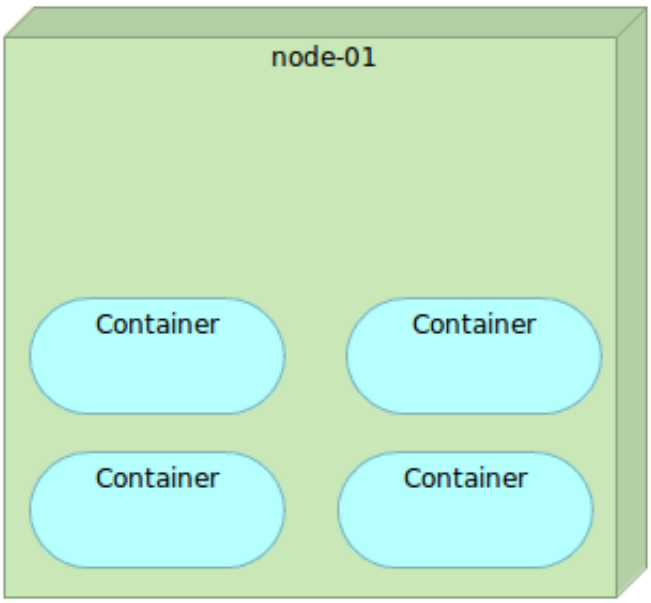
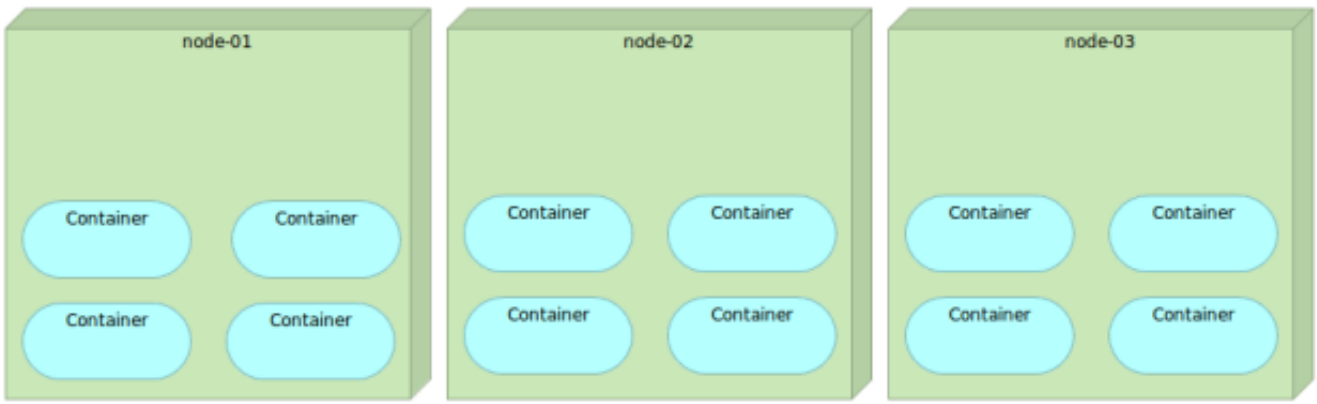
Technology Conversations

Service Discovery: Zookeeper vs etcd vs Consul

The more services we have, the bigger the chance for a conflict to occur if we are using predefined ports. After all, there can be no two services listening on the same port. Managing a tight list of all the ports used by, lets say, hundred services is a challenge in itself. Add to that list the databases those services need and the number grows even more. For that reason we should deploy services without specifying the port and letting Docker assign a random one for us. The only problem is that we need to discover the port number and let others know about it.



Things are getting even more complicated when we start working on a distributed system with services deployed into one of the multiple servers. We can choose to define in advance which service goes to which server but that would cause a lot of problems. We should try to utilize server resources as best we can and that is hardly possible if we define in advance where to deploy each service. Another problem is that automatic scaling of services would be difficult at best and not to mention automatic recuperation from, let's say, server failure. On the other hand, if we deploy services to the server that has, for example, least number of containers running, we need to add the IP to the list of data needed to be discovered and stored somewhere.



There are many other examples of cases when we need to store and retrieve (discover) some information related to the services we are working with.

In order to be able to locate our services we need at least the following two processes to be available for us.

- **Service registration** process that will store, as a minimum, the host and the port service is running on.
- **Service discovery** process that will allow others to be able to discover the information we stored during the registration process.

Besides those processes, we need to consider several other aspects. Should we unregister the service if it stops working and deploy/register a new instance? What happens when there are multiple copies of the same service? How do we balance the load among them? What happens if a server goes down? Those and many other questions are tightly related to the registration and discovery processes. For now, we'll limit the scope only to the **service discovery** (common name that envelops both aforementioned processes) and the tools we might use for such a task. Most of them feature some kind of highly available distributed key/value storage.

Service discovery tools

The main objective of **service discovery tools** is to help services find and talk to one another. In order to perform their duty they need to know where each service is. The concept is not new and many tools existed long before Docker was born. However, containers brought the need for such tools to a completely new level.

The basic idea behind **service discovery** is for each new instance of a service (or an application) to be able to identify its current environment and store that information. Storage itself is performed in a registry usually in key/value format. Since the discovery is often used in distributed system, registry needs to be scalable, fault tolerant and distributed among all nodes in the cluster. Primary usage of such a storage is to provide, as a minimum, IP and port of the service to all interested parties that might need to communicate with it. This data is often extended with other types of information.

Discovery tools tend to provide some kind of API that can be used by a service to register itself as well as by others to find the information about that service.

Let's say that we have two services. One is a provider and the other one is its consumer. Once we deploy the provider we need to store its information to the **service discovery registry** of choice. Later on, when the consumer tries to access the provider, it would first query the registry and call the provider using the IP and port obtained from the registry. In order to decouple the provider from the specific implementation of the registry, we often employ some kind of **proxy service**. That way the consumer would always request information from the fixed address that would reside inside the proxy that, in turn, would use the discovery service to find out the provider information and redirect

the request. We'll go through **reverse proxy** later on in the book. For now it is important to understand the flow that is based on three actors; consumer, proxy and provider.

What we are looking for in the service discovery tools is data. As a minimum we should be able to find out where the service is, whether it is healthy and available and what is its configuration. Since we are building a distributed system with multiple servers, the tool needs to be robust and failure of one node should not jeopardize data. Also, each of the nodes should have exactly the same data replica. Further on, we want to be able to start services in any order, be able to destroy them or replace them with newer versions. We should also be able to reconfigure our services and see the data change accordingly.

Let's take a look at few of the commonly used options to accomplish the goals we set.

Manual configuration

Most of the services are still managed manually. We decide in advance where to deploy the service, what is its configuration and hope beyond reason that it will continue working properly until the end of days. Such approach is not easily scalable. Deploying a second instance of the service means that we need to start the manual process all over. We need to bring up a new server or find out which one has low utilization of resources, create a new set of configurations and deploy it. The situation is even more complicated in case of, let's say, a hardware failure since the reaction time is usually slow when things are managed manually. Visibility is another painful point. We know what the static configuration is. After all, we prepared it in advance. However, most of the services have a lot of information generated dynamically. That information is not easily visible. There is no single location we can consult when we are in need of that data.

Reaction time is inevitably slow, failure resilience questionable at best and monitoring difficult to manage due to a lot of manually handled moving parts.

While there was excuse to do this job manually in the past or when the number of services and/or servers is low, with emergence of service discovery tools, this excuse quickly evaporates.

Zookeeper

ZooKeeper is one of the oldest projects of this type. It originated out of the Hadoop world, where it was built to help in the maintenance of the various components in a Hadoop cluster. It is mature, reliable and used by many big companies (YouTube, eBay, Yahoo, and so on). The format of the data it stores is similar to the organization of the file system. If run on a server cluster, Zookeper will share the state of the configuration across all of nodes. Each cluster elects a leader and clients can connect to any of the servers to retrieve data.

The main advantages Zookeeper brings to the table is its maturity, robustness and feature richness.

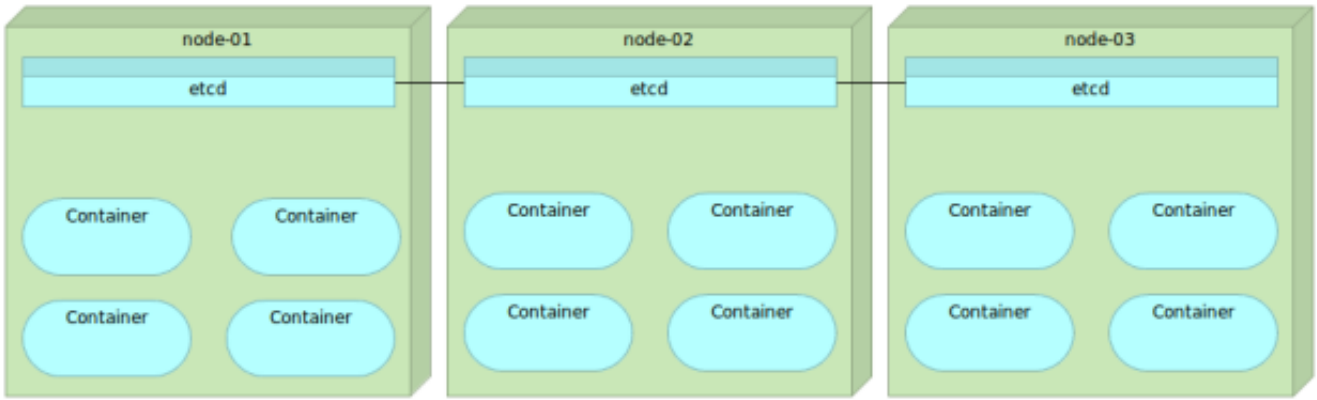
However, it comes with its own set of disadvantages as well, with Java and complexity being main culprits. While Java is great for many use cases, it is very heavy for this type of work. Zookeeper's usage of Java together with a considerable number of dependencies makes it much more resource hungry that its competition. On top of those problems, Zookeeper is complex. Maintaining it requires considerably more knowledge than we should expect from an application of this type. This is the part where feature richness converts itself from an advantage to a liability. The more features an application has, the bigger the chances that we won't need all of them. Thus, we end up paying the price in form of complexity for something we do not fully need.

Zookeeper paved the way that others followed with considerable improvements. "Big players" are using it because there were no better alternatives at the time. Today, Zookeeper it shows its age and we are better off with alternatives.

etcd

etcd is a key/value store accessible through HTTP. It is distributed and features hierarchical configuration system that can be used to build service discovery. It is very easy to deploy, setup and use, provides reliable data persistence, it's secure and with a very good documentation.

etcd is a better option than Zookeeper due to its simplicity. However, it needs to be combined with few third-party tools before it can serve service discovery objectives.



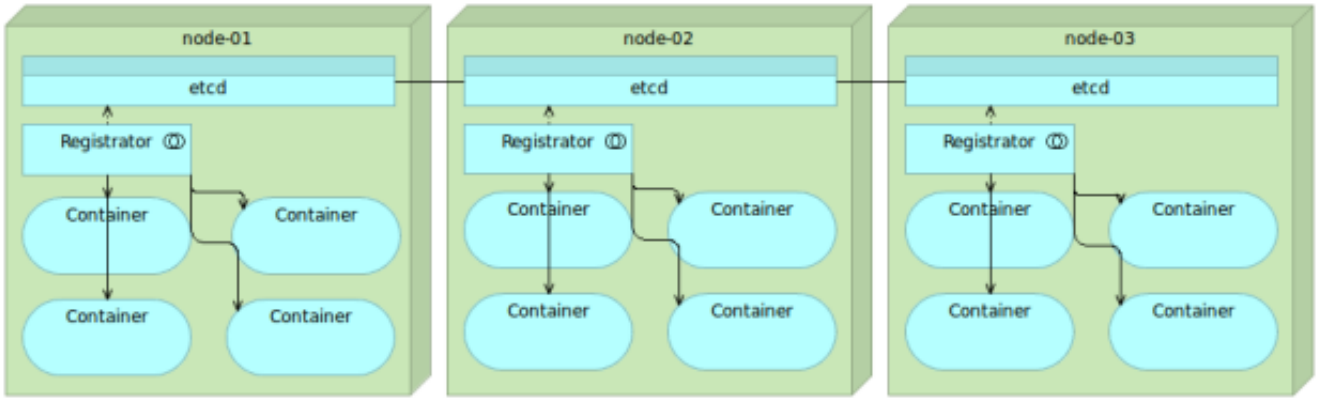
Now that we have a place to store the information related to our services, we need a tool that will send that information to etcd automatically. After all, why would we put data to etcd manually if that can be done automatically. Even if we would want to manually put the information to etcd, we often don't know what that information is. Remember, services might be deployed to a server with least containers running and it might have a random port assigned. Ideally, that tool should monitor Docker on all nodes and update etcd whenever a new container is run or an existing one is stopped. One of the tools that can help us with this goal is Registrator.

Registrator

Registrator automatically registers and deregisters services by inspecting containers as they are

brought online or stopped. It currently supports **etcd**, **Consul** and **SkyDNS 2**.

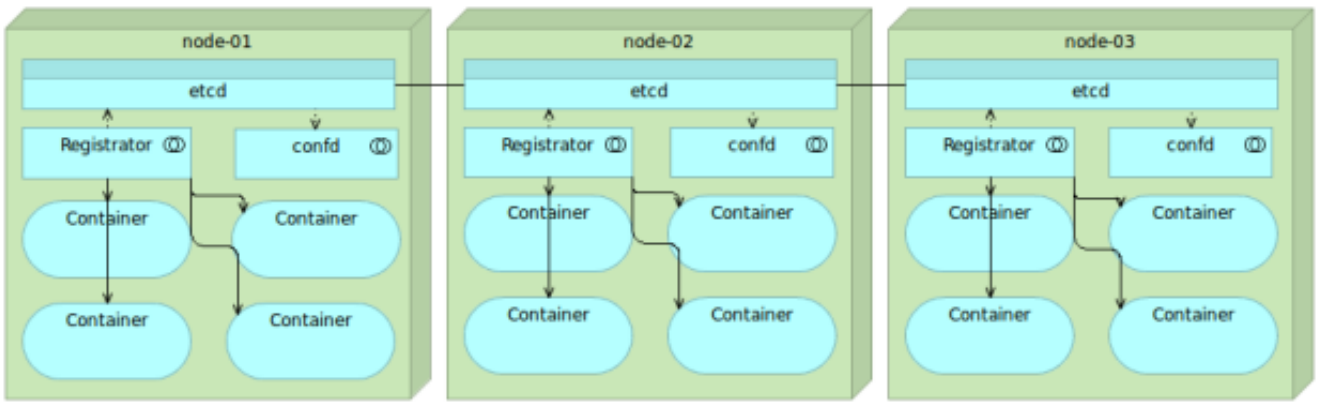
Registrator combined with **etcd** is a powerful, yet simple combination that allows us to practice many advanced techniques. Whenever we bring up a container, all the data will be stored in etcd and propagated to all nodes in the cluster. What we'll do with that information is up to us.



There is one more piece of the puzzle missing. We need a way to create configuration files with data stored in **etcd** as well as run some commands when those files are created.

confd

confd is a lightweight configuration management tool. Common usages are keeping configuration files up-to-date using data stored in **etcd**, **consul** and few other data registries. It can also be used to reload applications when configuration files change. In other words, we can use it as a way to reconfigure all the services with the information stored in etcd (or many other registries).



Final thoughts on etcd, Registrator and confd combination

When etcd, Registrator and confd are combined we get a simple yet powerful way to automate all our service discovery and configuration needs. This combination also demonstrates effectiveness of having the right combination of "small" tools. Those three do exactly what we need them to do. Less than this and we would not be able to accomplish the goals set in front of us. If, on the other hand, they were designed with bigger scope in mind, we would introduce unnecessary complexity and

overhead on server resources.

Before we make the final verdict, let's take a look at another combination of tools with similar goals. After all, we should never settle for some solution without investigating alternatives.

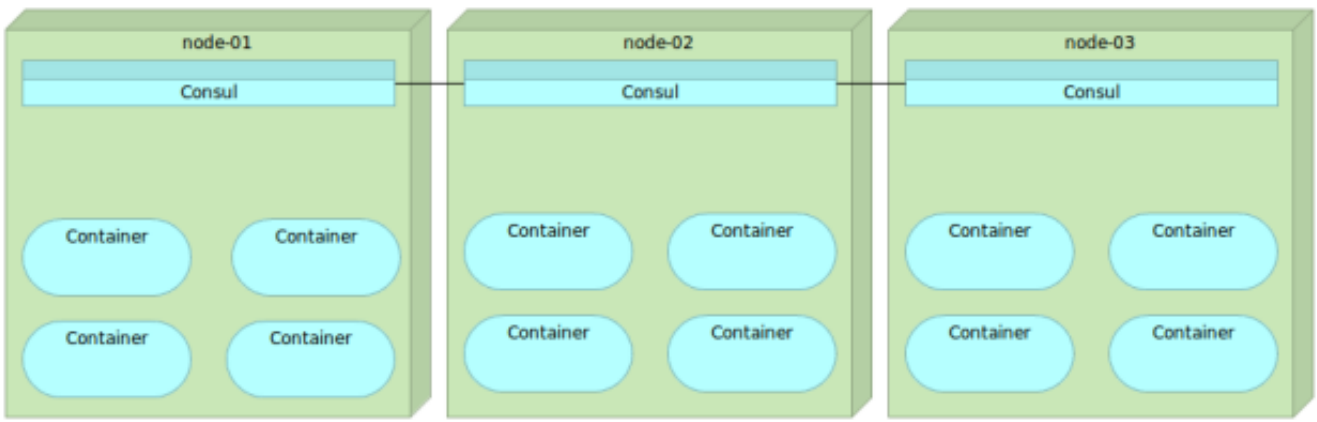
Consul

Consul is strongly consistent datastore that uses gossip to form dynamic clusters. It features hierarchical key/value store that can be used not only to store data but also to register watches that can be used for a variety of tasks from sending notifications about data changes to running health checks and custom commands depending on their output.

Unlike Zookeeper and etcd, Consul implements service discovery system embedded so there is no need to build your own or use a third-party one. This discovery includes, among other things, health checks of nodes and services running on top of them.

ZooKeeper and etcd provide only a primitive K/V store and require that application developers build their own system to provide service discovery. Consul, on the other hand, provides a built in framework for service discovery. Clients only need to register services and perform discovery using the DNS or HTTP interface. The other two tools require either a hand-made solution or the usage of third-party tools.

Consul offers out of the box native support for multiple datacenters and the gossip system that works not only among nodes in the same cluster but across datacenters as well.



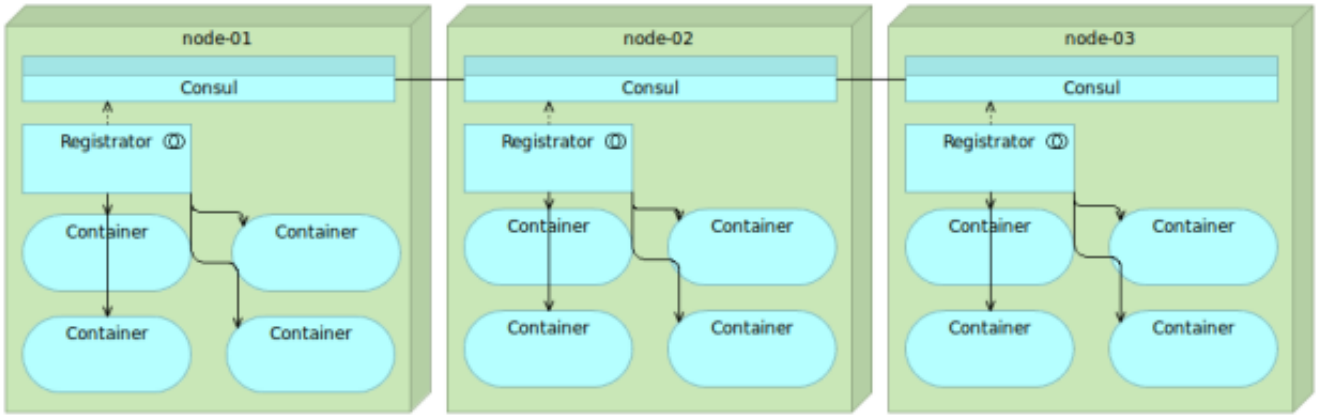
Consul has another nice feature that distinguishes it from the others. Not only that it can be used to discover information about deployed services and nodes they reside on, but it also provides easy to extend health checks through HTTP requests, TTLs (time-to-live) and custom commands.

Registrator

Registrator has two Consul protocols. The **consulkv** protocol produces similar results as those

obtained with the etcd protocol.

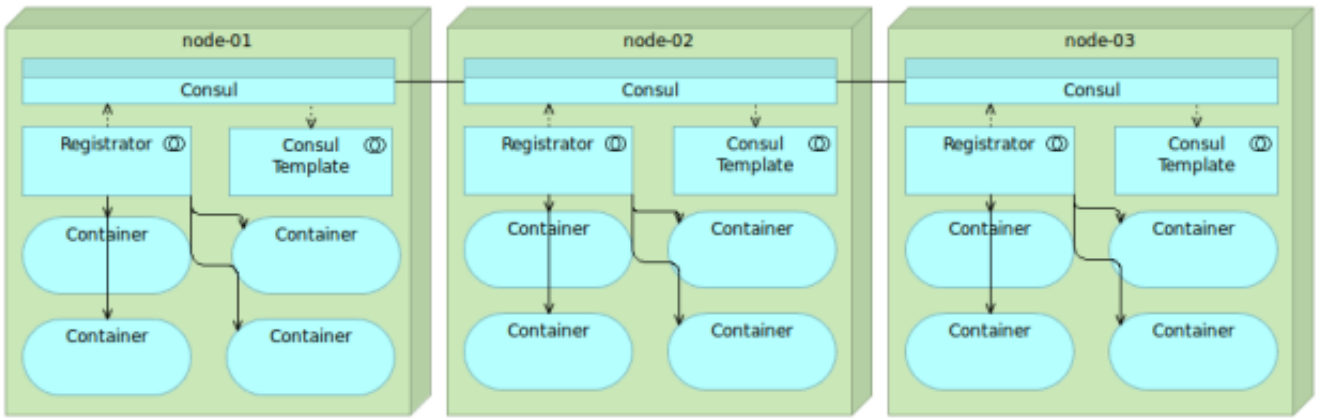
Besides the IP and the port that is normally stored with **etcd** or **consulkv** protocols, Registrator's **consul** protocol stored more information. We get the information about the node the service is running on as well as service ID and name. With few additional environment variables we can also store additional information in form of tags



consul-template

confd can be used with Consul in the same way as with etcd. However, Consul has its own templating service with features more in line with what Consul offers.

The consul-template is a very convenient way to create files with values obtained from Consul. As an added bonus it can also run arbitrary commands after the files have been updated. Just as confd, consul-template also uses Go Template format.



Consul health checks, Web UI and datacenters

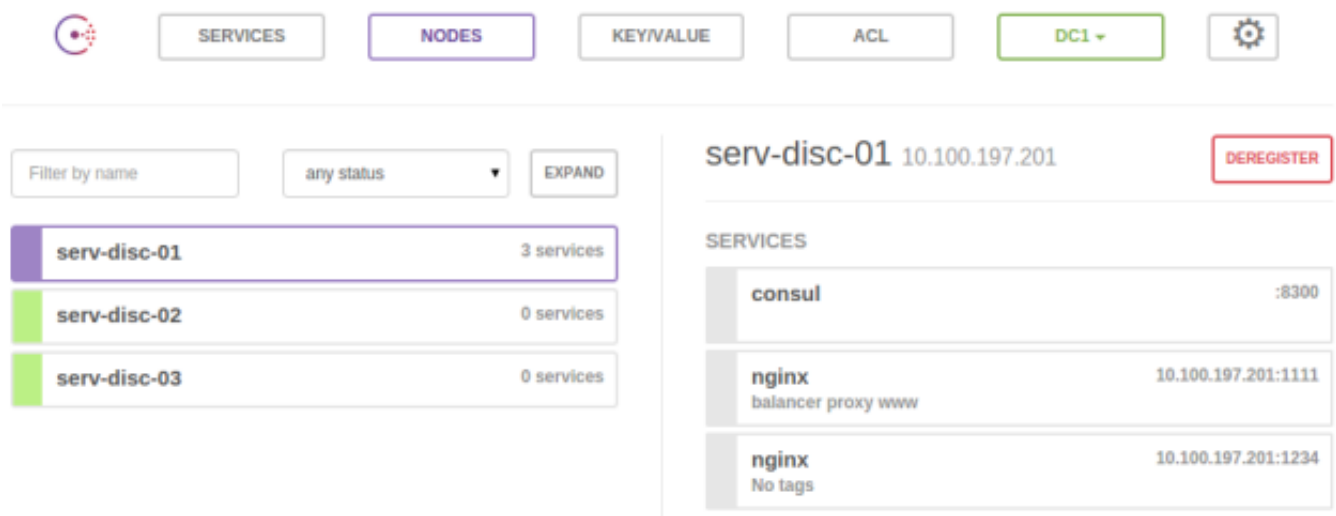
Monitoring health of cluster nodes and services is as important as testing and deployment itself. While we should aim towards having stable environments that never fail, we should also acknowledge that unexpected failures happen and be prepared to act accordingly. We can, for example, monitor memory usage and if it reaches certain threshold move some services to a

different node in the cluster. That would be an example of preventive actions performed before the "disaster" would happen. On the other hand, not all potential failures can be detected on time for us to act on time. A single service can fail. A whole node can stop working due to a hardware failure. In such cases we should be prepared to act as fast as possible by, for example, replacing a node with a new one and moving failed services. Consul has a simple, elegant and, yet powerful way to perform health checks and help us to define what actions should be performed when health thresholds are reached.

If you googled "etcd ui" or "etcd dashboard" you probably saw that there are a few solutions available and might be asking why we haven't presented them. The reason is simple; etcd is a key/value store and not much more. Having an UI to present data is not of much use since we can easily obtain it through the etcdctl. That does not mean that etcd UI is of no use but that it does not make much difference due to its limited scope.

Consul is much more than a simple key/value store. As we've already seen, besides storing simple key/value pairs, it has a notion of a service together with data that belongs to it. It can also perform health checks thus becoming a good candidate for a dashboard that can be used to see the status of our nodes and services running on top of them. Finally, it understands the concept of multiple datacenters. All those features combined let us see the need for a dashboard in a different light.

With the Consul Web UI we can view all services and nodes, monitor health checks and their statuses, read and set key/value data as well as switch from one datacenter to another.



Final thoughts on Consul, Registrator, Template, health checks and Web UI

Consul together with the tools we explored is in many cases a better solution than what etcd offers. It was designed with services architecture and discovery in mind. It is simple, yet powerful. It provides a complete solution without sacrificing simplicity and, in many cases, it is the best tool for service discovery and health checking needs.

Conclusion

All of the tools are based on similar principles and architecture. They run on nodes, require quorum to operate and are strongly consistent. They all provide some form of key/value storage.

Zookeeper is the oldest of the three and the age shows in its complexity, utilization of resources and goals it's trying to accomplish. It was designed in a different age than the rest of the tools we evaluated (even though it's not much older).

etcd with **Registrator** and **confd** is a very simple yet very powerful combination that can solve most, if not all, of our service discovery needs. It also showcases the power we can obtain when we combine simple and very specific tools. Each of them performs a very specific task, communicates through well established API and is capable of working with relative autonomy. They themselves are **microservices** both in their architectural as well as functional approach.

What distinguishes Consul is the support for multiple datacenters and health checking without the usage of third-party tools. That does not mean that the usage of third-party tools is bad. Actually, throughout this blog we are trying to combine different tools by choosing those that are performing better than others without introducing unnecessary features overhead. The best results are obtained when we use right tools for the job. If the tool does more than the job we require, its efficiency drops. On the other hand, tool that doesn't do what we need it to do is useless. Consul strikes the right balance. It does very few things and it does them well.

The way Consul propagates knowledge about the cluster using gossip makes it easier to set up than etcd especially in case of a big datacenter. The ability to store data as a service makes it more complete and useful than key/value storage featured in etcd (even though Consul has that option as well). While we could accomplish the same by inserting multiple keys in etcd, Consul's service accomplishes a more compact result that often requires a single query to retrieve all the data related to the service. On top of that, Registrator has quite good implementation of the consul protocol making the two an excellent combination, especially when consul-template is added to this picture. Consul's Web UI is like a cherry on top of a cake and provides a good way to visualize your services and their health.

I can't say that Consul is a clear winner. Instead, it has a slight edge when compared with etcd. Service discovery as a concept as well as the tools we can use are so new that we can expect a lot of changes in this field. Have an open mind and try to take advises from this article with a grain of salt. Try different tools and make your own conclusions.

This entry was posted in Continuous Integration, Delivery and Deployment, Docker and tagged confd, Consul, consul-template, Docker, etcd, Registrator, Service Discovery, Zookeeper on September 8, 2015 [http://technologyconversations.com/2015/09/08/service-discovery-zookeeper-vs-etcd-vs-consul/] by Viktor Farcic.

14 thoughts on “Service Discovery: Zookeeper vs etcd vs Consul”

alp
September 15, 2015 at 9:13 am

It’s pretty confusing that you put 4 “Docker” labels inside one node, it totally looks like you’re running 4 docker engines on a machine. You probably mean containers, but it’s hard to get that. There was plenty of space to add “…Container” to those boxes as well.

Viktor Farcic Post author
September 15, 2015 at 7:59 pm

You’re right. Naming those boxes “Docker” is wrong. I changed it to “Container”. Thank you for spotting this.

alp
September 15, 2015 at 8:04 pm

Also, what do you think about service discovery in dynamic port assignments. For instance when you run `docker run -P redis` the service starts on some random high port, but most Service Discovery software gives you DNS A records (for instance http://redis.mesos) and that only contains IP address. Some offer SRV records (which also contains port number of the service) however nearly none of the languages/libraries out there supports looking up for SRV records first to resolve the port number and then A records. Therefore, service discovery (unless services are run on some predefined static ports) is still a bit crippled experience unless you use some networking layer like weave, calico etc.

Viktor Farcic Post author
September 15, 2015 at 8:17 pm

Registrator will store all IP/port combinations of all containers to etcd, Consul or SkyDNS2 (never tried the last one). IP address without a port is close to useless if you're having more than one container deployed to a single node. Running services on predefined ports quickly becomes hard to manage. That's why Registrator works like a charm. There's also VulcanD. It's an interesting solution but I have impression that it takes too much of the control out of our hands (at least in the flow I'm using).

I might have misunderstood your comments and what I wrote doesn't make sense. If that's the case, please let me know.

alp
September 15, 2015 at 8:21 pm

It totally makes sense, but then your application needs to be aware of service management (in the sense it needs to query etcd/consul to learn the IP:port), is that right? I think so because applications/libraries don't resolve SRV records from FQDNs. So when you say 'new RedisClient("redis.mycluster")', say, Java, will resolve the IP addr of redis.mycluster; and not the port number. This sort of creates a close coupling between the infrastructure you are using and the service. You can read more on SRV records here: http://progrium.com/blog/2014/08/20/consul-service-discovery-with-docker/

Quoting from there:

> *"Built-in DNS resolvers in our environments don't lookup SRV records, however, the library support to do SRV lookups ourselves is about as ubiquitous as HTTP. This took me a while to realize. It means we all have a client, even more lightweight than HTTP, and it's made specifically for looking up a service."*

Viktor Farcic Post author
September 15, 2015 at 8:41 pm

All requests to my applications go through nginx/HAProxy. When a container is run, Registrator stored its IP/port combinations to Consul/etcd. From there on, Consul-Template/confd update nginx/HAProxy configurations and reload them. In both cases, they let existing requests finish and route all new ones according to the new configuration. I combine that with blue-green deployments.

The flow is more or less as follows:

1. First release (blue) is running and nginx is routing all requests to it.
2. New release is deployed (green)
3. nginx configurations are updated with new IPs and ports.
4. nginx is reloaded.
5. The first release (blue) is stopped after all requests to it are served.
6. Repeat from point one for next releases.

All communications between services and from the outside are going through nginx. I rarely use direct linking between containers (only in very simple settings). In other words, service A would make requests to service B directly but through a fixed address on a fixed port (i.e. 80) and nginx would make sure that request reaches the real destination (IP/port multiplied if there is load balancing).

The Pogrium article you referenced is old. I read it at the time but a lot has changed since then. The whole Docker ecosystem, service discovery and other tools (mostly related to microservices) is evolving very fast and many new tools appeared since a year ago when that article was written. For example, the first (buggy, not production ready) release of Consul Template was done few months after that article. Registrator came more or less at the same time. ConfD is a bit older but it took a while until the combination of with other tools we use today become available (mostly thanks to CoreOS effort). I'm not trying to trash the article. I'm just trying to say that its old and that we live in a rapidly changing world when something is considered old only a year after it came into being

alp
September 15, 2015 at 9:54 pm

Thanks, I guess that's the answer I wanted to hear. I was wondering how you use nginx/HAProxy for service discovery purposes. It doesn't seem to be covered in your blog post so I couldn't guess. Are you using those for TCP load balancing (so not just HTTP)? Where are you setting up the nginx/HAProxy? How do services know where this server is? Do you still use DNS (and point all records to nginx/HAProxy somehow)? For instance if you want to discover a redis container running on a random port, how does it happen from a microservice's point of view? Where does it talk to and discover that redis container in the distributed setup? BTW thanks for the discussion I really appreciate, you should blog more on this.

Viktor Farcic Post author
September 15, 2015 at 10:32 pm

Hi,

Actually, this article is only part of the chapter in the book I'm writing. If you send me your email (you can find mine in the about section), I can send you the whole chapter. You'd get the practical hands-on part that is missing from this article and I'd get someone to review it for me (without any obligation for the feedback). How does that sound?

alp
September 15, 2015 at 10:37 pm

Sounds great. ahmetb$microsoft$com. Thanks.

Marcos
October 30, 2015 at 12:31 am

Through getting them to sign up for one thing (permission-based marketing) after which maintaining them in your loop
or inner group.

Rizwan
November 18, 2015 at 6:19 am

Wonderful article. Would love to get my hands on your book. When is it coming out?

Viktor Farcic Post author
November 19, 2015 at 8:24 am

The book is a huge project and it'll take a bit more time until we finish it. It will be out in March 2016.

rizwankh

November 19, 2015 at 9:07 am

Any draft version that I can whet my appetite with. I am willing to pay for it.

Viktor Farcic Post author

November 19, 2015 at 10:34 am

How about following. I'll send you drafts and you send me feedback? Can you please send me an email (mine is in the about section) so that we can continue correspondence outside the blog?