Universität
Konstanz

Efficient and failure-aware replication of an XML database

Master thesis presented by

Dirk Kirsten
(01/775833)

Submitted to the Department of Computer and Information Science at the University of Konstanz

1st Referee	Prof. Dr. Marc Scholl
2nd Referee	Prof. Dr. Marcel Waldvogel
Mentor	Dr. Christian Grün

September 2014

ABSTRACT

ENGLISH

BaseX is a native XML database with a single point of failure. If the database server fails, the data is unreachable. Therefore, we introduce the concept of replication to BaseX to achieve high availability. We explore several update management strategies and introduce an update management suitable for the specific requirements of BaseX. As a primary is needed as updating location within our replication architecture, we present several leader election algorithms and show and implement a modified version thereof to fit our requirements. To detect failures, a probabilistic accrual failure detector is used, which is able to detect failures fast and reliably. The thesis shows a favorable performance of the systems in terms of replication and transaction latency of the replication system. Hence, the used replication system high availability with a minimal performance impact in terms of transaction runtimes.

DEUTSCH

BaseX ist eine native XML Datenbank, welche momentan über einen Single Point of Failure verfügt. Wenn ein einzelner Datenbankserver ausfällt, sind alle Daten nicht verfügbar. Daher wird BaseX in dieser Arbeit derart erweitert, indem wir das Konzept der Replikation einführen und somit Hochverfügbarkeit erreichen. Wir präsentieren eine Übersicht über diverse Update-Management-Strategien und führen eine geeignete Strategie für BaseX ein, um die angestrebten Designziele zu erreichen. Wir nutzen einen probabilistischen, anwachsenden Fehlerdetektor um Fehler im Netzwerk oder für einzelne Datenbankinstanzen schnell und zuverlässig erkennen zu können. Da alle Updates in einer als Primary designierten Instanz durchgeführt werden, betrachten wir verschiedene Leader-Election-Algorithmen und präsentieren einen Algorithmus, welcher schnell und zuverlässig einen neuen Primary wählt. Die Arbeit zeigt, dass das Replikationssystems in Bezug auf Replikations- und Transaktionslatenz eine gute Leistung bietet und dass somit die präsentierte Architektur eine hohe Verfügbarkeit mit einem minimalem Leistungsdefizit kombiniert.

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

The number of problems that are not well-suited for relational database technologies has been increasing in the last couple of years. While relational databases were almost exclusively the choice for an application ten years ago, ever-increasing data volumes and altered application requirements have spawned a number of new database technologies. Terms such as Big Data, NoSQL or semi-structured data are ubiquitous in the database world and cater to a broad amount of solutions. While the terms do not have much in common and are often not even properly defined, they often share one similarity: The need for a different storage concept from the traditional relational algebra. This has led to an increased usage and market share for non-relational database systems. All these non-relational databases can be labeled as NoSQL databases, incorporating diverse concepts like column stores, graph databases, key-value stores or document-centric databases. Many of these concepts have in common that the strict service terms of many relational database management systems (RDBMS), being Atomicity, Consistency, Isolation and Durability (ACID), are relaxed to less strict standards to achieve other design goals such as an increased availability or high-performance.

NoSQL databases are beneficial for many use-cases in which a problem cannot easily and efficiently be put into the strict structure of relational systems. However, many RDBMS excel at aspects often labeled *enterprise-ready*, with high levels of provided services. While there is a trend in relaxing ACID constraints and the business need for modeling data in a less strict manner, the strong service level and strict guarantees of relational systems are important for many use cases. Relational database managements systems have been present in the IT industry since the 70s and have been extremely well researched and understood. To apply advanced technologies like failover management, workload distribution, security, scalability and replication to non-relational database systems in an efficient manner remains a major challenge.

BaseX[1] is such a non-relational database, being a high-performance and flexible native XML database system and query processor. XML itself is a W3C standard for document markup and is a data storage and data exchange format with a hierarchical structure.

With increasingly important service requirements with minimal service outages came a number of challenges for native XML database systems, which are already known in a relational context. Downtime of a database system for an application can be very costly and as many applications are run on commodity hardware, failure is not only a theoretical option, but inevitable. Replication, i.e. storing data multiple times on different locations is an important feature for a database when used for high-available data. It enables the database to be still available, although one or multiple data storage locations experience failures. It is even more important, considering that by using BaseX as XQuery processor it cannot only be used as a database, but as a whole technology stack including the application layer logic, i.e. a potential outage would lead to a complete application downtime.

Hence, it is not enough anymore for a database system to support an excellent read/write performance. Users expect a number of features within a full-fledged database system and by including more and more enterprise-level features within the database, BaseX can greatly expand the number of potential use cases.

1.2 CONTRIBUTION

In this thesis we will introduce the concept of replication to BaseX. Multiple instances of BaseX can be joined together to form a replica set and thereby provide replication services. We denote BaseX instances within a replica set as *members*. At any point in time at most one member of a replica set serves as a *primary* with all the remaining members being *Secondaries*. The primary will be automatically selected from the replica set and is the only member than can accept updating queries. The primary is also responsible for coordinating the replica set and allowing new members to join or existing members to leave the set. In case of a failure, a new primary will be automatically elected and serve as the new coordinator to achieve high availability and enable an automatic failover mechanism. We will detect

1 http://www.basex.org, *A light-weight and high-performance XML database engine and XPath/XQuery 3.0 processor*

failures by sampling heartbeats and predict a failure using an accrual failure detector.

The replication set not only increases availability, but also allows to distribute read-only queries to all members within the replicate set and thus load-balance the query workload to several members. Based on the read-/write ratio this can lead to a severe performance increase.

While the thesis is strongly coupled with the implementation for BaseX, there are also several general contributions. We develop a new election algorithm for systems with unreliable communication channels and with only a subset of the members being eligible. Also, the implementation for BaseX holds several lessons for other database systems, especially document-centric ones.

1.3 OVERVIEW

This thesis is organized as follows: Within Chapter 2 we will introduce the concept of replication in detail and the different aspects of it in a database context. After discussion of the existing research in Section 2.2 we will present the novel design and architecture of the replication system within BaseX in Section 2.3. To achieve high availability, we must be able to detect failures and react on them, what we will discuss in Section 2.4. In Section 2.5 we will investigate an algorithm for automatic leader election to determine a writable location within the replica set. Therefore, a review of existing leader algorithms is presented and a modified approach for our specific requirements is shown. In Chapter 3 we benchmark the proposed replication system and show that the system behaves well for the design goals we aimed for. Based on the presented arguments and performance we will draw a conclusion in Chapter 4.

2

REPLICATION

2.1 PRELIMINARIES

Replication in a database context is the creation and maintenance of multiple copies of a *data item*, whereby a data item is simply a single piece of information. Using replication, each data item is stored not just as one *physical data item*, but stored on multiple locations as several data items. As these data items should be consistent, they form one *logical data item*, i.e. the user is unable to distinguish which physical data item he accesses. Ideally, it should appear to the user of the database system as if only a single copy of one particular data item exists. Hence, the user should just interact with the logical data item, without the need to know internal details about the physical storage location of data items. It is therefore the primary responsibility of a replication mechanism to map operations on a logical data item to all physical data items of this data item.

Replication can enhance a database system in various ways and can have multiple design goals. The most important design goals in practice today are high availability, strong fault tolerance and increased performance and a combination thereof.

Increasing availability Availability is the proportion of time for which a service responds within a reasonable time frame. High availability is therefore a required goal for many applications to cater to their users, whereby the level of availability which is required and must be guaranteed by the database system highly depends on the use case. Whereas even a short downtime in many industries cannot be tolerated[1], others might tolerate

1 The survey "Understanding the Cost of Data Center Downtime" by Emerson Network Power (found at `http://www.emersonnetworkpower-partner.com/ArticleDocuments/SL-24661.pdf.aspx`) shows that the cost of a 90 minute downtime for companies such as banks, telecommunications companies, internet service providers or cloud/co-location facilitates amounts to an average of $505,000

a longer period of outtakes and still consider it a highly available system. Server failures and network partitions are the two most relevant factors in a degrading availability. Without replication the outage of a single server will lead to the complete unavailability of a database system. If instead a data item is stored in up to n locations, the availability is dramatically increased, because only one location has to be available to serve the requested data item. If we consider the probability p of a server failure, the expected availability of an object stored in such a replicated system is $1 - p^n$, i.e. the probability of a failure increases significantly with an increasing number of storage locations.

A special case of high availability is the presence of mobile replicas. With the increased usage of mobile devices like smartphones and tablets with limited network connectivity comes a number of new challenges. Whereas in a traditional server settings we do expect occasional failures, the connectivity for mobile replicas is very limited to begin with. However, it would still be beneficial to have some kind of synchronization, as it could allow to automatically replicate changes made at a mobile device during network unavailability to the main servers and vice versa.

Fault tolerance The availability of data is a first requirement to deliver a correct result to the client. However, this available data can still be not correct, especially it can be out of date. Strict consistency is often dropped to less strict consistency criteria for performance reasons, opening the possibility that a client could read stale data. When an update is marked as complete to the user before it is committed on all storage locations there can be different versions of a data object stored on the distinct storage locations. A fault-tolerant service will always, despite a certain number and type of failures, return the correct and most up-to-date information back to the client. If there are $2n + 1$ servers in the cluster and up to n servers misbehave, the replica set can still be fault-tolerant as the majority of servers are still behaving correctly and can outvote the misbehaving minority of servers.

Improving performance Data-bound operations are often limited by I/O performance. If the same data is present at not only one server, but instead at n servers, the read capacity can theoretically be increased by factor n. However, the server has to make sure it returns the most recent state of a data item. Many replication systems relax this condition to increase the performance of a system and not always return the most up-to-date date item. One example of increasing performance by replicating data is *caching*, e.g., when using DNS to lookup an internet address and to determine the

IP address. As this is fairly static data it can be cached by the browser and by a re-run of the same address the cached date can be instantly accessed. Many database replication systems perform read operations on all storage locations within the systems to fairly distribute the workload and increase throughput.

2.1.1 *Design Goals*

XQuery is often compared to SQL, as both are query languages. This is true in respect to XQuery being the primary and natural query language for XML data, just as SQL is the primary query language for relational data. However, XQuery is even more powerful by providing a full-fledged functional programming language[2], with the latest specification of XQuery 3.0 providing advanced concepts such as native function items and higher order functions. Therefore, XQuery can be used not just as querying language, but also as full application programming stack. With the introduction of RESTXQ in [Ret12] and the support for it within BaseX, it can even serve as a full server-side web application stack.

2.1.2 *Consistency*

Consistency in a database context is the guarantee that any transaction sees the effect of other transactions committed in the past and guarantees that no database constraints are violated. The database will always transition from one valid state to another valid state.

We will now define a consistency model for the further discussion of replication strategies. This is especially important, as consistency is a broad term and it is important for the user to know which level of consistency to expect without bothering with internal protocol implementation details.

Transaction Consistency

For a data item x we can use two distinct *operations* to operate on x: $read(x)$ and $write(x)$. A *transaction* T is a partial order of such read and write operations. Two operations $O_i^{T_k}(x)$ and $O_j^{T_l}(x)$, being a member of transactions T_k and T_L respectively, accessing the same data item are defined as *conflicting operations* iff one of them is a write operation. It follows that two read

2 Although SQL is turing complete, it is mainly a declarative querying language and in practice not used for programming application logic.

operations can never be in conflict with each other. Transactions containing such a conflicting operation are by extension also *conflicting transactions*.

A history H is defined as the partial order of execution over a set of transactions $T = \{T_1, T_2, \dots, T_n\}$ and thereby specifies an interleaved order of operation execution [ÖV11].

Definition 2.1. *A complete history H_T^c is a partial order $\{\Sigma_T, \prec_H\}$ with the following properties:*

1. $\Sigma_T = \cup_{i=1}^{n} \Sigma_i$.
2. $\prec_H \supseteq \cup_{i=1}^{n} \prec_{T_i}$.
3. $\forall O_i^{T_k}(x), O_j^{T_l}(x) \in \Sigma_T$ *with* $O_i^{T_k}(x)$ *and* $O_j^{T_l}(x)$ *being conflicting operations in distinct transactions* T_k *and* T_l *there exists an ordering relation such that either* $O_i^{T_k}(x) \supseteq_H O_j^{T_l}(x)$ *or* $O_j^{T_l}(x) \supseteq_H O_i^{T_k}(x)$.

This basically defines that a complete history has a domain which is the union of the domains of all of its transactions and that the ordering relation of the history is a subset of the ordering relations of all transactions.

A history is *serial* if for each pair $T_i, T_j \in T$ either all operations of T_i execute before T_j or all operations of T_j execute before T_i. It immediately follows that a serial history maintains consistency, as separate transactions are applied in an serial manner and each individual transaction provides atomicity.

Two histories $H_1 = \{\Sigma_T, \prec_{H_1}\}, H_2 = \{\Sigma_T, \prec_{H_2}\}$, i.e. defined over the same set of transactions T, are *conflict equivalent* if for each pair of conflicting operations O_i, O_j for all $O_i \prec_{H_1} O_j$ also $O_i \prec_{H_2} O_j$ and for all $O_i \prec_{H_2} O_j$ also $O_i \prec_{H_1} O_j$ holds.

Definition 2.2. *A history is serializable (SR) if it is conflict equivalent to a serial history.*

Definition 2.2 is the main criteria to guarantee transaction consistency. A database provides strict consistency if the transaction execution history is serializable.

A weaker serialization guarantee is provided by *snapshot isolation*, introduced by Berenson et al. [BBG+95]. It has gained significant traction within the research community and with commercial database vendors. Hereby read operations are never blocked, but might read stale data from a previous snapshot of the database. This clearly violates the serializability constraint, but increases performance characteristics.

BaseX uses a strict two-phase locking (2PL) protocol to ensure strict consistency. 2PL as introduced by Bernstein et al. [BSW79] requires that no transaction is allowed to request a lock after it released any of its locks. This basically divides the protocol in two phases; a growing phase in which all locks are obtained, followed by a shrinking phase in which the locks are released. A simple locking graph is shown at Figure 2.1.

The discussion until now was just focused on non-replicated systems. By introducing replication and the existence of multiple storage locations and local histories the serializability correctness criteria has to be enhanced [BG83].

Definition 2.3. *A replicated system is one-copy serializable (1SR) if the execution history is conflict equivalent to a serial history over non-replicated data items.*

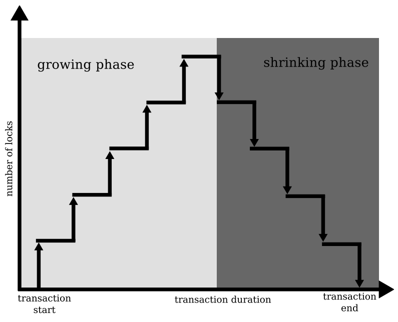


Figure 2.1: Two-phase locking graph.

1SR is the main criteria for strong transactional consistency when using a replicated DBMS. Hence, we refer to a replication algorithm to confirm to strict transactional consistency if it is 1SR.

Similar to strict consistency, snapshot isolation can also be enhanced to be used in a replicated context [LKPJ05]. *Relaxed concurrency serializability* (RC serializability) is the enhanced concept for an even weaker transactional consistency in a replication context [BFG+06].

Mutual Consistency

While transactional consistency requires that the global execution history is serializable, mutual consistency requires replicas to converge to the same value for a data item.

We distinguish two classes of mutual consistency: Strong and weak mutual consistency. Using strong mutual consistency it is required that all replicas hold the same value for a data item after an updating transaction. This is mainly achieved by using a distributed 2PL protocol, thereby updates are propagated and committed to all affected replicas before committing the transaction as a whole.

2.2 UPDATE MANAGEMENT STRATEGIES

The main characteristic of a replication system is how updates are handled. Gray et al. introduced in [GHOS96] how replication protocols can be classified by two orthogonal criteria: The time when an update is replicated to other nodes and thereby affecting the one-copy serializability of a system, mainly divided into *lazy* and *eager* execution and the location of an updated, either at any node being *update anywhere* or only at the node holding the respective data item, being *primary copy*. primary copy can either be that one single master is responsible for all data items or each data item has one distinct master, i.e. there are multiple updating replicas in the network.

EAGER PROPAGATION Using eager replication updates are applied to all replicas of an object with the context of a transaction and before committing the transaction itself. Hence, after a commit all replicas hold the same value for the update data item. Typically Two-Phase Commit (2PC) is used for proper locking, although other approaches are also possible. 2PC provides strict consistency, but suffers from a comparatively slow performance. In the context of BaseX, Erat showed in his bachelor thesis in [Era13] that this holds true also for BaseX as 2PC is the currently used local locking protocol. This effect is even worse when locking is done in a network environment with the increased response times within a network.

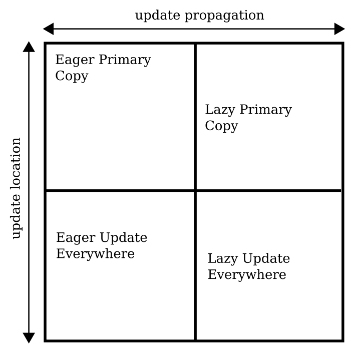


Figure 2.2: Several databases compared to their replication update locations and propagation strategies.

There are three main benefits of eager protocols. First, they typically provide mutual consistency by enforcing SR1 and thereby avoid transaction inconsistencies. Secondly, read operations can be applied on all nodes in the network and it is enforced that only the most up-to-date value of a data item is read. Finally, updates to replicas are applied atomically. Hence, when after recovering from a failure the still provide full ACID capabilities.

The main disadvantage is that all nodes have to locally commit an update before it can be committed to the user. Therefore the response time of a transaction is limited by the slowest replica in the system as it has to wait for all replicas before returning. This also entails that if one replica is unavailable, updating transitions cannot terminate. For this reason, the scalability for eager protocols is rather limited. As Gray showed in [GHOS96] a ten-fold increase in nodes results in a thousand fold increase in deadlocks or reconciliations.

LAZY PROPAGATION Lazy update propagation protocols differ from eager protocols mainly in the aspect that lazy protocols commit before the updates are propagated to all replicas. Whereas eager protocols wait for replicas to update their local data item, lazy protocols do not wait for this commit and instead immediately commit after a transaction was committed at the replica where the transaction is executed.

This has the main advantage that the response time is lower for lazy protocols. It also improves the scalability as replicas are much looser coupled. However, this performance increase comes at the price of the loss of strict consistency. The values of data items at separate replicas can be mutually inconsistent and data items can be out of date if the update propagation is not already applied at all replicas. Hence, a local read operation can read stale data or a transaction may not see its own updates, an effect known as *transaction inversion*.

PRIMARY COPY LOCATION In a *primary copy update* strategy, an updating operation has to be executed at the master site of the data item to be updated. Each data item is assigned to one location and this replica serves as *master copy* for this data item, all other replicas being *slave sites*. A special case of a primary copy is a *single master* replication strategy as it restricts all data items to have the same master copy replica, i.e. all updates are executed at one single location.

The advantage of primary copy algorithms is mainly that update strategies are easy since they only happen at one location. Thus, the need for additional network communication costs and overhead is reduced as there is no need for synchronization between different replicas. One location will always have the most up-to-date value for a data item. The disadvantage of the centralized approach is that the specific centralized nodes present a natural bottlenecks. If only one single location is responsible for a data item, heavy load on this location can impact performance significantly. Also, the presence of a failure is increased, as it also presents a possible single point of failure.

UPDATE ANYWHERE LOCATION Update anywhere location strategies allow an updating operation to be applied at the replica location where the transaction was executed. The local updating operation will then be propagated to all other replicas within the network.

Performance will increase as write operations can be more evenly distributed across all replicas. However, a data item can be updated at separate locations concurrently. Depending on the propagation strategy this might be solvable by a distributed concurrency control algorithm and still provide

1SR, but in other cases it will lead the database system to be non-1SR for the global history.

2.2.1 *Eager Primary Copy*

In a single master eager protocol one replica is responsible for all update operations. A transaction with at least one updating operation $Write(x)$, x being a replicated data item, is directly executed at the single master as shown in the example in Figure 2.3a. Both reads and write on the master are serialized using the Transaction manager of the replica, in most cases using 2PC to provide strict consistency. The master applies a *Write* on its local copy of the data item and propagates the update to all other replicas. Write operations have to be transmitted at the slave replicas in the same order as on the master, e.g., by using timestamps or logical clocks. After all slave replicas replied back to the master that the commit was successful, the updating transaction returns as committed to the user. Read operations can be executed at any replica, as they all hold the most up-to-date value for each data item. However, the executing replica has to require a read lock at the master replica using a centralized concurrency control algorithm. The read can then be executed at the respective replica and after that the master can be notified to release the read lock. An example replication workflow is shown in Figure 2.3a.

This approach has the main drawbacks that the response time depends on the slowest replica in the network and that the single master has to execute all updating operations, possibly resulting in a bottleneck.

One can relax the requirement to store all up-to-date values at a single master. Instead, each data items is associated to one replica which is responsible for all updating operations on this particular data item. As there are now multiple updating replicas available, the write load can be better distributed. Unfortunately, it is more complicated to provide a global serialization order. One possible solution is a primary copy two-phase locking (PC2PL) protocol, whereby the lock manager is distributed over all replicas and each lock manager is responsible for locking the specific data items on the respective replica.

2.2.2 *Eager Update Anywhere*

An eager update anywhere replica control algorithm employs the possibility of executing updates on any replica. Before committing the value, the operation is propagated to all other replicas and commit them on their respective local copy. Each replica is responsible for monitoring all transactions executed at that replica. When the other replicas have all notified the master of their successful update operation, the transaction commits to the user. The difficulty in this approach is the possibility of multiple updating operations on different replicas at the same time. Updating the same data item at two independent sites could lead to inconsistencies. Hence, a distributed concurrency control algorithm has to be used to serialize the execution order of all updating transactions.

As in other eager approaches, read operations can again be safely issued at any replica, as each data item will have the most up-to-date value.

An advantage of this approach is that a client does not have to care to which replica it should send a query. As read and writes can be executed at any replica it can simply be sent to any replica. Hence, transactions can be load balanced to any replica. A disadvantage is the communication overhead of a distributed concurrency control algorithm.

2.2.3 *Lazy Primary Copy*

Within a lazy primary copy replication strategy an updating operation is applied and executed at the master copy and propagates the change to the slave sites as shown in Figure 2.3b. The main difference to eager primary copy algorithms is that in this case, the propagation takes place after the commit has already been finalized on the master copy and the transaction has successfully committed. As a consequence, read operations on data items located at slave sites could read stale data, if a propagation from a master copy could still be pending when the read operation is already executing. One resulting consistency anomaly can be transaction inversion in which a user might not see its preceding write operation when reading data.

The execution order of all updating operations should be one-copy serializable on all replicas. Using a single master this is quite straightforward as a simple timestamp allows for the correct ordering. However, if the master copies are located at different replicas, this becomes more complex. One replica can receive propagated updated data items from multiple locations and determining an equal order on all replicas is complex.

One possible approach is to use vector clocks or some other mechanism guaranteeing total order of operations instead of timestamps. If transactions arrive out of order, the normal solution in a local database system would be to abort the transaction. However, in a lazy protocol the transaction already terminated and an abort is therefore no longer possible. Therefore, an algorithm for update reconciliations to compensate and converge to a consistent state is required.

Write operations will always be applied to the master copy replica location, whereas the location of read depends on the concrete algorithm. If the consistency is related and outdated data can be returned, read operations can be executed at any location. If not, a read operation should be executed by the master copy replica or a replica which is guaranteed to have the most up-to-date value for the required data item.

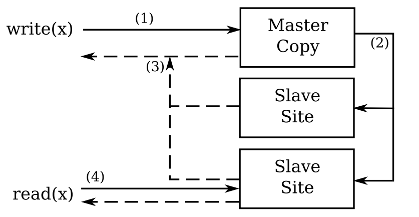
The main advantage is the low response time compared with eager protocols. As a transaction does not have to wait for a slave site to commit as well, the included network latency and response time to not effect the commit of the transaction itself. The main disadvantage is the less strict consistency guaranteed by the algorithm. Depending on the application it might be unacceptable to read outdated data. Many systems require a certain freshness of the data, e.g., in terms of time or number of updates a data item is allowed to be outdated to be still considered fresh.

2.2.4 *Lazy Update Anywhere*

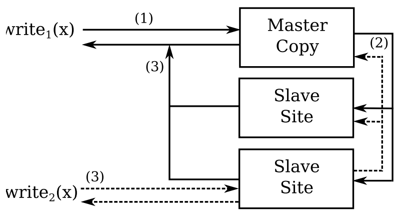
Lazy update anywhere replication algorithms use a less strict version of consistency. Updates occur on all replicas and are propagated to the non-executing replicas after the transaction was already committed at the local copy of a data item. Read and write operations are both executed at a local replica and are later on propagated to all replicas. Complication arises when the propagated updates are processed at the receiving replica. The order of the transaction and hence the serialization cannot be guaranteed as the same data item could have been concurrently updated at different replicas. These conflicting updates have to be reconciled and the result will be incorporated into the order of the transaction execution.

However, the reconciliation is a non-trivial part and is based on heuristics (e.g., timestamps or the priority order of replicas) and heavily dependent on application requirements. However, some updates will be lost, depending on the concrete algorithm.

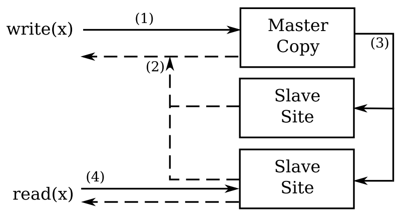
An example workflow of a write transaction using a lazy update anywhere scheme is shown in Figure 2.3d.



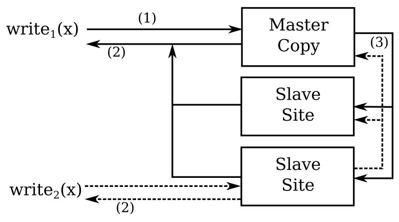
(a) Eager primary copy replication actions. (1) An updating operation is applied at the master copy location. (2) The write is propagated to all slave sites. (3) Each slave commits the write locally and after all sites have committed the transaction is committed. (4) A read operation can be executed at any replica.



(b) Eager update anywhere replication actions. (1) A data item is update by two separate updating transactions. (2) Both executing replicas propagate the update to each replica in the network and use a distributed concurrency control algorithm to establish a consistent serialization history. (3) Each transaction become committed after each site committed the update locally.



(c) Lazy primary copy replication replication actions. (1) An updating operation is applied at the master copy location. (2) The replica applies the update locally and commits the transaction. (3) The update is propagated to all slave sites, where it is locally applied. (4) A read operation can be executed at any replica, but may read stale data.



(d) Lazy update anywhere replication actions. (1) A data item is update by two separate updating transactions. (2) The update is locally applied at each respective replica and committed. (3) Both executing replicas propagate the update to each other replica in the network.

Figure 2.3: Four distinct categories of update management strategies for a replication system, differentiated by the update location and the update propagation.

2.3 REPLICA SET IN BASEX

In [Bre00] Brewer introduced the *CAP theorem*, establishing that a distributed system cannot provide consistency, availability and partition tolerance at the same time. In [Bre12] it was clarified that the properties are continuous and not binary. As we already have shown, there are many different levels of consistency providing miscellaneous guarantees. Also, it is quite clear that availability can range from 0% to 100%.

When designing a replication architecture for BaseX we especially aimed for the following design goals:

- As BaseX is not solely a database but can also serve as an application layer, high availability is highly important. Uptime requirements are critical for many applications and single points of failure are often not tolerable. Although BaseX does have a backup/restore functionality, at the time of writing there was no real reliable solution. Especially for web applications using RESTXQ it is often desirable to eliminate single points of failure and to harden failure tolerance. Therefore, this is our most important design goal.

- As BaseX is optimized for performance in many ways such as query processing as shown by Grün [Grü10] or for updating as shown by Kircher [Kir13]. Therefore, introducing replication should bring a minimal overhead for updating operations and present a primary design goal. It is also of importance to increase the performance in terms of throughput for read operations so large-scale deployments are easily achievable.

- Replication should be transparent to the user, i.e. the user should not be able to notice a difference between communicating with a single server or a replica set of BaseX instances.

- We want to design our replication system in an extendable way, such that further improvements are easily possible. The architecture should allow to include new replication algorithms. It should also allow to extend replication to a true distributed system where workload is distributed to a set of BaseX instances.

Building a replication system for BaseX involves communication through a network. Reliable, high-performance and correct network communication to distribute tasks within a network and avoid race conditions, deadlocks and livelocks is a non-trivial task and as we have only a limited amount of time and resources, we decided to use a framework to ease the implementation difficulties instead of implementing everything from the ground up.

As BaseX is written in Java, the framework must be able to run on the Java Virtual Machine (JVM). We decided to use Akka[3], an actor-based framework to build highly concurrent, distributed and fault tolerant applications. Actors are objects encapsulating a state and behavior. Actors can communicate to each other exclusively by using immutable messages. This is beneficial, as it avoids locking objects and the accompanying locking and possible race conditions. Actors form a hierarchy, each actor has a supervisor which can react to an unexpected behavior or a failure of the actor and take appropriate actions. Also, as actors communicate via messages, it provides an abstraction layer in which the underlying implementation is not relevant for the message transmission, i.e. it does not matter from an implementation perspective whether two actors run in the same JVM or on different JVMs on different machines, as akka will take care of the communication channel. However, it is important to know that akka does provide a strict delivery guarantee. Instead, a message is guaranteed to be at-most-once delivered, i.e. if this message is dropped, the message is not delivered at all without further actions. It is guaranteed that messages are delivered in-order per sender/receiver pair of actors.

Another characteristic of akka is that it is event-driven and non-blocking. Therefore akka is able to highly utilize the hardware without having to wait for blocking operations.

2.3.1 *Architecture*

A replica set in BaseX is a group of at least two *members*. In each replica set exactly one member will act as a *primary*, all other instances will act as a *secondary*. A primary is automatically elected from the set of members. The election process will be discussed in detail in Section 2.5. All update operations are performed solely by the primary. The primary will then propagate all changes to the secondaries, which will then asynchronously apply the operations.

Distributors are the only instances which are able to connect to members of the set and are themselves part of the replica set. This allows load balancing queries and handles member failures in a transparent manner to

3 `http://akka.io`, Akka is an open source Apache 2 licensed application developed by Typesafe Inc. It should be possible to develop a BSD-licensed application such as BaseX using a Apache 2 licensed library, although the legal aspects and ramifications are out of scope for this thesis

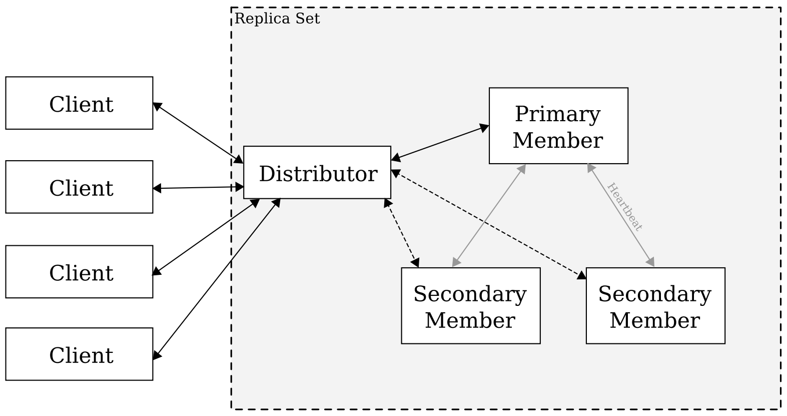


Figure 2.4: Replica set architecture with three members. There is one primary within a replica set with all other members being Secondaries. All members are connected through heartbeats. A distributor is always connected to a primary and if needed by a client connection can also send queries to Secondaries. All clients connect through a distributor with the replica set.

the user. *Clients* can connect to a distributor. Each connection can specify several parameters about how a query should be handled, e.g., whether it should be just sent to the primary or if it could also be executed at a secondary.

An architectural overview for a replica set with three members is given in Figure 2.4. Note that also multiple distributors could be present. As a distributor is a light-weight construct and should fail very rarely, this might often be not necessary, as a failing distributor would require the client to manually connect to another distributor instance.

Member state machine

A replica set member works as a finite state machine with well-defined transitions and states as shown in Figure 2.5. A member always starts in the *Unconnected* state. When the member is started with the intention to span a new replica set, it will transition to the *primary* state to listen for incoming new members. As a requirement for a replica set to be writable is to have at least two members, the replica set will remain in read-only mode. If the member is commanded to connect to an existing replica set, it will transition to the *StartUp* state. During start up, the new member handles the initial connection establishment and joins the replica set as described in more detail in the next chapter.

Depending on the state of the replica set, the member could transition directly to being a *secondary*, if there is a primary present and the set is

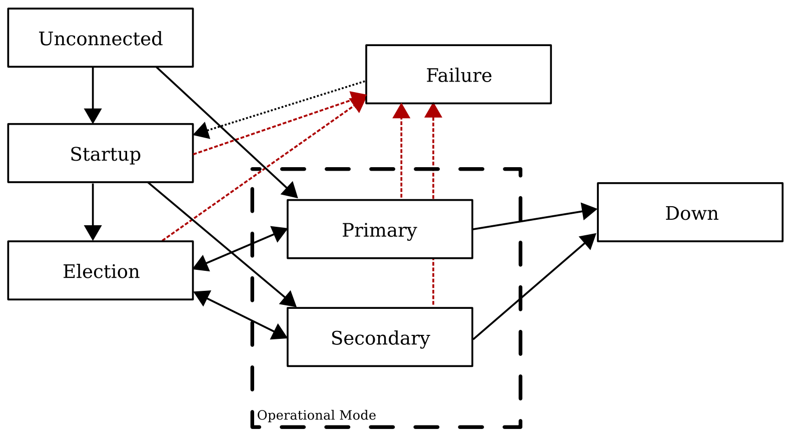


Figure 2.5: Member states with all possible transitions. The direction of an arrow indicates whether a transition in this direction is possible. The black continuous line indicate a normal transition. The red dashed lines represent a failure transition. The black dotted line is a failure recovery.

writable. If any of these conditions are not met, all members in the set will go into the *Election* state and elect a new primary. Thus, one member in the cluster will transition to the primary state, whereas all other members will become Secondaries.

When the replica set is gracefully shut down or a member decides to leave the replica set it will go into the *Down* state.

In any of these states except the unconnected and down states, whereby a member is already disconnected, a member can go into a *Failure* state if an unrecoverable error occurred.

Connection handling

During start-up of a new member it is set by the user whether a new replica set should be created or the member should connect to a new replica set. If the member should join an existing replica set, this connection has to be established.

First, the akka subsystem is started. As Akka guarantees only at-most-once delivery guarantees for messages sent we use an enhanced message channel with acknowledgments and retransmissions. This ensures that a message is actually delivered from one member to another. Using a reliable channel we establish a connection as shown in Figure 2.6 using the following protocol:

1. CONNECTION START The connection is initiated by the new member to a primary and sends a connection start with all member

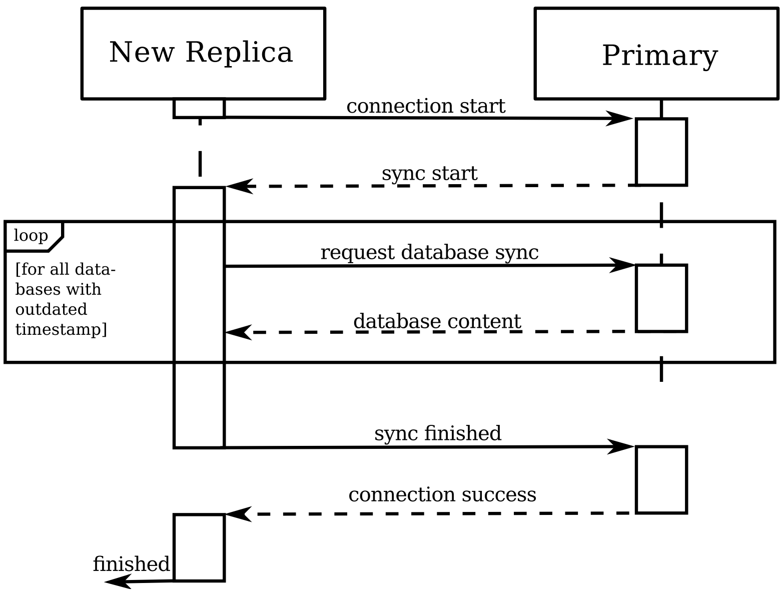


Figure 2.6: Connection establishment message workflow.

specific settings. The settings can be modified by the user within a configuration file. At the moment, the following information is sent:

- Whether the member is voting in an election
- The weight of this member, relevant for a voting within an election

The system can easily be extended to support more settings.

2. SYNC START After receiving the connection start, the primary responds by starting the synchronization process. Within this process, all databases from the primary are bulk-updated to the new member. Therefore, the current last updating timestamp of all databases at the primary is sent to the new member. Also, all database options are included in the message.

3. REQUEST DATABASE SYNC The new member processes the received synchronization start message and applies all database options from the primary at the local database context. After this, the member compares all database timestamps with the ones sent by the primary and sends for each database not having an up-to-date timestamp a *request database sync* message.

4. DATABASE CONTENT Each database request is answered by the primary by streaming the database content to the new member. Therefore, the internal storage files are streamed to the member.

5. SYNC FINISHED Finally, after all databases are updated, the new member signals the primary that the synchronization is finished.

6. CONNECTION SUCCESS The primary acknowledges the connection success and also sent back the complete topological information about the replica set, i.e. about all members within the set. This also includes the information about the new ID for the newly joined member, as an unique and random ID is assigned to a new member by the primary.

Following this exchange, a new member is connected to the replica set as secondary. Notice that no election will be automatically triggered by a new entering member. The replica will from now on receive and accept updating messages sent by the primary.

Using the replica set topology sent by the primary, the new secondary will establish a heartbeat to all other members within the replica set. The details on the used heartbeat algorithm are explained in chapter 2.4.1.

2.3.2 *Client connection*

A client can connect to a replica set by connecting to a distributor. The distributor will first authenticate the client and the client session is set up with possible different connection preferences.

Authentication

In the current client/server architecture a modification of CRAM-MD5 is used to identify clients [Wei10]. It follows the same message flow, but has few modifications in comparison to the published standard [KCK97]. Whereas the standard only considers ASCII characters, the strings in the implementation are UTF-8 encoded. Also, parts of a message are not separated by white spaces as in the original specification, but by a binary protocol. The current authentication message flow is as follows:

1. A client starts the connection and connects to a TCP server socket.

2. The server sends a message with the current unix timestamp to the client.

3. The client responds with a message containing the username, followed by the MD5 hash of a concatenation of the MD5-hashed password and the timestamp.

4. The server looks up the stored MD5 hash for the given user and also computes the MD5 hash of a concatenation of this hashed password and the sent timestamp. If equal, the server responds with success, otherwise it will return a failure.

However, CRAM-MD5 has a number of problems. As the algorithms predates the widespread usage of UTF-8 it was just designed for usage of ASCII, therefore the current implementation in the client/server infrastructure is already not standards-compliant. More importantly, in the latest specification of CRAM-MD5, severe security flaws are mentioned and it is concluded that "CRAM-MD5 is no longer considered to provide adequate protection"[Ner08]. First of all, the client is not able to verify the server during authentication. This enables an attack to set up a server with a fixed challenged instead of a timestamp and to use pre-computed hashes to identify a client password. Another vulnerability is the password-based nature. Although the implementation stores the password MD5 hashed, knowing the hashed password itself is enough information to successfully authenticate against a server.

Another problem is that hashing is not sufficient for securely storing passwords. A hash should also always be salted to prevent against pre-computed rainbow table password look-ups. However, using CRAM-MD5 there is no way to transport the salt, therefore the stored password cannot be salted. Also, the hashing algorithm is MD5, which is known to be cryptographically broken [WY05] since a number of years and in itself presents a security flaw.

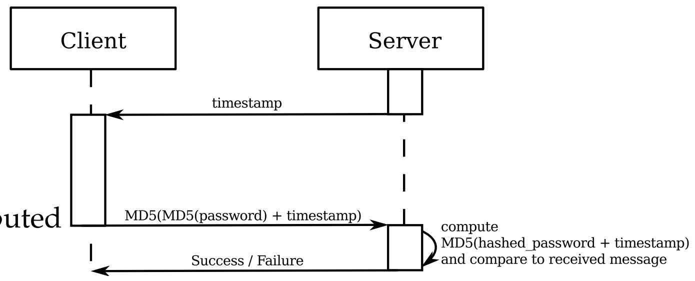

Client
Server
timestamp
MD5(MD5(password) + timestamp)
compute
MD5(hashed_password + timestamp)
and compare to received message
Success / Failure

Figure 2.7: Authentication by the Client/Server infrastructure using a modified version of CRAM-MD5.

Therefore we used a different protocol to securely authenticate a client against a server. We decided to use Salted Challenge Response Authentication Mechanism (SCRAM) as defined in [NMMW10], which is part of the family of Simple Authentication and Security Layer (SASL) protocols. SCRAM is widely used for services like SMTP, IMAP or XMPP.

The protocol provides the following functionality:

- Passwords can be hashed and salted on the server side
- Authentication happens on both sides, i.e. the client also checks that the server in fact actually did hold the correct password.
- The hashing algorithm is transparent and can be interchanged, what makes the protocol future-proof. Even if a weakness in a hashing algorithm is detected, we could easily switch to a new hashing function without breaking existing clients.

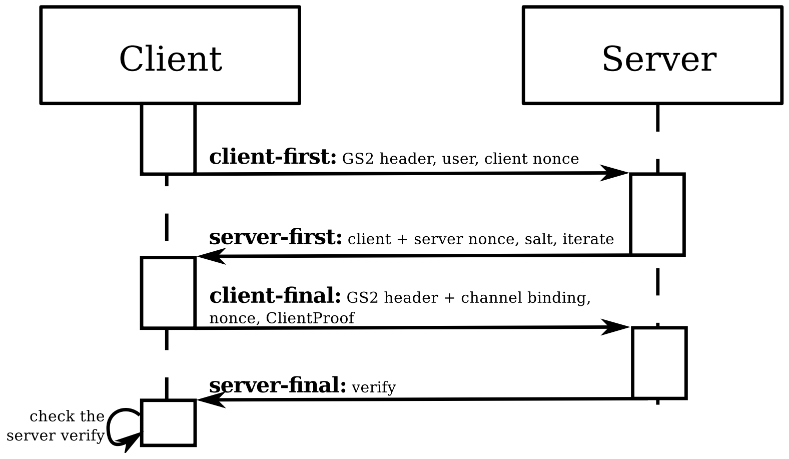

Client
Server
client-first: GS2 header, user, client nonce
server-first: client + server nonce, salt, iterate
client-final: GS2 header + channel binding, nonce, ClientProof
server-final: verify
check the server verify

Figure 2.8: SCRAM message exchange.

All messages in SCRAM are text-based and contain attribute/value pairs separated by commas. Each attribute has a one-letter name. The authentication process is initiated by a client. The message exchange as shown in Figure 2.8 works as follows:

1. CLIENT-FIRST The message starts with a GS2 header to ensure extensibility, e.g., for adding a security layer. The GS2 header must be either "n", "y" or "p". It is followed by the username and a random and unique nonce.
2. SERVER-FIRST The server responds with a nonce, computed by concatenating the client nonce and a random, unique nonce generated by the server. It also includes the salt used for hashing the users password and an iterate count. The iterate count indicates

how many times the hashing algorithm must be applied. For SHA-1 it should be at least 4096 to slow down potential attacks.

3. CLIENT-FINAL The client sends a base64-encoded GS2 header and channel binding data. Also, it sends back the same nonce. Furthermore, it includes the client proof computed by hashing the authentication message with the salted password as key and XORed with the client password. The authentication message itself is a concatenation of the client-first, server-first and client-final messages.

4. SERVER-FINAL The server computes the client proof itself and checks whether it equals the one sent by the client. If it does not, the authentication failed and is aborted. Otherwise, the server sends back a server signature to proof it does hold the valid password and there is no man-in-the middle-attack present. The server signature is computed by hashing the authentication message with the used salted password as input key. The client will then verify the server signature and if not equal has to consider the authentication process as failed.

After the authentication the initial mode settings are transmitted to the distributor.

Query execution mode

Once a client is successfully connected to a replica set via a distributor, it can send commands and queries to replica set to be executed. The client can set several *execution modes*, affecting which member within the set will actually execute the query. Database commands will always be sent to the primary.

The following modes, as also shown in Figure 2.9, are supported, with *primary Only* being the default:

PRIMARY ONLY A query will always be sent to the primary to be executed there. This is the only mode to be used for updating queries as only the primary can update data within a replica set. It is also the only mode which ensures that the most up-to-date data will be returned. Hence, it delivers strict consistency from client perspective. If an application does not tolerate to see stale data, and to experience possible isolation anomalies, this mode should be used.

mode	process updating query	can return stale data	load-balancing
primary Only	√		
secondary Round-Robin		√	round-robin
Weighted secondary		√	round-robin
Specific Member	√	√	

Figure 2.9: Comparison of different query modes for a client connection to determine the executing replication member.

SECONDARY ROUND ROBIN The query will be sent to any secondary. The secondary to be used will be selected based on a simple round-robin choice, i.e. the next secondary in a circular order of all Secondaries will be selected. This provides a good option if it is acceptable to read stale data and all nodes should receive an equal amount of queries to process. This does not ensure that each secondary receives the same amount of workload, as execution times can vary greatly between queries and are not taken into account when distributing the query.

WEIGHTED SECONDARY A weighted round-robin scheme is used to determine the node which is used to execute a query. Based on the weight, which can be defined for each member within a replica set, a percentage of queries is sent to a specific member. The percentage of queries to execute for a specific member within a replica set is proportional to the weight of this member in relation to the overall sum of all weights within the replica set. Note that the application must be able to handle stale data as well.

SPECIFIC MEMBER Using this mode a member has to be given as accompanying information. The query will be sent only to this specific member. If the member is not available, the query will fail. The member can be either a primary or a secondary. If you know that this member is a primary, the query can also be updating.

The choice what client mode to use depends heavily on the application. Of course, an application can open up multiple sessions to use different channels. It might be a common strategy to use primary Only session for updating queries and to use a secondary round-robin sessions, to distribute read-only queries to several members in the replica set. This could lead to isolation anomalies by the reading of stale data, but this might be tolerated by an application or handled on the client side to ensure strict ACID properties.

2.3.3 *Replication workflow*

Based on our aims for the replication within BaseX we now have to decide which of the presented update management strategy we want to use. Motivated by our requirement to not significantly decrease write performance, we decided to use a lazy replication approach. Using eager replication, especially in a widely spawned network of replication members, would have deteriorated write performance. Additionally, as shown in chapter 2.6 we observed that many other database systems, relational and non-relational alike, do use a lazy approach for replication.

Also, in many usage scenarios for BaseX reading stale data might be sufficient for an application. As RESTXQ has become an important technology within BaseX and many web applications make use of it, they might trade the performance increase of reading stale data against the loss in strict consistency. However, as we support multiple client query modes we still have the possibility of ensuring that only most-recent data is read from a client session perspective. This way, only the primary will execute queries and we cannot load-balance the queries, but as performance and reliability are an inherent trade-off, this presents a reasonable design decision.

Following this decision, we still have to decide whether to use a primary copy approach or a Update Anywhere update strategy. Update anywhere has the advantage that write executions can be distributed to all members in the set and the write performance can therefore be increased. However, concurrent updating operations on multiple locations which are not synchronized before a commit point can result in a write/write conflict. Hence, we would need a solution for resolving conflicts, which would try to resolve the conflict automatically for a number of cases. However, the theoretical problem remains that this concept introduces inconsistency within the database system. In addition, conflict resolution is difficult to implement and notoriously error-prone to coordinate. As BaseX is an ACID-compliant database, we did not want to trade the increased performance by introducing inconsistencies.

Therefore, we decided against using a lazy update anywhere approach and instead use a lazy primary copy approach. This has the advantage that concurrency control is greatly simplified, as all updates take place at one specific member. Also, it does provide a good performance for many write/read scenarios. As disadvantage, this introduces the possibility of reading stale data. However, using an appropriate query mode we can avoid seeing consistency anomalies per session. It increases the flexibility for a user by offering an approach to gain performance, but loose consistency guarantees.

Non-updating query execution

Let us suppose that a client wants to execute a read-only query at a replica set. By establishing a connection to a distributor we have to set a query execution mode. If reading stale data is not acceptable, *primary only* mode has to be used. Otherwise, any mode can be used with *secondary round robin* being a reasonable choice for many applications to provide load-balancing. If the performance of the members varies widely, it might be a reasonable setup to set the weight of a member according to its performance and by using the *weighted secondary* mode increase the load for members with higher performance while lowering it for members with less performance. Note that depending on the query there might be different bottlenecks. Whereas many queries with heavy database reads might be bound by I/O operations, other queries could be computationally expensive and thus CPU bound.

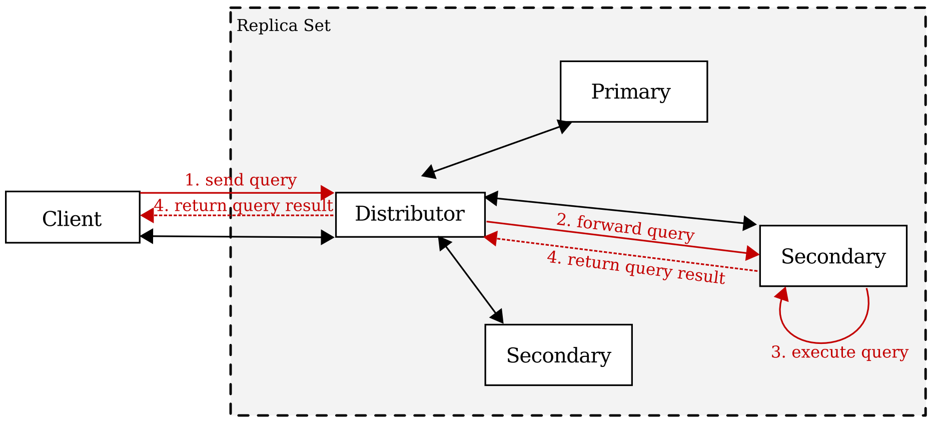


Figure 2.10: A read-only query is sent to the distributor. Based on the query mode it will be distributed to a certain member within the replica set. The member will execute the query and return the result back to the client.

Regardless of the selected mode, a client replica session is operational after the initial connection establishment. As shown in Figure 2.10 a query can now be sent to the distributor, which will forward it to the appropriate member according to the specified mode. The member itself will execute the query, as it would be done in any non-replicated BaseX instance. As the query is read-only blocking is not necessary. The result of the query will now be returned to the client.

All queries sent to Secondaries must be read-only, otherwise they will be aborted. This means that there can be no potential for an update, i.e. no updating expressions are allowed on Secondaries. This means, that even if a query in the end did not update any value due to the evaluation of the query, a query is considered updating if it contains updating expressions.

Updating query execution

Updating queries have to be sent to a replica set using a *primary only* mode, with the only exception by using *specific member* mode set to the current primary. However, this seems to be only in very specific application scenarios to be a viable solution due to the fact that a primary is automatically elected and can change at any point in time.

So a query is sent to the distributor, which forwards the query to the primary. There are two ways a database can be updated within BaseX. Either via a database command or using XQuery Update[4].

If the write operation is issued by a command, the command will simply be forwarded to all Secondaries, as commands are deterministic. The commands are then re-applied at each individual secondary.

If the write operation is done via XQuery Update, there are two separate modes we explored to update values. The mode has to be set in the configuration at the start of a replica set member.

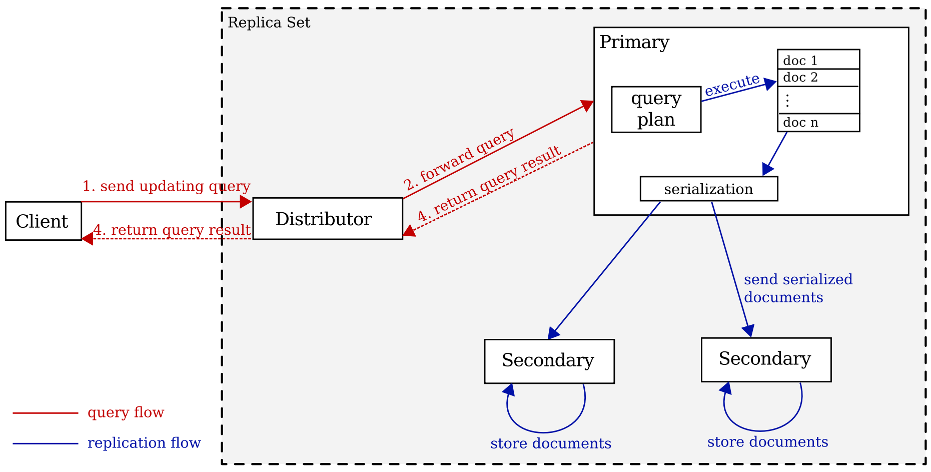


Figure 2.11: After an updating query is executed at a primary, the result is immediately sent back to the client. During execution, a list of affected document as created. This list is used to serialize all affected documents and these documents will be sent to the Secondaries.

4 XQuery Update Facility 1.0 is a language extension for XQuery designed to update instances of the XQuery 1.0 and XPath 2.0 Data Model. It is a W3C Recommendation since March 2011 and fully supported by BaseX.

The first one is *document-centric*, so if any part of a document is updated, the complete document is replicated. This is inspired by the implementation of replication by other XML databases. While eXists relies completely on document replication, MarkLogic replicates fragments. However, fragments are usually set to be the size of a document (but a document can be split up into multiple fragments), it is in fact quite similar. Also, as XML data is stored in documents, XML-based applications are usually document-centric, it is quite natural decision to also replicate data based on documents. Using the document replication mode a query will be executed and apply all changes on the local instance. During the update, identifiers of all modified documents are stored in a list. After the change is committed this list is used by the replication mechanism. It will serialize all documents and send them to all connected Secondaries within a transaction. The Secondaries will receive the documents, require all locks for the affected databases, store the new documents and finally release the locks. This way, the execution of the replication is atomic on the secondary. The workflow is shown in 2.11.

Another approach is using the *Atomic Update Cache* (AUC), which was introduced by [Kir13] to efficiently apply bulk update operations. The AUC holds all atomic updates of a transaction in document order depending on their location. The updates stored in the AUC are propagated to all Secondaries. The Secondaries will then acquire the locks for the affected databases, apply all updates on the specific secondary and release the locks. This approach is clearly favorable if only a small part of a document is updated as it avoids the overhead of serializing the complete document. Due to the timely constraints of this thesis, the implementation was just done in an early testing fashion, but showed promising results.

2.4 FAILOVER HANDLING

Members within a replica set can always fail, due to a number of reasons. The application could misbehave, a network partition could occur, a server might crash or a whole data center might experience a power outage. For this reason it is inevitable to consider failures when implementing replication for BaseX.

Failover handling consists of two major parts: First, a failure has to be reliably detected. This is a non-trivial task, as it is quite difficult to differentiate between simple message loss and a unrecoverable misbehavior of a member. We will solve this problem *probabilistically* using an accrual failure detector.

Secondly, an appropriate action has to be initiated if a failure is detected. Therefore, we incorporated a failover management strategy into the members of a replica set.

2.4.1 *Failure detector*

Any asynchronous distributed system, i.e. informally any system without an upper bound on message delays, faces the problem that consensus cannot be reached deterministically in a system with possible failures. This is due to the problem that a crashed process cannot be distinguished from a very slow one. In [CT96], Chandra and Toueg introduced the concept of *unreliable failure detectors* to detect crashed services within a distributed system.

There are two major criteria with an inherent trade-off of any failure detector. How *aggressive* a failure detector is determines how fast a failure can be detected, whereas *conservative* failure detection reduces the risk of wrongly suspecting a running service as failed, although it is still working correct. It is quite natural that the risk to wrongly suspect a node to be failed increases if the time to measure a possible failure decreases, i.e. both criteria are in conflict. Therefore, a failure detector should provide a good indication of failed nodes while maintaining reasonably low false positive estimates. As network topologies can vary widely and latency and responsiveness are very different between e.g., a large-scale distributed system deployed over several continents versus a small-scaled system within a local network, reliable failure detection remains a major challenge.

As one of our design goals for replication within BaseX includes to support very different setups, this challenge is also crucial for a replication system in BaseX to be versatile enough to adapt to very different usage scenarios. For example, it might be desirable to use replication within a replica set spanned around multiple data centers throughout the world. This can be beneficial in a number of ways; one reason could be to reduce latency for users from different parts of the world as it is commonly done using content delivery networks (CDN). It could also be part of a major availability strategy in case a complete data center experiences an outage. On the other side, it is quite common for databases to be replicated within a small local network to improve performance, increase throughput or to increase the level of availability. As the network behaves widely different for these scenarios, it becomes quite clear that a simple binary failure detector solution is not adaptable enough. In general, a binary failure detector receives monitoring input about the processes to supervise and produces a binary output, indicating whether a process is either correct or faulty.

Failure detectors can be described as a three-layered architecture. On first level, a failure detector is *monitoring* a group of services for their responsiveness, which is commonly done by sampling heartbeats or query/response times. We will focus from now on solely on heartbeats, because we used them in our actual implementation and from an theoretical point of view they can easily be interchanged with some other form of measurement of node responsiveness. The next layer is an *interpretation* of the monitoring results. Using a binary failure detector, these two layers are tightly coupled as the measurement is, based on the specific binary model, directly translated into suspecting a working or a faulty process. Following such a result, a failure detection has to take an *action* to handle the faulty node. In many cases, this will involve restarting or shutting down a faulty process. While there is always a threshold to determine a binary output, this threshold does not have to be a singular value. Instead, using an *adaptive failure detector*, this threshold can increase or decrease to incorporate changing network conditions.

For further discussion, we introduce a formal definition of failure detectors. We assume a distributed system consists of a set of processes $\Pi = \{p_1, \dots, p_n\}$. We denote to a process p as being *faulty* if its behavior deviates from its specification and as *correct* if it is not faulty. All faulty processes are member of the set $faulty(F)$ and all correct processes are member within the set $correct(F) = \mathbb{P} - faulty(F)$. We assume the existence of a global time within the domain $\mathbb{T}$, being an infinite countable set of a real number without an upper bound.

A failure detector is a set of *failure detector modules*, each one attached to a process to return information on a failure pattern within an execution. A failure pattern denotes a function $F : \mathbb{T} \mapsto 2^{\Pi}$, with $F(t)$ being the set of processes that have failed before or at time t. A failure detector history H with range $\mathcal{R}$ is defined as a function $H : \Pi \times \mathbb{T} \mapsto \mathcal{R}$, with $H(p,t)$ being the value output function by the failure detector module of process p at time t. If $q \in H(p,t)$ we say that *p suspects q at time t in H*. In the remainder, we will use two processes $p, q \in F$ with q monitoring p.

A class hierarchy of unreliably failure detectors was defined by Chandra and Toueg in [CT96]. The failure detectors we will explore are within the *eventually perfect* class, denoted as $\diamond\mathcal{P}$. A history of an eventually perfect failure detector will have the following properties:

Property 1 (strong completeness). *Eventually each faulty process in F is permanently suspected by every correct process.*

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{T}, \forall p \in failed(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q, t')$$

Property 2 (eventual strong accuracy). *There is a time t after which correct processes are not suspected by any correct process.*

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{T}, \forall p \in F, \forall q \in correct(F), \forall t' \geq t : p \notin H(q, t')$$

Following these properties we can define an eventual perfect failure detector as:

Definition 2.4 (eventually perfect failure detector). *An eventually perfect failure detector $\diamond\mathcal{P}$ is any failure detector with a history $H(q, t)$ which holds for all distinct processes p and q the strong completeness (Property 1) and eventual strong accuracy (Property 2) properties.*

The quality of service provided by a failure detector can be defined by several metrics such as:

- The *detection time* is the elapsed time since p fails and until q suspects p permanently.
- The *average mistake time* measures the time a process q wrongly suspects a process p of being faulty.
- The *query accuracy probability* is the probability that for a given random time t the output of a failure detector is correct.

Accrual failure detector

Compared to a binary failure detector, an accrual failure detector will decouple the dependency between monitoring and interpretation and will instead move the interpretation to the network level. As requirements can differ so widely, interpretation of the monitoring results is best left to an application. To refer back to the example from above, it is quite clear that the same response times will have a very different meaning whether a process within the same LAN or on another continent is supervised. Also, it might be appropriate to take different actions at different levels of suspicion that a process may have failed. Therefore, an accrual failure detector will output a value on a continuous scale, representing the degree of confidence that the supervised process has indeed failed. An accrual failure detector can easily be converted to a binary one if a single threshold is defined. The advantage of using accrual failure detection is that we can manage multiple thresholds to trigger different actions. For example, in a distributed system with the responsibility to distribute jobs to many nodes, a low threshold could be defined to temporarily suspend the submission of new jobs to a node and another higher threshold to shutdown the node indefinitely.

As already mentioned, a failure detector cannot be implemented deterministically in all possible asynchronous systems with failures. This condition also holds for accrual failure detectors, as it is mainly an abstraction from a binary failure detector. However, an accrual failure detector can be implemented *probabilistically*.

The confidence level can be formally defined by the suspicion level of an accrual failure detector:

Definition 2.5 (suspicion level). *The suspicion level of process q with respect to process p is the function $sl_{qp} : \mathbb{T} \mapsto \mathbb{R}_0^+$. The function sl_{qp} has a finite resolution.*

In addition, the suspicion level also satisfies the following two properties:

Property 3 (accruement). *If process p is faulty, then eventually the suspicion level $sl_{qp}(t)$ is monotonously increasing at a positive rate, i.e. the suspicion level may remain constant just for a bounded number of queries Q:*

$$\forall p \in faulty(F), \exists K \exists Q \in \mathbb{Z}^+, \forall k \geq K :$$
$$(sl_{qp}(t_q(k)) \leq sl_{qp}(t_q(k+1))) \land (sl_{qp}(t_q(k)) < sl_{qp}(t_q(k+Q)))$$

Property 4 (upper bound). *If process p is correct, then the suspicion level $sl_{qp}(t)$ has an upper bound SL_{max}.*

$$\forall p \in correct(F), \exists SL_{max}, \forall t \in \mathbb{T} : sl_{qp}(t) \leq SL_{max}$$

From this we can directly define the class of accrual failure detectors $\diamond\mathcal{P}_{ac}$:

Definition 2.6 (accrual failure detector). *An accrual failure detector is any failure detector within range $\mathcal{R} = \mathbb{R}_0^+$ and a history $H(q,t)(p) = sl_{qp}(t)$ which holds for all distinct processes p and q the accruement (Property 3) and upper bound (Property 4) properties.*

This definition contains a number of interesting properties, which make them well-suited for a failure detector and for a real-world implementation. For example, the upper bound is unknown. Otherwise, applications could just compare the suspicion level to the known bound, but it is desirable to leave the interpretation to the application itself. Also, the accruement property allows for stationary periods and does not employ strict monotony. Otherwise, an implementation might have difficulties to return an increased value upon every query. This would entail to either access a hardware clock with a fine enough resolution (which might not be available) or to artificially increase the suspicion level. Also note that the accruement property is defined without any reference to a time. This way the model does not require access to any synchronized clock.

An accrual failure detector of class $\diamond\mathcal{P}_{ac}$ has the same computational power as an binary failure detector of class $\diamond\mathcal{P}$, i.e. any problem solvable by a failure detector within $\diamond\mathcal{P}$ can also be solved by a failure detector within $\diamond\mathcal{P}_{ac}$.

φ accrual failure detector

The φ accrual failure detector provides a probabilistic implementation of an accrual failure detector and was defined by Hayashibara et al. in [HDYK04].

The suspicion level is given by a value $\varphi = sl_{qp}$, which is dynamically adjusted to reflect current network conditions. The architecture to compute φ can be described by three phases:

1) SAMPLE HEARTBEATS: The monitoring process q sends heartbeats to the monitored process p and stores the response time in a sampling windows with a fixed size. Whenever a new heartbeat response arrives, it is stored in the window and after the oldest heartbeat has been deleted in case the window was at full capacity.

2) ESTIMATE THE HEARTBEAT RESPONSE TIME: The response times of a heartbeat can be estimated by using a normal distribution. The mean μ and the variance σ^2 of the samples within the sampling window can be used as parameters for a distribution function. The probability $P_{later}(\Delta t)$, that a given heartbeat will return more than t time units later than the previous heartbeat, can be modeled using the *cumulative distribution function* (CDF) $\Phi(t_\Delta)$ for a normal distribution, which uses the *probability density function* $f(x; \mu, \sigma^2)$:

$$\begin{aligned} P_{later}(t_\Delta) &= 1 - \Phi(t_\Delta) && (2.1) \\ &= 1 - \int_{t_\Delta}^{+\infty} f(x;\mu;\sigma^2)dx && (2.2) \\ &= 1 - \frac{1}{\sigma\sqrt{2\pi}} \int_{t_\Delta}^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx && (2.3) \end{aligned}$$

3) COMPUTE φ: Based on the estimate we can now compute φ. We denote t_{pre} as the time unit when the previous heartbeat arrived and t_{now} as the current time unit. Then φ is defined as:

$$\varphi(t_{now}) = -\log_{10}(P_{later}(t_{now} - t_{last}))$$

It is easy to show that a φ accrual failure detector is a failure detector of class $\diamond\mathcal{P}_{ac}$. As φ is defined using a CDF and every CDF over a random variable is monotonically increasing, the accruement property (Property 3) immediately follows. Similarly, as the CDF for the normal distribution has a lower bound of 0 and an upper bound of 1 and is directly used to define φ, it follows that the property upper bound (Property 4) is also valid.

Assuming that the network is probabilistically stable, this means when using a threshold Φ to suspect a process of being faulty compared to the φ value, the likelihood of a false positive is $10^{-\Phi}$. Thus, an application can increase the threshold Φ to increase the conservativeness of the failure detection, i.e. minimize the probability to wrongly suspect a process to be faulty. Alternatively, the threshold can be lowered to increase the aggressiveness of failure detection, i.e. to detect failures faster. This adaption of the threshold can be done by the application in a flexible manner and does not depend on static parameters.

Thereby, an φ accrual failure detector allows a flexible framework for failure detection without the need to tune parameters statically for a single application. As was shown in [HDYK04] this comes without any significant performance drops in reference to the above define qualities of services and therefore provides much more flexibility with the same quality of service levels as provided by other adaptive failure detectors such as Chen or Bertier failure detector.

CDF estimation accrual failure detector

In [SPTU07] Satzger et al. described an interesting approach for a new adaptive accrual failure detector, which we will refer to as *CDF estimation* *accrual failure detector*. It is largely based on the φ accrual failure detector, but instead of precisely calculating the CDF, the CDF is estimated. Suppose a process p maintains a list S with the latest X received heartbeat times and also maintains the last received heartbeat time t_{last}.

The sample times within S and their corresponding frequency can be described as a histogram. This histogram can be seen as a rough estimate of a probability density function. From the history the cumulative frequencies of values within S can easily be computed and this cumulative frequency can be seen as an estimate of the CDF.

Algorithm 2.1 CDF estimation accrual failure detector

```
1:  function INIT
2:      $t_{last} \leftarrow -1$                ▷ last arrived heartbeat time
3:      $S \leftarrow$ newQueue()   ▷ $S$ is initialized as a new, empty list with a maximum size of $\mu$
4:  end function
5:  function RECEIVEHEARTBEAT($m_j$, $t_{now}$)
6:      if $f = -1$ then
7:          $f \leftarrow t_{now}$
8:      else
9:          $t_\Delta = t - t_{last}$
10:         $t_{last} = t_{now}$
11:         append $t_\Delta$ to $S$
12:         if size$(S) > \mu$ then   ▷ If the queue is full, remove the element first inserted
13:             remove head of S
14:         end if
15:     end if
16: end function
17: function GETSUSPECTLEVEL($q$, $t_{now}$)
18:     $t_\Delta = t_{now} - t_{last}$
19:     $count \leftarrow 0$
20:     for all $\{x \in S | x \leq t_\Delta\}$ do
21:         $count \leftarrow count + 1$
22:     end for
23:     return $\frac{count}{size(S)}$
24: end function
```

So we can now model the suspicion level for this failure detector similar according to equation 2.1, but this time we will use an estimate of the CDF to calculate the probability $P_{later}(t_{now} - t_{last})$ by using the set $S_{min,max} = \{x \in S | x \leq (max - min)\}$.

$$sl_{qp}(t_{now}) = P_{later}(t_{now} - t_{last}) \tag{2.4}$$

$$= \frac{|S_{t_{last},t_{now}}|}{|S|} \tag{2.5}$$

So to provide such an CDF estimation accrual failure detector, the given suspicion level can be computed using Algorithm 2.1.

In comparison to the φ accrual failure detector, this has three main benefits:

- The calculation cost to compute an suspicion level is reduced; by using an appropriate data structure for S, a suspicion level can be computed within $\mathcal{O}(\log|S|)$.
- The φ accrual failure detector is restricted to networks with a roughly normal distribution of heartbeat arrival times, whereas this approach has no such limitation. However, this advantage is quite likely rather weak in real-world situations, as network arrival times will mostly be adequately modeled by a near normal distribution.
- It simplifies the implementation by its straight-forward model.

The presented failure detector is also an accrual failure detector. The accruement property (Property 3) holds, as a cumulative function of non-negative values is always monotonously increasing. The upper bound property (Property 4) also holds as the estimation is normalized and will never exceed 1.

The experiments by Satzger et al. showed that the CDF estimation accrual failure detector is similar in performance to the φ accrual failure detector in terms of falsely suspected processes and the average detection time.

Because of a similar performance, lower computational cost and simplified implementation we decided to use the CDF estimation accrual failure detector for a replica set within BaseX.

2.4.2 *Failover management*

Each secondary within the replica set sends a regular heartbeat using in a configurable interval t_Δ to the primary. This way, a primary can detect faulty Secondaries. If a secondary is suspected to be faulty, it will be removed from the replica set. All connected distributors will be notified by the primary of this removal to not send queries to this secondary anymore via the non primary-only query execution modes.

As we use an accrual failure detector to detect faulty members, a suspicion level is collected for each secondary on the primary. The higher the suspicion level *sl*, the more likely is the member to actually be failed. The suspicion level is a continuous random variable without an upper bound. We assume that $sl \sim \mathcal{N}(\mu, \sigma^2)$, i.e. the normal distribution. We sample mean μ and variance σ^2 over the set of the latest suspicion levels of all Secondaries.

The CDF of a normal distribution is defined as

$$\Phi(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt \tag{2.6}$$

As a CDF describes the probability that a random variable with a given probability distribution will hold a value less or equal to x, it can be used to model a statistically significant outlier of the data set. Our goal is to detect any secondary with a suspicion level significantly larger than the average. Hence, we define the following definition to define a secondary as malfunctioning:

Definition 2.7. *A secondary s is deemed faulty, if for the suspicion level sp_s, the CDF $\Phi(x)$ of a normal distribution and a given λ_1 the following holds true:*

$$\Phi(sp_s) \geq \lambda_1, \lambda_1 \in \mathbb{R} \wedge 0 \leq \lambda_1 \leq 1$$

The value λ_1 can be configured by the user. Therefore, a user can choose the value according to application needs. Deploying a replica set in a local area network will most likely result in less variance than using a replica set in a globally deployed cloud network environment. Hence, the user can set λ_1 accordingly to not trigger many false positives or to quickly detect failed nodes. Also, it will often be suitable to lower the value of λ_1 if replica set has just a few members. In this case, the statistic significance of the mean and standard deviation are decreased, i.e. an outlier is more likely.

Also, a primary sends a heartbeat message to all other members within the replica set (which by definition will be Secondaries) in the same interval t_Δ. Using the same accrual failure detector, we will use this heartbeat again to compute suspicion level for the possible failure of the corresponding member, in this case the primary. An example for a failing primary, the detection and the taken action is shown in Figure 2.12.

Contrary to primary, which provides a set of suspicion levels of several Secondaries, this time we only have this one suspicion level. Hence, it will be assumed that a primary is faulty if the suspicion level is greater than a given *suspicion level threshold*, denoted as λ_2. Similarly to other parameters this value can be configured by the user within the configuration settings.

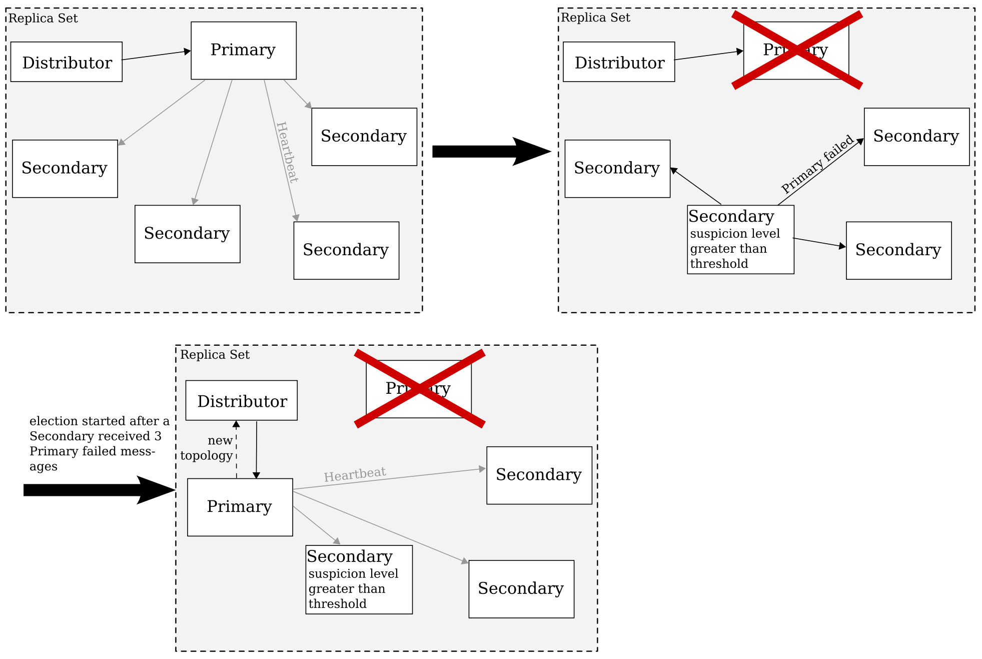


Figure 2.12: A primary sends heartbeats to all Secondaries. If a secondary suspects a primary to have failed due to non-arriving heartbeats, it notifies all Secondaries within the set. If any secondary gets a majority of such messages, an election is initiated. When the new primary is elected it will notify all distributors of the new replica set topology.

If a failure is detected by a secondary, it will communicate it to all other Secondaries. If any secondary receives a majority of votes for the removal of a primary, it will initiate a leader election and remove the former primary from its view of the replica set topology. The newly elected primary will inform all connected distributors about the new topology of the replica set.

Assuming the replica set consists of n members there will be $2(n-1)$ heartbeat messages sent every t_Δ time units. t_Δ can be set within the configuration and defaults to 2 seconds.

2.5 LEADER ELECTION

2.5.1 *Problem definition*

Leader election in general is an algorithm for choosing a distinct process from a set of processes to serve a special functionality within this set. Within a BaseX replica set there is a primary, which acts as a leader and has the special responsibility to update data items and propagate the changes to all secondaries. The primary is also responsible for members joining and leaving the set.

Each replica set member can have one vote in an election. A member can be configure to be non-voting and will not participate in an election process. Exactly one member will be elected as new primary at the end of a leader election run. A member can be configured to solely serve as secondary, i.e. is not eligible during the election. Only members with the most recent *updating timestamp* of any visible member within the set are eligible. The updating timestamp is the timestamp of the last applied updating operation.

While an election is in progress, there is no primary present in the replica set. Thus, updating operations will not be accepted. This is the primary reason why we designed a leader election algorithm which will elect a new primary fast to minimize the time a replica set is in a read-only state.

There are several reasons why a election may be triggered:

- a new replica set is initialized
- the primary of a replica set voluntarily steps down
- a secondary loses contact with the primary
- the primary loses contact to the majority of the secondaries

After triggering an election it is checked if the election will take place. The election is initiated iff a calling member sees a majority of the members in a set to avoid establishing two separate replica sets due to network partitions. If the election does not take place, no new primary will be elected and the replica set will be put into read-only mode until the issue is resolved by an administrator. Thus, a replica set cannot stay fully functional if more than half the members in a replica set fail.

To formalize election algorithms, we introduce the notion that a leader process is elected from a set $\Pi = \{p_1, p_2, \dots, p_n\}$ with n processes. Each process has a *process number*, which is unique and totally ordered within the set. The process with the highest process number will win the election.

A process can call an election to initiate a leader election, but one process does not call more than one election at a time. As elections can be called by multiple processes simultaneously, up to n leader elections can be initiated.

Based on this conditions, we can formalize the requirements for any run of a leader election algorithm to hold the safety and liveness property.

Property 5 (safety). *At most one process p_i is a leader.*

$$\forall p_i, p_j \in \Pi, i \neq j : \neg(leader(p_i) \wedge leader(p_j))$$

Property 6 (liveness). *Eventually, all processes are either a leader or non-leader and are aware of at least one elected leader.*

$$\forall p_i \exists p_j \in \Pi : (leader(p_i) \vee nonLeader(p_i)) \wedge leader(p_j)$$

2.5.2 *Bully algorithm*

The bully algorithm was introduced by Garcia-Molina in [Gar82]. It is named *bully*, because the process with the highest process number will force processes with smaller process number to accept it as leader, i.e. it will bully them into acceptance.

It makes the following assumptions:

i. Message delivery between processes is reliable.

ii. The system is synchronous and uses timeouts to detect process failures. The time time T is an upper bound for the total elapsed time from sending a message to a process until receiving a response.

iii. Every process knows the process number of any other process in the set and communicates with each other processes.

Although a process knows the process number of each process in the set, it does not known whether the process is currently available or not.

The election protocol is divided into two parts. First, a process p_i that calls an election, notifies all processes with a higher process number of the election. If any node responds, p_i knows it can not be the new leader and will finish its participation in the election and wait for a new leader to be announced. However, if no process answers within the time T, the initiating process assumes that all other processes are unavailable and that the process itself is the new leader. All notified processes now behave in the same way and notify on their own processes with a higher process number than themselves. During an election run, each process only sends out such notifications once. If a process does not get any response in T it will start the second part of the protocol. The second part of the protocol is the new leader announcement phase, whereby a new leader sends out an announcement of its leadership to all processes. The protocol is shown in Algorithm 2.2.

Algorithm 2.2 Bully election algorithm

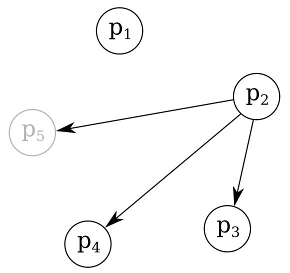
```
1: function CallElection($p_i$)                                   ▷ $p_i$ calls an election
2:     for all $p_j \in \Pi, pc(p_j) > pc(p_i)$ do
3:         Send Election to $p_j$
4:     end for
5:     wait $T$
6:     if $\neg received(ANSWER)$ then
7:         send announce
8:     end if
9: end function
10: function ReceiveElectionMessage($p_j$, $p_i$)   ▷ $p_j$ received an ELECTION message from $p_i$
11:     Send Answer to $p_i$
12:     call CallElection($p_j$)
13: end function
14: function Announce($p_i$)                              ▷ $p_i$ announces itself as leader
15:     for all $p_k \in \Pi, p_i \neq p_k$ do
16:         Send Announce to $p_k$
17:     end for
18: end function
```

The protocol execution is shown by example in Figure 2.13. There are 5 processes within the process set $\Pi = \{p_1, p_2, p_3, p_4, p_5\}$ and the process number is in the range of the natural numbers $\mathbb{N}$ (and therefore totally ordered). The process number of each process is equal to the index number of the process.

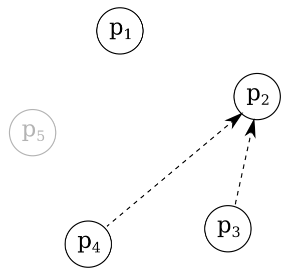
The safety property is clearly met with this algorithm, as a process with a lower process number will detect the existence of a process with a higher process number. The liveness property also holds, as the upper message time T and the reliable delivery of messages will eventually result in a process to announce itself. Also, by definition each process not being a leader is a non-leader.

2.5.3 *Modified Bully algorithm*

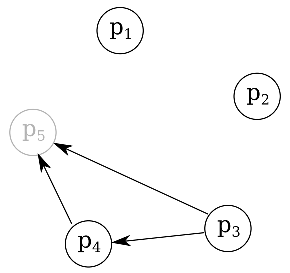
In [MMM04] Mamum et al. introduced a modified bully algorithm, based on the original bully algorithm by Garcia-Molina. Therefore, it is based on the same assumptions as the original bully algorithm. The algorithm is shown in Algorithm 2.3.



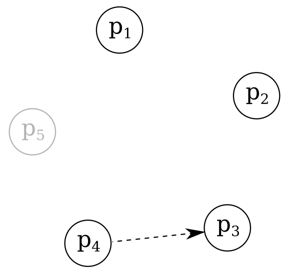
(a) p_2 calls an election and notifies all process with a higher process number



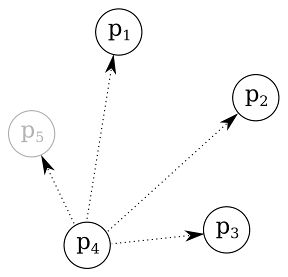
(b) All processes answer the notification. As p_5 is unavailable it does not sent a response.



(c) The notified processes now notify in turn their processes with a higher process number, i.e. p_3 notifies p_4 and p_5 and p_4 notifies p_5.



(d) p_4 answers the notification. p_4 does not receive any response within time T.



(e) As p_4 did not receive any response it announces itself as the new leader to all processes.

Figure 2.13: Bully protocol execution for an election with five processes. p_5 is currently not available.

Any process can call an election by notifying all processes with a higher process number. A notified process will respond to the sender. If the election initiator does not receive any response, it can reliably assume all other processes have failed and announces itself as coordinator. Up to this point, the algorithm is identical to the original bully algorithm. The modification is build on the handling when a response is actually received. Collecting all responses, the process knows the active process with the highest process number. Therefore, it can immediately announce this process as the new leader. The initiating process coordinates the complete election run.

An example protocol execution is shown in Figure 2.14.

In the worst case, i.e. the election is called from the process with the lowest process number, $O(n^2)$ messages in a process set with n processes are needed for electing a new leader using the original bully algorithm.

Algorithm 2.3 Modified bully election algorithm

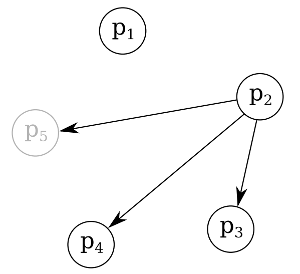
```
 1: function CALLELECTION($p_i$)                                  ▷ $p_i$ calls an election
 2:     for all $p_j \in \Pi, pc(p_j) > pc(p_i)$ do
 3:         Send ELECTION to $p_j$
 4:     end for
 5:     wait $T$
 6:     if $\neg received(OK)$ then
 7:         send ANNOUNCE($p_i$)
 8:     else
 9:         $p_{min} \in \Pi \leftarrow \forall p_k \in \Pi : pc(p_{min}) \leq pc(p_k)$
10:         send ANNOUNCE($p_{min}$)
11:     end if
12: end function
13: function RECEIVEELECTIONMESSAGE($p_j$, $p_i$)    ▷ $p_j$ received an ELECTION message
        from $p_i$
14:     Send OK to $p_i$
15:     call CALLELECTION($p_j$)
16: end function
17: function ANNOUNCE($p_i$)                          ▷ $p_i$ announces itself as leader
18:     for all $p_k \in \Pi, p_i \neq p_k$ do
19:         Send ANNOUNCE to $p_k$
20:     end for
21: end function
```

Using the modified version, only $O(n)$ messages are required in the worst case. In the best case, i.e. the process with the highest process number calls the election and immediately announces itself, both protocols need $O(n)$ messages.

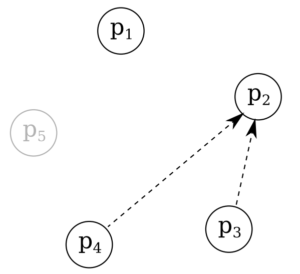
2.5.4 *Modified Bully algorithm with unreliable delivery guarantees and limited eligibility*

The original and the modified bully algorithm both assume that message delivery between processes is reliable. However, as Akka only guarantees at-most-once delivery guarantees, we have to consider lesser assumptions. Hence, we developed the following assumptions:

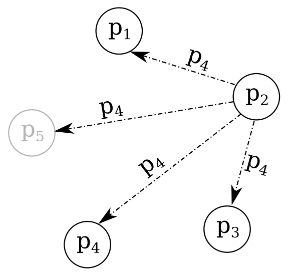
- The set of process numbers is totally ordered.



(a) p_2 calls an election and notifies all process with a higher process number



(b) All processes answer the notification. As p_5 is unavailable it does not sent a response.



(c) p_2 identifies the responding process with the highest process number. This process will become the new leader, in this case p_4. p_4 is now send via a COORDINATOR message to all processes.

Figure 2.14: Modified bully protocol execution for an election with five processes. p_5 is currently not available. Note that the first two phases are identical to the original bully algorithm.

- Each process knows the process number of each other process in the set. However, a process does not know if another process is active or eligible at a certain time.
- The eligible process with the highest process number is elected as new leader.
- Messages between processes are sent at most once and are sent in order for each sender-receiver pair.

Apart from unreliable message delivery guarantees, another distinction is the proposed eligibility of a process. In the two previously introduced algorithms we always assumed that each process can be elected. However, we do have a different application requirement. When the primary of a replica set fails we want a new primary with the most recent update timestamp to take over, as otherwise the loss of data is increased. Therefore, only processes with the highest timestamp of the replica set can be elected. This eligibility criteria could also be something different from a timestamp and it could be easily adapted. However, for the simplicity of explanation we assume from now on that the eligibility criteria is to have the most recent timestamp in the replica set.

Also, the most recent timestamp could be known when starting an election. Suppose, the election is gracefully initiated by a step-down of the current primary. This way, the primary will suspend accepting updating operations and call an election, knowing the most recent timestamp ts_{max}. Of course, when a primary fails unexpectedly and the failure is detected by a secondary, the most recent timestamp is not known to any process in the set. Therefore, we denote to an unknown timestamp as $\perp$. To check for message drops, we retry sending a message up to κ times.

The algorithm can be divided into three phases. In the first phase (shown in Algorithm 2.4) the process p_i calling the election sends a notification to all processes with a higher process number than itself. It waits to receive replies from all processes and, as already mentioned, retries sending the message up to κ times. A processes receiving a notification replies with an Ok response, accompanied with its updating timestamp ts. The calling processes now checks all received responses and identifies a *candidate process*, which must have the most recent timestamp and is the process with the highest process number. If ts_{max} was given when calling the election and the candidate process has the same timestamp, we can stop the election process. As no process can have a more recent timestamp and a higher process number, the candidate process must be elected as new leader. Hence, the second phase of the protocol can be skipped and the candidate process will immediately be announced as new leader.

Otherwise, the calling election process will now check the processes with lower process numbers for availability and eligibility, as shown in Algorithm 2.5. Therefore, a notification is sent to all processes with a lower process number (up to κ times). The processes reply with their last update timestamp. After receiving all responses, we can now determine the absolute ts_{max} by getting the most recent timestamp of all responses from processes with a lower process number and the timestamp of candidate process $p_{candidate}$. As $p_{candidate}$ represents a local maximum for all processes with a higher or equal process number, it is guaranteed that we compute the most recent timestamp of all processes still available in the set.

Hence, we can now transition to the third and final phase by announcing $p_{candidate}$ as new leader. This phase is formalized in Algorithm 2.6.

The announcement phase is once again with a acknowledgement message. Otherwise, the message is retransmitted up to κ times. The announcement of the new leader is done by the election calling process. All processes are informed of the new leader and acknowledge the message receive with an Ack.

Regarding the implementation, we would like to mention that using blocking operations in the algorithm does not translate to blocking calls within our implementation. Wait calls and sleeps can also be implemented in an event-driven way. Given that Akka is based on asynchronous events and the overall beneficial performance of non-blocking operations, we implemented the algorithm in a non-blocking way.

An example protocol execution for this modified version of the bully algorithms is shown in Figure 2.15.

Algorithm 2.4 Phase 1 of the modified bully algorithm with unreliable delivery guarantees and timestamp eligibility.

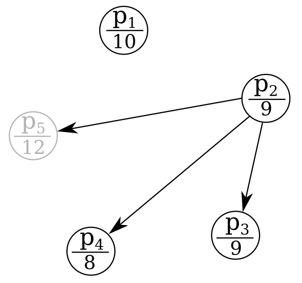
```
1: function CheckHigher($p_i$, $ts_{max}$)    ▷ $p_i$ calls an election, $ts_max$can be the unknown value $\perp$ or a real maximum value
2:     $x \leftarrow 0$
3:     while $x < \kappa$ and not all responses received do
4:         for all $p_j \in \Pi, pc(p_j) > pc(p_i), noResponse(p_j)$ do    ▷ Send a message to all processes with a higher process number, which did not sent back a response yet
5:             Send Election to $p_j$
6:         end for
7:         wait $T$
8:         $x \leftarrow x + 1$
9:     end while
10:    $\mathcal{S} \leftarrow p_j \in receivedResponses \cup p_i$    ▷ Set of all process which did sent a response and the calling process
11:    $ts_{candidate} \leftarrow p_j \in \mathcal{S}, pc(p_j) \geq pc(p_i) : max(ts_{p_j})$    ▷ most recent known timestamp of all processes
12:    $p_{candidate} \leftarrow max(\{p_j \in \mathcal{S}, ts_{p_j} = ts_{candidate}\})$    ▷ eligible process with the highest process number
13:    if $ts_{max} \neq \perp \wedge ts_{max} = ts_{candidate}$ then    ▷ If $ts_{max}$ was given and any process holds the same timestamp as $ts_{max}$, announce this process as new leader
14:        Call announce($p_{candidate}$)
15:    else
16:        Call TransferElection($p_i$, $p_{candidate}$)
17:    end if
18: end function
19: function ReceiveElectionMessage($p_j$, $p_i$)    ▷ $p_j$ received an Election message from $p_i$
20:    Send Ok($ts_{p_j}$) to $p_i$
21: end function
```

Algorithm 2.5 Phase 2 of the modified bully algorithm with unreliable delivery guarantees and timestamp eligibility.

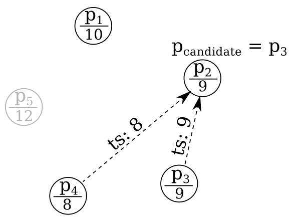
```
1:  function CHECKLOWER($p_i$, $p_{candidate}$)   ▷ Transfers the election responsibility to the reachable process with the lowest process number
2:      $x \leftarrow 0$
3:      while $x < \kappa$ and not all responses received do
4:          for all $p_j \in \Pi, pc(p_j) < pc(p_i), noResponse(p_j)$ do   ▷ Send a message to all processes with a lower process number, which did not sent back a response yet
5:              Send ELECTION to $p_j$
6:          end for
7:          wait $T$
8:          $x \leftarrow x + 1$
9:      end while
10:     $\mathcal{S} \leftarrow p_j \in receivedResponses \cup p_{candidate}$   ▷ set of all processes with lower process number and the candidate process
11:     $ts_{max} \leftarrow p_j \in \mathcal{S}, pc(p_j) \geq pc(p_i) : max(ts_{p_j})$   ▷ most recent timestamp of all processes
12:     $p_{candidate} \leftarrow max(\{p_j \in \mathcal{S}, ts_{p_j} = ts_{max}\})$   ▷ eligible process with the highest process number
13:     Call ANNOUNCE($p_{candidate}$)
14: end function
```

Algorithm 2.6 Phase 3 of the modified bully algorithm with unreliable delivery guarantees and timestamp eligibility.

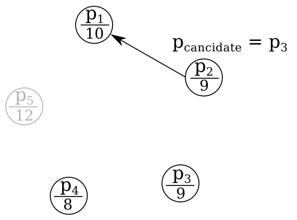
```
1:  function ANNOUNCE($p_i$)   ▷ $p_i$ announces itself as leader
2:      while $x < \kappa$ and not all responses received do
3:          for all $p_k \in \Pi, p_i \neq p_k, noResponse(p_k)$ do
4:              Send ANNOUNCE to $p_k$
5:          end for
6:          wait $T$
7:          $x \leftarrow x + 1$
8:      end while
9:  end function
10: function RECEIVEACKNOWLEDGMENT($p_i$, $p_{leader}$)
11:     store $p_{leader}$ as new leader
12:     Send ACK to $p_i$
13: end function
```



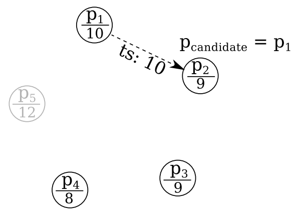
(a) p_2 calls an election and notifies all process with a higher process number



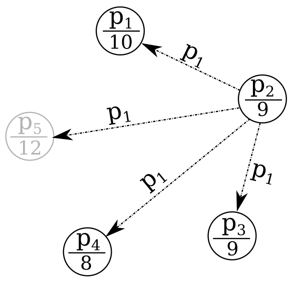
(b) All processes answer with their respective timestamp As p_5 is unavailable it does not sent a response. Based on the timestamps and process numbers, a p_3 is determined to be *candidate*, as it has the highest process number of all currently involved processes with the maximum timestamp.



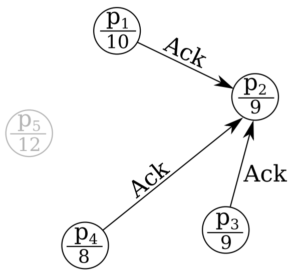
(c) The calling process now has to check the processes with lower process numbers, as they could have a more recent timestamp than the current candidate. Hence, all processes with a process number lower than p_3 are notified.



(d) p_1 answers the notification and sends back the timestamp 10. As this is greater than the timestamp of the current candidate, p_1 becomes the new candidate. As we have checked all process, p_1 will become the new leader.



(e) All process are notified of the new leader p_1.



(f) Each process has to acknowledge the new leader to allow retransmissions for dropped messages.

Figure 2.15: Modified bully protocol with unreliable delivery guarantees and timestamp eligibility execution for an election with five processes. Each circle represents a process, whereby the upper half is the process name with the process number as index and the lower half is the most recent update timestamp. p_5 just failed, therefore a new leader has to be elected.

2.6 RELATED WORK

Many DBMS use replication and a wide variety of consistency guarantees and concurrency control algorithms are enforced. We will now shortly describe how various other relational and NoSQL databases handle replication and what levels of consistency they provide.

IBM DB2 is a relational DBMS and uses a lazy update anywhere approach. It supports different replication modes, Q replication, SQL replication and event publishing which share the characteristic that transactional data is sent to replicas after commit [03; KKLK09]. It uses a publish-subscriber model with message queues or staging tables to propagate updates. It provides transactional, but not mutual consistency. It supports an advanced conflict resolver to reconcile conflicts.

Oracle supports a number of different replication modes [GF03]. The databases can be set up to support eager or lazy, primary copy or update anywhere replication concepts or any combination thereof. Using eager replication, strict 2PC is used to ensure consistency. Lazy replication supports conflict resolution. Updates can either occur solely on a single master (primary copy) or on multiple masters (update anywhere). Hence, Oracle supports transactional consistency and can support mutual consistency, if required by the application.

MongoDB is a document database, based on a binary JSON format, and uses a lazy primary copy approach and was a major inspiration for our design [14b]. MongoDB features a replica set, which automatically elects one of its member as primary with all other replicas being Secondaries. The primary is the only member within the replica set accepting updates, therefore the replication uses a primary copy approach. These updates will be propagated after the transaction has already been committed, making it a lazy approach. Submitted queries can use different levels of read and write concerns to define when and how a query is processed and committed. Hence, MongoDB can provide transactional consistency and provides

	eager propagation		**lazy propagation**		**consistency types**	
database	**primary copy**	**update anywhere**	**primary copy**	**update anywhere**	**transactional consistency**	**mutual consistency**
DB2				√	√	
Oracle	√	√	√	√	√	√
MongoDB			√		√	√
Cassandra		√		√	√	√
eXist			√			√
MarkLogic			√		√	√

Figure 2.16: Comparison of replication types and provided levels of consistency of several popular DBMS.

mutual consistency. However, as MongoDB by default only supports eventual consistency, also the replication consistency level is limited to eventual consistency.

Apache Cassandra, a column store built for commodity hardware and cloud usage, uses an update anywhere protocol. It can be used both in an eager or in a lazy fashion as it supports tunable levels of consistency for write and read queries [Bla10]. Therefore, Cassandra supports both transactional and mutual consistency, and also drops these strict consistency levels for higher availability and lower response time, depending on the application use case.

eXist is an XML database and XQuery processor and uses a master/slave replication strategy. The master is the only updating location. An update will trigger the replication manager after commit and serialize the complete document and then send the complete document to all replicas [14a]. Thus, it is a lazy primary copy approach. This provides mutual consistency, but no transactional consistency when data is also read from slaves.

MarkLogic is a document-centric XML database. It uses two different approaches of replication[13], database and flexible replication. Both are lazy primary copy approaches, but only database replication enforces transactional consistency. Flexible replication is mainly a trigger mechanism after data has been updated, while database replication is a more advanced concept by detecting and replicating changes to another fragment.

3

PERFORMANCE

Up to now, we introduced our design goals and based on them we chose an architecture to fit these goals. Many individual components of the replication system were tailored to fit the specific needs of a document-centric XML database. In this chapter we want to explore how well the architecture behaves to our expectations.

3.1 TEST ENVIRONMENT

We use the XMark data set as introduced in [SWK+02] to run our experiments. XMark is a standardized data set often used for performance measurement for XML data. The data is modeled after an Internet auction site, consisting of open and closed auctions, items and users. Hence, the data set has some semantic meaning and is not simply an artificial data set.

Using the provided data generator, we can choose a *scaling factor* to determine the size of the created document. The size for different factors we use is shown in Figure 3.1. The data will be generated into one single document. We can also specify a *split count*, to evenly split up the documents into several parts. The split factors indicates the number of elements representing a business object (such as auction, user or item). Hence, the files have a natural split, but are only roughly equal in size. However, the file size varies just in a small margin and is not that relevant.

scaling factor	document size	# of elements and attributes
0.01	1.2 MB	21048
0.1	12 MB	206130
1	112 MB	2048180

Figure 3.1: Scaling factors for XMark data and the resulting document size and number of elements and attributes combined.

As testing environment we use Amazon EC2 instances. We use *t2.micro* instances, which have the following performance parameters:

- Intel Xeon Processors operating at 2.5GHz with Turbo up to 3.3GHz
- 1 virtual CPU. CPU is given with a burstable performance, i.e. 6 CPU credits are given per hour, so in idle times they accrue and during heavy workload a CPU Turbo with more dedicated CPU is provided.
- 1 GiB memory
- 30 GB SSD-backed storage
- Red Hat Enterprise Linux, 64bit

As our implementation uses Java 8, we do need a JRE with support for that, as opposed to non-replicated BaseX which does use Java 7. We use Oracle Java 8 Update 20.

3.2 TRANSACTION LATENCY

One of our main design goals was that we do not want to negatively impact performance, i.e. increasing the time until a transaction is committed should be avoided. The time between submitting a transaction and getting a result back is called *transaction latency*. As XMark provides no test queries for XQuery update, we came up with the following four queries to test transaction latency:

Q1: `replace node /site/regions/africa/item[@id = "item0"]/quantity with <quantity>2</quantity>`

Q2: `for \$item in /site/*/* return insert node attribute {'test'} {5} into \$item`

Q3: `delete node /site/people/person/profile/interest`

Q4: `for \$item in /site/people/person/name return replace value of node \$item with 'Max Mustermann'`

Query *Q1* replaces a single element, so it represents a small update operation in comparison to the complete document. Query *Q2* inserts a text attribute into each element. Therefore, this is a much more costly operation as many elements are affected. However, in terms of our replication system it should not make a difference, because both cases require to transmit the whole document, regardless of the percentage of the affected nodes. *Q3* deletes certain elements and *Q4* replaces the value of each `<name/>` element with `Max Mustermann`. The queries are intended to show a broad variety of different update operations with a different percentage of affected nodes in the data set.

The test is done using the traditional Client/Server infrastructure of BaseX 7.8.2 as comparison and the new replication architecture, with 2, 3, 5 and 10 members. Each members runs on a separate Amazon EC2 instance, all located in the EU region (Ireland). A distributor is running on a separate instance as well. We use the XMark data set with scale factor 1, so the input document is circa 112MB in size. The execution times for the four queries can be seen in Figure 3.2.

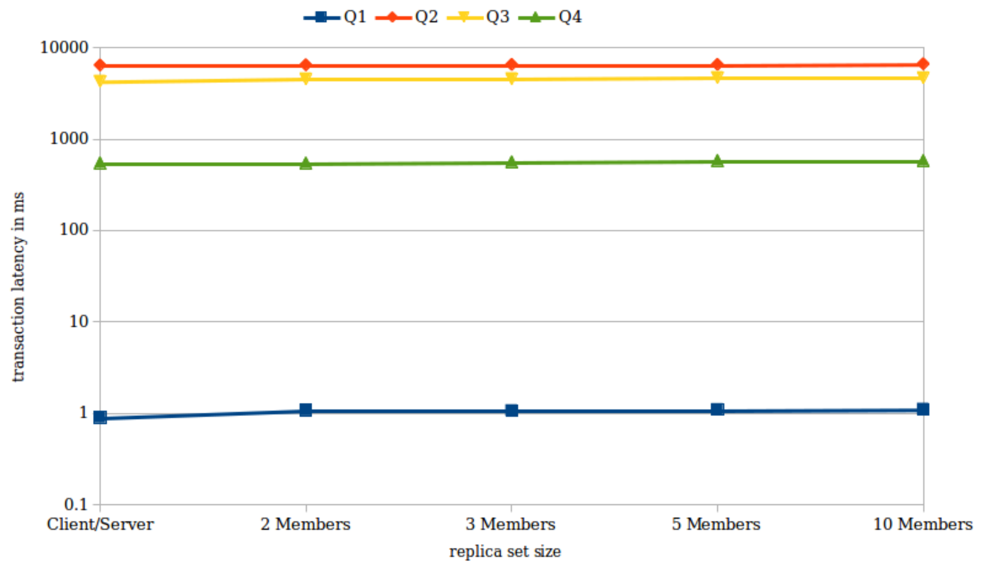


Figure 3.2: The transaction latency is shown for the Client/Server architecture and the replication architecture with 2, 3, 5 or 10 members. The *y* axis shows the query execution times for the four queries in milliseconds.

It is clearly visible that the queries have a quite distinctive runtime from each other, but all behave the same in terms of replication; the replication and the number of members within the replica set seem to have no significant affect on the transaction latency.

3.3 REPLICATION LATENCY

As we use a lazy replication approach, a query is committed before the updated data is replicated to the secondaries. Therefore, it is interesting to know how long it takes for an update to actually be applied on a secondary. We call the time between committing an updating query on the primary and the data being applied on a secondary *replication latency*.

We tested this on three different data sets, using the scaling factors 0.01, 0.1 and 1. We created a forth data set, again with scaling factor 1, but this time with a split count of 500, resulting in splitting up the data set in 138 individual files, ranging from 103KB to 1.4MB in size. We used a replica set with 3 members, so 2 secondaries. We measured the replication latency to both secondaries and took the average as measured data. All members are again located on separate EC2 instances, with a distributor located on a forth instance. All instances are in the EU region (Ireland). The results are shown in Figure 3.3.

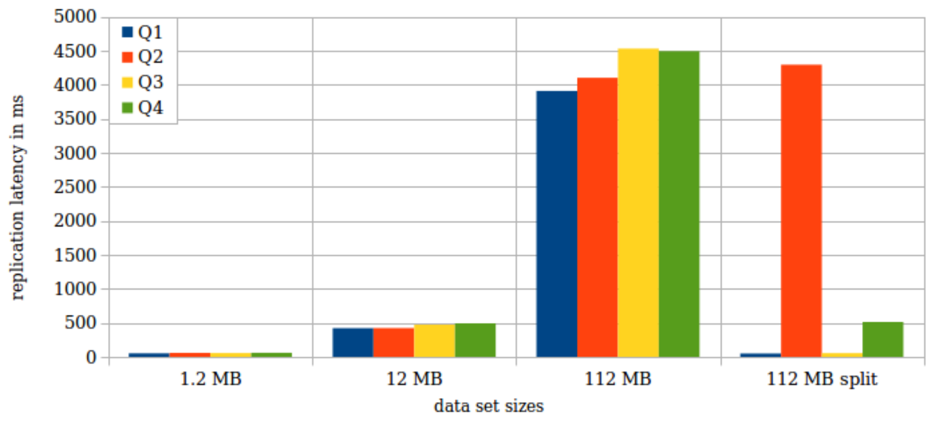


Figure 3.3: The replication latency is shown for the XMark data set with scaling factor 0.01, 0.1, 1 and 1 with a split count of 500. The measured time in milliseconds is the average time it took to apply the update for the four queries on the secondaries.

From the results we can draw several conclusions. First, if the document size increases, the replication latency increases linearly. Secondly, all queries on non-split data set have roughly the same replication latency. Both of this observations fit our model, which replicates complete documents.

The split document now shows the importance on how to model the data for this replication model. For queries *Q1* and *Q3* the replication latency is minimal, as now only a single document has to be serialized and transmitted to the secondaries. *Q2* still has roughly the same replication latency as without splitting the data, as almost all documents have to be updated. The updates of *Q4* affect a small portion of the documents, therefore much less data has to be transmitted.

It is quite obvious from this data, that a favorable data structure for our replication architecture is to have a collection of many small documents instead of one single big one. This fits well with many XML applications, as mostly a large number of small files are handled.

3.4 LEADER ELECTION PERFORMANCE

The primary within a replica set is the only member accepting updating operations. It is automatically elected using the leader algorithm described in Section 2.5. As we use a single primary (and therefore a possible single point of failure) we want to mitigate this by assuring that a new leader is promptly elected if the current primary fails or steps down.

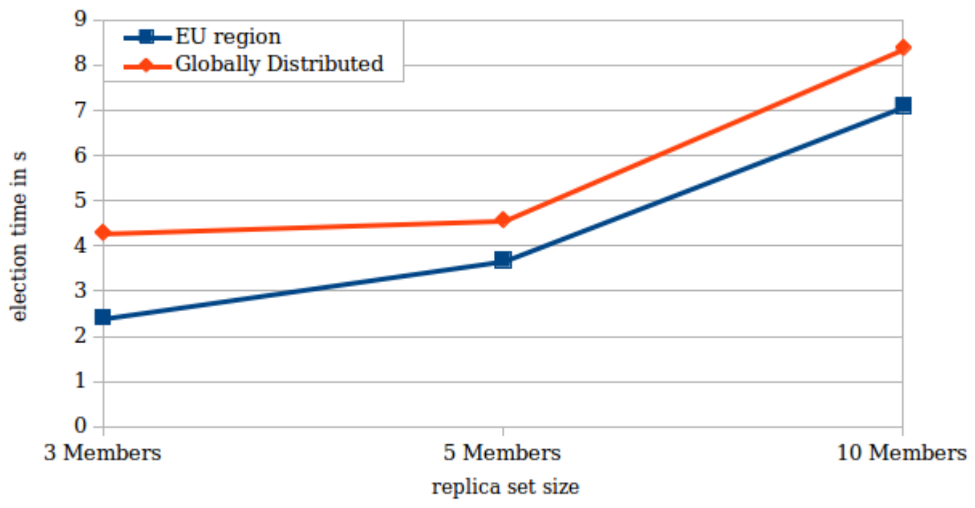


Figure 3.4: Two replica sets with different member regions are measured for their election time of a new leader.

Therefore, we measured the time it takes to elect a new leader. Our primary steps down voluntarily, because otherwise the uncertainty of the failure detection would also have to be taken into account and distort the result. We used a replica set with 3, 5 and 10 members, respectively. We tested this on two different member instances, both are based on Amazon

EC2. In the first case, all members are again located in the EU (Ireland) region. In the second test case, the members are distributed world wide; 3 members are located in EU (Ireland), US West (Oregon) and Asia Pacific (Tokyo) respectively. The 5 members replica set is also located in Asia Pacific (Sydney) and South America (São Paulo). The 10 member replica set has the same regions as the 5 member replica set, but each region houses two members.

We measured the time between the stepping-down of the former primary and until the primary is announced and fully functional again. The results are shown in Figure 3.4.

The data indicates that the election time increases linearly with the amount of members in the set, which is what we expected. It also shows that the difference between the locally deployed and the globally distributed replica set is rather small, i.e. a globally distributed replication seems quite feasible. This can have many scenarios, ranging from catastrophic failover management for outages of whole data-centers or regions, to moving data closer to the users and thereby reducing latency.

4 CONCLUSION

In this thesis, we presented several possible replication management strategies. Based on their provided properties and our design targets for replication in BaseX, mainly providing high availability and a high transaction latency, we choose an appropriate architecture. We decided to use a lazy primary copy approach, where a distinguished member of a replica set is responsible for all updating operations and updates are propagated after they were committed.

Other than the update management strategy, such a system consists of several importants components. We discussed security and showed how we can securely authenticate a user using SCRAM.

When a member of the replica set fails, we can detect this using a probabilistic accrual failure detector, which provides the possibility to detect failures in miscellaneous network settings with a low false positive rate. Therefore, a user can set appropriate suspicion level thresholds, to allow for different network latency.

We also discussed leader election algorithms to elect a primary within our replica system. As our primary is the only member accepting updates, it is crucial that this election is done promptly to avoid downtimes. We presented the bully algorithm and a modified version of the bully algorithm and presented our own modification, to allow for different message delivery guarantees and restricting to a limited eligibility of members.

We measured the performance of our system and can conclude that the system performs well in many scenarios. The transaction latency is basically identical to the Client/Server architecture, but offers the extended service level of high availability. The transaction latency is good for systems with small updating documents, which ensures that clients can see a reasonable recent version of the data when reading data items from secondaries. The used election algorithm provides a fast election of a new primary, so that the amount of time a replica set is in read-only mode is minimized.

As the problem itself is open-ended, i.e. improvements can nearly always be made by modifying the replication to different design goals, many possible future problems present themselves. We already begun implementing an approach where only the AUC is replicated instead of complete documents. Also, a freshness constraint on the replicated data on the secondaries might be useful for many applications. It might also be beneficial to provide additional update management strategies, e.g. an eager replication, for different application requirements.

So while there is still room for improvement, we presented an adaptable and reliable architecture for a replication system within BaseX. It performs well in many use cases and allows BaseX to offer an additional service level.

ACKNOWLEDGEMENTS

I would like to thank the whole BaseX team for providing me the opportunity to not only work academically on BaseX, but to also work with the product in real-world projects, incorporating interesting technologies and encountering challenging projects. A special thanks goes to Dr. Christian Grün who mentored this thesis and is always readily available for fruitful discussions and important insights.

I am also very grateful that Prof. Dr. Marc H. Scholl and Prof. Dr. Marcel Waldvogel agreed to serve as referees for this thesis.

BIBLIOGRAPHY

[03] *DB2 Replication Guide and Reference*, 2003.

[13] *MarkLogic Server, Database replication guide*, 2013. [Online]. Available: https://docs.marklogic.com/guide/database-replication.pdf (visited on 08/01/2014).

[14a] *eXist-db Replication and Messaging*, 2014. [Online]. Available: http://exist-db.org/exist/apps/doc/replication.xml (visited on 08/24/2014).

[14b] *MongoDB Replication Concepts*, 2014. [Online]. Available: http://docs.mongodb.org/manual/core/replication/ (visited on 08/10/2014).

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *ACM SIGMOD Record*, ACM, vol. 24, 1995, pp. 1–10.

[BFG+06] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, "Relaxed-currency serializability for middle-tier caching and replication," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ACM, 2006, pp. 599–610.

[BG83] P. A. Bernstein and N. Goodman, "The failure and recovery problem for replicated databases," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, ACM, 1983, pp. 114–122.

[Bla10] B. Black, *Introduction to Cassandra: Replication and Consistency*, 2010.

[Bre00] E. A. Brewer, "Towards robust distributed systems," in *Principles of Distributed Computing*, (Invited Talk), Portland, Oregon, Jul. 2000.

[Bre12] E. Brewer, "CAP twelve years later: How the"rules"have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[BSW79] P. A. Bernstein, D. W. Shipman, and W. S. Wong, "Formal aspects of serializability in database concurrency control," *Software Engineering, IEEE Transactions on*, no. 3, pp. 203–216, 1979.

[CT96] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[Era13] J. Erat, "Fine granular locking in xml databases," Bachelor thesis, University of Konstanz, 2013. DOI: urn:nbn:de:bsz:352-235049.

[Gar82] H. Garcia-Molina, "Elections in a distributed computing system," *Computers, IEEE Transactions on*, vol. 100, no. 1, pp. 48–59, 1982.

[GF03] J. Garmany and R. Freeman, *Oracle Replication: Snapshot, Multi-master & Materialized Views Scripts (Oracle In-Focus)*. Rampant TechPress, 2003.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *ACM SIGMOD Record*, ACM, vol. 25, 1996, pp. 173–182.

[Grü10] C. Grün, "Storing and querying large xml instances," PhD thesis, University of Konstanz, 2010. DOI: urn:nbn:de:bsz:352-opus-127142.

[HDYK04] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The φ accrual failure detector," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, IEEE, 2004, pp. 66–78.

[KCK97] J. Klensin, R. Catoe, and P. Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response," RFC 2195, September, Tech. Rep., 1997.

[Kir13] L. Kircher, "Polishing Structural Bulk Updates in a Native XML Database," Master thesis, University of Konstanz, 2013. DOI: urn:nbn:de:bsz:352-154882.

[KKLK09] A. Krug, D.-M. U. Krönert, D.-D. M. Lörzer, and K. Küspert, *Untersuchung verschiedener Replikationsverfahren unter IBM DB2 LUW am konkreten Beispiel der Digitalen Bibliothek der Friedrich-Schiller-Universität Jena und des Freistaates Thüringen*, 2009.

[LKPJ05] Y. Lin, B. Kemme, M. Patiño-Martinez, and R. Jimenez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ACM, 2005, pp. 419–430.

[MMM04] Q. E. K. Mamun, S. M. Masum, and M. A. R. Mustafa, "Modified bully algorithm for electing coordinator in distributed systems.," *WSEAS Transactions on Computers*, vol. 3, no. 4, pp. 948–953, 2004.

[Ner08] L. Nerenberg, "The CRAM-MD5 SASL Mechanism," SASL Working Group, July, Tech. Rep., 2008.

[NMMW10] C. Newman, A. Menon-Sen, A. Melnikov, and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM)," RFC 5802, Tech. Rep., Jul. 2010.

[ÖV11] T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, ser. Computer science. Springer, 2011.

[Ret12] A. Retter, "Restful xquery - standardised xquery 3.0 annotations for rest," 2012. [Online]. Available: http://www.adamretter.org.uk/papers/restful-xquery_january-2012.pdf.

[SPTU07] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, "A new adaptive accrual failure detector for dependable distributed systems," in *Proceedings of the 2007 ACM symposium on Applied computing*, ACM, 2007, pp. 551–555.

[SWK+02] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "Xmark: a benchmark for xml data management," in *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB Endowment, 2002, pp. 974–985.

[Wei10] A. Weiler, "Client-/Server-Architektur in XML Datenbanken," Master thesis, University of Konstanz, 2010. DOI: urn:nbn:de:bsz:352-opus-123668.

[WY05] X. Wang and H. Yu, "How to break md5 and other hash functions," in *Advances in Cryptology–EUROCRYPT 2005*, Springer, 2005, pp. 19–35.

LIST OF FIGURES

LIST OF ALGORITHMS