electronics

Article

A Method of Transparent Graceful Failover in Low Latency Stateful Microservices

Kęstutis Pakrijauskas * and Dalius Mažeika

Department of Information Systems, Faculty of Fundamental Sciences, Vilnius Gediminas Technical University, Sauletekio Avn., 11, 10223 Vilnius, Lithuania
* Correspondence: kestutis.pakrijauskas@vilniustech.lt

Abstract: Microservice architecture is a preferred way to build applications. Being flexible and loosely coupled, it allows to deploy code at a high pace. State, or, in other words, data is not only a commodity but crucial to any business. The high availability and accessibility of data enables companies to remain competitive. However, maintaining low latency stateful microservices, for example, performing updates, is difficult compared to stateless microservices. Making changes to a stateful microservice requires a graceful failover, which has an impact on the availability budget. The method of graceful failover is proposed to improve availability of a low latency stateful microservice when performing maintenance. By observing database connection activity and forcefully terminating idle client connections, the method allows to redirect database requests from one node to another with negligible impact on the client. Thus, the proposed method allows to keep the precious availability budget untouched while performing maintenance operations on low latency stateful microservices. A set of experiments was performed to evaluate stateful microservice availability during failover and to validate the method. The results have shown that near-zero downtime was achieved during a graceful failover.

Keywords: stateful microservice; availability; database; cluster; Kubernetes; failover; connection pool

Citation: Pakrijauskas, K.; Mažeika, D. A Method of Transparent Graceful Failover in Low Latency Stateful Microservices. *Electronics* **2022**, *11*, 3936. https://doi.org/10.3390/electronics11233936

Academic Editor: Filipe Araujo

Received: 26 October 2022
Accepted: 24 November 2022
Published: 28 November 2022

1. Introduction

Microservices, an implementation variant of Service-Oriented Architecture, are single-purposed, loosely coupled, autonomous services that constitute an application. They are designed to be resilient and independent from other services. Components of microservices are built to handle failures rather than prevent them. Techniques such as graceful degradation, decoupling, retries, caching, and redundancy allow to improve reliability of microservices [1]. According to ISO 25010 availability, a subcharacteristic of reliability, is "*the degree to which a system, product or component is operational and accessible when required for use*" [2]. Availability of a microservice is affected by maintenance time if it is treated as a repairable item [3]. To minimize the impact on availability, maintenance of stateful microservices requires graceful failover between components that provide high availability. Thus, high availability is important to ensure elasticity and flexibility of stateful microservices.

Relational database management systems remain the cornerstone of many software applications, whether in a Monolith on in a Microservice-based application, even with the emergence of data storage management systems such as non-relational databases [4] or distributed file systems [5]. Adoption of containers, OS-level virtualization–grows in the industry as they were built with microservices in mind. By abstracting the OS level, containers allow one to simplify the software deployment process. The performance of container-based software is better compared to Virtual Machine (VM) based software [6].

Orchestration systems, such as Kubernetes or Docker Swarm, further reduce the effort required to manage containers. Container orchestration systems introduced persistent, non-ephemeral storage to support stateful services since many applications need a database. It is suggested that transient data is stored on containers running databases [7] as containers provide more flexibility with regards to scaling and high availability [8,9].

Shorter downtime allows an organization to remain competitive in the market or, for a public organization, to provide better services. Reduced manpower requirements result in smaller teams that work faster than larger teams [10]. The saved-up availability budget can then be used for other purposes such as upgrades, security patches, and other improvements [11].

Many studies focus on increasing availability or reliability of database systems. However, modern database high availability solutions, including the aforementioned ones, focus on reducing the failover time. Either by increasing consistency of replicas by means of sharing storage [12] sharing memory [13], novel persistence framework for container based deployments [14], novel replication algorithm [15],. or by improving connection pooling towards persistent systems [16,17]. We find that research on availability increase during managed failover operations is lacking.

Reusing connections allows to reduce the latency for database requests. Avoiding the time-consuming cycle of connection establishment allows to increase throughput and reduce latency towards a database [16]. Reused connections are always open towards a database, thus it becomes a challenge to tell if they are active or not.

In this research we are focusing on improving database availability during managed graceful failover–switchover–operation. Maintenance is an important aspect in the life cycle of a microservice. Performing preventive maintenance on a stateful microservice may have an impact on its availability. Reused connections allows to reduce the latency towards a database. However, it brings additional complexity to the application and its maintenance. We propose a method that allows to mitigate the effect of managed database failover on the availability of a low latency application. In this paper we have shown that the proposed method allows to achieve near-zero downtime with minimal impact on a low latency application. Managed failover becomes nearly transparent to database clients using connection pooling. Thus, with employment of the proposed method, maintenance of low latency stateful microservices can be performed with minimal impact on availability.

The rest of the paper is organized as follows: Section 2 presents a literature review on the topics of stateful microservice availability and reliability, and the principles the proposed method is based upon; Section 3 describes the method we propose; Section 4 outlines the investigated architecture; Section 5 contains results of the experiment; conclusions are presented in Section 6.

2. Background and Related Work

Applications use database systems to store and maintain their data. The system involves large numbers of objects, such as storage mechanisms, query and programming languages, drivers, logic, and mapping between an application and its database(s). One important object is the connection. Establishing a connection to a database requires numerous resources and actions: back-and-forth authentication process between client and server, authorization, memory allocation, etc. The object-relational persistence framework suggests that a connection should serve one transaction and be terminated afterwards. However, since connection creation is time-consuming, connection pooling supports short-lived connections. A connection pool allows to reuse already established connections: the connection is returned to the pool after being used [16]. Reusing connections from a pool allows to reduce request latency as there is no need to perform the time-consuming process of connection establishment.

A database server process either creates a new thread or assigns an idle thread for each client connecting to it. The server-process, or rather the threads it allocates, execute requests coming from connected clients. A thread then executes a query sent by the connection allocated to it. Subsequently, the thread either closes the socket or, in the case of reused connections, keeps the socket open and waits for additional incoming queries [18].

As the socket waits for more incoming queries, the connection remains open. Thus, without explicitly checking its status on either the database or the client side, it is unclear if a query is being executed or not. Closing a socket without knowing if it is active or not may result in a failed query.

An additional challenge is routing requests to different nodes of a distributed database cluster. There are two high-level approaches to routing requests [19]:

- Allowing clients to send requests to any node of a cluster. The node then either processes the request or forwards the request to another appropriate node.
- Client sending requests via routing level. At the routing level a decision is made on which node the client should connect to.

Regardless of which routing approach is taken, the challenge of making the decision of where to route the request remains. This challenge is commonly known as service discovery [19].

Request routing and load balancing are important parts of high availability assurance for database systems. Marinho et al. has presented a LABAREDA service for predictive and elastic load balancing. Its purpose is to predict Service Level Agreement (SLA) violations using prediction models [20]. The service uses Autoregressive Integrated Moving Average and Exponential Moving Average prediction models to predict and balance workloads against a replicated database. Although the proposed service has sufficient capacity to predict SLA violations and take necessary action to keep performance metrics within SLO, its impact on database availability during a failover is limited.

A replication-based middleware system, Hihooi, presented by Georgiou et al., includes a novel routing algorithm [21]. The system itself is designed to enable high scalability, consistency, and elasticity for transactional databases. Hihooi is a middleware system which is positioned between database engines and applications. It intercepts requests towards a database and routing algorithms direct them to the most consistent replica(s). The algorithm, based on inspection of request transactions, allows to avoid delays of read requests by directing them to replicas with consistent data. The method proposed in this research is based on inspection of transaction contents as well. However, the inspection is done on the database level rather than on Transaction manager level as is the case with Hihooi.

The availability of the databases is ensured by distributing the data over multiple nodes. Consistency between distributed nodes is ensured by replicating the data. There are two replication strategies [22]:

Single-Primary replication. This strategy implies that all writes are directed to a single node in a distributed database and then replicated to the remaining nodes. This kind of setup reduces the risk of consistency conflicts. However, in case of failure of the primary node, it takes time to elect a new primary to take writes. Single-Primary failover is not a trivial operation. Additional actions, such as reconfiguration of distributed database nodes to change the source of replication, pause application traffic, reconfiguration of application clients, etc. have to be taken to failover a replicated database with one Primary node [11].

Multi-Primary replication. This strategy implies that more than one node in a distributed database can take writes and distribute them across the remaining nodes. In case of failure of one of the nodes any other node in the multi-Primary distributed database can take over. Additional load balancing solutions or proxies can be used to further minimize downtime and mask incompleteness of a multi-Primary distributed database in the event of a disaster. An important drawback is the need for conflict resolution. Writes coming from multiple sources increase the risk of conflicts which have to be resolved. On the other hand, writes can be directed to one of the nodes of the multi-Primary distributed database, thus creating a pseudo-Single-Primary setup [11]. As Multi-Primary replication allows writes to be taken by any node in the cluster, transitioning a connection from one node to another is an option. Connections can be gradually shifted towards other database nodes while maintaining constant throughput. However, in that case, conflict resolution may have an impact of response time due to multiple nodes in the cluster processing write requests.

Distributed database node outage can be unexpected, such as caused by failure or expected, for example when taken down for maintenance. In this research, we consider only expected outage.

The mechanism of taking down a node in a distributed database depends on the replication strategy. In single-Primary setups any node except for the Primary node can be taken out of the cluster at any time. Upon returning online to the cluster it has to catch up on the changes happened since last known transaction.

Taking down the Primary node in Single-Primary setup is not a trivial matter. The Primary node has to step down, and a new Primary has to be elected before it can be taken offline. Stepping down requires connections to be drained from the stepping down Primary node. Clients them have to be reconfigured to use the newly elected Primary node [19].

In Multi-Primary setup connections have to be drained from node to be taken out of the cluster. Applications should not be able to connect to the node that is in the process of being taken out. Adding a node back to the cluster implies that applications can connect to the node again. Various techniques are used to allow or disallow applications to connect to a node: changing the connection setting in an application, changing the Domain Name System (DNS) records, and/or changing the node pool in a proxy or load balancer layer [11].

In both cases the cluster becomes out of sync–data have to be replicated to nodes returning or being added to the distributed database cluster. Out-of-sync nodes have to catch up for database cluster to become consistent again.

Taking a distributed database node down for maintenance has an impact on the database cluster. The latency between the replicated data on distributed database nodes is called Replication lag [11]. In addition to describing the degree of inconsistency of a database cluster, replication lag may also result in a negative user experience manifested in inconsistent reads and/or writes [19,23]. Thus taking down a node in a cluster, built for reliability, may have impact on availability. Time to return back to the cluster is important to reduce the impact of replication lag.

In certain cases the data must be fully copied onto a node returning to the distributed database cluster. For example due to replication lag exceeding the set Service Level Objective (SLO) [11]. Copying data from one to another may take a significant amount of time on larger databases. There is a risk that the copied data will not be consistent if the database is being actively written [19]. In addition, copying data can result in degraded performance of a distributed database cluster, which, in turn, is a risk to SLO.

Deployment of certain database changes, such as memory setting or minor software version changes, requires a restart. Replication allows to have the service to stay online and process requests. However, pooled connections pose a challenge when they have to be drained from a database node There are two approaches on what to do with the connections already established towards a database node: forcefully terminate or drain them by reconfiguring the application [11]. In the first case, connections are terminated, the operator has to accept the impact to availability. In the second case, the operator has to make changes to the client application, for example, change the data source or reload the configuration. Client application reconfiguration is a sound approach to drain connections from a database node as impact to availability is avoided. Although it involves more components and increases coupling of microservices.

Causes of failing requests can be of two types: transient and persistent. Transient failures are caused by non-permanent conditions, for example, interim loss of connection, reconfiguration, temporary resource exhaustion, etc. Persistent failures are caused by conditions that cannot be resolved without human intervention [24].

Microsoft recommends configuring retries to handle different types of exceptions. Exception handing based on the cause allows to provide more reliable service by mitigating transient failures. If properly configured retries allow self-correcting faults to have less significant impact on the availability of an application [24,25].

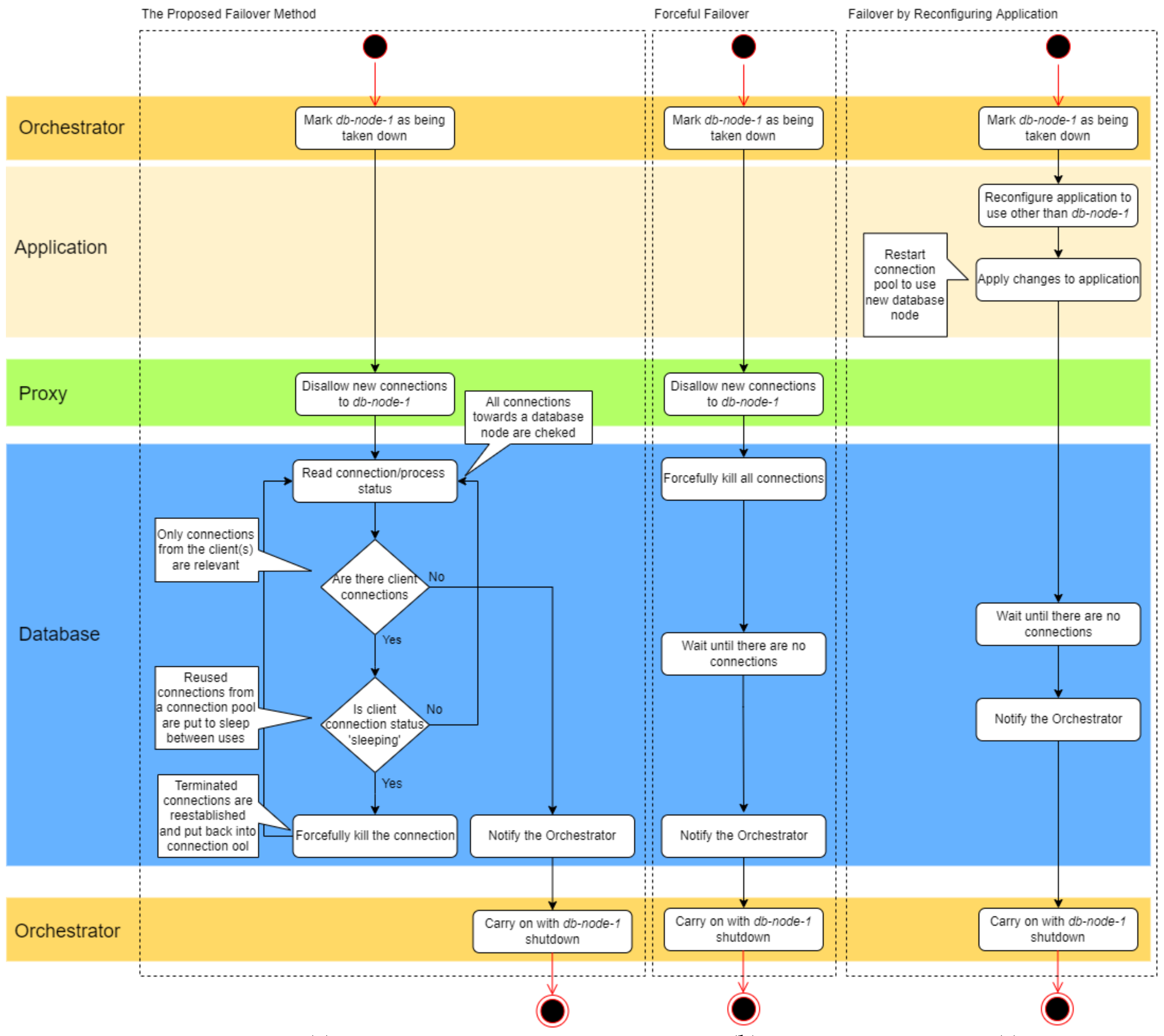
There are three exception handling strategies [26]:

- Cancel the failing request. A failed request is canceled in case it is caused by a persistent condition.
- Retry immediately. A failed request was caused by a transient condition that allows it to be immediately repeated. The failure condition, such as network packet corruption, is unlikely to be encountered again, and thus it is safe to retry the request immediately.
- Retry after delay. A failed request was caused by a transient condition that cleared after a certain amount of time. The failure condition, for example, a network connectivity issue or request throttling, needs time to be corrected. Within that time, it prevents a request from being executed successfully. Therefore, retries are issued after a delay.

The fact that retries are an important factor of error handling was demonstrated by Dai [27]. The Retryer by Dai [27] verifies if requests should be retried against the database. It introduces a retry logic on the client side which verifies if a failed request needs to be attempted again. It demonstrates that a retry mechanism, although a complex one, can improve database recovery after transient errors.

3. The Method for Transparent Failover

In this section we present the proposed method and how it functions towards improving availability of stateful microservices during failover in low latency applications. Given that a connection pool is used by a client when a node of a database cluster has to be shut down, its orchestrator, whether a person or an automated script, has to drain the connections either from the application side or on the database side. In the first case, the orchestrator needs to have access to the application. In the latter case, there is a risk of terminating actively used connections. The idea behind the proposed method is to forcefully close only idle (sleeping) client connections to a database node. The flow diagram below (Figure 1) compares the graceful failover method we propose (Figure 1a) with forceful failover method (Figure 1b), and failover by reconfiguring application (Figure 1c).



The Proposed Failover Method
Forceful Failover
Failover by Reconfiguring Application
Orchestrator
Application
Proxy
Database
Orchestrator
Mark db-node-1 as being taken down
Reconfigure application to use other than db-node-1
Restart connection pool to use new database node
Apply changes to application
Disallow new connections to db-node-1
All connections towards a database node are cheked
Read connection/process status
Forcefully kill all connections
Only connections from the client(s) are relevant
Are there client connections
No
Yes
Wait until there are no connections
Reused connections from a connection pool are put to sleep between uses
Is client connection status 'sleeping'
Notify the Orchestrator
Terminated connections are reestablished and put back into connection ool
Forcefully kill the connection
Carry on with db-node-1 shutdown

(**a**) (**b**) (**c**)

Figure 1. Flow diagrams of proposed graceful failover method (**a**) forceful failover (**b**), and failover requiring changes on application side (**c**).

The first step in the proposed method (Figure 1a) is taken by the orchestrator. It marks the node set for shutdown as being taken down. Next the database proxy, or a load balancer, takes the node out from the database cluster: new connections towards the database node are disallowed. Yet the existing connections towards the database node are kept alive, whereas all incoming connections are redirected towards other nodes in the cluster.

As the database proxy may not be aware of connection content the forceful closure of connections is taken care of by the database node. The status of all client connections is read on the database node. The database node loops thru the statuses of the established client connections and terminates if status shows idle until all are closed. Since terminated connections were not actively used by a client application–they were in a free connection pool–its termination does not make any impact on the client application. Only an attempt to reconnect is made. Once all connections are drained from the database node, the orchestrator proceeds with shutting it down. A pooled connection is reestablished after being closed by the database. The application can continue querying the database since new connections are routed by database proxy to other nodes in a database cluster but the one that is being shut down.

Figure 2 illustrates how application client connections transition from one database node to another. In the given example a pooled connection is transitioned from *db-node-1* to *db-node-2*. Transition events are shown as *e1* through *e6* in Figure 2.

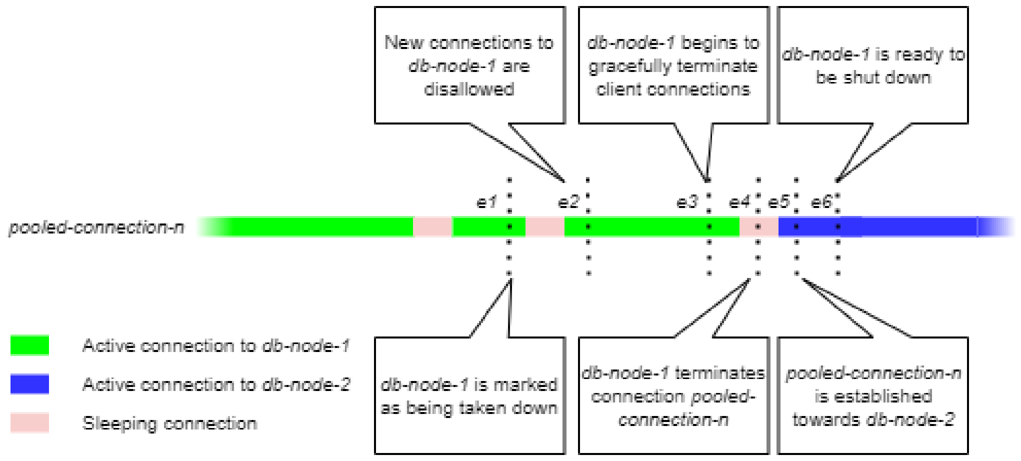

New connections to db-node-1 are disallowed
db-node-1 begins to gracefully terminate client connections
db-node-1 is ready to be shut down
e1
e2
e3
e4
e5
e6
pooled-connection-n
Active connection to db-node-1
Active connection to db-node-2
Sleeping connection
db-node-1 is marked as being taken down
db-node-1 terminates connection pooled-connection-n
pooled-connection-n is established towards db-node-2

Figure 2. Client connection transition.

At the first event *e1*, the *db-node-1* is marked as being shut down. However, no other action is taken against it. At *e2, the* database proxy cordons the *db-node-1* disallowing new connections towards it. The *db-node-1* begins terminating incoming client connections at *e3*. The *pooled-connection-n* connection to enters 'sleeping' state. This state of the connection is then detected by the *db-node-1,* and at *e4* it is terminated. Once *pooled-connection-1* is reestablished, the database proxy directs it to *db-node-2* at *e5*. Once all connections have been transitioned to *db-node-2, db-node-1* is marked as ready to shutdown at *e6*.

4. Implementation of the Method

The method we propose allows to transition low latency database connections from one node to another. However, the distributed database cluster and its clients has to have a certain setup for the aforementioned method to function. As shown in Figure 3, the setup consists of:

- Database cluster setup for multi-Primary replication
- Proxy layer that can direct client requests to a specific node
- A client connection termination mechanism
- A connection pool to enable low latency database connections
- A retry mechanism on the client side to handle terminated connections
- An orchestrator to oversee the failover.

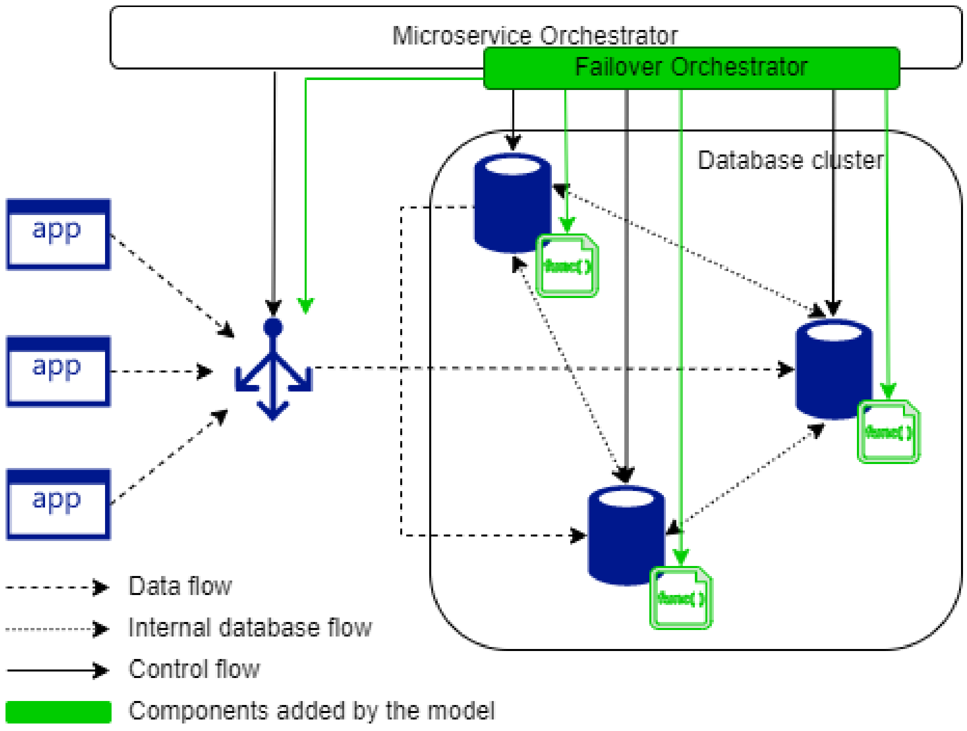


Figure 3. Components of the proposed method.

Multi-Primary replication would allow distributed database cluster to continue serving client requests while a graceful failover is happening.

A proxy layer, whether an appliance or a service, would allow to balance requests between nodes while switchover is performed (Listing 1). Once switchover is started new and re-established client connections are directed towards another node in the cluster but not the one being taken out.

Listing 1. Pseudocode of the connection directing algorithm at proxy layer.

```
directDBConnections
Input: dbNode, connectionAction
begin
    // higher weight increases the node priority
    defaultNodePriority = 10
    if connectionAction == 'disallow'
        // weight valued 0 disallows new connections towards the node
        nodePriority = 0
        call setNodePriority (dbNode, nodePriority)
        returnMsg = 'connectionsDissallowed'
    end
    if connectionAction == 'allow'
        call setNodePriority (dbNode, defaultNodePriority)
        returnMsg = 'connectionsAllowed'
    end
    return returnMsg
end
```

```
Output: returnMsg
```

The client connection termination mechanism has to read the content of a database process to decide on its termination. The algorithm is quite straightforward (Listing 2):

1. Collect a list of active client connections
2. If there are client connections, iterate through the list and check their status:
   - Terminate the connection if status is 'sleeping'
   - Skip to next connection if status is other than 'sleeping'
3. Refresh the list of client connections and check the status again

Listing 2. Pseudocode of the connection termination algorithm.

```
terminateClientConnections
Input: clientConnectionIdentifier
begin
    // create list of client connections
    clientConnectionList ← getClientConnections(clientConnectionIdentifier)
    while LEN(clientConnectionList) > 0 do
        // loop thru client connections
        for clientConnection in clientConnectionList
        clientConnectionState ← getClientConnectionState(clientConnection)
            if clientConnectionState == 'sleeping'
                terminateConnection(clientConnection)
            else
                skipToNext()
            end
        end
        // refresh list of client connections
        clientConnectionList ← getClientConnections(clientConnectionIdentifier)
    end
    returnMsg = 'connectionsDrained'
    return returnMsg
end
Output: returnMsg
```

Connections that handle internal cluster traffic, such as replication or heartbeat, are not affected by the orchestrator. Interfering with these kinds of connections may produce unwanted results. The internal cluster connections are handled by the cluster. Client connections can be identified by different parameters such as host name, target database, or username.

The client is configured to retry failed requests upon loss of connection towards the database.

The failover orchestrator, as the name suggests, oversees the failover of client connections between database nodes. The orchestrator issues commands to database nodes and database proxy during the entire managed failover process (Listing 3). By knowing the status of client connections along with cluster configuration appropriate and timely commands allow to limit the impact of managed failover on availability of a database cluster. It can be part of a microservice orchestrator that manages microservices. For example, the failover orchestrator can be implemented as part of the Kubernetes Operator.

Listing 3. Pseudocode failover orchestrator.

```
failoverOrchestrator
Input: dbNodeList, proxyNode
begin
    for node in dbNodeList
        clientConnectionState ← proxyNode.directDBConnections(node, 'disallow')
            if clientConnectionState == 'connectionsDissallowed'
                terminateClientConnections.state ← node.terminateClientConnections()
            end
            if terminateClientConnections == 'connectionsDrained'
                // call a generic maintenance procedure
                call performDBNodeMaintenance(node)
            end
        clientConnectionState ← proxyNode.directDBConnections(node, 'allow')
            if clientConnectionState == 'connectionsAllowed'
                // start maintenance of other database nodes
                skipToNext()
            end
    end
    return SUCCESS
end
Output: none
```

There are a couple of advantages of the proposed method compared to other techniques for error-free managed failover:

- Support for connection pooling. Connection pooling allows to achieve higher throughput and lower latency compared to reopened connections [16].
- Low risk of double-write or other unwanted result upon request retry [27]. Since the connection is terminated between request execution, there is no need to retry a request;. it is sufficient to re-establish a connection. Even if Retryer by Dai [27] is a sound approach, it is more complex as, unlike the proposed method, it requires complex logic on the client side.

One of the 'traditional' approaches to managed failover is the acceptance of impact on availability. The proposed method allows to perform a managed failover with near-zero loss of availability.

The other approach to managed failover is rerouting requests by changing the connection string on the client side. The proposed method eliminates the need to make changes on the client side in order for switchover to happen without a failing request. This reduces coupling if application consists of many microservices.

5. Evaluation of the Method

We performed an experiment to analyze how the proposed method operates under different conditions and compare it to forceful failover method. We created a prototype environment with all the necessary components: database cluster behind a proxy, failover orchestrator, and a client. We evaluated the two aforementioned failover methods under varying database load and retry delays.

The prototype environment was set up on a three-node Kubernetes cluster in Google Cloud Platform (GCP). The three nodes were identical: E2 cost-optimized instances with 6 GB of memory, 4 CPUs, and 50 GB standard persistent disk (PD). The MySQL Galera cluster was chosen as this database management system has capability of multi-master replication. ProxySQL was used as the database proxy. The investigated architecture is displayed in Figure 4.

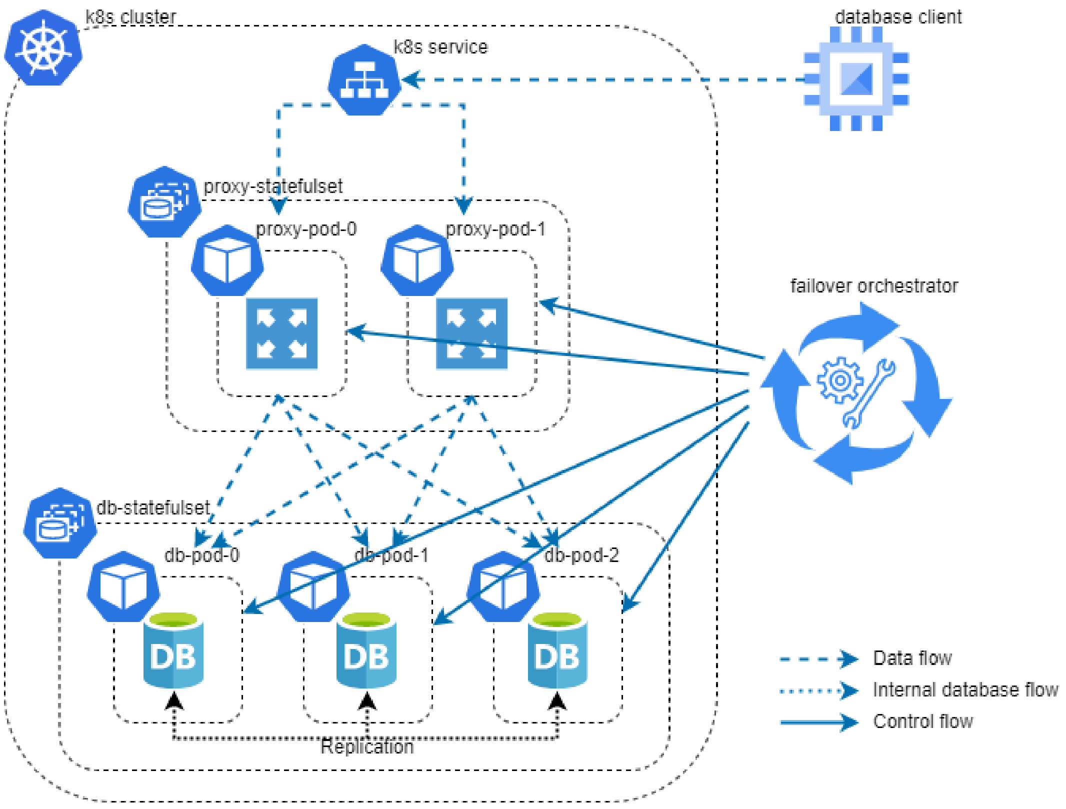


Figure 4. The investigated architecture of the multi-master database cluster.

The database client was set up on a VM in GCP. The VM was an E2 cost-optimized instance with 4 CPUs and 8 GB of RAM, running Ubuntu 18.04. The failover orchestrator was set up on an E2 cost-optimized instance with 1 CPUs and 1 GB of RAM, running Ubuntu 18.04. Software versions and resource allocation for MySQL Galera cluster and ProxySQL nodes are listed in Table 1.

Table 1. Software Versions and Resource Allocation.

Software and Version	Number of Pods	CPU Limit	Memory Limit
MySQL Galera Cluster 8.0.25	3	600 m	1024 MB
ProxySQL 2.0.18	2	300 m	256 MB

Different numbers of simultaneous requests towards the database were issued. It is important to know how load (saturation) impacts the performance and effectiveness of the method.

The proxy layer has capability to assist in graceful failover of underlying database nodes. Thus we used this feature of the proxy layer when performing our experiment. Incoming requests were redirected to the appropriate database nodes. Database node was put into 'OFFLINE_SOFT' state in ProxySQL to disallow connections towards it. The state disallows new connections to the database node, however already established connections are kept intact.

The probe request executes the *probeRequest* stored procedure. The stored procedure inserts data that indicate a successful request. After the failover is complete the count of records per session is compared to the number of requests issued per session. The number of issued requests is the maximum *ordinal_number* of a session. The database objects are listed in Listing 4. We assume that Select statements would not be affected by managed failover. Retry of read-only requests have a limited risk of encountering an exception compared to write request. In addition, a retry of a Select statement has insignificant impact on availability of read-only requests.

Listing 4. Database objects.

```
CREATE TABLE IF NOT EXISTS logtable(
        id INT NOT NULL AUTO_INCREMENT,
        log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        session_id VARCHAR(64),
        ordinal_number INT,
        PRIMARY KEY (id)
);

CREATE PROCEDURE probeRequest (sessionID VARCHAR(64), ordinalNumber INT)
BEGIN
        INSERT INTO logtable (session_id, ordinal_number)
        VALUES (sessionID, ordinalNumber);
END
```

In addition two types of retries were evaluated: an immediate retry and a backed-off retry after 1 s. Immediate retry is the simplest retry pattern. We expect the error causing condition would allow to retry the request immediately when the proposed failover method is used. A backed off retry is recommended in case the error condition is expected to clear after a certain delay [26]. We assume that 1 s is a sufficient amount of time before retrying the request. The other two nodes in the cluster will be ready to accept incoming requests within the 1 s of delayed retry.

Parameters of the experiment:

- Number of parallel requests (saturation): 100, 200, 300, 400.
- Connection termination: graceful termination (the method), forceful termination.
- Exception handling pattern: immediate retry, delayed retry (1 s).

We are evaluating all possible parameter sets. Availability is affected by maintenance time and failure rate (occurrence of failures) [3]. We consider the time needed to move all connections from one node to another as failover duration. As suggested by Hauer et. al., we will use the ratio of failed and successful requests to measure availability [28]. The failure ratio is the percentage of failed requests to the number of parallel requests. Given how connections are directed from one database node to another, we assume that the maximum number of failed requests per experiment execution will likely not exceed the saturation.

$$\text{failure ratio} = \frac{\text{number of failed requests} * 100}{\text{number of parellel requests}} \tag{1}$$

We ran the experiment five times with each parameters set to compare mean failover duration and failure ratio.

6. Results and Discussion

As expected the proposed method had an insignificant impact on availability compared to the forceful failover mechanism. The mean failure rate during forceful failover for different parameter sets ranges from 53 to 94 (Figure 5). It increases along with the number of parallel requests. The proposed method of graceful failover allows to reduce the mean failure rate to near zero: the highest mean failure rate is 3 with deviation up to 9. Although failure rate is still affected by saturation, the impact is insignificant compared to forceful failover.

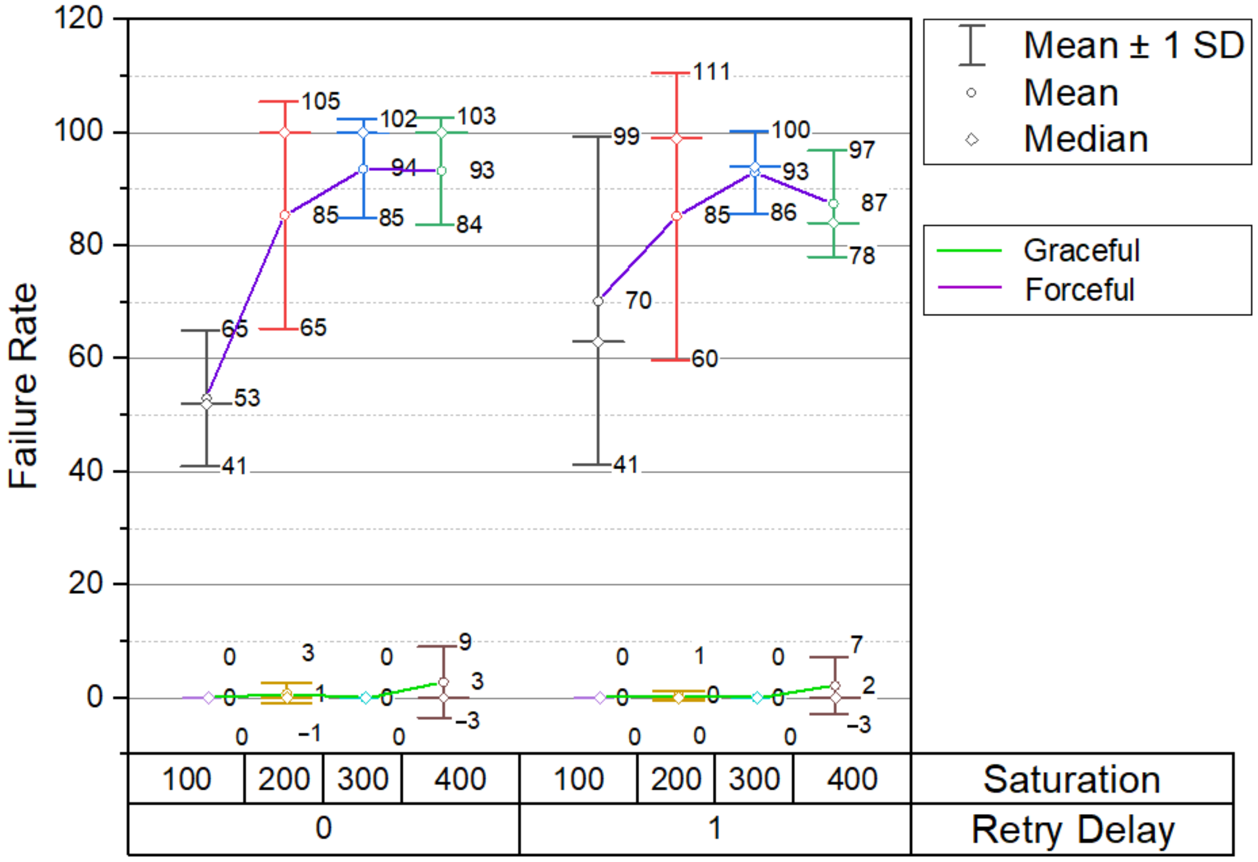


Figure 5. Dependency of failure rate on failover method, saturation and retry delay.

Backed-off retries make an insignificant impact on availability during graceful failover. Retries backed off by 1 s have a positive impact on forceful failover at higher saturation. At 300 parallel requests mean failure rate drops to 93 and at 400 to 87.

We have inspected the failed requests and found that the reason for failure was a deadlock. At high concurrency requests saturate the database node and queries block one another resulting in deadlocks. Deadlocks did not occur with graceful failover as clients moved from one node to another in a gradual manner not overloading the target database node. Although, in the case of forceful failover, database host overload could be alleviated by gradual reconnection so that contention for database resources is lower. However, it may require additional modifications to exception handling on the client side.

Failover from one database node to another takes more time when failing over gracefully (Figure 6). While the mean forceful failover time ranges from 54 to 63 s, graceful failover may take up to three times longer with a mean ranging from 58 to 197 s. Failover duration increases along with saturation, and the process is slowed down by the necessity to inspect the high number of connections. No action may be taken against a connection since it can be actively used by the client at the time of connection termination process. Thus, it is inspected multiple times during prep for failover. The inspection is done in round robin fashion therefore the duration depends on the number of established connections which is high in a highly saturated database cluster. Although graceful failover needs more time to complete there is a certain benefit to it. Client connections are moved to another node in gradual manner and thus the target node is not overloaded.

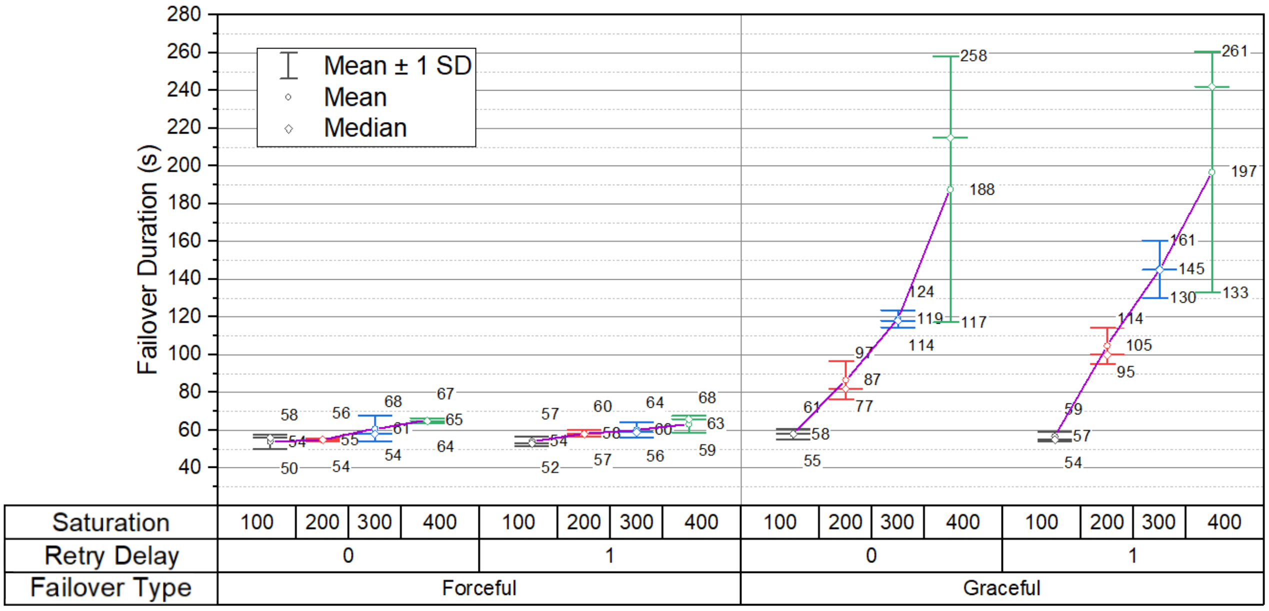


Figure 6. Dependency of failover duration on failover method, saturation and retry delay.

As the other methods and techniques to improve availability have their own advantages the proposed method has its own. First of all the proposed method works with pooled connections allowing to reduce database request latency while giving the flexibility to failover between database cluster nodes with negligible impact on availability. Also managed failover is transparent to an application. Not only is there no action needed on the client side, but, quite importantly, no other exceptions are raised. This reduces coupling between the database cluster and its client in a microservice application. Since no action has to be taken on the client side an operator of the database cluster can perform maintenance on it independently. In addition the method was proved to be functional using open-source software such as MySQL Galera Cluster and ProxySQL. However, the possible implementation options are not limited to the listed software as long as its functionality allows the support of the proposed method.

The method we propose has several drawbacks. A drawback we have identified is the prerequisite of quite a specific architecture for the method to work. A Multi-Primary database setup is needed to achieve a high degree of availability during managed failover. Compared to Single-Primary replication setup a Multi-Master setup is more complex. Complexity results in slower performance due to communication latency and conflict resolution algorithms. However, if the requirements of an application permit, impact of the aforementioned drawback can be limited when database cluster is configured as pseudo-Single-Primary. That would limit the need for conflict resolution as only one database node would accept write workloads. In addition failover time increases when the proposed method is used. In certain setups, as experiments have shown, it may take three times longer to shift connections from one node to another. Although the graceful failover time is longer compared to forceful failover, how big of a disadvantage it is would depend on the real world scenario. Since maintenance is an activity usually planned in advance additional minutes needed to perform the failover may be insignificant in the grand scheme of things. Yet another important aspect is that the proposed method has limited impact on fault-tolerance. Despite allowing to achieve near-zero availability during managed failover, the method is not designed for disaster recovery.

7. Conclusions

The proposed method based on observation and timely connections termination allows to keep failure rate near-zero at high concurrency; only inactive pooled connections are terminated and gradually re-established towards another node without overloading it.

The results have shown that the method we propose can significantly reduce the impact of managed failover on availability. While forceful failover results in no less than half of the parallel requests failing, the proposed graceful failover method allows to achieve near-zero failure rate during failover of a low latency stateful microservice–a clustered database with pooled connections established towards it. Forceful failover failure rates increase as saturation increases. Although graceful failover failure rate increases as well, it is not that significant compared to the other failover type. The achieved results show that retry delay makes an impact only on the forceful failover while graceful failover is unaffected by it. Even with a delayed retry forcefully terminated connections overload the database node upon reconnection resulting in failed requests. With the proposed method the failure rate remains near zero despite a high number of simultaneous connections. However, the time needed to gracefully failover from one database node to another increases along with the number of simultaneous client connections towards a database node. Forceful failover time is affected insignificantly by the number of parallel requests. Implementation of the method does not require substantial modifications on the client side–it needs to retry requests upon a transient connection fault. In addition managed failover operation can be scripted out or, in the case of Kubernetes, can function as part of Kubernetes Operator. Near-zero failure rate during a managed failover operation enables database developers and administrators to perform different maintenance operations at will: change resource allocation or database options.

In this research we have shown that database maintenance operations could have negligible impact on availability in systems requiring low latency while observing a low degree of coupling.

Author Contributions: Methodology, D.M.; Software, K.P.; Investigation, K.P.; Writing—original draft, K.P.; Writing—review & editing, D.M.; Supervision, D.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pereira, P.A.; Bruce, M. *Microservices in Action*; Manning: Greenwich, CT, USA, 2019; ISBN 9781617294457.
1. *ISO/IEC ISO/IEC 25010:2011*; Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. ISO Publishing: Geneve, Switzerland, 2011.
2. O'Connor, P.D.T.; Kleyner, A. *Practical Reliability Engineering*; 5th ed.; John Wiley & Sons, Ltd.: Hoboken, NJ, USA, 2012; ISBN 978-0-470-97981-5.
3. Brewer, E. CAP twelve years later: How the "rules" have changed. *Computer* **2012**, *45*, 23–29. https://doi.org/10.1109/MC.2012.37.
4. Thanh, T.D.; Mohan, S.; Choi, E.; SangBum, K.; Kim, P. A taxonomy and survey on distributed file systems. In Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management, Gyeongju, Korea, 2–4 September 2008; Volume 1, pp. 144–149. https://doi.org/10.1109/NCM.2008.162.
5. Chae, M.S.; Lee, H.M.; Lee, K. A performance comparison of linux containers and virtual machines using Docker and KVM. *Clust. Comput.* **2019**, *22*, 1765–1775. https://doi.org/10.1007/s10586-017-1511-2.
6. Good, B. To Run or Not to Run a Database on Kubernetes: What to Consider. Available online: https://cloud.google.com/blog/products/databases/to-run-or-not-to-run-a-database-on-kubernetes-what-to-consider (accessed on 23 October 2020).
7. Vitess What Is Vitess. Available online: https://vitess.io/docs/overview/whatisvitess/ (accessed on 11 June 2022).
8. Zalando Patroni. Available online: https://patroni.readthedocs.io/en/latest/ (accessed on 11 June 2022).
9. Newman, S. *Building Microservices: Designing Fine-Grained Systems*; 1st ed.; O'Reilly Media: Sebastopol, CA, USA, 2015; ISBN 978-1-491-95035-7.

10. Campbell, L.; Majors, C. *Database Reliability Engineering*; 1st ed.; O'Reilly Media: Sebastopol, CA, USA, 2017; ISBN 978-1-491-92594-2.
11. Kim, J.; Salem, K.; Daudjee, K.; Aboulnaga, A.; Pan, X. Database high availability using SHADOW systems. In Proceedings of the ACM SoCC 2015—Proc. 6th ACM Symposium on Cloud Comput, Kohala Coast, HI, USA, 27–29 August 2015; pp. 209–221. https://doi.org/10.1145/2806777.2806841.
12. Zamanian, E.; Yu, X.; Stonebraker, M.; Kraska, T. Rethinking database high availability with RDMA networks. *Proc. VLDB Endow.* **2018**, *12*, 1637–1650. https://doi.org/10.14778/3342263.3342639.
13. Hong, S.; Li, D.; Huang, X. Database docker persistence framework based on swarm and ceph. In Proceedings of the HPCCT 2019: 2019 the 3rd High Performance Computing and Cluster Technologies Conference, Guangzhou, China, 22–24 June 2019; pp. 249–253. https://doi.org/10.1145/3341069.3342985.
14. Depoutovitch, A.; Chen, C.; Chen, J.; Larson, P.; Lin, S.; Ng, J.; Cui, W.; Liu, Q.; Huang, W.; Xiao, Y.; et al. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In Proceedings of the SIGMOD/PODS'20: International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 1463–1478. https://doi.org/10.1145/3318464.3386129.
15. Hohenstein, U.; Jaeger, M.C.; Bluemel, M. Improving connection pooling persistence systems. In Proceedings of the 2009 First International Conference on Intensive Applications and Services, Valencia, Spain, 20–25 April 2009; pp. 71–77. https://doi.org/10.1109/INTENSIVE.2009.18.
16. Liu, F. A method of design and optimization of database connection pool. In Proceedings of the 2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics, Nanchang, China, 26–27 August 2012; Volume 2, pp. 272–274. https://doi.org/10.1109/IHMSC.2012.161.
17. Sippu, S.; Soisalon-Soininen, E. *Transaction Processing: Management of the Logical Database and Its Underlying Physical Structure*; Springer: Berlin/Heidelberg, Germany, 2017; ISBN 978-3-319-12292-2. https://doi.org/10.1007/978-3-319-12292-2.
18. Kleppmann, M. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*; Spencer, A., Beaugureau, M., Eds.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2017; ISBN 9781449373320.
19. Marinho, C.S.S.; Moreira, L.O.; Coutinho, E.F.; Filho, J.S.C.; Sousa, F.R.C.; Machado, J.C. LABAREDA : A Predictive and Elastic Load Balancing Service for Cloud-Replicated Databases. *J. Inf. Data Manag.* **2018**, *9*, 94–106.
20. Georgiou, M.A.; Paphitis, A.; Sirivianos, M.; Herodotou, H. Towards auto-scaling existing transactional databases with strong consistency. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), Macao, China, 8–12 April 2019; pp. 107–112. https://doi.org/10.1109/ICDEW.2019.00-26.
21. Seybold, D.; Hauser, C.B.; Volpert, S.; Domaschka, J. Gibbon: An availability evaluation framework for distributed databases. In *Lecture Notes in Computer Science*; Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics; Springer: Cham, Switzerland, 2017; Volume 10574, pp. 31–49. https://doi.org/10.1007/978-3-319-69459-7_3.
22. Petrov, A. *Database Internals*; 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2019; ISBN 9781492040347.
23. Mauri, D.; Coriani, S.; Hoffman, A.; Mishra, S.; Popovic, J. *Practical Azure SQL Database for Modern Developers*; Apress Berkeley: CA, USA, 2021; ISBN 9781484263693. https://doi.org/10.1007/978-1-4842-6370-9.
24. Coriani, S. Introducing Configurable Retry Logic in Microsoft.Data.SqlClient v3.0.0-Preview1. Available online: https://devblogs.microsoft.com/azure-sql/configurable-retry-logic-for-microsoft-data-sqlclient/ (accessed on 27 July 2022).
25. Microsoft Inc. Retry Pattern. Available online: https://docs.microsoft.com/en-us/azure/architecture/patterns/retry (accessed on 27 July 2022).
26. Dai, H. Effective apply of design pattern in database-based application development. In Proceedings of the 2012 Fourth International Conference on Computational and Information Sciences, Chongqing, China, 17–19 August 2012; pp. 558–561. https://doi.org/10.1109/ICCIS.2012.138.
27. Hauer, T.; Hoffman, P.; Lunney, J.; Ardelean, D.; Diwan, A. Meaningful Availability. In Proceedings of the NSDI'20: Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation, Santa Clara, CA, USA, 25–27 February 2020; pp. 545–557.