Cross-layer management of a containerized NoSQL data store

Evdoxos Bekas* and Kostas Magoutis*†
*Department of Computer Science and Engineering
University of Ioannina
Ioannina GR-45110, Greece
†Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Heraklion GR-70013, Greece

***Abstract*—Internet-scale services increasingly rely on NoSQL data store technologies for scalable, highly available data persistence. To increase resource efficiency and deployment speed, such services are adopting new deployment models based on container technologies. Emerging container management systems (CMS) offer new models of interaction with containerized applications, exposing goal-oriented management capabilities. In this paper, we study the interaction of a containerized NoSQL data store, RethinkDB, with a popular open-source CMS, Kubernetes, demonstrating that events exposed by the CMS can be leveraged to drive adaptation actions on the NoSQL data store, improving its availability and quality of service. Typical adaptation actions in data stores often involve movement of data (e.g., migrating data replicas to a new node) and thus have high overhead. In this paper we demonstrate that lower-cost adaptation actions based on targeted reconfigurations of replica groups are possible and often offer rapid response to performance degradation (such as when a node experiences a temporary resource shortage). Such reconfigurations can be exercised by a cross-layer management system that bridges between CMS and NoSQL data store. We evaluate a prototype implementation of this management system in the context of Kubernetes and RethinkDB using the Yahoo Cloud Serving Benchmark on Google Container Engine. Our results demonstrate that infrastructure-level events provided by the CMS can drive proactive, low-cost data-store adaptation actions that improve overall system manageability, availability, and performance.**

I. Introduction

The growing adoption of container technologies has brought container management systems [1], [2], [3], [4] (CMS) to the forefront of infrastructure management for large-scale data centers. Large cloud service providers such as Amazon, Google, and IBM are nowdays offering container clouds such as the Amazon EC2 Container Service [5], Google Container Engine [6] and IBM Bluemix Container Service [7]. Similar to all middleware layers, NoSQL data stores have become available in containerized distributions that can be deployed in commercial and open-source CMSs. Just as in the case of cloud technologies and other tiered virtualization solutions before them, containerized applications have limited information about the underlying infrastructure managed by a CMS. In this paper we explore how a containerized application using a NoSQL data store can benefit from cross-layer communication of infrastructural events to enable previously infeasible adaptivity mechanisms to the NoSQL system in an autonomic manner.

Autonomic management features have already been incorporated in distributed NoSQL data stores, namely automated addition of new nodes or removal of crashed nodes, load balancing, elasticity, quality of service, etc. Such features rely on monitoring the condition of the virtualized resources allocated to the data store. In virtualized environments however, a data store does not have immediate access to information visible to the underlying infrastructure management system, such as forthcoming disruptions affecting virtualized resources (decommissioning and replacement of a node, interference of resources across co-located workloads, etc). Our first goal in this paper is to enable a communication path between a CMS and a NoSQL data store that will notify the latter about infrastructure activities impacting its resources, as a way to drive proactive adaptation policies.

Current adaptation actions on data stores are expensive however, as they typically require data movement: For example, changing the primary key of a large table is known to be an expensive operation [8]. Load balancing performed by moving data across nodes, either by migrating replicas or by moving records across shards, are both expensive operations. A recent example of a lightweight adaptation action that was shown to be beneficial in data stores using primary-backup replication with a "strong leader"[1], is to reconfigure replica groups by moving the leader away from nodes that are, or will soon be, heavily loaded [9]. In this way, a certain level of load balancing is feasible at low cost (the short availability lapse that such reconfigurations entail), occasionally leading to large benefits.

Our second goal in this paper is thus to show that such a lightweight reconfiguration mechanism, when used proactively in response to infrastructure-level events provided by the underlying CMS, can improve data store *load balancing* and *availability*, without involving data movement actions.

Key to the enablement of such lightweight adaptation actions is explicit control over replica roles and their placement in data stores through a programmatic API. Such support is of-

[1]In such systems, write operations (and often reads as well) are satisfied by any majority of replicas, which always involves the leader. The leader (or primary) typically takes a higher load than follower replicas (or backups).

978-3-901882-89-0 @2017 IFIP

fered by systems such as RethinkDB [10] and MongoDB [11]. We chose to base the prototype of our architecture on the former due to its more straightforward cluster management API [12], but our implementation can be extended to cover the latter as well [13]. Such support is not ubiquitous in NoSQL data stores however, as replica roles and locations are not controllable in some data stores (e.g., using consistent hashing [14], [15], [16]) or are controlled internally.

Our contributions in this paper are:

- A cross-layer management architecture and prototype implementation of a containerized NoSQL document store (RethinkDB) over a widely-deployed container management system (Kubernetes). A manager component intermediates by automatically mapping infrastructure events (such as new node added, node soon to be decommissioned) to data-store adaptation actions.
- An evaluation demonstrating the benefits of such a cross-layer management architecture in two scenarios on Google Container Engine deployments: Proactively reconfiguring a RethinkDB cluster on advance notification that a RethinkDB server is going to be decommissioned, and restoring service when a new server is available; and reconfiguring a RethinkDB cluster in the event of a hotspot affecting performance of a RethinkDB server.

The rest of this paper is organized as follows: In Section II we describe background and related work, and in Section III the design and implementation of our management architecture. In Section IV we present the evaluation of our prototype, highlighting key results, and finally in Section V we conclude.

II. Background and Related work

In this section we provide background on container-management systems (CMS), NoSQL systems and their adaptation policies, along with a more in-depth description of two components with which our prototype interacts, Kubernetes and RethinkDB, relating to previous research along the way.

Borg [1], [2] was the first container-management system developed at Google at the time container support was becoming available in the Linux kernel. It aims to provide a platform for automating the deployment, scaling, and operations of application containers across clusters of hosts. Borg manages both long-running latency-sensitive user-facing services and CPU-hungry batch jobs. It takes advantage of container technology for better isolation and for sharing machines between these two types of applications, thus increasing resource utilization. Omega [3], also developed at Google, has similar goals to Borg. It stores the state of the cluster in a centralized Paxos-based transaction-oriented store accessed by the different parts of the cluster control plane (such as schedulers), using optimistic concurrency control to handle conflicts. Thus different schedulers and other peer components can simultaneously access the store, rather than going through a centralized master.

Kubernetes [4] is an open source container-cluster manager originally designed by Google and inspired by Borg and Omega, donated to the Cloud Native Computing Foundation [17]. We describe Kubernetes in more detail in Section II-A. Google offers Kubernetes as the standard CMS in Google Container Engine (termed GKE) [6] within Google Cloud Platform. Another commercial CMS is Amazon's EC2 Container Service (ECS). ECS, like Kubernetes and Google's GKE, supports lifecycle hooks [18], namely upcall notifications of instance bootstrap or termination, an important cross-layer management primitive leveraged by our architecture.

Prior work on cluster management services includes Autopilot [19], an early approach to automate datacenter operations. Other approaches to cluster and datacenter management opting for a goal-oriented declarative style, have been a subject of research for some time [20]. Cross-layer management has previously been explored as way to break through the transparency enforced by multiple organizational layers in distributed systems [21], [22].

Adaptivity policies in data stores have been explored in the past. Cogo *et al.* [23] describe an approach to scale up stateful replicated services using replica migration mechanisms. Konstantinou *et al.* [24] describe a generic distributed module, DBalancer, that can be installed on top of a typical NoSQL data-store and provide a configurable load balancing mechanism. Balancing is performed by message exchanges and standard data movement operations supported by most modern NoSQL data-stores. While these approaches achieve adaptivity to a certain extent, they operate at the level of the application/NoSQL data store and cannot apply proactive policies based on advance information available at the infrastructure level. Cross-layer interaction between distributed storage systems and the underlying managed infrastructure has been explored by Papaioannou *et al.* [25], where co-location of virtual machines on the same physical host is detected by the middleware or exposed by the storage system (HDFS) to ensure that the placement of replicas avoids failure correlation to the extent possible. Wang *et al.* [26] propose cross-layer cooperation between VM host- and guest-layer schedulers for optimizing resource management and application performance.

Standard adaptation solutions available in stateless systems, such as steering load away from a temporarily overloaded server via load-balancing actions, are not easily applicable to data stores since this requires data migrations, a heavyweight and time-consuming activity. Replica-group reconfigurations, used in this paper, is a lightweight alternative that is especially fit for temporary overload conditions in data stores. Previous work using replica group reconfigurations to mask the impact of underperforming primaries includes ACaZoo [9], a data store that triggers re-elections at replica groups with primaries on nodes that are about to engage in heavy compaction activities, typical in systems using log-structure merge (LSM) trees. By moving the primary role of a replica group away from a node that is about to either become overloaded or to crash, we ensure better performance (as the primary will not block progress of the entire replica group) and availability. In this paper we use similar replica-group reconfigurations when notified about infrastructure-level events by the CMS.

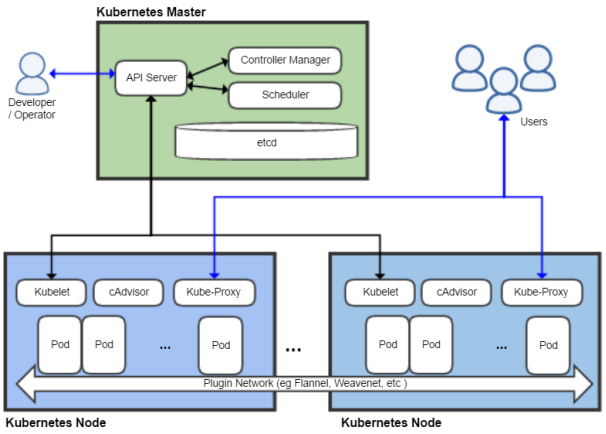


Fig. 1. Kubernetes architecture [27]

A. Kubernetes

The Kubernetes [4] architecture is depicted in Figure 1 [27]. At its core is a shared persistent store (*etcd* [28] in Figure 1) with components watching for changes to relevant objects. State in Kubernetes is accessed exclusively through a domain-specific REST API that applies higher-level versioning, validation, semantics, and policy, in support of clients.

Each Kubernetes node comprises an on-machine agent called a *kubelet*, and a component analyzing resource usage and performance characteristics of running containers called *cadvisor*. A group of one or more containers (e.g., Docker containers), the shared storage for those containers, and options about how to run the containers, is organized into a single entity called a *pod* akin to an application-specific logical host.

B. RethinkDB

RethinkDB [10] is an open-source document oriented NoSQL database. It stores JSON documents with dynamic schemas, and supports sharding (horizontal partitioning) and replication and has its own query language called *ReQL* [29] to express operations on JSON documents. RethinkDB features a web-based administration dashboard that simplifies sharding and replication management, and offers information about data distibution as well as monitoring and data exploration features. RethinkDB uses B-Tree indexes and makes a special effort to keep them and the metadata of each table in memory, so as to improve query performance. Cache memory size in RethinkDB is set automatically in accordance with total memory size, but can also be manually configured.

RethinkDB implements sharding automatically, evenly distributing documents into shards using a range sharding algorithm [30] based on table statistics and parameterized on the table's primary key. In the event of primary replica failure, one of the secondaries is arbitrarily selected as primary, as long as a majority of the voting replicas for each shard remain available. To overcome a single-node failure a RethinkDB cluster must be configured with three or more servers, tables must be configured to use three or more replicas, and a majority of replicas should be available for each table. If the failed server is a backup, performance is not affected as progress is possible with a surviving majority.

An important management feature offered by RethinkDB is its programmatic cluster management API [12] accessible via ReQL queries to automate different types of reconfiguration. Internally, RethinkDB maintains a number of system tables that expose database settings and the internal state of the cluster. An administrator can use ReQL through a language like Javascript to query and interact with system tables using conventional ReQL commands, just as with any other RethinkDB table. To exercise granular control over sharding and replication, the administrator uses the *table_config* table, which contains documents with information about the tables in a database cluster, including details on sharding and replication settings.

RethinkDB offers three primary commands for managing shards and replicas:

- *table_create*, for creating a table with a specified number of shards and replicas.
- *reconfigure*, for changing sharding and replication settings of a table (number of shards, number of replicas, identity of primary), after determining the current status via *table_config*. The location of replicas can be changed via updates to *table_config*.
- *rebalance*, for achieving a balanced assignment of documents to shards, after measuring the distribution of primary keys within a table and picking split points.

Another feature offered by the RethinkDB cluster management API is *server tags*, used to associate table replicas with specific servers, physical machines, or data centers.

III. Design and Implementation

Our cross-layer management architecture comprises a container management system (CMS) at the lower layer, managing the infrastructure, and a containerized NoSQL data store at the top, deployed over a number of pods (Figure 2). While our prototype is based on two specific technologies, Kubernetes and RethinkDB, the architecture is more general and can accommodate other CMS and data stores, such as the Amazon EC2 Container Service (ECS) and MongoDB respectively. The *manager* intermediates between the two layers. Its main purpose is to set goals regarding data store deployment (e.g., number of servers (pods) backing a cluster) and to reconfigure a cluster when notified about events of interest, namely

- New server (pod) joined the cluster
- Server about to be decommissioned
- Server about to experience performance interference

For concreteness and without loss of generality, in what follows we describe our implementation using Kubernetes and RethinkDB concepts. Pod resources for a data store cluster are described in a Kubernetes *deployment* spec. The manager uses Kubernetes' *ReplicaSet* mechanism [31] to set the desired goal on the number of pods that should be available to support the RethinkDB cluster. Kubernetes performs liveness monitoring through its internal heartbeat mechanism

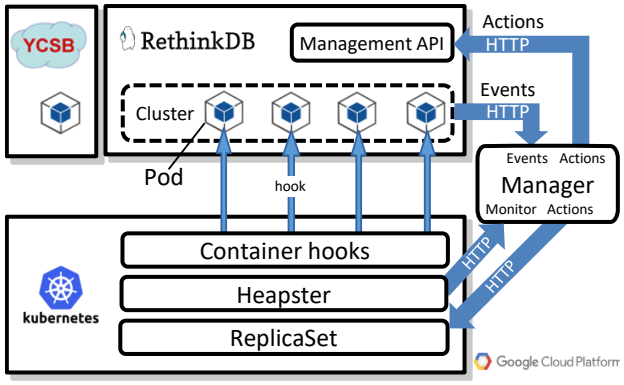


Fig. 2. Cross-layer management architecture

as well as application-specific *liveness probes* [32]. After detecting a pod failure, it deploys a new RethinkDB pod to maintain the replication[2] goal. The manager is implemented as a containerized node.js RESTful Web service, deployed in a dedicated pod managed by Kubernetes. The specific events it handles and actions it undertakes are described below:

Events. The manager exposes endpoints invoked by *container hooks* to communicate events relating to the infrastructure (pods) where RethinkDB is deployed. A hook provides information to a container about events in the container's management lifecycle. As soon as a hook is received by a container's hook handler, a HTTP call is executed to the corresponding manager endpoint. We utilize two existing types of hooks:

- PRESTOP, raised before termination of a container. The container will not terminate until the manager acknowledges the HTTP call.
- POSTSTART, raised after the creation of a container.

We envision a third type of hook:

- PERFWARNING, carrying notification of impending performance interference that will impact a specific pod.

The PERFWARNING hook will enable a manager to respond with proactive adaptation actions aiming to mask the impact of such interference. To evaluate the benefits of a PERFWARNING hook, we approximate its functionality at the manager by periodically polling the *Heapster* container cluster monitoring service (Figure 2) to detect when a pod P_i in the cluster becomes resource-limited. Heapster, natively supported in Kubernetes, collects cluster compute resource usage metrics and exports them via REST endpoints. By polling Heapster periodically, the manager can detect skewed overload conditions, namely a pod P_i exhibiting high utilization (exceeding 80%) while also exceeding the cluster average utilization by 30%:

$U(P_i) \geq 80\%$ and
$U(P_i) \geq 130\% \times \text{average}(U(P_1), \ldots, U(P_n))$

[2]In this context, replication refers to the number of RethinkDB servers backing a cluster, not the number of data replicas backing a shard.

The constants 80% and 130% where empirically determined to work well in detecting pod overload conditions, distinguishing from a uniformly overloaded system that would justify increasing capacity through an elasticity action. Determining the exact values that would maximize the benefit from adaptation actions is a subject of ongoing research.

Actions. A PRESTOP hook is handled by the manager by demoting[3] all replicas on that pod. This will mask the impact of the pod's impending decommission: With success of read/write operations on RethinkDB (as well as many other replicated data stores) being dependent on the availability of a majority of replicas, loss of a single backup replica per shard (data partition) will not affect performance. Our evaluation (Sections IV-B1 and IV-B2) shows that the impact of reconfiguration (brief unavailability of a shard) is lower than that of unannounced node crash, as the latter includes the additional overhead (timeouts) of detecting the failure.

A POSTSTART may convey different types of events: First, it may be that the new pod is a replacement of a recent crash (i.e., restores the size of the cluster). The master needs to be involved here because although a newly bootstrapping RethinkDB server can check if the specific RethinkDB cluster it intends to join is online and then join it, the RethinkDB runtime cannot automatically reconfigure the cluster to utilize the new node. Thus in the event of a POSTSTART hook from a replacement node, the manager balances the cluster by migrating replicas to the new pod, restoring pre-crash state. If the new pod is increasing cluster size, which happens when expanding capacity during an elasticity action, the manager will decide to perform replica migrations to the new pod so as to increase the overall capacity of the cluster.

In the event that the manager detects an overload condition on one of the pods using monitoring (approximating a PERFWARNING event) in an otherwise normally-loaded cluster, it decides to perform reconfigurations demoting primaries hosted on the overloaded node. As demonstrated in our evaluation (Sections IV-B3 and IV-B4), this provides rapid response to the performance degradation experienced by the application.

The use of reconfiguration actions to rapidly ease an overload condition on a single node does have a drawback if the overload condition is expected to last a long time. Since replicas on the overloaded node are not updated frequently, a subsequent failure of a second node may leave some of the shards with less than a majority of live *and* up-to-date replicas, reducing their performance. This eventuality is demonstrated in the experiment of Section IV-B5. Thus, if the manager receives feedback that an isolated overload condition is expected to last a long time, it should opt to reconfigure *and* migrate replicas outside of an overloaded node. Implementation of such functionality would require the ability to predict the duration of performance interference and convey it as a parameter to PERFWARNING.

[3]Switch their role from primary to backup, electing other primaries.

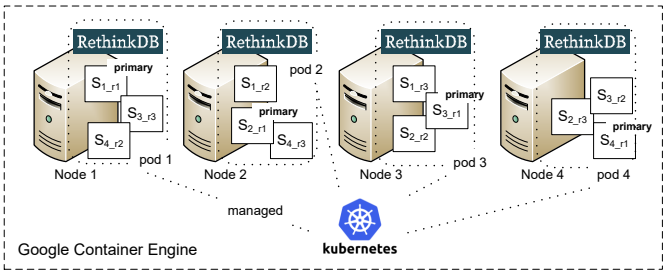


Fig. 3. The experimental testbed

The manager invokes the RethinkDB management API to perform the following operations:

- *movePrimariesFromServer*: Demote primaries of all shards hosted at a given server. This is performed by retrieving the table configuration through *table_config* and then re-assigning shard primaries to other servers in a balanced manner by applying updates to *table_config*.
- *movePrimariesFromNode*: Demote primaries of all shards hosted at all servers of a given node. This is performed using the Kubernetes API to retrieve the servers (pods) placed at a given node and then applying *movePrimaries-FromServer* on each of them.
- *addServer*: Include a new server to the configuration of a table using the *reconfigure* command.
- *createReplica*: Create replica of a shard on a specific server (pod) updating the *table_config*.

The above described management architecture supports the scenarios evaluated in this paper:

Unscheduled downtime of a node. RethinkDB is able to adapt to a crashed server (pod) without the engagement of the manager through standard replication recovery mechanisms. When a new pod is made available by Kubernetes, the manager is notified through the POSTSTART hook, understands that it is replacement server, and proceeds to reconfigure the cluster.

Scheduled downtime of a node. In this case, the manager is notified when a pod is about to be decommissioned through the PRESTOP hook. It then reconfigures RethinkDB to demote all primaries hosted at that pod. When a new pod becomes available, actions are similar to the previous case.

Hot-spot node(s) impacting application performance. Through the Heapster monitoring API, the manager is able to detect a hot-spot on a node. In such case it demotes primaries hosted there. If the manager detects a uniformly overloaded status across the cluster, it decides to allocate a new pod and migrate a subset of replicas to it.

IV. EVALUATION

A. Experimental testbed

The experiments were conducted on a 4-node container cluster on Google Cloud Engine (GCE). Each node has 2 vCPUs (2.6 GHz Intel Xeon E5 with hyperthreading), 13GB of RAM, and a solid-state drive (SSD) for persistent storage. Each GCE node hosts one Kubernetes pod containing a RethinkDB server (Figure 3). RethinkDB is configured for 4 shards. Each pod initially contains 3 replicas of 3 different shards. Each replica is denoted S_{i_rj}, which stands for "shard i, replica j", where i $\in \{1,2,3,4\}$ and j $\in \{1,2,3\}$. The YCSB benchmark [33] is configured for Workload A (50% reads, 50% updates) [34], 16 client threads, and load target of 1000 operations/sec. RethinkDB is configured for soft durability (writes are acknowledged immediately). The software versions used are Kubernetes 1.4.8 and RethinkDB 2.3.5.

B. Experimental results

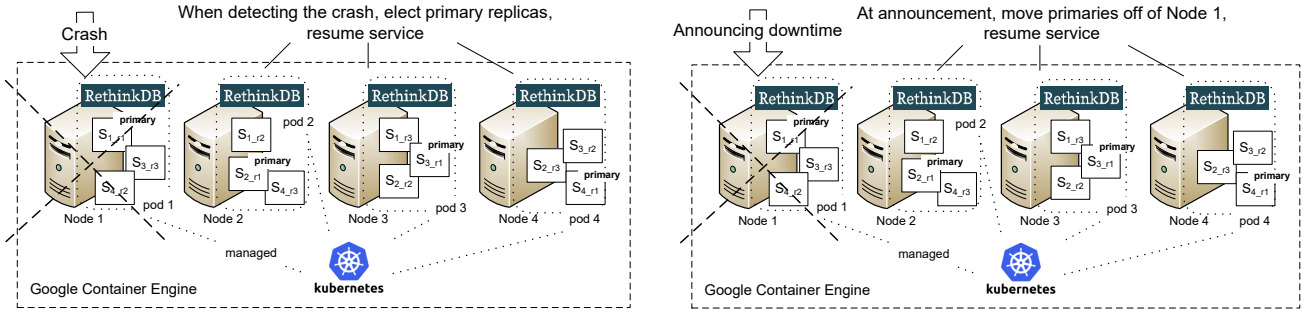
We perform a series of experiments aiming to address the following two research questions:

1) Does proactive adaptivity in the case of previously announced downtime of a node reduce the performance impact that would otherwise be experienced if the downtime was unscheduled (e.g., a crash)?
2) Can load balancing actions via replica group reconfigurations reduce the performance impact of *hot-spots* (overloaded nodes) on data store performance?

We first look into the benefits of proactive vs. reactive adaptation when a RethinkDB server is decommissioned. We note that proactive adaptation is only possible with a previous announcement of downtime, while reactive adaptation makes sense only in the case of unscheduled downtime (e.g., crash).

1) Reactive adaptivity to unscheduled downtime of a node: This scenario is illustrated in Figure 4a. Figure 5a depicts YCSB latency and throughput during execution. At 150s a server is decommissioned ("pod shutdown") causing all replicas hosted there to crash. Crashed primary replicas cause a brief period of unavailability, out of which RethinkDB recovers (at 165s) by electing a new primary for those crashed. At 180s a new node is made available by Kubernetes (via the ReplicaSet mechanism) and joins the RethinkDB cluster. The manager initiates the creation of replicas at the new node via a reconfiguration of the YCSB table, causing RethinkDB to start *backfilling* (transfering state to) simultaneously all the replicas. After the replicas are created, some of them are assigned primaries to restore balance. The performance impact of this process is seen at 220s–260s.

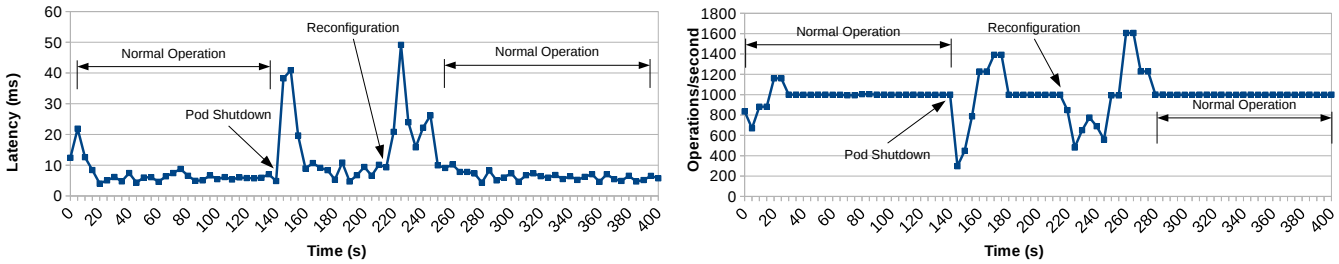
2) Proactive adaptivity to scheduled node downtime: A different scenario involving a scheduled maintenance activity that will take down a node (Node 1) is illustrated in Figure 4b. Unlike the previous case, the manager is notified in advance and thus reconfigures the RethinkDB table to demote any primaries hosted at Node 1 (electing primaries at other nodes), aiming to reduce impact on cluster performance. In Figure 6a we observe that reconfiguration takes place at 140s. Decommissioning Node 1 at 170s has only a small impact on performance.



(a) Reactive adaptivity to unscheduled downtime of a node

(b) Proactive adaptivity to scheduled downtime of a node

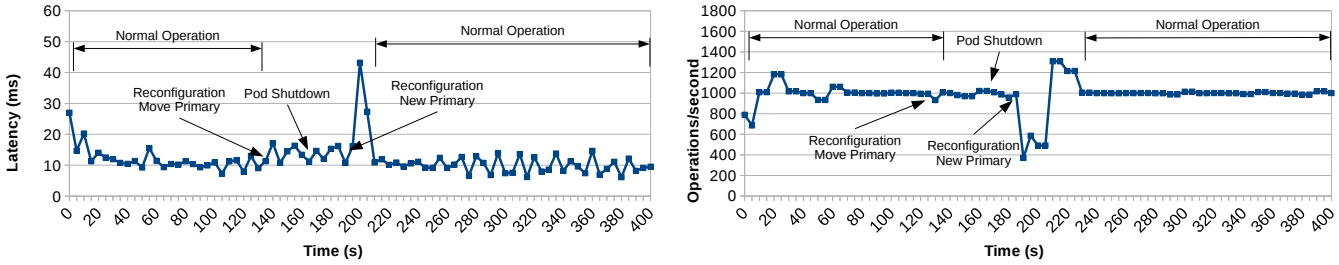
Fig. 4. Reactive vs. proactive adaptivity to downtime of a node (Node 1)



(a) Mean operation latency during reconfiguration phase.

(b) Throughput during the reconfiguration and restoration phase.

Fig. 5. Reactive adaptivity to unscheduled downtime of a node (scenario of Figure 4a)



(a) Mean operation latency during reconfiguration phase.

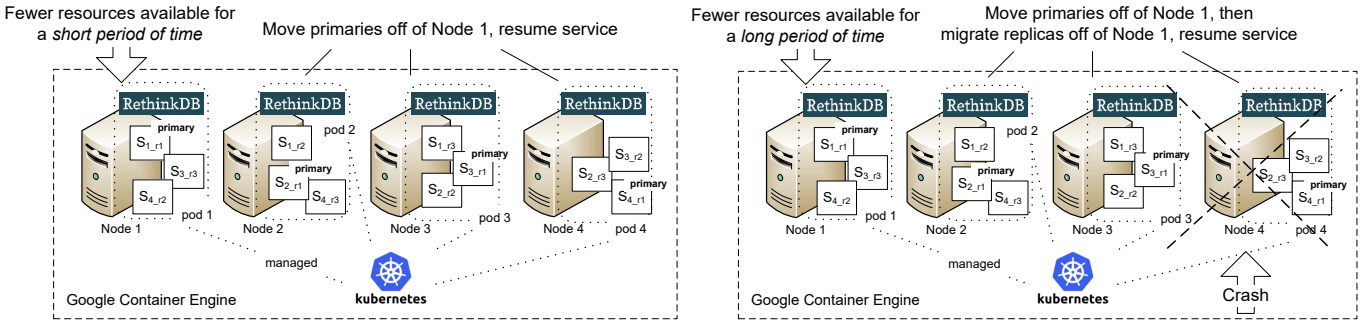
(b) Throughput during the reconfiguration and restoration phase.

Fig. 6. Proactive adaptivity to scheduled downtime of a node (scenario of Figure 4b)

The new server joins the cluster at 200s, at which point the manager starts backfilling replicas there. Finally, the manager reconfigures the table again at 195s to promote an appropriate number of backup replicas into primaries (one in this case) on the new server to reach a well-balanced table configuration. Comparing Figure 5 to Figure 6 makes it clear that proactive adaptivity leads to smoother performance overall.

We next turn our attention to adaption actions performed by the manager in the context of a load imbalance on some node in the cluster. To evaluate the benefits of adaptation in this case, we set up a third experiment in which a resource-intensive process kicks in on one of the nodes in the cluster at some point in time, draining CPU resources and impacting the overall performance of the cluster. The cause of the impact is the fact that the affected node hosts the primary replica of one of the shards in the cluster, and thus all operations addressing this shard are being delayed. This challenge can be addressed as described below:

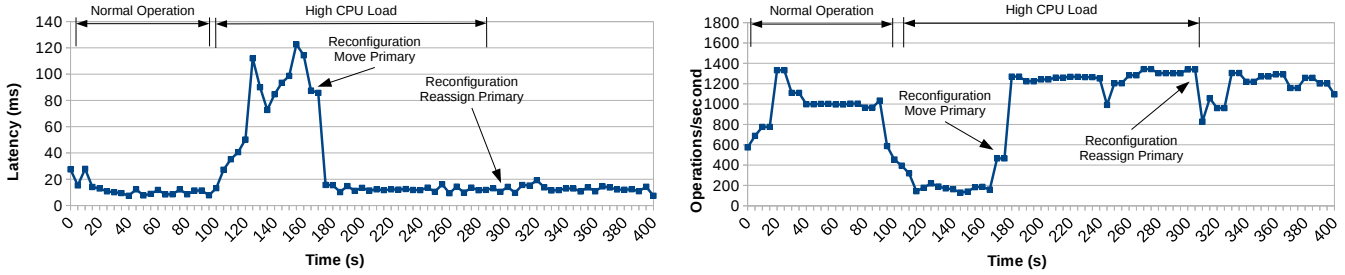
3) Offloading a brief hot-spot via reconfiguration: The next scenario is illustrated in Figure 7a. As shown in Figure 8a, at 100s a resource-intensive activity drains resources on a cluster node, with significant impact on overall cluster performance as operation latency surges from about 5ms to 80-120ms while throughput drops from 1000 ops/sec to 200 ops/sec. Drastically reduced performance is due to those shards (replica groups) whose primaries are hosted at the affected node. At about 170s the manager decides to start a reconfiguration of RethinkDB cluster nodes, demoting any primaries hosted at the affected node and electing primaries in other nodes in the cluster



(a) Temporary offload via replica-group reconfiguration

(b) Long-term offload: replica reconfiguration *and* migration

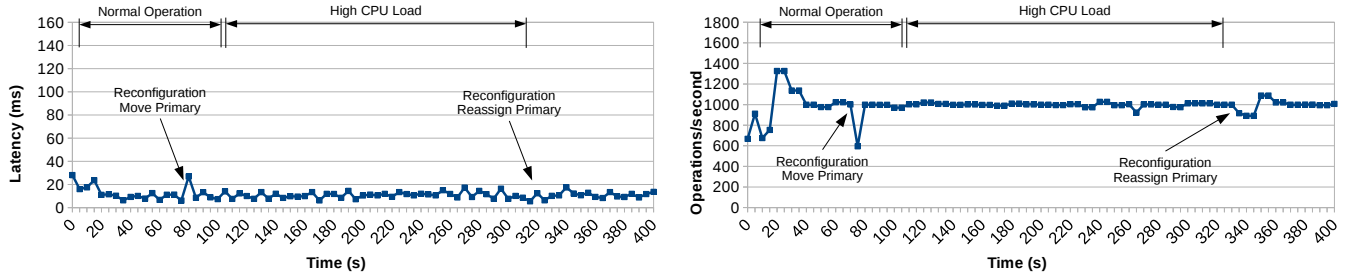
Fig. 7. Offloading a hot-spot (Node 1): Brief vs. long-lasting hot-spot



(a) Mean operation latency during reconfiguration phase

(b) Throughput during the reconfiguration and restoration phase

Fig. 8. Offloading a brief hot-spot via reconfiguration (scenario of Figure 7a)



(a) Mean operation latency during reconfiguration phase

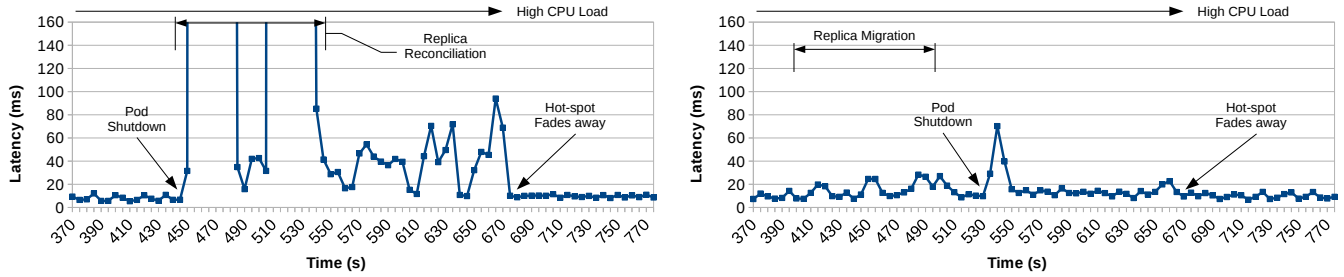
(b) Throughput during the reconfiguration and restoration phase

Fig. 9. Offloading a brief hot-spot via proactive reconfiguration (scenario of Figure 7a)

(making sure to spread load uniformly). The affected node now hosts only backup replicas, no longer posing an impact on performance: Each shard is backed by three replicas, and any two nodes (some majority) are sufficient for reads/writes to the shard to make progress.

After reconfiguration of the cluster, latency is restored to about 10ms, significantly better than during the surge (80-120ms). As the YCSB node is producing load at a constant rate of 1000 ops/sec, service-side request queues are building a backlog during the surge. The throughput rise at 190s is due to the emptying of those queues. At 310s, the manager initiates a reconfiguration of the cluster back to its original state, promoting a replica to a primary on Node 1, better balancing load. This brings latency down to pre-surge levels of about 10ms. CPU utilization at Node 1 during the reconfiguration and restoration phase is illustrated in Figure 11.

4) Offloading a brief hot-spot via proactive reconfiguration: This scenario shows the benefit of applying the reconfiguration action proactively, in advance of the hot-spot. As shown in Figure 9, a reconfiguration is applied at 80s prior to the resource-intensive activity coming into effect at that node at 110s. As our implementation does not yet support a predictive implementation of PERFWARNING event (Section III), this is triggered manually in this case. The proactive action is able to mask completely the adverse effect of the hot-spot on application performance. After the end of the resource-intensive activity, another reconfiguration restores the replica roles (promoting a primary) on the affected node.



(a) Mean operation latency during server failover without migrating backups (b) Mean operation latency during server failover after migrating backups

Fig. 10. Offloading a longer hot-spot via reconfiguration with (optional) replica migration (scenario of Figure 7b)

Reconfiguration by reassigning shard primaries is an effective short-term remedy to maintain performance in the presence of a temporary hot-spot. However, a downside is that it leaves the system vulnerable to service unavailability in the case of a subsequent crash, as is demonstrated next:

5) Offloading a longer-term hot-spot via reconfiguration and replica migration: The next scenario is illustrated in Figure 7b. This experiment focuses on the time following the movement of primaries off of Node 1 in the previous experiment and while the hot-spot is still on (lasts longer in this case). To focus on the behavior during a subsequent crash we show results starting at 370s into the run. We focus on the two possible outcomes that may occur when a second node (Node 4) crashes, depending on whether new instances of the replicas located in Node 1 have been created on other nodes prior to the crash on Node 4 (Figure 10b) or not (Figure 10a).

In Figure 10a we observe that the pod crash on Node 4 at about 450s results in severe performance degradation. The crash of Node 4 brings down 3 replicas (of different shards), one of which is a primary and two are backups. Shards with replicas in Nodes 1 and 4 are now left with two replicas, one of which outdated (on Node 1) since replicas there were not updated quickly enough. Reads are thus delayed in the interval 450s-550s while the slow replicas are "catching up" (being reconciled) through backfilling. Even after reconciliation (550s-680s), replicas at the hot-spot node continue to delay progress, since the shard depends on them to form a majority.

We next consider the case where the manager *migrates* replicas out of Node 1 (400s-500s in Figure 10b), an action that poses a slight latency cost during data transfer. The migration lasts for about 100s as the replicas are migrated one by one to reduce the performance impact. However, crash of a pod at 535s has now little effect on performance, as all shards have at least two up-to-date replicas to satisfy read and write operations from. Replica migrations are a worthwhile action in this case since the hot-spot lasted longer and the system was more vulnerable to a subsequent crash.

When possible, providing the manager with knowledge on the time duration of a hot-spot can guide the choice of an appropriate policy: reconfigure shards (demote primaries on a hot-spot node) if the hot-spot is expected to be short, otherwise reconfigure *and* migrate replicas out of a hot-spot node.

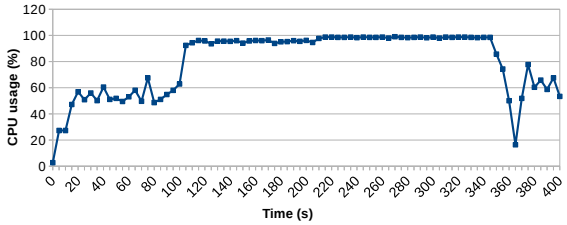


Fig. 11. CPU utilization on hot-spot node (including all activities)

V. Conclusions

In this paper we presented a cross-layer management architecture that leverages advance notification about infrastructure activities affecting data store performance, such as upcoming decommissioning of a node or performance interference affecting a node, to perform lightweight adaptation actions that in many cases have a rapid positive impact compared to heavyweight data migrations. The functionality achieved was previously infeasible since it relies on advance notifications about infrastructure-level events, unavailable to middleware systems over standard virtualized infrastructures. Notification about nodes going down is provided by Kubernetes through an existing notification mechanism, container hooks, also available in other container management systems. Notifications about performance interference events are not immediately supported as hooks, hence we approximated them in our prototype via monitoring. The prototype bridges between the Kubernetes container management system and containerized deployments of the RethinkDB NoSQL data store. Our evaluation on the Google Container Engine demonstrates that cross-layer management delivers the availability benefits of proactively handling the departure of a RethinkDB server, as well as the performance benefits of masking the performance impact of a hot-spot through lightweight replica reconfigurations.

References

[1] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, (Bordeaux, France), April 22-24, 2015.

[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Queue*, vol. 14, pp. 10:70–10:93, Jan. 2016.
[3] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (Prague, Czech Republic), April 14-17, 2013.
[4] "Kubernetes." http://kubernetes.io/. Accessed: 2017-01-07.
[5] "Amazon EC2 Container Service." https://aws.amazon.com/ecs/, note = Accessed: 2017-01-07.
[6] "Google Container Engine." https://cloud.google.com/container-engine/, note = Accessed: 2017-01-07.
[7] "IBM Bluemix Container Service." https://www.ibm.com/cloud-computing/bluemix/containers, note = Accessed: 2017-01-07.
[8] M. Ghosh, W. Wang, G. Holla, and I. Gupta, "Morphus: Supporting Online Reconfigurations in Sharded NoSQL Systems," in *Proceedings of the 2015 IEEE International Conference on Autonomic Computing*, ICAC '15, (Washington, DC, USA), July 7-10, 2015.
[9] P. Garefalakis, P. Papadopoulos, and K. Magoutis, "ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees," in *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, October 6-9, 2014*, (Nara, Japan).
[10] "RethinkDB." https://www.rethinkdb.com/. Accessed: 2017-01-07.
[11] "MongoDB." https://www.mongodb.com/. Accessed: 2017-01-07.
[12] "RethinkDB cluster management API." https://rethinkdb.com/blog/1.16-release/. Accessed: 2017-01-07.
[13] "MongoDB replication." https://docs.mongodb.com/manual/replication/. Accessed: 2017-01-07.
[14] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving Large-scale Batch Computed Data with Project Voldemort," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, (San Jose, CA), February 14-17, 2012.
[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (Stevenson, WA, USA), October 14-17, 2007.
[16] "Basho Riak NoSQL database." http://docs.basho.com/riak/kv/. Accessed: 2017-01-07.
[17] "Cloud Native Computing Foundation." https://www.cncf.io/. Accessed: 2017-01-07.
[18] "Autoscaling lifecycle hooks." http://docs.aws.amazon.com/autoscaling/latest/userguide/lifecycle-hooks.html. Accessed: 2017-01-07.
[19] M. Isard, "Autopilot: Automatic data center management," *SIGOPS Operating Systems Review*, vol. 41, pp. 60–67, Apr. 2007.
[20] K. Bhargavan, A. Gordon, T. Harris, and P. Toft, "The Rise and Rise of the Declarative Datacentre," Tech. Rep. MSR-TR-2008-61, Microsoft Research, May 2008.
[21] A. Papaioannou, D. Metallidis, and K. Magoutis, "Cross-layer management of distributed applications on multi-clouds," in *IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, 11-15 May, 2015*, (Ottawa, ON, Canada), May 11-15, 2015.
[22] K. Magoutis, M. Devarakonda, N. Joukov, and N. G. Vogl, "Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems," *IBM J. Res. Dev.*, vol. 52, no. 4, 2008.
[23] V. V. Cogo, A. Nogueira, J. Sousa, M. Pasin, H. P. Reiser, and A. N. Bessani, "FITCH: supporting adaptive replicated services in the cloud," in *Procedings of 13th IFIP International Conference Distributed Applications and Interoperable Systems (DAIS 2013)*, (Florence, Italy), June 3-5, 2013.
[24] I. Konstantinou, D. Tsoumakos, I. Mytilinis, and N. Koziris, "DBalancer: Distributed Load Balancing for NoSQL Data-stores," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), June 22-27, 2013.
[25] I. Kitsos, A. Papaioannou, N. Tsikoudis, and K. Magoutis, "Adapting data-intensive workloads to generic allocation policies in cloud infrastructures," in *Proceedings of Network Operations and Management Symposium (NOMS)*, (Hawaii, USA), pp. 25–33, April 16-20, 2012.
[26] L. Wang, J. Xu, and M. Zhao, "Application-aware cross-layer virtual machine resource management," in *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, (San Jose, CA, USA), September 17-21, 2012.
[27] "Kubernetes." https://en.wikipedia.org/wiki/Kubernetes. Accessed: 2017-01-07.
[28] "etcd distributed key-value store." https://github.com/coreos/etcd. Accessed: 2017-01-07.
[29] "ReQL." https://www.rethinkdb.com/docs/introduction-to-reql/. Accessed: 2017-01-07.
[30] "RethinkDB Architecture." https://www.rethinkdb.com/docs/architecture/. Accessed: 2017-01-07.
[31] "Kubernetes ReplicaSets." http://kubernetes.io/docs/user-guide/replicasets/. Accessed: 2017-01-07.
[32] "Kubernetes liveness probe." https://kubernetes.io/docs/tasks/. Accessed: 2017-01-07.
[33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (Indianapolis, Indiana, USA), June 10-11, 2010.
[34] "YCSB Core Workloads." https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. Accessed: 2017-01-07.