ulm university universität
uulm

Gibbon: An Availability Evaluation Framework For Distributed Databases

Daniel Seybold, Christopher B. Hauser, Simon Volpert, and Jörg Domaschka

This is a post-peer-review, pre-copyedit version of an article published in Lecture Notes in Computer Science book series (LNCS, volume 10574). The final authenticated version is available online at
https://doi.org/10.1007/978-3-319-69459-7_3

Gibbon: An Availability Evaluation Framework For Distributed Databases

Daniel Seybold, Christopher B. Hauser, Simon Volpert, and Jörg Domaschka

Ulm University, Institute of Information Resource Management, Ulm, Germany
{daniel.seybold, christopher.hauser, simon.volpert,
joerg.domaschka}@uni-ulm.de

Abstract. Driven by new application domains, the database management systems (DBMSs) landscape has significantly evolved from single node DBMS to distributed database management systems (DDBMSs). In parallel, cloud computing became the preferred solution to run distributed applications. Hence, modern DDBMSs are designed to run in the cloud. Yet, in distributed systems the probability of failures is the higher the more entities are involved and by using cloud resources the probability of failures increases even more. Therefore, DDBMSs apply data replication across multiple nodes to provide high availability. Yet, high availability limits consistency or partition tolerance as stated by the CAP theorem. As the decision for two of the three attributes in not binary, the heterogeneous landscape of DDBMSs gets even more complex when it comes to their high availability mechanisms. Hence, the selection of a high available DDBMS to run in the cloud becomes a very challenging task, as supportive evaluation frameworks are not yet available. In order to ease the selection and increase the trust in running DDBMSs in the cloud, we present the Gibbon framework, a novel availability evaluation framework for DDBMSs. Gibbon defines quantifiable availability metrics, a customisable evaluation methodology and a novel evaluation framework architecture. Gibbon is discussed by an availability evaluation of MongoDB, analysing the take over and recovery time.

Keywords: distributed database, database evaluation, high availability, NoSQL, cloud

1 Introduction

The landscape of database management systems (DBMSs) has evolved significantly over the last decade, especially when it comes to large-scale DBMSs installations. While relational database management systems (RDBMS) have been the common choice for persisting data over decades, the raise of the Web and new application domains such as *Big Data* and *Internet of Things (IoT)* drive the need for novel DBMS approaches. NoSQL and only recently NewSQL DBMSs [15] have evolved. Such DBMSs are often designed as a distributed database management system (DDBMS) spread out over multiple *database nodes* and supposed to run on commodity or even virtualised hardware.

Due to their distributed architecture, DDBMSs support horizontal scalability and consequently the dynamic allocation and usage of compute, storage and network resources [1] based on the actual demand. This fact makes Infrastructure as a Service (IaaS) cloud platforms a suited choice for running DDBMSs, as it provides elastically, on-demand, self-service resource provisioning [19].

Even though distributed architectures can be used to improve the availability of the overall system, this is not automatically the case. In particular, in distributed systems the probability of failures is the higher the more entities are involved, *i.e.* in the case of DDBMSs the more data nodes are used. When cloud resources are used, the situation is worsened, as failures on lower level may affect multiple virtualised resources and cause mass failures [13, 16].

With respect to DDBMSs, the common approach to availability is to replicate data items across multiple database nodes to ensure high availability in case of resource failures. However, the desire for availability is hindered by the CAP theorem stating that availability is achieved by scarifying consistency guarantees or partition tolerance [6]. However, the choice between two of these three attributes is not a binary decision and offers multiple trade-offs [5]. This has led to a very heterogeneous DDBMS landscape not only with respect to the sheer number of systems, but also the availability mechanisms they provide [11].

Hence, we find ourselves in the situation that more DDBMSs exist than ever, each of them promising availability features and often enough even high availability. Further, many of these DDBMS are operated on IaaS infrastructures with an increased risk of mass failures. Yet, for none of the DDBMSs it is known how it actually behaves under failure conditions and how the failure condition affects the availability of the DDBMSs. At the same time, no supportive frameworks for evaluating availability of DDBMSs exist [24] and in consequence, selecting a DDBMS becomes a gamble for users if availability is a major selection criterion.

In order to increase the trust in running DDBMSs on cloud resources and easing the selection of high available DDBMSs, we present Gibbon, a novel framework for evaluating the availability of DDBMSs. It explicitly supports cloud failure scenarios. Hence, our contribution is threefold: (1) we identify quantifiable metrics to evaluate availability; (2) we define an extensible evaluation methodology; (3) we present a novel availability evaluation framework architecture.

The remainder is structured as follows: Section 2 introduces the background on availability, DDBMS and cloud computing, while Section 3 analyses the impact of failures. Section 4 defines the availability metrics while Section 5 presents the evaluation methodology. Section 6 presents the framework architecture. Section 7 discusses the presented framework and Section 8 presents related work. Section 9 concludes.

2 Background

In order to consolidate the context of the Gibbon framework, we introduce in this section the background on DDBMSs, availability, and DDBMSs on IaaS.

2.1 Distributed Database Systems

Per definition, a DBMS manages data items grouped in collections or databases. For DDBMS these data items are spread out over multiple database nodes each of which runs management logic. Hence, the databases nodes communicate and cooperate in order to realise the expected functionality.

The use of distribution has two conceptionally unrelated benefits: (i) more data can be stored and processed, as the overall available capacity is the sum of the capacity of the individual database nodes. In this usage scenario the *sharding (partitioning) strategy* defines which data items are stored on which of the database nodes. (ii) data items can be stored redundantly on multiple database nodes protecting them against failures of individual database nodes. A *replication degree* of n denotes that a data item is stored n times in the system.

Replication When sharding is used without replication, no tolerance against node failures exists. On the other hand, using replication without sharding means that all data is available on all database nodes. This is also referred to as full replication. When sharding and replication is used in parallel, each database node will contain only a subset of all data items [9]. Depending on the *replication strategy* [22] a user is allowed to interact with only one of the replicas (master-slave replication) or all of them (multi-master).
In the master-slave approach one of the n physical copies of a data item has the master role. Only this item can be changed by users. It is the task of the database node hosting this item to synchronize slave nodes. The latter only execute read requests. A failure of the master will require the re-election of a new master amongst the remaining slaves. In the multi-master approach, any replica can be updated and the hosting database nodes have to coordinate changes.
Geo-replication caters for mirroring the entire DDBMS cluster to a different location in order to protect the data against catastrophic events.
Replica Consistency Having multiple copies of the same data item in the system requires to keep the copies in sync with each other. This is particularly true for multi-master approaches. Here, two approaches exist: synchronous propagation ensures consistency is coordinated amongst all database nodes hosting a replica before any change is confirmed to a client. Asynchronous propagation in turn delays this so that multiple diverging copies of the item can exist in the system and clients can perform conflicting updates unnoticed.
Node and Task Types Besides storing data items, a DDBMS has several other tasks to do: *management tasks* keep track of the location of data items, routing queries to the right destination, and detecting node failures. *Query tasks* process queries issued by the client and interact with the database nodes according to the distribution information the management task stores. Depending on the actual DDBMS these tasks are executed by all database nodes in peer-to-peer manner, isolated in separate nodes, or even part of the database driver used by the client.
Storage Models For DBMS three top-level categories are currently in use [15]: *relational*, *NoSQL* and *NewSQL* data stores. Relational data stores target transactional workloads, providing strong consistency guarantees based on the ACID

paradigm [17]. Hence, relational data stores are originally designed as single server DBMS, which have been extended lately to support distribution (*e.g.* MySQL Cluster[1]). NoSQL storage models can be classified into key-value stores, document-oriented stores, column-oriented stores and graph-oriented stores. Compared to relational, their consistency guarantees are weaker and tend to towards BASE [21]. This weakening consistency makes those DDBMS better suited for distributed architectures and eases the realisation of features such as scalability and elasticity. NewSQL data stores are inspired by the relational data model and target strong consistency in conjunction with a distributed architecture.

2.2 Availability for DDBMSs

Generally, *availability* is defined as *the degree to which a system is operational and accessible when required for use* [14]. Besides *reliability* (the *measure of the continuity of correct service"* [3]), availability is the main pillar of many fault-tolerant implementations [11].

In this paper, we solely focus on the availability aspect and assume that DDBMSs are reliable, *i.e.* work as specified. As our primary metric, we use what Zhong et al. call *expected service availability* and define availability of a DDBMS as the proportion of all successful requests [27] over all requests.

The availability of the DDBMS can be affected by two kinds of conditions [11]: (i) A high number of requests issued concurrently by clients, overloading the DDBMS such that the requests of clients cannot be handled at all or are handled with an unacceptable $latency > \Delta t$. (ii) Failures occur that impact network connectivity or availability of data items. The failure scenarios we consider are subject to Section 2.3.

2.3 Cloud Infrastructure

IaaS clouds have become a preferable way to run DDBMSs. Due to its cloud nature, IaaS offers more flexibility than bare metal resources. IaaS provides processing, storage, and network to run arbitrary software [19]. The processing and storage resources are typically encapsulated in a virtual machine (VM) entity that also includes the operating system (OS). VMs run on hypervisors on top of the physical infrastructure of the IaaS provider. As cloud providers typically operate multiple data centres, IaaS eases to span DDBMSs across different geographical locations. The location of cloud resources can be classified into geographical locations *(regions)*, data centres inside a region *(availability zones)*, *racks* inside a data centre and *physical hosts* inside a rack.

This set-up heavily influences availability as failures on different levels of that stack can have impact on individual database nodes (running in one VM), but also on larger sets of them. For instance, the failure of a hypervisor will lead to the failure of all VMs on that hypervisor.

[1] `https://www.mysql.com/products/cluster/`

An exemplary DDBMS deployment on IaaS is depicted in Figure 1. Here, an 8-node DDBMS is deployed across two regions of one cloud provider. Each database node is placed on a VM and the VMs rely on different availability zones of the respective region. The example also illustrates the use of heterogeneous physical hardware: availability zones B and C are built upon physical servers without disks and dedicated storage servers. Availability zones A and D are built upon servers with built-in disks.

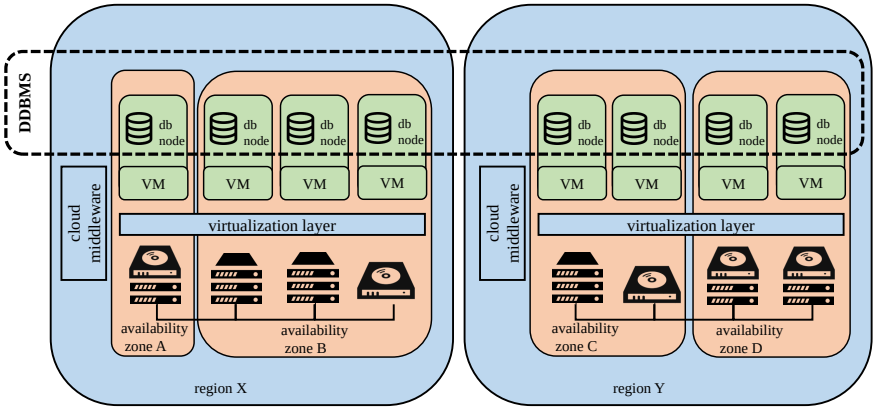

Fig. 1. DDBMS on IaaS

3 Failure Impact Analysis

Section 2.2 stated that two types of events impact the availability of DDBMSs: overload and failures. Dealing with overload has seen much attention in literature (*cf.* Section 8) so that this paper will focus on the latter that has barely received attention in the past [24]. In particular, our work addresses the capabilities of DDBMSs availability mechanisms to overcome failures in IaaS environments.

3.1 Replication for Availability

In Section 2.1, we introduced replication and partitioning as two major, but basically unrelated concepts used in DDBMSs. This section investigates the impact of their use under failure conditions.

No Replication Apparently, when a database node fails and no replication is used, all data items stored on that database node become unavailable. The impact on the overall availability of the DDBMS depends on the access pattern of the failed shard, but for uniform distribution, the availability will drop to $\frac{N-1}{N}$ while this database node is unavailable.

Table 1. DDBMS Failures in a Cloud Infrastructure

failure level	emulate	recovery
DDBMS node	forcibly terminate VM	replace VM
VM	forcibly terminate VM	replace VM
physical server	forcibly terminate all VMs on server	replace VMs/move zone
availability zone	forcibly terminate all VMs in zone	move zone or region
region	forcibly terminate all VMs in region	move region/provider
cloud provider	forcibly terminate VMs hosted by provider	move provider

Master-slave Replication When master-slave replication is used, the failure of a single database node has less impact, as copies of the data item are in the system. Yet, the process of detecting the failure and finding a new master for all data items from the failed database node needs time no matter if the new master is elected manually or automatically [11]. Hence, for uniform distribution of access, availability will still drop to $\frac{N-1}{N}$. Yet, hopefully for a shorter time.
Multi-master Replication When using multi-master replication the failure of a single database node does not affect the overall availability, as any other database node hosting a replica of the requested data item can be contacted. Nevertheless, depending on the driver implementation and the routing, a small portion of requests may fail, when they are connected with the failed database node at the time of failure.
Functional Nodes Some DDBMSs make use of additional hosts that function as entry points for clients or as a registry storing the mapping from data item to database node. The failure of these node also has impact on the overall availability. In particular, if configured wrong, the failure of any of those nodes can render the entire DDBMSs unavailable.

3.2 Failures and Recovery

This section investigates failures and recovery of DDBMSs hosted on Clouds. In particular, it derives how to emulate the failure of certain resources for the sake of evaluating availability metrics of different DDBMSs.

From Figure 1 we can see that a cloud-operated DDBMS sits on top of several layers of hard- and software. Hence, even assuming that both DDBMS code and the operating system surrounding it are correct, the large stack leaves opportunities that can go wrong: On the infrastructure level, servers, network links, network device, power supplies, or cooling may fail. Similarly, on software level, management software and hypervisors can fail; algorithms, network, and devices can be buggy, misconfigured, or in the process of being restarted.

Any of these failures can affect virtual machines and virtual networks, but also physical servers, physical networks, entire racks, complete availability zones, or even entire data centers. Table 1 lists these failure levels with a way to emulate the failure and an action that helps to recovery from the failure.

The failure of a *database node* or a *virtual machine* can be represented as virtual machine unavailability and can be emulated by forcibly terminating the

Table 2. Input Parameters

	Input Parameter	Description
Deployment	node replication	strategy (single-/multi-master, selection), replicas, laziness
	geo-replication	equals "node replication"
	partitioning	number of partitions, strategy (range, hash), data access (client, proxy, routing)
	resources	hierarchical infrastructure model of DDBMS (*cf.* Figure 1)
Evaluation	failure spec	number of failing nodes per level (cf. Table 1), number of failing nodes per types
	recovery Spec	restart policies, number of database nodes to add (per node type if existent)
	workload spec	requests per second, read/write request ratio, number of data items

virtual machine. Here, it is important to ensure that no additional clean-up tasks, *e.g.* closing network connections get executed. The failure of a *physical server* leads to the unavailability of all virtual machines hosted on that server. The failure of a full *availability zone* or even an entire *region* leads to the unavailability of multiple physical servers and hence unavailability of all hosted virtual machines.

4 Availability Metrics

From the previous sections we derive input parameters and output metrics to evaluate the availability of DDBMSs running on cloud infrastructures. Input parameters describe the deployment and evaluation specifications of the DDBMS, while output metrics describe the experienced availability after the input parameters have been applied. Hence, a tuple of input parameters and output metrics provide the base for the availability evaluation (*cf.* Section 5).

4.1 Input Parameters

The input parameters as listed in Table 2 comprise the *deployment* and *evaluation* specification. The deployment specification combines the replication and partitioning characteristics (*cf.* Section 3.1), failure and recovery characteristics (*cf.* Section 3.2) and deployment information of the DDBMS nodes. The first two input parameters define the *replication* setting of the DDBMS, *i.e.* node replication level and cross data centre geo-replication level. None, one, or both replication levels might be configured for the DDBMS under observance. The replication first is defined by the strategy, defines the amount of replicas the replication will have in normal, healthy state, and the update laziness, how the write requests are synchronized between replicas.

Partitioning defines the setting for data partitioning, if present. If partitioning takes place, the amount of partitions are specified, and how the distribution

Table 3. Output Metrics

	Output Metric	Description
Statistics	accessibility	DDBMS is accessible for client requests (read and write)
	performance impact	throughput (read/write requests per second), latency of requests
	request error rate	amount of failed requests due to data unavailability
Times	take over time	time until the failure spec is being masked by the DDBMS
	recovery time	time until the recovery spec is applied by the DDBMS

strategy of data items to partitions is handled (group based, range based, with hashing functions). For accessing the distributed data items, the data access is of importance, namely if the client connects to the correct partition directly, via a proxy or requests are routed automatically.

The *resources* parameter contains a full model of the allocated infrastructure resources for the DDBMS. This model includes all infrastructure entities involved from geographic location, to physical servers, virtual machines and DDBMS nodes (cf. Figure 1). If the DDBMS differentiates node types, the type is reflected in the resource information as well.

Further, Table 2 also presents the *evaluation specification* parameters. The *failure specification* parameter defines a failure scenario which will be emulated by the Gibbon framework. For each level of potential failures described in Table 1, the amount of failing resource entities and (optionally) the DDBMS node type to fail is defined. The *recovery specification* parameter on the other hand describes the emulated recovery plan such as restarting virtual machines or adding new nodes to the DDBMS. The input parameters can skip the optional failure recovery parameter.

For simulating failure and recovery specification, the DDBMS is continuously under an artificial workload, defined by the *workload specification* parameter. This parameter defines the amount of read and write requests, as well as the total amount of data items stored in the DDBMS.

4.2 Output Metrics

The output metrics presented in Table 3 represent the experienced availability after the input parameters deployment and evaluation specification have been applied. The output metrics are results of continuously monitoring the metrics while input parameters are applied.

The *accessibility* α defines if the database is still reachable by clients (accepting incoming connections) and accepts read and write requests. While accessibility represents `boolean` values over time, the *performance impact* ϕ represents the throughput the DDBMS can handle during the evaluation scenario, including the amount of requests and the latency for request handling. For instance the performance may be decreased if replicas are down. In case of node failures, not all data partitions of a DDBMS may be available until the failure is handled.

The *request error rate* ϵ describes as output metric the amount of failed requests due to data unavailability over the evaluation time.

The output metrics *take over time* and *recovery time* specify the measured time the DDBMS required to identify the applied failure specification, and the time it takes to apply the recovery specification. The accessibility, the performance impact and the data loss rate are time series values over the time the evaluation scenario is being applied. These values are measured periodically at runtime, are then aggregated and statistically represented in an percentile ranking over the time separately for the time it takes to apply i) the failure specification and ii) the recovery specification.

From the described output metrics, the overall availability metric of the evaluated DDBMS can be calculated. From the output metrics, only a configurable amount of percentiles (e.g. > 90) are considered. Each of the three metrics accessibility, performance impact and data loss rate gets its configurable weighting factor $W_\alpha, W_\epsilon, W_\phi$ resulting in the overall availability metric defined as $\Theta = \alpha * W_\alpha + \epsilon * W_\epsilon + \phi * W_\phi$

5 Availability Evaluation

In this section we present an extensible methodology to evaluate the availability capabilities of DDBMSs based on the defined input parameters and output metrics (*cf.* Section 4). Therefore we define an adaptable evaluation process, which emulates the previous failure levels and enacts the respective recovery actions. This process enables the monitoring of the defined availability metrics in order to analyse the high availability efficiency of the evaluated DDBMS. First, we introduce the evaluation process, defining the required evaluation states and transitions. Second, we present an algorithm to inject cloud resource failures on different levels, based on a predefined failure specification.

5.1 Evaluation Process

Emulating cloud resource failures and monitoring the defined availability metrics, requires the definition of a thorough and adaptable evaluation process, from the DDBMS deployment in the cloud over the simulation of cloud resource failures to the recovering of the DDBMS. A fine-grained evaluation process is depicted in Figure 2, where the monitoring periods of the availability metrics are presented in the white box, the evaluation process state in the yellow box and the framework components in the blue box. In the following we introduce the evaluation process states and the monitoring periods, while the framework components are described in Section 6.

The depicted evaluation process in Figure 2 illustrates an exemplary evaluation, which runs through all defined states exactly once by executing the respective transitions (*cf.* Table 4 and Table 5). Yet, it is also possible to leave out dedicated transitions such as *recovery action*. The Gibbon framework executes each evaluation process and it also supports the combination of multiple evaluation processes into an *evaluation scenario*.

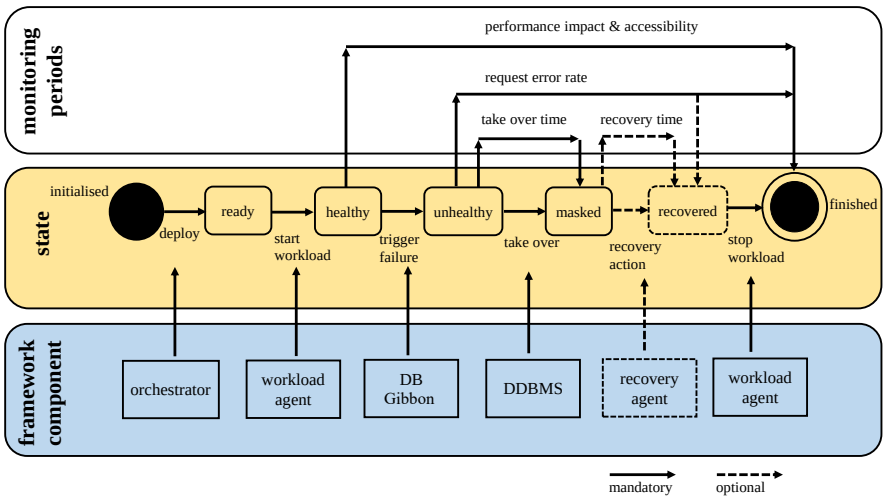

Fig. 2. Evaluation Process

Throughout each evaluation process, the defined availability metrics (*cf.* Section 4) need to be monitored. Yet, the monitoring period for each availability metric depends on the evaluation process state as depicted in Figure 2. The *performance impact* and *accessibility* is monitored from the *healthy* to the finished state to analyse the performance development over the intermediate states and compare it with the performance at the beginning. The *request error rate* is monitored during the *unhealthy* to the *recovered* state to analyse the development of the failed request rate. The *take over time* is monitored in the transition from the *unhealthy* to the *masked* state while the *recovery time* is monitored in the transition from the *masked* to the *recovered* state.

5.2 DB Gibbon Algorithm

In order to emulate failing cloud resources on the different levels, we build upon the concepts of Netflix's Simian Army [26] and adapt these concepts for DDBMS in the cloud. Therefore, we define an algorithm, which is able to enact the introduced cloud failure levels (cf. 3.2) for DDBMSs. Following the Simian Army [26] concepts, we name our algorithm the *DB Gibbon*. The algorithm is depicted in Listing 1.1 and requires as input parameters a list of `failures` <`List`<`failure` >> and the `dbDeployment`. The `failures` list contains n `failure` objects with the attributes `failureLevel`, indicating the cloud resource failure level (*cf.* Table 1), `failureQuantity` specifying the number of failures to be enacted by the DB Gibbon and `nodeType` specifying the failing nodes type. Possible `nodeTypes` are <`ANY, DATA, MANAGEMENT, QUERY`> The `dbDeployment` parameter contains the information of the deployed DDBMS topology, *i.e.* the mapping of each node to the cloud resource. Based on these parameters the DB Gibbon

Table 4. Evaluation states

State	Description
Initialised	A new evaluation process is triggered
Ready	all nodes of the DDBMS deployed in the cloud and the configuration is finished, *i.e.* the DDBMS is operational
Healthy	all nodes of DDBMS are serving requests
Unhealthy	n nodes of the DDBMS are not operational due to cloud resource failures, the DDBMS has not yet started take over actions.
Masked	the DDBMS initiated automatically the take over of n replicas to reestablish the availability of all data records. The DDBMS is operational again but with $-n$ nodes
Recovered	the number of nodes complies again with the initial number of nodes, all nodes of DDBMS are serving requests
Finished	The evaluation process is finished

Table 5. Evaluation transitions

Transition	Description
Deploy	The DDBMS is being deployed and configured, *i.e.* dedicated VMs are allocated in specified locations
Start workload	A constant workload is started against the deployed DDBMS
Trigger failure	a specified cloud resource failure is emulated by the DB Gibbon component (*cf.* Section 5.2, provoking the failure of n nodes
Take over	DDBMS recognizes the failure of n nodes and initialises the take over of the remaining replicas by propagating the new locations for the currently unavailable data records.
Recovery action	the DDBMS is restored to its actual number or nodes by adding n new nodes to the DDBMS. In this process the new nodes are integrated in the running DDBMS and the data is redistributed.
Stop workload	The workload is stopped

algorithm enacts the specified failures in the transition to the *unhealthy* state as depicted in Figure 2. After enacting all failures, the algorithm returns the updated deployment, which is used to trigger the *recovery action* by calculating the difference to the initial deployment and deriving the required recovery actions.

Listing 1.1. DB Gibbon Algorithm

```
input: failures<List<failure>>, dbDeployment
output: dbDeployment
begin
 for each failure in failures
  if failure.failureLevel == failureLevel.VM
   def VMs List<VM> ← dbDeployment.getVMsOfNodeType(failure.nodeType)
   for (int i : failure.failureQuantity)
    def failedVM VM ← failRandomVM(VMs)
    dbDeployment ← updateDeployment(dbDeployment, failedVM)
   end

  else if failure.failureLevel == failureLevel.AVAILABILITY_ZONE
   for (int i : failure.failureQuantity)
```

```
    def VMs List<VM> ← dbDeployment
    .availabilityZone(i).getVMsOfNodeType(failure.nodeType)
    def failedVMs List<VM> ← failVMs(VMs)
    dbDeployment ← updateDeployment(dbDeployment, failedVMs)
   end

  else if failure.failureLevel == failureLevel.REGION
   for (int i : failure.failureQuantity)
    def VMs List<VM> ← dbDeployment.region(i)
    .getVMsOfNodeType(failure.nodeType)
    def failedVMs List<VM> ← failVMs(VMs, failureSeverity)
    dbDeployment ← updateDeployment(dbDeployment, failedVMs)
   end

  //only private cloud deployments
  else if failure.failureLevel == failureLevel.PHYSICAL_HOST
   for (int i : failure.failureQuantity)
    def VMs List<VM> ← dbDeployment.physicalHost(i)
    .getVMsOfNodeType(failure.nodeType)
    def failedVMs List<VM> ← failVMs(VMs, failureSeverity)
    dbDeployment ← updateDeployment(dbDeployment, failedVMs)
   end
  else
   return FAIL
 end
 return dbDeployment
end
```

6 Architecture

This section presents the architecture of the novel Gibbon Framework for executing the introduced evaluation methodology (*cf.* Figure 2). A high-level view on the architecture is depicted in Figure 3, introducing the technical framework components and their interactions between each other. The entry point to the Gibbon framework represents the evaluation API, expecting the evaluation scenario specification. This specification comprises four sub specifications for the respective framework components. In the following each framework component is explained with respect to the required specification and its technical details.

Orchestrator Orchestrator receives the *deployment specification*, which comprises the description required cloud resources and the actual DDBMS with its configuration. Hence, the orchestrator interacts with the cloud provider APIs to provision the cloud resources and to orchestrate the DDBMS on these resources. As carved out in our previous work, the usage of advanced Cloud Orchestration Tools (COTs) over basic DevOps tools is preferable as COTs abstract cloud provider APIs and provide monitoring and adaptation capabilities during runtime [4]. The monitoring capabilities comprise general system metrics as well as customisable application specific metrics. Hence, the monitoring capabilities of COTs can be exploited to measure metrics such as the DDBMS nodes state or ongoing maintenance operations, which are required to express the availability metric (*cf.* Section 4). COTs offer run-time adaptations such as deleting or suspending cloud resources or DDBMS nodes, adding new cloud resources and DDBMS nodes or the execution of additional applications on the existing

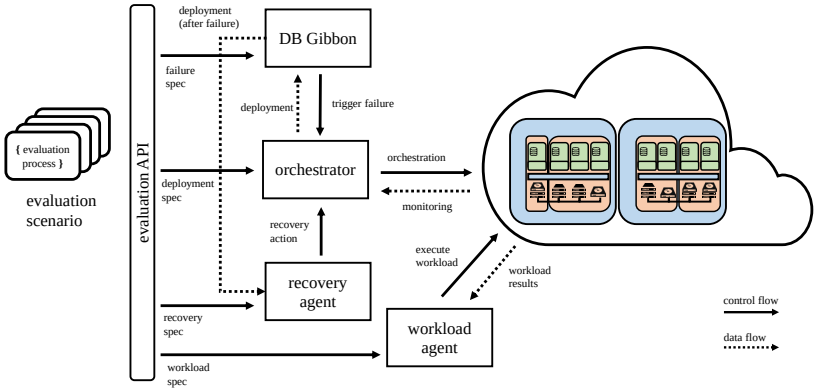

Fig. 3. Gibbon Evaluation Framework Architecture

DDBMS nodes. These capabilities are used by the DB Gibbon and recovery agent components. The Gibbon framework builds upon the Cloudiator COT[2] [10], which supports the main public IaaS providers (Amazon EC2[3], Google Compute[4], Microsoft Azure[5]) as well as private clouds built upon OpenStack[6] and provides advanced cross-cloud monitoring and adaptation capabilities [12].

DB Gibbon This component is responsible to emulate cloud resources failures based on the provided *failure specification*. It queries the COT for the current DDBMS deployment information, *i.e.* the mapping of nodes to cloud resources and their location. Based on the deployment information and the failure specification, it executes the DB Gibbon algorithm (*cf.* Section 5.2). In the execution phase of the algorithm the DB Gibbon interacts with the COT to enact the actual failures on cloud resource level, *e.g.* deleting all VMs of the DDBMS which rely in availability zone A.

Recovery Agent The component receives the *recovery specification*, which comprises the defined recovery actions (*cf.* Section 3.2). It collects the DDBMS deployment state from the the DB Gibbon after its execution and map the recovery actions to the failed resources. To enact the actual recovery actions, the recovery agent interacts with the COT, which executes the cloud provider API calls and orchestrates the DDBMS nodes.

Workload Agent It keeps the DDBMS under constant workload during the evaluation process to measure the performance development during the different evaluation states by receiving a *workload specification*, describing the targeted operation types, the data set size and the number of operations. Further, the workload agent is responsible to measure the performance metrics and store them

[2] `http://cloudiator.org/`
[3] `https://aws.amazon.com/ec2/`
[4] `https://cloud.google.com/compute/`
[5] `https://azure.microsoft.com`
[6] `https://www.openstack.org/`

for further analysis in the context of the availability metrics. Our framework uses the Yahoo Cloud Serving Benchmark (YCSB) as workload agent [8] as the YCSB supports distributed execution, multiple DDBMSs and easy extensibility.

7 Discussion

In this section, we discuss the effectiveness of the Gibbon framework based on a concrete evaluation scenario for MongoDB[7]. First, we describe the applied deployment, failure, recovery, and workload specifications and second, we discuss early evaluation results of the accessibility, take over and recovery time.

7.1 Evaluation Scenario: MongoDB

We select MongoDB as the most prevalent document-oriented data store[8] as the preliminary DDBMS to evaluate. MongoDB is classified as CP in the context of the CAP theorem but still provides mechanisms to enable high availability [11], such as automatic take over in case of node failures. MongoDB's architecture is built upon three different services: (1) `mongos` act as query router, providing an interface between clients and a sharded cluster, (2) `configs` store the metadata of a sharded cluster, (3) `shards` persist the data. A group of shards build a replica set with one `primary` handling all requests and n `secondaries`, synchronizing the primary data and taking over in case the `primary` becomes unavailable.

The applied evaluation scenario builds upon a single evaluation process as depicted in Figure 2. Yet, in this preliminary evaluation process we do not apply a constant workload during the evaluation but use three different data set sizes which are inserted in MongoDB during the deployment of MongoDB. Hence, we only measure the metrics *take over time* and *recovery time*. Accessibility is measured by periodic connection attempts by a MongoDB client.

The *deployment specification* comprises one `mongos` and three `shards` nodes, building a MongoDB replica set with one `primary` and two `secondaries` with full-replication and no sharding. For the sake of simplicity we did not deploy a production-ready setup with multiple `mongos` and `config` nodes. All nodes are provisioned on a private cloud based on OpenStack version Kilo with full and isolated access to all physical and virtual resources. All nodes are provisioned within the region *Ulm* and the availability zone *University*. Each node runs on a VM with 2 vCPUs, 4GB RAM, 40GB disk and Ubuntu 14.04.

As *failure specification* we applied one `failure` object with the attributes `failureLevel=VM, failureQunatity=1 nodeType=data` to the DB Gibbon.

The *recovery specification* defines to add of a new data node (*i.e.* secondary in MongoDB), as soon as MongoDB reaches the *masked* state after a node failure. MongoDB is configured to elect a new primary if the recent primary node failed.

As stated above, we do not apply a constant workload during the evaluation, the *workload specification* only defines the data set by number of `records=100K, 400K, 800K` and `record size=10KB`.

[7] `https://www.mongodb.com/`

[8] `http://db-engines.com/en/ranking_trend`

7.2 Evaluation Results

The preliminary evaluation results reveals two insights: the behaviour in case of node failures and in the case of adding a new replica to the system.

In case of a node failure, the behaviour depends on the failed node type. If a secondary fails, connected clients will loose their read-only connections and have to reconnect to another secondary or to the primary node. In this case we can assume, that at least the primary node is still *accessible*, so clients can reconnect immediately. If the primary fails, clients will loose their read/write connections, but may connect immediately to a secondary node for read requests. Remaining secondaries will recognize that the primary fails and will elect the new primary after a configurable timeout. During this timeout and election phase, no clients can issue write requests, the DDBMS is hence only *accessible* for read requests and not for write requests. Per default, the primary failover timeout is configured to ten seconds. In repeated experiments an average duration for election and primary take over of five seconds ($\pm$ one second) is measured. Hence, the overall *take over time* is 15 seconds.

Whenever a node failure happens, the evaluation scenario adds a new replica after the remaining nodes elected a primary. The new secondary node has to synchronize the stored data from other nodes, for consistency reasons from the primary node. The replication time depends on the size of stored data. For 100k, 400k, and 800k items the median for replication time with 30 runs takes 31s, 300s, and 553s with a standard deviation of 7s, 15s, and 25s. The *recovery time* hence depends on the amount of data to replicate, plus a fixed amount of time it takes to allocate new resources on the Cloud. The DDBMS is *accessible* throughout the replication, yet with reduced resources due to the ongoing synchronisation.

8 Related Work

The view on availability in distributed systems evolved over the last two decades. The CAP theorem published in 2000, states that any networked shared-data system can only have two of the three properties consistency, availability and partition tolerance [6]. A revisited view on the CAP theorem is presented in 2012 [5], reflecting how emerging distributed systems such as DDBMSs adapted their consistency, availability and partition tolerance properties as the decision for two out of the three properties is not a binary decision [5].

The classification of DDBMSs according to their CAP properties in CA or CP DDBMSs is a widely discussed topic in database research. A first overview and analysis of emerging DDBMSs is provided by [7], analysing various DDBMSs with respect to their availability capabilities in the context of the CAP theorem.

DDBMSs mechanism to provide high availability are discussed by [15], breaking down the technical replication strategies from master-slave replication to masterless, asynchronous replication of 19 DDBMSs. Yet, the high availability mechanisms are only discussed on a theoretical level and no evaluation of their efficiency is proposed. Further, cloud resources are introduced as the preferable

resources to run DDBMSs but the different failure levels and their implication to the availability of the DDBMS are not considered.

A similar approach is followed by [23], adding a dedicated classification of common DDBMSs with respect to their CAP properties, *i.e.* AP or CP. Further, the usage of cloud resources is discussed with respect to virtualisation and data replication across multiple regions. Yet, the focus relies on enabling consistency guarantees of wide-area DDBMSs while side-effects by using cloud resources that effect as well as ensure availability in DDBMS are not considered in detail.

An availability- and reliability-centric classification of DDBMSs is presented by [11]. Hereby, the challenges to provide non-functional requirements such as replication, consistency, conflict management, and partitioning are broken down in a fine grained classification schema and a set of 11 DDBMS are analysed. Two availability affecting issues and solutions are presented, overloading and node failures: *i)* Is a DDBMS not available due to overloading, the DDBMS needs to be scaled out. *ii)* Replicas need to be in place to overcome database node failures. Yet, the proposed solutions are on an architectural level and the actual capabilities of DDBMSs are not evaluated in real-world scenarios.

While theoretical classifications of DDBMSs provide a valuable starting point for a first selection of DDBMSs, the final selection still remains challenging due to the heterogeneous DDBMSs landscape. Hence, database evaluation frameworks provide additional insights in DDBMS capabilities by evaluation dedicated evaluation tiers based on different workload domains. While available evaluation frameworks such as YCSB [8] or YCSB++ [20], focus on the evaluation performance, scalability, elasticity and consistency, the availability tier is not yet considered by these frameworks [24]. First approaches in availability evaluation are based on the YCSB and focus the on decreased availability due to an overloaded database [25] [18]. While an evaluation framework focusing on the availability of cloud-hosted DDBMSs is not yet available, an approach to enact synthetic failures on cloud resources is described by [26] and implemented at Netflix. Yet, this approach only describes the failure scenarios in the cloud and does not propose evaluation metrics or an evaluation methodology for cloud applications in general and DDBMS in particular. A first approach towards the availability metrics is presented by [2], yet the focus relies on the resilience of DBMSs, while DDBMSs and their availability mechanism are not considered. Existing failure-injection tools such as the DICE fault injection tool[9] or jepsen[10] either inject failures on node or DBMS level but do not support the injection of resource failures in different granularity.

9 Conclusion and Future Work

In the last decade the database management system (DBMS) landscape grew fast, resulting in a very heterogeneous DBMSs landscape, especially when it

[9] `https://github.com/dice-project/DICE-Fault-Injection-Tool`
[10] `https://github.com/jepsen-io/jepsen`

comes to distributed database management systems (DDBMSs). As cloud computing is the preferable way to run distributed applications, cloud computing seems to be the choice to run DDBMSs. Yet, the probability of failures increases with the number of distributed entities and cloud computing adds another layer of possible failures. Hence, DDBMSs apply data replication across multiple DDBMS nodes to provide high availability in case of node failures. Yet, providing high availability comes with limitations with respect to consistency or partition tolerance as stated by the CAP theorem. As these limitations are not binary, a vast number of high availability implementations in DDBMSs exist. This makes the selection of a DDBMS to run in the cloud a complex task, especially as supportive availability evaluation frameworks are missing.

Therefore, we present Gibbon, a novel availability evaluation framework for DDBMSs to increase the trust in running DDBMSs in the cloud. We describe levels of cloud resource failures, existing DDBMS concepts to provide high availability and distill a set of five quantifiable availability metrics. Further, we derive the DDBMSs specific technical details, affecting the availability evaluation processes. Building upon these findings, we introduce the concept of extensible evaluation scenarios, comprising n evaluation processes. Further, we present the DB Gibbon, which emulates cloud resource failures on different levels.

The Gibbon framework executes the defined evaluation scenarios for generic DDBMSs and cloud infrastructures. Its architecture comprises an orchestrator to deploy the DDBMS in the cloud, a workload agent, a recovery agent and the DB Gibbon to inject cloud resources failures. As preliminary evaluation, we evaluate the take over and recovery time of MonogDB in a private cloud, by injecting failures on the virtual machine level.

Future work will comprise an in-depth evaluation of multiple well-adopted DDBMSs[8] based on the Gibbon framework. Further, the definition of a minimal evaluation scenario to derive a significant availability rating, is in progress. In this context, the statistical calculations of the overall availability rating index will be refined. Finally, the portability of Gibbon evaluate the availability of generic applications running in the cloud will be evaluated.

Acknowledgements The research leading to these results has received funding from the EC's Framework Programme HORIZON 2020 under grant agreement numbers 644690 (CloudSocket) and 731664 (MELODIC). We also thank Daimler TSS for the encouraging and fruitful discussions on the topic.

References

1. Abadi, D., Agrawal, R., Ailamaki, A., Balazinska, M., Bernstein, P.A., Carey, M.J., Chaudhuri, S., Chaudhuri, S., Dean, J., Doan, A., et al.: The beckman report on database research. Communications of the ACM (2016)
2. Almeida, R., Neto, A.A., Madeira, H.: Resilience benchmarking of transactional systems: Experimental study of alternative metrics. In: PRDC (2017)
3. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. TDSC (2004)

4. Baur, D., Seybold, D., Griesinger, F., Tsitsipas, A., Hauser, C.B., Domaschka, J.: Cloud orchestration features: Are tools fit for purpose? In: UCC (2015)
5. Brewer, E.: Cap twelve years later: How the" rules" have changed. Computer (2012)
6. Brewer, E.A.: Towards robust distributed systems. In: PODC (2000)
7. Cattell, R.: Scalable sql and nosql data stores. Acm Sigmod Record (2011)
8. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: SoCC (2010)
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. ACM SIGOPS operating systems review (2007)
10. Domaschka, J., Baur, D., Seybold, D., Griesinger, F.: Cloudiator: a cross-cloud, multi-tenant deployment and runtime engine. In: SummerSOC (2015)
11. Domaschka, J., Hauser, C.B., Erb, B.: Reliability and availability properties of distributed database systems. In: EDOC (2014)
12. Domaschka, J., Seybold, D., Griesinger, F., Baur, D.: Axe: A novel approach for generic, flexible, and comprehensive monitoring and adaptation of cross-cloud applications. In: ESOCC (2015)
13. Ford, D., Labelle, F., Popovici, F.I., Stokely, M., Truong, V.A., Barroso, L., Grimes, C., Quinlan, S.: Availability in globally distributed storage systems. In: OSDI (2010)
14. Geraci, A., Katki, F., McMonegal, L., Meyer, B., Lane, J., Wilson, P., Radatz, J., Yee, M., Porteous, H., Springsteel, F.: IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries: 610. IEEE Press (1991)
15. Grolinger, K., Higashino, W.A., Tiwari, A., Capretz, M.A.: Data management in cloud environments: Nosql and newsql data stores. JoCCASA (2013)
16. Gunawi, H.S., Hao, M., Suminto, R.O., Laksono, A., Satria, A.D., Adityatama, J., Eliazar, K.J.: Why does the cloud stop computing? lessons from hundreds of service outages. In: SoCC (2016)
17. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. CSUR (1983)
18. Konstantinou, I., Angelou, E., Boumpouka, C., Tsoumakos, D., Koziris, N.: On the elasticity of nosql databases over cloud management platforms. In: CIKM (2011)
19. Mell, P., Grance, T.: The nist definition of cloud computing. Tech. rep., National Institute of Standards & Technology (2011)
20. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In: SoCC (2011)
21. Pritchett, D.: Base: An acid alternative. Queue (2008)
22. Sadalage, P.J., Fowler, M.: NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education (2012)
23. Sakr, S.: Cloud-hosted databases: technologies, challenges and opportunities. Cluster Computing (2014)
24. Seybold, D., Domaschka, J.: Is distributed database evaluation cloud-ready? In: ADBIS (2017)
25. Seybold, D., Wagner, N., Erb, B., Domaschka, J.: Is elasticity of scalable databases a myth? In: IEEE Big Data (2016)
26. Tseitlin, A.: The antifragile organization. Communications of the ACM (2013)
27. Zhong, M., Shen, K., Seiferas, J.: Replication degree customization for high availability. In: EuroSys (2008)