SUE B.V.
De Ooyen 9
4191 PB Geldermalsen
+31 345 656 666

MDP Research Document

How to deal with latency-sensitive workloads in a global multi-cloud and multi-cluster Kubernetes-based platform?

Version 1.2

12-01-2022

Table of contents

The Ideal Scenario

An application can be deployed globally where the effect of (network) latency towards the end-user is brought to a minimum. This should be done on a global multi-cloud and multi-cluster Kubernetes-based system.

The worst-case scenario would be for instance where PHP app 1 in the EU is connecting to SQL DB 1 in the US and has a 3-second delay.

After the initial interview with the client, it was agreed upon to use AWS and Azure as platforms.

At a minimum, a quality answer to the question should be given. The ideal scenario would be to implement a working proof of concept.

Latency, jitter, and other measurements

Multiple online spaces have been researched to come upon a good measurement of latency. But first, a short description of terms and some causes.

Latency

Latency (RFC1242, 3.8) is the amount of time it takes for a data packet to go from one place to another. For example, Server 1 in Tokyo sends a data packet to Server 2 in Amsterdam. The Tokyo server sends the packet at 00:00:00.000 UTC, and the Amsterdam server receives it at 00:00:00.243 UTC. The latency is the difference in time, for this example 0.243 seconds or more commonly annotated as 243 ms (milliseconds).
Latency can exist between servers, but also between a client and e.g. a data centre. The measurement of latency helps developers, engineers, clients, and businesses understand how quickly something will load for users, or be sent between servers.
One of the principal causes of latency is distance, either between servers or between client and server. If a client residing in Hamburg would request a server in Quebec, there will almost always be more latency than if the request went to a server in Frankfurt.
After a request is made, the client usually needs a response. The time between the initiation of the request and the response is called the Round Trip Time (RTT for short). The RTT is double the amount of latency since the data travels both upstream and downstream.
Lastly, the packet probably has to traverse multiple networks to get to its destination. The more networks the packet needs to go through, the more delays there are. This can be caused for example by Internet Exchange Points (IXPs), which will always add a few ms to the RTT.

Acceptable measurements

Multiple different sources state multiple different values as to be ideal, below we will explore some of them.
Cloudalize states that < 60ms is strong, 60 - 150 ms is weak, and 150+ is poor.
Ideally, as latency is the main focus, we want it to be around 10–15, which is good in e.g. the gaming industry where latency is one of, if not the most important metric they measure.

Jitter

Jitter (RFC3393) is essentially the deviation of latency. For example, during your connection, your requests have a measured latency range between 30 and 70 ms. In this case, the jitter would be 40 ms. In this case, the connection would be called "jittery", and data requests/sends might get out of sync. The result is a choppy and/or unstable connection, a good example is a bad VoIP call or terrible video conference. Jitter can be caused by multiple phenomena, for example, variation in the routing and forwarding of packets. It can be caused by misconfiguration, but most of the time a busy network causes this. This will cause latency to vary, and thus cause jitter.

Acceptable measurements

Multiple different sources state multiple different values as to be ideal, below we will explore some conclusions made out of them.
Cloudalize states that < 10 ms is stable, and multiple different sources state that anything over 30 ms will cause issues.

Other important concepts

Bandwidth is the maximum amount of data that can pass through the network.
Throughput is the average amount of data that passes over a given amount of time. It is not always equivalent to bandwidth as it is affected by latency and other factors. Latency is a measurement of time, regardless of data size. A small file can have the same latency as a large file, but it will for example be handled faster causing the response to be more swift.

Conclusions

From this research we can conclude several points to take into account in our answer and possible designs:

1. Distance between client and server should be minimized so that latency is low.
2. Distance between server and server should be minimized so that latency is low.
3. The data packet should be sent through as few networks as possible so that latency is low.
4. The network should not be too congested, as routing might change and add a jitter.
5. Low-latency data and large/slow data should be treated differently in our system.

Cluster-centric

In this section, we will dive into the research done from a cluster-centric perspective. This includes the hardware considerations (location and resources), and the network/connectivity between them.

Hardware/resources

Since the application is Kubernetes-based, we will need to look for the proper hardware to host it, and also look at the resources this hardware has available.

Locations

As the platform is global, we need to think carefully about where we are going to place the servers.
Firstly, a general point can be made about creating a global network.
A global network is often not truly global for the user. Let's take Google Maps as an example; when you are in Germany, there is no need to connect to an Indonesian server, the latency would be way too much, and there would be little to no benefit from it. This remains true the other way around. Thus; there is no need to have all your data, everywhere, at all times. This is a large-scale example, but the same goes for smaller scales. From this, we can conclude two things:

1. We do not need global data replication.
2. If data is needed in a different continent/region, it should be transferred to that specific place.

Earlier, in the "Latency, jitter, and other measurements" conclusions we stated that;

1. the distance between the client and server must be minimized
2. the distance between servers should be minimized
3. the data should go through as few networks as possible.

Consideration 1 can easily be solved by placing multiple servers across the world, per continent. Let's say Europe for example; you'd place one instance in Frankfurt, one in Paris, one in London, and so on.
This also (partially) solves consideration number 2, because this way, servers are close to each other. The question is, how do you handle data retrieval? You could create a full-fledged database for each location, but this might be unnecessarily expensive. It also could slow down the system as a whole depending on how your data fetching process is done.

Instead, we have several other ideas:

1. You could create one centrally placed database in every region, where you'd measure the latency between all servers in that region to find the best option. Every location within the region would query this database for their lookups. If that data was needed in another region, it can be transferred there.
2. You could create the same system, but instead of every location querying the central DB, you create a sort of 'cache memory' at every location. This way, the more common lookups can be done locally, and when more extensive data is needed, the system can query the main/central database for that region. If your data is needed in another region, it can be transferred there. This one also solves conclusion 3.

How would you determine the best place to place the central DB?

TARGET >>> \/\/SOURCE	**Europe Frankfurt** *eu-central-1*	**Europe Paris** *eu-west-3*	**Europe Milan** *eu-south-1*	**Europe London** *eu-west-2*	**Europe Stockholm** *eu-north-1*	**Europe Ireland** *eu-west-1*
Europe Frankfurt *eu-central-1*	0.0	10.5	11.1	15.6	28.5	26.2
Europe Paris *eu-west-3*	10.2	0.0	20.5	9.0	29.8	20.6
Europe Milan *eu-south-1*	11.4	20.2	0.0	28.5	30.5	31.8
Europe London *eu-west-2*	16.7	9.9	27.8	0.0	32.4	11.3
Europe Stockholm *eu-north-1*	21.1	30.7	30.8	32.5	0.0	42.8
Europe Ireland *eu-west-1*	25.8	18.0	36.8	13.0	41.6	0.0

Use the AWS Inter-Region Latency Map to calculate which **target** has the lowest **total** latency.

85,2	89,3	127	98,6	162,8	132,7

In this case, it is Frankfurt. This would be the best place for the central European database, as this is where the most requests would have the lowest latency. Do this for every continent, and make sure it is compliant with your maximum latency needs.

Kubernetes

To come upon an idea on how to correctly create a Kubernetes multi-cluster-based platform, research had to be done on what exactly Kubernetes is and how it works since none of us have any experience with it, at the start of this project.
Kubernetes (k8s, Kube) is a container orchestration platform that automates the processes of deploying, maintaining, and scaling a containerized infrastructure.
A Kubernetes cluster needs at least one master node which will control the worker nodes. This node will also contain a file, connecting the workers to the master node (this is possible by external IP address so different cloud providers do not have any influence).

K8S is a platform that was first developed by a Google team, but later they donated it to the CNCF. It is an open-source platform that can automate Linux container operations. It removes most of the manual processes that come with deploying and scaling containerized applications. It effectively allows users to cluster groups of hosts running Linux containers and manage them quickly and efficiently

Kubernetes supports clusters with up to 5000 nodes, and is designed to accommodate configurations that meet all of the following criteria:

- No more than 110 pods per node
- No more than 5000 nodes
- No more than 150000 total pods
- No more than 300000 total containers

Manual K8s in AWS & Azure

For amazon web services, using EC2, it is possible to create instances, make one into a master node, and then create worker nodes. The master can then also control worker nodes in Azure VMs. This way, you can control both cloud providers' worker nodes using one master node. See this how-to (Almeida, 2022) for more.
Some benefits are;

1. Control over instance type: you'd want to choose c3 or c4 instances, as these are optimized for networking.
2. Control over instance placement: When you manually create instances, you can choose to launch them in different availability zones or regions. This can help to reduce latency by ensuring that your instances are physically close to the users or services that are interacting with them.
3. Granular control over network configurations: you can customize network configurations such as security groups, network ACLs and VPCs. This way, you could better optimize network traffic and minimize the hops the data needs to travel.
4. Fine-grained scaling: more flexibility to scale up or down specific instances. This causes you to have more control over capacity, and thus have better control over latency.

Most of the downsides stem from manual creation;

1. Higher operational overhead: this can lead to increased latency when dealing with scaling, updates, maintenance, or troubleshooting.
2. Limited automatic scaling: when these instances are created manually, you have to either create an external service when scaling or do it manually. This can lead to latency issues, if not done properly.

AWS EKS / AZURE AKS

These 2 services are built for deploying, maintaining, and auto-scale Kubernetes clusters in AWS or Azure, this makes it possible to easily set up a Kubernetes cluster without the need for the configuration making this process a lot easier.

- Runs and scales the Kubernetes control plane across multiple AZs to ensure high availability
- Acts as a load balancer, and detects unhealthy control plane instances.
- Works together with many AWS services to provide scalability and security.
- Always runs up-to-date versions of the open-source Kubernetes software. It can easily be combined with existing plugins and tooling.

EKS control plan architecture

EKS runs a control plane for each cluster. The control plane consists of at least two API Servers and three etcd instances, which are used to store and manage the critical information the system needs to keep running.
VPC network policies are used to restrict traffic between control plane components within a single cluster. This way control plane components cannot view or receive communication from others, except when this is authorized with Kubernetes policies. When using EKS, a user will provide the worker nodes and link them to Amazon EKS endpoints. AWS then handles all management tasks for the Kubernetes control plane, including upgrades, patches, and security configurations.

Both services have good SLAs and high availability of 99.5%. This ensures high availability of the service and low downtime for the end user. AKS has one extra feature worth mentioning. That's the automatic and continuous monitoring of the pods in the cluster. If it detects a bad node it will automatically repair the pod.

Hardware Conclusions

This is what we decided on as our answer to this part of the question (hardware);

Firstly, we will keep in mind the locations of our hardware, and place it close to the users. There, we will keep a local cache in each, and one central database per region. The location of this database should be determined by calculating the best spot per region, so that every 'slave' or instance with a memory cache, has as little latency as possible. If the data is needed in another region, it can be transferred there.
Furthermore, we decided to work on AWS EKS and Azure AKS, as we felt these had the most compelling benefits concerning latency. The other option felt a bit too weak and unstable in this field.

Networking/Connectivity

In this section, we discuss the different options for connecting clouds and their pros and cons.

VPN Connection

The first idea is to have an Azure VPN gateway talk with an AWS Virtual Private Gateway. By default, the AWS VPN connection will have 2 Public IPs, one per tunnel. Azure doesn't have a default for this, so we'll be using Active/Passive from the Azure side. This will allow for a highly available VPN connection. This is the most common method of connecting clouds.

There are some cons to this method, though. One con is unpredictable routing through the public internet, which can cause higher latency and result in poorer application performance. Another con is limited throughput, which is caused by having multiple tunnels to support the load. It results in a lot of time wasted with load balancing and ensuring that the VPN tunnels don't get clogged and fail. One of the biggest cons is the data transfer fee between AWS and Azure. As a result of data travelling the public internet to get to the other cloud, the costs are calculated per GB by the cloud providers which can get very costly.

Virtual Router Connection

The second idea is to set up private connectivity with a virtual router. A virtual router has the functionality of a physical router, but with the benefit of it being virtualized in software to run on nonvendor-specific hardware. With a private virtual router, a user can get security, reliability, and private connectivity at a lesser cost as data won't have to travel the public internet.

Like all methods, some cons come with it, the main one being that running a virtual router on a server affects performance. If you use an operating system of a router in a virtual machine, the throughput of the virtual router is affected drastically.

Private Lines Connection

The third idea is to build private lines. By building private lines to the cloud providers using dedicated circuits, you can gain a private connection with traffic that doesn't have to pass over the public internet. It is a more reliable method of connection than a VPN connection, at the cost of latency.

Some cons come with this method too, like latency. Even with private circuits to the cloud providers, there is still the need to backhaul traffic to your data centre. Another con is that it is far more expensive to create this connection than the other methods. It is the most reliable method but with the latency issues, I don't think it is worth the cost.

Istio Mesh Connection

The fourth idea is to use Istio mesh to connect the two clouds. Istio is a service mesh, which is a dedicated infrastructure layer that can be added to microservice applications, which are applications created as collections of microservices performing their functions. It allows users to add capabilities like observability, security, and traffic management.

There are some cons to this method. For starters, the service mesh is a relatively new technology. This comes with the usual problems of the documentation being outdated, lack of forums or helps with any problems or errors, different versions being installed on different products, open issues on GitHub, etc.

Route53 Connections

The fifth method is using Route53 mapping to both AWS and Azure. Route53 is a highly available and highly scalable cloud DNS service provided by AWS. It is very user-friendly, easy to use, and easy to manage. The registration of domains is also a lot cheaper than other domain registrars.

Some of the cons of this method include the inability to handle multiple requests simultaneously, which is inconvenient if you have a lot of users. Another con is that there are no detailed logs, instead, users must use CloudTrail to get detailed logs from route53. Another con is that the third-party integration is a bit slow, which makes exporting anything outside of the solution will slow it down.

Liqo

Liqo is an open-source tool that makes it possible for users to enable dynamic and seamless Kubernetes multi-cluster topologies. It has a relatively small and new team behind it, with the deployment of v0.2.1 only happening in 2021
Liqo provides a multitude of services, like peering between clusters, seamless offloading of workloads to remote clusters, an extensive network fabric that allows for a pod to pod or pod-to-service connections, and a native storage fabric that supports the remote executions of workloads by utilizing the data gravity method

Liqo aims to create a near-to-zero-config and straightforward way to create a multi-cluster topology compliant with the most used CNIs (Calico, Cilium, etc.). Liqo allows you to connect to different clusters, either on-prem or in the cloud (AKS, EKS, GKE).
It does not require any patch to an already deployed cluster. It allows users to control resources located in remote clusters by merely accessing the standard Kubernetes objects in the local cluster.
An example scenario you could use Liqo is:

- Discover and join with the remote cluster, this lets the remote cluster act as an extra node.

Networking Conclusions

Out of all the methods that have been researched, Liqo could work the best for this problem. Using its inter-cluster communication (powered by Wireguard). This does the same as our initial idea of setting up a VPN peering service, but this time it’s done by a provider who has experience in this field.

Application-centric

At the application level, we can make some adjustments to lower the latency.

Improving bandwidth

Bandwidth is the channel through which all traffic travels to the internet so any improvement or optimization would improve latency. There are many cases at the application level where the bandwidth to the internet is shared with different endpoints in the network. This is why if we add more bandwidth we will notice an improvement in latency at the application level. Another method that is closely related to bandwidth is the use of different channels to manage priority traffic on our network. This means that the traffic we mark as privileged will have more bandwidth when it is sent over the network.

Networking devices

Another common reason for high latency is the number of hops the packet has to make from one point to another. If the network packet needs to travel through 20 devices (routers, switches, etc...) it will be slower than if it only needs to travel through the cable. Therefore, if the number of network devices could be reduced, latency would be improved. It should be noted that this also involves the distances that the network packet must travel as it is not the same to travel 1km as 100 km. The shorter the distance and the fewer device hops the packet makes, the less latency there will be.

Cilium

The foundation of Cilium is a background process that runs in user space and performs the work of generating and compiling BPF programs, as well as interacting with the runtime that provides the containers.
In the form of BPF programs, systems are implemented to ensure container connectivity, integration with the network subsystem (physical and virtual networks, VXLAN, Geneve), and load balancing.
The background processing is complemented by an administration interface, an access rules repository, a monitoring system, and integration modules with support for Kubernetes, Mesos, Istio, and Docker.
The performance of a Cilium-based solution with a large number of services and connections is twice ahead of tables-based solutions due to the high overhead of rule lookup.

General Conclusion

For the research question, we opted for researching latency optimization for a global multi-cluster multi-cloud Kubernetes environment.

For hardware, we place clusters near every user, each having a local cache. Then we place a central database per region. You'd place the central database where overall, there is the least amount of latency within the region. Common requests will be made to the cache in the nearest region, and larger/uncommon requests will be made to the central database. You should be able to transfer data between central databases if you were to change continents for example. You should use AWS EKS and Azure AKS, as these provide a better scalable full solution. If you wanted to have really granular control over your setup, and for example choose your instance types, you would choose manual creation.

For Connectivity, we had to keep in mind all the services that support both Kubernetes and allow for high redundancy. We came up with Liqo. Liqo allows for inter-cluster communication. This connection is secured by WireGuard, which implements encrypted virtual private networks, and allows for high-speed performance, and low attack surface. The connection between clusters is done through VPN peering, and it does not require any patch to be implemented in an already deployed cluster. It allows users to control resources located in remote clusters by merely accessing the standard Kubernetes objects in the local cluster.

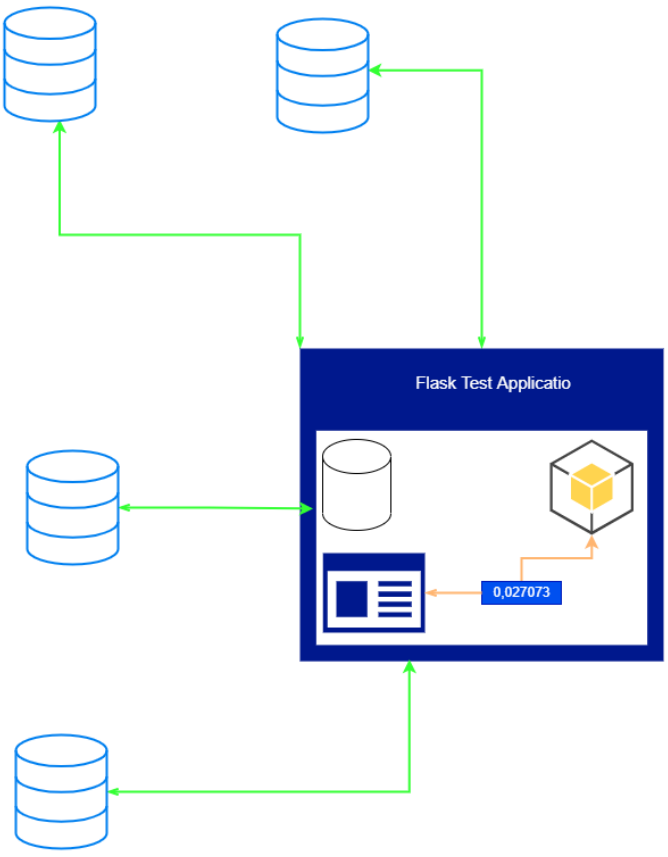
For the Application, the most important part was to keep in mind how to send and receive data processed through the application. We kept in mind how and where the packages travelled and how to be produced on both ends (application and database).

Proof of Concept

This is what was created to test our conclusion to the research question:

Latency testing application

https://git.fhict.nl/I494139/test-app-flask
We conclude that most of the latency comes when querying data from SQL servers like when loading image, video, or just text data,
So, we have developed an application to make simple queries to the database to fetch some data from it and record the latency between the testing application and the databases located in different zones or regions.
The application is written in Flask and uses a local database to save the target IP, username, and password.
From a security perspective, even though this application is used inside the corporation, we have written an encryption mechanism to encrypt and decrypt the password saved in the local database.
The application itself has a median of 0.027073 latency per second between the front end and back end.



As shown above, this is an overview of our test application.

Hardware setup

https://git.fhict.nl/I489369/mdp-sue-s3-group1
We attempted multiple times to get the hardware set up the right way, and in the end, we created some terraform and yaml (for k8s) code. Unfortunately, due to time constraints, we were unable to finish it and get it working the way we wanted it to before the deadline. All code developed is in the repository linked above.

For the testing setup, we wanted to create 2 Kubernetes clusters (one in AWS and one in Azure) in different regions of the world so we could test the latency between them. Also, there will be a centralized database(master database), and cache in the multi-cluster environment to reduce the latency.

Useful resource list

awsping
Console tool to check the latency to each AWS region
https://github.com/ekalinin/awsping

AWS Inter-Region Latency Map
A chart that displays latencies between 18 regions of AWS (China and Gov excl.)
https://latency.bluegoat.net/

Set up Multi-cloud k8s
https://faun.pub/multi-cloud-setup-of-kubernetes-add3cfa9221f

Sources

- Almeida, A. T. A. P. B. &. (2022, 23 January). *How to Build a Multicloud Kubernetes Cluster in AWS and Azure Step by Step*. https://bluexp.netapp.com/blog/cvo-blg-how-to-build-a-multicloud-kubernetes-cluster-in-aws-and-azure

- AWS. (2022, 05 April). What is amazon EKS? https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html

- Azure. (2018, June) Azure Kubernetes Service (AKS). https://azure.microsoft.com/en-us/products/kubernetes-service/#overview

- MGoedtel, SreejaBhattacharya-MSFT, MakendranG, axelgMS, justindavies, chmill-zz, zr-msft, SriHarsha001, v-hhunter, laurenhughes, jluk, arunpnair, mlhoop. (2022, 19 December). Azure Kubernetes Service (AKS) node auto-repair. https://learn.microsoft.com/en-us/azure/aks/node-auto-repair

- Amazon EKS Pricing Calculator. (2023, 11 January). https://calculator.aws/#/addService/EKS

- Azure AKS Pricing Calculator. (2023, 11 January). https://azure.microsoft.com/en-us/pricing/calculator/