Journal of Universal Computer Science, vol. 20, no. 14 (2014), 1876-1902
submitted: 28/3/14, accepted: 11/11/14, appeared: 1/12/14 © *J.UCS*

Epidemia: Variable Consistency for Transactional Cloud Databases

Itziar Arrieta-Salinas, José Enrique Armendáriz-Iñigo
(Departamento de Ingeniería Matemática e Informática
Universidad Pública de Navarra, 31006 Pamplona, Spain
{itziar.arrieta, enrique.armendariz}@unavarra.es)

Joan Navarro
(Research Group of Internet Technologies and Storage
La Salle - Ramon Llull University, 08022 Barcelona, Spain
jnavarro@salleurl.edu)

Abstract: Classic replication protocols running on traditional cluster-based databases are currently unable to meet the ever-growing scalability demands of many modern software applications. Recent cloud-based storage repositories overcome such limitations by fostering availability and scalability over data consistency and transactional support. However, many applications that cannot resign from their transactional nature are unable to benefit from the cloud paradigm. This paper presents Epidemia, a distributed storage architecture featuring a hybrid approach that combines classic database replication with a cloud-inspired infrastructure to provide transactional support and high availability. This architecture is able to offer different consistency levels according to the client demands, thanks to a replication strategy based on epidemic updates in which the replicas of each data partition are organized hierarchically. Additionally, the behavior of a prototype implementation under different workload scenarios is evaluated. Conducted experiments verify that (1) configuration parameters such as the partitioning scheme or the replication protocol play a crucial role on system's throughput, and (2) the existence of replica hierarchies that are asynchronously updated is able to alleviate the scalability limitations of traditional replicated databases by directing transactions that tolerate a certain staleness in the versions of retrieved data items to these replicas.

Key Words: Distributed Databases, Transaction Processing, Cloud Computing, Elasticity, Data Consistency, Transactions
Category: H.2.4

1 Introduction

The ambitious requirements regarding availability and fault tolerance demanded by many modern software applications entail the need for replicating immense amounts of data. Cluster-based databases have traditionally been considered as the proper choice [Daudjee & Salem(2006), Wiesmann & Schiper(2005)] despite the scalability limitations derived from the cost of maintaining strong consistency among replicas. [Gray et al.(1996)]. This represents a considerable drawback, as many modern systems (e.g., Web 2.0 applications) require high availability and scalability to cope with an ever-growing storage demand.

This context has motivated a new class of data storage systems called NoSQL (Not only SQL), which operate in cloud platforms and therefore come in hand with a promise of high scalability at a low cost. First NoSQL systems (ranging from raw data storage systems such as HDFS [HDFS(2014)] to key-value data stores such as Dynamo [DeCandia et al.(2007)], including table-oriented solutions like Bigtable [Chang et al.(2008)]) are able to achieve high performance and scalability levels by relaxing traditional ACID (Atomicity, Consistency, Isolation and Durability) properties and relying on weak consistency models such as eventual consistency [Vogels(2009)], which entails that queries may obtain outdated (though consistent) values until an unknown point in time. The data semantics considered in this kind of systems are commonly called BASE (Basically Available, Soft state and Eventual consistency) [Brewer(2012)] in opposition to the ACID properties.

Although the functionality provided by these cloud stores suffices for their target applications (e.g., web indexing, multimedia storage or content delivery networks), there are still many use cases which cannot take advantage of the cloud paradigm because they are unable to resign from their transactional nature.

Latest trends derived from NoSQL systems attempt to overcome this drawback by providing transactional support to a certain extent while meeting the principles of the cloud philosophy. Some of the most representative examples of cloud-based systems that provide transactional functionalities include Amazon DynamoDB [Sivasubramanian(2012)], Relational Cloud [Curino et al.(2011a)], ElasTraS [Das et al.(2009)], Google Megastore [Baker et al.(2011)] and ecStore [Vo et al.(2010)]. In general, these solutions restrict the scope of transactional support in different ways to provide high availability and elasticity.

This paper presents *Epidemia*, a distributed storage architecture that implements transactional support by using classic database replication techniques over a cloud-inspired infrastructure. This hybrid approach is aimed at offering a broad range of QoS levels according to application demands by varying the tradeoff between consistency and availability, thanks to a replication strategy based on epidemic updates. Thoroughly, the contributions of this paper are:

- A revision of the key challenges to provide transactional support in cloud systems and current approaches to face them.
- A novel architecture that exploits data replication using novel cloud tendencies, aiming at providing a highly available and elastic service with transactional support. This architecture consists of a dynamic set of cluster-based databases, each maintaining a hierarchy of versions where the topmost level of the hierarchy holds the newest version and consists of a set of replicas that are controlled by a replication protocol (which is determined depending on the current workload characteristics). The rest of hierarchy levels are

updated in an epidemic way by asynchronously propagating updates from one level to the next. Thus, depending on the consistency level demanded by a transaction, it can be forwarded to replicas containing newer or older versions.

- A set of correctness arguments that ensure the feasibility of this approach.
- An empirical evaluation using the standard benchmarks proposed by the YCSB [Cooper et al.(2010)] over a prototype implementation of Epidemia to assess the influence of different configuration settings (e.g., the partitioning scheme, the replication protocols used or the arrangement of backups into hierarchy levels) on system's throughput.

The remainder of this paper is structured as follows. Section 2 explores the key challenges on providing transactional support in cloud systems and extracts the main lessons learned from current approaches. Section 3 is devoted to the motivation and system model of Epidemia. Section 4 outlines a general correctness proof that shows its feasibility. Section 5 presents the evaluation of the implemented prototype following the proposed architecture. Finally, Section 6 summarizes the conclusions of this work.

2 Key Challenges to Provide Transactional Support in Cloud Systems

Current solutions for providing transactional support in cloud environments can be abstracted according to the diagram depicted in Fig. 1, which splits the main functionalities of these systems into four logical layers: i) the *Workload Management* layer, which monitors resource utilization and deals with data allocation issues; ii) the *Transaction Management* layer, responsible for the correct execution of distributed transactions; iii) the *Replication Management* layer, devoted to handling data replication across nodes to ensure data availability; and iv) the *Storage Management* layer, which ensures data durability over physical media. In the following, we elaborate on the specificities of each layer and review existing strategies to deal with them.

2.1 Workload Management

Cloud-based architectures feature an elastic scale-out to handle varying workloads [Curino et al.(2011b), Curino et al.(2010), Elmore et al.(2011)], thus enabling an attractive pay-as-you-go model. Therefore, a critical challenge is to minimize the operating costs while fulfilling service level agreements. In the following, we highlight some of the key concepts concerning different strategies to maximize performance while minimizing resources usage.

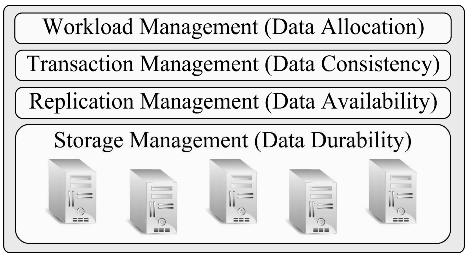


Figure 1: An abstraction of a cloud-based storage architecture with transactional support.

2.1.1 Data Partitioning

An efficient way to support transactions in the cloud is to reduce the interaction among replicas to the minimum. A common solution is to partition data in such a way that as many transactions as possible can be entirely executed within one single partition, thus requiring no coordination with the rest of data partitions [Curino et al.(2010), Cheung et al.(2012), Pavlo et al.(2012)].

For instance, Relational Cloud [Curino et al.(2011a)] recurs to partitioning upon detecting that a single host is unable to handle an entire database. To partition a database, the system analyzes queries to minimize the number of transactions that need to access several partitions. This is done by a module called Schism [Curino et al.(2010)], which uses a graph-based partitioning algorithm to achieve efficient database partitions, where partitioning is done even at the table level. On the other hand, Kairos [Curino et al.(2011b)] is the component responsible for monitoring and consolidating databases in Relational Cloud. In Kairos, the consolidation is stated as a non-linear optimization program, aiming to minimize the number of servers and balance load while achieving near-zero performance degradation.

Another approach for tackling partitioning issues is physiological partitioning (PLP) [Pandis et al.(2011)], a transaction processing technique that logically partitions the physical data accesses. To alleviate the difficulties imposed by page latching and repartitioning, PLP uses a new physical access method inspired by a multi-rooted B+Tree.

2.1.2 Live migration

In line with the problem of elasticity in cloud environments, the task of *live migration* [Elmore et al.(2011), Das et al.(2011)] consists in performing a data migration process (which might be motivated by cost-saving or performance considerations) from one or more servers to another while interfering with other operational processes as little as possible. Some representative approaches that address this issue are Zephyr [Elmore et al.(2011)], dedicated to the live migration of shared-nothing transactional databases; and Albatross [Das et al.(2011)], which delves into the case where data is stored in a network attached storage.

2.1.3 Load Balancing

Load balancing consists in forwarding incoming requests to the most appropriate resource. This decision can be taken according to different criteria.

For instance, in [Brantner et al.(2008)], system resources can be managed according to the economic constraints defined by the user. Likewise, a distributed replica placement algorithm is introduced in [Shorfuzzaman et al.(2012)], which is used to determine the position of a minimum number of replicas. This strategy finds the optimal tradeoff between the overall cost of replicating an object and the QoS satisfaction for a given traffic pattern. In a similar way, ElasTraS [Das et al.(2010)] includes a load balancing algorithm that adds or removes Owning Transaction Managers (the components that own data partitions and provide transactional guarantees) when observing a change in load and usage patterns over a period of time.

2.2 Transaction Management

As mentioned before, one of the key challenges when partitioning a database is maximizing the proportion of single-partition transactions. However, there may be cases, especially in dynamic environments such as cloud applications, in which some transactions (named multi-partition transactions) need to access several partitions. The issue here is twofold: on the one hand, network stalls appear since transactions are fragmented [Jones et al.(2010)] and different fragments are executed at different partitions; and, on the other hand, some coordination has to be provided to commit a transaction and thus maintain global consistency.

Regarding the first issue, some solutions choose to speculatively execute transactions that are ordered after a fragment that is waiting for its commit [Jones et al.(2010)], or to use transaction flow graphs to determine rendezvous points along the fragments of a multi-partition transaction [Pandis et al.(2010)]. To commit a multi-partition transaction and thus give a consistent view of data through different partitions, several techniques have been proposed so far: a single coordinator [Jones et al.(2010)], multiple coordinators [Maia et al.(2010)], two-phase commit [Bernstein et al.(1987)] and its variants [Aguilera et al.(2009), Vo et al.(2010)] or rendezvous protocols [Pandis et al.(2010)].

2.3 Replication Management

In general, cloud database systems have put aside traditional replication techniques [Daudjee & Salem(2006), Wiesmann & Schiper(2005)] to overcome their scalability limitations [Gray et al.(1996)]. Thus, instead of relying on full replication approaches, cloud storage systems typically exploit data partitioning and replicate partitions up to a given K level, so that there exist up to K physical copies of each data item [Das et al.(2009), Vo et al.(2010), Curino et al.(2010), Jones et al.(2010), Maia et al.(2010)]. The most common replication technique is the primary-backup strategy, with either an optimistic approach [Vo et al.(2010)] or a pessimistic one [Jones et al.(2010)]. Replication can also be done by means of state machine replication [Maia et al.(2010)] or Paxos [Aguilera et al.(2009)]. Another approach consists in relying on a fault-tolerant distributed storage system, thus delegating replication to the lower level of its architecture as done in ElasTraS [Das et al.(2009)].

On the subject of which partitions should be replicated and to what level, there are different alternatives. In the case of read-only transactions, a possible solution is to replicate their associated partitions in all replicas to exploit the benefits of access locality [Vo et al.(2010)]. As for update transactions, the replica placement policy can be defined by means of graph analysis [Curino et al.(2010)], histogram analysis of accesses to a given range, or monitoring the workload until the K level is met [Vo et al.(2010)].

2.4 Storage Management

Storing data to permanent media ensures data durability but may decrease system performance since (1) access times are considerable, (2) hardware failures are frequent [Schroeder & Gibson(2007)], and (3) the bottleneck effect appears when several concurrent operations are issued against a single storage device. Therefore, latest approaches in cloud-based storage systems attempt to keep as much data as possible in main memory.

These in-memory solutions are used for enhancing the performance and scalability of both SQL databases [Stonebraker et al.(2007)] and key-value stores [Lakshman & Malik(2010), Vo et al.(2010)]. In these solutions, each replica maintains all its partitions in main memory, therefore avoiding intensive writing to disk [Stonebraker et al.(2007)], plus saving disk stalls and the cost of a distributed storage [Das et al.(2009)]. In this case, the replication degree needs to be high enough to ensure durability. Other systems [Lakshman & Malik(2010), Vo et al.(2010)] rely on in-memory data structures to reduce response time, but periodically dump this information to disk to ensure durability.

On the other hand, the thread-to-data policy has been shown to be effective through exploiting the regular pattern of data accesses [Jones et al.(2010), Pandis et al.(2010)]. Together with this, the use of stored procedures avoids any interaction with the user during the transaction execution [Cheung et al.(2012)], hence removing client stalls and allowing to trivially serialize single-partition transactions.

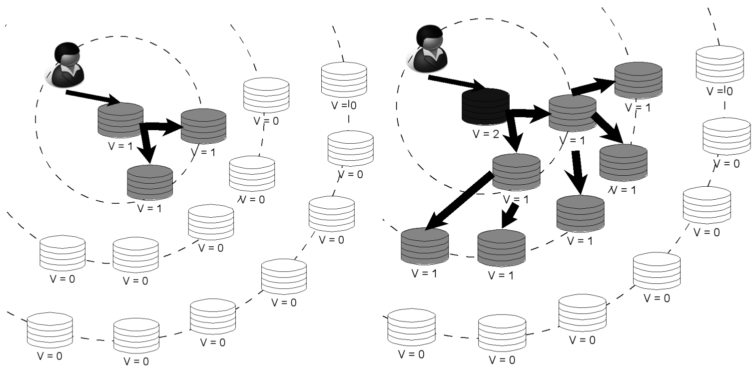
3 Design Rationales of Epidemia

This section describes the system architecture proposed in this work, which combines a cloud inspired scheme with traditional database replication concepts to provide transactional support as well as high data availability by offering different consistency levels according to the demands of client applications.

3.1 Motivation

It is well known that database replication protocols perform differently depending on the workload characteristics. For instance, a read intensive application will probably obtain a higher throughput using a *primary-backup* protocol [Daudjee & Salem(2006)] than with other techniques. In primary-backup approaches, only the replica that acts as primary is in charge of executing all updates. Hence, the protocol has to concern only about propagating updates to the backups. However, the throughput of update operations in primary-backup solutions is limited by the capacity of the primary [Gray et al.(1996)]. In contrast, a database whose items are frequently updated might benefit from an *update-everywhere* replication strategy that relies on total order broadcast [Wiesmann & Schiper(2005)]. In this kind of protocols, updates can be performed by any available replica, thus avoiding the potential bottleneck and single point of failure phenomena derived from having a single primary. Nevertheless, update-everywhere approaches can only scale up to a certain limit, as the cost of propagating updates increases with the number of involved replicas. In other words, database clusters do not scale due to the strong consistency maintained among replicas, thus leading to network and database stalls [Armendáriz-Iñigo et al.(2007)].

This problem could be alleviated if some (let us say M) of the replicas involved in the replication protocol acted as primaries for other backup replicas ($K - M$), which would asynchronously receive updates from their respective primaries. At the same time, backup replicas could act as primaries for other replicas, thus creating a hierarchy where updates would be propagated in an epidemic way. Therefore, replicas closer to the core will have the most recent values for data items, whereas replicas of lower levels will have older versions of the data items as shown in Fig. 2, which depicts an example of a primary-backup partition (i.e., $M = 1$) with $K = 15$ replicas. Initially, a client executes a transaction that updates the primary replica, which will propagate the changes



(a) t_1 commits and gets propagated ($V = 1$).

(b) t_2 commits ($V = 2$)

Figure 2: Example of the epidemic propagation of two update transactions t_1 and t_2.

to its associated secondaries (Fig. 2.a). The secondaries respectively behave as pseudo-primaries to their associated replicas. Meanwhile, the client can execute another update transaction (Fig. 2.b).

This concept of a hierarchy of database replicas resembles the architecture of OceanStore [Kubiatowicz et al.(2000)], an object storage system targeted at providing global-scale persistent storage for file systems or streaming multimedia applications, among others. OceanStore features a two-tier hierarchy of replicas: when an object is updated, the primary tier performs a Byzantine agreement protocol, whereas the secondary replicas propagate the update among themselves epidemically. The result of the agreement protocol is then multicast down the tree to the rest of secondaries.

If applied to a distributed database, this hierarchical architecture entails the possibility of offering a range of consistency guarantees to applications. In general, variable consistency solutions aim at providing high availability and scalability by distinguishing between transactions that demand strong consistency and those that admit a relaxation in consistency guarantees, and executing the latter in a more efficient manner. For example, consistency rationing [Kraska et al.(2009)] allows the consistency level to be automatically switched between session consistency and serializability depending on the specified policies. It is also worth mentioning RedBlue [Li et al.(2012)] consistency, which defines two types of operations: blue operations, which can be executed optimistically and lazily replicated; and red operations, which demand strong consistency and are serialized with respect to each other.

In our proposal, variable consistency can be provided thanks to the hierarchical structure of data partitions: transactions demanding strong consistency will be directed to the core level of the hierarchy, whereas transactions tolerating a certain staleness in their retrieved data will use the lower levels. Furthermore, versions can be associated with timestamps, so that transactions can execute queries stating the level of freshness of the returned data [Lomet et al.(2012), Cipar et al.(2012)].

Apart from providing variable consistency, the proposed architecture is targeted at ensuring elasticity and high scalability by taking into account workload characteristics to configure not only the hierarchical structure, but also other parameters such as the partitioning scheme and the replication protocol used in the core level of the hierarchies.

Following the cloud philosophy, a "*Consistency as a Service*" (CaaS) model could be defined, where clients specify their staleness limit on a per-transaction basis. For instance, this can be applied to web pages in which some elements are seldom updated and do not have strong consistency guarantees (such as translations of interface messages or some multimedia files), whereas other items have critical consistency requirements (e.g., account balances or the availability of an item to be purchased) and thus their associated queries must return the last updated value.

3.2 System Model

Taking the aforementioned motivation of having a tree of consistent versions for each data partition as a starting point, we propose Epidemia, whose system model (depicted in Fig. 3), can be divided into the following components: i) a set of *client applications* that interact with the system, ii) a *metadata manager* (MM) that holds the system state (which is stored in the metadata repository) and orchestrates the communication with both clients and replicas and iii) a set of *replication clusters* ($\mathcal{RC}$s), each storing one data partition.

The asynchronous communication among the different components is performed via message-exchange. In order to ensure that messages are neither lost nor reordered with respect to the order in which they were sent, we assume FIFO quasi-reliable point-to-point channels [Schiper(2006)]. Furthermore, the propagation of updates among the replicas of the core level of each $\mathcal{RC}$ is done by means of the multicast primitive provided by a Group Communication System (GCS) [Schiper(2006)]. In what follows, the different system components and their interactions are described in more detail.

3.2.1 System clients

Client applications interact with Epidemia by means of a custom client library that acts as a wrapper for the management of connections with both the MM and the replicas that serve the transactions.

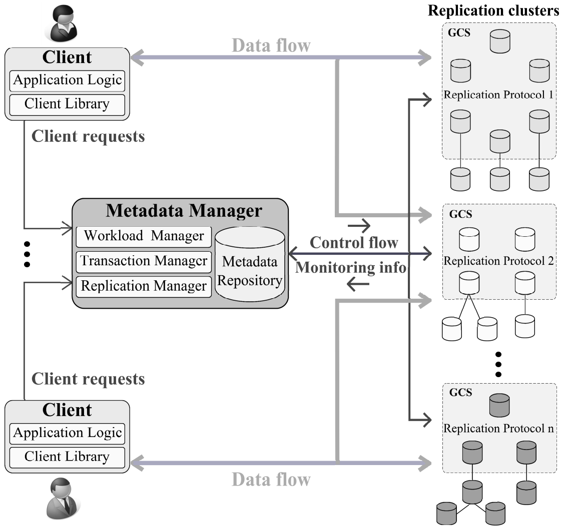


Figure 3: Epidemia's system model.

Each transaction t_{ij} is identified by the identifier of the client c_i that submits t_{ij} and an increasing local sequence number j generated by c_i. Transactions are non-interactive, in the sense that all the operations of each transaction are sent and processed together (as it usually happens in stored procedures [Cheung et al.(2012)]), in order to avoid client stalls. Moreover, transactions in Epidemia can demand a specific freshness degree, where the higher the freshness value, the more recent the versions of the retrieved data items must be. This freshness level can be established in terms of absolute values, version numbers or timestamps.

Each c_i holds a cache that matches the most used transaction types to the replicas that can execute those transactions. This cache has to be refreshed by querying the MM periodically, and also every time a cached replica fails to execute a request.

In order to issue a new transaction t_{ij}, the client first checks whether the cache contains information about the replica (or replicas in the case of a multi-partition transaction) that can handle the corresponding transaction type so that t_{ij} can be directly submitted there. Otherwise, c_{ij} must query the MM to know the replica/replicas that can execute t_{ij}. To this end, t_{ij} will be submitted to one of the MM nodes, which will determine the partition (or partitions in the case of multi-partition transactions) involved in the execution of t_{ij} and select one of the replicas of the $\mathcal{RC}$ storing each of the partitions participating in the transaction. In response to the client request, the MM will send the address of the selected replica/replicas.

Once the replica/replicas that must execute t_{ij} are known, t_{ij} is submitted to the replica r_k selected from each involved partition p_k (where $k \in \{1, \cdots, n\}$, being n the number of partitions involved in t_{ij}). After executing t_{ij}, each r_k will send the result back to the client. The received results must be merged in case $n > 1$ to form the final result.

The client library also deals with failed requests to the MM (by sending a request to another MM node when a timeout expires without a request being answered), as well as with failed requests to replicas (by sending another request to the MM to get the address of another replica that can execute the transaction).

3.2.2 The Metadata Manager

The metadata manager (MM) is in charge of maintaining the metadata repository, which contains:

- A mapping between each data item and the partition it is stored in, which can be established using different granularity levels depending on the partitioning scheme, e.g., by associating each table to a data partition or by horizontally splitting subsets of items belonging to the same table into different partitions.
- A set of available replicas. Replicas can be added (or removed) to Epidemia by modifying this data structure on the fly.
- A mapping between each data partition and the replicas that belong to the corresponding $\mathcal{RC}$. For each replica, it is also necessary to store the level in the hierarchy of versions it is located in.
- Status of each replica, including parameters such as number of transactions per second executed, average number of pending transactions, rate of read-only transactions executed or CPU usage.

Since the information stored in the MM is relatively small and less frequently updated in comparison with the data of the applications stored in the system, it can be distributed among a small set of nodes and synchronized using a Paxos-like protocol [Lamport(1998)] to provide fault tolerance while ensuring data consistency and leveraging system scalability. More specifically, following the approach taken in ElasTraS [Das et al.(2010)], we rely on Zookeeper [ZooKeeper(2014)] (an open source variant of Paxos) to ensure the consistent replication of the MM.

The information of the metadata repository is used and updated by the functional modules included in the MM (see Fig. 3): the workload manager, the transaction manager and the replication manager.

The *workload manager* monitors the set of active replicas in the $\mathcal{RC}$s. Every active replica must periodically send a heartbeat message to the workload manager attaching information regarding its status (such as CPU usage or number of pending transactions). Using this information, the workload manager is able to make decisions on the optimal system configuration. Thus, the workload manager must include a set of rules to dynamically adapt the system configuration to the current workload. Note that the definition and implementation of the decision-making tool that defines and applies these rules is out of the scope of this work [Curino et al.(2010)]. Upon deciding that a certain change in the configuration must be performed, the workload manager will send a message to the involved replicas to reconfigure them in the appropriate manner. The different configuration message types that the workload manager can send to a replica r_n are: i) *join*, to ask r_n to become part of an $\mathcal{RC}$ (r_n will have to obtain all the necessary data items from its parent replica or from one of the replicas that already belong to the core level in case r_n is also assigned to the core level); ii) *leave*, to tell r_n to become offline; ii) *upgrade*, to move r_n to a hierarchy level with a more recent version of the data partition; iv) *downgrade*, to change r_n to a hierarchy level with a staler version of data items; v) *add-child*, to notify a parent replica that, from then on, it will have to asynchronously propagate all the updates it applies to a given replica and vi) *remove-child*, to indicate a parent replica that it will no longer have to propagate its updates to a given child replica.

Furthermore, the workload manager is in charge of handling request messages from clients. Upon receiving a request for a transaction t_{ij}, the workload manager must identify the partitions involved in the execution of t_{ij} and choose one replica from each partition. This election must take into account the requirements of t_{ij} (i.e., the demanded freshness degree and whether it is a read-only or an update transaction) and include load balancing mechanisms to distribute client requests among the replicas.

The workload manager also determines the partitioning scheme and adapts it to the current demands by dynamically splitting and merging partitions. This requires to find the optimal strategy that minimizes multi-partition transactions while making the best possible use of available resources by means of a partitioning algorithm [Curino et al.(2010), Cheung et al.(2012), Pavlo et al.(2012)], which falls out of the scope of this work.

On the other hand, the *transaction manager* deals with the coordination of multi-partition transactions. As mentioned in Section 2.2, there are different ways of coordinating multi-partition transactions to maintain global consistency. In particular, we have devised an approach based on the proposal of [Pandis et al.(2010)]. Upon detecting a dependency between actions of a transaction t_{ij} accessing data of different partitions, the transaction manager associates a rendezvous point (RVP) to t_{ij}. RVPs separate the execution of t_{ij} into different phases. Actions belonging to different phases of t_{ij} cannot be executed concurrently. The transaction manager also designates one replica from each partition involved to act as a coordinator with the replicas of the other partitions to transfer the necessary information and resolve the RVP.

Finally, the *replication manager* is responsible for choosing the most adequate replication protocol for each $\mathcal{RC}$ depending on workload patterns. Note that the characteristics of the involved replication protocols must be taken into account in order not to compromise data consistency. For instance, the transition from a primary-backup replication protocol to an update-everywhere one can be done straightforwardly: the replication manager will send a protocol change request to the primary, which will multicast a FIFO message to the rest of replicas of the core level using the GCS; this message acts a synchronization point, since replicas will change their protocol after having received and applied all previous updates from the primary. In contrast, to change an update-everywhere protocol for a primary-backup one, it is necessary to ensure that updates from other replicas are not inconsistently executed with respect to the sequence of updates executed at the new primary. For example, the assigned primary can multicast a synchronization message using total order so that replicas will start the primary-backup protocol when delivering that message (all updates from other replicas different than the primary delivered after the synchronization message will have to be discarded).

3.2.3 Replication Clusters

Data items are distributed among a set of disjoint partitions, each managed by a different replication cluster ($\mathcal{RC}$). The replicas that form each $\mathcal{RC}$ are organized in a hierarchical way. Each hierarchy level is associated to a freshness degree where the higher the freshness degree, the more recent the versions of stored data items are.

The core level of each hierarchy comprises a set of replicas that propagate updates among themselves by means of a traditional replication protocol (as determined by the replication manager) that makes use of a GCS to handle the messages among replicas and monitor the replicas belonging to the group. On the other hand, the replicas that do not belong to the core level are distributed into several levels forming a tree whose root is the aforementioned core level, where a replica of a given level acts as a backup for a replica of its immediately upper level and may also act as a primary for replicas of its lower level.

With the aim of exploiting the advantages of in-memory approaches (see Section 2.4), such as avoiding disk stalls and using the thread-to-data policy, we assume that every replica keeps all its data in main memory. Consequently, replicas are stateless in the sense that, in case a replica leaves the system and then joins again, it must obtain all the data items of the partition it belongs to. This is done by transferring the whole state from one or more replicas to the new one. In case the requirements of a client application demanded stringent durability guarantees, this solution could be adapted to force replicas to transfer their data to a persistent storage device, either on a regular basis or upon receiving an order from the MM.

When a replica r_l receives a transaction t_{ij} from a client, it first checks that the partition where the client intends to execute t_{ij} is the same as the one managed by the $\mathcal{RC}$ that r_l belongs to.

In case t_{ij} is a read-only transaction, the freshness related to the hierarchy level to which r_l belongs must be capable of fulfilling the freshness limit demanded by t_{ij}. If it is satisfied, r_l executes t_{ij} and sends the result to the client. Here we are assuming that read-only transactions are executed without delay, although their execution could be delayed to meet different consistency constraints and add complexity to the notion of freshness. For instance, in case t_{ij} demanded read-your-writes consistency [Vogels(2009)], before executing t_{ij} the replica would have to apply all update transactions t_{ik} such that $k < j$.

On the other hand, if t_{ij} is an update transaction, it can only be processed if r_l belongs to the core level of the hierarchy and is not a read-only replica. In case this is true, t_{ij} is delegated to the replication protocol. Once t_{ij} is committed, each replica of the core level will be able to asynchronously propagate the changes to its children through the point-to-point connections. Upon receiving an update from its parent, the child replica must apply it and then propagate it to its own children. Transaction updates are propagated in the form of writesets (the set of tuples that are created, modified or deleted by the transaction), instead of sending the whole SQL statement.

Although in this proposal the propagation of updates throughout the hierarchy is done in a structured way (i.e., the parent replica transfers updates to its assigned children), other alternatives based on gossip protocols could be implemented [Hopkinson et al.(2009)]. An unstructured method for propagating updates among hierarchy levels would facilitate the management of parent replica failures, as in the current approach, the MM has to reassign children replicas to a new parent (which might have to transfer lost updates to their new children) upon detecting that a parent replica might have failed.

3.3 Interaction between the Metadata Manager and Replicas

As pointed out before, the workload manager periodically receives heartbeat messages from replicas, which include information such as average number of requests per second received, ratio of read-only transactions, average freshness level demanded by transactions, and so on. These messages allow the MM to make decisions on the best system configuration.

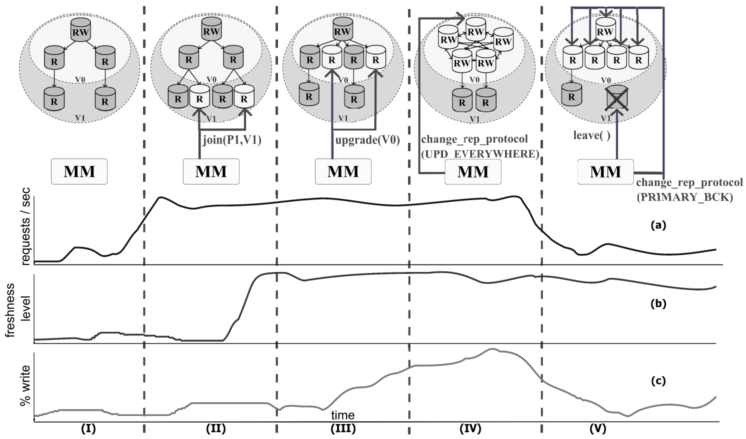


Figure 4: Example of interaction between the metadata manager and system replicas.

Fig. 4 will help us illustrate some of the possible scenarios that the MM may encounter, and the steps it should take upon detecting certain situations in order to adapt the system configuration to current demands. In particular, the three graphics show the evolution across time of (a) number of requests per second submitted to the system, (b) average freshness level demanded by transactions (where the lower the freshness level is, the more tolerant to accept outdated values the transaction is) and (c) rate of transactions including write operations.

The initial configuration (Scenario I) consists of three replicas managed by a primary backup replication protocol in the core hierarchy layer (v0), along with two more replicas in the lower hierarchy layer (v1) that serve as backups for two replicas of the upper layer. Hence, the only replica that can execute update transactions is the primary of the core layer.

An elastic service should dynamically add the necessary resources while interfering with other running processes as little as possible upon detecting a workload increase. In Fig. 4, when the MM detects a relevant increment in the number of requests per second, it adds two more replicas (Scenario II). Since the number of write operations is low and most transactions do not require a high freshness level, the new replicas are incorporated to the lower hierarchy layer. Consequently, the primary replica will not be burdened by the addition of these new nodes.

On the other hand, it should be taken into account that the freshness level demanded by transactions has an influence on the load of the replicas of the core level, because transactions that require a high freshness level cannot be delegated to lower levels of the hierarchy. Thus, upon detecting an increase in the average freshness level, the MM may need to add more resources to the core level to avoid an overload situation, as in the case of Scenario III.

As the performance of replication protocols varies greatly depending on the read/write rate of transactions, the proposed system is also capable of adapting itself to such changes. In Fig. 4, upon detecting that the number of writes increases, the MM changes the replication protocol of the core layer from a primary-backup scheme to an update-everywhere scheme (see Scenario IV), so that all replicas in the core layer can execute update transactions.

Finally, it should be noted that, apart from adapting itself to tolerate increasing workloads, an elastic service should also minimize resources usage. This situation is represented in Scenario V, in which the MM removes one of the replicas upon detecting an important decrease in the number of requests. Moreover, since the rate of transactions that include write operations also decreases, the MM changes the replication protocol to primary-backup again.

4 Correctness Arguments

In the following, we will prove the feasibility of the presented system architecture by stating a sketch of the correctness guarantees it provides, in a similar way as done in [Das et al.(2010)]. The different system components provide certain guarantees that will be taken as a basis point to discuss system correctness in terms of both safety and liveness. We assume that the system does not tolerate Byzantine failures or malicious behavior.

As explained in Section 3.2.2, the MM is assumed to be distributed among a small set of nodes that are synchronized by means of a Paxos-like protocol [Lamport(1998)] to provide fault tolerance while guaranteeing consistency. This ensures that the information stored at the metadata repository of the MM is not lost or left in an inconsistent state even in the presence of arbitrary failures, including network partitions. Thanks to Paxos, the following guarantees are ensured: i) only a single MM node can own a lease (a *znode* in the case of Zookeeper) at any instant of time, which gives permission to modify the data stored in the metadata repository (*MM safety*); and ii) the MM progresses if a majority of nodes are non-faulty and can communicate among themselves (*MM liveness*).

The MM is in charge of determining the set of replicas that constitute the core hierarchy level of each $\mathcal{RC}$. Thus, since at any time there exists a correct MM node that properly controls the behavior of the core replicas (by determining the replication protocol that is executed at the core level, as well as the replicas that must join or leave this hierarchy level), the correctness at the core layer of each $\mathcal{RC}$ depends on the behavior of the replicas that belong to it.

The replication protocol running at the $\mathcal{RC}$ core level is devoted to maintaining the consistency at this level. We can guarantee that the replication protocol ensures data consistency of the corresponding partition as long as it has been shown to be correct [Daudjee & Salem(2006)] and provide one copy (1C) schedules even in the presence of replica failures [Bernstein et al.(1987), Fekete & Ramamritham(2010)]. Therefore, if a transaction only accesses the core nodes of a single partition, it will behave as if it were executed in a traditional replicated database; hence, the consistency criterion fulfilled will correspond with the consistency guarantees ensured by the replication protocol that manages that core level, normally one-copy serializability (1CS) [Bernstein et al.(1987)] or one-copy Snapshot Isolation (1SI) [Lin et al.(2009)] depending on the replication protocol. In case a single partition transaction accesses other levels of the hierarchy apart from the core level, the consistency criterion fulfilled will be 1SI, as update transactions are serially executed at the replicas of the core level whereas read-only transactions can be forwarded to lower hierarchy levels assuming that they might obtain a stale (but consistent) snapshot of the database. However, it is not that cheap to execute 1SI multi-partition transactions without incurring in extra messages [Vo et al.(2010)] that penalize performance. Thus, it can be assumed that no notion of consistency across data is generally provided for multi-partition transactions, but that data versions are obtained from a valid committed snapshot in each partition. The resulting schedule does not satisfy any of the conditions stated in [Lin et al.(2009)]; hence, in the case of multi-partition transactions, we obtain one-copy-multi-version (1MV) schedules.

Apart from this, transactions executed at the core replicas must eventually get propagated to the rest of replicas inside their associated $\mathcal{RC}$ no matter how many replica failures and network partitions occur, so as to ensure global correctness. Since a replica belonging to a hierarchy layer different than the core layer of an $\mathcal{RC}$ receives its updates from a replica of the upper layer via a FIFO quasi-reliable point-to-point channel, updates are propagated from one hierarchy level to the following, and are received (and therefore executed) in order. This corresponds to the notion of eventual consistency [Vogels(2009), Fekete & Ramamritham(2010)]; i.e., there is some time point when if update transactions stop then all replicas will converge to the same state. In case the parent fails, this situation will be eventually detected by the MM, which will choose a new replica that will send pending updates to the children replicas. Indeed, this can be understood as a definition of a global liveness property, as the system ensures that in-background update propagation is done correctly.

5 Evaluation

This section presents the performance evaluation of Epidemia using a prototype implementation that serves as a proof of concept to empirically validate the feasibility of our approach.

5.1 Implementation Details

In order to empirically measure the performance of the proposed system architecture, we have built a prototype (using Java 1.6) that covers the basic functionality of all Epidemia's components.

Instead of developing a distributed implementation of the MM to provide fault tolerance and scalability, we have developed a centralized component for the sake of simplicity. The implemented version of the MM maintains the metadata repository in main memory and builds the replica hierarchies for the partitions as indicated in the configuration at startup time. This configuration is maintained throughout the execution of each experiment, in order to properly evaluate the differences among different scenarios.

Upon receiving a request from a client, the MM examines the partitions involved in the requested transaction (which will be those containing any items accessed or modified by the transaction) and selects one replica for each participating partition. This selection is randomly performed among the replicas of the $\mathcal{RC}$, taking into account that update transactions must be directed to replicas of the core node that are able to execute update transactions; whereas read-only transactions specify a required freshness level, which is simply a number associated to the maximum hierarchy level that can manage the transaction.

For the following experiments, we have developed two different replication protocols: a *primary-backup* protocol, in which there is one primary replica (deterministically selected among the replicas currently participating in the protocol) that executes update transactions whereas the rest of them execute read-only transactions and receive updates from the primary via FIFO uniform reliable multicast; and an *update-everywhere* protocol, which is derived from the former considering that all replicas act as primaries and propagate updates among themselves using total order uniform reliable multicast [Schiper(2006)] to guarantee that all replicas deliver messages in the same order irrespective of the replica that sent them.

5.2 Experimental Settings

Our testing configuration consists of eight computers, each equipped with an Intel Core 2 Duo processor at 2.13 GHz, 2 GB of RAM and a 250 GB hard disk. All machines run the Linux distribution OpenSuse v11.2 (kernel version 2.6.22.31.8-01), with a Java Virtual Machine 1.6 executing the application code. Two additional computers with the same configuration are used for running the clients and the MM instance respectively. Each replica holds a local PostgreSQL database (version 8.4.7) [PostgreSQL(2014)], whose configuration options have been tuned so that it behaves as an in-memory only database. Spread 4.0.0 [Stanton(2014)] has been used as GCS, whereas point-to-point communications have been implemented using TCP channels.

The experiments herein presented have been executed using *OLTPBenchmark* [Difallah et al.(2013)], a multi-threaded load generator that implements a series of standard OLTP/Web benchmarks. In particular, we have selected the *OLTPBenchmark* implementation of YCSB (Yahoo! Cloud Serving Benchmark) [Cooper et al.(2010)], a collection of micro-benchmarks designed to represent data management applications that require high scalability. We have chosen YCSB as the benchmark for this series of experiments mainly because its data schema allows a very straightforward partitioning scheme by horizontally splitting the database into subsets of data records. In the YCSB implementation of *OLTPBenchmark*, there exists one table of records with one numeric key and ten text fields. The available set of transactions that can be executed against this table are: *read*, which retrieves the record that matches the specified key; *insert*, which inserts a new record; *update*, which updates all the fields of one record with the exception of its key; *delete*, which deletes one record; and *scan*, which reads the set of records whose keys belong to a given interval. The database used for the experiments contains a total of 1 million 1KB records for a total size of 1GB.

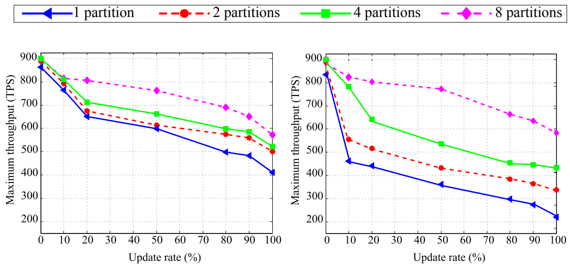
5.3 Experiments

The remainder of this section details the experiments performed using the proposed prototype of Epidemia. In particular, we have studied the influence of different configuration settings (such as the data partitioning scheme or the replication protocols) on the maximum throughput that the system can reach.

5.3.1 Influence of the Partitioning Scheme on System's Throughput

In the following experiments, we have assessed the influence of the number of data partitions on system's throughput depending on workload characteristics and the replication protocol used. We have used 100 clients submitting a total workload ranging from 100 to 1000 TPS (transactions per second). Four different data partitioning schemes have been tested using 8 replicas: i) 1 partition comprising all data items (the 8 replicas belong to the partition); ii) 2 partitions, each storing 500K records (4 replicas per partition); iii) 4 partitions, each storing 250K records (2 replicas per partition) and iv) 8 partitions, each storing 125K records (1 replica for each partition). In these experiments all replicas take part in the replication protocol, as we are considering only one hierarchy level.

Fig. 5 shows the maximum throughput obtained for the four partitioning configurations mentioned, using (a) update-everywhere and (b) primary-backup



(a) Update-everywhere. (b) Primary-backup.

Figure 5: Maximum throughput depending on the rate of update vs. read transactions.

as the replication protocols for managing system replicas. We have used two of the transaction types provided by YCSB: *read* and *update*, which are always single-partition transactions, as each transaction accesses one record. Accessed records are selected according to a Zipfian distribution. The graphics in Fig. 5 show the influence of the proportion of read/update transactions (ranging from 0% to 100% update transactions) on the system's throughput.

Fig. 5 shows that, the higher the rate of update transactions, the lower the maximum throughput that can be obtained due to the overhead imposed by update propagation. In addition, both replication protocols have almost the same throughput when there is a low rate of updates, as read transactions are handled in the same way. In contrast, primary-backup replication is more costly if there is a high rate of updates, since the primary acts as a bottleneck. We shall remark that as uniform delivery is responsible for the most part of multicast latency, the cost of update multicasts is the same in primary-backup replication, where only FIFO order is needed, and update-everywhere replication, which requires total order. In fact, Spread uses the same level of service for providing uniform reliable multicast, regardless of the ordering guarantees [Stanton(2014)].

As for the influence of the number of partitions on system's throughput, the results of Fig. 5 verify that, in the case of single-partition transactions, the more data partitions the database is divided into, the more efficient the system is. This is due to the fact that the number of replicas that have to propagate their changes among themselves is inversely proportional to the number of partitions in these experiments, so the cost of propagating changes is lower when we have more partitions in the system. This difference between the throughput of different partitioning schemes is more noticeable in primary-backup replication. In this case, there is one primary replica managing each partition. Therefore, in a configuration with one data partition there is only one primary replica that becomes saturated easily, as it is the only one that can execute update transactions;

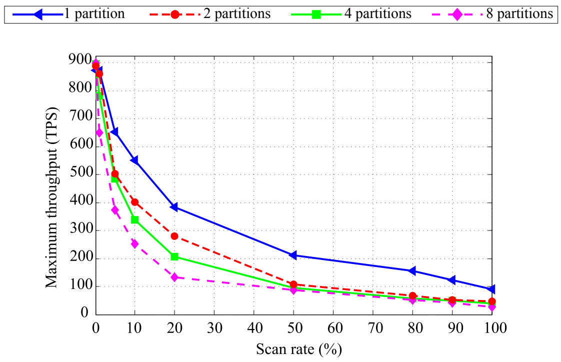


Figure 6: Maximum throughput depending on the rate of scan vs. read transactions.

in contrast, if there is more than one partition the saturation point raises up, as there are more replicas able to handle update transactions for their respective partitions.

The aforementioned results may lead to the wrong conclusion that partitioning the data scheme as much as possible may provide the best performance possible. Actually, we have to bear in mind that the partitioning scheme must also minimize multi-partition transactions, as they are more costly than single-partition transactions. We have verified this statement by repeating the same experiments as before, but in this case using two different types of read-only transactions from the YCSB: *read*, which is always single-partition; and *scan*, which has been slightly modified in order to force it to access several non-consecutive short ranges belonging to different parts of the database within the same transaction.

Fig. 6 shows the maximum throughput obtained depending on the proportion of read/scan transactions (ranging from 0% to 100% scan transactions), using update-everywhere as the replication protocol (as all transactions are read-only, the replication protocol used has no influence on the results). This figure clearly shows that when there is a high proportion of scan transactions, the throughput is lower in cases where the database is partitioned, because every scan transaction accesses one replica from each partition.

In conclusion, partitioning the database scheme requires finding the appropriate trade-off to maximize the throughput of single-partition transactions using the available resources while minimizing multi-partition transactions.

5.3.2 Using Several Hierarchy Levels

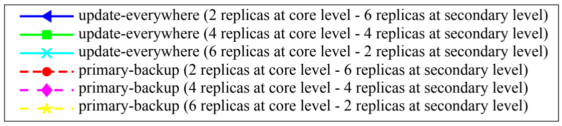
In order to test whether Epidemia can improve the performance of traditional replicated databases by providing additional backup replicas that are asynchronously updated to serve transactions that tolerate a certain degree of staleness, we have configured the system with several hierarchy levels of replicas and have performed a series of experiments using two YCSB workload types: *workload A*, which models an update-heavy workload and is formed by 50% of read transactions and 50% of update transactions; and *workload B*, which models a read-heavy workload and is formed by 95% of read transactions and 5% of update transactions. In both configurations, records are selected according to a Zipfian distribution.

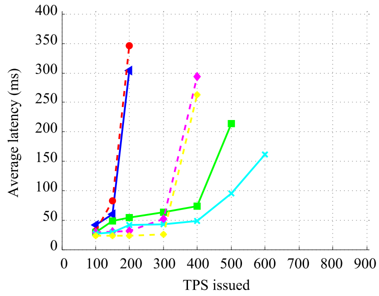
Fig. 7 shows the average response time (in milliseconds) depending on the number of TPS issued to the system for different scenarios using workloads A and B from the YCSB, which have been generated using 100 clients. In this case, we have used one single data partition managed by 8 replicas storing all data items. We have tested different arrangements of the hierarchy: i) 2 replicas in the core layer and 6 backup replicas in the secondary layer, ii) 4 replicas in the core layer and 4 backup replicas in the secondary layer, and iii) 6 replicas in the core layer and 2 backup replicas in the secondary layer. In these experiments, we have set a predefined freshness level to read transactions, which determines whether the transaction accepts or not stale versions of data items. In particular, we have varied the ratio of read transactions that accept old values, setting this value to 25% for experiments a) and b) of Fig. 7, 50% for experiments c) and d), and to 95% for experiments e) and f) of the same figure.

In Fig. 7, the average response time remains stable as the number of transactions issued is increased, until a saturation point (that depends on the system settings and the workload characteristics) is reached. At this point, the system is unable to process all incoming requests, so the latency increases dramatically.

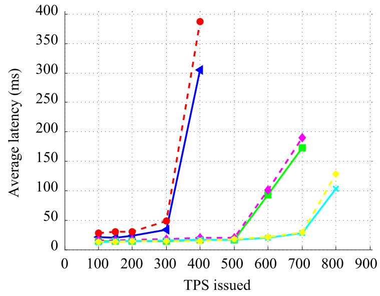
In the experiments regarding workload A, the system saturates earlier when using primary-backup replication. This is due to the fact that, although the propagation of updates from the primary to the backup replicas should be cheaper than in the case of update-everywhere replication (because primary-backup does not require total order), Spread actually uses the same mechanism regardless of the ordering guarantees [Stanton(2014)]. As for the experiments for workload B, there are no relevant differences between using update-everywhere or primary-backup replication for the core layer because 95% of transactions are read-only and both protocols process them identically.

When most transactions demand a high freshness level (i.e., they must be executed at the replicas of the core layer), the most efficient configurations are those in which most replicas are located in the core layer. In this case, having several hierarchy layers does not entail a relevant improvement on system's

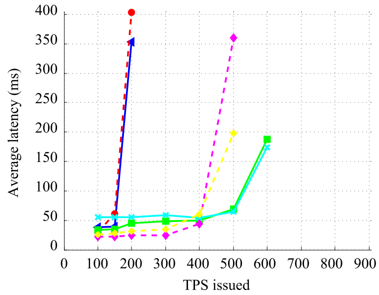




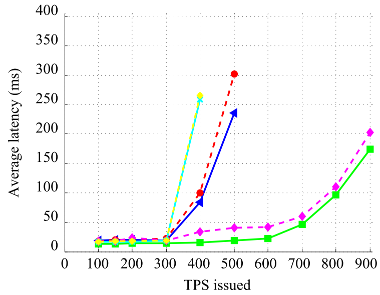
(a) Workload A - 25% read transactions accept old values.



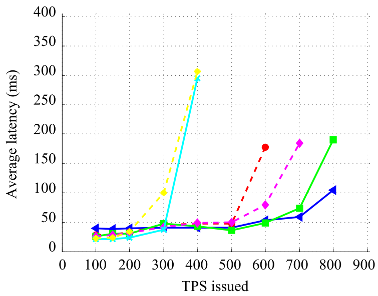
(b) Workload B - 25% read transactions accept old values.



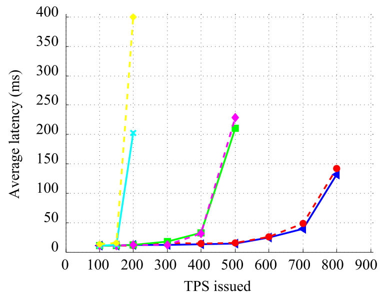
(c) Workload A - 50% read transactions accept old values.



(d) Workload B - 50% read transactions accept old values.



(e) Workload A - 95% read transactions accept old values.



(f) Workload B - 95% read transactions accept old values.

Figure 7: Average latency depending on the TPS issued for the YCSB workloads A and B with one data partition of 8 replicas.

performance (see Fig. 7.a and Fig. 7.b). Note that the configurations in which there are only two replicas at the core layer are especially inefficient when using workload A with 25% of transactions accepting old values, even using update-everywhere replication, since those two replicas have to execute 50% of incoming transactions (which are update transactions), as well as most read-only transactions and therefore they saturate with a low rate of issued TPS.

On the contrary, when most read transactions accept stale values, the more replicas are located in the secondary layer, the better the system performs. For instance, in Fig. 7.e and Fig. 7.f, the configuration with 2 replicas in the core layer using update-everywhere replication and 6 replicas in the secondary layer is the one that outperforms all the others. In the case of Fig. 7.c and Fig. 7.d, in which 50% of read transactions tolerate old values, the system performs best when replicas are equally distributed between the core layer and the secondary layer.

Summing up, we have confirmed that the existence of several hierarchy levels can contribute to increase system's overall throughput by mitigating the load that replicas participating in the replication protocol are subjected to. Of course, achieving an optimal performance would require an in-depth study of the system's behavior under different scenarios, so as to provide the metadata manager with the needed knowledge base to take the adequate decisions to adapt configuration parameters to the workload.

6 Conclusions

We have presented Epidemia, a distributed storage architecture that combines a cloud inspired scheme with traditional database replication concepts to provide a highly scalable and available service with transactional support, thanks to a new replication technique based on epidemic updates which is able to provide different consistency levels according to the demands of each client application. Following the cloud philosophy, the proposed system also features an elastic management of resources, intended to scale out to meet client demands even in the event of load bursts while minimizing resources usage.

The experiments conducted with the developed prototype have verified that one of the key challenges when configuring a partitioned database is finding an appropriate partitioning strategy so that data partitions are large enough to provide fault tolerance and maximize the proportion of single-partition transactions, while small and numerous enough to reach high scalability levels. Moreover, we have shown that the existence of a hierarchy of backups that are asynchronously updated is able to alleviate the scalability limitations of traditional replicated databases by directing transactions that tolerate a certain staleness in the versions of retrieved data items to these backups.

The developed prototype provides a sound foundation for future extensions, such as a fully operational metadata manager including decision-making algorithms to dynamically configure the system by splitting or merging data partitions and upgrading or downgrading replicas along hierarchies. Apart from this, replicas could be provided with live migration mechanisms [Elmore et al.(2011), Das et al.(2011)] to transfer the necessary information to replicas that join a replication cluster or are upgraded in a hierarchy. An extended implementation could also be used for exploring complex formulations for data freshness, such as associating transactions to client sessions to ensure read-your-writes consistency [Vogels(2009)].

From the business model viewpoint, it might be possible to exploit the proposed system architecture to offer a new cloud service called *Consistency as a Service* (CaaS) that could complement the functionalities of existing cloud services.

References

[Aguilera et al.(2009)] Aguilera, M. K., Merchant, A., Shah, M. A., Veitch, A. C., & Karamanolis, C. T. (2009). Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, **27**(3).

[Armendáriz-Iñigo et al.(2007)] Armendáriz-Iñigo, J. E., Juárez-Rodríguez, J. R., de Mendívil, J. R. G., Decker, H., & Muñoz-Escoí, F. D. (2007). *k*-bound GSI: a flexible database replication protocol. In Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, editors, *SAC*, pages 556–560. ACM.

[Baker et al.(2011)] Baker, J., Bond, C., Corbett, J. C., Furman, J. J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., & Yushprakh, V. (2011). Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234.

[Bernstein et al.(1987)] Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[Brantner et al.(2008)] Brantner, M., Florescu, D., Graf, D. A., Kossmann, D., & Kraska, T. (2008). Building a database on S3. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 251–264. ACM.

[Brewer(2012)] Brewer, E. (2012). Cap twelve years later: How the "rules" have changed. *Computer*, **45**(2), 23–29.

[Chang et al.(2008)] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, **26**(2).

[Cheung et al.(2012)] Cheung, A., Arden, O., Madden, S., & Myers, A. C. (2012). Automatic partitioning of database applications. *PVLDB*, **5**(11), 1471–1482.

[Cipar et al.(2012)] Cipar, J., Ganger, G. R., Keeton, K., III, C. B. M., Soules, C. A. N., & Veitch, A. C. (2012). Lazybase: trading freshness for performance in a scalable database. In *EuroSys*, pages 169–182.

[Cooper et al.(2010)] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, *SoCC*, pages 143–154. ACM.

[Curino et al.(2010)] Curino, C., Zhang, Y., Jones, E. P. C., & Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, **3**(1), 48–57.

[Curino et al.(2011a)] Curino, C., Jones, E. P. C., Popa, R. A., Malviya, N., 0002, E. W., Madden, S., Balakrishnan, H., & Zeldovich, N. (2011a). Relational Cloud: a database service for the cloud. In *CIDR*, pages 235–240. www.crdrdb.org.

[Curino et al.(2011b)] Curino, C., Jones, E. P. C., Madden, S., & Balakrishnan, H. (2011b). Workload-aware database monitoring and consolidation. In [Sellis et al.(2011)], pages 313–324.

[Das et al.(2009)] Das, S., Agrawal, D., & Abbadi, A. E. (2009). ElasTraS: an elastic transactional data store in the cloud. In *USENIX Workshop on Hot Topics in Cloud Computing*, pages 7–7, Berkeley, CA, USA. USENIX Association.

[Das et al.(2010)] Das, S., Agarwal, S., Agrawal, D., & Abbadi, A. E. (2010). ElasTraS: An elastic, scalable, and self managing transactional database for the cloud. Technical report, CS, UCSB.

[Das et al.(2011)] Das, S., Nishimura, S., Agrawal, D., & Abbadi, A. E. (2011). Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, **4**(8), 494–505.

[Daudjee & Salem(2006)] Daudjee, K. & Salem, K. (2006). Lazy database replication with snapshot isolation. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *VLDB*, pages 715–726. ACM.

[DeCandia et al.(2007)] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 205–220. ACM.

[Difallah et al.(2013)] Difallah, D. E., Pavlo, A., Curino, C., & Cudré-Mauroux, P. (2013). Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, **7**(4), 277–288.

[Elmore et al.(2011)] Elmore, A. J., Das, S., Agrawal, D., & Abbadi, A. E. (2011). Zephyr: live migration in shared nothing databases for elastic cloud platforms. In [Sellis et al.(2011)], pages 301–312.

[Fekete & Ramamritham(2010)] Fekete, A. D. & Ramamritham, K. (2010). Consistency models for replicated data. In B. Charron-Bost, F. Pedone, and A. Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 1–17. Springer.

[Gray et al.(1996)] Gray, J., Helland, P., O'Neil, P. E., & Shasha, D. (1996). The dangers of replication and a solution. In H. V. Jagadish and I. S. Mumick, editors, *SIGMOD Conference*, pages 173–182. ACM Press.

[HDFS(2014)] HDFS (2014). The Apache Software Foundation, Welcome to Hadoop distributed file system. `http://hadoop.apache.org/hdfs/`.

[Hopkinson et al.(2009)] Hopkinson, K. M., Jenkins, K., Birman, K. P., Thorp, J. S., Toussaint, G., & Parashar, M. (2009). Adaptive gravitational gossip: A gossip-based communication protocol with user-selectable rates. *IEEE Trans. Parallel Distrib. Syst.*, **20**(12), 1830–1843.

[Jones et al.(2010)] Jones, E. P. C., Abadi, D. J., & Madden, S. (2010). Low overhead concurrency control for partitioned main memory databases. In *SIGMOD Conference*, pages 603–614.

[Kraska et al.(2009)] Kraska, T., Hentschel, M., Alonso, G., & Kossmann, D. (2009). Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, **2**(1), 253–264.

[Kubiatowicz et al.(2000)] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S. E., Eaton, P. R., Geels, D., Gummadi, R., Rhea, S. C., Weatherspoon, H., Weimer, W., Wells, C., & Zhao, B. Y. (2000). Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, pages 190–201.

[Lakshman & Malik(2010)] Lakshman, A. & Malik, P. (2010). Cassandra: a decentralized structured storage system. *Operating Systems Review*, **44**(2), 35–40.

[Lamport(1998)] Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, **16**, 133–169.

[Li et al.(2012)] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., & Rodrigues, R. (2012). Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA. USENIX Association.

[Lin et al.(2009)] Lin, Y., Kemme, B., Jiménez-Peris, R., Patiño-Martínez, M., & Armendáriz-Iñigo, J. E. (2009). Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, **34**(2).

[Lomet et al.(2012)] Lomet, D. B., Fekete, A., Wang, R., & Ward, P. (2012). Multi-version concurrency via timestamp range conflict management. In *ICDE*, pages 714–725.

[Maia et al.(2010)] Maia, F., Armendáriz-Iñigo, J. E., Ruiz-Fuertes, M. I., & Oliveira, R. (2010). Scalable transactions in the cloud: Partitioning revisited. In R. Meersman, T. S. Dillon, and P. Herrero, editors, *OTM Conferences (2)*, volume 6427 of *Lecture Notes in Computer Science*, pages 785–797. Springer.

[Pandis et al.(2010)] Pandis, I., Johnson, R., Hardavellas, N., & Ailamaki, A. (2010). Data-oriented transaction execution. *PVLDB*, **3**(1), 928–939.

[Pandis et al.(2011)] Pandis, I., Tözün, P., Johnson, R., & Ailamaki, A. (2011). Plp: Page latch-free shared-everything oltp. *PVLDB*, **4**(10), 610–621.

[Pavlo et al.(2012)] Pavlo, A., Curino, C., & Zdonik, S. B. (2012). Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD Conference*, pages 61–72.

[PostgreSQL(2014)] PostgreSQL (2014). PostgreSQL 8.4 documentation. `http://www.postgresql.org`.

[Schiper(2006)] Schiper, A. (2006). Dynamic group communication. *Distributed Computing*, **18**(5), 359–374.

[Schroeder & Gibson(2007)] Schroeder, B. & Gibson, G. A. (2007). Understanding disk failure rates: What does an mttf of 1, 000, 000 hours mean to you? *TOS*, **3**(3).

[Sellis et al.(2011)] Sellis, T. K., Miller, R. J., Kementsietsidis, A., & Velegrakis, Y., editors (2011). *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. ACM.

[Shorfuzzaman et al.(2012)] Shorfuzzaman, M., Graham, P., & Eskicioglu, M. R. (2012). Allocating replicas in large-scale data grids using a qos-aware distributed technique with workload constraints. *IJGUC*, **3**(2/3), 157–174.

[Sivasubramanian(2012)] Sivasubramanian, S. (2012). Amazon dynamodb: a seamlessly scalable non-relational database service. In *SIGMOD Conference*, pages 729–730.

[Stanton(2014)] Stanton, J. (2014). The Spread toolkit. `http://www.spread.org`.

[Stonebraker et al.(2007)] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., & Helland, P. (2007). The end of an architectural era (it's time for a complete rewrite). In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors, *VLDB*, pages 1150–1160. ACM.

[Vo et al.(2010)] Vo, H. T., Chen, C., & Ooi, B. C. (2010). Towards elastic transactional cloud storage with range query support. *PVLDB*, **3**(1), 506–517.

[Vogels(2009)] Vogels, W. (2009). Eventually consistent. *Commun. ACM*, **52**(1), 40–44.

[Wiesmann & Schiper(2005)] Wiesmann, M. & Schiper, A. (2005). Comparison of database replication techniques based on total order broadcast. *IEEE TKDE*, **17**(4), 551–566.

[ZooKeeper(2014)] ZooKeeper (2014). A high-performance coordination service for distributed application. `http://hadoop.apache.org/zookeeper/`.