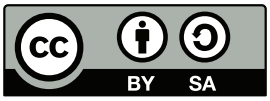
Choreographing Immutable Infrastructure

Kasper Kronborg Isager
IT University of Copenhagen
kasi@itu.dk

May 15, 2017



Contents

Preface **1**

1 Introduction **1**
1.1 State of the art . . . 1
1.2 Problem definition . . . 2
1.3 Contribution . . . 3

2 Related work **3**

3 Background **4**
3.1 Definitions . . . 5
3.1.1 Services . . . 5
3.1.2 Servers . . . 8
3.2 Upgrade strategies . . . 9
3.2.1 Rolling upgrades . . . 9
3.2.2 Blue-green upgrades . . . 10

4 Analysis **11**
4.1 Strategy limitations . . . 11
4.2 Partitioned upgrades . . . 11
4.2.1 Partitioning servers . . . 12
4.3 Findings . . . 15

5 Implementation **15**
5.1 Representations . . . 16
5.1.1 Infrastructure . . . 16
5.1.2 Upgrades . . . 17
5.2 Partitioning servers . . . 18
5.3 Demonstration . . . 20
5.3.1 Orchestrating infrastructure . . . 20
5.3.2 Evolving infrastructure . . . 22
5.3.3 Lessons learned . . . 24

6 Limitations **25**
6.1 Long-tail . . . 25
6.2 Service dependencies . . . 26
6.3 Allocation counts . . . 26
6.4 Backwards compatibility . . . 27

7 Conclusion **27**

List of Figures **29**

References **30**

Appendices **32**

A Source code **32**

A.1 Partitionist 32

A.1.1 `infrastructure.go` 32

A.1.2 `deployment.go` 38

A.1.3 `orchestrate.go` 42

A.2 Demonstration 43

A.2.1 `digitalocean.go` 43

A.2.2 `bluegreen.go` 45

A.2.3 `rolling.go` 46

A.2.4 `partitioned.go` 47

A.2.5 `cluster.go` 47

Preface

This thesis was written during the spring semester of 2017 under the supervision of Søren Debois as a completion of my Bachelor's degree in Software Development at the IT University of Copenhagen.

The subject of my thesis–*immutable infrastructure*–is one that has long piqued my interest. My hope is that this thesis and its findings can help improve adoption of immutable infrastructure by tackling a very central issue; *upgrades*.

I would like to give thanks to my supervisor for his enthusiasm for my undertaking and for his guidance and feedback while writing this thesis. More cups of coffee than I care to admit were harmed in its making and I am thankful that Søren found my work interesting; the coffee was not harmed in vain.

1 Introduction

The concept of immutable infrastructure is a new one first presented in [Fowler, 2013]. The idea is simple, although powerful when put to proper use: Instead of changing existing server infrastructure when upgrading servers, do away with the existing servers and start up new ones with the upgrades applied. In other words, we no longer reconfigure; we dispose and rebuild. By putting the process of disposing and rebuilding at the centre of the operational workflow, a variety of operational issues such a server failures, security breaches, disaster recovery, and more, all become trivial from an operational perspective; dispose and rebuild.

Immutable infrastructure does present a problem in the practicality of disposing and rebuilding servers: How do we prevent service disruption when disposing servers? This problem becomes complex in modern infrastructures were several services are often co-located on the same servers.

A variety of different strategies for performing immutable server upgrades with minimal, or even without, service disruption already exist. For all intents and purposes, immutable server upgrades is a solved problem. Modern cloud providers and service orchestrators already support zero-downtime upgrades at both the infrastructure and application layer.

1.1 State of the art

When one starts digging into immutable infrastructure as used in the industry, problems do turn up. First of all, people cannot seem to agree on what immutable infrastructure actually is. In [Dunn, 2014] the author, who is the product manager at Chef, an infrastructure automation tool, makes the claim that immutable infrastructure is impossible to achieve as a system will always maintain some form of state. This rather

strict definition is in [Motlik, 2014] rebutted and the author, then CTO at Codeship, a platform for continuous integration and delivery, goes on to claim that immutable infrastructure is instead defined by state isolation rather than absence.

Related to the problem of defining immutable infrastructure in terms of state is how to handle immutable infrastructure when state *is* involved. In [Pais, 2014], Fowler goes on to say:

> "Everything needs to be stateless where possible. As I mentioned previously, we have 'cheats' like managed database systems. I'm not going to comment on how you have to change architecture to have your own immutable, disposable database systems since it's thankfully not a problem I've needed or wanted to solve yet."

If outsourcing state to a managed system is not an option, because of for example security or budget concerns, immutable infrastructure can become an obstacle. [Motlik, 2014] hints at a solution when talking about state isolation:

> "The main advantage when it comes to state in immutable infrastructure is that it is siloed. The boundaries between layers storing state and the layers that are ephemeral are clearly drawn and no leakage can possibly happen between those layers."

What is missing here is the *how* of it. How do we draw the boundaries between the stateful and ephemeral layers of the infrastructure? In the case of a database system for example, how do we know if it is safe to dispose of a database server without also disposing of its associated state? While we can gain answers to these questions given enough knowledge of each specific situation, we lack a general framework for dealing with state in immutable infrastructure.

1.2 Problem definition

The problem we wish to solve in this thesis is determining how to effectively upgrade servers in immutable infrastructures without disrupting service even when state is involved. Upgrades may also be further constrained by operators as to allow prioritising for example fast upgrades over complete elimination of service disruption.

In order to solve this problem, we first present and expand upon the concepts involved when dealing with immutable infrastructure. In doing so, we wish to create a common understanding of what immutable infrastructure involves which in turn allows us to identify, analyse, and solve the problems related to dealing with state. Finally, we design a server upgrade strategy that allows us to upgrade immutable servers regardless of whether state is involved or not.

1.3 Contribution

The primary contribution made by this thesis is a server upgrade strategy that recognises that neither of the existing strategies can cover all reasonable infrastructure configurations. The proposed strategy instead relies on inferring the upgrade constraints of each server considered for an upgrade and then partitions servers into independent upgrade groups based on these constraints in addition to those supplied by operators. An implementation of the upgrade strategy and its associated algorithms is also provided along with a demonstration of how the strategy works in practise.

2 Related work

Immutable infrastructure has received a lot of attention for the past few years. [Stella, 2016] is an entire book dedicated to the subject and deals with many of the same things that we discuss in section 3; what is an immutable server, how do we deal with upgrades, and what should we do about stateful services. In relation to the latter, [Stella, 2016, p. 28] however states the following:

> "The simple way to get immutable infrastructure benefits in your data services is to use data services that provide external service level agreements and are managed by others."

This again resonates with the problem definition we presented in section 1. While the literature on immutable infrastructure is not unsubstantial, stateful services are more often than not left unaddressed.

The closest related, albeit old, literature we have managed to find is [Ajmani et al., 2003] which recognises the need for upgrading systems of servers without service disruption even when state is involved. Their approach to upgrades is however different from what we employ in immutable infrastructures. The infrastructure they address is mutable and they perform upgrades through restarts rather than rebuilds. They also employ sophisticated procedures and adapters in order to convert state between upgrades, determine when servers should upgrade, and to enable backwards compatibility between services during an upgrade.

An important assumption made in [Ajmani et al., 2003] is that upgrades are rare, for example happen less than once a month. While this may have been the case 14 years ago, it is no longer so. With the advent of *continuous delivery* as presented in [Humble and Farley, 2010], upgrades have become frequent and having to implement sophisticated procedures before every upgrade is no longer an option.

3 Background

As touched upon in section 1, with immutable infrastructure we dispose of existing servers and rebuild rather than reconfigure them when performing upgrades. Let us consider an example of a single server running a single service that we wish to upgrade from version "1" to version "2". When depicting server upgrades, we will use solid lines (———) for indicating resources before the upgrade, dotted lines (·····) for indicating resources after the upgrade or a step thereof, and dashed lines (- - - -) for indicating a group of resources being upgraded.

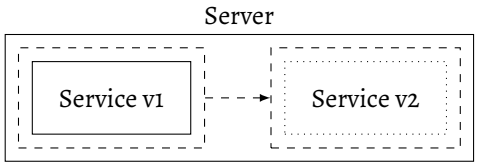


Figure 1: A mutable upgrade of a service from version "1" to "2".

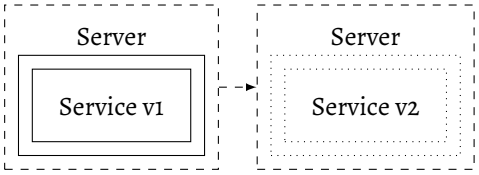


Figure 2: An immutable upgrade of a service from version "1" to "2".

In figure 1, the operator accesses the existing server and modifies it in-place, whereas in figure 2 the operator creates a new server and destroys the old one. When mutable server upgrades as seen in the first example are used, we define an infrastructure as *mutable*. Similarly, an infrastructure is defined as *immutable* when immutable server upgrades as seen in the second example are used.

[Fowler, 2013] draws a parallel between immutable servers and the immutable data structures found in modern functional programming languages, a parallel that should be clear from these examples. In many ways, immutable servers and immutable data structures are similar, though only when observed from the outside. That is, immutable servers are only immutable from the perspective of an operator who is not allowed to change them in-place. An important fact is that immutable servers are not immutable on the *inside*. Services running on the servers might very well maintain state, generate logs, and more. When upgrading immutable servers, an operator must take care when dealing with state that must be persisted even when servers are disposed as touched upon in [Morris, 2013]. We will go into the details of how to deal with state later in this section.

3.1 Definitions

While the concepts dealt with in this thesis, such as "service" and "server", are generally well-understood, it makes sense to come up with some generalised definitions that are applicable for all the different infrastructure configurations considered.

As an example, think for a second of how you would define a server. Your definition will likely involve both hardware (e.g. a physical machine in the shape of a network host) and software (e.g. an application providing a service to clients) or a combination of the two (e.g. virtual machines provisioned by cloud providers), depending on how strict your definition is. Things start to break down when immutable infrastructure comes along and dictates "dispose of your server!", to which your answer might likely be "which?".

If defined as hardware, it may be tempting to throw your physical machine out the window and replace it, which, while immutable, is a wasteful way of operating your infrastructure. If your definition is more strict and you define a server as the application providing a service to clients, you might reinstall the application, which leaves us back at the mutable upgrade example from figure 1. The definition involving virtual machines is closer to what we are looking for and fits with the immutable upgrade example from figure 2. While better, it is still not good enough; what would happen to our definition once we start nesting virtual machines?

What follows are the definitions of the terms "server" and "service" as used throughout this thesis. These definitions are meant as a way of creating a common understanding of immutable infrastructure that we will later use when solving the problem of dealing with state when performing immutable server upgrades.

3.1.1 Services

A service is a software process run on top of a *service host*. A service can maintain persistent *state*, as is the case with for example relational databases, key-value stores, and more. Any non-persistent state a service might maintain, such as logs or in-memory caches, is not of interest as such state is disposable.

Service hosts The concept of a service host is one we introduce to deal with the concept of disposability prevalent in immutable infrastructure. A service host provides a layer that can be disposed and rebuilt in the event that changes are to be made to the host itself or the services run on top of it. Some services, such as hardware or software virtualisation, may provide additional service hosts on top of which other services can run. Some examples of service hosts include:

- **A physical machine**; disposal and rebuild would require re-installing the operating system on the machine. A physical machine is the lowest layer possible.

- **A virtual machine**; disposal and rebuild would require removing the old virtual machine and provisioning a new one.
- **A software container**; disposal and rebuild would require the same as with a virtual machine though on a smaller scale.

It is important to note that only the closest service host to a given service requires a disposal and rebuild when changes are to be made to the service. As service hosts can run other service hosts, a stack of hosts can be modelled. This in turn provides a way of limiting the effect that changes to one service–or a group of services–might have on other services.

An example of a typically employed service host stack is that used in modern cloud architectures as described in [Moreno-Vozmediano et al., 2012]. In these architectures, the physical machines are managed by a cloud operating system which in turn exposes virtual machines for use by cloud users. When a cloud user wishes to dispose of a virtual machine or change it in a way that requires a rebuild, the cloud operating system only has to dispose of that specific virtual machine and the underlying physical machine remains untouched.

A cloud user might also add service hosts of their own, such as software container engines, on top of which they can run their business related services. As a consequence, when changes are to be made to these services, the underlying virtual machine also remains untouched.

Service state As touched upon in section 1, service state is a point of much frustration when dealing with immutable infrastructure. In defining service state, we wish to identify the aspects of state necessary for determining how to perform an immutable upgrade of any server running services that maintain state.

Services maintaining persistent state will be defined as *stateful* whereas services maintaining either non-persistent or no state at all will be defined as *stateless*. This definition of statelessness somewhat relates to the definition of stateless servers made in [Coulouris et al., 2011, p. 528] where a stateless server is defined by the fact that no state recovery is required when restarting the server. Similarly, no additional considerations are required when disposing servers running only stateless services.

When considering stateful services, the state maintained can be either *external* or *internal*. Internal state is state that lives directly on the server running the corresponding service; disposal of the server therefore also implies disposal of the state. External state lives outside the server running the service and disposal of the server does therefore not imply disposal of the state. Some examples of external state include:

- Detachable block-level storage provided by distributed file systems. These can either be managed by cloud providers or directly by operators by means of for example [CephFS, nd].

- Software container volumes such as those found in [Docker, nd] or [rkt, nd].

State may additionally be considered either *concurrent* or *non-concurrent*. While concurrent state can be accessed by multiple services simultaneously, non-concurrent state can only be accessed by a single service.

An example of both concurrent and non-concurrent state is found in figure 3. Here, a single state in the form of a block device is shared between all participants in a master/slave setup. From the perspective of the master, the state is non-concurrent; only one master may access the state as both reads and writes are performed. From the perspective of the slaves however, the state is concurrent as slaves only read from the state; writes are instead passed on to the master.

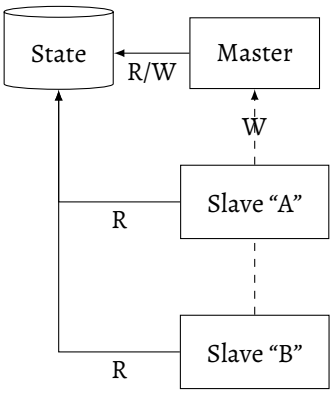


Figure 3: A master/slave setup employing both concurrent and non-concurrent external state.

Lastly, state may also be considered either *replicated* or *non-replicated*. Replicated state is state that is available in more than one location; the disposal of a single replica, such as the internal state of a server or an external block device, does therefore not imply disposal of the state itself. In contrast, non-replicated state is only available in a single location and cannot survive the disposal of its medium, such as the server on which it lives. The example found in figure 3 is a good example of state that may from the outside appear replicated, but is indeed non-replicated; the shared state exists only in a single location and as such cannot survive the disposal of its medium.

Service replication Separate from the concept of state replication is the concept of *service replication*. A service is defined as replicated if two or more instances of the service are run across separate servers. Replicating services is required for achieving fault tolerance, that is tolerating the loss of a number of service instances without causing service disruption.

Service disruption Service disruption occurs when either too few or no instances of a service are available to clients. In case of non-replicated services, service disruption will occur as soon as the single instance of the service becomes unavailable or unable to serve all clients. In case of replicated services, service disruption can occur if too many replicas are unavailable simultaneously such that not all clients can be served by the remaining replicas. The number of instances that are allowed to be unavailable during an upgrade will be referred to as the *upgrade tolerance*.

3.1.2 Servers

In defining servers it makes sense to first consider the typical distinction made between *physical* and *virtual* servers as hinted at in [Coulouris et al., 2011, p. 319] and as outlined by the previous cloud architecture example. These two types of servers very much fit with the definition of service hosts. As such, it is beneficial to generalise the definition of a server to the closest service host to a given service. Using this definition, the terms "server" and "service host" become seemingly interchangeable with the distinction being that "server" is always defined within the context of a given service.

The benefit of this definition of servers is that it becomes easy to talk about a service and its associated service host in isolation of any additional layers of service hosts that may exist; we merely refer to the two as a service and its server. The definition also allows us to disregard the implementation details of the underlying service host.

Looking back at the simple examples presented in figures 1 and 2, the benefits should become clear: Even though the term "server" is used, it may be substituted with "virtual machine", "software container", or any other type of service host imaginable, and the examples would still be perfectly valid.

Server lifecycle An important aspect of servers is their *lifecycle* which will be defined as the order in which disposals and rebuilds occur.

In order to prevent service disruption during an upgrade, rebuilds can occur before disposals as seen in figures 1 and 2. This ensures that a replacement for a server is available before disposing of it, though this will incur a *provisioning surge* where more servers are provisioned than are actually required.

In some cases, however, it is not possible to build the new servers before disposing of the old ones. Whether or not this is possible depends on the state employed by the services running on a given server and as such the lifecycle of a server can be inferred from its state. The important observation is that in order to be able to perform rebuilds before disposals, it is required that state can be moved from the old server to the new without first disposing of the old server.

Since internal state is inherently non-concurrent, as only the server on which the state lives can access it, internal state must be replicated in order for the server to have a valid lifecycle. Rebuilds can then be performed before disposals as the old servers replicate their state to the new servers before being disposed.

With external state, concurrency plays an important role. If the external state is replicated, it can be treated the same way as internal state; replicate the state from the old server to the new. If the external state is however not replicated, the concurrency of the state will determine whether rebuilds can happen before disposals. If the state is non-concurrent, then the old server must be disposed before the new server can access the state; this is for example the case with the master from figure 3. If the state

is concurrent, then both the old server and the new can access the state simultaneously and the old server need not be disposed before the new one is built, as is the case with the slaves from figure 3.

3.2 Upgrade strategies

We will now have a look at different strategies for performing immutable server upgrades. For each strategy, we will briefly outline how server upgrades are handled by the strategy and based on this identity its limitations.

3.2.1 Rolling upgrades

The *rolling upgrade* is a widely used upgrade strategy employed in both mutable and immutable infrastructures alike. Instead of upgrading all servers at the same time, which can cause service disruption in the event that servers have to be disposed before being rebuilt, in a rolling upgrade servers are upgraded either one at a time or in groups of a certain size. While groups are processed sequentially, the servers within each group can be upgraded in parallel. It is also a possibility not to separate servers into distinct groups, but instead maintain the invariant that at most a fixed number of servers are being upgraded at any given moment.

If new servers can be built before the old ones are disposed then rolling upgrades provide a way of limiting the provisioning surge during an upgrade. If old servers must be disposed before being rebuilt then rolling upgrades can eliminate service disruption if the chosen group size does not exceed the upgrade tolerance.

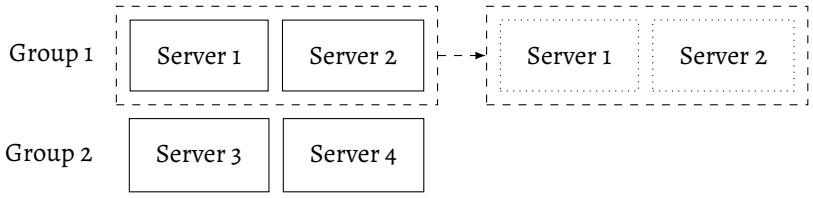


Figure 4: The first step in a rolling upgrade of 4 servers with a group size of 2.

Limitations A drawback of rolling upgrades is the time it takes to upgrade an entire cluster as server upgrades happen in groups. The total upgrade time therefore depends on the group size, which in turn depends on either the upgrade tolerance or the maximum allowed provisioning surge.

Rolling upgrades also become tricky when multiple services with different upgrade tolerances are run on the same servers. In this case, the lowest upgrade tolerance takes precedence over the higher ones, which can further degrade the upgrade time.

Another drawback of rolling upgrades is the requirement that only backwards compatible upgrades be made. This due to the fact that the new versions of services will run alongside the old versions of the same services during an upgrade.

3.2.2 Blue-green upgrades

The *blue-green upgrade*, described in [Humble and Farley, 2010], is like the rolling upgrade employed in both mutable and immutable infrastructures. While a rolling upgrade gradually upgrades servers in the same cluster, a blue-green upgrade instead maintains two separate clusters referred to as blue and green. Once the servers in the green cluster have been built, traffic is moved from the blue cluster to the green and the servers in the blue cluster are disposed. This makes blue-green upgrades optimal in terms of time taken to perform an upgrade since all new servers can be built simultaneously. As the new and old servers are run in isolation, blue-green upgrades additionally allow for backwards incompatible upgrades to be made.

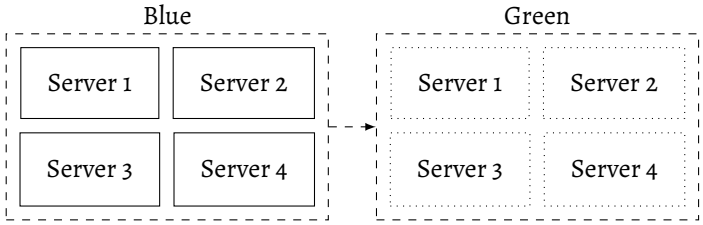


Figure 5: A blue-green upgrade of 4 servers.

A variation of the blue-green upgrade is the *canary* upgrade, also described in [Humble and Farley, 2010]. While in a blue-green upgrade all traffic is moved to the green cluster when ready, in a canary upgrade traffic is instead moved gradually.

Limitations A drawback of blue-green upgrades is the requirement that new servers must be built before the old ones are disposed. This is not always possible per our definition of server lifecycle; when external, non-concurrent, non-replicated state is employed, the old servers must be disposed before being rebuilt. When such state is employed, a blue-green upgrade is not an option and a rolling upgrade must instead be used.

While blue-green upgrades need not consider the upgrade tolerance of services, provisioning surge can also become a problem. As blue-green upgrades require building every new server before disposing of the old ones, infrastructure capacity must be doubled during an upgrade.

4 Analysis

In this section, we analyse the limitations of existing strategies for upgrading immutable infrastructure and present a new strategy with the goal of overcoming these limitations.

4.1 Strategy limitations

As outlined in section 3.2, both the rolling and blue-green upgrade strategies each have their own set of limitations. While rolling upgrades can be used regardless of server lifecycle requirements as long as the upgrades are backwards compatible, it is the least efficient of the strategies in terms of time taken to perform an upgrade. Blue-green upgrades are more efficient in this regard, but cannot be used when servers must be disposed before being rebuilt. Additionally, blue-green upgrades incur a high provisioning surge due to infrastructure capacity being doubled during an upgrade.

Let us step back for a moment and try not to view these two strategies as completely separate. If we constrain the blue-green upgrade to only operate within a single cluster, a blue-green upgrade can be regarded as a rolling upgrade with a single group. If we further constrain the blue-green upgrade to only be backwards compatible[1], the only thing preventing us from performing a blue-green upgrade over a rolling upgrade is server lifecycle requirements.

With these two constraints in place, we only need to look at server lifecycle in order to determine which strategy to use. This does however introduce another limitation as we then assume a single strategy as the best fit for the infrastructure as a whole. If, for example, a single server out of a hundred requires that a rolling upgrade be performed over a blue-green upgrade, we will be forced to choose a less efficient upgrade strategy on the basis of a single server deviating from the rest.

Provisioning surge is also a constraint we might need to consider. If an operator limits the allowed provisioning surge such that a blue-green upgrade becomes impossible, we are again forced to perform a rolling upgrade.

4.2 Partitioned upgrades

We should by now have made our case for a new upgrade strategy that recognises that neither the rolling nor the blue-green upgrade strategy is sufficient. While the previous constraints imposed on the blue-green strategy allow us to make it more compatible with the rolling upgrade, we are still forced to choose between two seemingly distinct strategies. What we want to achieve is a single, unified strategy that provides us with the flexibility of the rolling upgrade as well as the efficiency of the blue-green upgrade.

[1] We will discuss later why this is a reasonable constraint given the flexibility it affords us.

To this end, we introduce the concept of a *partitioned upgrade*. The idea of a partitioned upgrade is similar to that of a rolling upgrade in that servers are separated – *partitioned* – into groups. Unlike rolling upgrades, groups in a partitioned upgrade are processed in parallel, and the servers within each group are then upgraded sequentially. We will refer to the set of groups in a partitioned upgrade as the *upgrade plan*.

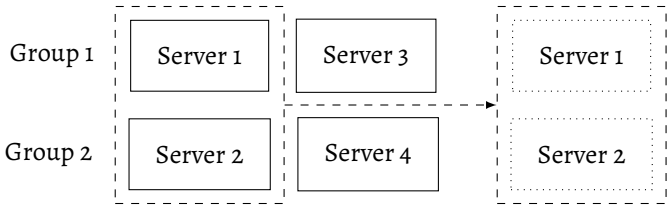


Figure 6: The first step in a partitioned upgrade of 4 servers.

A partitioned upgrade may seem like a different way of doing the exact same thing as a rolling upgrade, but the inversion turns out to be invaluable. Where the rolling upgrade allows us to upgrade servers within the same group independent of each other, the partitioned upgrade allows us do the same with groups instead. Whenever we split a group into two or more subgroups we therefore express that the subgroups can be upgraded independent of each other. What partitioned upgrades give us is the ability to *divide and conquer*.

An example of how we can benefit from this can be made by comparing figures 4 and 6. If, for example, we conclude that server 3 can be upgraded in parallel with server 1 and 2, with a rolling upgrade we would need to move server 3 out of group 2 and put it into group 1. With a partitioned upgrade though, we can split group 1 into two groups to achieve the same thing.

4.2.1 Partitioning servers

In a partitioned upgrade, we start out with a single group of servers and assume that servers have to be upgraded one at a time, corresponding to a rolling upgrade with a group size of 1. Our goal is then to partition the servers in such a way that the number of servers in each group is minimised. Every time we partition a group into subgroups, we therefore optimise the upgrade plan as a whole. Ideally, each group should only contain a single server, corresponding to a rolling upgrade with only a single group, and therefore matching the efficiency of a blue-green upgrade.

In the following, we describe each of the four partitioning steps performed when creating a partitioned upgrade plan.

Partitioning by lifecycle The first constraint we partition servers by is lifecycle. Given the initial group of servers, we split this group into two such that servers that must be

disposed before being rebuilt go in one group and the remaining servers go in the other.

The servers that must be disposed before being rebuilt can then be further partitioned according to the upgrade tolerances of the services running on the servers. Similarly, the servers that can be rebuilt before being disposed can be further partitioned by provisioning surge if required.

Partitioning by services Before partitioning by upgrade tolerance the servers that must be disposed before being rebuilt, we need to first partition them by the services they run as upgrade tolerances are defined at the service level.

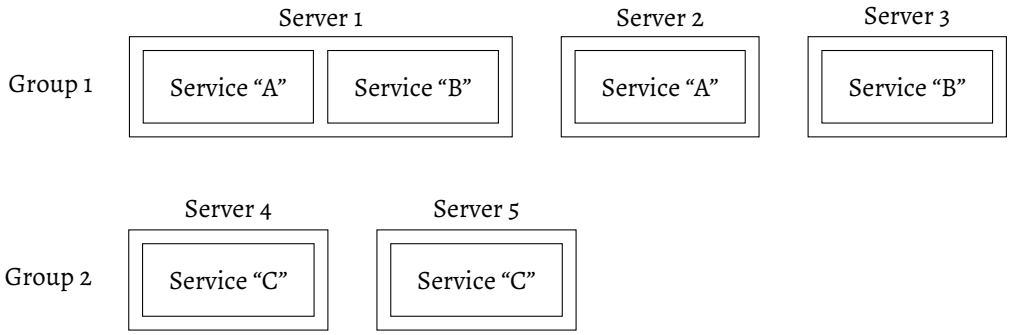


Figure 7: Partitioning of 5 servers according to the services they run.

To partition servers by the services they run, we need to group servers that run the same services. If, for example, we encounter a server that runs service “A”, we must add this server to the group of servers running service “A”. If each server only runs a single service, we will end up with groups that each contain servers running only that specific service, which would be the optimal outcome.

As mentioned previously, several services are often run co-located on the same servers. If, for example, we encounter a server that runs both service “A” and “B”, we must join the groups of servers running service “A” and “B” and add the server to this joined group. If all servers are connected by their services we will end up with a single group, leaving us no better off than a rolling upgrade would.

What inspired this partitioning step was [Locksmith, nd], a distributed reboot manager. Similar to rolling upgrades, Locksmith enables servers to perform rolling reboots when mutable server upgrades are performed. Locksmith organises servers in groups and uses a distributed semaphore within each group to avoid exceeding the upgrade tolerance of the group. While Locksmith relies on operators to define groups, we instead infer the groupings based on the services run on the servers.

Partitioning by upgrade tolerance Once partitioned by services, we can further partition servers by the upgrade tolerances of the services they run. To partition servers

by upgrade tolerance, the servers must be grouped such that each service only appears in at most a number of groups equal to its upgrade tolerance. Since groups are processed in parallel and servers within each group in sequence, we therefore ensure that the upgrade tolerance is not exceeded.

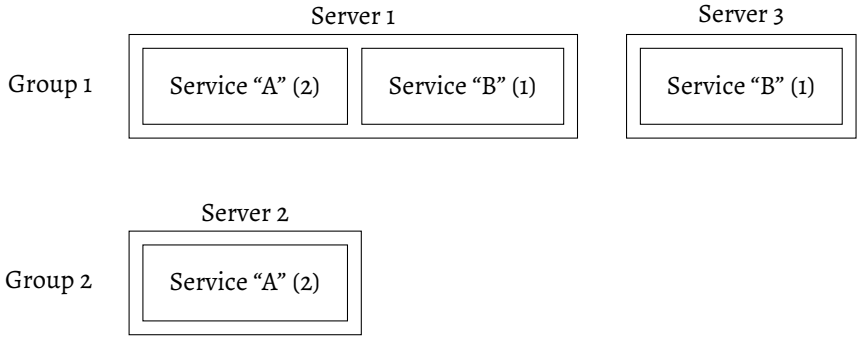


Figure 8: Partitioning of 3 servers from group 1 of figure 7 according to upgrade tolerances listed in parentheses. Note that service "A" will be unavailable during this upgrade.

If all services within a group of servers have the same upgrade tolerance, partitioning is trivial. In this case, we evenly distribute the servers across the groups and move on. If co-located services have different upgrade tolerances, partitioning becomes more tricky. In this case, the servers must still be evenly distributed across the groups, but each server can only be put into the first n groups, where n is the lowest upgrade tolerance of the services running on the server.

Partitioning by provisioning surge If limiting provisioning surge is required, we can further partition the servers that can be built before the old servers are disposed. Partitioning by provisioning surge is almost identical to partitioning by upgrade tolerance with the difference that provisioning surge is defined at the server layer rather than the service layer. To partition by provisioning surge, the servers must therefore be evenly distributed across a number of groups equal to the provisioning surge.

Distributing servers In describing partitioning of servers by upgrade tolerance and provisioning surge, we specified that servers must be evenly distributed among groups. This is done in order to ensure that the processing time of each group is roughly the same to allow as many servers as possible to be upgraded in parallel. What we did not specify was by which measure servers were to be distributed.

The first measure we might consider is *count*. If we distribute servers across groups such that each group contains roughly the same number of servers, our upgrade plan will be efficient under the assumption that the time taken to dispose and rebuild each server is also roughly the same. However, experience tells us that this is rarely the case.

To this end, we instead specify *weight* as the measure by which we distribute servers. The weight of a server is a relative measure that estimates how long it takes to perform a disposal and rebuild of the server. As such, a server with a weight of 4 would take twice as long to dispose and rebuild as a server with a weight of 2. By distributing servers evenly according to their weights, we can create an efficient upgrade plan even when heavy servers are among the servers to be upgraded.

4.3 Findings

In analysing existing upgrade strategies, specifically the blue-green and rolling upgrade strategies, we have identified limitations that make these strategies difficult to use in all infrastructure configurations. The blue-green upgrade only makes sense when stateless or certain types of stateful servers are to be upgraded, whereas the rolling upgrade can be used regardless of the type of servers being upgraded at the cost of efficiency. Operators are therefore forced to choose between the two strategies based on knowledge of the infrastructure considered for an upgrade.

To address the limitations of existing strategies, we have introduced a new strategy: the partitioned upgrade strategy. Existing strategies regard the servers considered for an upgrade as homogeneous; the blue-green upgrade require that all servers can be rebuild before being disposed, and the rolling upgrade require that all servers be content with the same group size. The partitioned upgrade strategy instead recognises that servers can be heterogeneous and partitions servers into groups based on inferred and supplied upgrade constraints. These constraints consist of lifecycle requirements, allocated services, upgrade tolerances, and provisioning surge.

5 Implementation

In this section, we describe a reference implementation of the partitioned upgrade strategy, and in particular its partitioning steps. The reference implementation is written in Go so knowledge thereof is assumed. While the central pieces of the implementation have been included in this section and in appendix A.1, the full code is available online at `https://github.com/kasperisager/partitionist/`.

The implementation consists of the following components:

- A representation of infrastructure
- A representation of upgrade plans
- A partitioner that optimises upgrade plans

When depicting code in this section, we will often make use of the ellipses in order to highlight fragments of longer functions. When breaking functions into fragments,

we will include all statements leading to the corresponding scope to allow the reader to keep track of the location of the fragment within the function:

```
1 func foo() bool {
2         // ...                 Excluded fragment
3         if condition {
4                 // ...         Excluded fragment
5                 return true // Current fragment
6         }
7         // ...                 Excluded fragment
8 }
```

5.1 Representations

In the following, we describe the data structures used for representing infrastructure and upgrades. These representations can be serialised to and from JSON in order to enable integration with external tools that can interact with concrete infrastructure.

5.1.1 Infrastructure

In the implementation, we define an abstract representation of infrastructure for describing the servers and services that make up the infrastructure.

In order to verify various invariants regarding infrastructure descriptions, such as upgrade tolerances not exceeding the number of servers to which a given service has been allocated, all structures described below define a `Validate()` method. Validations cascade, so validating for example a server would also validate its allocated services and their associated state.

State State is represented by three flags that indicate whether the state is internal or external, replicated or non-replicated, and concurrent or non-concurrent. These properties can either be inferred or identified automatically by looking at for example database configuration or the file systems table, or they can be supplied by an operator.

Services A service is represented by a unique identifier for identifying instances of the service across servers, an upgrade tolerance, and optional state. While the upgrade tolerance must be supplied by an operator, the identifier and state can often be inferred or identified automatically if, for example, a service scheduler such as [Nomad, nd] is used within the infrastructure.

Servers A server is represented by a unique identifier for identifying servers when integrating with external tools, a list of service allocations, and a weight. Since a many-to-many relation exists between servers and services, allocations are used as an associative entity. A map from services to instance counts was initially used, but this did

not allow for simple serialisation to JSON as keys must be strings. The services run on a server can often be identified automatically if, for example, service discovery such as [Consul, nd] is used within the infrastructure.

Clusters A cluster is represented by a list of servers and a maximum allowed provisioning surge. If no limit to provisioning surge applies, it can be set to the number of servers in the cluster. The servers within a cluster can often be identified automatically if, for example, infrastructure provisioning tools such as [Terraform, nd] are used.

5.1.2 Upgrades

In addition to the representation of infrastructure, we also define a representation of upgrade plans that can be used for orchestrating infrastructure. An upgrade plan is represented by a list of groups that each contain a list of servers and the lifecycle requirements of those. Since servers are partitioned by lifecycle, the servers within a group will have the same lifecycle requirements.

An example of an orchestrator that processes groups in parallel and servers in sequence would in Go look something like the following, sans goroutine synchronisation boilerplate:

```
func process(group Group) {
        for _, server := range group.Servers {
                if group.Lifecycle.CreateBeforeDestroy {
                        // ...
                } else {
                        // ...
                }
        }
}

func orchestrate(plan Plan) {
        for _, group := range plan.Groups {
                go process(group)
        }
}
```

A more complete orchestrator abstraction is found in appendix A.1.3. This abstraction performs the heavy lifting of processing upgrade plans and delegates the responsibility of disposing and rebuilding servers to clients through an interface.

As the infrastructure and upgrade representations are serialisable, a serialised version of the upgrade plan contains everything needed to perform the upgrade. As mentioned previously, this makes it possible to integrate with external tools that can then orchestrate the concrete infrastructure. As might by now be apparent, we lean towards the infrastructure management tools provided in the [HashiCorp, nd] suite, but our implementation is in no way limited to these as we will later see.

5.2 Partitioning servers

Partitioning by lifecycle To partition a group of servers by lifecycle, we split it into two groups with the servers that can be built before the old ones are disposed going into the first group and the servers that must be disposed before being rebuilt going into the second group.

Partitioning by services To partition a group of servers by the services they run, we make use of a union-find data structure as described in [Sedgewick and Wayne, 2011, p. 216]. Our union-find implementation is based on the one provided in [Sedgewick and Wayne, 2011, p. 221] and is available online at `https://github.com/kasperisager/union/`.

For every service of every server, we do the following. First, we check if the service has an associated *representative server*. If not, we mark the server of the service as the representative server for that service, otherwise we join the server of the service with the representative server in the union-find data structure. Finally, for every server we find its root within the union-find data structure and add it to the group corresponding to the root.

Distributing by weight The problem of evenly distributing servers across group according to their weights is one closely related to the problem of multiprocessor scheduling as described in [Silberschatz et al., 2014, p. 278]. As a generalisation of this problem, [Lee, 1991] considers the problem of scheduling a number of independent jobs across a number of identical machines in order to minimise the total finishing time. To this end, Lee first describes the *longest processing time algorithm*, referred to as LPT, and applies it to the generalised problem. Lee then proposes a modified LPT algorithm with improved bounds on an optimal solution. The basic LPT algorithm should however be a good enough fit for our problem of distributing servers across groups according to their weights, so that is what we will use.

The LPT algorithm is simple to implement and consists of two steps. In the first step, we sort the list of servers by weight in decreasing order:

```
1  import "sort"
2
3  func sortByWeight(servers []Server) {
4          sort.Slice(servers, func(i int, j int) bool {
5                  return servers[i].Weight >= servers[j].Weight
6          })
7  }
```

As the only ordering we define for servers is by weight, an unstable sort will suffice, hence the use of `sort.Slice()` over `sort.SliceStable()`. Once sorted, we assign each server, starting from the beginning of the sorted list, to the group with the lowest

weight. Whenever a server is assigned to a group, the total weight of the group is increased by that of the server.

In order to make the assignment step as efficient as possible, we make use of a heap-based priority queue as described in [Sedgewick and Wayne, 2011, p. 313]. We also implement indexing in order to support partitioning by upgrade tolerance where a server can only be put into a subset of the available groups. Our priority queue implementation is based on the one provided in [Sedgewick and Wayne, 2011, p. 333] and is available online at `https://github.com/kasperisager/pqueue/`.

The assignment step varies somewhat between partitioning by surge and by upgrade tolerance, though the following function is provided for assigning a server to a specific group:

```
import pqueue "github.com/kasperisager/pqueue"

func assignToGroup(queue pqueue.PriorityQueue, groups []Group, server Server, index
 ↪  int) {
        weight, _ := queue.Priority(index)
        group := &groups[index]

        group.Servers = append(group.Servers, server)
        queue.Update(index, weight+server.Weight+1)
}
```

Since we use weights starting at 0, we add 1 to the weight of the server to ensure that the total weight of the group increases by a non-zero amount.

Partitioning by upgrade tolerance To partition a group of servers by upgrade tolerance, we make use of the LPT algorithm as described previously. First, we sort the servers in decreasing order by weight. Next, we determine how many new groups to create by locating the highest upgrade tolerance and ensuring that it does not exceed the number of servers in the group to avoid ending up with more groups than needed. Finally, the servers are partitioned into groups according to their upgrade tolerances. This last bit is worth taking a closer look at, as this is where the indexed priority queue shines:

```
func (g Group) partitionByTolerance(cluster Cluster) []Group {
        // ...
        for _, server := range g.Servers {
                // ...
                var index int

                if j, _ := queue.Peek(); j <= t {
                        index = j
                } else {
                        for j := 1; j < t; j++ {
```

```
                                pi, _ := queue.Priority(index)
                                pj, _ := queue.Priority(j)

                                if pi > pj {
                                        index = j
                                }
                        }
                }

                assignToGroup(queue, groups, server, index)
        }
        // ...
}
```

That is, if the head of the queue is within the upgrade tolerance of the server then we can add the server to the group at the head. Otherwise, we scan the queue within range of the upgrade tolerance to find the group with the lowest weight.

Partitioning by provisioning surge To partition a group of server by partitioning surge, we again make use of the LPT algorithm. We sort the servers and partition them into a number of groups equal to the provisioning surge, again ensuring that we do not create more groups than needed. When partitioning by provisioning surge, we can always add servers to the group at the head of the queue.

5.3 Demonstration

To demonstrate how the partitioned upgrade strategy functions and performs in practise, we have provided an implementation of the orchestrator interface from appendix A.1.3. The orchestrator integrates with the cloud provider DigitalOcean through their official CLI, `doctl`, and can be found in appendix A.2.1. We highlight the most important parts of the orchestrator in the following.

5.3.1 Orchestrating infrastructure

In demonstrating how to integrate with a concrete infrastructure provider, we have implemented an orchestrator that can orchestrate server instances, called droplets, in DigitalOcean. In accordance with the orchestrator interface, all orchestrators must provide `Rebuild(Server)` and `Dispose(Server)` methods. First, we provide some constants for use within these methods:

```
const (
        version = 1
        doctl   = "doctl"
        size    = "512mb"
        image   = "ubuntu-16-04-x64"
```

```
        region  = "ams2"
)
```

These specify the current infrastructure version, the name of the `doctl` binary, as well as the size, image, and region of the droplets to orchestrate. The method for rebuilding droplets looks like this:

```
func (orchestrator Orchestrator) Rebuild(server partitionist.Server) {
        old := fmt.Sprint(server.ID, "-v", version)
        new := fmt.Sprint(server.ID, "-v", version+1)

        log.Printf("rebuild %s => %s\n", old, new)

        rebuild := exec.Command(doctl, "compute", "droplet", "create", new,
                "--wait",
                "--size", size,
                "--image", image,
                "--region", region,
        )

        if err := rebuild.Run(); err != nil {
                log.Fatal(err)
        }
}
```

That is, given a server ID (`server-01`) we compute the slugs of the droplet to dispose and the droplet to rebuild (`server-01-v1` and `server-01-v2`). The infrastructure version number is prepended to the slug in order to avoid clashes when droplets are rebuilt before being disposed. Finally, a new droplet is created. The method for disposing droplets looks similar:

```
func (orchestrator Orchestrator) Dispose(server partitionist.Server) {
        old := fmt.Sprint(server.ID, "-v", version)

        log.Printf("dispose %s\n", old)

        dispose := exec.Command(doctl, "compute", "droplet", "delete", old,
                "--force",
        )

        if err := dispose.Run(); err != nil {
                log.Fatal(err)
        }
}
```

Here, we only need the slug of the droplet to dispose which is then used for deleting the droplet. With these methods in place, we have an implementation of the orchestrator interface from appendix A.1.3. In this appendix, a method is also provided for orchestrating an upgrade plan based on an orchestrator implementation. This method

takes care of invoking the rebuild and dispose methods in the correct order for each server in a cluster. With our DigitalOcean integration in hand, we can therefore use this method along with an upgrade plan to orchestrate actual server instances on DigitalOcean.

5.3.2 Evolving infrastructure

We now take on the role of an imaginary software company and perform mock upgrades of a gradually evolving infrastructure configuration. While the implemented orchestrator does dispose of and rebuild actual server instances, nothing changes between the old and new servers as both use the same basic Ubuntu image. As immutable servers are typically created from pre-provisioned server images, there will be little to no difference in the time taken to create a server with a basic Ubuntu image and pre-provisioned one. If pre-provisioned images are *not* used, or expensive bootstrap procedures are performed, servers can, as mentioned previously, be assigned weights in order to account for this.

For each configuration, we will be faced with the choice of using either blue-green or rolling upgrades when upgrading servers. We will then compare the upgrade time of the chosen strategy with that of a partitioned upgrade in order to evaluate the efficiency of the two. While we provide descriptions of the different infrastructure configurations, their associated code can be found in appendix A.2.5.

First configuration The first infrastructure configuration we consider is one consisting of six servers each running the same service: A stateless application. When designing the infrastructure, we wanted to be able to perform efficient blue-green upgrades rather than rolling upgrades, so all state was delegated to a managed database service outside our infrastructure.

Figure 9: Total upgrade times in seconds for the first infrastructure configuration.

As seen in figure 9, the partitioned upgrade strategy is just as efficient as the blue-green upgrade in this configuration. Without providing anything other than the description of the infrastructure, the partitioned upgrade correctly infers that all servers can be rebuild before being disposed and, since no limit on provisioning surge is specified, can therefore be upgraded in parallel.

Second configuration By now, we have realised that using a managed database service is too expensive for the needs of our application. We therefore decommission

two application servers and instead set up two database servers, one write-master and one read-slave, and have them persist state to shared external block storage. If the write-master fails, we can promote the read-slave to write-master while the failed write-master recovers.

We realise that blue-green upgrades can now no longer be performed as the database service uses external, non-concurrent, non-replicated state. We can split the servers into two groups and perform a blue-green upgrade of the application group and a rolling upgrade of the database group. We however also realise that this manual work will not scale well once the complexity of our infrastructure increases. We instead turn to rolling upgrades and must upgrade our infrastructure one server at a time in order to keep a database write-master available during the upgrade.

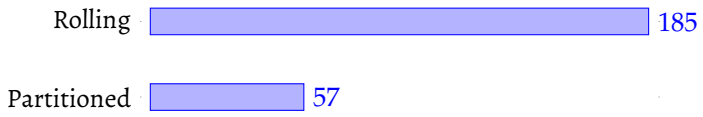


Figure 10: Total upgrade times in seconds for the second infrastructure configuration.

As seen in figure 10, the partitioned upgrade strategy by far outperforms the rolling upgrade strategy in this configuration. This due to the fact that the partitioned upgrade is able to separate the application and database servers for us and upgrade them separately. While the complexity involved in manually upgrading the application and database servers separately is minimal, the partitioned upgrade will scale better as it can perform this separation automatically.

Third configuration After having run our own database servers for some time, we realise that two dedicated servers is more capacity than needed. To increase fault tolerance and better utilise our server capacity, we use three servers for running co-located instances of both the application and database services and keep the remaining three servers dedicated to the application service. We can now perform the rolling upgrade with two servers at a time to keep a database write-master available.

Figure 11: Total upgrade times in seconds for the third infrastructure configuration.

As seen in figure 11, the time taken to perform rolling upgrades has improved but still does not match that of the partitioned upgrade. The partitioned upgrade still has to sequentially upgrade two of the servers running databases, so its upgrade time remains the same as with the previous configuration.

Fourth configuration The infrastructure is beginning to look production ready. Services are run in a fault tolerant manner, server capacity utilisation is optimised, and we have full control of our databases. To keep our operations team happy, we decide to add metrics collectors to two of the servers dedicated to the application service. While one would be enough for capacity purposes, we run two in order to achieve a minimum level of fault tolerance. State is persisted to shared external block storage with one metrics collector active at any given time and the other on stand-by in case of failure. In order to keep at least one metrics collector available during an upgrade, we must again perform rolling upgrades one server at a time.

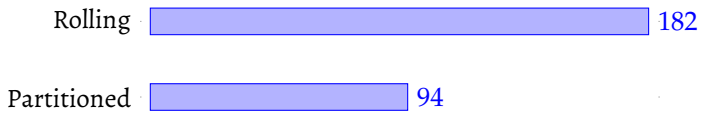


Figure 12: Total upgrade times in seconds for the fourth infrastructure configuration.

As seen in figure 12, the performance gains won for the rolling upgrade in the third configuration are now gone. The partitioned upgrade also takes longer to perform, but is now twice as fast as the rolling upgrade.

5.3.3 Lessons learned

The different infrastructure configurations presented above do use to limitations of the rolling upgrade against it in order to demonstrate the efficiency of the partitioned upgrade. The configurations are however not unreasonable; modern job schedulers allow intelligent packing of services onto servers in order to best utilise server capacity, making co-located services the norm. The partitioned upgrade recognises and embraces this, allowing it to efficiently upgrade even complex infrastructure configurations.

The infrastructure configurations we considered were also very small ones, consisting only of six servers. While the partitioned upgrade was able to achieve upgrade times up to three times faster than the rolling upgrade even on such a small infrastructure, we expect to see larger improvements as the size of the infrastructure increases. If, for example, the number of application servers in the second configuration was much higher, the partitioned upgrade strategy would be unaffected by this as it would recognise that all of these servers could be upgraded in parallel. In this case, the rolling upgrade would still have to upgrade servers one at a time in order to keep a database write-master available.

6 Limitations

While the partitioned upgrade strategy does improve upon the rolling upgrade strategy and can match the efficiency of the blue-green upgrade strategy when possible, it does introduce additional limitations and fails to address some existing ones. In the following, we discuss these limitations and how some of them can be addressed outside the upgrade strategy.

6.1 Long-tail

When performing partitioned upgrades, we can weight servers according to the time taken to perform disposals and rebuilds of a given server, allowing us to evenly distribute processing time across groups in the upgrade plan. While this in turn allows us to account for servers that we anticipate will have increased disposal and rebuild times, the partitioned upgrade does not provide us with a way of handling unanticipated increases thereof.

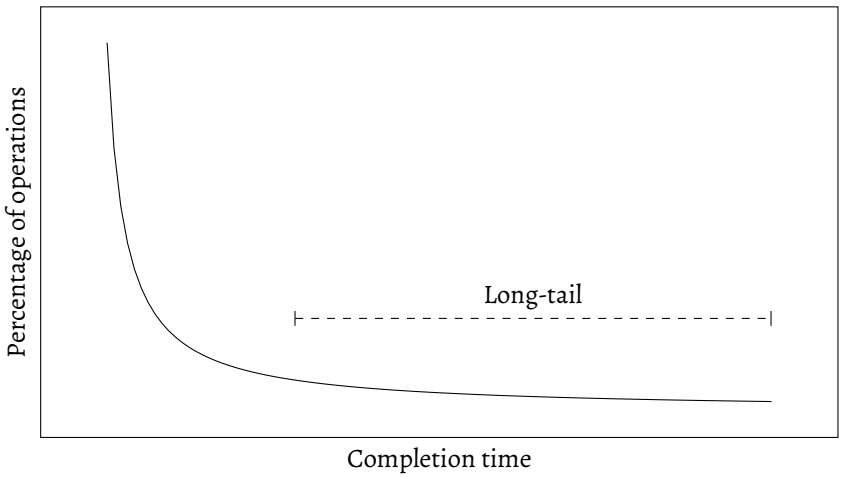


Figure 13: Illustration of long-tail in the distribution of completion times of API operations.

As shown in [Lu et al., 2014], cloud provider APIs are susceptible to various failures, causing a *long-tail* in the distribution of completion times of API operations as illustrated in figure 13. This long-tail is not something the partitioned upgrade strategy is able to address directly due to the way servers are partitioned into independent groups. If, for example, a server rebuild fails within a group after the server is disposed, we are not able to proceed with the next server in the group as this might violate an upgrade tolerance constraint.

[Lu et al., 2014] does however introduce several tail-tolerant mechanisms that can be used for tolerating API long-tail. Among these mechanisms are request patterns

that can be implemented around cloud provider APIs to effectively remove the long-tail. While the partitioned upgrade strategy itself is not able to address the long-tail, orchestrator implementations, such as the one provided in appendix A.2.1, can make use of these tail-tolerant mechanisms to remove the long-tail.

6.2 Service dependencies

Dependencies between services, i.e. the requirement that some services be upgraded before others, is not addressed by the partitioned upgrade strategy. Service dependencies can occur when for example relational databases are part of the infrastructure. If schema upgrades are to be performed, these must often happen before the application services that are to make use of the upgraded schema are themselves upgraded.

While the partitioned upgrade strategy can be extended in the future with an additional partitioning step that addresses service dependencies, we will for now refer to the work done in [Wolski and Laiho, 2004] on the *upgrade food chain*. The upgrade food chain is a directed graph used for capturing code, data, and schema dependencies of services. One way of partitioning servers according to service dependencies is similar to that of partitioning by services. Instead of using union-find to determine server connectivity based on services, we can instead use depth-first search as described in [Sedgewick and Wayne, 2011, p. 543] to determine server connectivity based on the upgrade food chain.

6.3 Allocation counts

Another limitation of the partitioned upgrade strategy is that service allocation counts are not yet considered. Since a server might run multiple instances of the same service, it is beneficial to ensure that as few service instances as possible are unavailable during an upgrade.

As an example, consider 3 servers running 8, 6, and 2 instances respectively of the same service with an upgrade tolerance of 2. If the servers running 8 and 6 instances are partitioned into separate groups, we risk making 14 out of the 16 instances unavailable during the upgrade. It would be better if the servers running 8 and 6 instances were partitioned into the same group and leaving the server running 2 instances in a group of its own. This way, at most 10 rather than 14 instances would be unavailable during the upgrade.

In order to address this in the partitioned upgrade strategy, we would need to distribute servers across groups in such a way that the sum of the maximum allocation counts of a given service across groups is minimised. Consider the first case of the previous example with 8 and 2 instances in the first group and 6 instances in the second group. In this case, the maximum allocation count of the first group is 8 while it for the second group is 6. This results in a sum of 14, i.e. the number of service instances

we risk making unavailable during the upgrade. In the second case, the maximum allocation count of the first group is once again 8, but is now 2 for the second group, resulting in a sum of 10.

We leave open the problem of addressing service allocation counts as it might be better suited for those familiar with *integer programming*.

6.4 Backwards compatibility

As mentioned previously in section 4.1, the partitioned upgrade strategy assumes that only backwards compatible upgrades are performed. Here, we define backwards compatibility in terms of compatibility between the services that are to be upgraded. This due to the fact that multiple versions of the same services might co-exist for the duration of the upgrade. As also mentioned previously, we believe that constraining upgrades to be backwards compatible affords us enough flexibility to be justifiable.

If backwards incompatible upgrades were to be addressed by the partitioned upgrade strategy, it would end up being no different than the blue-green upgrade strategy as backwards incompatible changes require isolation of the old and new servers.

7 Conclusion

Over the course of this thesis we have outlined the background for immutable infrastructure, described existing strategies for upgrading immutable servers, identified and analysed the limitations of these strategies, and finally designed and implemented an upgrade strategy, the partitioned upgrade, that addresses these limitations. Unlike existing strategies, the partitioned upgrade strategy embraces modern infrastructures where job schedulers are used for intelligently packing services onto servers, often resulting in co-located services. The partitioned upgrade strategy is agnostic to both this as well as state employed by the services and, as demonstrated, is able to perform upgrades as fast as the blue-green upgrade strategy and up to three times faster than the rolling upgrade strategy.

Since the partitioned upgrade strategy is in essence a hybrid of the existing strategies, it is able to achieve the versatility of the rolling upgrade strategy while also aiming for the efficiency of the blue-green upgrade strategy. A drawback of the partitioned upgrade strategy is its inability to deal with backwards incompatible upgrades, which the blue-green upgrade strategy addresses. Other limitations, such as not considering service allocation counts and dependencies, also remain unaddressed though these limitations can be addressed in the future.

The versatility and efficiency of the partitioned upgrade strategy can be of great benefit to operations teams operating large-scale infrastructures. The versatility allows a single strategy to be used across the infrastructure while the efficiency allows

releases to happen more often as upgrades can be performed faster. The declarative approach of the strategy – describe the infrastructure and let the strategy infer the constraints – also aligns well with the way modern infrastructures are often declaratively defined in code.

In the end, it is our hope that the partitioned upgrade strategy can find a use alongside infrastructure provisioning tools. While the integration we have provided with DigitalOcean is a first step in the right direction, it is only meant as a means of showing what is possible with our orchestrator abstraction. Future work will focus on providing integrations with the tools provided in the [HashiCorp, nd] suite such that as many aspects as possible of the partitioned upgrade strategy can be automated.

List of Figures

1 A mutable upgrade of a service from version “1” to “2”. 4
2 An immutable upgrade of a service from version “1” to “2”. 4
3 A master/slave setup employing both concurrent and non-concurrent external state. 7
4 The first step in a rolling upgrade of 4 servers with a group size of 2. . 9
5 A blue-green upgrade of 4 servers. 10
6 The first step in a partitioned upgrade of 4 servers. 12
7 Partitioning of 5 servers according to the services they run. 13
8 Partitioning of 3 servers from group 1 of figure 7 according to upgrade tolerances listed in parentheses. Note that service “A” will be unavailable during this upgrade. 14
9 Total upgrade times in seconds for the first infrastructure configuration. 22
10 Total upgrade times in seconds for the second infrastructure configuration. 23
11 Total upgrade times in seconds for the third infrastructure configuration. 23
12 Total upgrade times in seconds for the fourth infrastructure configuration. 24
13 Illustration of long-tail in the distribution of completion times of API operations. 25

References

[Ajmani et al., 2003] Ajmani, S., Liskov, B., and Shrira, L. (2003). Scheduling and Simulation: How to Upgrade Distributed Systems. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 43–48. USENIX.

[CephFS, nd] CephFS (n.d.). Ceph Block Storage. `http://ceph.com/ceph-storage/block-storage/`.

[Consul, nd] Consul (n.d.). Service discovery and configuration made easy. `https://www.consul.io/`.

[Coulouris et al., 2011] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley, 5th edition.

[Docker, nd] Docker (n.d.). Manage data in containers. `https://docs.docker.com/engine/tutorials/dockervolumes/`.

[Dunn, 2014] Dunn, J. (2014). Immutable Infrastructure: Practical or Not? `https://blog.chef.io/2014/06/23/immutable-infrastructure-practical-or-not/`.

[Fowler, 2013] Fowler, C. (2013). Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components. `http://chadfowler.com/2013/06/23/immutable-deployments.html`.

[HashiCorp, nd] HashiCorp (n.d.). Devops defined: Accelerating the application delivery lifecycle. `https://www.hashicorp.com/devops-defined/`.

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 1st edition.

[Lee, 1991] Lee, C.-Y. (1991). Parallel machines scheduling with nonsimultaneous machine available time. *Discrete Applied Mathematics*, 30(1):53–61.

[Locksmith, nd] Locksmith (n.d.). Reboot manager for Container Linux. `https://github.com/coreos/locksmith`.

[Lu et al., 2014] Lu, Q., Zhu, L., Xu, X., Bass, L., Li, S., Zhang, W., and Wang, N. (2014). Mechanisms and Architectures for Tail-Tolerant System Operations in Cloud. In *HotCloud*.

[Moreno-Vozmediano et al., 2012] Moreno-Vozmediano, R., Montero, R. S., and Llorente, I. M. (2012). Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures. *Computer*, 45(12):65–72.

[Morris, 2013] Morris, K. (2013). Immutable Server. https://martinfowler.com/bliki/ImmutableServer.html.

[Motlik, 2014] Motlik, F. (2014). Why You Should Build an Immutable Infrastructure. https://blog.codeship.com/immutable-infrastructure/.

[Nomad, nd] Nomad (n.d.). Distributed, highly available, datacenter-aware scheduler. https://www.nomadproject.io/.

[Pais, 2014] Pais, M. (2014). Virtual Panel on Immutable Infrastructure. https://www.infoq.com/articles/virtual-panel-immutable-infrastructure.

[rkt, nd] rkt (n.d.). Mount Volumes into a Pod. https://coreos.com/rkt/docs/latest/subcommands/run.html.

[Sedgewick and Wayne, 2011] Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-Wesley, 4th edition.

[Silberschatz et al., 2014] Silberschatz, A., Galvin, P. B., and Gagne, G. (2014). *Operating System Concepts*. Wiley, 9th edition.

[Stella, 2016] Stella, J. (2016). Immutable Infrastructure: Considerations for the Cloud and Distributed Systems. https://fugue.co/assets/docs/Immutable_Infrastructure_Fugue.pdf.

[Terraform, nd] Terraform (n.d.). Write, plan, and create infrastructure as code. https://www.terraform.io/.

[Wolski and Laiho, 2004] Wolski, A. and Laiho, K. (2004). Rolling upgrades for continuous services. In *International Service Availability Symposium*, pages 175–189. Springer.

Appendices

A Source code

All code listed here is provided under the MIT license while Partitionist as a whole is licensed under the GPLv3. This due to the fact that the priority queue and union-find implementations are themselves licensed under the GPLv3 in compliance with the license requirements of [Sedgewick and Wayne, 2011].

A.1 Partitionist

A.1.1 `infrastructure.go`

```
// Copyright (C) 2017 Kasper Kronborg Isager
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
package partitionist

import "fmt"

type State struct {
        // Whether or not the state is concurrent, i.e. can be accessed by multiple
        // servers simultaneously.
        Concurrent bool `json:"concurrent"`

        // Whether the state is internal or external to the server. Internal state is
        // state that is destroyed alongside the server on which the state lives when
        // the server itself is destroyed whereas external state can survive
        // destruction of the server.
        External bool `json:"external"`

        // Whether the state is replicated to more than one server. This is used for
```

```
        // validating whether servers with internal state can actually be upgraded
        // without discarding their associated state.
        Replicated bool `json:"replicated"`
}

type Service struct {
        // An ID for uniquely identifying instances of the service across servers.
        // This is also used as the partition key when partitioning servers during
        // deployment planning.
        ID string `json:"id"`

        // The availability tolerance of the service, that is the number of servers
        // running the service that are allowed to be unavailable at any given moment
        // during a deployment. Tolerances must be greater than zero.
        Tolerance int `json:"tolerance"`

        // The optional state associated with the service. This is used for inferring
        // the lifecycle of the servers onto which the service will run.
        State *State `json:"state"`
}

type Allocation struct {
        // The service being allocated to the associated server.
        Service Service `json:"service"`

        // The number of instances of the service allocated to the associated server.
        // Instance counts must be greater than zero.
        Instances int `json:"instances"`
}

type Server struct {
        // An ID for uniquely identifying the server. This is primarily meant as a
        // means of letting external tools identify servers in upgrade plans.
        ID string `json:"id"`

        // The service allocations for the server.
        Allocations []Allocation `json:"allocations"`

        // The relative weight of the server. Weights are used for distributing
        // servers evenly across upgrade steps. Weights must be greater than or equal
        // to zero.
        Weight float32 `json:"weight"`
}

type Cluster struct {
        Servers []Server `json:"servers"`
        Surge   int      `json:"surge"`
}

type Lifecycle struct {
```

```
        RebuildBeforeDispose bool `json:"rebuildBeforeDispose"`
}

// Validate checks that a state is legal.
func (state State) Validate() error {
        if !state.External && !state.Replicated {
                return fmt.Errorf("Internal state must be replicated")
        }

        return nil
}

// Validate checks that a service is legal.
func (service Service) Validate() error {
        if service.Tolerance <= 0 {
                return fmt.Errorf(
                        "%s: Tolerance (%d) must be greater than 0",
                        service.ID,
                        service.Tolerance,
                )
        }

        if service.State != nil {
                if err := service.State.Validate(); err != nil {
                        return err
                }
        }

        return nil
}

// Validate checks that an allocation is legal.
func (allocation Allocation) Validate() error {
        if allocation.Instances <= 0 {
                return fmt.Errorf(
                        "%s: Number of instances (%d) must be greater than 0",
                        allocation.Service.ID,
                        allocation.Instances,
                )
        }

        if err := allocation.Service.Validate(); err != nil {
                return err
        }

        return nil
}

// Validate checks that a server is legal.
func (server Server) Validate() error {
```

```
136         if server.Weight < 0 {
137                 return fmt.Errorf(
138                         "%s: Weight (%d) must be greater than or equal to 0",
139                         server.ID,
140                         server.Weight,
141                 )
142         }
143
144         for _, allocation := range server.Allocations {
145                 if err := allocation.Validate(); err != nil {
146                         return err
147                 }
148         }
149
150         return nil
151 }
152
153 // Validate checks that a cluster is legal.
154 func (cluster Cluster) Validate() error {
155         if cluster.Surge <= 0 {
156                 return fmt.Errorf("Surge (%d) must be greater than 0", cluster.Surge)
157         }
158
159         // Keep track of the total number of allocations of each service, that is the
160         // number of servers containing one or more instances of each service.
161         allocations := make(map[string]int)
162
163         for _, server := range cluster.Servers {
164                 if err := server.Validate(); err != nil {
165                         return err
166                 }
167
168                 for _, allocation := range server.Allocations {
169                         allocations[allocation.Service.ID]++
170                 }
171         }
172
173         for _, server := range cluster.Servers {
174                 for _, allocation := range server.Allocations {
175                         service := allocation.Service
176                         allocated := allocations[service.ID]
177
178                         if service.Tolerance >= allocated {
179                                 return fmt.Errorf(
180                                         "%s: Not enough allocations (%d) to satisfy
    ↪  tolerance (%d)",
181                                         service.ID,
182                                         allocated,
183                                         service.Tolerance,
184                                 )
```

```
185                         }
186                 }
187         }
188
189         return nil
190 }
191
192 // Lifecycle computes the lifecycle of a server.
193 func (server Server) Lifecycle() Lifecycle {
194         lifecycle := Lifecycle{
195                 RebuildBeforeDispose: true,
196         }
197
198         for _, allocation := range server.Allocations {
199                 state := allocation.Service.State
200
201                 if state == nil {
202                         continue;
203                 }
204
205                 // In case of external, nonconcurrent, nonreplicated state, the new
    ↪   server
206                 // cannot be built before the old one is diposed as the two servers
    ↪   cannot
207                 // access the same external state simultaneously.
208                 if state.External && !state.Replicated && !state.Concurrent {
209                         lifecycle.RebuildBeforeDispose = false
210                 }
211         }
212
213         return lifecycle
214 }
215
216 // Tolerance computes the tolerance of a server.
217 func (server Server) Tolerance() (int, bool) {
218         var tolerance int
219
220         if len(server.Allocations) == 0 {
221                 return tolerance, false
222         }
223
224         for _, allocation := range server.Allocations {
225                 service := allocation.Service
226
227                 if tolerance == 0 || service.Tolerance < tolerance {
228                         tolerance = service.Tolerance
229                 }
230         }
231
232         return tolerance, true
```

```
}

// Tolerance computes the maximum tolerance of a list of servers.
func MaxTolerance(servers []Server) int {
	length := len(servers)
	tolerance := 0

	for _, server := range servers {
		t, ok := server.Tolerance()

		if ok && t > tolerance {
			tolerance = t
		}
	}

	if tolerance == 0 || length < tolerance {
		tolerance = length
	}

	return tolerance
}

// Tolerance computes the minimum tolerance of a list of servers.
func MinTolerance(servers []Server) int {
	length := len(servers)
	tolerance := length

	for _, server := range servers {
		t, ok := server.Tolerance()

		if ok && t < tolerance {
			tolerance = t
		}
	}

	return tolerance
}

// Instances computes the total number of instances of all services allocated
// to a server.
func (server Server) Instances() int {
	var count int

	for _, allocation := range server.Allocations {
		count += allocation.Instances
	}

	return count
}

```

```
283 // InstancesOf computes the total number of a instances of a specific service
284 // allocated to a server.
285 func (server Server) InstancesOf(service Service) int {
286         for _, allocation := range server.Allocations {
287                 if allocation.Service.ID == service.ID {
288                         return allocation.Instances
289                 }
290         }
291
292         return 0
293 }
```

A.1.2 deployment.go

```
1  // Copyright (C) 2017 Kasper Kronborg Isager
2  //
3  // Permission is hereby granted, free of charge, to any person obtaining a copy
4  // of this software and associated documentation files (the "Software"), to deal
5  // in the Software without restriction, including without limitation the rights
6  // to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
7  // copies of the Software, and to permit persons to whom the Software is
8  // furnished to do so, subject to the following conditions:
9  //
10 // The above copyright notice and this permission notice shall be included in
11 // all copies or substantial portions of the Software.
12 //
13 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
14 // IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
15 // FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
16 // AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
17 // LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
18 // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
19 // SOFTWARE.
20 package partitionist
21
22 import (
23         "sort"
24
25         pqueue "github.com/kasperisager/pqueue"
26         union "github.com/kasperisager/union"
27 )
28
29 type Plan struct {
30         Groups []Group `json:"groups"`
31 }
32
33 type Group struct {
34         Servers   []Server  `json:"servers"`
35         Lifecycle Lifecycle `json:"lifecycle"`
36 }
```

```
37
38  // Partition creates a partitioned upgrade plan for a cluster.
39  func (cluster Cluster) Partition() (*Plan, error) {
40          if err := cluster.Validate(); err != nil {
41                  return nil, err
42          }
43
44          return &Plan{
45                  Group{Servers: cluster.Servers}.partitionByLifecycle(cluster),
46          }, nil
47  }
48
49  func sortByWeight(servers []Server) {
50          sort.Slice(servers, func(i int, j int) bool {
51                  return servers[i].Weight >= servers[j].Weight
52          })
53  }
54
55  func assignToGroup(queue pqueue.PriorityQueue, groups []Group, server Server, index
     ↪  int) {
56          weight, _ := queue.Priority(index)
57          group := &groups[index]
58
59          group.Servers = append(group.Servers, server)
60          queue.Update(index, weight+server.Weight+1)
61  }
62
63  func (g Group) partitionByLifecycle(cluster Cluster) []Group {
64          length := len(g.Servers)
65
66          if length == 0 {
67                  return []Group{}
68          }
69
70          fst := Group{Lifecycle: Lifecycle{true}}
71          snd := Group{}
72
73          for _, server := range g.Servers {
74                  lifecycle := server.Lifecycle()
75
76                  if lifecycle.RebuildBeforeDispose {
77                          fst.Servers = append(fst.Servers, server)
78                  } else {
79                          snd.Servers = append(snd.Servers, server)
80                  }
81          }
82
83          return append(
84                  fst.partitionBySurge(cluster),
85                  snd.partitionByService(cluster)...,
```

```
        )
}

func (group Group) partitionByService(cluster Cluster) []Group {
        length := len(group.Servers)

        if length == 0 {
                return []Group{}
        }

        representatives := make(map[string]int)
        components := union.New()

        // First pass: Join servers in groups according to their associated services
        // such that servers running the same services will be grouped. This ensures
        // that the resulting groups can be treated indenpently as no two groups will
        // contain servers running the same services.
        for i, server := range group.Servers {
                for _, allocation := range server.Allocations {
                        service := allocation.Service

                        if j, ok := representatives[service.ID]; ok {
                                components.Join(i, j)
                        } else {
                                representatives[service.ID] = i
                        }
                }
        }

        parts := make(map[int][]Server)

        // Second pass: Partition the servers into their associated groups.
        for i, server := range group.Servers {
                j := components.Find(i)
                parts[j] = append(parts[j], server)
        }

        groups := make([]Group, 0, len(parts))

        for _, servers := range parts {
                groups = append(
                        groups,
                        Group{servers,
 ↪  group.Lifecycle}.partitionByTolerance(cluster)...,
                )
        }

        return groups
}

```

```
func (group Group) partitionBySurge(cluster Cluster) []Group {
        length := len(group.Servers)

        if length == 0 {
                return []Group{}
        }

        sortByWeight(group.Servers)

        surge := cluster.Surge

        if length < surge {
                surge = length
        }

        queue := pqueue.New(pqueue.Ascending)
        groups := make([]Group, surge)

        for i := range groups {
                queue.Push(i, 0)
                groups[i].Lifecycle = group.Lifecycle
        }

        for _, server := range group.Servers {
                index, _ := queue.Peek()
                assignToGroup(queue, groups, server, index)
        }

        return groups
}

func (group Group) partitionByTolerance(cluster Cluster) []Group {
        length := len(group.Servers)

        if length == 0 {
                return []Group{}
        }

        sortByWeight(group.Servers)

        tolerance := MaxTolerance(group.Servers)
        queue := pqueue.New(pqueue.Ascending)
        groups := make([]Group, tolerance)

        for i := range groups {
                queue.Push(i, 0)
                groups[i].Lifecycle = group.Lifecycle
        }

        for _, server := range group.Servers {
```

```
185                 t, ok := server.Tolerance()
186
187                 if !ok {
188                         t = tolerance
189                 }
190
191                 var index int
192
193                 if j, _ := queue.Peek(); j <= t {
194                         index = j
195                 } else {
196                         for j := 1; j < t; j++ {
197                                 pi, _ := queue.Priority(index)
198                                 pj, _ := queue.Priority(j)
199
200                                 if pi > pj {
201                                         index = j
202                                 }
203                         }
204                 }
205
206                 assignToGroup(queue, groups, server, index)
207         }
208
209         return groups
210 }
```

A.1.3 orchestrate.go

```
1  // Copyright (C) 2017 Kasper Kronborg Isager
2  //
3  // Permission is hereby granted, free of charge, to any person obtaining a copy
4  // of this software and associated documentation files (the "Software"), to deal
5  // in the Software without restriction, including without limitation the rights
6  // to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
7  // copies of the Software, and to permit persons to whom the Software is
8  // furnished to do so, subject to the following conditions:
9  //
10 // The above copyright notice and this permission notice shall be included in
11 // all copies or substantial portions of the Software.
12 //
13 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
14 // IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
15 // FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
16 // AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
17 // LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
18 // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
19 // SOFTWARE.
20 package partitionist
21
```

```
import "sync"

type Orchestrator interface {
        Rebuild(server Server)
        Dispose(server Server)
}

func Orchestrate(orchestrator Orchestrator, plan Plan) {
        var wait sync.WaitGroup
        defer wait.Wait()

        wait.Add(len(plan.Groups))

        for _, group := range plan.Groups {
                go func(group Group) {
                        defer wait.Done()

                        for _, server := range group.Servers {
                                if group.Lifecycle.RebuildBeforeDispose {
                                        orchestrator.Rebuild(server)
                                        orchestrator.Dispose(server)
                                } else {
                                        orchestrator.Dispose(server)
                                        orchestrator.Rebuild(server)
                                }
                        }
                }(group)
        }
}
```

A.2 Demonstration

A.2.1 digitalocean.go

```
// Copyright (C) 2017 Kasper Kronborg Isager
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
```

```
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
package digitalocean

import (
        "fmt"
        "log"
        "os/exec"

        partitionist "github.com/kasperisager/partitionist"
)

const (
        version = 4
        doctl   = "doctl"
        size    = "512mb"
        image   = "ubuntu-16-04-x64"
        region  = "ams2"
)

type Orchestrator struct{}

func (orchestrator Orchestrator) Rebuild(server partitionist.Server) {
        old := fmt.Sprint(server.ID, "-v", version)
        new := fmt.Sprint(server.ID, "-v", version+1)

        log.Printf("rebuild %s => %s\n", old, new)

        rebuild := exec.Command(doctl, "compute", "droplet", "create", new,
                "--wait",
                "--size", size,
                "--image", image,
                "--region", region,
        )

        if err := rebuild.Run(); err != nil {
                log.Fatal(err)
        }
}

func (orchestrator Orchestrator) Dispose(server partitionist.Server) {
        old := fmt.Sprint(server.ID, "-v", version)

        log.Printf("dispose %s\n", old)

        dispose := exec.Command(doctl, "compute", "droplet", "delete", old,
                "--force",
        )

```

```
67          if err := dispose.Run(); err != nil {
68                  log.Fatal(err)
69          }
70  }
```

A.2.2 bluegreen.go

```
1   // Copyright (C) 2017 Kasper Kronborg Isager
2   //
3   // Permission is hereby granted, free of charge, to any person obtaining a copy
4   // of this software and associated documentation files (the "Software"), to deal
5   // in the Software without restriction, including without limitation the rights
6   // to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
7   // copies of the Software, and to permit persons to whom the Software is
8   // furnished to do so, subject to the following conditions:
9   //
10  // The above copyright notice and this permission notice shall be included in
11  // all copies or substantial portions of the Software.
12  //
13  // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
14  // IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
15  // FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
16  // AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
17  // LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
18  // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
19  // SOFTWARE.
20  package main
21
22  import (
23          . "github.com/kasperisager/partitionist"
24
25          example "../.."
26          provider "../../provider/digitalocean"
27  )
28
29  var cluster = example.Config1
30
31  func main() {
32          orchestrator := provider.Orchestrator{}
33          groups := make([]Group, len(cluster.Servers))
34
35          for i := range groups {
36                  groups[i].Lifecycle = Lifecycle{true}
37          }
38
39          for i, server := range cluster.Servers {
40                  groups[i].Servers = append(groups[i].Servers, server)
41          }
42
43          Orchestrate(orchestrator, Plan{groups})
```

```
}
```

A.2.3 rolling.go

```
// Copyright (C) 2017 Kasper Kronborg Isager
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
package main

import (
        . "github.com/kasperisager/partitionist"

        example "../.."
        provider "../../provider/digitalocean"
)

var cluster = example.Config4

func main() {
        orchestrator := provider.Orchestrator{}
        tolerance := MinTolerance(cluster.Servers)
        groups := make([]Group, tolerance)

        for i := range groups {
                groups[i].Lifecycle = Lifecycle{false}
        }

        for i, server := range cluster.Servers {
                group := i % tolerance
                groups[group].Servers = append(groups[group].Servers, server)
        }

        Orchestrate(orchestrator, Plan{groups})
}
```

A.2.4 partitioned.go

```
// Copyright (C) 2017 Kasper Kronborg Isager
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
package main

import (
        . "github.com/kasperisager/partitionist"

        example "../.."
        provider "../../provider/digitalocean"
)

var cluster = example.Config1

func main() {
        orchestrator := provider.Orchestrator{}
        plan, _ := cluster.Partition()
        Orchestrate(orchestrator, *plan)
}
```

A.2.5 cluster.go

```
// Copyright (C) 2017 Kasper Kronborg Isager
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
```

```
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
package example

import . "github.com/kasperisager/partitionist"

var Application = Service{
        ID: "application",
        Tolerance: 2,
}

var Database1 = Service{
        ID: "database",
        Tolerance: 1,
        State: &State{
                External: true,
        },
}

var Database2 = Service{
        ID: "database",
        Tolerance: 2,
        State: &State{
                External: true,
        },
}

var Metrics = Service{
        ID: "metrics",
        Tolerance: 1,
        State: &State{
                External: true,
        },
}

var Config1 = Cluster{
        Surge: 6,
        Servers: []Server{
                {
                        ID: "bsc-test-01",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
```

```
                        ID: "bsc-test-02",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-03",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-04",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-05",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-06",
                        Allocations: []Allocation{{Application, 1}},
                },
        },
}

var Config2 = Cluster{
        Surge: 6,
        Servers: []Server{
                {
                        ID: "bsc-test-01",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-02",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-03",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-04",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-05",
                        Allocations: []Allocation{{Database1, 1}},
                },
                {
                        ID: "bsc-test-06",
                        Allocations: []Allocation{{Database1, 1}},
                },
        },
```

```
}

var Config3 = Cluster{
        Surge: 6,
        Servers: []Server{
                {
                        ID: "bsc-test-01",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-02",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-03",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-04",
                        Allocations: []Allocation{{Application, 1}, {Database2, 1}},
                },
                {
                        ID: "bsc-test-05",
                        Allocations: []Allocation{{Application, 1}, {Database2, 1}},
                },
                {
                        ID: "bsc-test-06",
                        Allocations: []Allocation{{Application, 1}, {Database2, 1}},
                },
        },
}

var Config4 = Cluster{
        Surge: 6,
        Servers: []Server{
                {
                        ID: "bsc-test-01",
                        Allocations: []Allocation{{Application, 1}},
                },
                {
                        ID: "bsc-test-02",
                        Allocations: []Allocation{{Application, 1}, {Metrics, 1}},
                },
                {
                        ID: "bsc-test-03",
                        Allocations: []Allocation{{Application, 1}, {Metrics, 1}},
                },
                {
                        ID: "bsc-test-04",
                        Allocations: []Allocation{{Application, 1}, {Database2, 1}},
```

```
                },
                {
                        ID: "bsc-test-05",
                        Allocations: []Allocation{{Application, 1}, {Database2, 1}},
                },
                {
                        ID: "bsc-test-06",
                        Allocations: []Allocation{{Application, 1}, {Database2, 1}},
                },
        },
}
```