**HashiCorp Consul**

Microservices and other distributed systems can enable faster, simpler software development. But there's a trade-off resulting in greater operational complexity around inter-service communication, configuration management, and network segmentation. [HashiCorp Consul](https://www.hashicorp.com/products/consul) is an open-source tool that solves these new complexities by providing service discovery, health checks, load balancing, a service graph, mutual TLS identity enforcement, and a configuration key-value store. These features make Consul an ideal control plane for a service mesh.

In this video, HashiCorp co-founder and CTO Armon Dadgar gives a whiteboard overview of the following topics:

* An introduction to monolithic vs service-oriented architectures
* Service discovery in a monolith
* Service discovery challenges in a distributed system and Consul's solution
* Configuration management in a monolith
* Configuration challenges in a distributed system and Consul's solution
* Network segmentation in a monolith
* Network segmentation challenges in a distributed system and Consul's solutions
* The definition of "service mesh"

**Transcript**

Hi. My name is Armon, and today I wanted to talk about an introduction to Consul.

When we look at traditional architectures for delivering an application, what we have is kind of the classic monolith. When we talk about the monolith, it's a single application that we're deploying, but it typically has multiple, discrete subcomponents. As an example, suppose we're delivering the desktop banking application, it may have multiple sub-pieces where subsystem A is—let's say—the login to the system. Subsystem B might be showing the balance of our account. C might be wire transfer. D might be foreign currency. Now, even though these are independent functions—logging in versus showing our balance—we're delivering it and packaging our application as a single, monolithic app. So, we're deploying it as a single unit.

Now, what we've seen over the last few years is a trend away from this. The challenge with this is: Suppose there's a bug with our login system. We can't just patch that bug in this system and just change A. We have to coordinate with all of these groups and redeploy the application as a single unit. To fix this, what we'd like to do is instead deploy them as discrete services. This is what might be called a microservices or service-oriented architecture. Basically, we're taking the same monolithic application and taking all of these subcomponents and now delivering them as a discrete application. So now if there's a bug in A—let's say our login system—we can just patch and redeploy A without having to coordinate across these different systems.

What this really buys us is a set of development agility. We don't need to now coordinate our development efforts across many different groups. We can develop independently and then [deploy at whatever cadence we want](https://www.hashicorp.com/blog/using-consul-at-bol-com-the-largest-online-retailer-in-the-netherlands-and-belgium). So, A might want to deploy on a weekly basis, while D might want to deploy on a quarterly basis. This has great advantages for our development teams. The challenge is there's no such thing as a free lunch. What we've gained in development efficiency, in many cases introduces many operational challenges for us. So, let's go through some of those.

**Service discovery in a monolith**

The first one, the most immediate, is discovery. What I mean by that is: Let's say service A wants to call service B. The way you would traditionally do this (in a monolithic app) is service B would expose a method, mark it as public, and then service A can just call it. They're in the same application. It's just a function call. So, when A is calling a function in B, this takes nanoseconds. We're doing an in-memory jump, and it's all in-process so we don't worry about: What happened to our data, how did the data get there, did we encrypt it? It's an in-memory function call.

All of that changes as we come into this distributed world. So now we have a system A that wants to talk into system B. Well, where is system B? It's no longer running on the same machine. It's no longer parts of the same application, and because we're going over a network, it's no longer nanoseconds. We can measure the latency impact in milliseconds between these notes.

**Service discovery challenges in a distributed system**

This first level problem is what we call discovery. How do these different pieces discover one another? There are a few approaches to this. Historically, what we would have done is probably front every one of these services with a load balancer. So, we'd put a load balancer in front of every service here, and then we'd hard-code the IP address of the load balancer. So, A hard codes the IP of the load balancer, and then the load balancer deals with the fact that there might be multiple instances of B. This allows A to skip discovery by hard coding this address, but it introduces a few different problems for us.

The first problem is—we now have a proliferation of load balancers. Here (in a monolithic app), it was sort of a different world. There was a limited number of applications that were packaging many different units of functionality as part of one app. So, there was probably still a load balancer over here (on the monolith), but we had one load balancer managing many different services, whereas here (in the service-oriented app) there's an explosion in the number of load balancers we have. So, these are representing additional costs that we now have.

The second level challenge is: We've introduced a single point of failure all over our infrastructure. So even though we're running multiple instances of B for availability, A is hard coding our load balancer. If we lose the load balancer, it doesn't matter that there are multiple instances of B. Effectively, that whole service has just gone offline.

The other challenge is: We're adding real latency. Instead of A talking directly to B, A is talking to a load balancer which is talking to B and the same path on the way back. So, we're actually doubling the network latency involved with every hop.

The final challenge is: These things tend to be manually managed; the load balancers. So, when I bring up a new instance of B, I file a ticket against the team that's managing the load balancer and wait days or weeks for that thing to get updated before traffic reaches my node. So, all of these are a problem.

**Solution: A central registry**

The way we think about it in Consul is: How do we solve this by providing [essential service registry](https://www.hashicorp.com/blog/spaceflight-uses-hashicorp-consul-for-service-discovery-and-real-time-updates-to-their-hub-and-spoke-network-architecture)? Instead of using load balancers, when these instances boot, they get registered as part of the central registry, so it gets populated in here (in the registry). So, we make a register, and now when A wants to discover and communicate with B, it queries the registry and says, "Where are all the upstream instances of this service?" And now instead of going through a load balancer, service A can directly communicate with an instance of B.

If one of the instances of B dies or has a health issue, the registry will pick that up and avoid returning that address to A. So, we get that [same ability of load balancers](https://www.hashicorp.com/blog/load-balancing-strategies-for-consul) to route around failures without actually needing a load balancer. Similarly, if we have multiple instances of B, we can randomly send traffic to different instances and load level across all of them. So, we get the same advantages of failure detection and load leveling across multiple instances without having to deploy these central load balancers.

The other side of it is—now we don't need these manually managed load balancers everywhere, so instead of having a proliferation of load balancers and waiting days or weeks, the moment an instance boots up, it gets programmatically put into the registry, and it's available for discovery and traffic routing. This helps simplify doing a service-oriented architecture at scale.

**Configuration management in a monolith**

The second big challenge we run into is configuration. When we looked at the monolith, what we probably had was a giant XML file that configured the whole thing. The advantage of this is that all of our different subsystems, all of our components, had a consistent view of the configuration. As an example, suppose we wanted to put our application in maintenance mode. We wanted to prevent it from writing to the database so that we could do some upgrades in the background. We would change this configuration file and then all of these subsystems would believe that we're in maintenance mode simultaneously.

**Configuration challenges in a distributed system**

Now, when we're in this world (the service-oriented app), we've sort of distributed our configuration problem. Every one of these applications has a slightly different view of what our configuration is. So now we have a challenge here: How do we think about configuration in our distributed environment?

**Solution: A central Key-value store**

The way Consul thinks about this problem is—instead of trying to define the configuration in many individual pieces distributed throughout our infrastructure, how do we capture it in a [central key-value store](https://www.consul.io/intro/getting-started/kv.html)? We define a key centrally that says, "Are we in maintenance mode?" And then we push it out to the edge and configure these things dynamically. Now we can change the key centrally from, "Are we in maintenance mode? False to true". And push that out in real time to all of our services, giving them a consistent view—moving away from having a sharded distributed configuration everywhere to defining it and [managing it centrally](https://www.hashicorp.com/blog/twelve-factor-applications-with-consul).

**Network segmentation in a monolith**

The third challenge is when we looked at this classic monolithic architecture, we would divide our network traditionally into three different zones.

* We'd have zone one, which was our wild, demilitarized zone: Traffic coming in from the public internet.
* Then we have our application zone, which was largely receiving traffic from the DMZ through a set of load balancers.
* Then we probably had a data zone behind us or a private zone.

Only the load balancer could reach into the application zone, and only the application zone could reach into the data zone. So, we had a pretty simple, three-tier zoning system that allowed us to segment our network traffic.

**Network segmentation challenges in a distributed system**

As we look at this world (of the service-oriented app), this pattern has changed dramatically. Now there's no longer a single, monolithic application within our zone, but many hundreds or thousands of unique services within this application zone. The challenge is—their traffic pattern is much more complicated now. Its many services have a complicated [east-west traffic flow](https://networkengineering.stackexchange.com/questions/18873/what-is-the-meaning-origin-of-the-terms-north-south-and-east-west-traffic). It's no longer sequentially from load balancer to application to database. Traffic might come into either—let's say—our desktop banking app, our mobile banking app, or our APIs. There might be multiple front doors depending on the access pattern, and these services communicate with each other in a complex east-west traffic flow. This third level challenge now becomes: How do we think about segmenting this network? How do we partition which services are allowed to talk to which other services?

**Solution: A service graph**

This third challenge becomes segmentation. The way Consul deals with this are with a feature we call "Connect". So again, centrally managing the definition around who can talk to whom. What this starts with is a few different components.

First, we start with what we call a service graph. With the service graph, we define—at a service level—who can communicate. So, we might say, '"A' is able to talk to B". We might say "C is allowed to talk to 'D'". And what you'll notice is we're not talking about IP to IP. We're not saying IP 1 can talk to IP 2. We're talking about "service A can talk to service B".

The nice thing about expressing that at this layer is that the rule is scale independent. What I mean by that is—if I have a rule that says my web server should be allowed to talk to my database, that might be expressed simply. I can say, "Web talks to database". But if I want to translate that to the equivalent firewall rules, well, if I have 50 web servers and I have five databases, that translates to 250 different firewall rules. So, this is what I mean. This rule is scale-independent, it doesn't matter if I have one, 10, 50 or 1000 web servers, it's the same rule. Firewall rules are the opposite. They're very much scale-dependent and tied to the management unit which is an IP. So, let's elevate that management up to this logical level where we don't really have to be tied to the actual scale.

**Solution: Mutual TLS**

The next part of this is how do we assert identity? This comes from a certificate authority, so when we say service A can talk to service B, how do we know what is service A and what is service B? The approach Consul Connect takes is to tie this back into a very well-known protocol, TLS. We issue TLS certificates that uniquely identify these services. So, we can uniquely say, "This is service A, and this is service B. Unlike saying, "There's an IP and we don't actually know what's running at that IP with any strong guarantee".

How do we actually enforce this? This translates into a set of proxies. The way we end up implementing the access control is through mutual proxy. So, on a box we might have service A and on that same box, we're running a proxy alongside it. This is sort of a sidecar proxy. And then similarly for service B, it's running on its own machine or its own container, and it also has a sidecar proxy. Now when A wants to communicate to B it's transparently talking to this proxy which is establishing communication on the other side to another proxy. That side terminates the connection, and hands it off to B.

This actually has a few different advantages. First, we're not modifying the code of A and B. They're both blissfully unaware that anything has changed. They're just communicating in the way they normally do. The proxies, on the other hand, are using these certificate authorities. So, the proxy on side A will use this certificate to say, "I am A and I'll verify the identity of B and vice versa". The proxy on B side will verify that it's talking to A. So now, we get this strong sense of identity between the two sides. And this is being done with mutual TLS.

The second advantage of using mutual TLS is now we establish an [encrypted channel](https://www.consul.io/docs/agent/encryption.html) between them. This is becoming increasingly important as we talk about regulations like [GDPR](https://www.hashicorp.com/gdpr). Increasingly our focus is on saying, "You know what? We don't actually trust our own network within our data center. We can't just assume that by virtue of being on the network, things are trusted". So as part of this shift, we're increasingly seeing a mandate to encrypt our data at rest—things that we're writing to our databases or writing to our object stores—but also data in transit. So as data is going in between our web application and our database, is it being encrypted? As it's flowing in between different services in our data center, are we encrypting that traffic?

The challenge is—we probably have many hundreds or thousands of applications that exist and are not TLS aware. So, the advantage of imposing it at the proxy layer is that we can get that guarantee of our data being encrypted in transit without needing to re-implement all of these applications.

The third piece of this is—just because A can prove it's talking to B, and B can prove it's talking to A, that's not enough because it's not clear that A should even be allowed to talk to B. This where the service graph comes in. The proxy is called back into the service graph, and looks for an arc like this—Is there a rule that allows service A to talk to service 'B?' If So, then the proxies allow that traffic to take place, A is allowed to talk directly to B, and they're none the wiser that this intermediate proxying is taking place.

**Consul and the service mesh**

So, as we come back and talk about this transition, what we are really trying to do is gain a set of developer efficiency by splitting our monolith and developing these services independently. We want them to be able to develop and deploy and be managed independently, but we've inherited a set of operational challenges. This goes back to our "no free lunch".

As we came into this world (of distributed systems), we now need to figure out how do we do service discovery, how do we configure in the distributed setting, and how do we segment access so it's actually safe to operate this distributed infrastructure? This set of challenges collectively are what we really refer to as a [service mesh](https://www.hashicorp.com/blog/smart-networking-with-consul-and-service-meshes). So, when we talk about Consul, what it's trying to do is provide this service mesh capability, which underneath, is three distinct pillars in service of allowing this microservice or service-oriented architecture to work.

I hope that's been a helpful introduction to Consul. There's a lot more detail on our website. Please check out our other resources. Thank you so much.