GitLab: Highly Available Architecture

Ryan Bertsche

 Thursday, June 29, 2017

Table of Contents

Abstract

The purpose of this white paper is to define a highly available architecture for GitLab on the Linux® on Z and IBM® LinuxONE™ platform. GitLab is an open source git application with a web frontend, that utilizes multiple open source technologies. The scope of this paper includes the architectural design requirements of creating a highly available instance of GitLab and all of it open source components, including: Postgres, Redis, Redis Sentinel, ETCD, HAProxy, GPFS™ and KeepaliveD. This paper goes beyond designing a single High Availability application, exploring the way a set of applications can each be made highly available in one environment, to form a highly available application stack. The entire infrastructure will be configured to be highly available for maximum uptime, rolling updates and system maintenance. Overall, this white paper will demonstrate cloud-like processes, tools and deployments for the Linux on Z platform.

Introduction

GitLab

The processes by which software applications and solutions are developed and deployed can vary greatly. Recently, there has been a transition across many industries to an Agile development model, which involves quick, iterative cycles of development. This process is often combined with the dev-ops process of deploying code. This modern development and deployment stack is designed as a cyclical feedback loop, where developers make frequent, small updates, that are automatically tested, deployed, and then the developer receives feedback on the results. This process involves lots of automation, and coordination of different tools and teams to build a cohesive workflow.

GitLab serves as the central hub for this modern development and deployment infrastructure. At its core, GitLab is a web interface for a git source control repository. Git source control serves as a central repository for source code, where all updates to the code are tracked. This source code is the basis for development environment, and the entire process of software development revolves around changes to source code. This style of development is most evident with open source software, and the use of GitHub. GitHub is a cloud based source control website that is nearly ubiquitous in the open source development world. There are a whole array of tools and workflows based around the GitHub model for open source application development, and it is used with great success. However, many enterprise customers, whose code is not open sourced and is a crucial part of their intellectual property and competitive advantage would not be comfortable with source code existing in the cloud. In fact, many would prefer that the source code remain completely on site and managed in house. GitLab is the solution to that problem. Being an open source product, it can be installed on internal, company owned infrastructure, and be completely managed and exist exclusively behind the company firewall. GitLab helps bridge the gap between modern cloud services and their workflows, and traditional enterprise infrastructure and security.

GitLab includes additional tools and communication features and “plugins” to the git source control interface. There are numerous plugins that can be configured to interact with GitLab in an enterprise environment to accomplish tasks like:

- Continuous deployment / continuous integration
- Bug Issue tracking
- SCRUM Board
- Test automation
- etc

There are even custom web hooks that can be used to integrate the GitLab workflow with almost any existing tool.

GitLab will act as the “Central Hub” in the application development, testing and deployment process in an Agile/DevOps environment. Consequently, the uptime of the GitLab stack is crucial.

Advantages of High Availability (HA)

Uptime and availability is important for any mission/business critical applications. If the application is not up and working, it is not doing its job, and likely disrupting business, both directly and indirectly.

For example, if the Point of Sale(POS) backend goes down in a retail store for 20 minutes, there are 20 minutes of sales that are lost. At scale, that can be a substantial amount of directly lost revenue for a company. Additionally, if a music streaming service has a partial outage that causes its songs to load very slowly, customers can be driven to a competitor’s service. This has immediate consequences of loss of customers, and can tarnish the company’s reputation in the long term. Web based services and applications can suffer greatly if consumers do not have a consistent and snappy experience. The same principals apply to non-customer-facing services and applications used internally by companies. If the internal structures and processes of a company become disrupted, they can create a shockwave of problems.

There are multiple techniques to alleviate the issues above. One of which is to use supported and tested enterprise grade equipment. Certain hardware and software stacks are built and tested to have a higher average uptime. Linux on Z and LinuxONE machines are particularly well known for their uptime and reliability.

Even with the benefits of more reliable hardware, it can still be beneficial use additional techniques to bolster the reliability of the systems. Mainly, clustered or distributed computing provides some distinct advantages in keeping an application highly available. This strategy consists of running multiple instances of the same application, where each instance of the application is coordinated in some way. The applications can operate in an active/backup distribution, where there is a replicated instance ready to come online if the active goes down. There is also a distributed style application, were data multiple instances of an application exist,

and work is distributed amongst them (most common in stateless applications). This is the type of replication that is common in cloud applications. In those cases, individual server uptime tends to be shaky, and a bit unreliable, so having many distributed instances of an application is the only way to keep it alive. In addition, this technique is sometimes used to scale an application when resource consumption exceeds the capacity of a single instance. This distributed model provides some additional advantages that have been commonly incorporated into the development lifecycle. Most notably, having multiple instances of an application allows for zero-downtime maintenance and updates of systems and applications. This not only increases uptime, but also decreasing time to upgrade, and reduces, possibly eliminating the need for scheduled downtime. This is a non-exhaustive analysis of the benefits of a highly available system, but it points to some of the key advantages.

Using the Linux on Z platform gives the unique position of using a hybrid of HA techniques to get the best of both worlds. For one, Linux on Z provides security advantages over cloud deployments, by the simple nature of being entirely behind company firewalls and other corporate protections. Secondly, there is an inherent management and network overhead to the distribution of applications. At scale, the more instances of an application there are, the less efficient each one is. For this reason, it is preferable to scale an instance vertically for performance capacity. In addition, the increases average uptime of each instance that is generally experienced on Linux on Z means that there are less instances needed overall to maintain HA status. Still, the advantages of distributing the application across failure zones will be taken advantage of, with less overhead cost(waste).

Designing High Availability (HA)

Overall Architecture

A key aspect of high availability is application clustering. To prevent an application from becoming unavailable, multiple instance of the application are created. These applications are logically grouped together, and act to any outside application as a single application instance. Depending on the requirements of the application being clustered, there are different implementations for joining multiple applications instances into a united cluster. There are three main types of application clusters that are used in this implementation of GitLab: Master/Slave, Distributed, and Stateless.

Master/Slave

The Master/Slave model consists of one primary application instance (Master) that is handling all the application load, and multiple backup or standby (Slave) application instances. As long as the Master instance is available, the Slave instances never accept connections or do any real application work. Instead, this model dictates that the Slave instance stays in sync with any changes in the data or configuration of the master server. If the master instance goes offline, or fails for any reason, one of the slaves will be promoted to master, and the application will stay available.

This style of clustering is popular among traditional databases, or other traditional workloads where data consistency is paramount. Since only 1 instance of the application cluster is ever active, there is a significantly smaller chance of getting inconsistent results from the application. The importance of this is especially apparent with databases, where writing and reading from the same data structure, on different running application instance simultaneously would lead to inconsistent results.

Logically, the master slave model is simple and easy to deal with. It can scale linearly from 1 node, with each additional node adding another zone of availability (As in 2 is twice as much availability as 1). Since only 1 instance is active at a time, a cluster of any size will not experience performance degradation from any number of application instances failing. In fact, a master/slave cluster of 5, 10, or 20 members would stay available if all but 1 node failed. As long as there is a single active node to be master, the application will work. For the most part, performance is not affected by the number of active nodes, because only 1 node is ever active. the relative simplicity of this application cluster design does have drawbacks.

First, the master/slave model does not scale outwardly. It can be debated whether this is a feature or a flaw, but either way, adding more instances to the cluster does not help performance in any way. If there is a cluster of *n* number of application instances, the will be *n-1* idle instances. Allocating resources for backups that sit mostly idle is not very efficient, especially when each slave instance of the application must be robust enough to take over the entire application workload of the master instance. Besides the scaling issue, there is an inherent issue in the way master/slave clusters failover. Master-slave relationships are always one master to many slaves. If the master becomes unavailable, 1 of the slaves is promoted to master. There are a few different ways for slaves to decide which instance is promoted, usually a set priority, or some other arbitrary test. This can lead to a serious problem, known as split brain. Split brain occurs when the cluster becomes partitioned into more than 1 distinct section, usually by a partial network failure. If a slave sees that the master is no longer reachable, and it is the highest priority slave, it will promote itself to master. At the same time, there may exist another partial cluster of the application that has its own master instance. So, in a single application cluster, there are two or more active masters, each with their own application and data state. This can completely corrupt an entire dataset, and almost certainly would require downtime and a rollback of data to fix.

Overall, the master slave model proves useful for maximum availability when dealing with traditional workloads and data crucial applications. It has the optimal failover rate, where a three-node cluster can lose two nodes, and still run without a hitch. This is especially useful for large applications, like relational databases, where horizontal scaling is virtually impossible and carrying the instance overhead of preventing split-brain (more about that next) is wasteful.

This master/slave architecture being used for the GitLab use case is not a self-managed model. Generally speaking, members for these master/slave clusters are unaware of the state of the other cluster members, and don't have the ability failover automatically. It is possible to utilize master/slave principles in the distributed model, but for the purposes of this use case, this is not the case. Any self-managed implementation will be considered part of the distributed model.

Distributed

There are a few different types of distributed clusters that can be split into multiple subcategories. For the purposes of this GitLab implementation, a distributed cluster will be classified as an active/active cluster, where every node in the cluster is always up and actively participating in the application logic. This also means that distributed cluster will be self-managed. This implies that each member of the cluster is communicating with and aware of the status of each of the other members. This allows for a horizontal scaling of the application cluster. Adding nodes not only adds more availability, it increases the performance capacity of the application. In this model, there still needs to be a single consistent state for the cluster. Therefore, the horizontal scaling cannot cause inconsistencies in the application.

There are two common ways the horizontal scaling is implemented. One is simply replicating the data across all the nodes in the cluster. This usually requires some specialized application logic to ensure that any updates by one node are properly distributed to each other instance in the cluster. This method of scaling is a direct evolution of the master slave model.

There is usually some tradeoff of speed vs consistency in this model. Either the application acknowledges quickly, and is eventually consistent, or the application waits for synchronization conformation from the other nodes, at the expense of speed. This type of distribution is particularly beneficial in situations where the state of the application is more heavily read than written. Read heavy workloads can spread their load across multiple cluster member quickly, without having to deal with much data inconsistency.

For applications that deal with more write-heavy workloads, or deal with carry large amounts of persistent data, another form of distributing the application is splitting (or sharding) the data across multiple instances. As suggested by the name, this process involves splitting the data across multiple application instances, where no single instance contains a full view of the data. By having data split across cluster members, the load of writes is spread across the cluster, without having to take the time to replicate the data to all other cluster members. Additionally, sharding scales to accommodate large amounts of data, because each application instances only have to hold a proportional fraction of the overall data. Even with the enterprise grade storage and servers that are used in Linux on Z environments, this reduces storage and memory constraints of individual nodes, and speeds up application performance. Splitting the data requires even more application management overhead, as the application must track where each piece of the data is stored, rebalance shards to keep data spread equally, and optimize data location to deal with access hotspots, etc. However, this doesn't really satisfy the demands of high availability, because a single application instance becoming unavailable would, at the very least, cause downtime for the section of data that instance contained. Therefore, this form for application distribution must also be paired with replication to be considered highly available.

Both distributed replication and sharding come with their fair share of application management overhead. The active/active model requires cluster members to use their resources to run cluster management logic. Each cluster member will use some of its processor time and other resources managing the cluster, instead of performing its core application function. In terms of pure application efficiency, two clustered application instances would not provide double the performance of a single, non-clustered version. Depending on the efficiency

of the clustering design used by an application, this overhead can become significantly costly with very large cluster.

Despite this clustering overhead, the ability of a cluster to manage itself provides many benefits. For example, cluster instances are more acutely aware of the state of the entire cluster, which allows them to maintain cluster health. This works to solve the problem of split brain that can plague the unmanaged master/slave application clustering model. To solve split brain, distributed systems implement a rule of *n/2 + 1* where n is the number of initial application instances. This rule requires that more than half of the original cluster members must be active and in communication with each other for the application cluster to be available. This prevents the clustered application from splitting into multiple distinct instances. However, it also means that with a cluster of 99 nodes, 50 nodes need to be active and communicating for the clustered application to stay available. Consequently, there needs to be twice as many nodes in a distributed cluster to maintain the same amount of availability as a master/slave cluster.

Overall, this method of creating application availability adds overhead, but provides necessary features that a simple master/slave cannot provide. In the case of this GitLab implementation, applications utilizing the managed distributed model will act as lightweight management for the master/slave configurations. This hybrid methodology is quite common, as it allows database applications that work more efficiently as master/slave clusters to store their own management state in a self-managed distributed cluster. This allows the simpler master/slave cluster to offload its own management to a more lightweight and management cluster. It also allows master/slave clusters, which tend to be larger instances, avoid the *n/2 + 1* requirement, instead offloading that to the distributed management cluster.

Stateless Clustering

An application that holds no persistent data, and can be created and destroyed with no loss of information is known as a stateless application. In terms of High Availability and clustering, stateless applications are the ideal, easiest, and overall best applications to work with. All the management and replication concerns of the other distribution models become moot when the application holds no state. There needs to be no consistency between different instances of the application, so instances can be added and removed arbitrarily. This is advantageous for both availability of the application and horizontal scaling.

In terms the availability, new instances can be placed in as many different availability zones are possible. If any number of instances go down, whether for maintenance, application updates, unexpected downtime, etc., new instances can be brought up to offset the loss. Application instances do not have to know about each other, so the application would still be available with only instance without any worry of split-brain. Stateless applications also tend to be very lightweight in terms of resource requirements, so increasing availability is relatively cheap. There is no management overhead, inter-instance communication or any other mitigating factor that would prevent starting up more stateless applications to increase availability. So, we get linear scaling with virtually no downside.

The benefits of stateless applications in terms of availability also apply to horizontal scaling. If the load on the overall application cluster increases, new instances can be brought on-line and the application load will be distributed to the new node. Being stateless, these new

nodes can be brought up quickly in the area with the most resources to spare. This greatly improves the overall availability of the system, as heavy application load can lead to overall application downtime. Quickly offsetting the load with small instance reduces stress on the application and keeps it available.

The new methodology in application management is cattle over pets. Stateless applications that are created and destroyed without much care are cattle. Like cattle, as with stateless applications, there are many of them, they are nameless, and if one of dies, it is replaced succinctly. In the case if pets, each one is individually loved and cared for. This is how enterprise applications have traditionally been treated. Each instance of an application tends to be large, finely tuned, and the shutdown would be painful.

in the move from pets to cattle, there are many cases where the transition isn't perfectly smooth, due to legacy code, or incompatible application design. One example of this is traditional relational databases. By nature, these databases are stateful and not good at being redundant. There are new database designs that more natively build around the distributed model, but they don't have the pedigree and codebase of a relational database. The same principal applies to many legacy workloads, where the time, effort, cost, disruption of rewriting the entire application as "cattle" would be astronomical. For these workloads, the legacy applications can be separated into multiple services, allowing a good portion of the application "logic" code can be run in a stateless mode.

Availability Zones

When designing a highly available application, instance failure should be expected. The purpose of taking the time to make an application highly available is to create enough redundancy to preempt a total application outage when failures do occur. Having multiple instances also allows maintainers and developers of applications to provide maintenance and apply upgrades without experiencing downtime.

These benefits are not intrinsic to having multiple instances of an application running. If there are 100 instances of an application running on a laptop, and the laptop is turned off, the application will be unavailable. The individual instances of the application need to be designed to run in environments that are unlikely to experience downtime at the same time. Compute resources must be logically mapped to groups that are likely to fail together, known as availability zones.

The granularity of the availability zones will vary greatly based on the scale, nature and intended audience of the application. For a heavily trafficked web service with worldwide public access, you would expect to have geographically dispersed availability zones. This prevents natural disasters, large scale power outages and other major events in one area from taking down the application worldwide. On a smaller scale, availability zones can be divided by physical datacenters, to prevent downtime from fires, flooding, network outages or other events.

GitLab High Availability Architecture

The general purpose of GitLab is to act as a source code repository and version management system for application development. Through a series of included tools, such as issue tracking, user access control, and scrum planning, as well as configurable add-ons and hooks, such as Jenkins, JIRA, test automation, deployment automation, infrastructure automation among others, GitLab serves as a central hub for the complete application lifecycle, including development, test, deployment, operations, and management.

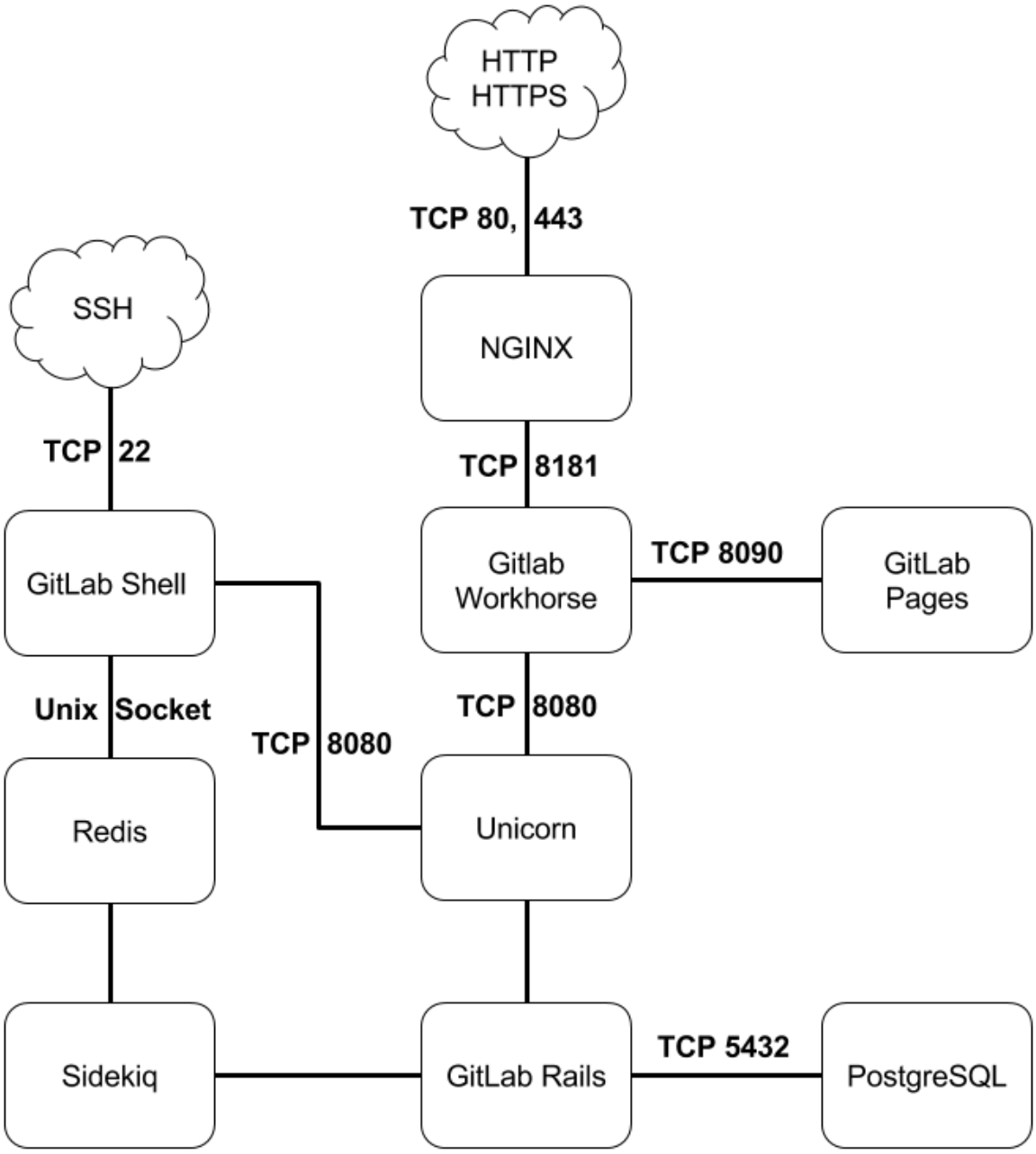
GitLab is an open source application that is developed as an open-source competitor to the immensely popular GitHub service. The GitLab company has a public deployment of GitLab that is hosted in the cloud and free to use. Additionally, the source code for the entire project is available for download and manual installation. There is also an RPM and deb package that acts as an all in one installer for the entire application stack. Recently, they even added a Docker container that is ready-made to run GitLab. It is possible to use these installations, and run GitLab on a single server with almost no special configuration necessary. A minimally configured GitLab instance can easily be brought up and running in less than 10 minutes. This is great for testing and temporary deployments, but building an enterprise ready, highly available deployment of GitLab requires looking at the individual components of the application stack and configuring each as a highly available application.

GitLab Architecture Overview

GitLab is not one monolithic application. It was designed as a series of components working together to form a working application stack. To make GitLab, or any other application, highly available, it is important to become familiar with the application architecture. Knowing each application component and how each interacts makes it easier to design a stable HA system.

Breaking GitLab down into its core components and splitting each into its own Highly Available cluster allows for a much greater level of uptime and availability. One of the most important aspects of making GitLab highly available is separating the stateless and stateful components of the application. Along the same lines, separating each stateful component into its own application cluster is also important. There are certain aspects of almost any software stack that must be persistent. For example, most applications have a database to store user data, and a filesystem to store files and other information. It is not architecturally possible to make most applications completely stateless without stripping out most of the application functionality and purpose. However, it is possible in the design and implantation of an application to separate the stateful components from the stateless components. For example, keep the database separate from the application code, and access the data through networked database access calls.

Naturally, this adds some complexity to the configuration, as there are about 7 different application clusters to manage instead of one. To ensure the availability of the GitLab application, each application must be designed and configured in a highly available manner.



https://gitlab.com/gitlab-org/gitlab-ce/blob/8-11-stable/doc/development/architecture.md

For this Highly Available deployment we will divide the GitLab application into 4 major components:

1. GitLab frontend application (Rails Web application_
2. PostgreSQL
3. Redis
4. Distributed File system

GitLab Frontend

At its core, GitLab is a Ruby on Rails web application. This web application also manages git repositories, and some comment and bug requests to supplement the git projects. All of the logic and "brains" of the GitLab application are a part of this Ruby on Rails application. In addition to the we frontend, there are a few other components that will be grouped together as part of the GitLab frontend. First, NGINX is a web server that is used to server the Rails web content. he frontend component also consists of the GitLab shell and GitLab Workhorse, which handle the git commands directed to the web application. For example, any command line git commands, like clone, fork, push and/or pull would be directed to one of these components to execute the git commands. Additionally, there is the Sidekiq and Unicorn, which help manage background Ruby processing and http server requests specifically. The common thread with all of these application components is that they are essentially stateless. These components hold all the logic, and manage the execution of tasks, but do not persist data. This is crucial, because there will be multiple instances of the frontend, and they must not present an inconsistent view of the overall GitLab application. Throughout this whitepaper, this Ruby on Rails web application, as well as all of the other Ruby code and configuration files, and stateless components will together be referred to as the "GitLab Frontend".
This is served through a These components are kept together because they are all stateless. There are a few other minor components of the frontend, but they are all stateless too

PostgreSQL

Postgres is the database component of GitLab, where long-term data is stored. This data includes usernames and passwords, repository permissions, user settings, comments, issues, and just about any other data that is not directly controlled by git. PostgreSQL is an open source relational database. PostgreSQL must be available and consistent to each instance of the GitLab frontend application. All the GitLab frontends must connect to the same instances of Postgres.

Redis

Redis is an in-memory key-value store database. It is an open source database, that is generally used for fast access and short-term storage. In the GitLab stack, Redis serves as a consistent session cache and job queue. This means that user session information, and tasks that are being performed are placed in the Redis instance. Redis is a way that static components in GitLab Frontend persist state and talk to other components, so this must be consistent and shared for all instances of GitLab

Distributed File System

At its core git is a source control and versioning management software for source code. The git protocol works by saving files onto a file system, so naturally all instances of GitLab need to

access the same filesystem. There are also a few other shared folders outside of git, including a folder for SSH keys, uploaded assets (pictures) and other minor files that must be consistent across GitLab instances. The exact folders that will be shared is covered later in the paper.

Additional Components

In addition to the 4 main GitLab components, there are some other applications that will help to manage the application cluster. These include:

- HAProxy
- Keepalived
- Redis Sentinel
- Patroni
- Etcd

HAProxy

HAProxy is an open source web proxy and load balancer that we will use for multiple purposes in this configuration. It can act as a gateway to a service by distributing requests to multiple hosts on other systems. It can also perform health-checks of varying complexity to present only active cluster members to the requesting service.

Keepalived

Keepalived is a Linux application that is used to keep an application highly available. For the purposes of this GitLab deployment HAProxy is being used to provide access to the highly available services. However, the HAProxy application also needs to be redundant. Keepalived satisfies this need by running on 2 identical HAProxy nodes, and making sure 1 of the 2 nodes is always configured with a floating IP address. All other services look for HAProxy at this floating IP address, effectively making HAProxy Highly Available.

Redis Sentinel

Redis Sentinel acts as a management cluster for a group of Redis instances. Standard Redis only has support for basic master/slave replication, which is not robust enough for enterprise deployments. Redis Sentinel is a management cluster that manages the Redis cluster, and determines the master and handles the failover of Redis servers. We will talk to Redis Sentinel when we want to know which Redis server is master.

Patroni

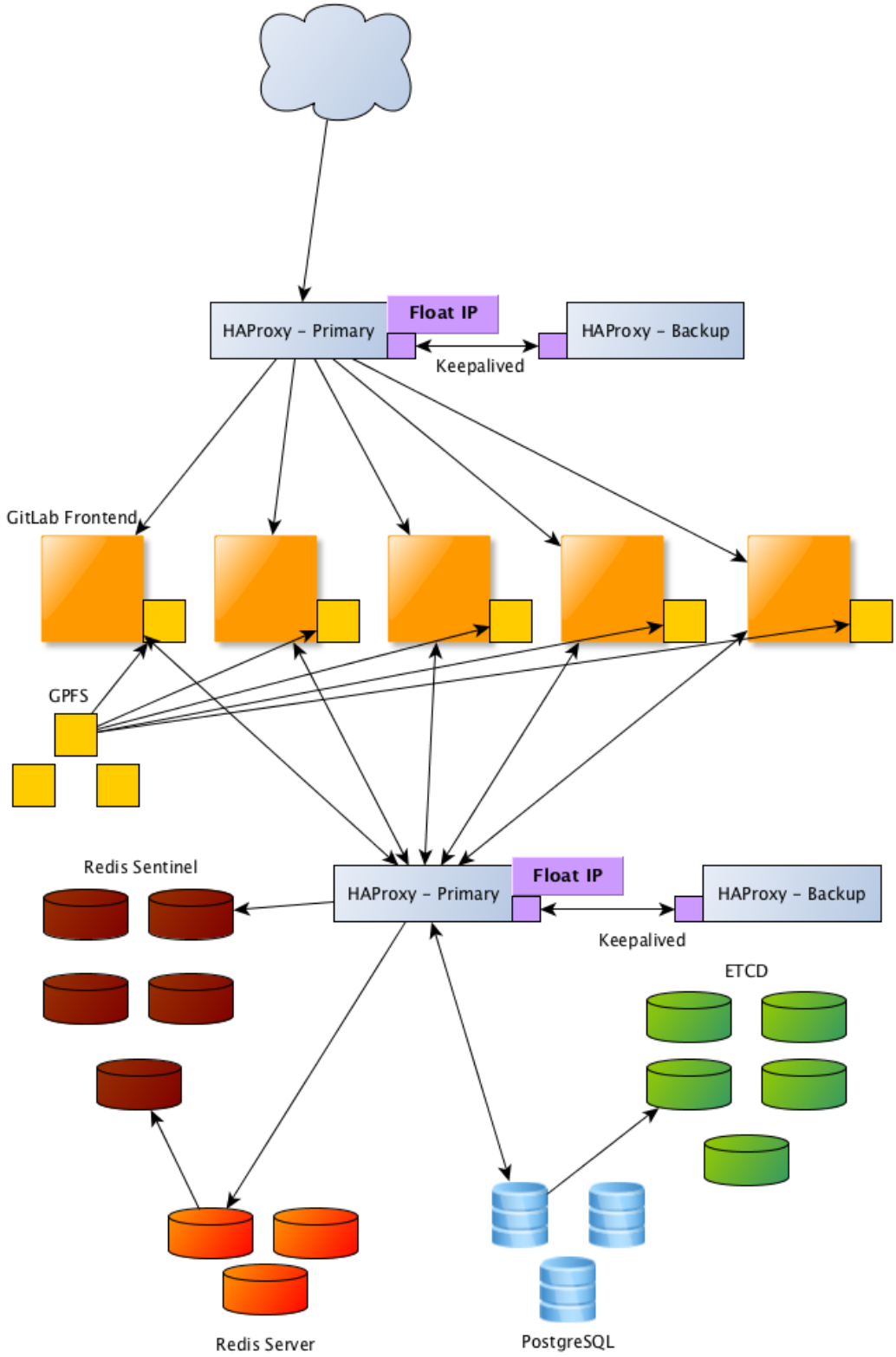
Patroni is an open source framework written in python that is used to automate the replication and failover of PostgreSQL server instances. Similar to Redis, Postgres instances only have built in support for basic master-slave replication. This does not include configuration for adding or removing nodes, or automatic failover. Without Patroni, a failed node would have to be manually failed over, and manual IP address configuration would have to occur. This would fail to satisfy the availability that GitLab needs for this implementation. Patroni runs on the same nodes as the PostgreSQL instances.

ETCD

ETCD is a distributed configuration manager. This open source tool is used by Patroni to maintain a consistent state across the PostgreSQL cluster. By saving the configuration for all the PostgreSQL servers in Etcd, all the Patroni instances are always in a correct and consistent state. It also prevents more than 1 server from becoming master, and gives a singular location to query on information about the Patroni (PostgreSQL cluster).

GitLab HA Implementation

Now that the different components of GitLab have been identified and the different methods of creating highly available clusters have been introduced, the two concepts will be applied. A fully HA GitLab architecture will be defined. Each component of GitLab will be broken out into a highly available cluster, to create a greater GitLab HA cluster.



PostgreSQL

Postgres is the relational database backend used by GitLab. Postgres is important in storing information about our users, their access rights and repository information, all the project issues, comments, SCRUM boards, etc. The actual git projects are stored on the file system, but virtually every other piece of information that is in GitLab is stored in Postgres. Historically, Postgres was not built to be distributed or highly available. In the more recent 9.X releases, many useful features have been added to help build and maintain replicas of Postgres data for backups and failovers. The default GitLab configuration has even started making progress in supporting high availability Postgres out of the box. However, automatic replication and failover is not currently available out of the box for either GitLab or vanilla Postgres. Being a full featured relational database, the configuration of a single, stand-alone Postgres can be an involved task. Manually configuring and maintaining a fully replicated Postgres cluster would be a considerable task, even for Postgres veterans. Luckily, there are a few things that can be done to make this significantly easier.

Firstly, by running on Linux on Z systems, Postgres can be vertically scaled more than a traditional x86 server. This can greatly reduce the number of Postgres instances needed to meet the user demand, when compared to a horizontal scaling situation. This process is not explicitly described in this tutorial, but would simply involve assigning more resources to the virtual instance running Postgres. However, this only reduces the amount of Postgres instance we need to stay highly available, not completely reduce it.

To manage the multi-tenanted Postgres Cluster, a Postgres configuration framework named ‘Patroni’ will be used. Patroni is an open source Postgres management software that sits on the same server instances as Postgres and manages their configurations, backups, replications, and restorations. It does this using REST API calls to a key-value database, to coordinate Postgres state, status and availability with the other Postgres instances. In this case, Etcd will be used as the key-value store, but Patroni currently supports Consul, Zookeeper, and Exhibitor as key-value store alternatives to Etcd. Each Patroni instance queries the Etcd cluster for current state of the Postgres cluster, and updates information about the specific node it is governing.

Etcd is a key-value store database that is distributed by design, which eliminates it as a single point of failure. It is a lightweight, API enabled cluster to store metadata for Postgres that will be highly available itself.

At its very core, Patroni is a set of python scripts that configures and manages an instance of Postgres, and reports its status to the Etcd cluster. The Etcd cluster has a configurable time to live for all data, so each Patroni instance must continually check into the Etcd instance to stay connected. Patroni also reads the Etcd data to know what other Postgres instances are in the cluster, and which instance is the leader. The Patroni instances will then automatically configure the Postgres instance they manage to be either a master or a replica. If it is designated as a replica Patroni makes sure it doesn’t fall behind the master. If something goes wrong and one instance fails, the time-to-live will expire in Etcd and the next time Patroni reads the Etcd file, it will update the cluster state accordingly. Postgres has multiple different ways of configuring replications, but generally there is one master node, and multiple slaves/replicas. Those salves would normally have to be manually promoted to master in the

event of a master meltdown. With Patroni, that failover would happen automatically in a time, and the cluster would keep living. In addition, if a new instance of Patroni tries to join the cluster, and it is behind the master, it will automatically rerun the logs and restore itself to current state. Postgres is entirely configured and controlled via the Patroni application, so there are many more advanced features and configurations in Patroni that are available.

The last component of the PostgreSQL cluster is HAProxy. HAProxy is a load balancer that will forward requests made to a specified port on the HAProxy application to a group of external IPs, based on rules defined in the HAProxy application. In this case, a single port on the HAProxy server acts as the access point to Postgres. This is the address that GitLab will be configured to look for to reach Postgres. HAProxy has a 'health check' tool that queries each of the Postgres instances, and determines if it is a master based on the response.

The high availability configuration of Postgres is a fusion of managed distributed applications and master/slave active backup applications. Postgres is a master slave application, where there is only ever one master, and all the other instances are configured to stay up to date, and ready to act as master if ever needed. The Patroni application that is co-installed with Postgres adds the extra logic needed for Postgres to handle automatic failover and high availability that native Postgres cannot currently handle. Since Etcd manages the state of the Postgres cluster, the high availability design does not have to account for split-brain. Therefore, there only needs to be 3 instances to reach the availability requirements for GitLab.

Etcd tracks and manages the state of the Postgres cluster. Etcd is fully distributed, so it also keeps track of its own state and cluster membership. Each Etcd member needs to explicitly know all other members in the cluster, and know the state of each. To maintain the availability requirements of the GitLab application, there needs to be 5 Etcd. This is due to the n/2+1 rule that means in a cluster of 5, 3 members must be available for the cluster to be available. So, to deal with 2 failures and remain up, there needs to be 5 Etcd Instances.

If there are at least 3 Etcd instances available, and 1Patroni instance available, the PostgreSQL component of GitLab will be available. The high availability of the HAProxy instances will be covered in a later section.

Redis

The third installment of our GitLab High Availability installation and configuration guide is Redis High Availability. Redis is an open source pub/sub style in-memory queue, that is also commonly used as a data cache. In GitLab, Redis acts as a cache for user sessions, a job queue, and a message hub to connect different pieces of the GitLab application. Redis is a key part of the GitLab infrastructure because it needs to be the same consistent queue for all instances of the GitLab frontend, and if that queue was to crash, not only would GitLab go down, significant amounts of data that was still in the Redis queue could be lost. That could lead to submitted, but pending jobs to fail.

Originally, Redis had only a basic master-slave server implementation to deal with High Availability. That solution was not very robust, and limited the ability of Redis to be deployed in a production environment. Over the last few years, Redis has developed two different technologies to deal with both sharding and High Availability and failover. Respectively, these are Redis Cluster and Redis Sentinel. In the GitLab HA use case, it makes the most sense to use

Redis Sentinel to get the availability, without dealing with the extra computational overhead complexity of cluster our data.

Redis Sentinel is a form of replication, while Redis Cluster is a form of sharding. Replication is the desired form of clustering in HA situations. The need for the horizontal scaling provided by Redis Cluster is taken care of by the vertical scaling capabilities of Linux on Z/ LinuxONE. Less instances means less management overhead and wasted resources.

In a manual failover situation, Redis server instances in a master/slave configuration could be used without Redis Sentinel. It is also possible for master/slave instances to be configured in such a way that a slave will watch the master and try to promote itself to master. However, Redis does not include built in logic that is needed to avoid split-brain, so having automatic failover could be dangerous. This is another example where it is more efficient to run a larger, vertically scalable database application in a simpler master/slave configuration and manage the state/failover with a separate application. Redis Sentinel is a more lightweight application, so having twice as many instances to mitigate split-brain doesn't consume too many resources. Redis Sentinel is the definitive source of truth for which Redis instance is master and which others are slave. It will set the master/slave parameter on the Redis Servers themselves, but the Sentinel cluster is still the only definitive source of truth. It is possible that a current Redis cluster that is a master falls out of connection with all the Sentinel members, but is still reachable to clients. Sentinel would promote an available Redis slave to the role of new master, and there will be 2 Redis instances in the same cluster group, reachable to clients that are available as master. However, the old master is unmanaged, and should be ignored and not treated as a master by the client or any other nodes. For that reason, Sentinel is the source of truth, and should always be queried to find the clusters current master.

The Postgres/Etcd cluster and the Redis/Redis Sentinel cluster roughly follow the same architecture: master/slave database cluster managed by a lightweight active/active management cluster. The main difference between the two clusters is that Redis Sentinel/Redis cluster is natively designed to work together. With Postgres and ETCD, the Patroni application manages all the cluster logic, and uses ETCD as a storage engine. This also means that there is no Postgres driver that can natively take advantage of the HA cluster. This means HAProxy is needed in the Postgres HA cluster. In the case of Redis, there is a Redis Sentinel ready driver that can communicate directly with Sentinel to find the master. The driver manages the logic of finding the current master and connecting to it. In GitLab, this Redis Sentinel driver support is available starting with version 8.11. In any version of GitLab before 8.11, or any other application using a Redis driver without Sentinel support, it is possible to use HAProxy to present the master service. This requires writing a HAProxy health check script that queries the Sentinel servers to find the current Redis master, and forward the connection to that server.

It is important to note that the way Redis nodes handle dynamic configuration changes (i.e. slave promoted to master) is to edit the Redis configuration file on the node being changed. This can be helpful, because the state of the node is always tracked in the configuration file. However, if any automation tools like Chef or Ansible are used to manage the Redis clusters and set the initial configuration, they must not continuously update the configuration. By default, most management software will try to keep the file consistent, but Redis Sentinel needs to be able to change the file. If Ansible or Chef keeps trying to update the file, the cluster will experience errors, and very likely fail.

GPFS

GitLab is a frontend interface that is built around the git protocol. Git is the core of the GitLab infrastructure, as it handles the storage, versioning, branching, and organization of the source code. The git protocol stores all of the source code data it manages directly on the filesystem. This makes a distributed filesystem that is highly available crucial to the uptime of the application. Even if all the other components of GitLab, including the web application, are available, if the git projects are not available, the entire application is essential down.

GPFS is a parallel file system, which gives the advantage of both speed and redundancy. GPFS is recently had a branding change and is sold under the name IBM Spectrum Storage. However, GPFS is the specific component of Spectrum Storage that this paper is referring to, so for the purposes of this paper, it will be referred to simply as GPFS. GPFS achieves this by striping data across multiple disks and disk paths to allowing this data to be read simultaneously from multiple disks (in parallel). On enterprise grade hardware, a parallel file system is capable of increased performance by utilizing some of the features of the modern storage subsystem. GPFS can take advantage of the enterprise grade storage subsystems used in the Linux on Z environment, such as auto caching disk hotspots, and flash disk for metadata.

GPFS works by creating disk pools on storage subsystems separate from the physical Z machines. Then a set of GPFS management nodes are configured with SAN paths to the disk. This set of nodes manage the GPFS cluster, and forum a quorum on the state of the cluster. The GPFS nodes communicate with each other using the SSH protocol. From there, file systems can be created and managed by the GPFS management cluster. For non-management nodes to access to the GPFS filesystem, they must be explicitly added to the GPFS cluster and attach the filesystem.

The highly available architecture for GitLab requires that all the GitLab frontend instances have access the GPFS filesystem. All the GitLab instances will need to be able to talk to one another, because of the mesh networking setup of the GPFS cluster. However, each of the GitLab Frontends will be configured as clients in the GPFS cluster, so adding or removing GitLab frontends will not affect the availability of the GPFS cluster. It's a requirement that the GPFS file system is attached and reachable for the GitLab frontend instance to be considered reachable. There cannot be instance of GitLab running that does not have access to the git project files in the shared filesystem.

The availability of data in GPFS is managed in multiple ways. One way this is done is having multiple nodes with multiple paths to the disk. For every pool of disk, there needs to be multiple nodes that have direct connection paths to that disk. That means the loss of a single node, or any minority of nodes will never cause the GPFS cluster to go down, or any of the data to become unavailable.

Another feature of GPFS is the striping of data across multiple disk pools that are a part of the filesystem. The redundant paths to disk are to deal with an outage of a GPFS management node, where the data striping provides availability if part of the storage pool becomes unavailable. This striping actually provides 3 different advantages:

- High Availability in the case of short outages and disk maintenance downtime
- Live backups in case of actual disk failure or long-term downtime

- Performance boost, by reading different pieces of data simultaneously

The performance benefits of GPFS are multi-faceted. The striping of data across nodes means that multiple GPFS of clients can read the same data at the same time from different nodes for increased performance. Additionally, a single GPFS client node reading a large file from the filesystem can read different segments of the data from different backend subsystem. This is especially useful in the git protocol, where git projects tend to be large. The reads tend to be especially large, because of clones and forks, where writes are usually small and incremental. All of this happens without the client ever seeing it. As far as the GitLab frontend and git protocol are concerned, it is dealing with a regular filesystem.

There are several important aspects to keep in mind when designing GPFS for high availability. First is the availability of the disk subsystems. How many storage machines are available to spread the data across. There needs to be at least 2 to avoid a single point of failure. With enterprise grade storage, uptime is generally high, so within each subsystem, how many distinct disks and disk pools are available. There is a certain amount of HA configuration that is done on the actual disk subsystem, and a certain amount done on the GPFS side. The full configuration of a GPFS system is beyond the scope of this architecture document.

When dealing with the GPFS quorum, not all nodes are going to part of the client. For the GitLab environment, it makes the most sense to keep the GitLab frontends as basic clients with no voting power. The goal is to keep the GitLab frontend as stateless as possible, and no node should influence the GPFS availability. It is important to know that Quorum in GPFS has to take into account split-brain, so having 3 quorum nodes in a cluster is not enough. Instead, it is better to go for at least 5, and as many as 7. More than that is likely unnecessary, except in very large clusters.

As GPFS is a fully supported and mature product, there is ample documentation and support available, so it doesn't make sense for this paper to repeat existing documentation.

Docs: https://www.ibm.com/support/knowledgecenter/SSFKCN/gpfs_welcome.html
For information on setting up the quorum: http://www-03.ibm.com/systems/resources/configure-gpfs-for-reliability.pdf

HAProxy

HAProxy is an open source load balancer that commonly used in production, and is considered battle-tested. Making an application highly available usually involves having multiple instances of the same service. To access the HA service, other applications must have a destination to send the request. Although there are multiple instances of the HA service, there needs to be a singular way to address the service that will intelligently forward the request to an available instance. HAProxy is the solution to this multiple instance addressing and routing problem. Simply put, HAProxy binds a clustered service to a port on the machine it is running on. For each service, there are a list of potential instances, and a health check to see if each is active. When HAProxy receives a request at a bound port, it forwards it to one of the active instances for that particular service.

This will be used internally to route traffic to the current master in the PostgreSQL master/slave configuration. It will also be used for the same purpose in routing Redis requests in GitLab versions prior to 8.11. A separate instance of HAProxy will also be used to route traffic for the GitLab frontend cluster. There are separate instances of HAProxy because GitLab Frontend traffic is publically routed, while the Postgres and Redis data is internal. This separation is more secure and logically separates internal and external requests.

HAProxy can be customized and configured for many different forms of load balancing and health checking. For example, the Postgres load balancer doesn't check if each of the instances is up, instead checking a REST endpoint to see if the queried instance is the Postgres master. There will only ever be one instance that the traffic gets routed to. In the case of the GitLab frontend, there are multiple stateless instances that the incoming traffic can be distributed to. For this, HAProxy will run a heath check on a GitLab frontend port, and spread the requests out based on the routing algorithm (i.e. round-robin, least-conn, random, etc.)

HAProxy is a crucial part of this HA architecture design for GitLab. It acts as a hub for other HA applications in the environment, and becomes a glaring single point of failure. Fixing the problem requires a slightly different approach then previous clusters. It is not possible to create a master/slave HAProxy in the same manner as Postgres or Redis because there would need to be another load balancer to load balance the cluster of load balancers. This would continue ad nauseam. The proposed, and one of the simpler solutions to this problem is Keepalived.

Keepalived is an application that shares a single IP address across separate server instances. The Keepalived application is responsible for determining which of the available servers is the master and gets the shared IP address. The rules for determining the priority of servers is configurable by the end user, and Keepalived will handle the execution of IP migration based on those rules. The server that is ruled the secondary waits as a backup, and Keepalived will assign it the IP address if it ever gets promoted to master. More specifically, Keepalived runs as a daemon on the machine running HAProxy. Instead of running one single HAProxy instance, there will be a cluster of HAProxy instances, each running Keepalived. Keepalived uses a protocol called VRRP (Virtual Router Redundancy Protocol) to assign a floating IP address to one of the HAProxy instances. Keepalived defines a group of instances, and a IP address that will be assigned to one of the instances at a time. There are set of priorities and rules in the configuration that are used to determine which of the nodes in the cluster gets assigned the IP address. Simply put, there are a group HAProxy nodes that get share a single IP address. This IP address is how all the other applications will reference HAProxy. At any given time, one of the instances in the HAProxy/Keepalived cluster will be assigned the floating IP address. If any nodes go down, Keepalived should be able to handle the transfer of the floating IP to another active IP address. Using this IP address swapping, there is failover durability for HAProxy. This GitLab HA architecture only uses 2 HAProxy nodes in a Keepalived pair, because the uptime is so great, and HAProxy is stateless, so if one goes down for maintenance, a new one could easily be brought up. If more durability and availability was wanted, more HAProxy instances could exist in the cluster.

GitLab Frontend

The GitLab frontend is where all the application logic for GitLab resides. All of the previously implemented components are part of the GitLab stack, but not really a part of the core GitLab application. Even though all the logic behind the GitLab application is part of the "GitLab frontend" it does not store any state, and can easily be made highly available. In this case stateless means that an individual GitLab instances does not hold any information in the long term. The GitLab frontend applies the application logic to the request, and saves the state to one of the other components.

The only state that the GitLab HA application holds is its configuration file. The configuration file in GitLab points to all HA components that have been built earlier in this paper. The GitLab frontend configuration also contains some self-referential information, like its own IP address and hostname. There are 2 common ways to manage configuration files of this manner: ETCD and Infrastructure automation templating (Ansible, Chef, Puppet, Salt Stack, etc.) In the architecture described in this paper, Ansible was used to deploy GitLab frontend instances, and the Ansible templating engine customized the configuration files on creation. This is a good solution in a manual allocation architecture like the one described here. The other type of architecture that is becoming common is the dynamically deployed applications. These dynamically scaling applications will be automatically spawned and destroyed based on the demand placed on the cluster. In situations like that, it is usually better to store the configuration in a place like Etcd, and have the newly spawned application instances fetch its configuration from a predetermined Etcd location. Both methods are acceptable, but the Etcd solution is a bit more complicated, and not necessary to manage the configuration of a GitLab cluster of this relatively small size.

HAProxy will be used as the load balancer for the GitLab frontend. However, it will be a separate load balancer from the one used in the previous HA components. This adds security by having a separation of the outside and inside networks. The HAProxy for GitLab will have its outside requests coming from outside the private network, and forward all its requests to the inside network. By having HAProxy act as the edge node between networks, there is a tight control of what can be accessed on the GitLab frontend machines. No one can try and connect to any unauthorized ports on GitLab, because the HAProxy will only forward specifically allowed requests.

The amount of GitLab nodes that are required is totally up to the implementer of this architecture. The example architecture given in this paper uses an instance in each availability zone. The idea of this is to never have GitLab frontend fail while the rest of the cluster is still up. Each one of the instances is standalone and stateless, so the number of nodes directly correlates to the amount of availability that GitLab frontend HA supplies. For HA purposes on Linux on Z, there should be at least 3 instances of the GitLab frontend. The GitLab frontend on Linux on Z can scale vertically as well as horizontally. Creating more instances will help divide the tasks across the frontend cluster. The best practice is to scale horizontally until the HA requirements are met, then scale vertically once all the availability zones are utilized. It

Additionally, GPFS must be installed on all the GitLab frontend clients so they can access the shared filesystem.

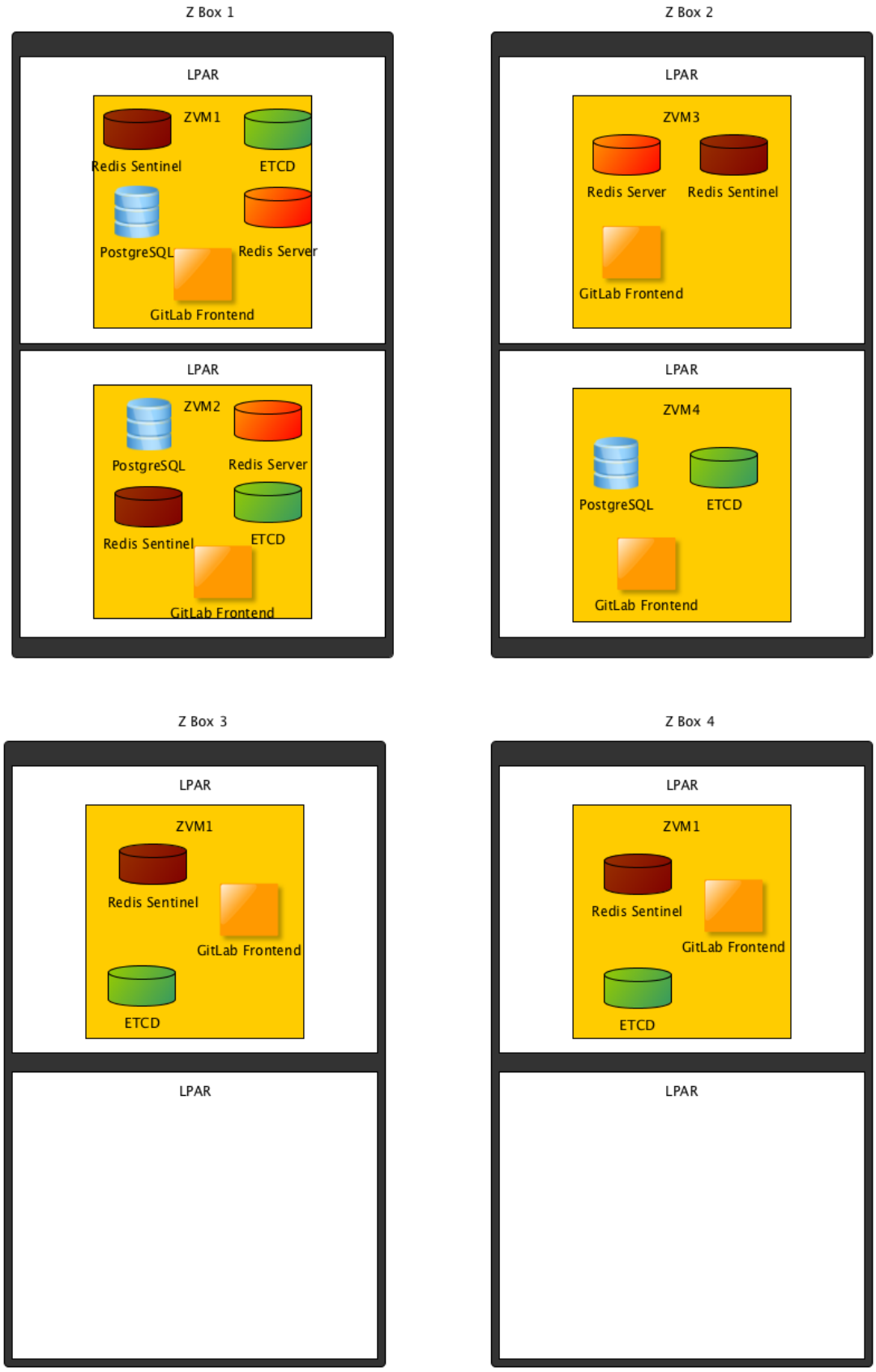
GitLab Availability Zones on Linux on Z

In this implementation GitLab is being used as internal enterprise application that critical to the in processes of a company. Downtime would not necessarily stop current external applications from working, but would stop development and updates of those applications. These uptime commitments determine the amount of availability that is required. If the entire development environment for a company exists inside one datacenter, then there is no point in creating geographically dispersed availability zones for GitLab. In this scenario, if the main datacenter goes down, then GitLab cannot be utilized anyway, so there is no purpose in having that much availability. For the purposes of this use case, availability is needed if the datacenter used by development is still available.

There are many factors to account for when designing availability zones within a single datacenter. It is crucial to design the system so that the failure of one single system will not bring down the whole application cluster. In the Linux on Z environment, the most granular form of virtualization is LPARS. A logical partition (LPAR) is the division of a computer's processors, memory, and storage into multiple sets of resources so that each set of resources can be operated independently with its own operating system instance and applications. Running on an LPAR is Z/VM, a virtualization host. Z/VM is the operating system on the LPAR, whose sole purpose is to act as a virtualization host for other guest instances. Inside of Z/VM is where the virtual Linux instances run. To increase availability, each Z/VM instance will be considered a separate availability zone. In this use case, there are multiple physical Linux on Z/LinuxONE instances. To further the availability, the instances are divided in such a way that losing one complete Linux on Z instance still would not cause downtime. This is most useful for instances where machines are being brought down for regular maintenance and updates. Since the overall GitLab application is divided into multiple clustered applications, components of the different applications can share the same availability zones. The smallest availability zone in this environment in the Z/VM host. Each Z/VM host runs inside of its own LPAR. Each LPAR runs on a physical z-box footprint. In this example, there are a total of 4 physical machines that are accessible. Z boxes 1 and 2 are the primary Linux on Z instances and are dedicated for running production like stuff. Both have ample free capacity, and good uptime. There is never any time where both should be down at the same time. Each run 2 instances of Z/VM, where each instance sits inside its own LPAR. Z boxes 3 and 4 are mixed workload devices that each run one instance of Z/VM inside its own LPAR. They are primarily used to add additional availability for our 5 cluster members, as well as running a stateless instance of the GitLab Frontend.

In this design, the GitLab application could withstand the shutdown of a complete Z box and stay available. In fact, no 2 LPARS becoming unavailable would cause GitLab to become unavailable. With the reliability of the Linux on Z environment, each individual component of GitLab can be vertically scaled to handle any extra performance, and excess redundancy is not needed, as machine uptime is solid. This is a hybrid solution of traditional enterprise grade hardware at scale, and distributed applications that combines benefits of each style. Vertical scale, reliability, uptime, and security are provided by the Linux on Z environment. The distributed ‘cloud’ model allows for rolling upgrades on application components, smaller more succinct application components, even more availability and greater deployment flexibility. The

desired availability for all components in the GitLab application is 2 availability zones going down and the application remaining up.



Conclusion

A cloud like highly available architecture can still be advantageous in a Linux on Z environment. The cloud scale capabilities are not reserved for x86 systems. In fact, the stability offered by Linux on Z allows a hybrid approach of vertical and horizontal scaling. This unique fusion allows for a mixture of new and old style workloads: enterprise with open source, monolithic with micro services, Z systems and Dev Ops.

This architecture focuses on an open source application that can be broken down into its individual components and made to be highly available. This also shows the architecture of a collection of popular distributed applications. The architectures for each of the components listed in this paper can be used outside of the scope of GitLab. Also, these HA applications can serve more than just the GitLab application. For example, Postgres used for GitLab can also serve other purposes within an organization. The vertical scaling combines with the availability zones allows for each of these components to be scaled to large levels, well beyond the needs of GitLab.