceedings*, ser. Lecture Notes in Computer Science, vol. 243. Springer, 1986, pp. 171–185.
34. M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, March 1995.
35. C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012, pp. 265–278.
36. F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, July 2003.
37. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D^{2}STM: Dependable distributed software transactional memory," in *Proc. of PRDC*. IEEE CS, 2009, pp. 307–313.
38. N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proc. of Middleware*. LNCS, Springer, 2010.
39. ——, "Scert: Speculative certification in replicated software transactional memories," in *The 4th Annual International Systems and Storage Conference*. IBM Research, 2011.

40. S. Peluso, J. C. M. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues, "Specula: Speculative replication of software transactional memory," in *The 31st Symposium on Reliable Distributed Systems*. IEEE, 2012.
41. B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE TKDE*, vol. 15, no. 4, pp. 1018–1032, 2003.
42. R. Palmieri, F. Quaglia, and P. Romano, "AGGRO: Boosting stm replication via aggressively optimistic transaction processing," *Proc. of NCA*, pp. 20–27, 2010.
43. M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: polymorphic self-optimizing replication for in-memory transactional grids," in *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, ser. Middleware'11. Springer-Verlag, 2011, pp. 309–328.
44. N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *Proc of SRDS*. IEEE CS, 2010, pp. 214–224.
45. P. Ruivo, M. Couceiro, P. Romano, and L. Rodrigues, "Exploiting total order multicast in weakly consistent transactional caches," in *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*. IEEE Computer Society, 2011, pp. 99–108.
46. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Distributed systems (2nd ed.)," S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, ch. The primary-backup approach, pp. 199–216.
47. C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
48. F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.
49. J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '96. ACM, 1996.
50. B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
51. N. Schiper, R. Schmidt, and F. Pedone, "Optimistic algorithms for partial database replication," in *Proceedings of the 10th international conference on Principles of Distributed Systems*, ser. OPODIS'06. Springer-Verlag, 2006, pp. 81–93.
52. D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation," in *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '07. IEEE Computer Society, 2007, pp. 290–297.
53. D. Serrano, M. Patiño Martínez, R. Jiménez-Peris, and B. Kemme, "An autonomic approach for replication of internet-based services," in *Proceedings of the 2008 Symposium on Reliable Distributed Systems*, ser. SRDS '08. IEEE Computer Society, 2008, pp. 127–136.
54. J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí, "Sipre: a partial database replication protocol with si replicas," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. ACM, 2008, pp. 2181–2185.
55. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978.
56. A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," PhD Thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, Tech. Rep., 1999.

57. M. J. Franklin, M. J. Carey, and M. Livny, "Transactional client-server cache consistency: alternatives and performance," *ACM Trans. Database Syst.*, vol. 22, no. 3, pp. 315–363, Sep. 1997.
58. M. J. Franklin, "Caching and memory management in client-server database systems," Ph.D. dissertation, 1993.
59. J. R. Quinlan, *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.
60. P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, May 2002.
61. M. Couceiro, P. Romano, P. Ruivo, and L. Rodrigues, "Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation," INESC-ID, Tech. Rep., Dec. 2012.
62. J. Mocito and L. Rodrigues, "Run-time switching between total order algorithms," in *Proceedings of the Euro-Par 2006*, ser. LNCS. Springer-Verlag, 2006, pp. 582–591.
63. P. Romano and M. Leonetti, "Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning," in *Computing, Networking and Communications (ICNC), 2012 International Conference on*. IEEE, 2012, pp. 786 –792.
64. J. Chen, G. Soundararajan, and C. Amza, "Autonomic provisioning of backend databases in dynamic content web servers," in *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ser. ICAC '06. IEEE Computer Society, 2006, pp. 231–242.
65. E.-H. Han, G. Karypis, and V. Kumar, "Text categorization using weight adjusted k-nearest neighbor classification," in *Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, ser. PAKDD '01. Springer-Verlag, 2001, pp. 53–65.
66. D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional auto scaler: elastic scaling of in-memory transactional data grids," in *Proceedings of the 9th international conference on Autonomic computing*, ser. ICAC '12. ACM, 2012, pp. 125–134.
67. P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981.
68. F. Raab, "Tpc-c - the standard benchmark for online transaction processing (oltp)." in *The Benchmark Handbook*. Morgan Kaufmann, 1993.
69. C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing." in *IISWC*. IEEE, 2008, pp. 35–46.