Patterns for Resilient Systems through Multi-Cloud Deployment

Cees de Groot, PagerDuty, Inc.

Cloud computing offers the unique possibility to increase a system's resilience by using data centers of multiple cloud providers, or multiple data centers of a single cloud provider. This paper contains a series or related patterns that support such a strategy: THREE DATA CENTERS, ONE COAST, OVERLAY NETWORK, TOPOLOGY AWARE, SERVICE REGISTRY, ASYNCHRONOUS REPLICATION, SYNCHRONOUS REPLICATION, GLOBAL LOCKS, AUTOMATE, and OUTSOURCE METRICS AND LOGGING.

Categories and Subject Descriptors: X.X [**Networks**] Cloud Computing

General Terms: Cloud Computing, Wide-Area Distribution

Additional Key Words and Phrases: Distributed Systems, DevOps

ACM Reference Format:

de Groot, C. 2016. Patterns for Resilient Systems through Multi-Cloud Deployment HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 23 (2016), 24 pages.

1. INTRODUCTION

Moving computing to cloud services can be very beneficial for a variety of reasons, including: money moves from Capex to Opex[1]; new services can be provisioned without a months-long hardware procurement cycle; there is no need to keep staff in a data center 24/7 to repair hardware failures.

[1] *Capex* is an acronym for capital expenditure; *Opex* for operational expenditure. The latter allows a business for more flexibility as it represents variable costs that can scale up and down with the needs of the enterprise

There is also a less-used advantage to this collection of patterns: deploying across multiple cloud instances can provide extra resilience. Data centers rarely go down, but if they do, it is usually a prolonged outage. Therefore, being able to keep a system running from a different data center is often a useful strategy.

For all but the largest companies, where operating a single data center (or co-located suite) is already a lot of work to take on, having multiple data centers is nigh impossible. The cloud provides a feasible alternative where a company of any size can spread infrastructure across multiple data centers without the drawbacks of a physical solution. Opening a new "data center" is as simple as firing up virtual machine instances in a different physical location of the current cloud provider, or swiping a credit card to be able to setup machine instances with an entirely different provider. This makes it cheap to add protection to even comparatively small systems against data center, or even provider-wide outages [DatacenterDynamics 2015].

The patterns in this collection are drawn from the author's experience with large distributed systems of this nature, and especially tie into current practices at PagerDuty, Inc.

1.1 Definitions

In the context of this paper, some common terms are used as follows:

Cloud. Here, one should specifically think of "the public cloud", meaning Infrastructure-as-a-Service offered by providers like Amazon, Microsoft, and Google. Even though nothing in this collection precludes the application of similar strategies in private or hybrid cloud setups, I'd like to stress the point here that "large company" resilience is attainable for small companies by leveraging the public cloud.

Data center. Cloud providers like the aforementioned refer to "availability zones" and similar terms which try to hide the fact that at the end of the day one is still using physical servers located in physical data centers. The term "data center" is used here to clarify that our focal point is on using multiple data centers regardless of how a given provider labels them.

Resilience. For the context of this pattern language, this term is used without any qualification to mean “resilience against data center outages”.

1.2 Target Audience

The intended audience of this paper is especially system designers working for companies that place great value on high availability even in the face of losing a data center; I hope that it will be interesting to anyone designing systems running on IaaS-type environments. The patterns assume a decent working knowledge of high availability designs and of cloud and DevOps patterns. [Sousa et al. 2015] is a good introduction to the latter.

1.3 Patterns in this collection

This collection starts by laying out how to choose the number and location of your data centers (THREE DATA CENTERS, ONE COAST) and how to tie them together in a single entity (OVERLAY NETWORK, SERVICE REGISTRY and TOPOLOGY AWARE). It continues with some patterns on how to move data between these and how to coordinate (ASYNCHRONOUS REPLICATION, SYNCHRONOUS REPLICATION and GLOBAL LOCKS). Finally, it stresses the vitality of some well known DevOps patterns that become extra important in the face of heterogeneous (cross-cloud provider) deployments: AUTOMATE and OUTSOURCE METRICS AND LOGGING.

2. THREE DATA CENTERS

The core premise of this paper is that systems can be made resilient even when a whole data center has an outage by applying principles that are well understood at the “cluster of machines“ level to “clusters of data centers“. The main question then becomes how to properly set the number of data centers needed to achieve the system’s non-functional requirements (which most likely revolve around resilience/availability and scalability).

One data center is putting your eggs in one basket; but how many is enough?

The more data centers you have to operate, the more complex operating them becomes; but fewer data centers increase the consequences of any single one going down. If you want to guard against losing a single data center, then the remaining data centers need to have plenty of reserve to absorb the load of the one that is now removed from the platform. It is easy to see that you need n data centers each having $n/(n-1)$ capacity, from which it follows that the total spare capacity carried $(1/(n-1))$ goes down with an increasing n. On the other hand, every data center requires a set of "overhead" machines–for monitoring, infrastructure, etcetera–which can be regarded as fixed relative to the number of systems running in that data center. With more data centers, the total of the fixed overhead per data center goes up, while the total of the spare capacity carried goes down. Often, the overhead increase wins it from the spare capacity decrease. Furthermore, you usually want to run consensus systems over your data centers requiring a quorum decision; these systems almost always work best if there is an uneven number of participants involved in order to avoid split-brain situations ([Wikipedia 2016b]).

Therefore,

Choose the minimum number of data centers that cannot cause a split brain situation but still will give redundancy: three. Have good reasons to go above that and consider always having an uneven number to make quorum consensus simpler.

2.1 Discussion

There are a number of conflicting forces at work here. Even in a cloud, data centers often have special machines that manage infrastructure and thus do not contribute compute and/or storage capacity to systems running the business logic; examples are login bastion hosts, machines collecting logs and metrics (however, see OUTSOURCE METRICS AND LOGGING for an argument in favor of not having such machines), machines responsible for running software deployment services, etcetera. This, in a sense, is fixed overhead per data center which needs to be amortized over the machines that do contribute capacity. Especially for smaller systems, this

overhead may make adding too many data centers infeasible. With more data centers, however, each data center contributes less capacity to the total capacity required and thus the spare capacity that needs to be carried around to protect against the outage of a data center becomes smaller. The detailed calculations are outside the scope of this paper, but it should be relatively simple to find, for a given system, a point where the spare capacity plus the total overhead is at a minimum. For example, if the total capacity required is 60 machines, and the fixed overhead per data center is 5 machines, then at $n = 4$ or $n = 5$ the total overhead ($n * 5$) plus the spare capacity ($total * (1/(n-1))$) are both 40 - with n above or below this point, this sum is higher[2].

The other consideration is more technical and less a matter of punching cost figures into spreadsheets. Various patterns in this paper argue to make the system work as a mostly seamless whole across data centers; what happens if one goes away? Distributed systems always have to contend with the potential of a "split-brain" situation, which is a situation where parts of the system are partitioned (often through a network interruption), continue to work independently, and end up with mutations on both sides that conflict when the parts are re-joined. A simple way to achieve resilience is by making sure that the members of a distributed system only work when there's a simple majority, or *quorum*, visible; there can be only one simple majority and thus only one part of a system that keeps working under a loss of network connectivity[3]. Simple majorities work best if there is an uneven number of participants; in the absence of this condition, the votes of some participants in the system need to be weighed as more important to make sure that a split-brain situation cannot occur. These weighing factors often make systems more brittle and harder to maintain, so if possible, an uneven number of data centers is the simplest solution.

[2]This example also shows that the cost of data center resilience can be high: for these example numbers, we go from 5 extra machines in a single data center to a minimum of 40 extra machines for a resilient setup.

[3]Note that this is more precisely "maximum one part" as a three-way split can easily result in none of the partitions having a majority and thus all of them refusing to work. This is an example of reducing availability to increase consistency. See also [Brewer 2012]

3. ONE COAST

You want multiple data centers and have settled on the number you need from THREE DATA CENTERS. Now you need to select which locations to use.

Where do we locate data centers?

There are two clear and opposite forces influencing your decision:

—Having your instances too close to each other does not protect against natural disasters;

—Having your instances too far apart from each other makes round trip times too long;

We[4] noticed that data centers of various vendors do tend to be clustered (local governments may give incentives, or local hydro-electric power generation may make it easy to obtain the coveted “green” label), but usually not so close that all but the worst natural disasters will have an impact.

Therefore,

Make round-trip time between data centers the primary selection tool. Stay on one coast (if in the USA), and when using data centers from multiple vendors, assume the worst about routing between the vendors–launch a virtual machine at each site and measure. Testing this is very cheap.

3.1 Discussion

Different cloud vendors tend to have very low latencies between their data centers. With fiber optic WAN connections, packets travel at a significant fraction of the speed of light [Miller 2012], which means that intermediate routers often introduce most of the latency. Private WAN interconnections minimize these “hops“, resulting in round-trip delays of modern WANs that come close to LAN delays ten or twenty years ago. Geographically sepa-

[4]More properly, my esteemed colleagues at PagerDuty responsible for site reliability engineering.

rate data centers of the same cloud vendor may therefore have, due to the use of private WANs, a lower latency than close data centers of different providers which usually get routed over the public Internet. For example, we observed lower and more consistent round-trip times between AWS regions in Oregon and Southern California than between Azure and AWS regions that are both located in Fresno, CA.

Again, there's a natural tension here: using all infrastructure from a single vendor will offer the best connectivity between data centers. However, there may be shared components in a vendor's infrastructure that can cause widespread outages, so adding more vendors may be an attractive option. Don't forget though that there's a clear financial drawback of using multiple vendors: one vendor will mean larger volume discounts, so there is an insurance premium to be paid for placing capacity among multiple providers.

4. OVERLAY NETWORK

Cloud providers usually work from the assumption that they are the only vendor; there is usually very good support for configuring and connecting systems inside one data center of their cloud, but less for interconnecting systems in separate data centers. With THREE DATA CENTERS on ONE COAST we need to find a means to connect them regardless of vendor functionality.

How do I connect systems in different data centers?

Multiple cloud vendors have their own flavors of what is called "virtual private clouds" (VPCs), which basically allow customers group clusters of virtual machines onto protected private networks. These tools are often very vendor-specific (so if you end up with multiple vendors, you need to learn multiple tools), and also require you to trust vendors to "do the right thing" on the network configuration. Furthermore, these tools only work within a single data center, which means that you still need firewalls and VPN connections between your VPCs. Systems running such "gateway to other data center" services will quickly become bottlenecks as more and more services need to go through them, loading the VPN nodes and making the central firewalls very complex.

Therefore,

Don't rely on VPCs, firewalls and VPNs to connect systems between multiple data centers. IPsec can make a mesh encrypted network that connect each node to each other node it needs to talk to, securing communication between each and every node whether local or remote; OS-level packet filters can then allow only inbound traffic from other nodes known to require communication, essentially acting as one firewall per node.

4.1 Discussion

There is a reason that firewalls are often implemented in hardware, or at least as optimized appliances - all of the traffic between "inside" and "outside" flows through them, so they need to be very fast. As a system grows larger, both the traffic through the firewall increases and the complexity of the firewall's rule base grows, making it very hard to keep up. Traffic between data centers often is routed through VPN connections that add encryption, which is computationally expensive. Spreading work over multiple VPN endpoints is possible, but further complicates configuration.

With good DevOps practices, it becomes feasible to automate nodes to the extent that each and every node keeps an encrypted connection to every other node it needs to talk to. An IPsec Key Exchange daemon like Racoon[NetBSD 2009] can be instructed by DevOps automation tools to provide the IPsec kernel with keys for the right set of nodes. In this way, each system in the cluster participates in its own protection against the outside world (usually supported by kernel firewall rules on the node that prevent communication outside of the IPsec network), and each system contributes a bit of CPU power to do the necessary encryption and decryption of packets, eliminating central firewall/VPN machines as bottlenecks.

Furthermore, the resulting overlay network becomes transparent between data centers; there is no difference, in terms of addressing and routing, between accessing local nodes or remote nodes. This makes it very easy to bind systems in different data centers into WAN-spanning clusters.

The most important caveat of this solution is that it will only work when there is a tool to AUTOMATE infrastructure setup available. Without it, maintaining the complex configuration required for such a setup becomes infeasible.

5. TOPOLOGY AWARE

With an OVERLAY NETWORK, location becomes transparent in the sense that there is no difference between addressing a local or a remote system. However, it is not always a good idea to ignore these differences.

How do I know whether I'm talking to a local or remote system?

When everything is configured correctly, it becomes very simple to see the group of data centers and the machines inside them as one system. Very often, this will just work, until you run into the fact that Peter Deutsch' Fallacies of Distributed Computing [Deutsch and Gosling 1997] are more visible on WAN connections than inside local area networks. Ignoring the topology of the network simplifies systems but causes needless traffic over WAN connections, risking running into said fallacies sooner (on top of that, most cloud providers charge for "external" traffic).

Therefore,

Make your systems topology aware so that as much traffic as possible is kept within a data center. Having your SERVICE REGISTRY be topology aware so that local instances of services are returned first on a lookup is a good start. A topology-aware configuration will prefer local instances of services (whether they result from registry lookups or other means, like configuration files) where possible.

5.1 Discussion

One of Deutsch' Fallacies is "latency is zero". When these were written, the most prevalent network probably was Ethernet in either 10base2 or 10baseT physical configurations. Even though 10baseT looked like a star topology,

the actual repeaters in use were hubs, basically replicating the coaxial bus of 10base2 inside them; CSMA/CD [Wikipedia 2016a] was used in both, meaning that especially on loaded networks, latencies were widely variable. As latency drives a large number of performance metrics, from partition sensitivity to the number of requests per second that can be made over a logical connection, reducing latency as much as possible is of paramount importance.

"Latency is zero" has become less and less important over time, as traffic inside a data center is now switched, fiber connections between racks of machines essentially operate at light speed, and intermediate routers only add minuscule delays. Round-trip times are measured in tens to hundreds of micro-seconds and one can get very far by essentially ignoring the network.

However, when adding multiple data centers, one is back in the '90s again. Latencies are widely variable, measure in the tens to hundreds of milliseconds, and ignoring the network when crossing the WAN will quickly lead to problems. Often, testing is done during "nice Internet weather", where everything will look nice and latencies are basically 50% of light speed, but there is pretty much a guarantee that no matter how traffic is routed between data centers, there will be bad days. Or weeks. Cable repairs in remote areas are hard, damaging them is not.

In order to be successful in a multi-data center setup, it's therefore very important to keep traffic flows inside the data center as much as possible; in fact, WAN traffic should be the exception and every time this happens, there should be a clear and good reason for it. Some examples of "good" WAN traffic are cluster state information and data replication; some examples of "bad" WAN traffic are web servers talking to remote databases or microservices talking to other microservice instances in remote data centers.

Systems need to be made aware of where services reside, and need to have a strong preference for local instances; in fact, when critical systems (like databases) are not available in a data center, it is probably preferable

to have client systems fail in that data center as well rather than going over the WAN, stressing remote systems needlessly[5].

6. SERVICE REGISTRY

When clusters and subsystems live in various data centers, it becomes hard to keep track of what lives where. The old-fashioned method of keeping hostnames in configuration files becomes problematic.

How can you find out what lives where?

Even when you AUTOMATE, it is still hard to have configuration files reflect the state of a whole network, especially with machines going away in far-away data centers. There are ways to circumvent this, for example by re-generating configuration files from dynamic state whenever the state changes, but this still brings delays in spreading configuration knowledge that may impact system performance. Furthermore, while such a method requires minimal changes to existing applications, usually a fresh configuration file can only be picked up through an application restart, reducing the availability of the system.

Therefore,

Use a Service Registry to keep an up-to-date view of which (instances of) services are available and where they live. It is helpful if the registry is TOPOLOGY AWARE. Some candidates are Consul [Hashicorp 2016a], Zookeeper [Apache 2016d] and etcd [CoreOS 2016].

[5]For example, a database will keep the connection open until the TCP shutdown has completed. When latencies go up, this takes longer meaning that more connections will stay in memory. This can potentially stress servers that are already under duress because of a partial outage to the point where they fail as well.

6.1 Discussion

When a system is stable, knowledge about the system is identical no matter where you look. The system can be said to be in a “converged“ state and is working at optimal efficiency. However, the more complex a system grows (in terms of the number of machines, the amount of services endpoints, etcetera), the more often changes will be made to it. With configuration systems that rely on (generated) files and application restarts, it will take a while for every change to propagate, and if this time is long enough and system changes are often enough, it is easy to see that the system potentially never converges. Such systems need extra spare capacity because they essentially always have machines that are not available to do work; this is both technically and economically unsound.

A service registry provides a central point where applications can ask for information about the network; instead of consulting local configuration files, the registry is contacted. Because data is now kept in a central location, having a service registry enables the quick dissemination of new knowledge around the cluster and often only needs a handful of nodes in each data center to be resilient. Application APIs are available to actively inform application instances of configuration changes they would be interested in, so that applications can quickly respond; convergence therefore can take place in seconds instead of hours, greatly reducing the time that the system is in a transitional state and thus minimizing the amount of time that the system has less than optimal resilience.

As the service registry should be TOPOLOGY AWARE, it needs to be setup in a topology aware fashion itself; either only returning local references (this can easily be done with any system by running it separately per data center) or by returning local references by default and remote references when requested (this needs support in the service registry).

6.2 Example

Consul is a service registry system that contains a large number of very desirable features. For example, it has a DNS interface which makes integrating applications a breeze, and it is based on the Raft consensus protocol

which makes the code simple to verify and reason about [Ongaro and Ousterhout 2014]. Most important, though, is the way how Consul handles multiple data centers.

Consul typically runs as a small daemon on every system. This makes a bootstrap process of “how do I find my service registry?” go away: it sits on `localhost` at well-known ports. Only a handful of nodes participate in the consensus algorithms; they are called “servers” and usually they are run on dedicated machines. Within a data center, the servers use Raft to elect a leader among them, and this leader receives all queries; node-local Consul agents are configured with one or more bootstrap nodes and then join a gossip network to replicate the state of the network (who are the members, where are the members, and who is the leader).

Optionally, Consul can join multiple clusters by so-called WAN bridges. This, in effect, creates a new gossip network between the servers over the WAN, enabling services in one data center to forward queries to another data center when the queries cannot be locally resolved; for example, this is the case when a client explicitly requests a service lookup in a remote data center. There is also support for enumerating known data centers and nodes in each data center, so that it is possible for subsystems that need it to gain complete knowledge of the configuration at a given point in time; this can be used for example to configure high availability clusters automatically, based on the location of available participants in the cluster.

In essence, Consul provides all the functionality that is needed to be TOPOLOGY AWARE, which makes it a very good fit for these kind of systems.

7. ASYNCHRONOUS REPLICATION

In a TOPOLOGY AWARE system, traffic and data are kept as much as possible in a single data center. Keeping data sets confined to a single data center makes the system vulnerable against outages, which means we need to find a way to spread it around.

How do I move data between data centers so that it does not have a large impact on performance?

Data can be replicated by transmitting changes that occur between transactional checkpoints. Often, data can have gaps and still be valid; data for analytical purposes (for example, user behavior metrics) can potentially miss minutes but still be valuable enough to serve as a source for information. Especially for this kind of data it is often not acceptable to wait for it to be safely replicated as this may put unacceptable delays on interactive user flows.

Therefore,

Send data that can be lost or re-created in asynchronous batches to other data centers, outside the scope of any transactions (either by using systems that forego transactions, or by sending the data after the transaction has been completed at the source). Queueing systems like Kafka [Apache 2016c] can be helpful here to transport data between data centers (“fire and forget”).

7.1 Discussion

Asynchronous replication will come at a cost: data will be lost if the source data center fails. There is no way around this: some transactions that have just been committed in the originating data center before it went down will not have propagated out of it. These transactions will be lost forever if we assume that the original data center is lost forever. This is the reason that this pattern stresses its usefulness for data that can be lost. Which data can be lost in the case of a data center outage (a very irregular event) is a business decision and therefore not in scope of this paper. An example may be a company that hosts weblogs: it may decide that, in case of a data center failover, losing some comments and posts is annoying to the user, but from a business perspective entirely acceptable.

If the business critically depends on certain data, then this data should not be replicated this way; SYNCHRONOUS REPLICATION should be used instead.

7.2 Examples

MySQL [Oracle 2016a] has been wildly popular with large scale websites because of its built-in asynchronous replication [Oracle 2016b]. Executed statements and/or row changes are written to a binary log on the *master* and can be fetched by *slaves* that replay the changes at their own pace. The log is written on transaction commit and thus only available to slaves outside the transaction scope; this means that replication can cross unreliable network links without impacting the observed speed of online transactions against the master.

A lot of "eventually consistent" storage systems work by accepting writes at one node in the cluster, and then replicating it across multiple nodes so that eventually, a quorum of nodes agrees on the data. Cassandra [Apache 2016a] is a good example, because it can be operated to be aware of multiple data center topologies. Data can be written to a local node of a data center-spanning cluster, and it will eventually arrive at nodes in other data centers, asynchronously.

8. SYNCHRONOUS REPLICATION

Some data is important enough not to lose. It can represent a transaction, or data that the system cannot recreate (because it originates externally, for example it came in through an API). Often, the data carries a monetary value or an external obligation.

How do I copy data between data centers so that it never gets lost?

Outages of any kind come with losses. Preventing or reducing the cost of these losses costs money, so these measures can be seen as an insurance premium. It is important to make a quick calculation of costs of loss and benefits of prevention of loss, but sometimes the balance will be that data cannot be lost even under very rare circumstances. This means that transactions need to span multiple data centers to insure against losing data that is in-flight in a data center when it goes down.

Therefore,

Replicate data between data centers synchronously when loss of data is deemed costly. Synchronous replication can severely impact the responsiveness of systems, but is an essential tool. In effect, the transaction should not complete until the data has landed in a quorum subset of the systems involved. Doing quorum writes obviously reduces availability, so whenever synchronous replication is involved there is a need to make sure that transactions are retried.

8.1 Discussion

Synchronous replication is a measure of last resort. It is expensive, slows systems down, has the risk of reduced availability, and forces applications to be aware of all that. An application that uses a local MySQL master cluster that asynchronously replicates to another data center can be blissfully unaware of that happening–for all practical purposes, the database behaves like this replication isn't happening and therefore behaves like a single local instance. When synchronous replication becomes part of the picture, however, a single transaction will become much slower (because of the round trip times involved) and has a higher risk of failing (because of network connectivity issues), thus the application should be written to take this into account. Often, attempts to hide this complexity in a library fail as business logic needs to be adapted to handle new kinds of failures; this increase in business logic complexity puts a premium on synchronous replication.

If there is any way that the system can get away with asynchronous replication, that should be the preference. The cost of losing a couple of in-flight transactions in a once-a-year event should be clearly weighed against the overhead of adding synchronous replication.

8.2 Example

Cassandra [Apache 2016a] is an elastic key/value store that has the interesting property of being data center aware. Cassandra nodes can be tagged with a data center (or even a rack) name, and the system can be

instructed to replicate data across data centers. With strong write consistency, Cassandra won't complete a write operation until at least a quorum of all data centers have reported back that the write has been written to disk. In this sense, one can be safe that when Cassandra reports back that a write was successful, the data has been durably stored in multiple data centers.

One drawback of Cassandra is that writes are atomic operations, but there is no locking. This means that conflicting writes are possible. There are ways to lessen the impact of conflicting writes which are out of scope of this paper as they are very Cassandra-specific, but generally speaking conflicting writes are bad and should be avoided.[6]

The way to avoid conflicting writes is to serialize them. A good way is to use GLOBAL LOCKS around the operation. The lock assures serialized writes; a read-before-write in the lock will detect write conflicts, and Cassandra will assure replication across data centers.

9. GLOBAL LOCKS

There are various circumstances where work needs to be coordinated between data centers. An example could be seen in the previous section, where SYNCHRONOUS REPLICATION in an otherwise transaction-less key/value store could result in write conflicts; another example is where a single process should work on data and have it only be restarted in (potentially) a different data center when the original one goes down. Coordination can also mean turning an asynchronous system into a synchronous one by waiting until the replication has happened while excluding other systems from writing by obtaining a lock during that wait period.

How do I coordinate work across data centers?

[6] Cassandra has only one resolution method, "last writer wins", which is usually not the resolution you want for business data. In contrast, systems like Riak [Basho 2016] have configurable policies which makes it possible to use, for example, CRDTs [Shapiro et al. 2011] to resolve conflicts or otherwise use knowledge of the business logic to do so.

Like synchronous replication, locking across data centers is expensive and not transparent to the locking process. Locks can fail, they take some time to obtain as they must be coordinated with instances of the locking system running in multiple data centers, and the name or key to lock on must be chosen carefully to balance between locking contention (lock names too coarse grained) and risk of conflicts (lock names too fine grained). It's a tool in the distributed system designer's toolbox, but not one that should be retrieved at the first hint that it might solve a problem. But there are points where having the ability to take locks across data centers is a requirement.

Therefore,

Have a system to implement locking across data centers. With these global locks, it becomes possible to coordinate systems and actions between data centers. A transaction can be scoped by taking a lock; writes to otherwise transaction-less systems can then be serialized on the lock. A single process to do work can be elected similarly: one process holds a long-running lock, and when it fails, the lock is released and other processes then have a chance to obtain it and start performing work.

9.1 Example

Zookeeper [Apache 2016d] is a distributed consensus system originally implemented by Yahoo to coordinate their massive (for that time) networks. Yahoo wrote the system to perform leader election, locking, and to act as a service registry. It has been around for a long time and can be considered battle-hardened, and it performs very well as a global locking service.

One way to run Zookeeper is to have 5 nodes over THREE DATA CENTERS. Zookeeper does not benefit from more machines (as the leader of the cluster is the bottleneck), and in this way the outage of any data center will leave a quorum intact; furthermore, two machines in different data centers can go down with no impact. This is a general pattern for data replication that is applicable to replicated systems like Cassandra and Kafka as well;

contrary to data center outages, machine outages are common, and with a 3 node setup a single node failure would immediately put the system at risk of not being available.

Interacting with Zookeeper is quite low level and requires a lot of knowledge on how the various primitives are implemented; in essence, it is more a toolbox than an end-user application. Various users have published “recipes” for interacting with Zookeeper in library form, and their use is recommended–Apache Curator [Apache 2016b] is a leading software component in this field.

10. AUTOMATE

With multiple data centers sourced from potentially different vendors, keeping everything in a sane state becomes quite hard.

How do you manage systems and networks across multiple providers?

A lot of small companies start deploying their systems by hand, using appropriately configured base images delivered by the cloud vendor and then installing any necessary software on them using the operating system’s packaging tools. This gets a system up and running very fast, but as the system scales, such a method becomes progressively more error prone; choosing when to cut over to an automated process is quite hard and often is only done after ample evidence in the form of a prolonged outage, security breach, or loss of data arrives. Especially when connecting systems in multiple data centers, this evidence will arrive rather quickly as such systems are inherently more complex than their single-data center counterparts.

Therefore,

Use an infrastructure automation tool to configure machines from executable descriptions; consider using a cloud vendor independent orchestration tool to bringing them online.

10.1 Discussion

Automation is not unique to this set of patterns, but it becomes extra important when using multiple vendors. Instead of relying on vendor-dependent tooling to manage and protect networks of machines, machines need to be interconnected with an OVERLAY NETWORK, machines in different vendors' data centers get slightly different settings because networks and available memory/cpu sizes differ, and so on.

Even with a handful of machines, this quickly becomes very error prone when doing manually. Whereas good DevOps practices can be postponed by a starting company for a while, when applying the patterns in this paper it should be considered a requirement from day one.

10.2 Examples

With the rise of the "DevOps" movement, the idea of "infrastructure as code" is rapidly gaining a strong foothold in the industry. Consequently, there is a wealth of often freely available tooling in this space. Chef [Chef 2016] and Puppet [Puppet 2016] are both strong contenders for the role of infrastructure automation tools, although other viable alternatives are available. An example of a third-party orchestration tool is TerraForm [Hashicorp 2016b], a tool that can bootstrap machines across a range of hosting providers and cloud services and then start the infrastructure automation tool on the first run to complete configuration.

11. OUTSOURCE METRICS AND LOGGING

Systems divided over multiple data centers are a complex beast. Not only do they exhibit their usual complex behavior (subsystem *A* deteriorating because subsystem *G*, which everyone thought was independent, had a hiccup), we are making it worse by mixing in round trips over WANs all the time.

How do I get a global view of system health?

Centralizing logging in a single data center is relatively simple (using built-in tools like rsyslog and software packages like Graphite [Graphite 2016]), but insights should be independent from any single data center being available. Building a logging or telemetry system that spans multiple data centers is not something that most companies should be prepared to tackle; even though it is often seen as a quick and cheap solution on the short term, it will usually take a lot of work to scale systems in the longer term.

Therefore,

Outsource metrics collection and logging to an external SaaS provider. These companies are dedicated to building scalable and available systems and let their customers' engineers focus on company-specific problems. Where for a single data center, there is a cost trade-off that often falls on the side of a DIY solution, spanning highly available metrics and log collection systems over multiple data centers should be out of scope for all but the largest companies.

11.1 Discussion

Moving to the cloud is a smart decision for many systems, as the cost of building systems becomes variable with the size of them and initial fixed investments all but disappear. Similarly, even a small HA logging system is a costly affair, requiring quite a number of machines and probably all sorts of homebrew work around them to make them work over multiple sites. Scaling these systems, if anything, is even harder. Graphite, for example, will happily run on a tiny machine and be quite usable on it. Synchronizing data from one Graphite server to another is feasible if one accepts the risk of losing data around outages. Scaling Graphite, though, can take man-months to get correct and then the work needed to keep multiple of these clusters in sync becomes even harder. This takes away valuable engineering time that should be put to better use: solving the problems that are unique to the organization.

11.2 Examples

Two SaaS providers in this field the author is familiar with are Datadog [Datadog 2016] and Splunk Cloud [Splunk 2016]. Both operate in a similar fashion: on every node, an agent is installed that performs local data collection and an initial round of massaging. Data is then forwarded to the SaaS provider, where it is stored and indexed and made available for subsequent retrieval. For both problem domains there is a growing list of companies that provide the service at various quality and pricing levels, so these two are by far not the only options.

12. ACKNOWLEDGEMENTS

This paper has been turned into a much better paper by Filipe Correia's shepherding to prepare it for PLoP 2016. I also would like to thank my colleagues Arup Chakrabarti and Joseph Pierri and my daughter Anne de Groot for taking the time to proofread and providing me with feedback and corrections. Thanks to Phyllis Alberts-Meijers for picking up the proverbial red pen outside the classroom to guide me on a path to better English. At the time of writing this, I don't know who will join in the workshop, but I do already know from past PLoP experience that your input will be very valuable. I hope that this collection of patterns will prove useful to the reader - thanks for taking your time to work through it.

REFERENCES

Apache. 2016a. The Apache Cassandra Project. https://cassandra.apache.org/. (June 2016). (Accessed on 06/20/2016).

Apache. 2016b. Apache Curator. https://curator.apache.org/. (June 2016). (Accessed on 07/15/2016).

Apache. 2016c. Apache Kafka. https://kafka.apache.org/. (June 2016). (Accessed on 06/20/2016).

Apache. 2016d. Apache ZooKeeper - Home. https://zookeeper.apache.org/. (February 2016). (Accessed on 07/09/2016).

Basho. 2016. Riak KV. http://docs.basho.com/riak/kv/2.1.4/. (April 2016). (Accessed on 07/15/2016).

Eric Brewer. 2012. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* 45, 2 (February 2012), 23–29. DOI:http://dx.doi.org/10.1109/MC.2012.37 Published on-line by InfoQ at https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed.

Chef. 2016. Chef–Automate Your Infrastructure. https://www.chef.io/chef/. (August 2016). (Accessed on 08/05/2016).

CoreOS. 2016. etcd Documentation. https://coreos.com/etcd/docs/latest/. (June 2016). (Accessed on 07/09/2016).

DatacenterDynamics. 2015. AWS suffers a five-hour outage in the US | News | DatacenterDynamics. http://www.datacenterdynamics.com/content-tracks/colo-cloud/aws-suffers-a-five-hour-outage-in-the-us/94841.fullarticle. (September 2015). (Accessed on 06/20/2016).

Datadog. 2016. Cloud Monitoring as a Service. https://www.datadoghq.com/. (July 2016). (Accessed on 07/15/2016).

Peter Deutsch and James Gosling. 1997. The Eight Fallacies of Distributed Computing. http://www.ibiblio.org/xml/slides/acgnj/syndication/cache/Fallacies.html. (? 1997). (Accessed on 07/02/2016).

Graphite. 2016. Graphite. https://graphiteapp.org/. (July 2016). (Accessed on 07/15/2016).

Hashicorp. 2016a. Consul by HashiCorp. https://www.consul.io/. (March 2016). (Accessed on 07/09/2016).

Hashicorp. 2016b. Terraform by HashiCorp. https://www.terraform.io/. (August 2016). (Accessed on 08/05/2016).

Kevin Miller. 2012. Calculating Optical Fiber Latency. http://www.m2optics.com/blog/bid/70587/Calculating-Optical-Fiber-Latency. (January 2012). (Accessed on 07/02/2016).

NetBSD. 2009. racoon–NetBSD Manual Pages. http://netbsd.gw.com/cgi-bin/man-cgi?racoon++NetBSD-current. (January 2009). (Accessed on 07/02/2016).

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

Oracle. 2016a. MySQL. https://www.mysql.com/. (June 2016). (Accessed on 06/20/2016).

Oracle. 2016b. MySQL 5.7 Reference Manual – Replication. https://dev.mysql.com/doc/refman/5.7/en/replication.html. (June 2016). (Accessed on 06/20/2016).

Puppet. 2016. Puppet - The shortest path to better software. https://puppet.com/. (August 2016). (Accessed on 08/05/2016).

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642

Tiango Boldt Sousa, Filipe Correia, and Hugo Sereno Ferreira. 2015. Patterns for Software Orchestration on the Cloud. *HILLSIDE Proc. of Conf. on Pattern Lang. of Prog.* 22 (October 2015), 12.

Splunk. 2016. Cloud Solutions. http://www.splunk.com/en_us/cloud.html. (July 2016). (Accessed on 07/15/2016).

Wikipedia. 2016a. Carrier sense multiple access with collision detection. https://en.wikipedia.org/wiki/Carrier_sense_multiple_access_with_collision_detection. (April 2016). (Accessed on 07/09/2016).

Wikipedia. 2016b. Split-brain (computing). https://en.wikipedia.org/wiki/Split-brain_(computing). (August 2016). (Accessed on 08/05/2016).