Consistency and Availability of Data in Replicated NoSQL Databases

Tadeusz Pankowski
Institute of Control and Information Engineering, Poznań University of Technology, Poznań, Poland
tadeusz.pankowski@put.poznan.pl

Keywords: Replicated Data Repositories, Strong Consistency, Eventual Consistency, Consensus Quorum Algorithms.

Abstract: Highly distributed NoSQL key-value databases have been designed to meet the needs of various applications in the field of social networks, Web search and e-commerce. High availability, fault tolerance and scalability (i.e. *Quality of Service*, QoS) of such systems is in conflict with the strong consistency (*Quality of Data*, QoD). Thus, a new functionality of services, balancing QoS and QoD, is needed. This functionality is referred as *Service Level Agreements*, SLA) (Terry et al., 2013). We present a system Lorq as a proposal of a solution to this problem. Lorq is a consensus quorum based algorithm for NoSQL data replication. In this paper, we discuss how different consistency levels are guaranteed by Lorq.

1 INTRODUCTION

Recently, we observe a rapid development of modern storage systems based on Internet services and cloud computing technologies (Terry et al., 2013). Examples of this kind of applications range from social networks (Lakshman and Malik, 2010), Web search (Corbett et al., 2013; Cooper et al., 2008) to e-commerce systems (DeCandia et al., 2007). Data management in such systems is expected to support needs of a broad class of applications concerning different performance, consistency, fault tolerance, and scalability (Abadi, 2012). Development of this new class of systems, referred to as NoSQL systems, differs from conventional relational SQL systems in the following aspects: (a) the data model is usually based on a NoSQL key-value model; (b) data management is focus on intensive simple write/read operations instead of ACID transactions processing; (c) NoSQL system architecture is a multi-server data replication architecture, which is necessary to meet needs concerning the performance, scalability and partition tolerance; (d) a various kinds of consistencies is offered, from the strong consistency to weak consistencies (eventual consistency).

In replicated systems, the following three features influence the design, deployment and usage of the system: consistency (C), availability (A), and partition tolerance (P). Partition happens when in result of a crash, a part of the network is separated from the rest. According to the CAP theorem (Gilbert and Lynch, 2012), all of them cannot be achieved. Because the partition tolerance is the necessary condition expected by the users, the crucial issue is then the trade-off between consistency and availability/latency.

Applications in such systems are often interested in possibility to declare their consistency and latency priorities (Terry et al., 2013). In general, except from strong consistency, a user may expect a weaker kind of consistencies, which are referred to as *eventual consistencies*. This is similar to declaring isolation levels in conventional SQL databases. In some companies, the price that clients pay for accessing data repositories depends both on the amount of data and on its freshness (consistency). For example, Amazon charges twice as much for strongly consistent reads as for eventually consistent reads in DynamoDB (Amazon DynamoDB Pricing, 2014). According to (Terry et al., 2013), applications *"should request consistency and latency that provides suitable service to their customers and for which they will pay"*. For example, Pileus system (Terry et al., 2013) allows applications to declare their consistency and latency priorities.

Novelties of this paper are as follows.

1. We propose and discuss a method and an algorithm for replicating NoSQL data. The algorithm is called Lorq (*LOg Replication based on consensus Quorum*). The main features of Lorq are the following: (a) data replication is realized by means of replicating logs storing update operations (treated as a single-operation transactions); (b) the processing and replication strategies guarantees that eventually all operations in

each replica are executed in the same order and no operation is lost.

2. A special attention is paid to different kinds of consistency, which can be guaranteed by the system.
3. In comparison to Raft (Ongaro and Ousterhout, 2014), we propose a different way for choosing a leader in Lorq. In particular, the procedure is based on timestamps and servers priorities. It simplifies the election procedure significantly.

In design and implementation of Lorq system, we have used modern software engineering methods and tools oriented to asynchronous and parallel programing (Asynchronous Programming with Async and Await, 2014).

The outline of this paper is as follows. The next sections reviews problems of NoSQL data replication. The Lorq algorithm is presented in Section 3. Some methods for achieving strong and eventual consistency in Lorq, are discussed in Section 4. Finally, Section 5 concludes the paper.

2 REPLICATION OF NoSQL DATA

In modern world-wide data stores, a number of goals should be met:

- *Scalability.* The system must be able to take advantage of newly added servers (nodes) for replication and redistribution of data. The aim of adding new hardware is to support large databases with very high request rates and very low latency.
- *Availability.* It is guaranteed that each request eventually receives a response. The case when a response is too late, is often treated as the lack of response at all, and the system can be understood to be unavailable.
- *Partitioning Tolerance.* Due to communication failures, the servers in the system can be partitioned into multiple groups that cannot communicate with one another.
- *Consistency.* In conventional databases, a consistent database state is a a state satisfying all consistency constraints. In replicated databases, consistency means the equality between answers to queries issued against different servers in the system. In the case of *strong consistency* all answers are identical and up-to-date (such as ACID transactions (Bernstein et al., 1987)). In the case of a *weak consistency*, some answers can be stale, however, the staleness of answers should be under control, and in the lack of updates, all answers converge to be identical. Then we say about *eventual consistency* (Vogels, 2009).

There is a fundamental trade-off between consistency (*Quality of Data*), and latency/availability and partition tolerance (*Quality of Service*).

2.1 NoSQL Data Model

Modern NoSQL data stores manage various variants of key-value data models (Cattell, 2010) (e.g., PNUTS (Cooper et al., 2008), Dynamo (DeCandia et al., 2007), Cassandra (Lakshman and Malik, 2010), and BigTable (Chang et al., 2008).

The main NoSQL data structure is a key-value pair (x, v), where x is a unique identifier, and v is a value (Cattell, 2010). The value component could be: *structureless* – then the value is a sequence of bytes and an application is responsible for its interpretation, or *structured* – the value is a set of pairs of the form $A : v$, where A is an attribute name, and v is a value, in general, values can be nested structures. Further on in this paper, we will assume that values in data objects are structured.

2.2 Strategies of Data Replication

Data replication can be realized using:

1. *State Propagation.* The leader (a master server) propagates its latest state to workers (slaves, followers), which replace their current states with the propagated data.
2. *Operation Propagation.* The leader propagates sequences of operations. It must be guaranteed that these operations will be applied by all workers to their states in the same order.

In both strategies, states of all database replicas will be eventually consistent. However, the freshness of these states is different and depends on the time needed for the propagation. Some serious problems follow from possible crashes of servers or communications between the servers. For instance, it may happen that the leader crashes before propagating the data or operations. In such cases, there must be means to prevent losing this data. From the operational point of view, the replication must take advantages from asynchronous and parallel processing in order to guarantee the required efficiency.

2.3 Consensus Quorum Algorithms

In a quorum-based data replication, it is required that an execution of an operation (i.e., a propagation of an update operation or a read operation) is committed if and only if a sufficiently large number of servers acknowledge a successful termination of this operation (Gifford, 1979).

In a quorum consensus algorithm, it is assumed that: N is a number of servers storing copies of data (replicas), R is an integer called *read quorum*, meaning that at least R copies were successfully read; W is an integer called *write quorum*, meaning that propagations to at least W servers have been successfully terminated. The following relationships hold between N, R and W:

$$W > N/2, \qquad (1)$$

$$R + W > N. \qquad (2)$$

To commit a read operation, a server must collect the read quorum, and to commit a write operation must collect the write quorum. Condition (1) guarantees that the majority of copies is updated, and (2) that among read copies at least one is up-to-data.

The aim of consensus algorithms is to allow a collection of servers to process users' commands (updates and reads) as long as the number of active servers is not less than $max\{W,R\}$. It means that the system is able to survive failures of some of its servers.

An algorithm based on quorum consensus works properly if a majority of servers is accessible and the most current version of data can be always provided to a client (i.e., it guarantees the strong consistency). Note that for $N = 3, R = 2, W = 2$ the system tolerates only one failure, and to tolerate at most two failures, we can assume $N = 5, R = 3, W = 3$. Let p be the probability of a server failure, then the probability that at least n servers will be available is equal to

$$P(N,p,n) = \sum_{k=0}^{n} p^k (1-p)^{N-k},$$

in particular, $P(5, 0.02, 3) = 0.99992$.

3 LOG REPLICATION BASED ON CONSENSUS QUORUM

During last decade, the research on consensus algorithms is dominated by Paxos algorithms (Gafni and Lamport, 2000; Lamport, 2006). Lately, a variant of Paxos, named Raft (Ongaro and Ousterhout, 2014), was presented as a consensus algorithm for managing a replicated log. Lorq is based on ideas underlying Paxos and Raft, and includes such steps as: (1) leader election; (2) log replication, execution and commitment of update operations; (3) realization of read operations on different consistency levels (Pankowski, 2015).

3.1 Architecture

The architecture of Lorq (Figure 1), like in Raft (Ongaro and Ousterhout, 2014), is organized having in mind: operations, clients, and servers occurring in the system managing data replication.

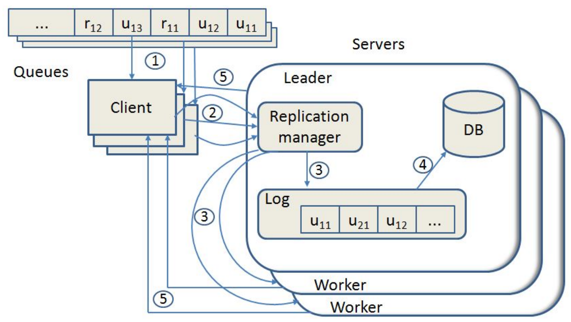

Figure 1: Architecture of Lorq system. There are update and read operations in queues. Update operations are delivered (1) by clients from queues to one leader (2). A replication module delivers them to leader's log and to logs at all workers (3). Sequences of operations in all logs tend to be identical and are applied in the same order to databases (4). States of all databases also tend to be identical (eventually consistent). A client may read (5) data from any server.

Operations. We distinguish three *update* operations: *set*, *insert*, and *delete*, and one *read* operation. In order to informally define syntax and semantics of operations, let us assume that there is a NoSQL database $DB = \{(x, \{A : a\}, t)\}$ storing a data object $(x, \{A : a\}, t)$, where t is the timestamp of the operation that as the last has updated the data. Operations are specified as follows:

- $set(dataId, dataVal, t)$ – the value of an existing data with identifier $dataId$ is set to $dataVal$, e.g., $set(x, \{A : b\}, t_1)$ and $set(x, \{B : b\}, t_2)$ against DB changes its state to $\{(x, \{A : b\}, t_1)\}$ and $\{(x, \{B : b\}, t_2)\}$, respectively.
- $ins(dataId, dataVal, t)$ – data $(dataId, dataVal)$ is inserted, or the value of the existing data object $dataId$ and its timestamp are modified accordingly, e.g.: successive execution of $ins(x, \{B : b\}, t_1)$ and $ins(y, \{C : c\}, t_2)$ against DB, changes its state to $DB = \{(x, \{A : a, B : b\}, t_1), (y, \{C : c\}, t_2)\}$.
- $del(dataId, dataVal, t)$ – the existing data object $(dataId, dataVal)$ is deleted, or the value of the existing data object $dataId$ is modified, e.g.: successive execution of $del(x, \{B : b\}, t_1)$ and $del(y, \{C : c\}, t_2)$ against the DB state considered above, gives $DB = \{(x, \{A : a\}, t_1)\}$.
- $read(dataId, t)$ – reads the key-value data with identifier $dataId$.

Queues. Operations from users are serialized and put in queues. An operation is represented as follows:

- any update operation, $opType(dataId, dataVal)$, $opType \in \{set, ins, del\}$, is represented as a tuple $(opId, clId, stTime, opType, dataId, dataVal)$;
- any read operation, $read(dataId)$ – as a tuple $(opId, clId, stTime, dataId)$,

operation identifier ($opId$), *client identifier* ($clId$), and *start timestamp* ($stTime$) are determined, when the operation is delivered by a client to the leader.

Clients. Each queue is served by one *client*.
Servers. A server maintains one replica of a database along with the log related to this replica, and runs the software implementing Lorq protocols. One server plays the role of the *leader* and the rest roles of *workers*. The state of each server is characterized by the server's log and database. The following variables describe the state of a server:

- $lastIdx$ – last log index (the highest position storing an operation),
- $lastOpTime$ – the $stTime$ stored at $lastIdx$ position,
- $lastCommit$ – the highest log index storing a committed operation,
- $currentLeader$ – identifier of the current leader, 0 – the leader is not elected,
- $lastActivity$ – the observed latest activity time of the leader.

3.2 States and Roles

Lorq system can be in one of the following three states (Figure 2):

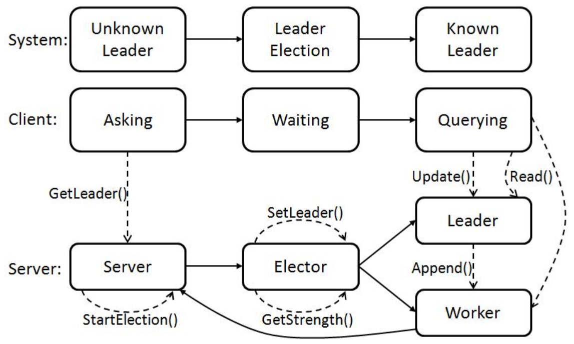

Figure 2: System states, client and server roles in Lorq. A client can ask about leader, wait for result of election or send queries (update or read). A server can play undefined role (unqualified server), take part in leader election (as elector), or act as leader or worker.

1. UnknownLeader – no leader is established in the system (when the system starts and a short time after a leader failure).
2. LeaderElection – election of a leader is in progress.
3. KnownLeader – a leader is known in the system.

Actors of the system, i.e., clients and servers, can play in those states specific roles.

3.2.1 Client in Asking Role

Initially, and when the system is in UnknownLeader state, a client asynchronously sends the request *GetLeader* to all servers. In response, each available server can return:

- 0 – if a leader can not be elected because the required quorum cannot be achieved (the system is unavailable);
- *currentLeader* – identifier of the leader that is either actually known to the server or has been chosen in reaction to the *GetLeader* request.

3.2.2 Client in Waiting Role

After issuing *GetLeader* request, the client plays the Waiting role. Next, depending on the reply, changes its role to Asking or Querying.

3.2.3 Client in Querying Role

A client reads from a queue the next operation, or reads again a waiting one if necessary, and proceeds as follows:

- determines values for $opId$, $clId$, and $stTime$;
- an update operation is sent to the leader using an $Update$ command
$$Update(opId, clId, stTime, \\ opType, dataId, dataVal); \quad (3)$$
and the operation is treated as a *"waiting"* one;
- a *read* operation is handled according to the required consistency type (*ConType*) (see Section 4), and is sent using a *ReadConType* command
$$ReadConType(opId, clId, stTime, dataId). \quad (4)$$

If the $Update$ command replays committed, the corresponding operation is removed out from the queue. If the timeout for responding elapses, the client changes its role to Asking (perhaps the leader failed and the operation will be next reissued to a new leader).

3.2.4 Server in Server Role

A server plays the Server role when the system starts, and when a worker detects that election timeout elapses without receiving any message from the leader (that means the leader failure). A server in Server role starts an election issuing the command $StartElection$ to all servers. Next, the systems goes to the LeaderElection state, and all servers change their roles to Elector.

3.2.5 Server in Elector Role

The community of servers attempt to chose a leader among them. The leader should be this server that has the highest value of $lastCommit$, and by equal $lastCommit$, the one with the highest identifier (or satisfying another priority criterion). The election proceeds as follows:

1. A server collects $lastCommit$ values from all servers (including itself), and creates a decreasingly ordered list of pairs $(lastCommit, srvId)$.
2. If the list contains answers from majority of servers, then the $srvId$ from the first pair is chosen as the leader and its value is substituted to $currentLeader$.
3. The $currentLeader$ is announced as the leader. The procedure guarantees that all servers will choose the same leader.

Next, the system goes to the KnownLeader state, the server chosen as the leader changes its role to Leader, and the remainder servers to Worker.

3.2.6 Server in Leader Role

The leader acts as follows:

1. After receiving $Update$ command (3), a leader issues asynchronously to each server (including itself) the command appending the operation to server's log

   $$\begin{aligned}&Append(opId, clId, stTime, opType, dataId,\\ &dataVal, lastIdx, lastOpTime, lastCommit)\end{aligned} \tag{5}$$

   The process can return with an exit code indicating:

   - success – the entry was successfully appended to the server's log,
   - inconsistency – the server's log is inconsistent with the leader's one. Then a procedure recovering consistency is carried out. After this, the leader retries sending the $Append$ command. It is guaranteed, that after a finite number of repetition, the command returns with success (unless the worker fails).

   If the number of servers responding success is at least equal to the write quorum (W), then the leader sends asynchronously to all these servers (also to itself) the request to execute the appended operation and to commit it. Next, without waiting for responses, replays committed to the client. If a worker does not respond to $Append$ (because of crash, delay, or lost of network packet), the leader retries $Append$ indefinitely (even after it has responded to the client). Eventually, all workers store the appended entry or a new election is started.
2. If the leader activity timeout elapses (given by *ActivityTimeout* that must be less than *ElectionTimeout*), the leader sends

   $$Append(lastIdx, lastOpTime, lastCommit)$$

   to each worker, i.e., $Append$ command with empty data-part (so-called *heartbeat*), to confirm its role and prevent starting a new election. The response to this message is the same as to regular *Append*. In particular, this is important for checking log consistency, especially for restarted workers.
3. When a new leader starts, then:
   - There can be some uncommitted entries in the top of the leader's log, i.e., after the *lastCommit* entry, ($lastCommit < lastIdx$). These entries must be propagated to workers by *Append* command in increasing order. If a delivered log entry is already present in worker's log, it is ignored.
   - Some *"waiting"* operations in a queue, i.e., denoted as already sent to a leader, could not occur in the leader's log (the reason is that they have been sent to a previous leader and that leader crashed before reaching the write quorum). Then these operations must be again sent by the client from the queue to the newly elected leader.
   - After the aforementioned two operations have been done, the client starts delivering the next update operation from the queue.

A server plays the Leader role until it fails. After recovery, it plays a role of a Worker.

3.2.7 Server in Worker Role

Let a worker receive an *Append* command (possibly with empty data-part) of the form (5), where leader's parameters are denoted as: *lastIdxL*, $lastOpTimeL$, *lastCommitL*. Then:

1. If $lastIdx = lastIdxL$ and $lastOpTime = lastOpTimeL$, then: if the data-part is not empty, then append the new entry; execute and commit all uncommitted entries at positions less than or equal to $lastCommitL$ in increasing order; reply success.
2. If $lastIdx < lastIdxL$, then reply inconsistency. The worker expects that the leader will decide to send all missing entries.
3. If $lastIdx = lastIdxL$ and $lastOpTime < lastOpTimeL$, then delete $lastIdx$ entry and reply inconsistency.
4. If $lastIdx > lastIdxL$ then delete entries at positions greater than or equal to $lastIdxL$ and reply inconsistency.

A server plays the Worker role until its failure or failure of the leader – then it returns to the Server role.

3.3 Example

Let us assume that there are five servers in the system ($N = 5$) (Figure 3), and the write quorum is three ($W = 3$). By "+" we denote committed entries, and by "?" the waiting ones (i.e., a client waits for committing the sent operation). Let us consider three steps in the process of replication presented in Figure 3.

Step 1. Operations a and b are already committed, bat c and d are waiting, i.e., their execution in the system is not completed. S_1 is the leader that propagated c and d to S_3, and after this failed. We assume that also S_5 failed. There are two uncommitted operations, c and d, in the top of S_3, these operations are also denoted as waiting in the queue.

Step 2. In the next election, S_3 has been elected the leader. The client receives information about the new leader and sends the waiting operations c and d to him. Since c and d are already in the S_3's log, the sending is ignored. S_3 propagates c and d to available workers S_2, and S_4. The write quorum is reached, so they are executed and committed. Next, S_3 receives from a client the operation e, and fails.

Step 3. Now, all servers are active and S_4 is elected the leader. The client receives information about the new leader and sends the waiting operation e to S_4. Next, the leader propagates e as follows:

- Propagation to S_1: e is appended at fifth position. Since leader's *lastCommit* is 4, thus c and d are executed in the S_1's database and committed.
- Propagation to S_2: e is successfully appended at fifth position.
- The leader (S_4) recognizes that the write quorum is already reached. Thus: asynchronously sends to S_1, S_2 and to itself the command to execute and commit e; replies committed to the client; continue propagations of e to S_3 and S_5.
- Propagation to S_3: e is already in S_3's log, so the append is ignored. Along with e, the *lastCommit* equal to 5 is sent to S_3. Because of this, e can be executed and committed.
- Propagation to S_5: there is an inconsistency between logs in S_4 and S_5. We are decreasing $lastIdx$ in S_4, $lastIdx := lastIdx - 1$, as long as the coherence between both logs is observed. This happens for $lastIdx = 2$. Now, all entries at positions $3, 4$ and 5 are propagated to S_5. Moreover, operations c, d and e are executed in S_5's database and committed.

In the result, all logs will be eventually identical.

4 CONSISTENCY MODELS FOR REPLICATED DATA

Some database systems provide strong consistency, while others have chosen eventual consistency in order to obtain better performance and availability. In this section, we discuss how these two paradigms can be achieved in a data replication system based on Lorq model.

4.1 Strong Consistency

Strong consistency guarantees that a read operation returns the value that was last written for a given data object. In other words, a read observes the effects of all previously completed updates (Terry, 2013; Terry et al., 2013). Since Lorq system is based on a consensus quorum algorithm, it is guaranteed that a majority of servers has successfully accomplished execution of all committed update operations in their databases. To guarantee strong consistency, the number of servers we should read from is at least equal to the read quorum (R). Next, from the set of read data values we chose the value with the latest timestamp.

However, reading from many servers is costly, thus, to improve performance, w can restrict ourselves to reading from one server only. So, if we need to have high availability, we can chose so-called *eventual consistency*.

4.2 Eventual Consistency

Eventual consistency is a theoretical guarantee that, *"if no new updates are made to the object, eventually*

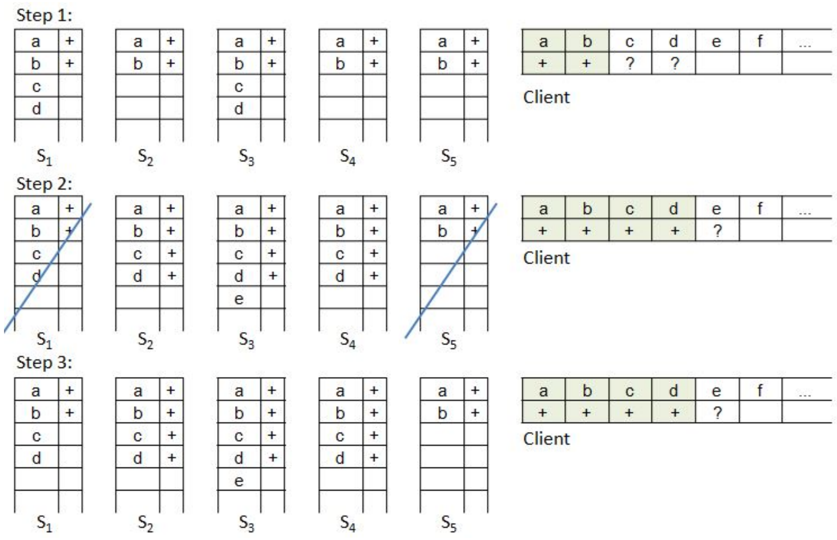

Figure 3: Illustration to a scenario of managing data replication in Lorq.

all accesses will return the last updated value" (Vogels, 2009). In this kind of consistency, reads return the values of data objects that were previously written, though not necessarily latest values. An eventual consistency guarantee can be requested by each read operation (or by a session), thereby choosing between availability, costs and data freshness. Next, we will discuss realization of some eventual consistency in Lorq system. We will consider four classes of eventual consistency: *consistent prefix*, *bounded staleness*, *monotonic reads*, and *read-my-writes* (Terry, 2013).

4.2.1 Consistent Prefix

A reader is guaranteed to see a state of data that existed at some time in the past, but not necessarily the latest state. This is similar to so-called *snapshot isolation* offered by database management systems (Terry, 2013).

In Lorq, this consistency is realized by reading from any server. It is guaranteed that databases in all servers store past or current states, and that these states were up-to-date sometime in the past. In general, the level of staleness is not known, since the read may be done from a server which is separated from others by communication failure.

4.2.2 Bounded Staleness

A reader is guaranteed that read results are not too out-of-data. We distinguish *version-based staleness* and *time-based staleness* (Golab et al., 2014).

In Lorq, the staleness of a read data x can be determine as follows. We read x from an active server's (worker or leader) database. Let $read(x) = (x, v, t_{db})$, where t_{db} is the timestamp of the last operation that updated x. Next, we find in the log the set of operations, which are intended to update x and have timestamps greater than t_{db}. Let O be the set of such operations, and

- $t_{lastop} = max\{t_o \mid o \in O\}$, i.e., t_{lastop} is the largest timestamp of operations in O; if O is empty, we assume $t_{lastop} = t_{db}$;
- $k = count(O)$, i.e., k denotes cardinality of O.

Then the operation $read(x)$ has the *version-based staleness* equal to k, and *time-based staleness* equal to $\Delta = t_{lastop} - t_{db}$.

In this way, we can assess whether the required freshness is fulfilled. If not, we apply to the auxiliary database $AuxDB = \{(x, v, t_{db})\}$ as many operations as needed from O, in increasing order, to guarantee the acceptable staleness. Finally, we perform $read(x)$ against the final state of $AuxDB$.

4.2.3 Monotonic Read

A reader is guaranteed to see data states that is the same or increasingly up-to-date over time (Terry, 2013). Assume that a client reads a data object x, $read(x) = (x, v, t)$. Then the client updates x with (x, v', t'), where $t' > t$. If next the user issues another read to this data object, then the result will be: either (x, v, t) or (x, v', t'), but never a state (x, v'', t''), where $t'' < t$.

In Lorq, the monotonic read is guaranteed by so-called *session guarantee* meaning that all read operations of a client are always addressed to the same worker. Then, the session terminates when the worker fails. Note, that if we would read from different workers, we could be unlucky and read from a delayed or from a server belonging to a separated partition.

4.2.4 Read-my-writes

It is guaranteed that effects of all updates performed by the client, are visible to the client's subsequent reads. If a client updates data and then reads this data, then the read will return the result of this update (or an update that was done later) (Terry, 2013).

In Lorq, the *read-my-writes* consistency is guaranteed in the way similar to that of bounded staleness.

- let $read(x)$ be a read operation issued by a client with identifier $clId$, and let $read(x) = (x,v,t)$ in an arbitrarily chosen server;
- let O_{clId} be the set of operations stored on server's log, and issued by $clId$;
- if O_{clId} is empty, then $read(x)$ is the result;
- otherwise, apply all operations from O_{clId}, in increasing order, to the auxiliary database $AuxDB = \{(x,v,t)\}$; next, perform $read(x)$ against the final state of $AuxDB$.

5 CONCLUSIONS AND FUTURE WORK

We proposed a new algorithm, called Lorq, for managing replicated data based on the consensus quorum approach. Lorq, like another consensus quorum algorithms, is devoted for data-centric applications, where the trade-off between consistency, availability and partition tolerance must be taken into account. The implementation of Lorq make advantages of the modern software engineering methods and tools oriented to asynchronous and parallel programing. In future work, we plan to extend the Lorq algorithm to take advantages of so-called *replicated data types* (Burckhardt et al., 2014; Shapiro et al., 2011). We plan also to prepare and conduct some real-system experiments. This research has been supported by Polish Ministry of Science and Higher Education under grant 04/45/DSPB/0136.

REFERENCES

Abadi, D. (2012). Consistency tradeoffs in modern distributed database system design. *IEEE Computer*, 45(2):37–42.

Amazon DynamoDB Pricing (2014). http://aws.amazon.com/dynamodb/pricing.

Asynchronous Programming with Async and Await (2014). http://msdn.microsoft.com/en-us/library/hh191443.aspx.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company.

Burckhardt, S., Gotsman, A., Yang, H., and Zawirski, M. (2014). Replicated data types: specification, verification, optimality. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14*, pages 271–284.

Cattell, R. (2010). Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., and *et al.* (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2).

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., and *et al.* (2008). PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288.

Corbett, J. C., Dean, J., Epstein, M., and *et al.* (2013). Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8.

DeCandia, G., Hastorun, D., Jampani, M., and *et al.* (2007). Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220.

Gafni, E. and Lamport, L. (2000). Disk Paxos. In *Distributed Computing, DISC 2000, LNCS* 1914, pages 330–344. Springer.

Gifford, D. K. (1979). Weighted Voting for Replicated Data. In *ACM SIGOPS 7th Sym. on Op. Systems Principles, SOSP'79*, pages 150–162.

Gilbert, S. and Lynch, N. A. (2012). Perspectives on the CAP Theorem. *IEEE Computer*, 45(2):30–36.

Golab, W., Rahman, M. R., AuYoung, A., Keeton, K., and Li, X. S. (2014). Eventually consistent: Not what you were expecting? *Commun. ACM*, 57(3):38–44.

Lakshman, A. and Malik, P. (2010). Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40.

Lamport, L. (2006). Fast Paxos. *Distributed Computing*, 19(2):79–103.

Ongaro, D. and Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*, pages 305–319.

Pankowski, T. (2015). A consensus quorum algorithm for replicated NoSQL data. In *BDAS 2015*, Communications in Computer and Information Science, pages 1–10. Springer.

Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. (2011). Conflict-free replicated data types. Lecture Notes in Computer Science 6976, pages 386–400.

Terry, D. (2013). Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89.

Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., and *et al.* (2013). Consistency-based service level agreements for cloud storage. In *ACM SIGOPS, SOSP'13*, pages 309–324.

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