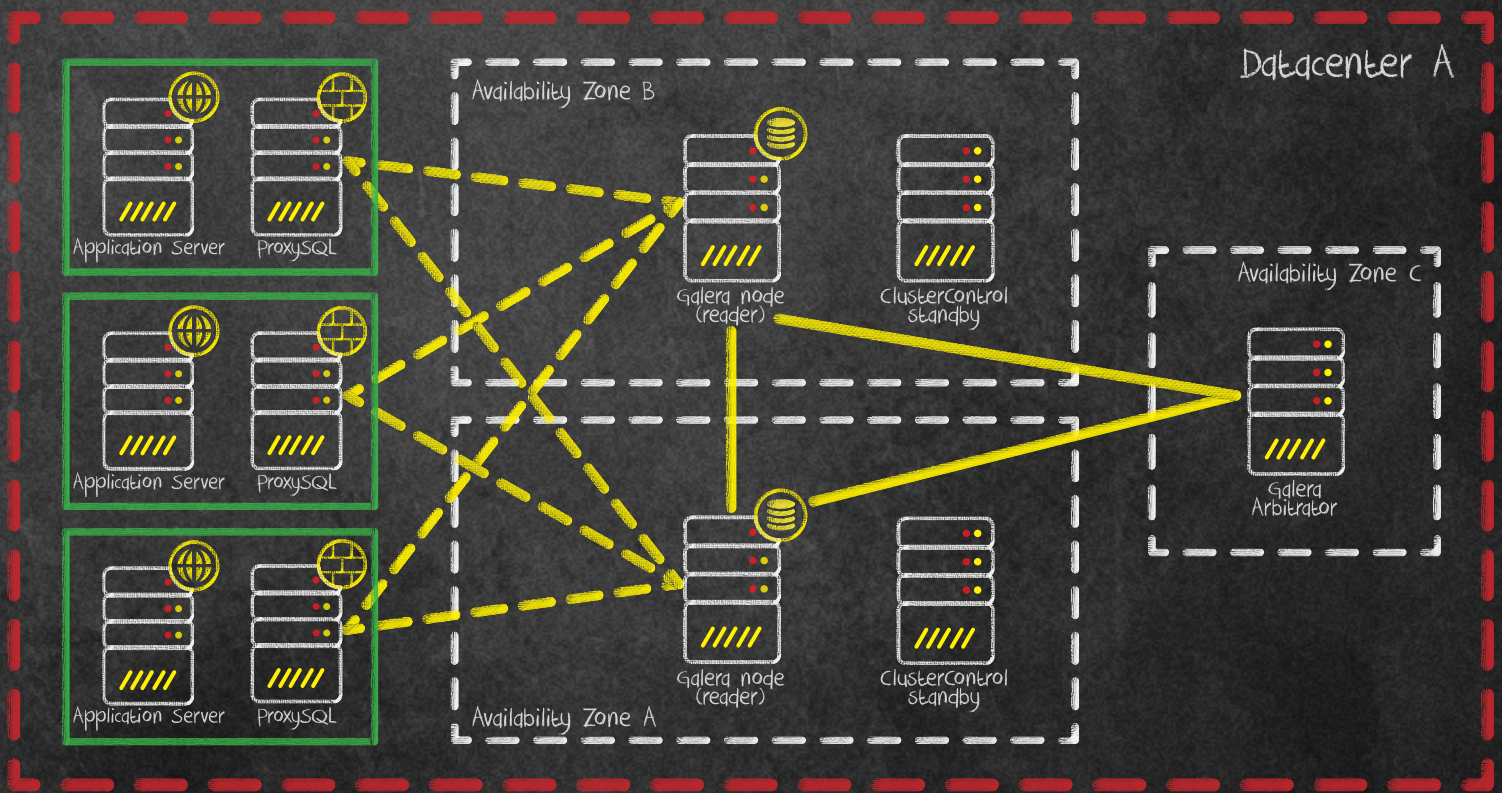
severalnines

How to Design Highly Available Open Source Database Environments



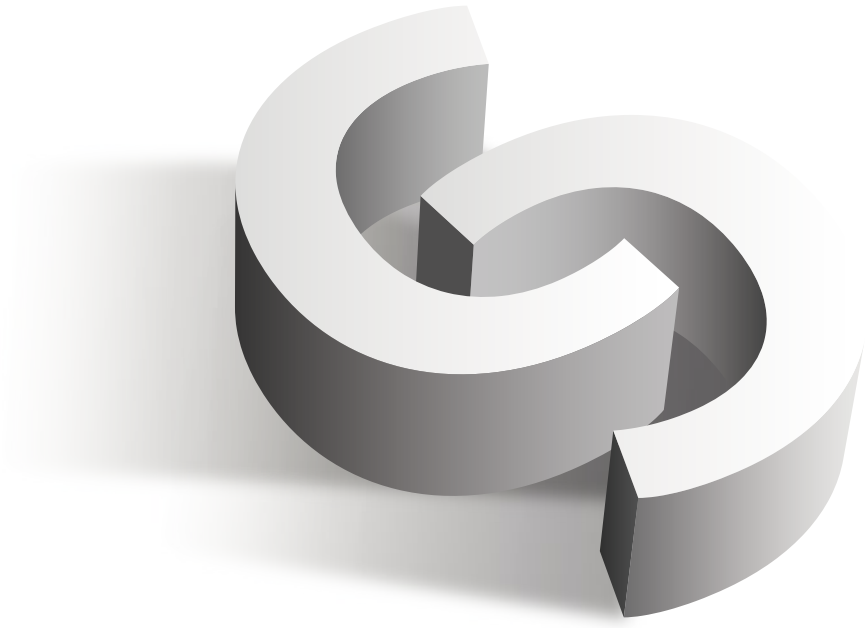


Table of Contents

Introduction - couple of words on “High Availability”

These days high availability is a must for any serious deployment. Long gone are days when you could schedule a downtime of your database for several hours to perform a maintenance. If your services are not available, you are losing customers and money. Therefore making a database environment highly available has typically one of the highest priorities.

This poses a significant challenge to database administrators. First of all, how do you tell if your environment is highly available or not? How would you measure it? What are the steps you need to take in order to improve availability? How to design your setup to make it highly available from the beginning?

There are many many HA solutions available in the MySQL (and MariaDB) ecosystem, but how do we know which ones we can trust? Some solutions might work under certain specific conditions, but might cause more trouble when applied outside of these conditions. Even a basic functionality like MySQL replication, which can be configured in many ways, can cause significant harm - for instance, circular replication with multiple writeable masters. Although it is easy to set up a ‘multi-master setup’ using replication, it can very easily break and leave us with diverging datasets on different servers. For a database, which is often considered the single source of truth, compromised data integrity can have catastrophic consequences.

In the following chapters, we’ll discuss the requirements for high availability in database setups, and how to design the system from the ground up.

High Availability basics

2.1. Measuring High Availability

2.1.1. What is High Availability?

To be able to decide if a given environment is highly available or not, one has to have some metrics for that. There are numerous ways you can measure high availability, we'll focus on some of the most basic stuff.

First, though, let's think what this whole high availability is all about? What is its purpose? It is about making sure your environment serves its purpose. Purpose can be defined in many ways but, typically, it will be about delivering some service. In the database world, typically it's somewhat related to data. It could be serving data to your internal application. It can be to store data and make it queryable by analytical processes. It can be to store some data for your users, and provide it when requested on demand. Once we are clear about the purpose, we can establish the success factors involved. This will help us define what high availability means in our specific case.

2.1.2. SLA's

When working with customers or partners, one would typically define some sort of Service Level Agreement (SLA). It is also quite common to define SLA's for internal services. What is an SLA? It is a definition of the service level you plan to provide to your customers. This is for them to better understand what level of stability you plan for a service they bought or are planning to buy. There are numerous methods you can leverage to prepare a SLA but typical ones are:

- Availability of the service (percent)
- Responsiveness of the service - latency (average, max, 95 percentile, 99 percentile)
- Packet loss over the network (percent)
- Throughput (average, minimum, 95 percentile, 99 percentile)

It can get more complex than that, though. In a sharded, multi-user environment you can define, let's say, your SLA as: "Service will be available 99,99% of the time, downtime is declared when more than 2% of the users is affected. No incident can take more than 15 minutes to be resolved". Such SLA can also be extended to incorporate query response time: "downtime is called if 99 percentile of latency for queries excede 200 milliseconds".

2.1.2.1. Nines

Availability is typically measured in "nines", let us look into what exactly a given amount of "nines" guarantees. The table below is taken from Wikipedia:

Availability %	Downtime per year	Downtime per month	Downtime per week	Downtime per day
90% ("one nine")	36.5 days	72 hours	16.8 hours	2.4 hours
95% ("one and a half nines")	18.25 days	36 hours	8.4 hours	1.2 hours
97%	10.96 days	21.6 hours	5.04 hours	43.2 min
98%	7.30 days	14.4 hours	3.36 hours	28.8 min
99% ("two nines")	3.65 days	7.20 hours	1.68 hours	14.4 min
99.5% ("two and a half nines")	1.83 days	3.60 hours	50.4 min	7.2 min
99.8%	17.52 hours	86.23 min	20.16 min	2.88 min
99.9% ("three nines")	8.76 hours	43.8 min	10.1 min	1.44 min
99.95% ("three and a half nines")	4.38 hours	21.56 min	5.04 min	43.2 s
99.99% ("four nines")	52.56 min	4.38 min	1.01 min	8.64 s
99.995% ("four and a half nines")	26.28 min	2.16 min	30.24 s	4.32 s
99.999% ("five nines")	5.26 min	25.9 s	6.05 s	864.3 ms
99.9999% ("six nines")	31.5 s	2.59 s	604.8 ms	86.4 ms
99.99999% ("seven nines")	3.15 s	262.97 ms	60.48 ms	8.64 ms
99.999999% ("eight nines")	315.569 ms	26.297 ms	6.048 ms	0.864 ms
99.9999999% ("nine nines")	31.5569 ms	2.6297 ms	0.6048 ms	0.0864 ms

As we can see, it escalates quickly. Five nines (99,999% availability) is equivalent to 5.26 minutes of downtime over the course of a year. Availability can also be calculated in different, smaller ranges: per month, per week, per day. Keep in mind those numbers, as they will be useful when we start to discuss the costs associated with maintaining different levels of availability.

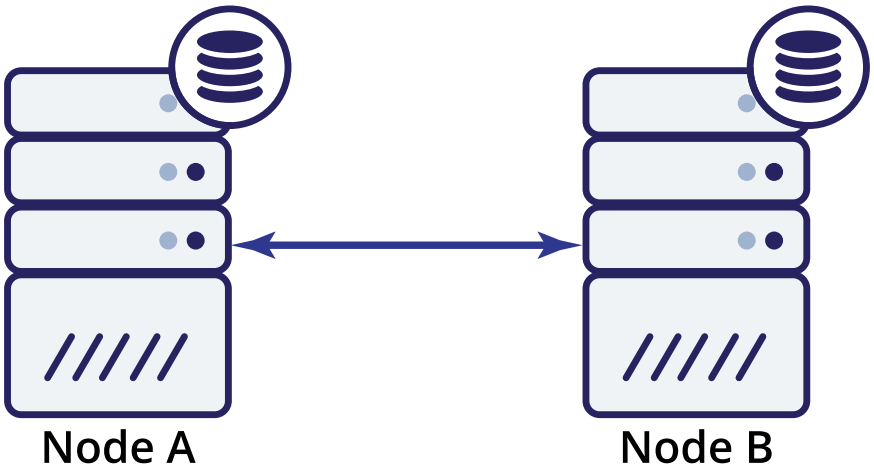
2.1.3. Measuring availability

To tell if there is a downtime or not, one has to have insight into the environment. You need to track the metrics which define the availability of your systems. It is important to keep in mind that you should measure it from a customer's point of view, taking the broader picture under consideration. It doesn't matter if your databases are up if, let's say, due to a network issue, no application cannot reach them. Every single building block of your setup has its impact on availability.

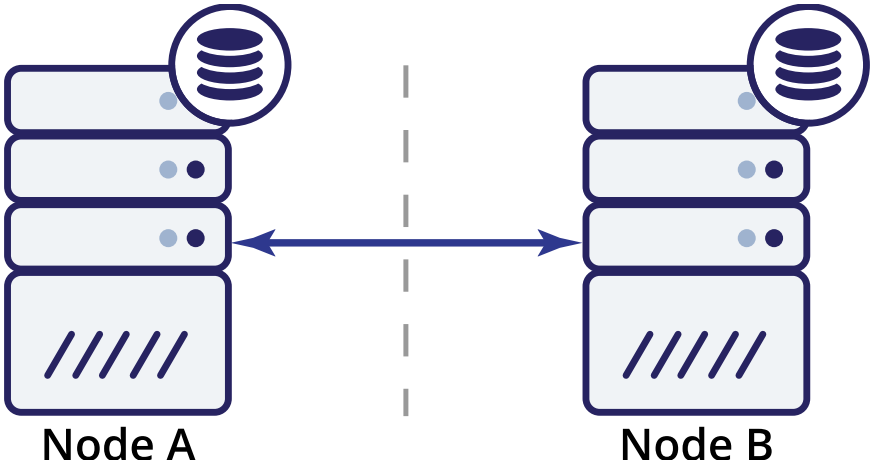
One of the good places where to look for availability data is web server logs. All requests which ended up with errors mean something has happened. It could be HTTP error 500 returned by the application, because the database connection failed. Those could be programmatic errors pointing to some database issues, and which ended up in Apache's error log. You can also use simple metric as uptime of database servers, although, with more complex SLA's it might be tricky to determine how the unavailability of one database impacted your user base. No matter what you do, you should use more than one metric - this is needed to capture issues which might have happened on different layers of your environment.

2.2. Magic number: "three"

Even though high availability is also about redundancy, in case of database clusters, three is a magic number. It is not enough to have two nodes for redundancy - such setup does not provide any built-in high availability. Sure, it might be better than just a single node, but human intervention is required to recover services. Let's see why it is so.



Let's assume we have two nodes, A and B. There's a network link between them. Let us assume that both A and B serves writes and the application randomly picks where to connect (which means that part of the application will connect to node A and the other part will connect to node B). Now, let's imagine we have a network issue which results in lost network connectivity between A and B.

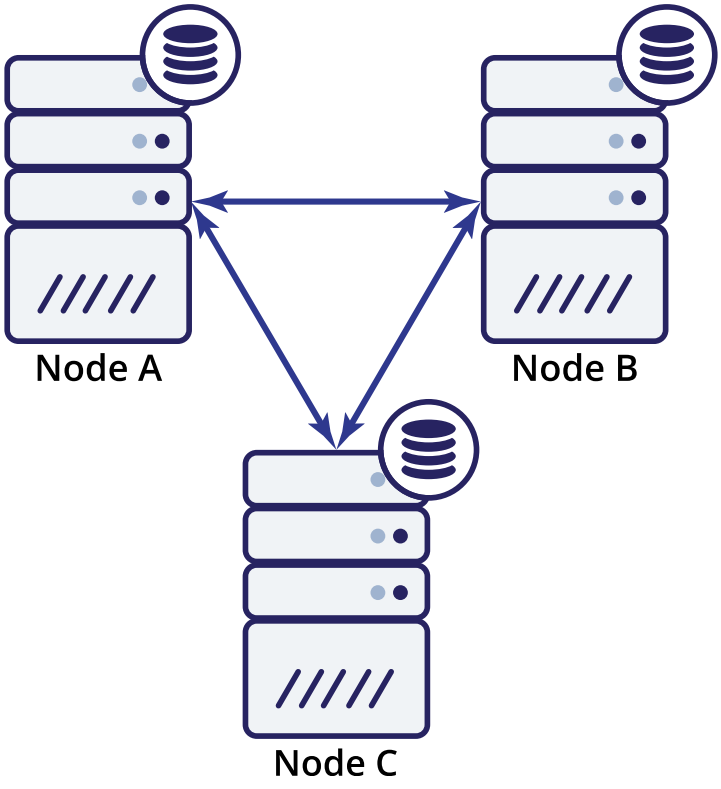


What now? Neither A nor B can know the state of the other node. There are two actions which can be taken by both nodes:

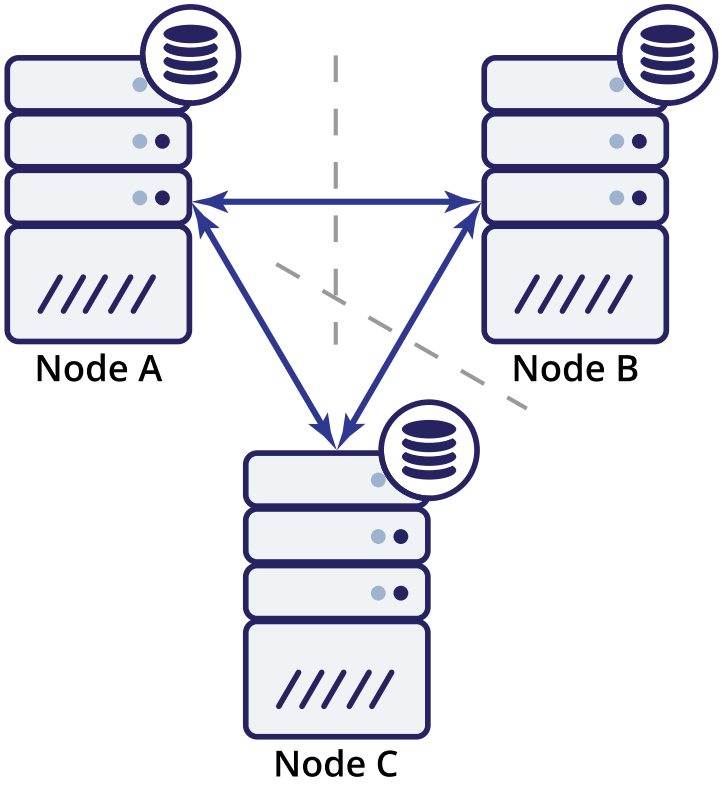
1. They can continue accepting traffic
2. They can cease to operate and refuse to serve any traffic

Let's think about the first option. As long as the other node is indeed down, this is the preferred action to take - we want our database to continue serving traffic. This is the main idea behind high availability after all. What would happen, though, if both nodes would continue to accept traffic while being disconnected from each other? New data will be added on both sides, and the datasets will get out of sync. When the network issue will be resolved, it will be a daunting task to merge those two datasets. Therefore, it is not acceptable to keep both nodes up and running. The problem is - how can node A tell if node B is alive or not (and vice versa)? The answer is - it cannot. If all connectivity is down, there is no way to distinguish a failed node from a failed network. As a result, the only safe action is for both nodes to cease all operations and refuse to serve traffic.

Let's think now how a third node can help us in such a situation.



So we now have three nodes: A, B and C. All are interconnected, all are handling reads and writes.



Again, as in the previous example, node B has been cut off from the rest of the cluster due to network issues. What can happen next? Well, the situation is fairly similar to what we discussed earlier. Two options - node B can either be down (and the rest of the cluster should continue) or it can be up, in which case it shouldn't be allowed to handle any traffic. Can we now tell what's the state of the cluster? Actually, yes. We can see that nodes A and C can talk to each other and, as a result, they can agree that node B is not available. They won't be able to tell why it happened, but what they know is that out of three nodes in the cluster two still have connectivity between each other. Given that those two nodes form a majority of the cluster, it makes possible to continue handling traffic. At the same time node B can also deduct that the problem is on its side. It cannot access neither node A nor node C, making node B separated from the rest of the cluster. As it is isolated and is not part of a majority (1 of 3), the only safe action it can take is to stop serving traffic and refuse to accept any queries, ensuring that data drift won't happen.

Of course, it doesn't mean you can have only three nodes in the cluster. If you want better failure tolerance, you may want to add more. Keep in mind, though, it should be an odd number if you want to improve high availability. Also, we were talking about "nodes" in the examples above. Please keep in mind that this is also true for datacenters, availability zones etc. If you have two datacenters, each having the same number of nodes (let's say three nodes each), and you lose connectivity between those two DC's, same principles apply here - you cannot tell which half of the cluster should start handling traffic. To be able to tell that, you have to have an observer in a third datacenter. It can be yet another set of nodes, or just a single host, with the task to observe the state of remaining dataceters and take part in making decisions (an example here would be the Galera arbitrator).

2.3. Single Points of Failure

High availability is all about removing single points of failure (SPOF) and not introducing new ones in the process. What are the SPOFs? Any part of your infrastructure which, when failed, brings downtime as defined in SLA, is called a SPOF. Infrastructure design requires a holistic approach, the different components cannot be designed independently of each other. Most likely, you are not responsible for the whole design - database administrators tend to focus on databases and not, for example, the network layer. Still, you have to keep the other parts in mind and work with the teams which are responsible for them, to make sure that not only the part you are responsible for is designed correctly but also that the remaining bits of the infrastructure were designed using the same principles. On top of that, such knowledge of how the whole infrastructure is designed, helps you to design the database stack too. Knowing what issues may happen helps to build some mechanisms to prevent them from impacting the availability of the database.

How to design your environment for High Availability?

In this section we will go through the steps which are crucial in building a healthy, highly available environment.

3.1. Identify Single Points of Failure

As mentioned in the previous section, you cannot focus on separate bits of your infrastructure: you have to identify interconnections and relations between the different parts in order to design a truly highly available environment. For instance, the network - does it have enough redundancy built in? Switches, routers - are they doubled? What about hardware - do you have redundancy in I/O subsystem? RAID array? What kind of disks do you use? Can they be replaced online or a server has to be stopped?

With cloud deployments, it is even more complex as you do not have the whole picture. Is the network redundant or not? Does it make sense to bother with bonding network interfaces or are they just virtual entities, related to a single, physical network interface? To what extent is the storage volume redundant? Most likely you will not get answers to all of those questions and, as result, you will have to assume the worst case scenario and prepare for it.

Once you've gone through all aspects of your environment, you can start to plan your database layer. How is your application going to connect to your database? Are you going to deploy a proxy layer? If so, how do you make sure that it will not become a single point of failure? Let's assume you will use proxies for improved flexibility. How are you going to handle service discovery on both application and proxy side? At the end, you'll be adding new databases and proxies to your environment. You need to point your application to newly added proxies. You need to modify your proxies to include newly added database nodes. Are you going to manage the list of infrastructure by hand, will you manage DNS entries by hand, or maybe use external solutions like Consul or Etc.d? If so, how you are going to make those services highly available? If you are going to create scripts to handle the service discovery and application reconfiguration, how are you going to ensure those scripts are always executed? Maybe you should create redundant "management" hosts? This may pose additional problems though. How do you ensure that those two copies of the infrastructure management scripts will be able to work together, and there will be no conflicts between them.

As you can see, going through the whole setup to identify potential single points of failure is quite a time-consuming task, yet it is needed. Please keep in mind, we are not talking about removing the SPOFs yet. For now we have to identify them and have a

plan to mitigate the risk. Next steps will be all about the risk itself and how to calculate it.

3.2. Decide what availability level you want to achieve

The problem with increasing availability of your environment is cost. Let's stop for a second and think about it. At the network level, redundancy means you have to add more than one appliance which serves a particular role. Instead of one switch, you need at least two. Instead of one router, you need at least two. This increases the overall cost of such environment. At the hardware level, it's similar - a server with redundant power supply is more expensive than a server without redundancy. A RAID array which can tolerate two disk failures will require more disks than an array which can tolerate just one failure at a time. At the application level, the same basic principle. The higher the availability, the higher the cost. A cluster which can automatically tolerate the failure of two nodes will require more nodes than a cluster which can tolerate just a single failure. The bottom line is - availability doesn't come for free.

With this in mind, you have to think about your requirements. What is the lowest availability you can accept? This is definitely not the simplest decision to take, but there is a thought process that you can follow.

What is the cost of downtime for my business? How much money will I lose if my services will be unavailable for, let's say, a minute? This can be defined in a customer SLA, or it is something you can calculate based on loss of revenue. Let's imagine that your website generates an income of €1000 per minute. Let's say, it comes out of 50 deals, €20 each. We are talking about averages here - you should have access to your own company numbers. Or, it can be the other way. Let's say that you sell services and each minute of downtime costs you €1000 in some sort of compensation to users. Let us then assume a minute's downtime is €1000 less in the company bank account.

€1000 per minute gives us €52,560 per year if we assume four nines availability (99.99% - 52.56 minutes per year). €1000 per minute gives us €261,600 if we assume three and half nines availability (99.95% - 4.38 hours per year).

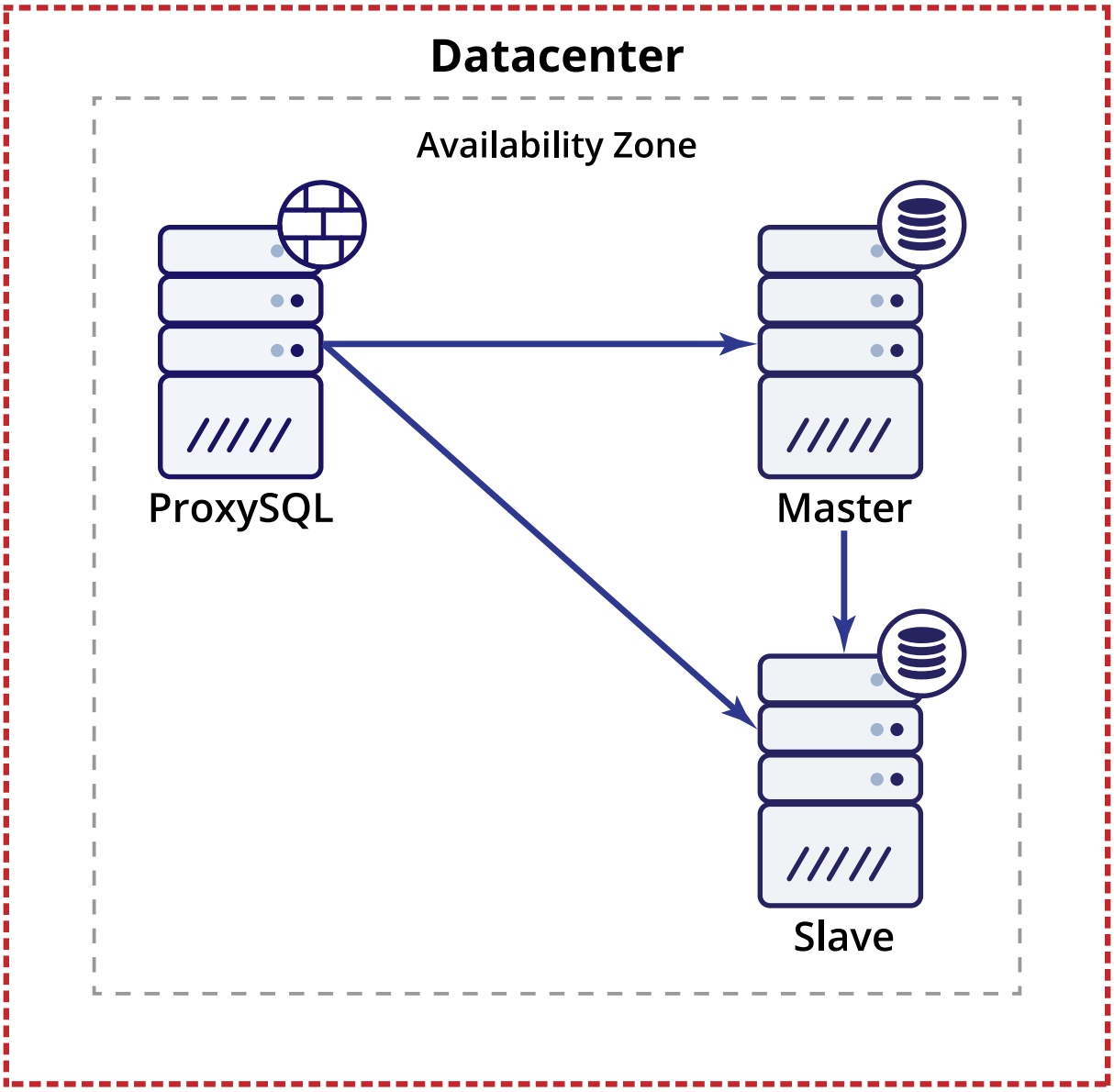
Knowing this we can start to think - what is the availability level you really need to achieve? Let's say that all the expenses needed to bring you to 99.95% availability sum up to €70000 per year. It definitely makes sense to pay that and not lose almost four times more than that in case of downtime. Does it make sense to add another €30000 to reach 99.99% availability? It would not seem so. Would it make sense to invest in 99.95% availability if your loss would be €100 per minute instead of €1000? Not really, you would do better settling for even lower availability level, which would lower the price tag.

This kind of calculation is crucial in understanding how you should design your environment. As we just shown, it doesn't make sense to design high availability just for the sake of high availability - it has to make business sense. Next step will be to determine what kind of failures you can tolerate and which ones are not acceptable.

3.3. Which failures you can tolerate?

Again, you have to stop for a bit and think about it for a moment. Let's say we settled for 99.95% availability, which sets our yearly downtime budget at a little less than 53 minutes. You need to think what can go wrong and how would it affect your downtime. Let's first come up with some examples on how your environment may look like. We will be focussing on the database tier but such planning should be done for every layer of your infrastructure.

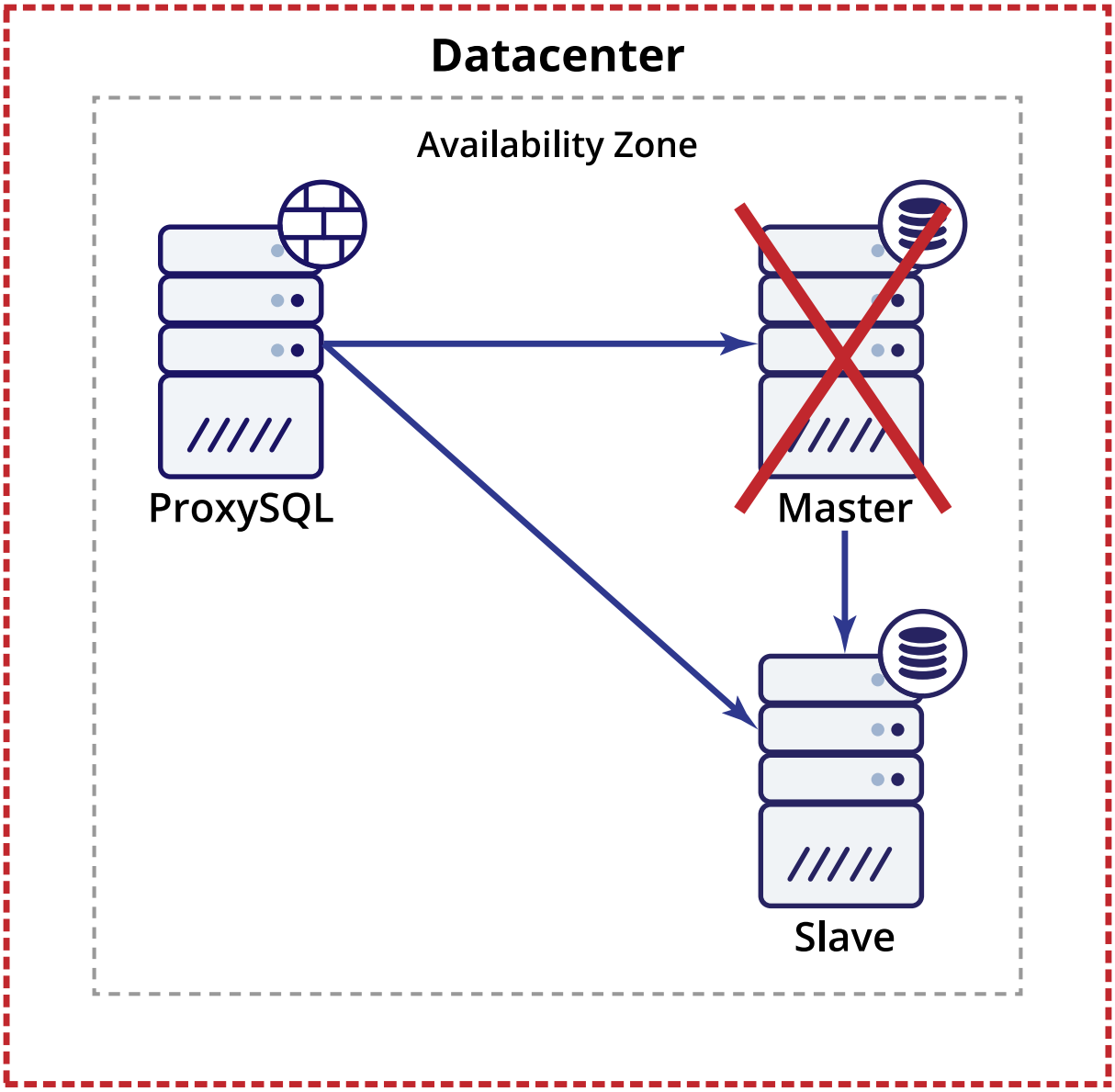
3.3.1. Overall setup



Let's say you have two database nodes in a single availability zone. It does not have to be AWS, we use this term in a more generic fashion - as part of a datacenter, which is self-sustainable. Let's assume this is a replication setup with one master and one replica. On top of that, you have a single ProxySQL instance, which routes queries from your application servers to your backend databases. CPU utilization on the nodes reaches 70%. Backups are in place in physical form (let's assume xtrabackup). To create and restore one you will need 30 minutes in total (10 minutes to create, 20 minutes to restore). There's no failover automation, failover is performed by hand, after a DBA is paged. Let's say this takes, under worst case conditions, 30 minutes for the on-call DBA

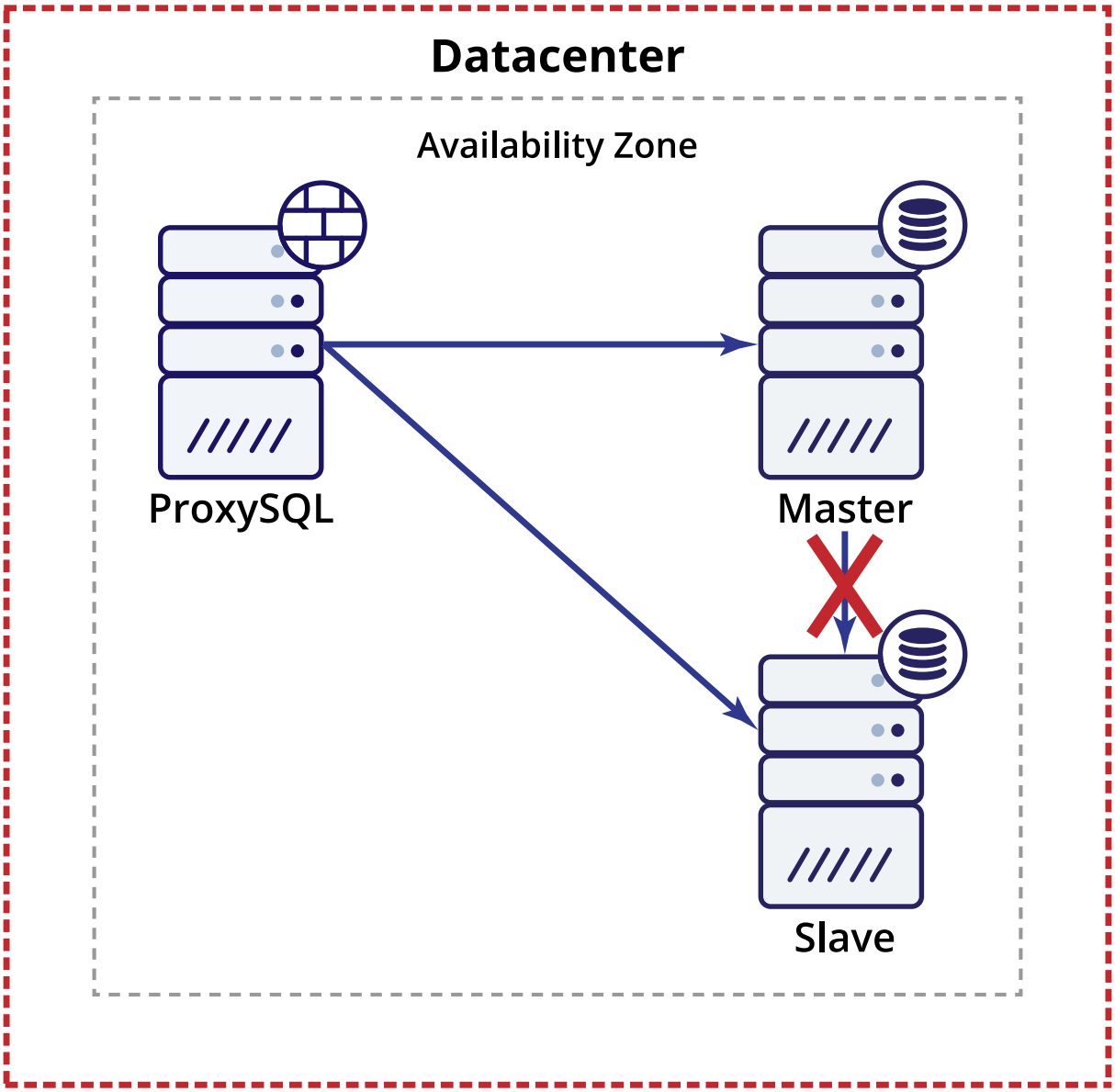
to wake up, react on the page, assess the situation and take an action. From a hardware perspective, let's say that redundancy is there (RAID10 with four disk drives, redundant power supplies etc.). Let's evaluate our single points of failures and what can do wrong.

3.3.2. Hardware failures



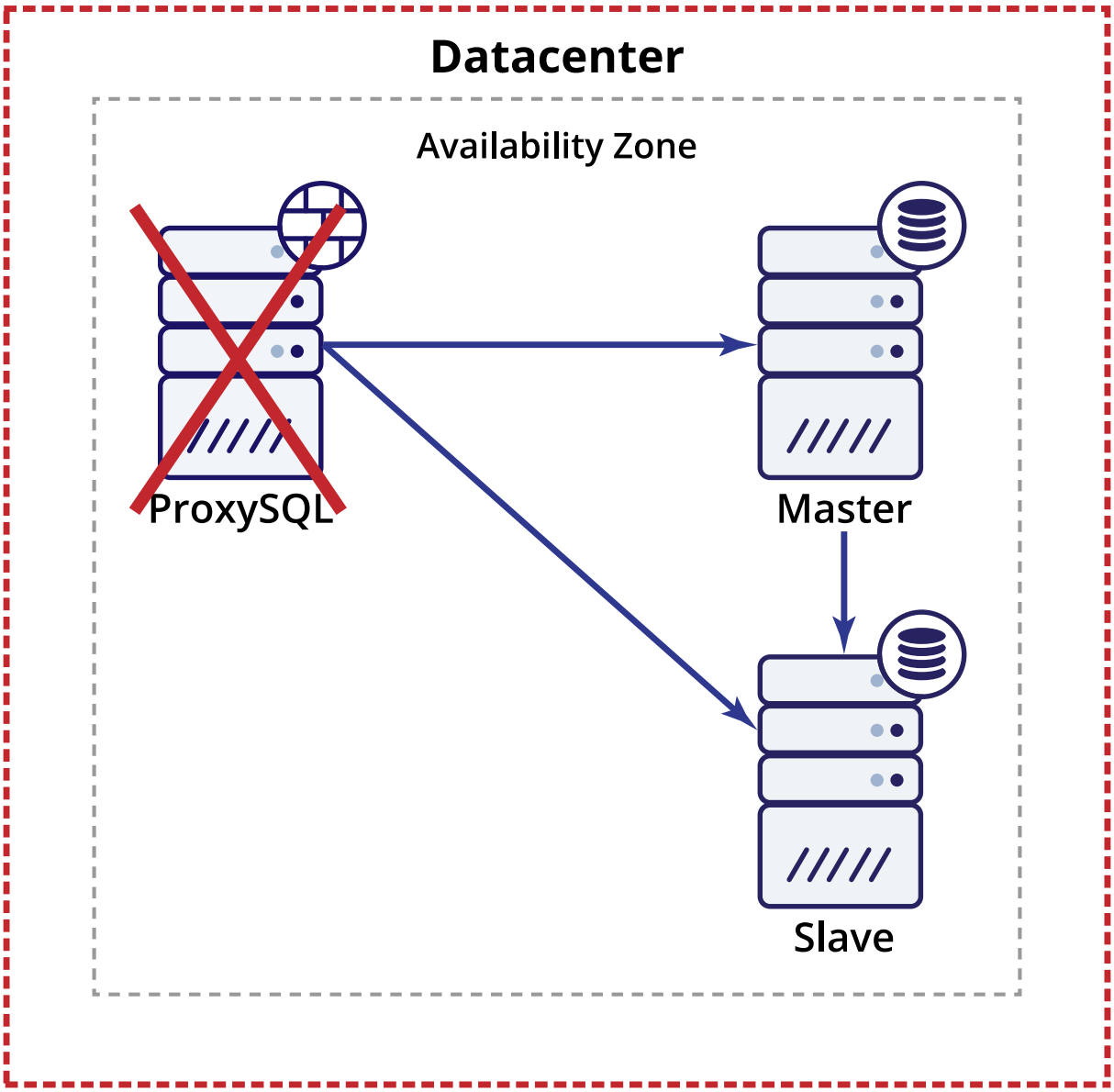
First of all, hardware failures. RAID10 on 4 disks is in place, so we are pretty well covered when it comes to disk failure. Just keep in mind that, sometimes, when disks are from the same series, they tend to fail in batches. Ideally, mix the disk drives from different production series. We can say that the total failure of a single node is rather unlikely. Having said that, the impact is pretty severe. If a master node fails, a slave has to be promoted to master. This takes around 30 minutes in our example. This alone is not enough - with two hosts running at 70% of the CPU utilization, one host will be overloaded with traffic. We need to provision another host, which will take at least 30 minutes to create and restore a backup, and some additional time to setup the replication. We also assume you have hardware in place to replace the old master with a new node. In the cloud, this is rather easy. In an on-prem environment, this alone may be a challenge and it adds to the costs. Anyway, we can say that this is definitely not a type of failure we can tolerate, even if its probability is quite low.

3.3.3. Network failures



We are not going into details of the network redundancy, although it is definitely something you should keep in mind. Let's assume for now there was an issue with the network link between master and slaves and they lost connectivity. It's not likely, assuming proper redundancy, but if it is the case, the slave will start lagging. Eventually, you will have to move all of your traffic to the master to avoid lag. Depending on how much lag your application can tolerate, you may have hope that the issue resolves itself in time, but if it won't, you are then facing severe consequences as the master alone cannot handle all of the traffic. Depending on where the issue originates from, provisioning a new node and setting it up as a slave may be enough. If you are unlucky though, you may find that the issue cut the master from a network segment where your slaves are created, making the issue even more severe. Also, any loss of connectivity between the proxy host and databases will result in severe consequences. We definitely cannot tolerate this kind of issue as it would render our whole application unavailable due to overloaded or unreachable database servers. The probability of it happening is very low though.

3.3.4. Proxy layer failures

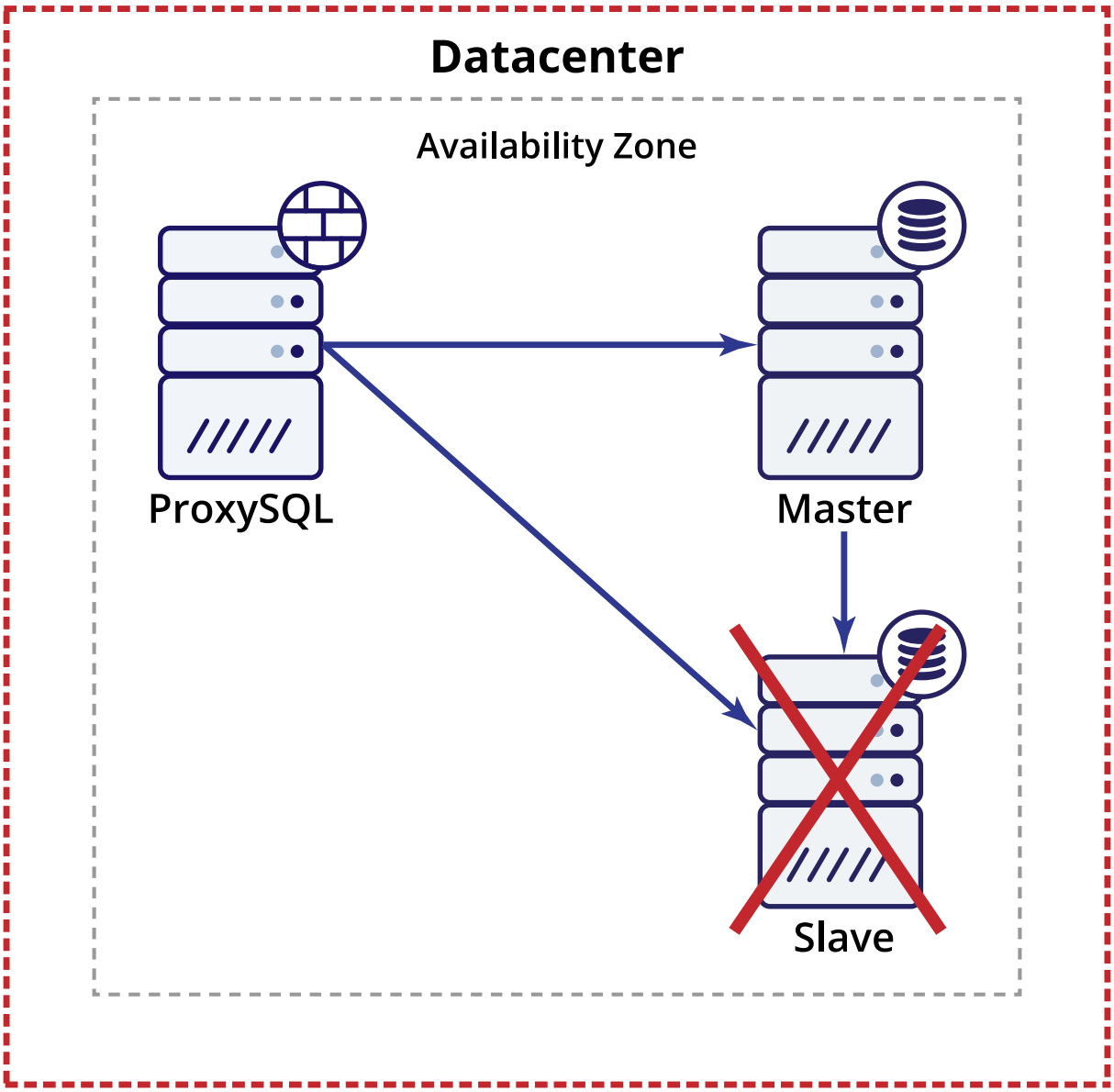


As we mentioned, we have a single ProxySQL instance which is a single point of failure. If it is not available, no traffic can reach the database, which makes it a severe issue. ProxySQL has an angel process which can restart the main ProxySQL process within a second of a crash - therefore we are protected against software being not available, which is more common than when a whole server goes down. We are not protected against hardware issues, so making it a very serious impact, albeit with low probability.

3.3.5. Database tier failures

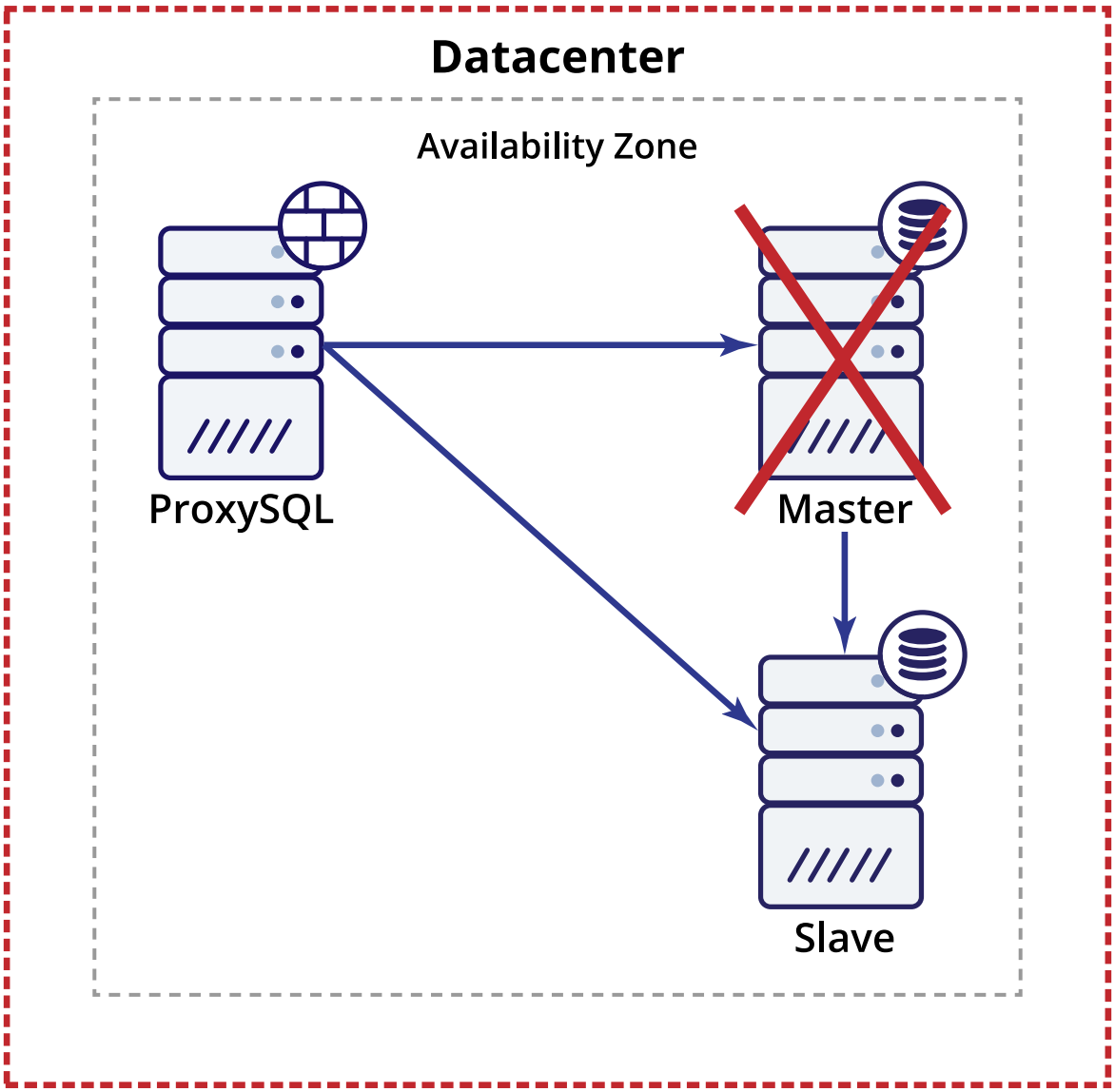
Many things may happen at the database level, let's look at some examples.

3.3.5.1. MySQL crash on slave



Crashes are not common but definitely more probable than whole servers going down. If, for whatever reason, MySQL on a slave goes down, ProxySQL will redirect all of the traffic to the master, therefore overloading it (please keep in mind that both servers have CPU utilization of 70%). When the slave recovers, traffic will be redirected back to it. Typically, such downtime shouldn't take more than couple of minutes (unless we are facing some bug in MySQL/InnoDB related to our workload) so the overall severity is medium, even though probability is also medium.

3.3.5.2. MySQL crash on master



This case is similar to the previous one with an important exception - when the master fails, no traffic will be processed, which is a worse situation than when your MySQL is overloaded - it still can process some requests, at least for a while, before it runs out of open connections. To recover, failover has to be performed (up to 30 minutes, as we assumed) and then the old master has to be slaved off the new master. It may be as fast as running CHANGE MASTER TO ... or as slow as provisioning it from scratch (additional 30 minutes). This poses a severe threat and the probability of it happening is medium.

3.3.5.3. Partial data loss

The case we are considering here can be the outcome of an accidental delete. Some data has been lost or modified and is not usable. It can be either a row or an entire table. The restore process will most likely look the same. You need to restore a backup on a separate host (20 minutes), and then find the missing data, extract it and apply on the master. Depending on the amount of data we are talking about, the whole process, including restoring original backup, can take some time, starting from 25 minutes. Most likely it'll be completed within 50 minutes, otherwise it'll be faster to provision everything from scratch. The probability of this to happen could be assessed as medium, impact may vary from low to high, depending on what kind of data we are talking about. We'd say on average it's low impact - most likely just a handful of customers are affected.

3.3.5.4. Complete data loss

Here you are in deep problem, there's no data at all on our database nodes. You have to restore backup on both hosts and then setup replication between them. Total time to restore will probably be around 30 minutes when done manually - you can implement some parallelization but only to some extent. Severity is high, probability is low.

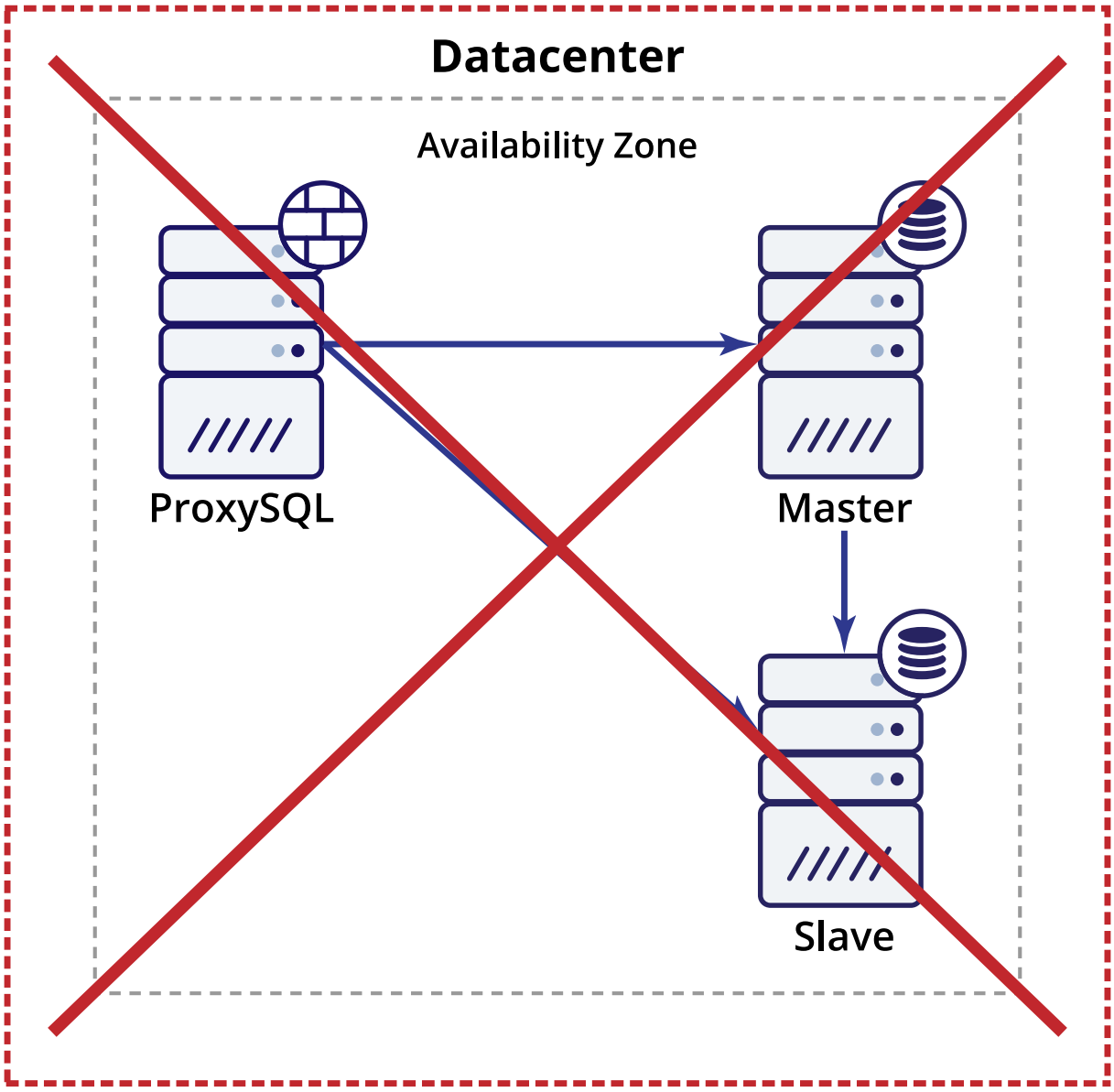
3.3.5.5. Temporary load spike

Your servers suffered from a temporary increase of load. Depending on how long the spike will last, severity will differ from low to medium. Worst case scenario, your databases will start to slow down significantly. Best case, it will be hardly visible. Probability of this to happen is medium to high, depending on your workload.

3.3.5.6. Increased load due to bad query

Your servers suffer from increased load triggered by an incorrectly written query, or a query which does not use indexes. Given that we use ProxySQL, you can easily rewrite or even totally block this query, making this a low severity case even though it's highly probable to show up.

3.3.6. Availability zone or a datacenter failure



All of the issues mentioned above happen within an environment limited to a single availability zone and a single datacenter. In case of issues with the underlying infrastructure, you cannot do anything. If, at any time, either the availability zone or datacenter would become non-available, our services will not be available either. On top of that, recovery time is unknown as it does not depend on the DBA - the ball is in the court of the hosting provider, or the people managing the datacenter. This makes it extremely hard to predict the duration of the impact. We know only that the impact will be extremely severe.

3.3.7. What issues cannot be tolerated?

We gave some examples of issues which may impact our high availability. We need now to discuss which of them can be tolerated and which cannot. Definitely, all issues that affect our SLA in a major way are to be avoided- if we cannot recover services within 53 minutes, we are in serious troubles. Here we can mention, for example, master crash or, generally speaking, issues with hardware. If we use a large part of our SLA for a given incident and we expect the type of incident to be pretty common, then it is also something we cannot really tolerate. Most likely, we'll suffer from this issue a couple of times per year. An example here can be a MySQL crash on the slave.

3.4. Remove SPOF's and reduce the impact of issues with high severity

3.4.1. Identify the culprit of the issues

Once you decide the type of issues you cannot tolerate, it's time to figure out their root causes. You have to identify the source of those issues before you can redesign your environment to mitigate them. Let's go through the examples covered in the previous section, and see if we can identify the source of the problems. It is very likely that many issues would be traced to the common root. Once we identify where the issues originate from, we will think how can we reduce their impact.

3.4.1.1. Hardware issues

Generally speaking, the main issue to solve here is the fact that, when one of the hosts is not available, the other one will not be able to handle all of the traffic. Another, serious issue is the failover time - if it is the master that is affected, a slave will have to be promoted and this takes time and effort. So, we have two culprits - not enough resources to handle traffic in a degraded state, and a slow failover process.

3.4.1.2. Network issues

Here we described a couple of problems and we need to consider several cases. If the problem is related to the master only, making it not available, most likely we will have to perform a failover. We also do not have enough resources to handle load if one of the hosts gets cut off from the network. We also considered lack of connectivity between proxy host and the database tier - there's no redundancy in the proxy layer to handle network failure (or any other, for that matter).

3.4.1.3. Proxy layer issues

As we mentioned above, there is no redundancy in the proxy layer - anything here which would make the proxy not reachable would have serious consequences.

3.4.1.4. Database tier issues

MySQL crashes on master and on slave - in both cases, the culprit is the lack of resources to handle traffic when one node goes down. On top of that, in case of master failure, failover takes too much time. Regarding partial data loss scenarios, we are limited to the time needed to recover from a backup. It takes quite some time to perform recovery and then dump the data and recover it on the production cluster. Full data loss - we are limited to how fast you can restore your backup. Load spikes - one of the culprits is, definitely, lack of resources to accommodate increased load for some time.

3.4.1.5. Infrastructure issues

Here it is simple - we have all our eggs in one basket, this will be the main culprit in case of a datacenter-wide outage.

3.4.2. How to minimize the impact of the issues?

In the section above, we came up with a list of culprits our troubles may originate from. Let's summarize these here:

- Not enough resources to handle failure of a single node
- Failover is not fast enough
- No redundancy in proxy layer
- Long backup recovery time
- No redundancy in terms of the infrastructure

What can we do here to reduce the impact? Let's go through it one by one.

3.4.2.1. Not enough resources to handle failure of a single node

The solution here is to use more slaves, to have more capacity for handling an increase in CPU utilization triggered either by failure of a node or, an occasional spike in the workload. While doing that, we have to keep in mind that all of the nodes should contain the same hardware specification - in order to make sure that all of them can take any kind of the role (slave or master). It may sound obvious, but we still see some shortcuts taken here.

Another important aspect here could be the automation of provisioning a new node. Ideally, as we said, you have a buffer to accommodate a traffic increase. If the traffic is even higher, though, you'd want to add a new slave as fast as possible. Two options here are worth considering. For starters, is the provisioning process the most efficient? Maybe you can switch the provisioning method to something else? For example, you may see from time to time people who use logical dumps as a provisioning method. It is a valid method but it is slow. Are you in an environment which provides volume snapshots? Maybe consider switching to it instead of running xtrabackup, as it might be faster that way. Second thing to consider - automation. Even though the provisioning process can be performed manually, it is still better to write code to automate it - humans tend to introduce latency and sometimes, human errors. If you ever had to reprovision a slave because you made some mistakes in slaving it off, then you'll know what we are speaking about here. Well written, well tested and well maintained code will be faster and not affected by human errors.

3.4.2.2. Failover is not fast enough

This is another serious issue which we identified. First of all, why is failover slow in our case? Well, it is slow because it's a manual failover. The worse case is if it happens in the middle of the night, when you are woken up by a page. First, you need to realize you got paged. Then you have to get up. A couple of minutes have passed already, and this is assuming you managed to be woken up by the first page attempt. Then, you need to get onto a VPN and connect to your company's network - which adds another couple of minutes. Next, you need to figure out what happened. SSH to the master, see if it is reachable, check the state of the slaves, verify that the failover is necessary and then perform it. By this time you may already be 15 - 20 minutes into the incident. A simple way of solving the problem is to perform the failover automatically. In that case, it will be executed in a matter of seconds from the time the master goes down. Of course, a human still has to be woken up to verify that everything happened as planned, but the reaction time is not that critical as before. Using tools like ClusterControl, Orchestrator or MHA, you can reduce the time needed for failover to tens of seconds at most, significantly reducing any impact the failover may have on your SLA.

3.4.2.3. No redundancy in the proxy layer

The simple solution here is to add redundancy. There are a couple of gotchas, though. First of all, you have to decide how are you going to design high availability of the proxy layer? Are you going to use a simple round-robin connectivity from the application? Are you going to use some sort of VIP in front of the proxy layer? Are you going to use a loadbalancer (e.g., ELB if you are on AWS) in front of a proxy? Those are the questions you'll have to answer when adding redundancy.

3.4.2.4. Long backup recovery time

You need to figure out if your backup process can be improved in terms of speed. Maybe you take a daily (or even weekly) full backup and the rest is covered by replaying binary logs? In that case adding some incremental backups, executed on an hourly basis, might help. Maybe there is a way to improve the parallelization of one or more of the backup process steps? Maybe you can improve recovery time by changing the backup type?

3.4.2.5. No redundancy in terms of the infrastructure

As with the proxy layer, the answer is to add redundancy. Maybe you should utilize more availability zones? Maybe you should span your infrastructure across multiple datacenters? This is not trivial task to accomplish, as it may well add more single points of failures. But that's the step you need to take if you want to minimize the risk of infrastructure failure.

3.5. Design the environment

We now know what kind of issues we have to deal with in order to improve our high availability. Keeping all of them in mind, it's now time to try and design an environment, which won't have all the flaws of the current setup and which will be able to meet our SLA and the availability levels we want to reach. As a reminder, here is the list of issues we identified:

- Not enough resources to handle failure of a single node
- Failover is not fast enough
- No redundancy in proxy layer
- Long backup recovery time
- No redundancy in terms of the infrastructure

3.5.1. Database tier design

Let's start with the last point - infrastructure redundancy. As we discussed earlier, "three" is the number we shall count therefore we will use three datacenters. Two will host database nodes, a third one will keep an "arbitrator". The issue is that MySQL replication does not support quorums, it is not aware of this concept. It makes it quite tricky to handle network splits. We have two options. We can either switch to Galera cluster and use Galera's quorum mechanism, spread four nodes between two datacenters (two nodes in each) and add a Galera arbitrator in the third datacenter. We can also stick to MySQL replication, but then we have to either prepare home-grown software which would handle network partitioning and quorum, or use something like Orchestrator/Raft for network partitioning detection and automated failover.

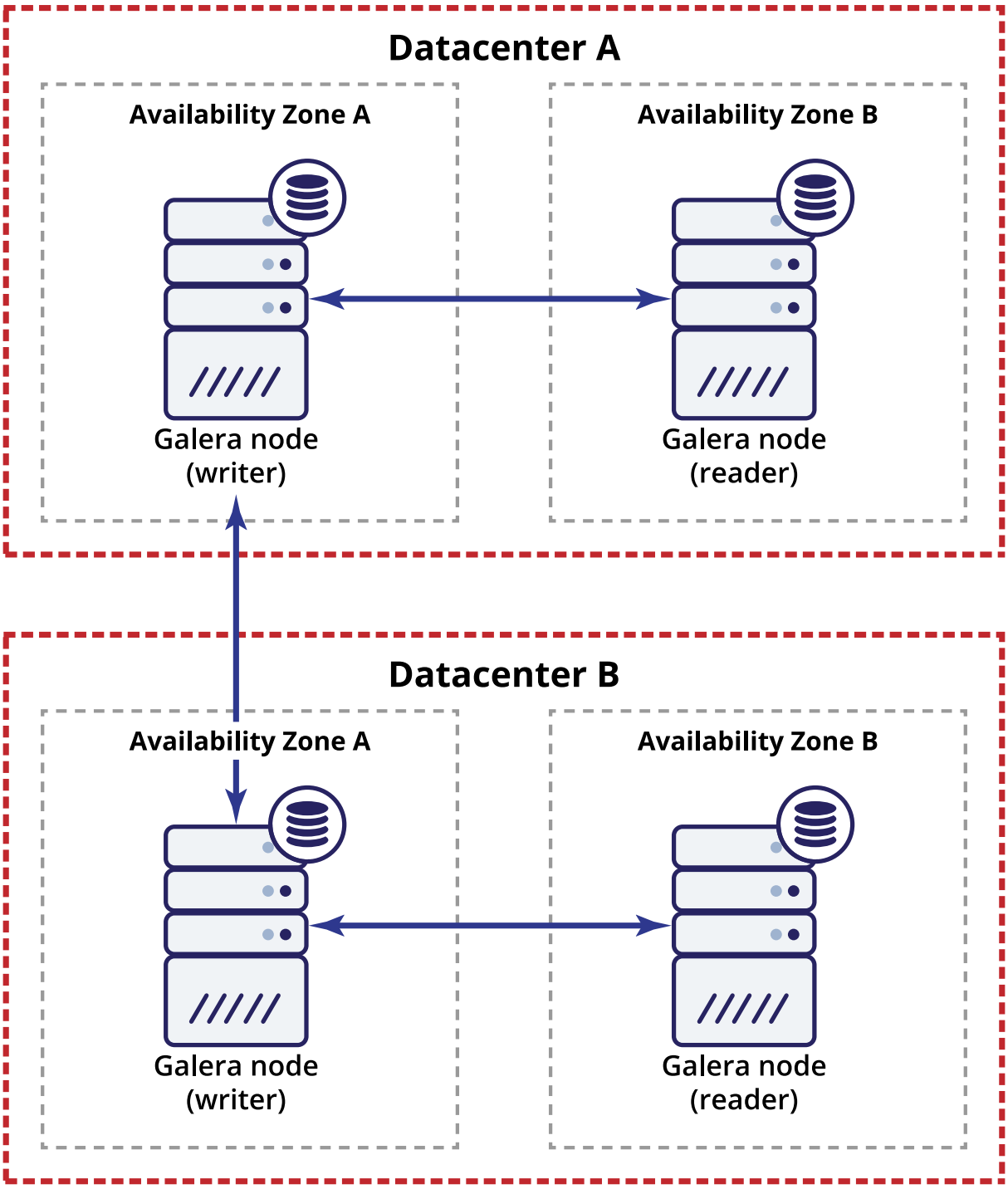
In our example, let's assume we decided to go with Galera cluster as we want to keep the network split handling within the database itself. Note that Galera is not always the best choice for all workloads, it is not a drop-in replacement for MySQL/InnoDB. Yes, it uses InnoDB as storage engine, it contains the entire dataset on every node, which makes JOINs feasible. But some of the performance characteristics of Galera (like the performance of writes which are strictly tied to network latency) differ from what you'd expect from replication setups. Schema change handling works slightly different too. Some schema designs are not optimal: if you have hotspots in your tables, like frequently updated counters, this may lead to performance issues. Batch processing works differently - instead of large batches, you want your transactions to be small. This is by no means a full list of differences between standalone MySQL/InnoDB and Galera cluster, but it illustrates that some work might be needed when switching to Galera. For the sake of simplicity, we are going to say that our workload is compatible with Galera and it will work just fine in our case.

So, to sum it up, four Galera nodes across two datacenters, and a Galera arbitrator in the third datacenter. This setup provides tolerance for failure of up to two nodes - a whole datacenter may go down and the other one will continue to handle services. On top of that, we'll distribute our nodes between different high availability zones, to make sure that the failure of a single availability zone won't take out all our nodes within that datacenter. It is quite important, considering how Galera can provision new nodes - it performs state transfer by copying all data from one of the existing, operational nodes. It is important to have such a node in the same datacenter, as moving the whole dataset over WAN will significantly increase the time needed to provision a new node.

There's actually one more decision we need to make: Are we going to use both datacenters at once, or will they be in an active-standby configuration? This is an important decision given that, as you may remember, two nodes can hardly handle the load. If one would become unavailable, the other would fail under the load. While we would have three nodes, sending traffic over the WAN to the remaining two nodes in the second datacenter is not the best option. We can mitigate this issue by increasing the number of nodes to six, three in each datacenter. We can also benefit from another feature of Galera and use two writers, one in each datacenter. As a result, the incoming traffic would be distributed between both datacenters, utilizing all four of Galera nodes and reducing the load on the individual instances. This can also be a good occasion to reduce expenses - as long as you can accept some performance issues if a whole datacenter will become unavailable, you can reduce the size of the nodes in a way that, load-wise, the cluster can handle the failure of a single node only. It is yet another trade-off you can do. Regarding expenses, we will reduce inter-datacenter traffic by utilizing Galera segments - only two nodes, one from each datacenter, will exchange traffic over WAN.

So far we covered two points - we can now handle a datacenter failure, we can handle failure of up to two nodes. Let's take a look at the failover times. If we are going to use ProxySQL along with Galera cluster (which is a very common setup, everything can be deployed from ClusterControl), assuming that ProxySQL will be configured correctly , "failover" can happen within seconds. To be precise, there's no such thing as failover in Galera cluster - all you need to do is to start writing to another node. ProxySQL can be used to detect the state of the nodes and redirect traffic from one to another if the existing writer is unavailable for some reason.

To sum up what we discussed so far, let's take a look at the database tier design:

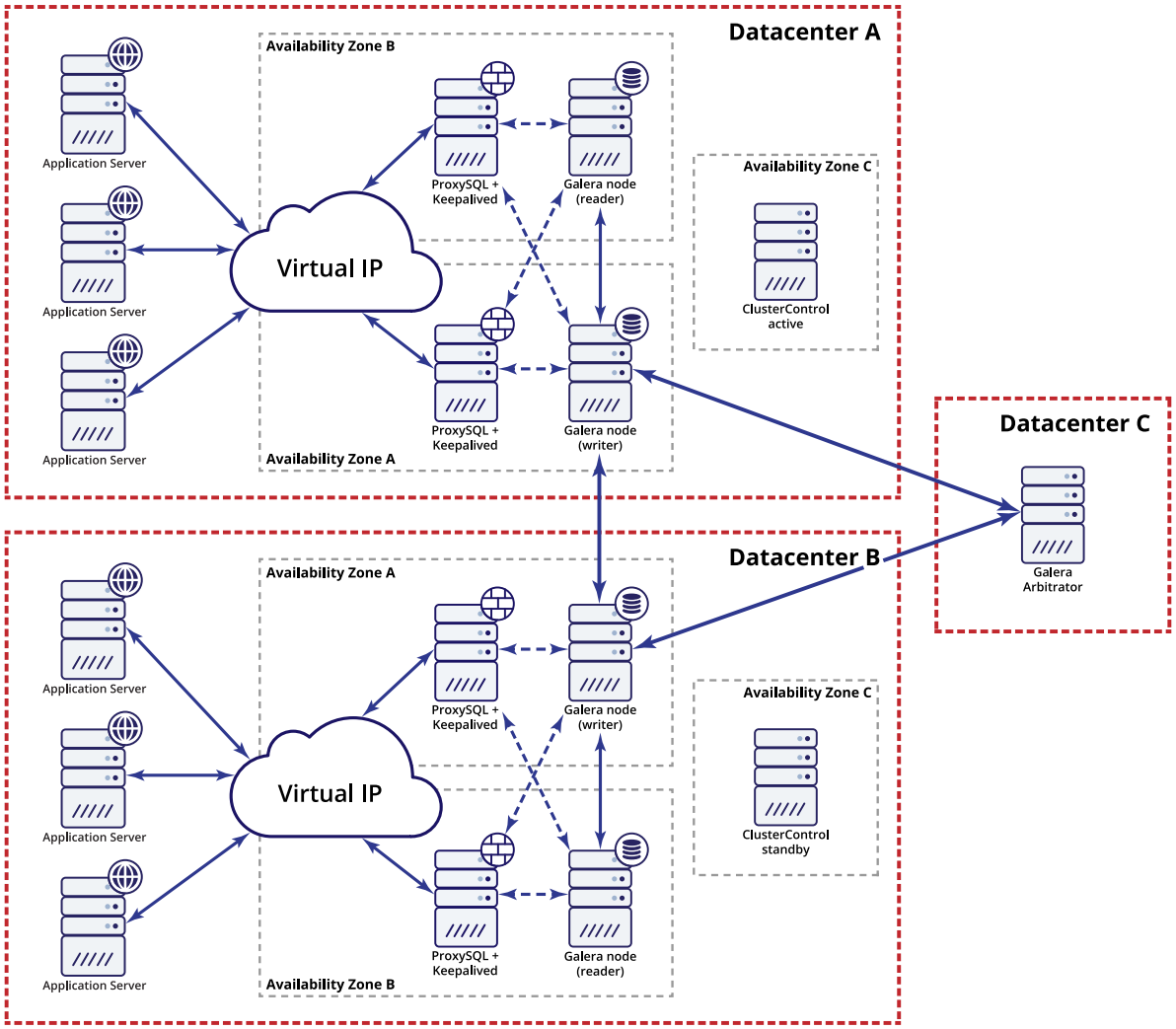


3.5.2. Proxy tier design

We had mentioned lack of redundancy in the proxy layer, let's address that. First of all, as we mentioned earlier, ProxySQL can handle its own failures. It uses an angel process which is intended to restart the 'proxysql' process should it become unavailable. We still have to account for host failures. There are many ways you can solve this problem, what you'll use typically depends on your particular setup. In quite small environments like the one we're designing, two deployment patterns are quite common.

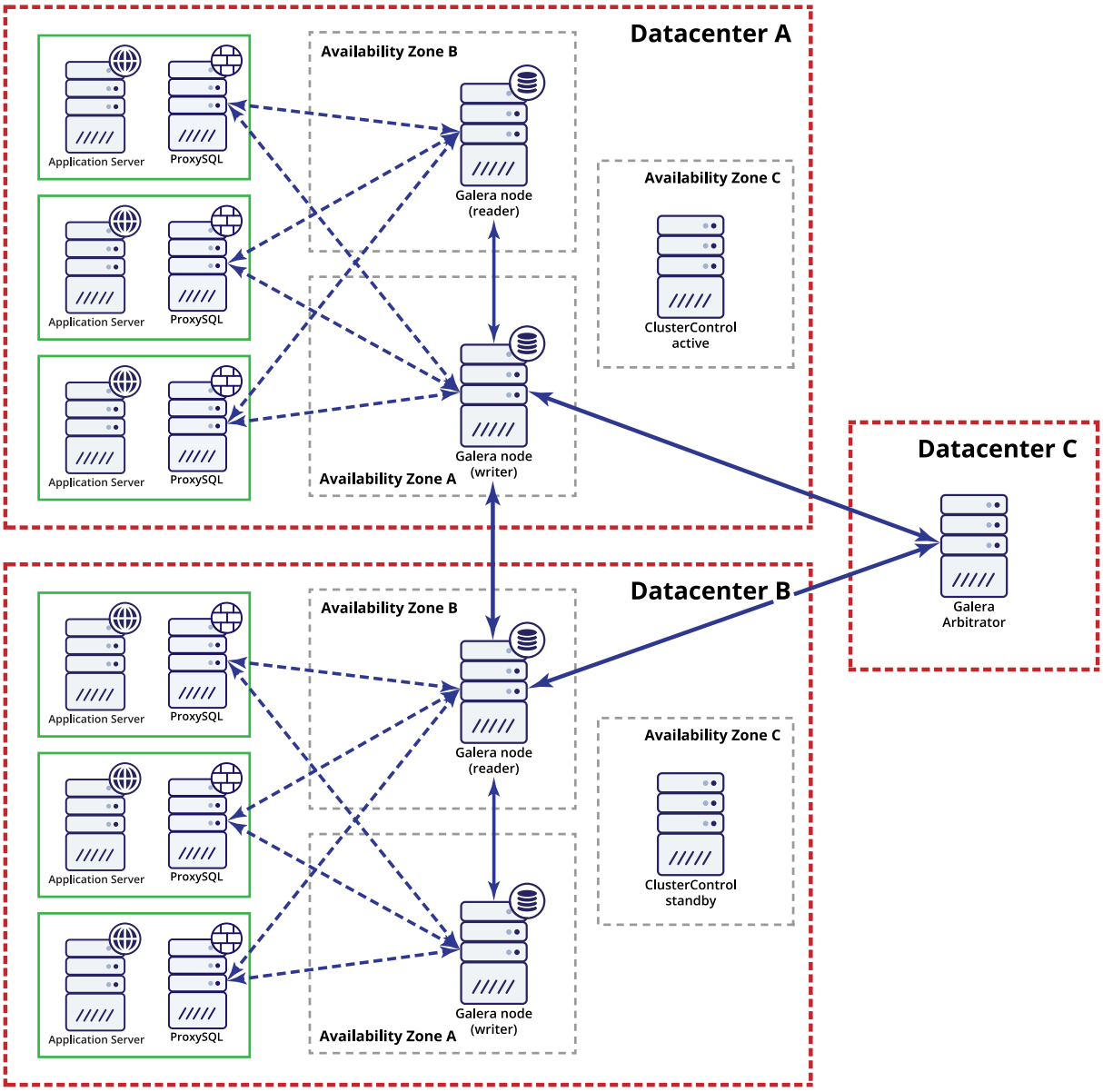
3.5.2.1. Deploy ProxySQL with Keepalived for VIP failover

The idea here is to leverage Keepalived for VIP failover. You need to decide where to deploy ProxySQL. Would it be on the database hosts or dedicated hosts? We'd recommend against collocating ProxySQL (or any other proxy for that matter) on the database servers - proxies will induce additional CPU utilization making it quite hard to predict when you should scale your database tier. We can use separate hosts dedicated to ProxySQL. ProxySQL, like other proxies, uses mostly CPU, so the instances don't have to have beefy storage - magnetic disks will work just fine. On top of two or three ProxySQL instances we can deploy Keepalived and configure it to monitor the ProxySQL process. In real life this will not monitor ProxySQL's processes as ProxySQL can restart its processes faster than Keepalived can detect it crashed. The idea here is to move VIP around if the whole node goes down making both ProxySQL and Keepalived not available. From the application point of view, all it has to do is to connect to the VIP and it will reach the database as long as one of the ProxySQL nodes would be available. Such setup can be easily deployed from ClusterControl, using just a couple of clicks. Here's how our setup could look like:



3.5.2.2. Deploy ProxySQL on application hosts

Another common pattern is to deploy ProxySQL on the same host as the application servers. Each application node is configured to connect to the local ProxySQL, using Unix sockets. Such setup can handle ProxySQL crashes within a second via the angel process we mentioned earlier. On the other hand, if the whole server crashes, that particular application node will go down along with ProxySQL, while all other application nodes can continue to connect via their respective ProxySQL instances. This particular setup has couple of nice features. One is security - ProxySQL, as of version 1.4.7, does not have support for client-side SSL. It can only setup SSL connection between ProxySQL and backend. Collocating ProxySQL on the application host and using Unix sockets is a good workaround for this limitation. Additionally, if you are going to use ProxySQL for caching your queries, it makes sense to keep it as close to the application as possible. Local connection via Unix socket will have lower latency than remote connection via TCP making the use of the cache faster. Here's how such a setup could look like:



3.5.2.3. Synchronization of the ProxySQL configuration

At any time, if you use more than one ProxySQL instance, you have to plan how you will keep them in sync regarding their configuration. Most likely, you want all of the instances to use the same query rules, the same set of users and the same configuration of the backend servers. If you scale your backend, you want to add the new nodes to all of the ProxySQL instances. Doing it by hand is quite daunting and error-prone. It's so easy to make some mistake and skip some of the changes one some of the instances. There are couple of ways you can work around this problem. ProxySQL has an ability to synchronize configuration across multiple nodes. When configured properly, every change made on a single node will be transferred to all of the remaining nodes in the ProxySQL cluster. This is very nice and efficient solution, it just makes it tricky to propagate changes just to a subset of nodes (should such need arise, although it is more common to have everything synced). Another option, if you use ClusterControl, is to use a built-in mechanism for syncing ProxySQL nodes. It gives you an option to define which node should be the source and which other node should be the destination. You also have more control over when exactly you want to sync such configuration, making tests easier. Finally, you can always revert to infrastructure orchestration tools like Ansible, Chef, Puppet or SaltStack. Most of them have modules which support ProxySQL, so it's quite easy to use them to maintain and propagate configuration changes.

3.5.3. Backup redesign

Last on the list of problems we identified is the time needed to recover from backup. The problem with backups is that the time needed to create and recover from a backup is correlated with the size of the data and you can't always improve it. There are still a couple of things you can try to speed up the process.

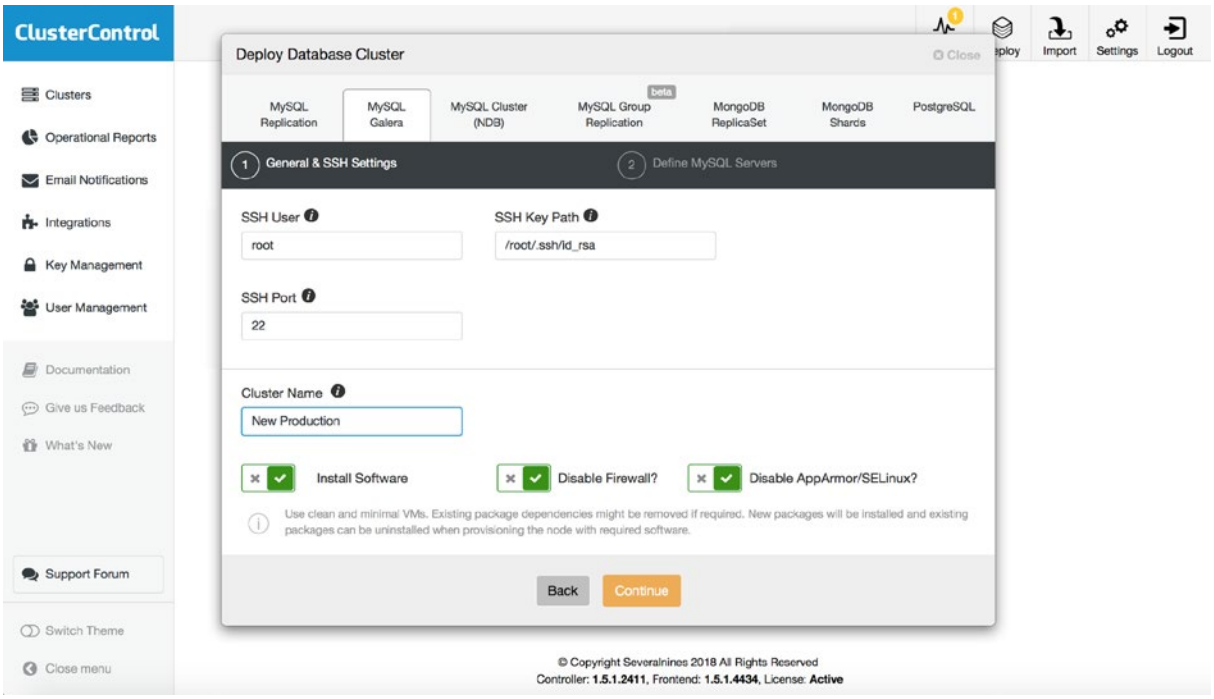
For starters, use physical backup - it will be so much faster than the logical backup. If your environment gives you an option to use filesystem snapshots, test them and see if they can help you to recover even faster. Of course, faster disks will also improve your backup and recovery times. If you happen to stream your data to external servers, consider keeping the last backup locally, on the node. It will be so much faster to restore the backup from a local copy than to stream it back over the network. Of course, in case of streaming, network performance also matters and you may want to look into upgrading it. Make sure you take backups often - more frequent backups means less binary logs to apply, and therefore faster recovery time. Automate the backup and restore process. Ideally, you'll just execute a script which will perform recovery for you - it'll be more efficient than having to perform the same operations manually. For partial recovery, consider using a delayed slave. Delay it by 5 - 10 minutes. You can use more slaves, with different delays for each of them. If you manage to catch the data loss event in time, you can just wait for the delayed slave to replicate up to that transaction, then stop the replication, dump missing data and restore it on the production cluster. Such process typically will be faster than restoring the data from scratch and then dumping the missing data to restore them on the production. The larger the dataset, the bigger will be the difference in recovery time in favor of the delayed slave.

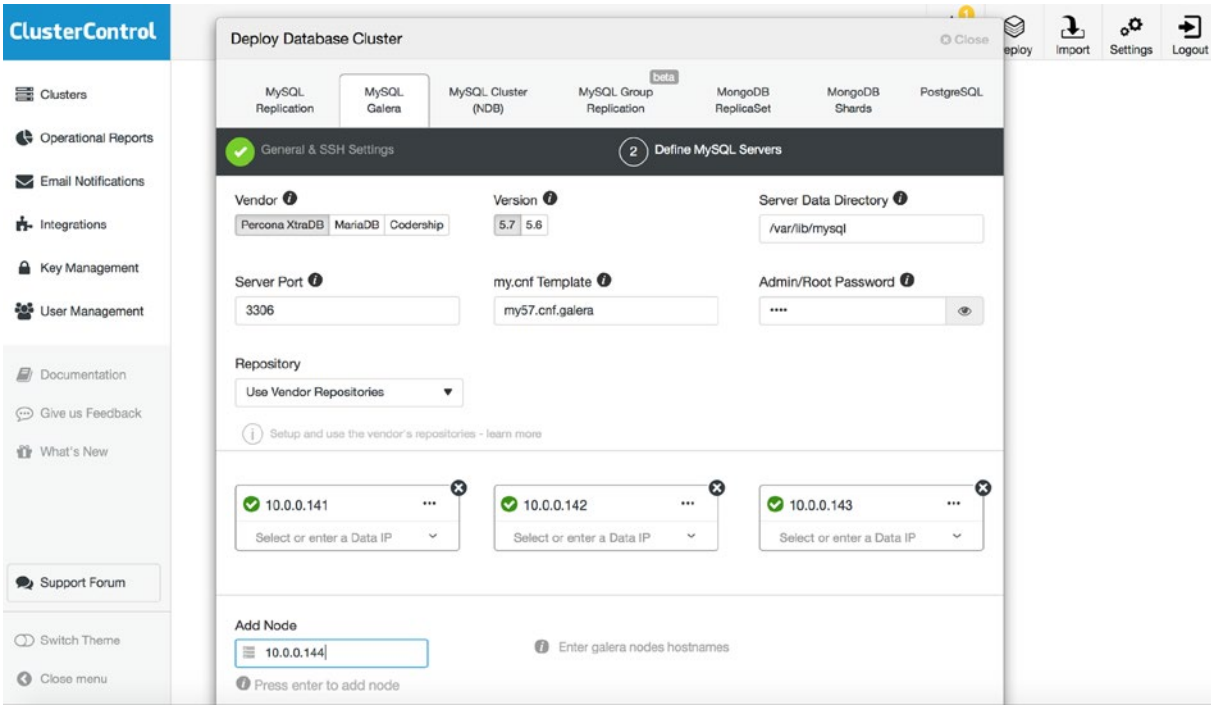
3.5.4. Deployment

We went through the planning phase, we have two designs to decide on. As this is a thought experiment, we won't make any decision on which design to use. With a small number of application servers, it may make sense to collocate ProxySQL and application. Otherwise it may become hard to coordinate everything and it might be better to utilize larger, dedicated instances for the proxies. Next, let's think about how

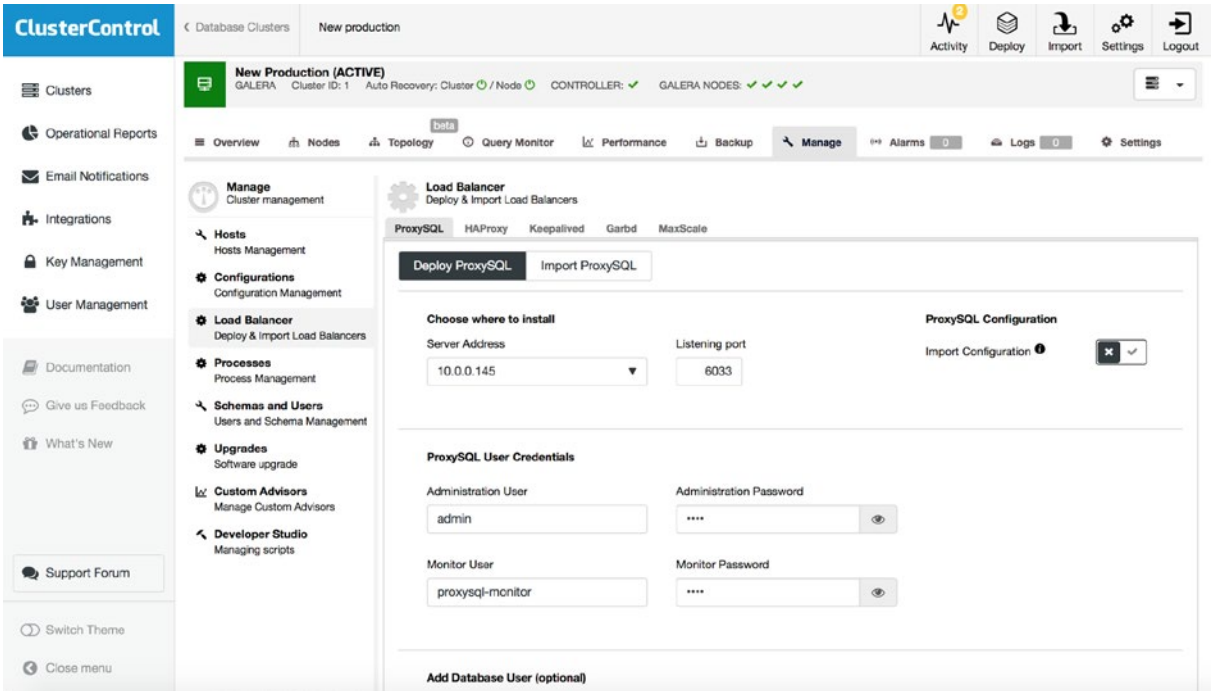
to deploy, manage and scale such setup. There are numerous ways to do that - starting from deploying everything by hand, which is a rather inefficient way of doing it. You can write scripts which will automate this process or write playbooks, cookbooks, recipes for different infrastructure orchestration tools like Ansible, SaltStack, Chef or Puppet. It will let you use a large number of modules designed to deploy different pieces of the infrastructure, but note that you still need to create and test the scripts to deploy the database cluster, taking into account the inter-node dependencies. Finally, we also have a specialized tool like ClusterControl that can readily deploy the database components we described above. We are going to deploy using ClusterControl, as it will also help us monitor and maintain the setup.

Here are the steps you would take in order to deploy such an environment with ClusterControl. First of all, you need to deploy a Galera cluster. You can do it through the deployment wizard.

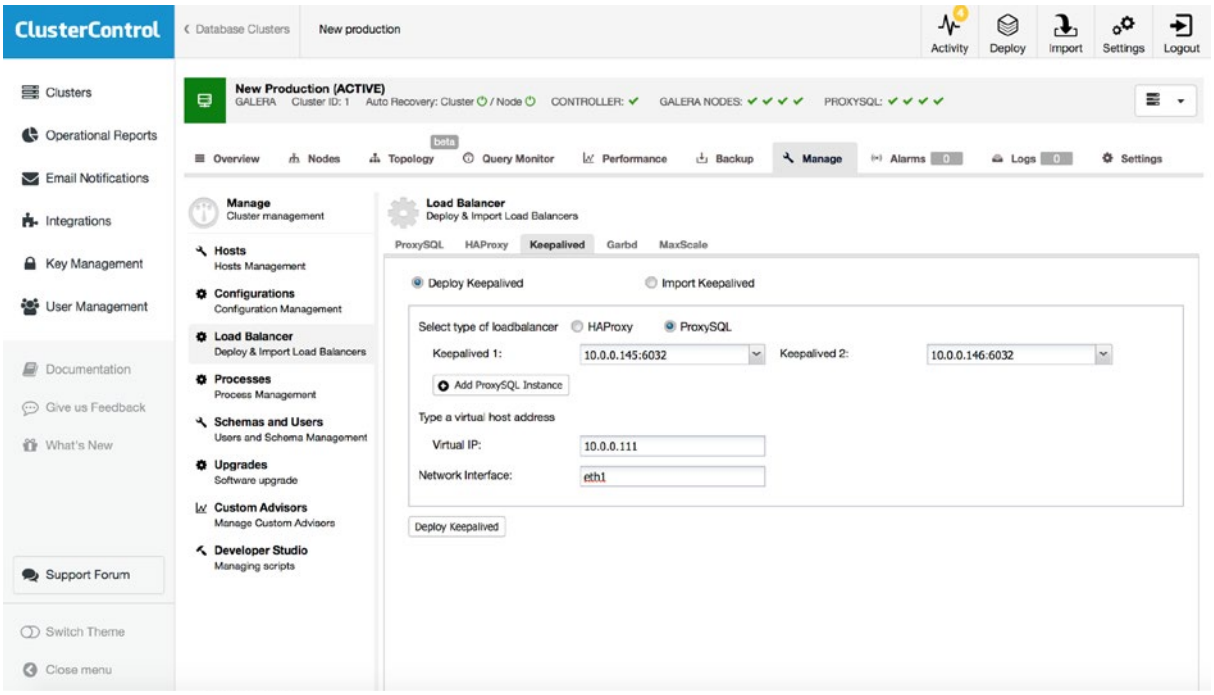




You have to define how ClusterControl can access your hosts and then decide on what Galera flavor you want to have deployed. You need to pass the IP's of the Galera cluster and ClusterControl will commence deployment.

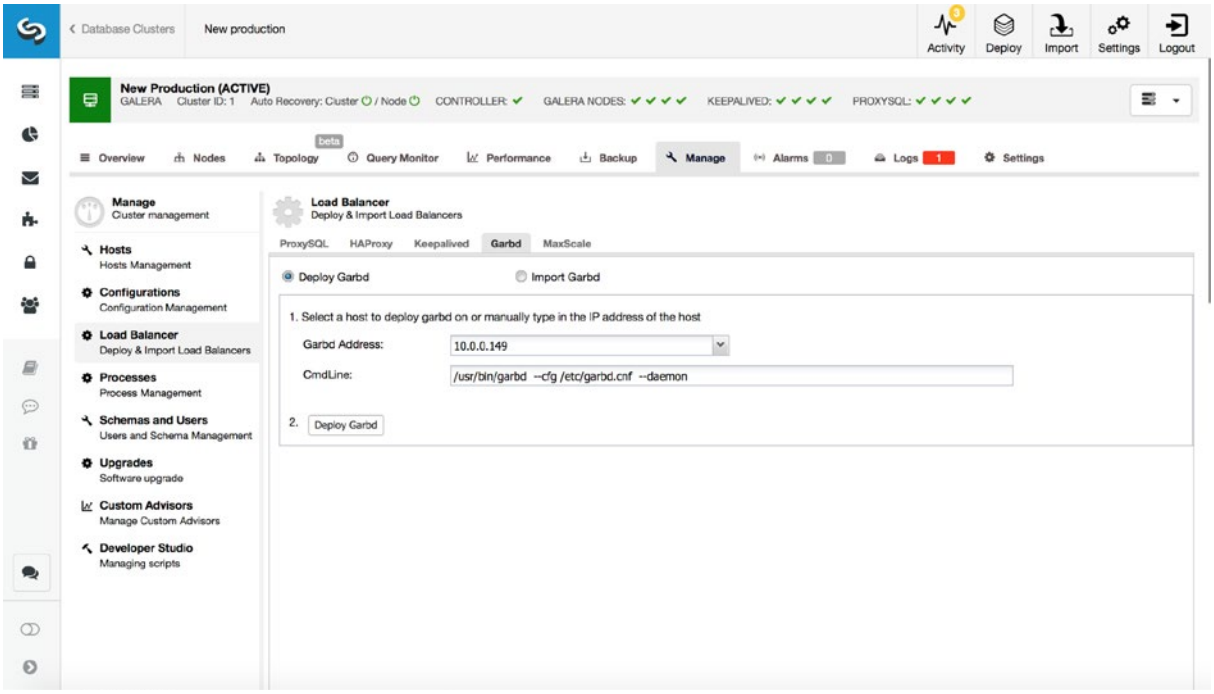


Next step will be to deploy four ProxySQL instances, which can be done also through ClusterControl. It is up to you if you will deploy ProxySQL on the application hosts or on separate, dedicated instances.

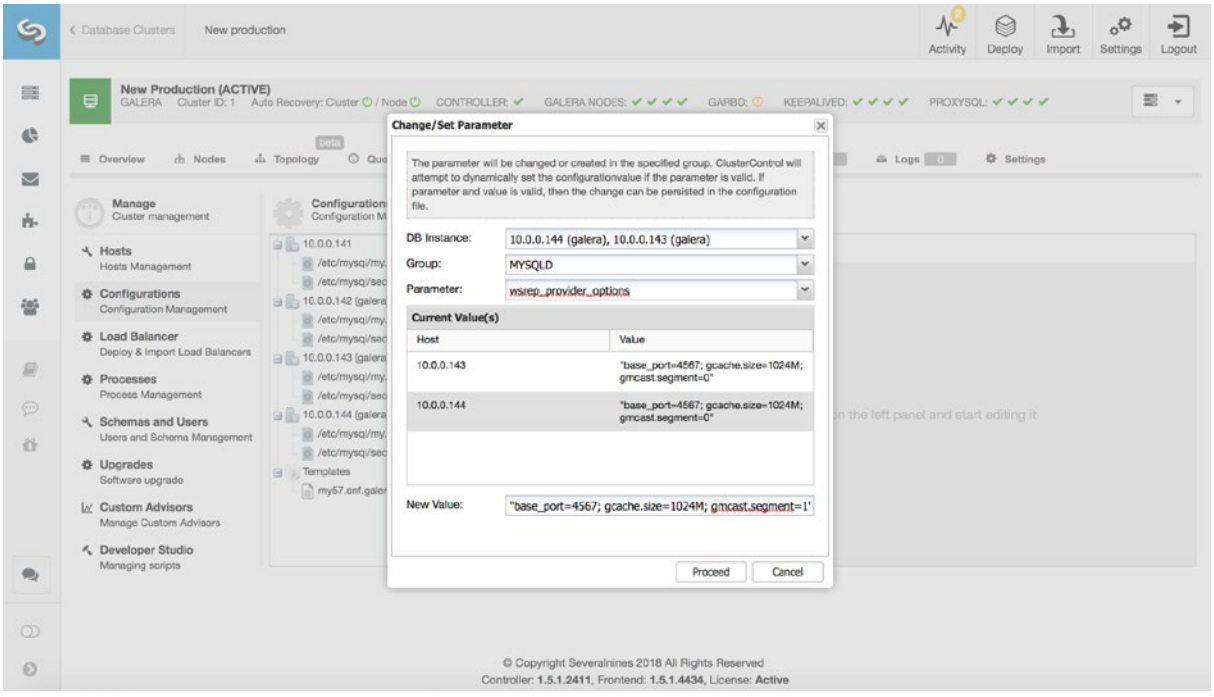


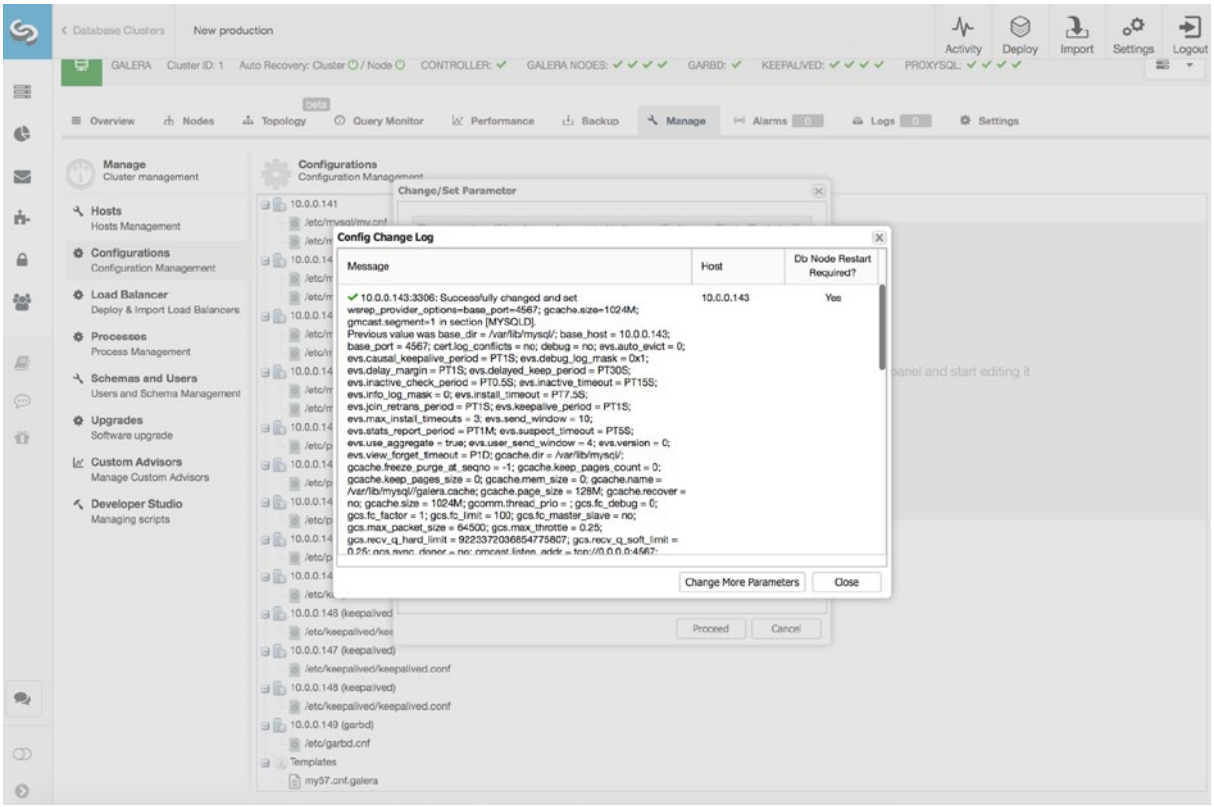
Next, if you deployed ProxySQL on separate instances, you may want to add Keepalived on top of it.

Then, finally, we will deploy a Galera arbitrator, garbd. It will be located in the third datacenter and it will help in case one of the main datacenters would become unavailable.

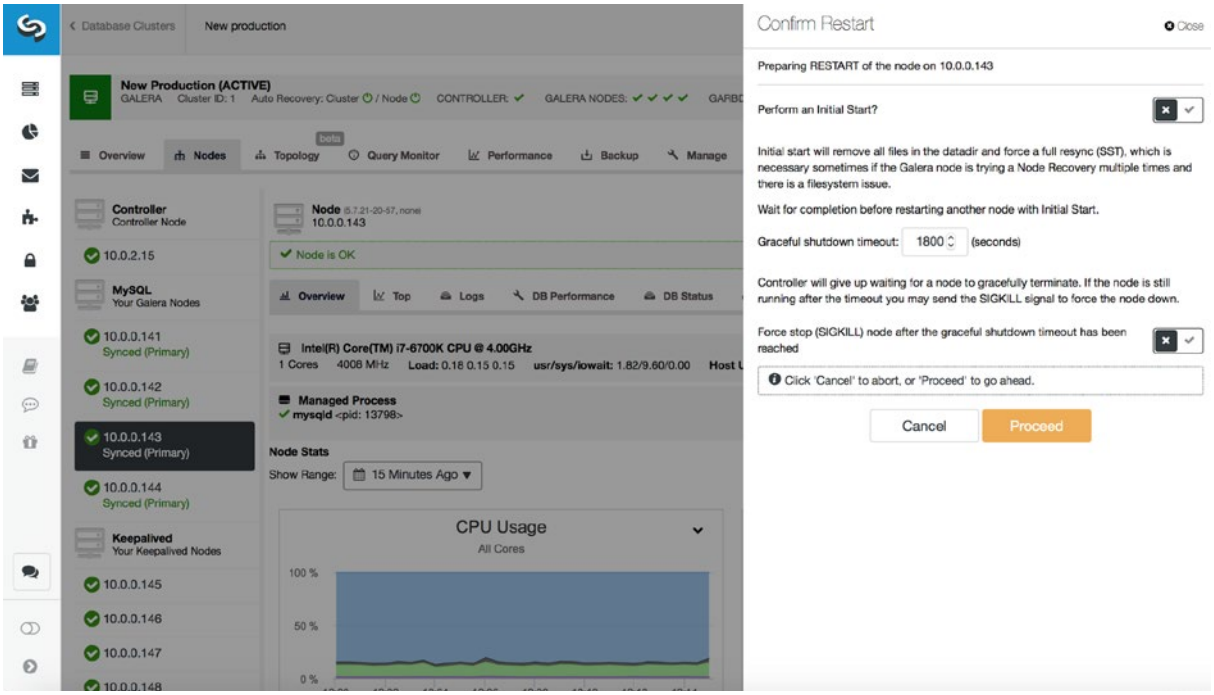


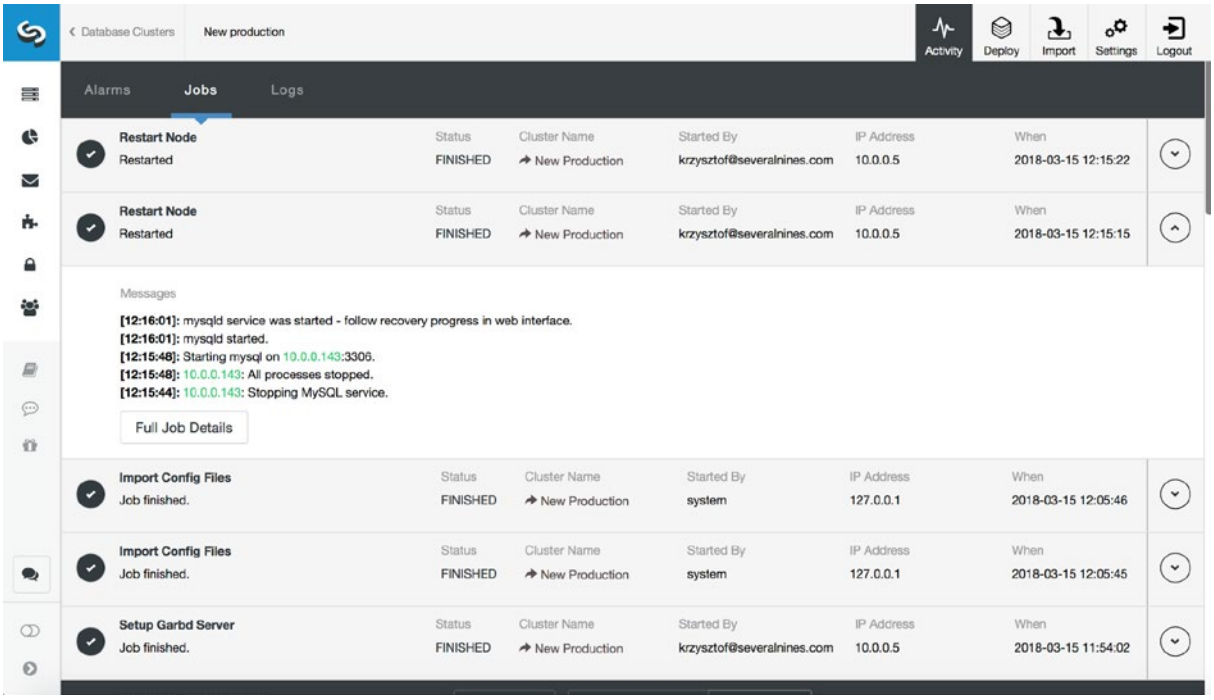
Finally, we want to make a change 'wsrep_provider_options' to set nodes in the second datacenter to another segment.



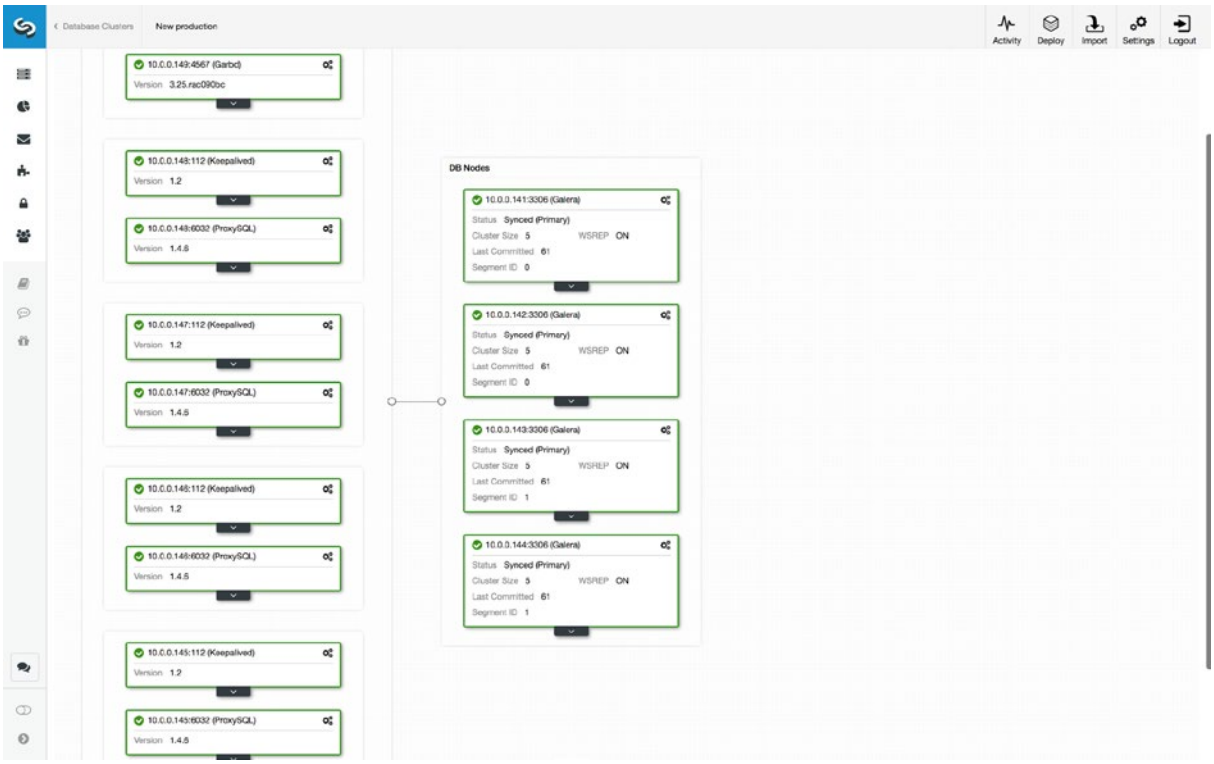


Once this is done, you will have to restart both affected Galera nodes to apply the changes.





Once this is done, we can finally see the whole topology matching our design:



3.6. Test your design

It's not enough to just plan your environment and then put it in production. You have to test it in order to understand if it delivers the availability level that you wanted. There are couple of methods you may want to use for testing.

For starters, manual tests. Kill some VM's and see if the remaining parts of your environment handled the situation or not. Generate some traffic (ideally, it will be the

traffic levels of your production, taken from slow or general log) and then see how everything behaves under load. Can your database tier handle failure of a single node? Can it handle (load-wise) failure of two nodes? How fast can you recover from backup? How does the load look like in the proxy tier?

You can also try to automate your tests by using tools like Chaos Monkey or similar. You can as well come up with your own solution. The idea behind such tools is to randomly disrupt your production, and test if the high availability mechanisms built into your environment are enough to handle failures. On small systems, like what we built in our example, such tools would be an overkill, but for larger deployments it may make sense to keep such tool running in the background, messing with the environment and testing your systems. Server crashes are unavoidable - on small setups you may ride your luck but for larger setups, statistics will inevitably come into play and nodes will fail.

If by any means your system didn't work as expected, you will have to redesign some parts of it. Once redesigned, test them again. Repeat until your setup matches your expectations and requirements related to the availability and SLA.

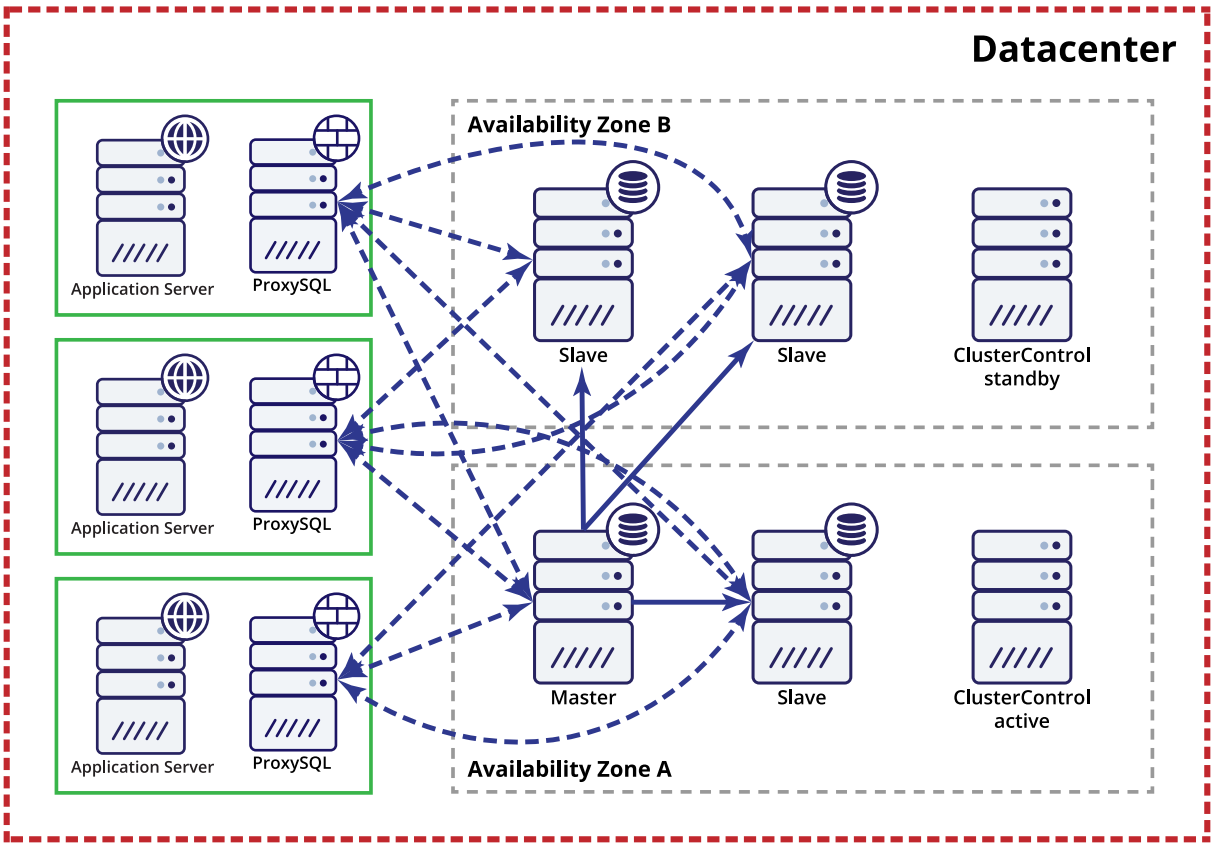
What is crucial to keep in mind - testing never ends. It's not that, once deployed on production, your new environment will be left alone. You should also test your production environment. In fact, Chaos Monkey messes with Netflix's production all the time (well, within business hours at least - to make sure it won't cause issues when the engineers are off). To test production, one can make use of the SLA. SLA, typically, is defined for some period of time. For a year, month, week, etc. If, at the end of that period, you still have some time left in your downtime pool, you can use that time to take some risks. Kill a proxy and see if Keepalived moves the virtual IP. Kill a database node and see if it can be recovered quickly. Maybe you have a new version of some script, which is intended to improve some aspects of the database maintenance - this is a great moment to test it.

Examples of the highly available setups

During the course of this whitepaper, we've shown you how to go about designing a highly available setup. In this section, we would like to give you some other examples of high availability setups. Please keep in mind what we discussed at the beginning of the previous chapter - high availability shouldn't be introduced just for the sake of it. It has to pay for itself. Some of the setups we will show you below are not as resilient as the one described already in this whitepaper, but they come at lower price tag. For example, why pay for the WAN links and traffic between multiple datacenters if you don't really need that level of redundancy?

4.1. Single datacenter, replication

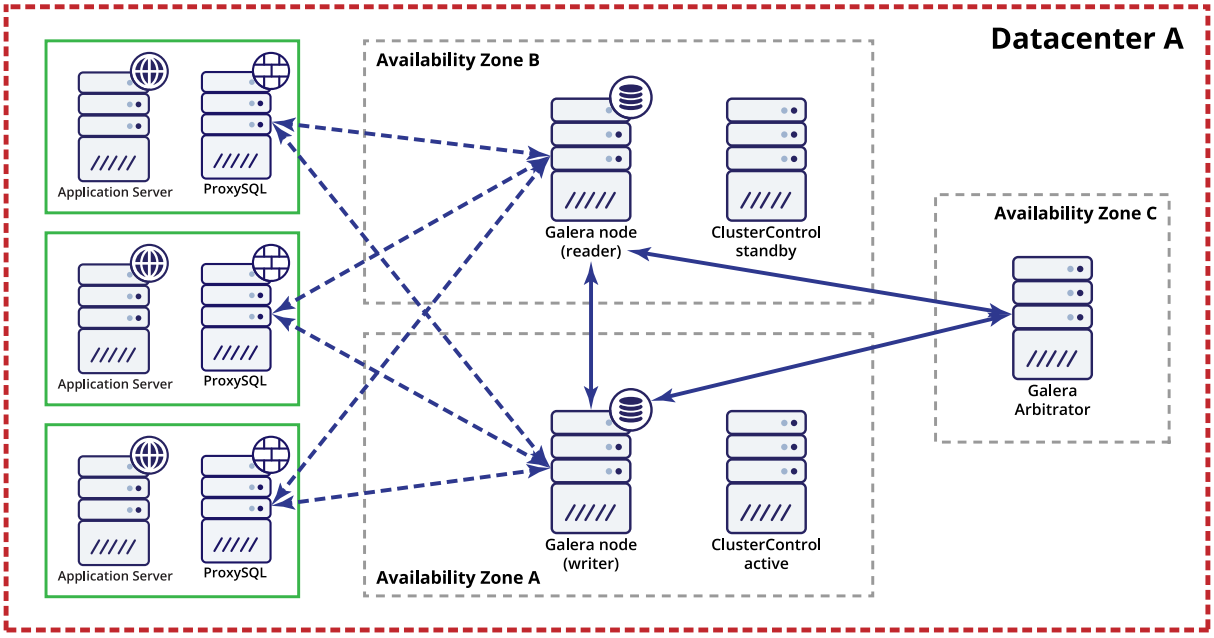
Here we assume that a single datacenter is enough and we don't need automated failover across multiple datacenters. You will need a proxy layer to route your traffic and react on topology changes. You will also need a tool which will take care of automated failover.



In this example we have a master and three slaves distributed across two availability zones - you can, of course, add as many slaves as you need for your load. Traffic is distributed by ProxySQL instances collocated on application hosts. Two ClusterControl servers in two availability zones, working in active - standby setup, take care of failover should the master crashe. What is important, though, is to make sure that the active ClusterControl is located in the same availability zone as the master. This is to avoid making the wrong decision to promote a slave to master, should the master be cut off the rest of the topology after a network partition. Another option would be to leverage a third availability zone and locate the ClusterControl instance there. ClusterControl will also make sure failed slaves are brought back to the topology, as long as it is possible. Of course, it can also be used for monitoring, scaling and managing your setup including executing topology changes, adding new slaves and so on.

4.2. Single datacenter, Galera cluster

In this example we will look at the minimalistic deployment of the Galera cluster within a single datacenter. Again, traffic is distributed by ProxySQL instances collocated on application hosts. We will also add two ClusterControl servers in active - standby for node and cluster recovery along with monitoring, scaling and management of your setup.

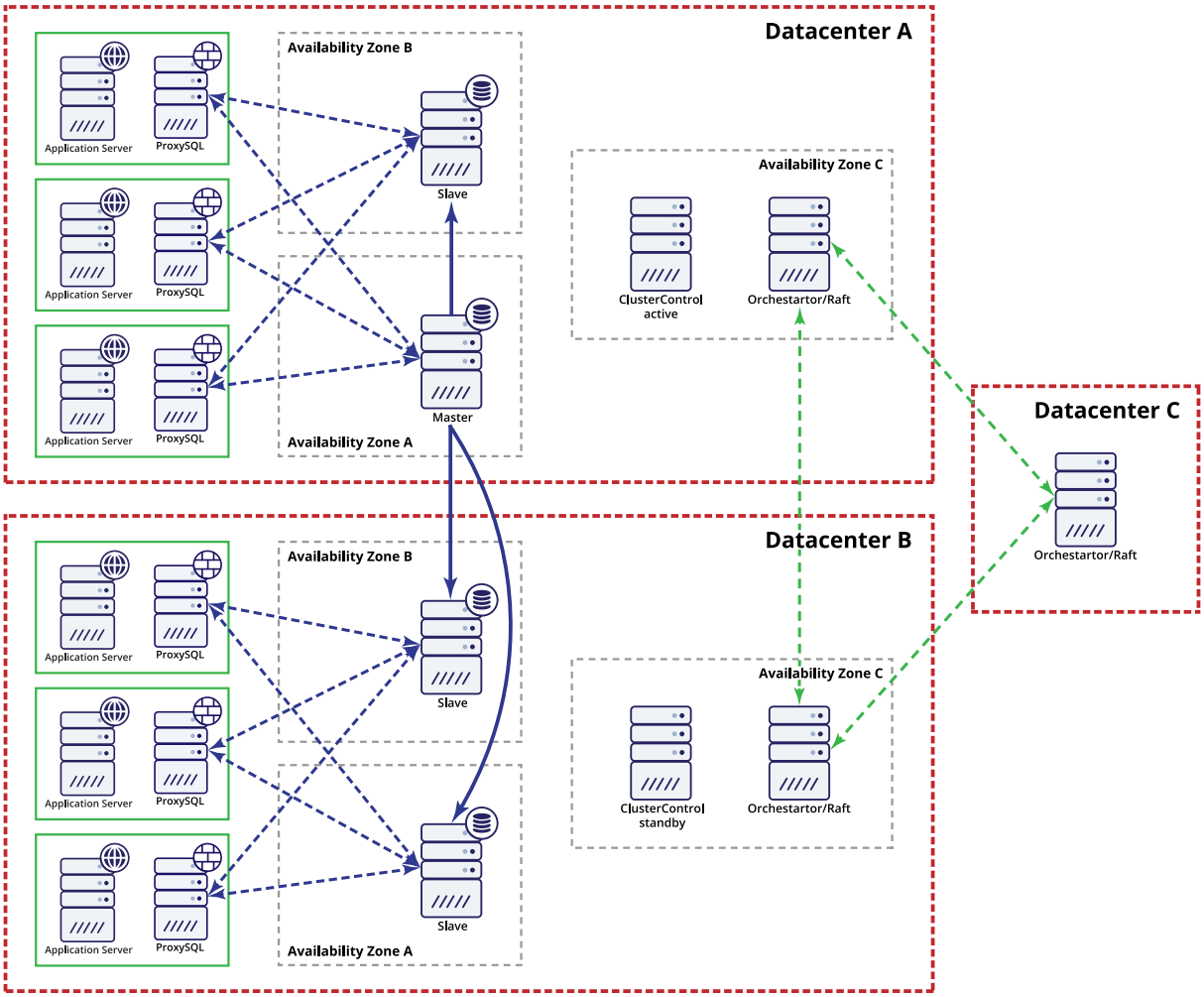


If you want to add more Galera nodes, this is perfectly fine - as long as you will have an odd number of nodes when counting with Galera Arbitrator (for example 4 nodes + garbd). The Galera nodes are distributed evenly across availability zones.

4.3. Multiple datacenter, replication

In this setup we show a replication setup spanning across multiple datacenters. Main issue with replication is that there is no quorum mechanism to detect a datacenter failure and promote a new master. One of the solutions here is Orchestrator/Raft.

Orchestrator is a well-known topology manager which can handle failovers. When used along with Raft, Orchestrator will become quorum-aware. One of the Orchestrator instances is elected as a leader and executes recovery tasks. Other instances do not perform any action other than monitoring of the topology. In case of network partitioning, the partitioned Orchestrator instances won't perform any action. On the other hand, the part of the Orchestrator cluster which has the quorum will elect a new master and make the necessary topology changes. ClusterControl is used for management, scaling and, what's most important, node recovery - failovers would be handled by Orchestrator but if a slave would crash, ClusterControl will make sure it will be recovered. Orchestrator and ClusterControl are located in the same availability zone, separated from the MySQL nodes, to make sure their activity won't be affected by network splits between availability zones in the datacenter.



About ClusterControl

ClusterControl is the all-inclusive open source database management system for users with mixed environments that removes the need for multiple management tools. ClusterControl provides advanced deployment, management, monitoring, and scaling functionality to get your MySQL, MongoDB, and PostgreSQL databases up-and- running using proven methodologies that you can depend on to work. At the core of ClusterControl is it's automation functionality that let's you automate many of the database tasks you have to perform regularly like deploying new databases, adding and scaling new nodes, running backups and upgrades, and more. Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

About Severalnines

Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

Severalnines' products are used by developers and administrators of all skills levels to provide the full 'deploy, manage, monitor, scale' database cycle, thus freeing them from the complexity and learning curves that are typically associated with highly available database clusters. Severalnines is often called the "anti-startup" as it is entirely self-funded by its founders. The company has enabled over 12,000 deployments to date via its popular product ClusterControl. Currently counting BT, Orange, Cisco, CNRS, Technicolor, AVG, Ping Identity and Paytrail as customers. Severalnines is a private company headquartered in Stockholm, Sweden with o ces in Singapore, Japan and the United States. To see who is using Severalnines today visit:

https://www.severalnines.com/company

Deploy

Manage

Monitor

Scale

Related Resources

MySQL Replication for High Availability

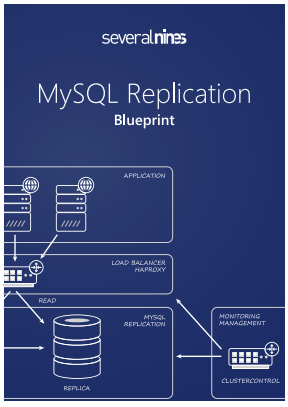
This tutorial covers information about MySQL Replication, with information about the latest features introduced in 5.6 and 5.7. There is also a more hands-on, practical section on how to quickly deploy and manage a replication setup using ClusterControl.

Download here

DIY Cloud Database on Amazon Web Services: Best Practices

Over the course of this paper, we cover the details of AWS infrastructure deployment, considerations for deploying your database server(s) in the cloud, and finish with an example overview of how to automate the deployment and management of a MongoDB cluster using ClusterControl.

Download here



MySQL Replication Blueprint

The MySQL Replication Blueprint whitepaper includes all aspects of a Replication topology with the ins and outs of deployment, setting up replication, monitoring, upgrades, performing backups and managing high availability using proxies.

Download here

Database Load Balancing for MySQL and MariaDB with ProxySQL - Tutorial

ProxySQL is a lightweight yet complex protocol-aware proxy that sits between the MySQL clients and servers. It is a gate, which basically separates clients from databases, and is therefore an entry point used to access all the database servers.

Read the tutorial

severalnines

These days high availability is a must for any serious deployment. Long gone are days when you could schedule a downtime of your database for several hours to perform a maintenance. Making a database environment highly available is one of the highest priorities nowadays alongside data integrity. For a database, which is often considered the single source of truth, compromised data integrity can have catastrophic consequences. This whitepaper discusses the requirements for high availability in database setups, and how to design the system from the ground up for continuous data integrity.

Deploy

Manage

Monitor

Scale