QuAD: A Quorum Protocol for Adaptive Data Management in the Cloud

Ilir Fetai
Technology Management
Swiss Federal Railways (SBB)
ilir.fetai@sbb.ch

Alexander Stiemer
Databases and Information Systems
University of Basel, Switzerland
alexander.stiemer@unibas.ch

Heiko Schuldt
Databases and Information Systems
University of Basel, Switzerland
heiko.schuldt@unibas.ch

***Abstract*—More and more companies move their data to the Cloud which is able to cope with the high scalability and availability demands due to its pay-as-you-go cost model. For this, databases in the Cloud are distributed and replicated across different data centers. According to the CAP theorem, distributed data management is governed by a trade-off between consistency and availability. In addition, the stronger the provided consistency level, the higher is the generated coordination overhead and thus the impact on system performance. Nevertheless, many OLTP applications demand strong consistency and use ROWA(A) for replica synchronization. ROWA(A) protocols eagerly update all (or all available) replicas and thus generate a high overhead for update transactions. In contrast, quorum-based protocols consider only a subset of sites for eager commit. This reduces the overhead for update transactions at the cost of reads, as the latter also need to access several sites. Existing quorum-based protocols do not consider the load of sites when determining the quorums; hence, they are not able to adapt at run-time to load changes. In this paper, we present QuAD, an adaptive quorum-based replication protocol that constructs quorums by dynamically selecting the optimal quorum configuration w.r.t. load and network latency. Our evaluation of QuAD based on Amazon EC2 shows that it considerably outperforms both static quorum protocols and dynamic protocols that neglect site properties in the quorum construction process.**

***Index Terms*—distributed data management; replication.**

I. Introduction

Typical OLTP applications need highly scalable and available database management systems (DBMS) in order to satisfy business requirements[1]. Due to availability and scalability demands, databases are usually distributed and replicated across different data centers [1]. However, distributed database systems (DDBS) face a trade-off between *A*vailability, *C*onsistency and tolerance to network *P*artitions. According to the CAP theorem [2], [3], DDBS can jointly provide only two of these three properties. Tolerance to network partitions in a distributed system cannot be sacrificed as it would require an absolutely reliable network. Thus, DDBSs can either guarantee availability by sacrificing consistency or vice versa.

Relational DBMSs (RDBMS) are widely used since they provide ACID guarantees, even though there are applications for which they do not scale [4]. In contrast to RDBMS, NoSQL datastores are built with scalability and high availability in mind. At the same time, the relaxed consistency guarantees provided by these systems is perceived as a burden by application developers [5] and force them to consider corner cases and to handle these in the application code. This shifts these problems from the database to the application level. Although recent NoSQL datastores provide tunable consistency on per request level, the correct consistency configuration is yet another dimension that needs to be considered during the application design, leading to an ever increasing design complexity. Strong consistency reliefs applications from the burden of handling corner cases. However, it generates higher overhead compared to weaker consistency levels, such as eventual consistency [4]. The deployment of applications that need to replicate data, possibly in different data centers, has made this trade-off even more tangible [6].

Different replication protocols have been developed with the goal of reducing the overhead for strong consistency (e.g., 1SR). Read-one-write-all (ROWA) or read-one-write-all-available (ROWAA) eagerly commit all replica sites, whereas quorum protocols only commit a subset of sites, which makes them more suitable for update-heavy transaction mixes. Current quorum protocols neglect the properties of sites (e.g., load or network distance) for quorum construction, which is crucial if data is replicated across different data centers.

The elasticity of Cloud services allows for their dynamic tailoring to the application demands by provisioning and deprovisioning these services based on, for instance, application load and failures. Hence, application developers require relational DBMSs that provide full ACID support, but yet are able to meet the performance requirements of their applications. Moreover, they require DBMSs to fully exploit the advantages of the Cloud, be able to consider the properties of the infrastructure hosting the applications, and to dynamically adapt if these properties change.

In this paper we present QuAD, an adaptive quorum protocol for providing 1SR data consistency on top of a fully replicated database system. QuAD considers infrastructure properties for determining the quorums, and is able to dynamically react to changes in the system, such as increased load at sites, new sites joining the system, or site failures, and consequently adapts its quorums to address these changes. The contribution of this paper is threefold: First, we provide a model for quorum construction that jointly considers the

[1]https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/

load and network latency between sites. Second, we show how QuAD is able to dynamically adapt to workload changes and failures by reorganizing the quorums. Third, we show the performance gain of QuAD by comparing it to other quorum-based protocols that i.) neglect site properties, ii.) consider only a subset of properties, and that iii.) are non-adaptive.

This paper is organized as follows: In Section II, we introduce the QuAD system model. Section III discusses the QuAD protocol and Section IV its implementation. Section V provides a thorough evaluation of QuAD on AWS. Section VI discusses related work and Section VII concludes.

II. System and Transaction Model

We consider flat transactions that run on top of a DDBS consisting of a set S of sites. We distinguish between logical objects LO and physical copies (PC) denoting copies of the LOs hosted at the sites. $pc_{i,j}$ is a physical copy of logical object o_i located at site s_j. We distinguish between read and write operations: $OP = \{r, w\}$. An action ac is an operation that acts on a specific $lo \in LO$, i.e., $ac \in OP \times LO$. Let A be the set of all actions. Then, a transaction t is a tuple with $t = (A_t, <_t)$ and $A_t = \{ac_1, ac_2, \cdots, ac_k\} \cup term$ with $ac_i \in A$ where $term$ is either a commit or an abort that succeeds all other actions, and $<_t \subseteq (A_t \times A_t)$ denotes the precedence relation defined on A_t, i.e., $ac_i <_t term \; \forall \; ac_i \in A_t$.

Transaction execution in a DDBS is characterized by the following two challenges: i.) the read and write activities of transactions on logical objects need to be mapped to physical copies. This task is commonly referred to as *replica control* and implemented by a *replica protocol (RP)*. ii.) the concurrent access of transactions to physical copies has to be coordinated. This task is commonly referred to as *concurrency control* and implemented by a *concurrency control protocol (CCP)*.

A RP can be classified according to *where* transactions are executed and *when* the results of updates are propagated to other replica sites in the system [7]. The '*where*' defines which site is allowed to execute a transaction. The '*when*' defines the point in time at which updates are propagated to other replica sites. RPs can be either *eager* or *lazy*. Eager RPs update all replica sites in the scope of the running transaction. Lazy RPs postpone the update propagation to other replica sites to dedicated refresh transactions.

A. Quorum Protocols

In quorum-based RPs, only a subset of replica sites is updated eagerly. This is in sharp contrast to ROWA and ROWAA, where either all replicas (ROWA) or all available replicas (ROWAA) have to be eagerly updated. However, the subsets in quorum protocols must be chosen in such a way that any two writes or a write and read on the same data object overlap. This is known as the *intersection property* and is crucial for guaranteeing 1SR consistency [8]. Committing only a subset of sites reduces the overhead for updates, but at an increased cost for reads, as they —in contrast to ROWA(A)— must also access a subset of sites. Quorum-based replication protocols differ in the size of read and write quorums, and thus in their overhead for read-only and update transactions.

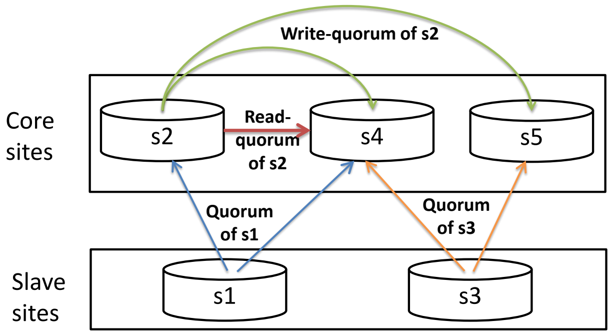


Fig. 1: Quorum construction in QuAD

Majority Quorum (MQ) is a simple quorum-based RP, in which each site has a non-negative number of votes. The quorums are then chosen in such a way that they exceed half of total votes [9], [10]. Hence, the write quorum can only be reached by a majority of sites, whereas a read quorum needs half of the sites if number of sites is even, or a majority of sites if it is odd, given that all sites have equal votes greater than zero (e.g., one). Other protocols, like tree-based [11] or grid-based [12] protocols, organize the sites in a logical structure which is exploited to determine the quorums.

B. Transaction Lifecycle

The lifecycle of transactions in quorum-based replication protocols is as follows. In the first step, transactions acquire (shared or exclusive) locks for all objects accessed, depending on the action to be executed. Read-only transactions access the read quorum by reading the values of the objects at every site that is part of the quorum, and construct the final result by taking those values that have the highest commit timestamp. Update transactions must first update the local versions at the executing site by accessing the read quorum, produce the results, and then propagate them eagerly to the sites of the write quorum. The quorum size and the properties of the sites in these quorums are the major factors for the overall transaction overhead. Existing quorum protocols generate quorums of different sizes, and (in contrast to QuAD) usually neglect the site properties, or consider them without the ability to adapt the quorums if these properties change. Both aspects are however crucial in highly volatile workloads and multi-data center deployments.

III. QuAD Protocol

The goal of QuAD is to build quorums that avoid 'weak' replica sites from the read and commit paths of transactions, where weak can have different meanings, such as slow and distant, but also expensive. QuAD considers the load of the sites and their distance, i.e., the round-trip time (RTT), when determining the quorums. Additionally, QuAD seeks a possibly balanced assignment of sites to quorums, since if sites are frequently included in the quorums, they may become a bottleneck [13].

QuAD assumes that any site may receive transactions for execution. This corresponds to usual approaches in which transactions are submitted to the *closest* data center (w.r.t. network latency/distance) and distributed by a load balancer. QuAD aims to reduce read and commit overhead by avoiding slow and distant sites from being accessed in the read and write quorums, which has a considerable impact on the performance.

A. *Quorum Construction*

Quorum construction in QuAD is motivated by the κ-centers problem [14]: given a set of n cities, the goal is to build κ warehouses so that the maximum distance of a city to a warehouse is minimized. Similarly, QuAD chooses the κ strongest sites to become *core sites* ($CS \in 2^S$), and the rest forms the set of *slave* sites ($SL \subseteq S$). The strength of a site is determined by its *score*.

In QuAD, each core site creates quorums that consist of core sites only, and each slave site constructs its quorums by including the majority of core sites. We denote the quorums of a core site as *core quorum*, and that of a slave site as *slave quorum*. A core read quorum (CQ_r) consists of the majority of core sites, and a core write quorum (CQ_w) of all core sites, in order to provide a high degree of availability. Slave quorums always consist of the majority of core sites and both the read and write quorums are the same ($SQ_r = SQ_w$). According to the transaction lifecycle (see Section II-B), an update transaction in QuAD submitted to a core site eagerly updates all core sites, and an update transaction submitted to a slave eagerly commits only the majority of core sites. Read-only transactions access the majority of core sites independently of the site they were submitted to. Note that we have a bi-directional communication between core sites, while it is only unidirectional between slave and core sites, i.e., a slave site accesses a core site, but never vice versa. This corresponds to the intuition of avoiding weak sites from the commit and read paths (Figure 1). The roles of the sites are not static and may change if site site properties (e.g., load) change. This may necessitate an adaption of quorums as old cores may be demoted to slaves, and slaves be promoted to core sites.

Site Score: The score of a site s_i is based on its load $load(s_i)$, and its distance $rtt(s_i)$ to all other sites. Let CN be the distance matrix with $rtt(i,j)$ denoting the RTT between s_i and s_j and $rtt(i,j) = rtt(j,i)$. The i^{th} row of CN defines the distance of s_i to all other sites in the system: $rtt(s_i) = [CN(i,1), \cdots, CN(i,|S|)]$:

$$\begin{pmatrix} 0 & rtt(1,2) & \cdots & rtt(1,|S|) \\ \vdots & \ddots & & \vdots \\ rtt(i,1) & 0 & \cdots & rtt(i,|S|) \\ \vdots & & \ddots & \vdots \\ rtt(|S|,1) & \cdots & \cdots & 0 \end{pmatrix} \tag{1}$$

The score of a site is defined as

$$score(s_i) = w_{rtt} \cdot rttsc(s_i) + w_{load} \cdot loadsc(s_i) \tag{2}$$

with $rttsc(s_i), loadsc(s_i) \in [0,1]$ and

$$\begin{aligned} rttsc(s_i) &= 1 - \frac{\|rtt(s_i)\|}{\max\limits_{\forall s_k \in S}(\|rtt(s_k)\|)} \\ loadsc(s_i) &= 1 - \frac{load(s_i)}{\max\limits_{\forall s_k \in S}(load(s_k))} \\ load(s_k) &= \frac{nrTrx(s_k)}{maxTrx(s_k)} \end{aligned} \tag{3}$$

Based on Eq. (2), the κ sites having the highest scores are chosen as core sites. Initially, as no load data is available on the sites, each one will get the same load score. This leads to the RTT becoming the determining factor for the site scores.

The choice of the number of core sites has a considerable impact on the overall performance of QuAD. It is also crucial for the availability of QuAD, as the core sites are included in quorums of both core sites and slave sites (see Figure 1). The lower the number of core sites, the lower the availability of QuAD, and the lower the commit and read overhead. However, the load balancing capabilities also decrease with decreasing number of core sites. With κ, it is possible to simulate the behavior of different protocols like ROWAA. If, for example, all sites are core sites ($\kappa = S$), then CQ_w will include all available sites. Consequently, it would be safe from a consistency point of view to access a single site in case of read-only transactions. Such a behavior corresponds to ROWA.

As any site in the system may receive transactions, we need to determine for each of them the read and write quorums. Choosing all core sites as part of the write quorum considerably impacts the fault tolerance of QuAD. Each slave site will choose the majority of core sites as part of its quorums, and the read and write quorums are equal. The main question is how to determine the quorums of the slaves, i.e., the subset of core sites that a slave site will be attached to, so that the cost is minimized? There are two main issues here. i.) we need to consider the cost of assigning a slave site to a subset of core sites so that in overall we minimize the average cost. ii.) if we include the same core site in too many quorums, that site will become a bottleneck and degrade the overall performance. This means that we need to update the costs each time we assign a slave site as the cost of the core sites will increase with every slave site assigned to them.

Cost Model: Determining the slave quorums corresponds to the assignment problem [15], which is defined as follows: Let WO denote the set of workers, and J the set of jobs. The goal is now to assign the jobs to the workers so that the overall cost is minimized: $min \sum_{j \in J} cost(wo, j)$. The assignment problem can be solved using the Hungarian algorithm, which has a complexity of $\mathcal{O}(n^3)$ with $n = max(|WO|, |J|)$ [16].

For the quorum construction of slave sites, we need to perform following steps: define i.) a cost model that considers both the load and RTT between sites, and ii.) a one-to-many mapping, i.e., a slave site is assigned to a subset (majority) of core sites. This means that the cost model needs to consider that. Let us assume a DDBS with five sites. Further, let s_1, s_2 and s_3 be the core sites, and s_4 and s_5 the slave sites.

In order to guarantee the intersection property, each slave site must include in its quorum two core sites, i.e., a quorum of a slave consists in this case of two core sites. In this scenario, there are three possible core site combinations, and thus three different assignments of slaves to quorums. s_4 can be assigned to $\{s_4, \langle s_1, s_2\rangle\}$, $\{s_4, \langle s_1, s_3\rangle\}$ or $\{s_4, \langle s_2, s_3\rangle\}$, and the same applies to s_5. To cope with the one-to-one mapping that is assumed by the assignment algorithms, we need to combine the cost of individual core sites to a single cost value. For example, the cost of assigning a slave site to a quorum may be calculated by considering the maximum cost of s_1 and s_2.

Let $Q \subseteq CS$, with $|Q| = \lfloor\frac{|CS|}{2}\rfloor + 1$ denote a set of core sites to which a slave site s_i needs to be assigned. The costs of assigning s_i to individual core sites $cs \in Q$, $cost(s_i, cs_1), \cdots, cost(s_i, cs_{|Q|})$ are known and can be combined to a single value using an aggregate function, such as max. In what follows, we use $cost(s_i, Q)$ to denote the cost of constructing a quorum consisting of the slave site s_i and the sites in Q. Let $w_1, w_2, w_3 \in \mathbb{R}^+$ denote the weights, then the cost $cost(s_i, Q)$ for assigning s_i to Q is defined as follows:

$$\begin{aligned} cost(s_i, Q) = w_1 \cdot \text{commCost}(s_i, Q) \\ +w_2 \cdot \text{loadCost}(s_i, Q) \\ +w_3 \cdot \text{balPen}(s_i, Q) \end{aligned} \tag{4}$$

$$\begin{aligned} \text{loadCost}(s_i, Q) &= \overline{load(s_i)} \cdot \max_{\forall cs_j \in Q}(\overline{load(cs_j)}) \\ \overline{load(s)} &= \frac{load(s)}{\max\limits_{\forall s_k \in S}(load(s_k))} \end{aligned} \tag{5}$$

$$\text{commCost}(s_i, Q) = \frac{\max\limits_{\forall s_j \in Q} rtt(s_i, s_j)}{\max\limits_{\forall s_k, s_m \in S \wedge k \neq m} rtt(s_k, s_m)} \tag{6}$$

$$\text{balPen}(s_i, Q) = \exp(\#\text{SL(Q)} + 1 - |SL|) \tag{7}$$

For the assignment of slaves to quorums, we use max as aggregation function as the slowest and most distant site is the limiting factor w.r.t. performance. The balancing penalty ($balPen$, Eq. (4) and (7)) is used to increase the cost of Q with increasing number of slaves assigned to Q, and it fulfills the purpose of a balanced assignment of the slaves to core sites, as otherwise they may become a bottleneck [13]. $\#SL(Q)$ in Eq. (7) defines the number of slave sites that have already been assigned to Q. The $balPen$ function will never become zero, and will get the maximum value of one if all slave sites are assigned to Q. Such a scenario would lead to Q becoming a bottleneck and degrading the overall performance.

B. Adaptive Quorums

QuAD is a dynamic protocol which continuously monitors the status of sites and their properties. The following cases may trigger a reconfiguration of quorums: i.) the load of sites changes, which may invalidate the quorums. We assume that the RTT between remains constant and therefore QuAD only monitors the load of all sites. ii.) sites may fail, which may lead to a reconfiguration in order to maintain the desired availability. iii.) new sites may join. In case of adaption of existing quorums, certain core sites may be demoted to slaves, and certain slaves may be promoted to core sites. Clearly, in order to guarantee strong consistency, a safe reconfiguration is necessary (the online reconfiguration is detailed in Section IV).

Load Prediction and Monitoring: QuAD periodically collects the load of all sites and configures the quorums based on Eq. (2) and (4). In QuAD, time is divided in periods. At the end of a period, the expected load for the next period is predicted using the exponential moving average (EMA), which proved to precisely predict certain workload types [17]. EMA considers historical data and weights them based on recency:

$$\begin{aligned} EMA(load_{p+1}) = \\ \alpha \cdot load_p + (1-\alpha) \cdot EMA(load_p)) \end{aligned} \tag{8}$$

α denotes the smoothing factor with $0 < \alpha < 1$. The choice of α is critical for the accurate prediction of the load. QuAD applies Eq. (8) with $\alpha \in \{0.1, 0.2, 0.3, \cdots, 0.9\}$. The value of α having the lowest overall mean absolute deviation is used for the prediction of the load for the next period. Each time a prediction is made, a new prediction interval is started and the old one is closed. In Eq. (8), p denotes the recently closed interval and $p+1$ the next interval for which the load is predicted. In EMA, predictions are recursively based on the past and require only the predicted load for the last period. Once the expected load for the sites has been predicted, the score of each site is determined in the next step based on Eq. (2). In our current work, we assume that all sites have the same capacity. It is, however, possible to use linear regression, predict the expected latency given certain expected load, and base the scoring of sites on the latency. This would account for heterogeneous sites w.r.t. their capacity.

Control of Quorum Adaption: The change in the site properties triggers a calculation of the quorums. QuAD incorporates a control mechanism to avoid frequent reconfigurations that would not generate enough gain for the expected workload. The decision on applying the new quorums is based on whether their score is higher compared to that of the old quorums by considering the reconfiguration cost. The score of a quorum q is defined as $score(q) = \prod_{s \in q} score(s)$. The overall average score of quorums is then:

$$\overline{score(Q)} = \frac{\sum\limits_{q}^{Q} score(q)}{|Q|} \tag{9}$$

Each slave site to be promoted must update its objects by contacting current core sites. Hence, this reconciliation generates costs that need to be considered when applying the new quorums. QuAD trades the gain in the score for the cost of reconfiguration when deciding to apply the new quorums:

$$\overline{score(Q_{new})} - \overline{score(Q_{old})} > \prod_{s}^{PromoSL} reconCost(s) \tag{10}$$

with $reconCost(s) = \frac{\#objectsToUpdate}{|LO|}$.

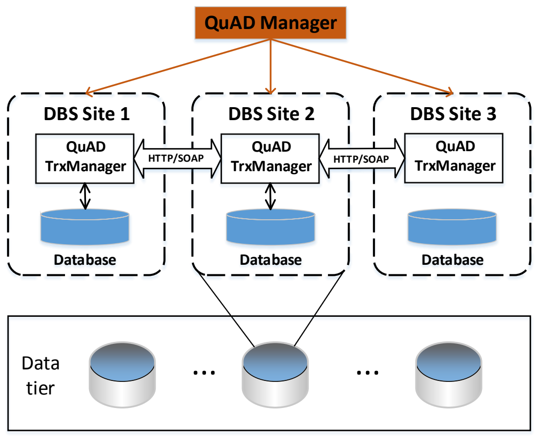


Fig. 2: QuAD Architecture

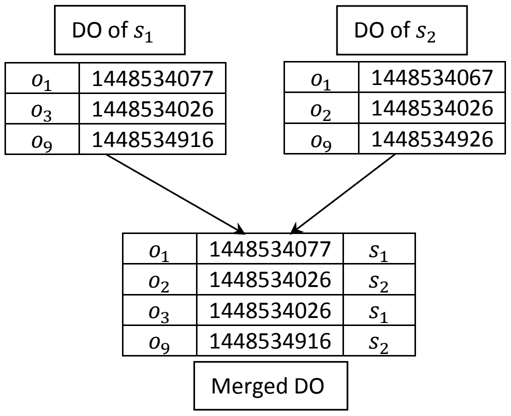
Sites Joining: If a new site joins, it becomes a slave and is assigned to a quorum consisting of core sites according to the cost matrix that was created during the last quorum construction. The recalculation of quorums is postponed to the next reconfiguration period, as then enough information will be available for determining the score of the sites. However, if the site joins immediately before the new period starts, then it is still not possible to determine its score, so it remains a slave. QuAD requires each site to run at least one period before it is considered for the scoring, otherwise it is labeled as a slave.

Failure Handling: In case of slave site failure, QuAD does not take any immediate action as the quorums are anyways adapted during the next reconfiguration period. From the viewpoint of QuAD, a failed slave site simply implies reduced processing capacity and has no further impact. However, core site failures reduce the availability of the system. Currently, QuAD tries to keep the number of core sites by promoting slave sites to cores. If no sites are available for promotion, then QuAD cannot provide the desired level of availability. We assume a system model in which the creation or deletion of sites is outside the QuAD control, and is steered by dynamic and cost based replication protocols as the one defined in [18].

IV. Implementation of QuAD

A QuAD site consists of a `QuAD-TransactionManager`, responsible for transaction execution, and a datastore (see Figure 2). The `QuAD-TransactionManager` executes transaction as described in Section II-B by considering the quorum configuration, which are determined and distributed to the sites by the `QuAD-Manager`. The latter is also responsible for collecting all necessary metadata from the sites, to determine and adapt the quorums as described in Section III at runtime by avoiding consistency violations.

Online Reconfiguration: Three types of events may trigger a reconfiguration of quorums: i.) the change of the site properties (e.g., their load), ii.) the deployment of new sites, and iii.) site failures. The `QuAD-Manager` is responsible for load prediction based on the collected metadata. In case of new sites joining, they must register to the `QuAD-Manager`, which has to notify all sites if a reconfiguration is applied.



DO of s_1	
o_1	1448534077
o_3	1448534026
o_9	1448534916

DO of s_2	
o_1	1448534067
o_2	1448534026
o_9	1448534926

Merged DO		
o_1	1448534077	s_1
o_2	1448534026	s_2
o_3	1448534026	s_1
o_9	1448534916	s_2

Fig. 3: Merging of DOs

For the detection of core site failures, each core site of QuAD needs to periodically notify the `QuAD-Manager` about its state. Without notification, a site is dropped from the list of available sites, and the quorums are invalidated immediately.

Once the quorums are determined and `QuAD-Manager` has decided to apply them, it will initiate the reconfiguration process, which has to be done in a consistent manner; otherwise, the consistency of data may be violated. Two aspects need to be treated with care: i) all sites must have a consistent view on the quorums to avoid that the intersection property is violated. ii) as described above, the role of the sites may change. It must be ensured that promoted slave sites reconcile in order to provide transactions access to consistent data.

One-Copy View on Quorums: If the reconfiguration is done in an unsafe manner, certain sites may observe intermediate configurations which may violate the intersection property and thus the 1SR guarantees. The QuAD reconfiguration protocol provides sites with a consistent view on the quorums, by using 2PC with the `QuAD-Manager` acting as coordinator. The new quorums are propagated to all sites during the *prepare* phase, which at the same time *initiates* the interruption phase the at the sites. During the interruption phase, incoming transactions are added to a wait queue, and their execution is resumed only after the commit.

Site Reconciliation: We distinguish between site reconciliation during promotion/demotion without failures in reaction to changes of the site properties, and during the promotion of one or more slave sites to core sites in reaction to core site failures. The reconciliation is executed as part of the reconfiguration workflow before sending the `prepare-ack` message to the `QuAD-Manager`.

Each site in QuAD manages a *difference object (DO)* that contains the latest timestamp for each object that has been modified since the last quorum reconfiguration. The DO of a site s_j is a set of tuples: $DO(s_j) = \{\langle k_{o_a}, \tau(k_{o_a})\rangle, \cdots, \langle k_{o_z}, \tau(k_{o_z})\rangle\}$, with k_{o_a} denoting the id of the object o_a, and $\tau(k_{o_a})$ its latest timestamp. Consider the promotion of slaves to cores after load changes. The demoted core sites multicast their difference object to all slave sites that are to be promoted. Each promoted slave merges the DOs received by taking the largest timestamp and entering the site id for each object, and drops out all objects for which the local timestamp is equal to or greater than the timestamp in the merged DO, as these objects are already up-to-date (Figure 3). QuAD supports the stop-and-copy approach in which the promoted slave sends a batch request to the demoted core site for pulling all objects in the merged DO that have the id of that demoted core site. The reconfiguration is finished only when all slave sites to be promoted have updated their data. Once the reconfiguration has finished, the system is ready to serve transactions based on the new quorums. We plan to implement an on-the-fly approach that will only pull those objects accessed by transactions on demand [19], [20].

In case of core site failures, the corresponding number of slave sites will be promoted to cores sites which then need to synchronize with all slave sites. This ensures that a site to be promoted contains the data of all slaves. However, we need to ensure that it also contains the core site data. It is sufficient that they synchronize with a single core site, as the core writes behave according to the write-all approach.

QuAD tolerates up to $n-1$ simultaneous core sites failures, with n denoting the number of core sites, under the assumption that no slave site fails at the same time. It remains available if the majority of cores is available independently on the number of failed slave sites.

V. Evaluation

The goals of the evaluation of QuAD are as follows: i.) we show the importance of considering site properties when constructing the quorums by a series of tests using the MQ protocol that neglects site properties for quorum construction. ii.) we compare the performance of QuAD to that of MQ using round-robin and random quorum construction strategies in a single-data center and a multi-data center setting. iii.) we compare the different construction strategies of QuAD and show their impact on the overall performance. iv.) we analyze the necessity of adapting the quorums if site properties change at runtime and show that QuAD is able to adapt its quorums.

A. Evaluation Setup

All evaluation runs use the TPC-C benchmark. The set of transaction mixes r/w with r and w defining the percentage of read-only and update transactions, are as follows: $r, w \in \{0, 0.2, 0.5, 0.8, 1.0\}$, $r + w = 1.0$. The TPC-C data is generated with 10 districts, 3,000 customers, and 10,000 stock entries. Each object has the same probability of being accessed (no hot-spots). Thus, the conflict rate between transactions is mainly influenced by the r/w ratio of the transaction mix.

The QuAD system consists of a $\#sites$ deployed on AWS EC2[2]. A test client, which runs separately from the system under test generates the desired workload consisting of the specified r/w transaction mix. The client starts a number of *WorkerThread*s submitting transactions sequentially (i.e., they wait for the response before they continue) to a specific site for execution.

[2]c1.medium machine types (http://aws.amazon.com/de/ec2/instance-types/)

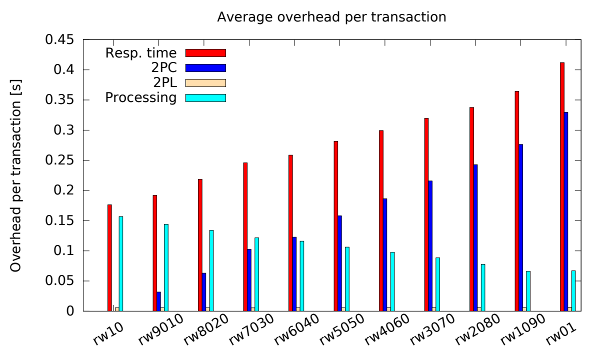


Fig. 4: MQ: Transaction overhead, varying r/w ratio

We run two different types of tests: a *sizeup* and a *speedup* test as described in [21]. In the *sizeup* test, an initial number of 10 *WorkerThread*s is started which submit transactions for 30 seconds, and after that collect the statistics and increase the number of *WorkerThread*s by 10, until the maximum of 150 workers is reached. The entire *sizeup* test runs for 450 seconds and is repeated 10 times. The goal of *sizeup* is to analyze the response time of transactions when the load increases. The goal of the *speedup* is to analyze the performance improvement of QuAD compared to other approaches. The speedup is calculated as follows: $speedup = \frac{resptime(approach)}{resptime(QuAD)}$, and there is an improvement if $speedup > 1$. During the speedup test, the load remains constant, i.e., the number of *WorkerThread*s does not change.

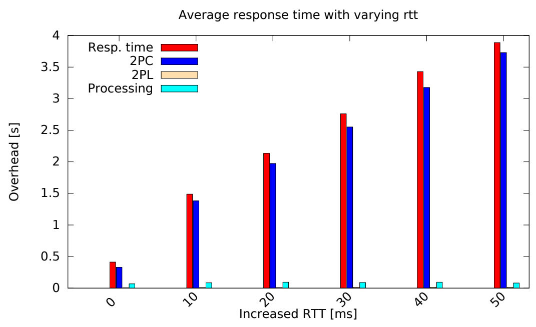
B. Performance Impact of Site Properties

The goal of the quorum protocols is to reduce the overhead for update transactions as only a subset of sites is eagerly committed. However, in order to guarantee strong consistency, reads must also access a subset of sites which, in contrast to ROWA(A), increases the overhead for reads. To show the necessity of considering site properties such as their RTT and load when constructing the quorums, we have conducted a first series of experiments using the MQ protocol with four sites. A test client with a single *WorkerThread* generates transactions for 450 seconds with a specific transaction mix. These transactions are submitted for execution to a dedicated site, which randomly constructs an initial majority quorum.

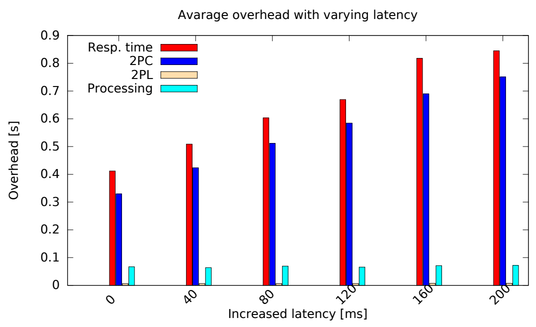
In Figure 4, we have depicted the overhead per transaction with all sites having the same properties (i.e., the load of the sites and the RTT between sites is the same) by varying the r/w ratio. As it can be seen, with an increase of the update ratio, the 2PC costs increase. The 2PL overhead remains constant as there are no concurrent transactions. The goal was to depict only the processing and 2PC cost. In summary, in case of all sites having the same properties, the transaction mix determines the overhead of the different transaction phases.

In a next experiment, we have varied the RTT (by using the `netem`[3] tool) and the load of a certain site. We ensured that the modified site is included in the read and write quorums. In Figure 5 we have depicted the results for the write-only mix.

[3]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem



(a) Varying RTT



(b) Varying load

Fig. 5: MQ: Transaction overhead for write-only mix

Both the increase of the RTT and the load leads to a higher overhead for 2PC which requires some rounds of network messages that also need to be processed at the receiving sites (in Figure 5b, latency corresponds to the additional latency relative to the latency of a site without any load). It should be mentioned that an increase of the overhead in one of the transaction phases has a cascading effect on 2PL and then on the total overhead [22]. Thus, it is crucial to construct the quorums in such a way so that the overhead for 2PC and processing is reduced as this would lead to a decrease in 2PL cost, which is the limiting factor to 1SR performance.

C. QuAD vs. MQ

Next, we compare the performance of QuAD to that of the MQ that uses round-robin (MQ-RR) and random (MQ-RA) for quorum construction, using a *sizeup* test in a single-data center and a multi-data center setting. The *WorkerThreads* submit transactions to sites according to the desired r/w ratio. The distribution of workers (transactions) to sites is based on the load to be generated at the sites. Note that we also report the result for the QuAD-inversed, in which the weaker the site the higher its score, simply for reasons of comparison.

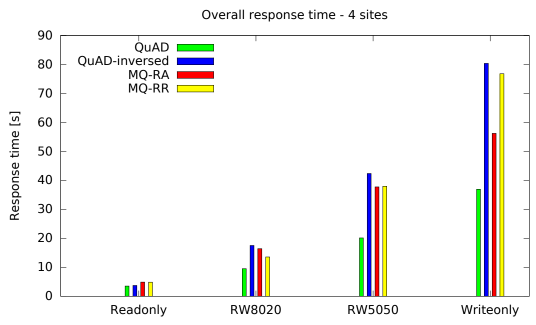
In the *single-data center* setup, the load is the determining factor for the performance, as the network distance (latency) between the sites is negligible. We have run the evaluations using 4, 8, and 16 sites, with a subset of sites being core sites. The distribution of *WorkerThreads* to sites determines the load generated at the sites. For the evaluation with four sites, one gets 40% of the overall load, the second one 30%, the third one 20%, and the last one 10%. In the evaluation with eight sites, the distribution is: 30%, 15%, 15%, 10%, 10% and the remaining 20% are evenly distributed to the rest of the sites. The load distribution in the case with 16 sites is accordingly.

Figure 6 depicts the overall averaged response time of transactions in the *sizeup* test showing that QuAD considerably outperforms both MQ-RR and MQ-RA, which neglect site properties when constructing the quorums. For update-heavy workloads, QuAD leads to a decrease of response time by more than 50%. The main reason is that quorums are constructed in such a way so that weak sites are possibly avoided.

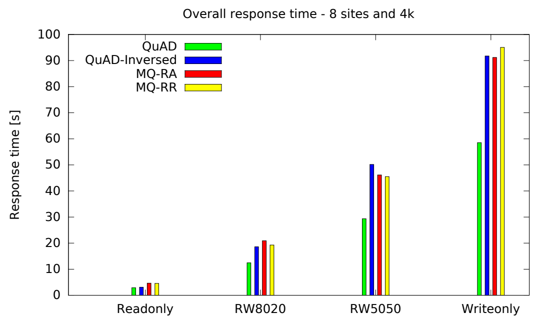
However, QuAD has a higher overhead for reads compared to ROWAA. In a simple speedup test comparing the performance of QuAD and ROWAA for read-only workloads with all sites having the same properties, the average response of ROWAA is 0.08 seconds, and that of QuAD 0.2 seconds (i.e., ROWAA leads to a speedup of 2.5 for read-only workloads).

One of the crucial aspects in QuAD is the choice of κ which impacts both performance and availability. We have run the same sizeup test with varying κ with 8 sites to show the trade-off between availability, which increases with increasing number of κ sites, and the optimization capabilities, which may rapidly decrease if the properties of the sites are similar. If a subset of sites is significantly better than the rest, there is even be an advantage in increasing the number of core sites, as the more core sites available, the more choices there are for assigning the weak sites. This, in turn, may be beneficial from a load balancing point of view. However, if there are many weak sites, the more cores exist, the more weak sites are to be included. The results are depicted in Figure 7 (two strong sites, the rest was weak) which shows that the increase in the number of core sites leads to a decrease of performance.

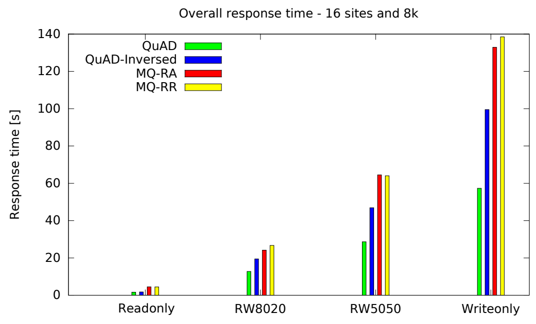
The goal of the *multi-data center* deployment is to evaluate the impact of the RTT on the overall performance. Therefore, we have increased the RTT between three sites (with ratio of 4:2:1). We evaluated QuAD using the *sizeup* metric with 4 sites ($\kappa = 2$), 8 sites ($\kappa = 4$), and 16 sites ($\kappa = 8$). Based on the scoring model defined in Section III-A, QuAD determines the core sites by considering both the load and RTT distance, and assigns the slave sites to quorums consisting of core sites based on the cost model defined in Eq. (4). However, since the load of all sites is the same , the RTT will be the determining factor for quorum construction. As depicted in Figure 8, QuAD significantly outperforms other approaches for update-heavy workloads by decreasing the average response time nearly by a factor of 3. The consideration of the RTT is more significant, especially for update-heavy workloads, as they are mainly network bound due to the 2PC communication. Note that as we use SOAP/HTTP for the communication between the sites, the RTT is a crucial factor for the overall performance.



(a) 4 sites; $\kappa = 2$

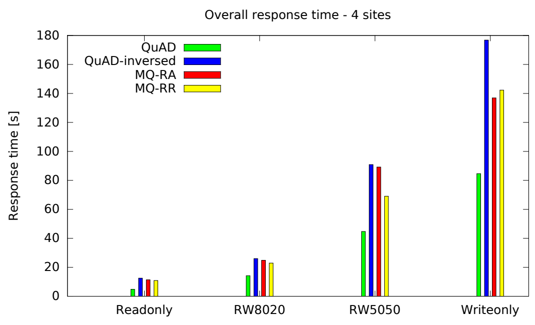


(b) 8 sites; $\kappa = 4$

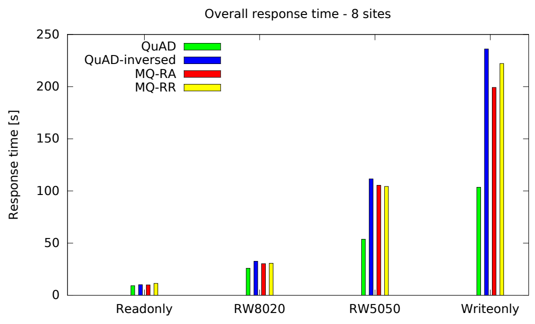


(c) 16 sites; $\kappa = 8$

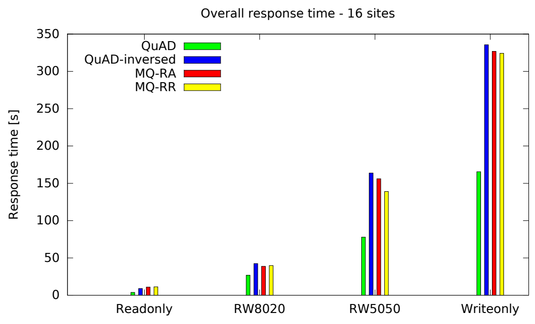
Fig. 6: Overall response time of transactions with varying site load (single-data center setting)



(a) 4 sites; $\kappa = 2$



(b) 8 sites; $\kappa = 4$



(c) 16 sites; $\kappa = 8$

Fig. 8: Varying RTT (multi-data center setting)

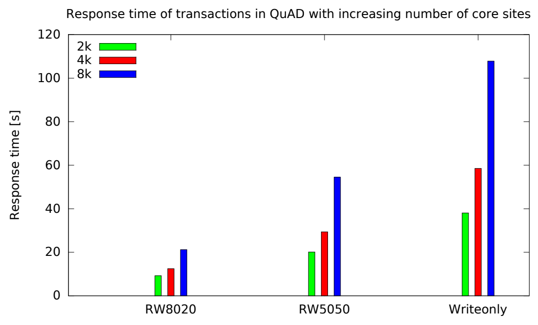


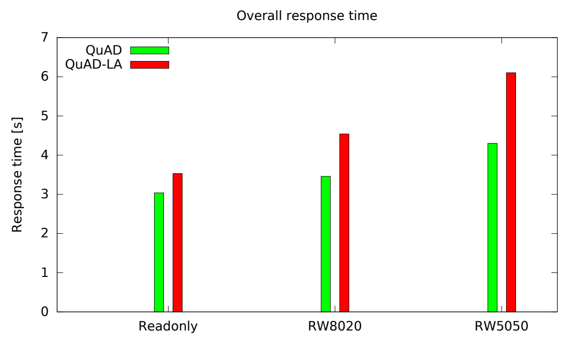
Fig. 7: Varying κ (8 sites)

D. *QuAD Quorum Construction Strategies*

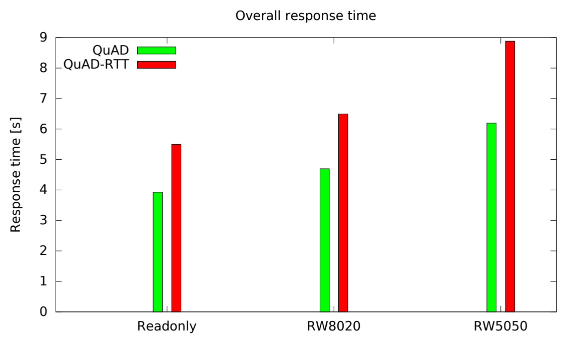
In this series, we compare QuAD to strategies that either only consider the load or the RTT. We use 4 sites with $\kappa = 3$. In this evaluation, the number of workers remains constant (speedup test). All transactions are submitted to the slave site.

First, we compare QuAD to an assignment which considers only latency (QuAD-LA). The core site with the smallest load has the greater distance to the slave, and as the load of the cores increases, the distance decreases by the same factor. QuAD-LA chooses the core quorum with the lowest maximum load and assigns the slave to that quorum, whereas QuAD chooses the quorum with the lowest cost by jointly considering load and RTT. As depicted in Figure 9a, QuAD outperforms QuAD-LA for all r/w ratios. The performance gain of QuAD is considerable for network-bound update-heavy workloads.

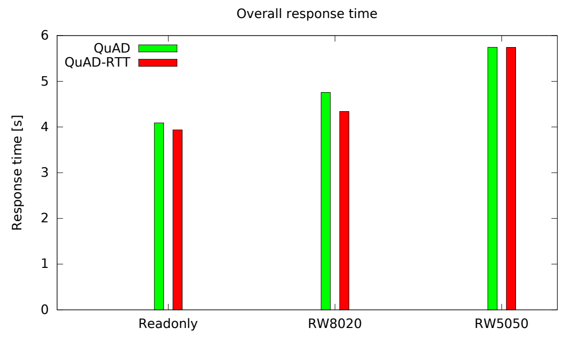
In the second step, we compare QuAD to an assignment which only considers RTT (QuAD-RTT). The core site with the smallest RTT has the highest load, and as the RTT of the cores to the slave increases, their load decreases. We conduct three different evaluations which differ in the load generated at the weakest core site from the load point of view. The first evaluation generates a load that corresponds to an average latency of 2,500 ms, the second to 500 ms and the third one



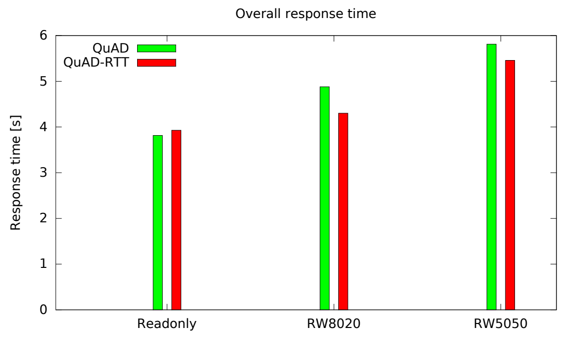
(a) QuAD vs. Latency only



(b) QuAD vs. RTT only (Large load at a core site)

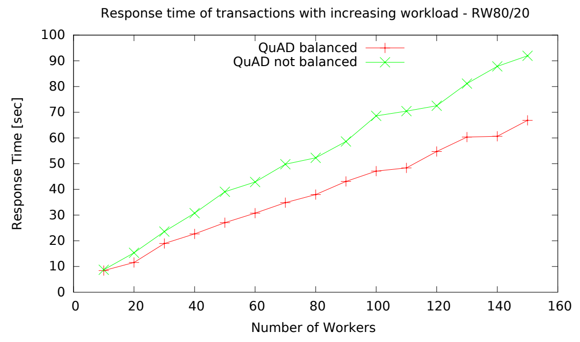


(c) QuAD vs. RTT only (Middle load at a core site)

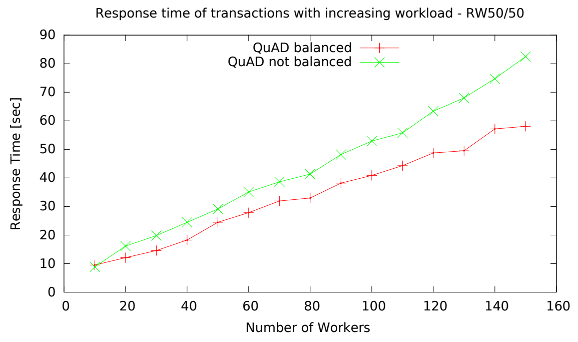


(d) QuAD vs. RTT only (Small load at a core site)

Fig. 9: Comparison of assignment types

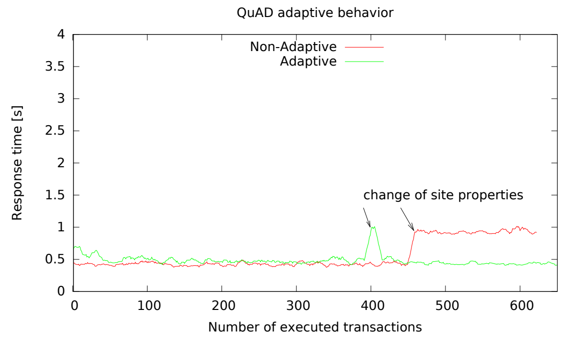


(a) $r/w = 80/20$ (80% reads – 20% writes)

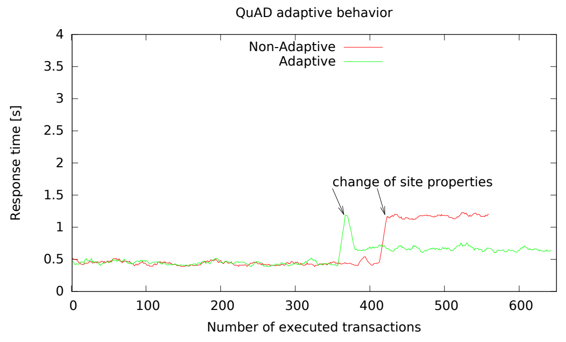


(b) $r/w = 50/50$ (50% reads – 50% writes)

Fig. 10: Balanced vs. non-balanced QuAD



(a) Swapping load of a core and a slave site



(b) Increasing load at one core site

Fig. 11: QuAD adaptive behavior

to 100 ms. For the high load evaluation, QuAD outperforms QuAD-RTT (Figure 9b). However, as the load of the weakest core site becomes smaller, the performance of QuAD-RTT gets better and it outperforms QuAD (Figures 9c and 9d). The reason is that in our SOAP/HTTP-based implementation, the network should have a higher weight compared to the load.

E. Balanced vs. Non-Balanced QuAD

When assigning the slaves to core quorums, QuAD tries to balance the assignment in order to avoid that certain core quorums become a bottleneck if too many slaves are assigned to them. In order to analyze this, we conduct a sizeup test comparing QuAD to a version that does not balance, i.e., that has a weight of zero for the balancing penalty in Eq. (4). As the load increases, the non-balanced version not only become bottlenecks w.r.t. the slave sites, but also transactions executed by them are impacted by the high load from the slaves, and the entire performance degrades (see Figure 10).

F. Adaptive Quorum Reconfiguration

Finally, we evaluate the ability of QuAD to adapt quorums w.r.t. to changes of site properties. Firstly, the load of a core and a slave site are swapped. QuAD reacts to the load swap and adapts the quorums (see Figure 11a). Compared to the non-adaptive quorum, this leads to a stabilization of the latency to the level before the changes in the load. Secondly, we increase the load at one of the core sites to a level that is between the load of two slave sites. As a consequence, QuAD demotes the core site to a slave, and the slave site having the lowest load is promoted to a core. However, as the load of the slave site is not decreased, the latency will remain at a higher level after the adaption of the quorums (Figure 11b).

VI. Related Work

In the last decade, the development of protocols that guarantee strong consistency in the presence of replication and that also incur low costs has attracted quite some attention.

Skute [18] is a dynamic replication mechanism based on an economic model and aims at minimizing the replication cost. In contrast to Skute, QuAD reduces the overhead for guaranteeing 1SR consistency of transactions on top of a fully replicated DBS. However, QuAD can very well be combined with *Skute*. The latter can create/destroy replica sites based on the cost model, and QuAD can further optimize the transaction overhead by dynamically adapting the quorums.

Schism [23] is a graph-based approach, able to partition the data by considering the transaction workload with the goal of reducing or completely avoiding distributed transactions. However, as soon as partitions need to be replicated, the choice of the replication protocol becomes crucial to the performance. QuAD can complement Schism by constructing quorums of the replicated partitions based on the site properties.

Spanner [24] is Google's highly scalable DDBS that provides strong 1SR guarantees. It is based on S2PL and Paxos for synchronous replication and uses True Time that assigns a commit timestamp to transactions in a scalable way. Based on the timestamps it is possible to globally order transactions.

VII. Conclusion and Outlook

In this work we have introduced QuAD, an adaptive and workload-driven quorum protocol for replicated databases tailored to applications that demand strong consistency. QuAD considers the load of sites and their network proximity for determining the optimal quorum configuration. The evaluation results show that QuAD outperforms static quorum protocols that do not consider site properties and that are thus not able to dynamically adapt to changes. In our future work, we plan to assess the adaptation overhead of QuAD and to compare it with further replication protocols. Moreover, we plan to extend QuAD by allowing it to learn the optimal configuration from application requirements (SLAs) and the actual workload.

References

[1] D. J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 3–12, 2009.

[2] E. Brewer, "Towards Robust Distributed Systems," in *PODC*, 2000.

[3] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[4] D. J. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *IEEE Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[5] J. C. Corbett *et al.*, "Spanner: Google's Globally-distributed Database," in *Proc. OSDI*, 2012, pp. 251–264.

[6] P. Bailis *et al.*, "Highly available Transactions: Virtues and Limitations," in *Proc. VLDB*, vol. 7, no. 3, 2013, pp. 181–192.

[7] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez, "Database Replication," *Synthesis Lectures on Data Management*, 2010.

[8] R. Jiménez-Peris *et al.*, "Are Quorums an Alternative for Data Replication?" *ACM Trans. Database Syst.*, vol. 28, no. 3, pp. 257–294, 2003.

[9] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *TODS*, vol. 4, no. 2, 1979.

[10] D. K. Gifford, "Weighted Voting for Replicated Data," in *Proc. SOSP*, 1979, pp. 150–162.

[11] D. Agrawal and A. El Abbadi, "The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data," in *VLDB*, 1990, pp. 243–254.

[12] S. Y. Cheung, M. H. Ammar, and M. Ahamad, "The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data," in *Proc. KDE*, 1990, pp. 438–445.

[13] A. Stiemer, I. Fetai, and H. Schuldt, "Comparison of Eager and Quorum-based Replication in a Cloud Environment," in *Proc. IEEE Big Data*, 2015, pp. 1738–1748.

[14] J. Bar-Ilan, G. Kortsarz, and D. Peleg, "How to Allocate Network Centers," *J. Algorithms*, vol. 15, no. 3, pp. 385–415, 1993.

[15] H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[16] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[17] M. Andreolini and S. Casolari, "Load Prediction Models in Web-based Systems," in *Proc. Valuetools*, 2006, p. 27.

[18] N. Bonvin, T. G. Papaioannou, and K. Aberer, "A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage," in *Proc. SoCC*, 2010, pp. 205–216.

[19] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms," in *Proc. SIGMOD*, 2011, pp. 301–312.

[20] A. J. Elmore *et al.*, "Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases," in *SIGMOD*, 2015, pp. 299–313.

[21] B. F. Cooper *et al.*, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. SoCC*, 2010, pp. 143–154.

[22] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2009.

[23] C. Curino *et al.*, "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1, pp. 48–57, 2010.

[24] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, 2013.