Discorso tesi

General overview

Kubernetes is a portable, extensible and open-source platform for managing containerized workloads and services, which can facilitate both declarative configuration and automation. The platform boasts a large and rapidly growing ecosystem. Services, support and tools are widely available in the Kubernetes world.
Container: With Linux kernel tools like Cgroups and namespaces, isolate processes so they can run independently. This independence is the goal of containers: the **ability to run multiple processes and applications separately to make the most of existing infrastructure while maintaining the level of security that would be guaranteed by having separate systems**.

In the context of containers this means that users and groups may have privileges for certain operations inside containers without having them outside. One of the specific goals of containers is to allow a process to have root privileges for operations inside the container, while at the same time making it a normal non-privileged process outside the container.
cgroups allow you to allocate resources (CPU time, system memory, network bandwidth) between user-defined task (process) groups running on a system. It is possible to monitor the cgroups that have been configured, prevent access to certain resources and even reconfigure them dynamically.
Containers are a good way to deploy and run your applications. In a production environment, you need to manage the containers that run applications and ensure that no service interruptions occur. For example, if a container goes down, a new container needs to be started.

Kubernetes provides you with:

- **Service discovery and load balancing:** Kubernetes can expose a container using a DNS name or its IP address. If the traffic to a container is high, Kubernetes is able to distribute the traffic across multiple containers so that the service remains stable.
- **Storage Orchestration:** Kubernetes allows you to automatically mount a storage system of your choice, such as local storage, disks provided by public clouds, and more.
- **Automated rollout and rollback:** You can use Kubernetes to describe the desired state for your containers, and Kubernetes will take care of changing the current state to achieve the desired one at a controlled rate. For example, you can automate Kubernetes to create new containers for your service, remove existing containers and adapt their resources to those required by the new container.
- **Load Optimization:** Provide Kubernetes with a cluster of nodes to run containers. You can educate Kubernetes about how much CPU and memory (RAM) each individual container needs. Kubernetes will allocate containers on nodes to maximize the use of available resources.
- **Self-healing:** Kubernetes restarts crashed containers, replaces containers, terminates containers that don't respond to health checks, and avoids getting traffic to containers that aren't yet ready to respond correctly.

- **Management of sensitive information and configuration:** Kubernetes allows you to store and manage sensitive information, such as passwords, OAuth tokens and SSH keys. You can deploy and update sensitive information and application configuration without having to rebuild container images and without revealing sensitive information in your system configuration.

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.
The worker node(s) host the Pods that are the components of the application workload.
The control plane manages the worker nodes and the Pods in the cluster.
In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

The control plane is actually made up of several components:

- kube-apiserver This is the frontend server for the control plane, handling API requests.
- etcd This is the database where Kubernetes stores all its information: what nodes exist, what resources exist on the cluster, and so on.
- kube-scheduler This decides where to run newly created Pods.
- kube-controller-manager This is responsible for running resource controllers, such as Deployments.
- cloud-controller-manager This interacts with the cloud provider (in cloud-based clusters), managing resources such as load balancers and disk volumes.

The control-plane components in a production cluster typically run on multiple servers to ensure high availability.

The nodes are composed of:

- kubelet This is responsible for driving the container runtime to start workloads that are scheduled on the node, and monitoring their status.
- kube-proxy This does the networking magic that routes requests between Pods on different nodes, and between Pods and the internet.
- Container runtime This actually starts and stops containers and handles their communications.

Historically the most popular option has been Docker, but Kubernetes supports other container runtimes as well, such as containerd and CRI-O.
Recently Docker has been deprecated.

Container runtime

The container runtime is the most important part because is the component responsible for executing the container.
In the recent years there have been numerous attempts at standardization of the format and execution of containers.

- The **Open Container Initiative** (OCI) which publishes specifications for containers and their images.
- The Kubernetes **Container Runtime Interface** (CRI), which defines an API between Kubernetes and a container runtime underneath.

Docker has been instrumental in popularizing containers and has historically been the most popular container runtime for Kubernetes environments. But Kubernetes recently announced that it will deprecate Dockershim—the underlying module that enables compatibility between Docker and Kubernetes—as part of the v1.24 release. This means that organizations will have to migrate from Docker to either containerd or CRI-O, which are the two runtimes that are compatible with the Kubernetes container runtime interface (CRI).

Examples of popular high-level runtimes include:

- Docker (Containerd): the leading container system, offering a full suite of features, with free or paid options. It is the default Kubernetes container runtime, providing image specifications, a command-line interface (CLI) and a container image-building service.
- CRI-O: an open-source implementation of Kubernetes' container runtime interface (CRI), offering a lightweight alternative to rkt and Docker. It allows you to run pods using OCI-compatible runtimes, providing support primarily for runC and Kata (though you can plug-in any OCI-compatible runtime).
- Windows Containers and Hyper-V Containers: two lightweight alternatives to Windows Virtual Machines (VMs), available on Windows Server. Windows Containers offer abstraction (similar to Docker) while Hyper-V provides virtualization. Hyper-V containers are easily portable, as they each have their own kernel, so you can run incompatible applications in your host system.

The OCI includes specifications for sandboxed and virtualized implementations:

- Sandboxed runtimes: provide increased isolation between the containerized process and the host, as they don't share a kernel. The process runs on a unikernel or kernel proxy layer, which interacts with the host kernel, thus reducing the attack surface. Examples include gVisor and nabla-containers.
- Virtualized runtimes: provide increased host isolation by running the containerized process in a virtual machine (through a VM interface) rather than a host kernel. This can make the process slower compared to a native runtime. Examples include kata-containers and the now deprecated clearcontainers and runV

The OCI provides runtime specifications. Runtimes implemented according to OCI specs are called low-level runtimes, because the primary focus is on container lifecycle management. Native low-level runtimes are responsible for creating and running containers. Once the containerized process runs, the container runtime is not required to perform other tasks. This is because low-level runtimes abstract the Linux primitives and are not designed to perform additional tasks.
The most popular low-level runtimes include:

- runC: created by Docker and the OCI. It is now the de-facto standard low-level container runtime. runC is written in Go. It is maintained under moby—Docker's open source project.

- crun: an OCI implementation led by Redhat. crun is written in C. It is designed to be lightweight and performant, and was among the first runtimes to support cgroups v2.
- containerd: an open-source daemon supported by Linux and Windows, which facilitates the management of container life cycles through API requests. The containerd API adds a layer of abstraction and enhances container portability.

Docker and Kubernetes are two tools to execute containers. Kubernetes can use as container runtime *containerd* or *CRI-O*. Both of them implement the CRI, while Docker's container runtime not.
CRI is a Kubernetes API that define how Kubernetes interacts with the container runtime. The higher-level container runtime (containerd or cri-o) use a lower-level container runtime to manage the container lyfecycle that must implement the OCI. One lower-level runtime that implement OCI is *runc*.

Kubernetes resources

Anything we create in a Kubernetes cluster is considered a resource: deployments, pods, services and more. Resources have a descriptor file (the *manifest* file) written in YAML.
YAML is a format for data serialization designed to be understood by humans.

Pods

Pods are **non-permanent** resources, they are destroyed and re-created continously to match the desired state of the cluster.
Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.
A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.
When Kubernetes schedules a Pod to run on a node, it creates a network namespace for the Pod in the node's Linux kernel. This network namespace connects the node's physical network interface, such as eth0, with the Pod using a virtual network interface, so that packets can flow to and from the Pod. The associated virtual network interface in the node's root network namespace connects to a Linux bridge that allows communication among Pods on the same node. A Pod can also send packets outside of the node using the same virtual interface.
Kubernetes assigns an IP address (the Pod IP) to the virtual network interface in the Pod's network namespace from a range of addresses reserved for Pods on the node. This address range is a subset of the IP address range assigned to the cluster for Pods, which you can configure when you create a cluster.
A container running in a Pod uses the Pod's network namespace. From the container's point of view, the Pod appears to be a physical machine with one network interface. All containers in the Pod see this same network interface. Each container's localhost is connected, through the Pod, to the node's physical network interface, such as eth0.

Deployments

Working together with the Deployment resource is a kind of Kubernetes component called a controller.
Controllers are basically pieces of code that run continuously in a loop, and watch the resources that they're responsible for, making sure they're present and working.
If a given Deployment isn't running enough replicas, for whatever reason, the controller will create some new ones. (If there were too many replicas for some reason, the controller would shut down the excess ones.
Either way, the controller makes sure that the real state matches the desired state.)
Actually, a Deployment doesn't manage replicas directly: instead, it automatically creates an associated object called a ReplicaSet, which handles that.
Why doesn't a Deployment just manage an individual container directly?
The answer is that sometimes a set of containers needs to be scheduled together, running on the same node, and communicating locally, perhaps sharing storage.
This is where Kubernetes starts to grow beyond simply running containers directly on a host using something like Docker.
It manages entire combinations of containers, their configuration, and storage, etc. across a cluster of nodes.
The `kubectl create deployment` command didn't actually create the Pod directly. Instead it created a Deployment, and then the Deployment created a ReplicaSet, which created the Pod.

ReplicaSets

A ReplicaSet is responsible for a group of identical Pods, or replicas.
If there are too few (or too many) Pods, compared to the specification, the ReplicaSet controller will start (or stop) some Pods to rectify the situation.
Deployments, in turn, manage ReplicaSets, and control how the replicas behave when you update them.
Valid only for stateless pods.

DaemonSets

Use a DaemonSet when you need to run one copy of a Pod on each of the nodes in your cluster.
If you're running an application where maintaining a given number of replicas is more important than exactly which node the Pods run on, use a Deployment instead

StatefulSets

If we want to deploy a stateful application we probably need a StatefulSet. Kubernetes is a container orchestration tool that uses many controllers to run applications as containers (Pods). One of these controllers is called StatefulSet, which is used to run stateful applications.
Deploying stateful applications in the Kubernetes cluster can be a tedious task. This is because the stateful application expects primary-replica architecture and a fixed Pod name. The StatefulSets controller addresses this problem while deploying the stateful application in the

Kubernetes cluster.
There are several reasons to consider using StatefulSets. Here are two examples:

- Assume you deployed a MySQL database in the Kubernetes cluster and scaled this to three replicas, and a frontend application wants to access the MySQL cluster to read and write data. The read request will be forwarded to three Pods. However, the write request will only be forwarded to the first (primary) Pod, and the data will be synced with the other Pods. You can achieve this by using StatefulSets.
- Deleting or scaling down a StatefulSet will not delete the volumes associated with the stateful application. This gives you your data safety. If you delete the MySQL Pod or if the MySQL Pod restarts, you can have access to the data in the same volume.
  You can also create Pods (containers) using the Deployment object in the Kubernetes cluster. This allows you to easily replicate Pods and attach a storage volume to the Pods. The same thing can be done by using StatefulSets. What then is the advantage of using StatefulSets?
  Well, the Pods created using the Deployment object are assigned random IDs. For example, you are creating a Pod named “my-app”, and you are scaling it to three replicas. The names of the Pods are created like this:

my-app-123ab
my-app-098bd
my-app-890yt

After the name “my-app”, random IDs are added. If the Pod restarts or you scale it down, then again, the Kubernetes Deployment object will assign different random IDs for each Pod. After restarting, the names of all Pods appear like this:

my-app-jk879
my-app-kl097
my-app-76hf7

All these Pods are associated with one load balancer service. So in a stateless application, changes in the Pod name are easily identified, and the service object easily handles the random IDs of Pods and distributes the load. This type of deployment is very suitable for stateless applications.
However, stateful applications cannot be deployed like this. The stateful application needs a sticky identity for each Pod because replica Pods are not identical Pods.
Imagine a MySQL database deployment. Assume you are creating Pods for the MySQL database using the Kubernetes Deployment object and scaling the Pods. If you are writing data on one MySQL Pod, do not replicate the same data on another MySQL Pod if the Pod is restarted. This is the first problem with the Kubernetes Deployment object for the stateful application.
The StatefulSet adds the ability to start and stop Pods in a specific sequence.
With a Deployment, for example, all your Pods are started and stopped in a random order. This is fine for stateless services, where every replica is identical and does the same job
Each replica in a StatefulSet must be running and ready before Kubernetes starts the next one, and similarly when the StatefulSet is terminated, the replicas will be shut down in reverse order, waiting for each Pod to finish before moving on to the next.
To be able to address each of the Pods by a predictable DNS name, such as redis-1, you also

need to create a Service with a clusterIP type of None (known as a headless service).
With a non-headless Service, you get a single DNS entry (such as redis) that loadbalances across all the backend Pods. With a headless service, you still get that single Pod Controllers service DNS name, but you also get individual DNS entries for each numbered Pod, like redis-0, redis-1, redis-2, and so on.

For stateful applications with a StatefulSet controller, it is possible to set the first Pod as primary and other Pods as replicas, the first Pod will handle both read and write requests from the user, and other Pods always sync with the first Pod for data replication. If the Pod dies, a new Pod is created with the same name.
In summary, StatefulSets provide the following advantages when compared to Deployment objects:

- Ordered numbers for each Pod
- The first Pod can be a primary, which makes it a good choice when creating a replicated database setup, which handles both reading and writing
- Other Pods act as replicas
- New Pods will only be created if the previous Pod is in running state and will clone the previous Pod's data
- Deletion of Pods occurs in reverse order

Jobs and CronJobs

Another useful type of Pod controller in Kubernetes is the Job. Whereas a Deployment runs a specified number of Pods and restarts them continually, a Job only runs a Pod for a specified number of times. After that, it is considered completed.
There are two fields that control Job execution: completions and parallelism. The first, completions, determines the number of times the specified Pod needs to run successfully before the Job is considered complete. The default value is 1, meaning the Pod will run once. The parallelism field specifies how many Pods should run at once. Again, the default value is 1, meaning that only one Pod will run at a time.
If it crashes, fails, or exits in any nonsuccessful way, the Job will restart it, just like a Deployment does. Only successful exits count toward the required number of completions.
In Unix environments, scheduled jobs are run by the cron daemon.
The two important fields to look at in the CronJob manifest are spec.schedule and spec.jobTemplate. The schedule field specifies when the job will run, using the same format as the Unix cron utility.

Managing resources

Namespaces

For example, you might have different namespaces for testing out different versions of an application, or a separate namespace per team.
As the term namespace suggests, names in one namespace are not visible from a different namespace.

This means that you could have a service called demo in the prod namespace, and a different service called demo in the test namespace, and there won't be any conflict.
If you don't specify a namespace when running a kubectl command, such as kubectl run, your command will operate on the default namespace.
If you're wondering what the kube-system namespace is, that's where the Kubernetes internal system components run so that they're segregated from your own applications.
If, instead, you specify a namespace with the --namespace flag (or -n for short), your command will use that namespace.
For example, to get a list of Pods in the prod namespace, run: kubectl get pods --namespace prod
namespaces are logically isolated from one another, they can still communicate with Services in other namespaces.

Assigning pod to nodes

You can constrain a Pod so that it is restricted to run on particular node(s), or to prefer to run on particular nodes. There are several ways to do this and the recommended approaches all use label selectors to facilitate the selection. Often, you do not need to set any such constraints; the scheduler will automatically do a reasonable placement (for example, spreading your Pods across nodes so as not place Pods on a node with insufficient free resources). However, there are some circumstances where you may want to control which node the Pod deploys to, for example, to ensure that a Pod ends up on a node with an SSD attached to it, or to co-locate Pods from two different services that communicate a lot into the same availability zone.
You can use any of the following methods to choose where Kubernetes schedules specific Pods:

- nodeSelector field matching against node labels
- Affinity and anti-affinity
- nodeName field
- Pod topology spread constraints

nodeSelector is the simplest recommended form of node selection constraint. You can add the nodeSelector field to your Pod specification and specify the node labels you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify.
Affinity and anti-affinity expands the types of constraints you can define. Some of the benefits of affinity and anti-affinity include:

- The affinity/anti-affinity language is more expressive. nodeSelector only selects nodes with all the specified labels. Affinity/anti-affinity gives you more control over the selection logic.
- You can indicate that a rule is soft or preferred, so that the scheduler still schedules the Pod even if it can't find a matching node.
- You can constrain a Pod using labels on other Pods running on the node (or other topological domain), instead of just node labels, which allows you to define rules for which Pods can be co-located on a node.
  The affinity feature consists of two types of affinity:
- Node affinity functions like the nodeSelector field but is more expressive and allows you to specify soft rules.

- Inter-pod affinity/anti-affinity allows you to constrain Pods against labels on other Pods.
  Node affinity is conceptually similar to nodeSelector, allowing you to constrain which nodes your Pod can be scheduled on based on node labels.
  There are two types of node affinity:
- requiredDuringSchedulingIgnoredDuringExecution: The scheduler can't schedule the Pod unless the rule is met. This functions like nodeSelector, but with a more expressive syntax.
- preferredDuringSchedulingIgnoredDuringExecution: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod.
  You can specify a weight between 1 and 100 for each instance of the preferredDuringSchedulingIgnoredDuringExecution affinity type. When the scheduler finds nodes that meet all the other scheduling requirements of the Pod, the scheduler iterates through every preferred rule that the node satisfies and adds the value of the weight for that expression to a sum.
  The final sum is added to the score of other priority functions for the node. Nodes with the highest total score are prioritized when the scheduler makes a scheduling decision for the Pod.
  Inter-pod affinity and anti-affinity allow you to constrain which nodes your Pods can be scheduled on based on the labels of Pods already running on that node, instead of the node labels.
  Inter-pod affinity and anti-affinity rules take the form "this Pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more Pods that meet rule Y", where X is a topology domain like node, rack, cloud provider zone or region, or similar and Y is the rule Kubernetes tries to satisfy.
  You express these rules (Y) as label selectors with an optional associated list of namespaces. Pods are namespaced objects in Kubernetes, so Pod labels also implicitly have namespaces. Any label selectors for Pod labels should specify the namespaces in which Kubernetes should look for those labels.
  You express the topology domain (X) using a topologyKey, which is the key for the node label that the system uses to denote the domain. For examples, see Well-Known Labels, Annotations and Taints.
  nodeName is a more direct form of node selection than affinity or nodeSelector.
  nodeName is a field in the Pod spec. If the nodeName field is not empty, the scheduler ignores the Pod and the kubelet on the named node tries to place the Pod on that node.
  Using nodeName overrules using nodeSelector or affinity and anti-affinity rules.
  Some of the limitations of using nodeName to select nodes are:
- If the named node does not exist, the Pod will not run, and in some cases may be automatically deleted.
- If the named node does not have the resources to accommodate the Pod, the Pod will fail and its reason will indicate why, for example OutOfmemory or OutOfcpu.
- Node names in cloud environments are not always predictable or stable.
  You can use **topology spread constraints** to control how Pods are spread across your cluster among failure-domains such as regions, zones, nodes, or among any other topology domains that you define. You might do this to improve performance, expected availability, or overall utilization.

Networking

- Intra-pod: All containers within a pod share a network namespace and see each other on localhost.
- Inter-pod networking: Two types of east–west traffic are supported: pods can directly communicate with other pods or, preferably, pods can leverage services to communicate with other pods.
- Ingress and egress: Ingress refers to routing traffic from external users or apps to pods, and egress refers to calling external APIs from pods.

Kubernetes requires each pod to have an IP in a flat networking name, space with full connectivity to other nodes and pods across the network. This IP-per-pod model yields a backward-compatible way for you to treat a pod almost identically to a VM or a physical host, in the context of naming, service discovery, or port allocations. The model allows for a smoother transition from non–cloud native apps and environments.
In Kubernetes, each pod has a routable IP, allowing pods to communicate across cluster nodes without NAT and no need to manage port allocations.
Because every pod gets a real (that is, not machine-local) IP address, pods can communicate without proxies or translations (such as NAT). The pod can use well-known ports and can avoid the use of higher-level service discovery mechanisms.
We distinguish between two types of inter-pod communication, sometimes also called East-West traffic:

- Pods can directly communicate with other pods; in this case the caller pod needs to find out the IP address of the callee and risks repeating this operation since pods come and go (cattle behaviour).
- Preferably, pods use services to communicate with other pods. In this case, the service provides a stable (virtual) IP address that can be discovered, for example, via DNS.

While in the case of Ingress we're interested in routing traffic from outside the cluster to a service, in the case of Egress we are dealing with the opposite: how does an app in a pod call out to (cluster-)external APIs? One may want to control which pods are allowed to have a communication path to outside services and on top of that impose other policies. Note that by default all containers in a pod can perform Egress. These policies can be enforced using network policies.

Services

A service provides a stable virtual IP (VIP) address for a set of pods.
While pods may come and go, services allow clients to reliably discover and connect to the containers running in the pods by using the VIP.
The "virtual" in VIP means it's not an actual IP address connected to a network interface; its purpose is purely to act as the stable front to forward traffic to one or more pods, with IP addresses that may come and go.
You specify the set of pods you want a service to target via a label selector, for example, for spec.selector.app=someapp
Kubernetes would create a service that targets all pods with a label app=someapp.
Note that if such a selector exists, then for each of the targeted pods a sub-resource of type

Endpoint will be created, and if no selector exists then no endpoints are created.
Keeping the mapping between the VIP and the pods up-to-date is the job of kube-proxy.
You could find out the Pod's IP address and connect directly to that address and the app's port number.
But the IP address may change when the Pod is restarted, so you'll have to keep looking it up to make sure it's up-to-date.
Worse, there may be multiple replicas of the Pod, each with different addresses.
Every other application that needs to contact the Pod would have to maintain a list of those addresses, which doesn't sound like a great idea.
A selector is an expression that matches a label (or set of labels). It's a way of specifying a group of resources by their labels.
Labels are key/value pairs that are attached to objects, such as pods.
Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system.
If you set the type field to NodePort, the Kubernetes control plane allocates a port from a range specified by --service-node-port-range flag (default: 30000-32767). Each node proxies that port (the same port number on every Node) into your Service.
ClusterIP: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.
ClusterIP addresses are drawn from a private, non-routable IP address space (subnet), often called the overlay network. Non-routable means that the ClusterIP address cannot be reached from outside the cluster. a problem arises when you want clients or users external to the cluster to have access to one or more pods or services
NodePort: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`.
LoadBalancer: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created. Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.
Some cloud providers allow you to specify the loadBalancerIP. In those cases, the load-balancer is created with the user-specified loadBalancerIP. If the loadBalancerIP field is not specified, the loadBalancer is set up with an ephemeral IP address. If you specify a loadBalancerIP but your cloud provider does not support the feature, the loadbalancerIP field that you set is ignored.
ExternalName: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

In iptables mode, kube-proxy creates iptables rules for kubernetes services which ensure that the request to the service gets routed (and load balanced) to the appropriate pods.
These iptables rules also help answer the second question mentioned above. As long as these iptables rules exist, requests to services will get routed to the appropriate pods even if kube-proxy process dies on the node. Endpoints for new services won't work from this node, however, since kube-proxy process won't create the iptables rules for it.
As outlined in this flow chart, there are rules in the PREROUTING chain of the nat table for kubernetes services. As per iptables rules evaluation order, rules in the PREROUTING chain are the first ones to be consulted as a packet enters the linux kernel's networking stack.

Rules created by kube-proxy in the PREROUTING chain help determine if the packet is meant for a local socket on the node or if it should be forwarded to a pod. It is these rules that ensure that the request to `<kubernetes-node-ip>:<NodePort>` continue to get routed to pods even if the NodePort is in use by another process.
As a part of the v1.25 release, SIG Network made this declaration explicit: that (with one exception), the iptables chains that Kubernetes creates are intended only for Kubernetes's own internal use, and third-party components should not assume that Kubernetes will create any specific iptables chains, or that those chains will contain any specific rules if they do exist.

Ingress

An API object that manages external access to the services in a cluster, typically HTTP.
Ingress may provide load balancing, SSL termination and name-based virtual hosting

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

conceptually, it is split up into two main pieces, an Ingress resource, which defines the routing to the backing services, and the Ingress controller, which listens to the /ingresses endpoint of the API server, learning about services being created or removed. On service status changes, the Ingress controller configures the routes so that external traffic lands at a specific (cluster-internal) service.

CNI

Cilium

Cilium is another CNI solution, based on **eBPF**, and is designed to be run at large scale. Whilst Cilium implements the standard NetworkPolicies, it is able to utilise the full packet introspection of eBPF, enabling it to have first class support for Layer 7 policy for a number of protocols, custom extensions using Envoy, large options for endpoint selection, as well as rich network monitoring introspection. For these reasons, Cilium becomes a very favourable choice for running Kubernetes at scale, with *complex network policy requirements*.
Kubernetes doesn't come with an implementation of Load Balancing. This is usually left as an exercise for your cloud provider or in private cloud environments an exercise for your networking team. Cilium can attract this traffic with BGP and accelerate leveraging XDP and eBPF. Together these technologies provide a very robust and secure implementation of Load Balancing.
Cilium and eBPF operate at the kernel layer. With this level of context we can make intelligent decisions about how to connect different workloads whether on the same node or between clusters. With eBPF and XDP Cilium enables significant improvements in latency and performance and eliminates the need for kube-proxy entirely.
Cilium's control and data plane has been built from the ground up for large-scale and highly dynamic cloud native environments where 100s and even 1000s of containers are created and destroyed within seconds. Cilium's control plane is highly optimized, running in Kubernetes clusters of up to 5K nodes and 100K pods. Cilium's data plane uses eBPF for efficient load-balancing and incremental updates, avoiding the pitfalls of large iptables rulesets. Cilium is fully IPv6-aware.

With standard Kubernetes networking each cluster is an island, requiring proxies to connect workloads across clusters for the purposes of migration, disaster-recovery, or geographic locality. Cilium Cluster Mesh creates a single zone of connectivity for load-balancing, observability and security between nodes across multiple clusters, enabling simple, high-performance cross-cluster connectivity.

eBPF is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules.
Historically, the operating system has always been an ideal place to implement observability, security, and networking functionality due to the kernel's privileged ability to oversee and control the entire system. At the same time, an operating system kernel is hard to evolve due to its central role and high requirement towards stability and security. The rate of innovation at the operating system level has thus traditionally been lower compared to functionality implemented outside of the operating system.
eBPF changes this formula fundamentally. By allowing to run sandboxed programs within the operating system, application developers can run eBPF programs to add additional capabilities to the operating system at runtime. The operating system then guarantees safety and execution efficiency as if natively compiled with the aid of a Just-In-Time (JIT) compiler and verification engine. This has led to a wave of eBPF-based projects covering a wide array of use cases, including next-generation networking, observability, and security functionality.
Today, eBPF is used extensively to drive a wide variety of use cases: Providing high-performance networking and load-balancing in modern data centers and cloud native environments, extracting fine-grained security observability data at low overhead, helping application developers trace applications, providing insights for performance troubleshooting, preventive application and container runtime security enforcement, and much more. The possibilities are endless, and the innovation that eBPF is unlocked has only just begun.
The combination of programmability and efficiency makes eBPF a natural fit for all packet processing requirements of networking solutions. The programmability of eBPF enables adding additional protocol parsers and easily program any forwarding logic to meet changing requirements without ever leaving the packet processing context of the Linux kernel. The efficiency provided by the JIT compiler provides execution performance close to that of natively compiled in-kernel code.

Flannel

Flannel manages an IPv4 network between multiple nodes in a cluster. It does not control how containers are networked to the host, only how the traffic is transported between hosts.
Flannel runs a small, single binary agent called flanneld on each host, and is responsible for allocating a subnet lease to each host out of a larger, preconfigured address space. Flannel uses either the Kubernetes API or etcd directly to store the network configuration, the allocated subnets, and any auxiliary data (such as the host's public IP). Packets are forwarded using one of several backend mechanisms including VXLAN and various cloud integrations.
Platforms like Kubernetes assume that each container (pod) has a unique, routable IP inside the cluster. The advantage of this model is that it removes the port mapping complexities that come from sharing a single host IP.
Flannel is responsible for providing a layer 3 IPv4 network between multiple nodes in a cluster.

Flannel does not control how containers are networked to the host, only how the traffic is transported between hosts. However, flannel does provide a CNI plugin for Kubernetes and a guidance on integrating with Docker.
Flannel is focused on networking. For network policy, other projects such as Calico can be used.
Flannel may be paired with several different backends. Once set, the backend should not be changed at runtime.
VXLAN is the recommended choice. host-gw is recommended for more experienced users who want the performance improvement and whose infrastructure support it (typically it can't be used in cloud environments). UDP is suggested for debugging only or for very old kernels that don't support VXLAN.

Calico

Calico is a widely adopted, battle-tested open source networking and network security solution for Kubernetes, virtual machines, and bare-metal workloads. Calico provides two major services for Cloud Native applications:

- Network connectivity between workloads.
- Network security policy enforcement between workloads.

Calico's flexible architecture supports a wide range of deployment options, using modular components and technologies, including:

- Choice of data plane technology, whether it be eBPF, standard Linux, Windows HNS or VPP
- Enforcement of the full set of Kubernetes network policy features, plus for those needing a richer set of policy features, Calico network policies.
- An optimized Kubernetes Service implementation using eBPF.
- Kubernetes apiserver integration, for managing Calico configuration and Calico network policies.
- Both non-overlay and overlay (via IPIP or VXLAN) networking options in either public cloud or on-prem deployments.
- CNI plugins for Kubernetes to provide highly efficient pod networking and IP Address Management (IPAM).
- A Neutron ML2 plugin to provide VM networking for OpenStack.
- A BGP routing stack that can advertise routes for workload and service IP addresses to physical network infrastructure.

We use Project Calico as the CNI overlay for the Kubernetes, because it supports BGP. An advantage of Calico is that it works as a routing method overlay network, rather than an encapsulation method overlay. This is important to the Networking team, because Ethernet packets remain in their native format, making them easy to see, capture, and troubleshoot with common tools like ping, curl, dig, traceroute, tcpdump, and ssldump. Encapsulation overlays that modify packets can make it more difficult to use these tried-and-tested tools – you have to unwrap the packets to see what's really going on. Calico also enables you to control the IP address pools allocated for the pods, which helps you quickly identify any networking issues – simply knowing the pod's IP address tells you immediately on which Kubernetes node the pod is running.

Microservices Applications

What is a microservice?

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.

- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.
  Besides for the services themselves, some other components appear in a typical microservices architecture:
  Management/orchestration. This component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.
  API Gateway. The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.
  Advantages of using an API gateway include:
- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.
- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.
- Out-of-the-box policies, like for throttling, caching, transformation, or validation.

Benefits

- Agility. Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application,

and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features may be held up waiting for a bug fix to be integrated, tested, and published.

- Small, focused teams. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- Small code base. In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.
- Mix of technologies. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- Fault isolation. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).
- Scalability. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- Data isolation. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- Complexity. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- Development and testing. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- Lack of governance. The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.
- Network congestion and latency. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats,

and look for places to use asynchronous communication patterns like queue-based load leveling.

- Data integrity. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- Management. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
- Versioning. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
- Skill set. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.
  Best practices
- Model services around the business domain.
- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.
- Data storage should be private to the service that owns the data. Use the best storage for each service and data type.
- Services communicate through well-designed APIs. Avoid leaking implementation details. APIs should model the domain, not the internal implementation of the service.
- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.
- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway.
- Keep domain knowledge out of the gateway. The gateway should handle and route client requests without any knowledge of the business rules or domain logic. Otherwise, the gateway becomes a dependency and can cause coupling between services.
- Services should have loose coupling and high functional cohesion. Functions that are likely to change together should be packaged and deployed together. If they reside in separate services, those services end up being tightly coupled, because a change in one service will require updating the other service. Overly chatty communication between two services may be a symptom of tight coupling and low cohesion.
- Isolate failures. Use resiliency strategies to prevent failures within a service from cascading.

Service mesh

A service mesh, like the open source project Istio, is a way to control how different parts of an application share data with one another. Unlike other systems for managing this communication, a service mesh is a dedicated infrastructure layer built right into an app. This visible infrastructure layer can document how well (or not) different parts of an app interact, so it becomes easier to optimize communication and avoid downtime as an app grows.
Each part of an app, called a "service," relies on other services to give users what they want. If a user of an online retail app wants to buy something, they need to know if the item is in stock. So, the service that communicates with the company's inventory database needs to communicate with the product webpage, which itself needs to communicate with the user’s online shopping cart. To add business value, this retailer might eventually build a service that

gives users in-app product recommendations. This new service will communicate with a database of product tags to make recommendations, but it also needs to communicate with the same inventory database that the product page needed—it's a lot of reusable, moving parts.
Modern applications are often broken down in this way, as a network of services each performing a specific business function. In order to execute its function, one service might need to request data from several other services. But what if some services get overloaded with requests, like the retailer's inventory database? This is where a service mesh comes in—it routes requests from one service to the next, optimizing how all the moving parts work together.
A microservices architecture lets developers make changes to an app's services without the need for a full redeploy. Unlike app development in other architectures, individual microservices are built by small teams with the flexibility to choose their own tools and coding languages. Basically, microservices are built independently, communicate with each other, and can individually fail without escalating into an application-wide outage.
Service-to-service communication is what makes microservices possible. The logic governing communication can be coded into each service without a service mesh layer—but as communication gets more complex, a service mesh becomes more valuable. For cloud-native apps built in a microservices architecture, a service mesh is a way to comprise a large number of discrete services into a functional application.
Deploying application's components in separate processes or containers to entrust isolation and encapsulation.
Sidecar can be attached to the parent application and provide support features to the application.
Dataplane is implemented as an array of proxies (e.g. Envoy) deployed as sidecars.
Each pod contains a proxy istance to manage the communication between microservices.
The control plane manage and configure proxies to route the traffic.
Without a service mesh, each microservice needs to be coded with logic to govern service-to-service communication, which means developers are less focused on business goals. It also means communication failures are harder to diagnose because the logic that governs interservice communication is hidden within each service.
Every new service added to an app, or new instance of an existing service running in a container, complicates the communication environment and introduces new points of possible failure. Within a complex microservices architecture, it can become nearly impossible to locate where problems have occurred without a service mesh.
That's because a service mesh also captures every aspect of service-to-service communication as performance metrics. Over time, data made visible by the service mesh can be applied to the rules for interservice communication, resulting in more efficient and reliable service requests. For example, If a given service fails, a service mesh can collect data on how long it took before a retry succeeded. As data on failure times for a given service aggregates, rules can be written to determine the optimal wait time before retrying that service, ensuring that the system does not become overburdened by unnecessary retries.
Virtual services, along with destination rules, are the key building blocks of **Istio's** traffic routing functionality. A virtual service lets you configure how requests are routed to a service within an Istio service mesh, building on the basic connectivity and discovery provided by Istio and your platform. Each virtual service consists of a set of routing rules that are evaluated in order, letting Istio match each given request to the virtual service to a specific real destination within the mesh. Your mesh can require multiple virtual services or none depending on your use case.

Istio

Virtual services play a key role in making Istio's traffic management flexible and powerful. They do this by strongly decoupling where clients send their requests from the destination workloads that actually implement them. Virtual services also provide a rich way of specifying different traffic routing rules for sending traffic to those workloads.
Why is this so useful? Without virtual services, Envoy distributes traffic using round-robin load balancing between all service instances, as described in the introduction. You can improve this behavior with what you know about the workloads. For example, some might represent a different version. This can be useful in A/B testing, where you might want to configure traffic routes based on percentages across different service versions, or to direct traffic from your internal users to a particular set of instances.
With a virtual service, you can specify traffic behavior for one or more hostnames. You use routing rules in the virtual service that tell Envoy how to send the virtual service's traffic to appropriate destinations. Route destinations can be versions of the same service or entirely different services.
A typical use case is to send traffic to different versions of a service, specified as service subsets. Clients send requests to the virtual service host as if it was a single entity, and Envoy then routes the traffic to the different versions depending on the virtual service rules: for example, "20% of calls go to the new version" or "calls from these users go to version 2". This allows you to, for instance, create a canary rollout where you gradually increase the percentage of traffic that's sent to a new service version. The traffic routing is completely separate from the instance deployment, meaning that the number of instances implementing the new service version can scale up and down based on traffic load without referring to traffic routing at all. By contrast, container orchestration platforms like Kubernetes only support traffic distribution based on instance scaling, which quickly becomes complex. You can read more about how virtual services help with canary deployments in Canary Deployments using Istio.
Virtual services also let you:

- Address multiple application services through a single virtual service. If your mesh uses Kubernetes, for example, you can configure a virtual service to handle all services in a specific namespace. Mapping a single virtual service to multiple "real" services is particularly useful in facilitating turning a monolithic application into a composite service built out of distinct microservices without requiring the consumers of the service to adapt to the transition. Your routing rules can specify "calls to these URIs of monolith.com go to microservice A", and so on. You can see how this works in one of our examples below.
- Configure traffic rules in combination with gateways to control ingress and egress traffic. In some cases you also need to configure destination rules to use these features, as these are where you specify your service subsets. Specifying service subsets and other destination-specific policies in a separate object lets you reuse these cleanly between virtual services. You can find out more about destination rules in the next section.

In addition to capturing application traffic, Istio can also capture DNS requests to improve the performance and usability of your mesh. When proxying DNS, all DNS requests from an application will be redirected to the sidecar, which stores a local mapping of domain names to IP addresses. If the request can be handled by the sidecar, it will directly return a response to the application, avoiding a roundtrip to the upstream DNS server. Otherwise, the request is forwarded upstream following the standard /etc/resolv.conf DNS configuration.

While Kubernetes provides DNS resolution for Kubernetes Services out of the box, any custom ServiceEntrys will not be recognized. With this feature, ServiceEntry addresses can be resolved without requiring custom configuration of a DNS server. For Kubernetes Services, the DNS response will be the same, but with reduced load on kube-dns and increased performance.
This functionality is also available for services running outside of Kubernetes. This means that all internal services can be resolved without clunky workarounds to expose Kubernetes DNS entries outside of the cluster.
In the above example, you had a predefined IP address for the service to which you sent the request. However, it's common to access external services that do not have stable addresses, and instead rely on DNS. In this case, the DNS proxy will not have enough information to return a response, and will need to forward DNS requests upstream.
This is especially problematic with TCP traffic. Unlike HTTP requests, which are routed based on Host headers, TCP carries much less information; you can only route on the destination IP and port number. Because you don't have a stable IP for the backend, you cannot route based on that either, leaving only port number, which leads to conflicts when multiple ServiceEntrys for TCP services share the same port.
To work around these issues, the DNS proxy additionally supports automatically allocating addresses for ServiceEntrys that do not explicitly define one. This is configured by the ISTIO_META_DNS_AUTO_ALLOCATE option.
When this feature is enabled, the DNS response will include a distinct and automatically assigned address for each ServiceEntry. The proxy is then configured to match requests to this IP address, and forward the request to the corresponding ServiceEntry.

Envoy

- Out of process architecture: Envoy is a self contained process that is designed to run alongside every application server. All of the Envoys form a transparent communication mesh in which each application sends and receives messages to and from localhost and is unaware of the network topology. The out of process architecture has two substantial benefits over the traditional library approach to service to service communication:
    - Envoy works with any application language. A single Envoy deployment can form a mesh between Java, C++, Go, PHP, Python, etc. It is becoming increasingly common for service oriented architectures to use multiple application frameworks and languages. Envoy transparently bridges the gap.
    - As anyone that has worked with a large service oriented architecture knows, deploying library upgrades can be incredibly painful. Envoy can be deployed and upgraded quickly across an entire infrastructure transparently.
- Modern C++11 code base: Envoy is written in C++11. Native code was chosen because we believe that an architectural component such as Envoy should get out of the way as much as possible. Modern application developers already deal with tail latencies that are difficult to reason about due to deployments in shared cloud environments and the use of very productive but not particularly well performing languages such as PHP, Python, Ruby, Scala, etc. Native code provides generally excellent latency properties that don't add additional confusion to an already confusing situation. Unlike other native code proxy solutions written in C, C++11 provides both excellent developer productivity and performance.

- L3/L4 filter architecture: At its core, Envoy is an L3/L4 network proxy. A pluggable filter chain mechanism allows filters to be written to perform different TCP proxy tasks and inserted into the main server. Filters have already been written to support various tasks such as raw TCP proxy, HTTP proxy, TLS client certificate authentication, etc.
- HTTP L7 filter architecture: HTTP is such a critical component of modern application architectures that Envoy supports an additional HTTP L7 filter layer. HTTP filters can be plugged into the HTTP connection management subsystem that perform different tasks such as buffering, rate limiting, routing/forwarding, sniffing Amazon's DynamoDB, etc.
- First class HTTP/2 support: When operating in HTTP mode, Envoy supports both HTTP/1.1 and HTTP/2. Envoy can operate as a transparent HTTP/1.1 to HTTP/2 proxy in both directions. This means that any combination of HTTP/1.1 and HTTP/2 clients and target servers can be bridged. The recommended service to service configuration uses HTTP/2 between all Envoys to create a mesh of persistent connections that requests and responses can be multiplexed over. Envoy does not support SPDY as the protocol is being phased out.
- HTTP L7 routing: When operating in HTTP mode, Envoy supports a routing subsystem that is capable of routing and redirecting requests based on path, authority, content type, runtime values, etc. This functionality is most useful when using Envoy as a front/edge proxy but is also leveraged when building a service to service mesh.
- gRPC support: gRPC is an RPC framework from Google that uses HTTP/2 as the underlying multiplexed transport. Envoy supports all of the HTTP/2 features required to be used as the routing and load balancing substrate for gRPC requests and responses. The two systems are very complementary.
- MongoDB L7 support: MongoDB is a popular database used in modern web applications. Envoy supports L7 sniffing, statistics production, and logging for MongoDB connections.
- DynamoDB L7 support: DynamoDB is Amazon's hosted key/value NOSQL datastore. Envoy supports L7 sniffing and statistics production for DynamoDB connections.
- Service discovery: Service discovery is a critical component of service oriented architectures. Envoy supports multiple service discovery methods including asynchronous DNS resolution and REST based lookup via a service discovery service.
- Health checking: The recommended way of building an Envoy mesh is to treat service discovery as an eventually consistent process. Envoy includes a health checking subsystem which can optionally perform active health checking of upstream service clusters. Envoy then uses the union of service discovery and health checking information to determine healthy load balancing targets. Envoy also supports passive health checking via an outlier detection subsystem.
- Advanced load balancing: Load balancing among different components in a distributed system is a complex problem. Because Envoy is a self contained proxy instead of a library, it is able to implement advanced load balancing techniques in a single place and have them be accessible to any application. Currently Envoy includes support for automatic retries, circuit breaking, global rate limiting via an external rate limiting service, request shadowing, and outlier detection. Future support is planned for request racing.
- Front/edge proxy support: Although Envoy is primarily designed as a service to service communication system, there is benefit in using the same software at the edge (observability, management, identical service discovery and load balancing algorithms, etc.). Envoy includes enough features to make it usable as an edge proxy for most modern

web application use cases. This includes TLS termination, HTTP/1.1 and HTTP/2 support, as well as HTTP L7 routing.

- Best in class observability: As stated above, the primary goal of Envoy is to make the network transparent. However, problems occur both at the network level and at the application level. Envoy includes robust statistics support for all subsystems. statsd (and compatible providers) is the currently supported statistics sink, though plugging in a different one would not be difficult. Statistics are also viewable via the administration port. Envoy also supports distributed tracing via thirdparty providers.
- Dynamic configuration: Envoy optionally consumes a layered set of dynamic configuration APIs. Implementors can use these APIs to build complex centrally managed deployments if desired.

Types of load balancing

No Load Balancing: the user connects directly to your web server, at yourdomain.com and there is no load balancing. If your single web server goes down, the user will no longer be able to access your web server. Additionally, if many users are trying to access your server simultaneously and it is unable to handle the load, they may have a slow experience or they may not be able to connect at all.
The simplest way to load balance network traffic to multiple servers is to use **layer 4 (transport layer) load balancing**. Load balancing this way will forward user traffic based on IP range and port (i.e. if a request comes in for http://yourdomain.com/anything, the traffic will be forwarded to the backend that handles all the requests for yourdomain.com on port 80). For more details on layer 4, check out the TCP subsection of our Introduction to Networking.
The user accesses the load balancer, which forwards the user's request to the web-backend group of backend servers. Whichever backend server is selected will respond directly to the user's request. Generally, all of the servers in the web-backend should be serving identical content–otherwise the user might receive inconsistent content. Note that both web servers connect to the same database server.
Another, more complex way to load balance network traffic is to use **layer 7 (application layer) load balancing**. Using layer 7 allows the load balancer to forward requests to different backend servers based on the content of the user's request. This mode of load balancing allows you to run multiple web application servers under the same domain and port. For more details on layer 7, check out the HTTP subsection of our Introduction to Networking.
In this example, if a user requests yourdomain.com/blog, they are forwarded to the blog backend, which is a set of servers that run a blog application. Other requests are forwarded to web-backend, which might be running another application. Both backends use the same database server, in this example.

- Round Robin: selects servers in turns. This is the default algorithm.
- leastconn: Selects the server with the least number of connections. This is recommended for longer sessions. Servers in the same backend are also rotated in a round-robin fashion.
- source: This selects which server to use based on a hash of the source IP address that users are making requests from. This method ensures that the same users will connect to the same servers.

Pod migration

Placing certain services near the consumers has great benefits, including low-latency response, bandwidth consumption savings, and data locality. However, there are also multiple challenges. One of the key challenges with the Kubernetes deployment model is the placement of the Kubernetes control plane that manages the workers that comprise the resource pools consumed by the applications and services. The two main options for control plane placement are:

- Deploying full-fledged cluster(s), complete with control nodes and worker nodes, everywhere you need your applications to be accessible
- Deploying worker nodes at the edge and connecting them to the central location hosting the control plane

You can simplify option one (the full-cluster model shown above) with innovative deployment models:

- A compact high availability (HA) cluster with a minimum of three nodes accommodating both control plane and worker node roles
- An all-in-one, single-node standalone cluster

Option two (the remote worker approach shown below) eliminates the overhead of having a dedicated control plane at each location. Still, it may not be feasible if there is a significant latency, intermittent connectivity, or a lack of sufficient bandwidth for the cluster's internal services or operations between the Kubernetes control plane and the worker locations to function correctly.

CRIU

CRIU (stands for Checkpoint and Restore in Userspace) is a utility to checkpoint/restore Linux tasks.
Using this tool, you can freeze a running application (or part of it) and checkpoint it to a hard drive as a collection of files. You can then use the files to restore and run the application from the point it was frozen at. The distinctive feature of the CRIU project is that it is mainly implemented in user space. There are some more projects doing C/R for Linux, and so far CRIU appears to be the most feature-rich and up-to-date with the kernel.
CRIU project is (almost) the never-ending story, because we have to always keep up with the Linux kernel supporting checkpoint and restore for all the features it provides.
Live migration (Live migration attempts to provide a seamless transfer of service between physical machines without impacting client processes or applications)
True live migration using CRIU is possible, but doing all the steps by hands might be complicated. The phaul sub-project provides a Go library that encapsulates most of the complexity. This library and the Go bindings for CRIU are stored in the go-criu repository.
In order to get state of the running process CRIU needs to make this process execute some code, that would fetch the required information. To make this happen without killing the application itself,CRIU uses the parasite code injection technique, which is also available as a standalone library called libcompel.

One of the CRIU features is the ability to save and restore state of a TCP socket without breaking the connection. This functionality is considered to be useful by itself, and we have it available as the libsoccr library.

DMTCP

DMTCP implements checkpoint/restore of a process on a library level. This means, that if you want to C/R some application you should launch one with DMTCP library (dynamically) linked from the very beginning. When launched like this, the DMTCP library intercepts a certain amount of library calls from the application, builds a shadow data-base of information about process' internals and then forwards the request down to glibc/kernel. The information gathered is to be used to create an image of the application. With this approach, one can only dump applications known to run successfully with the DMTCP libraries, but the latter doesn't provide proxies for all kernel APIs (for example, inotify() is known to be unsupported). Another implication of this approach is potential performance issues that arise due to proxying of requests.
Restoration of process set is also tricky, as it frequently requires restoring an object with the predefined ID and kernel is known to provide no APIs for several of them. For example, kernel cannot fork a process with the desired PID. To address that, DMTCP fools a process by intercepting the getpid() library call and providing fake PID value to the application. Such behavior is very dangerous, as application might see wrong files in the /proc filesystem if it will try to access one via its PID.
CRIU, on the other hand, doesn't require any libraries to be pre-loaded. It will checkpoint and restore any arbitrary application, as long as kernel provides all needed facilities. Kernel support for some of CRIU features were added recently, essentially meaning that a recent kernel version might be required.

CRIU can dump a task without preparations. DMTCP can dump only prepared tasks. """ A DMTCP coordinator process is created on a host (default: localhost). As new processes are created (via fork or ssh), the LD_PRELOAD environment variable (supported by the Linux loader) is used to preload the DMTCP library (dmtcphijack.so). That library runs before the routine main(). It creates a second thread (DMTCP checkpoint thread). The checkpoint thread then creates a socket to the DMTCP coordinator and registers itself. The checkpoint thread also creates a signal handler (SIGUSR2 by default) """ CRIU doesn't affect behavior of applications before and after checkpoint/restore. CRIU is independent from GLIBC and other libraries. DMTCP sets wrappers on a few system calls, so it can change behavior of applications. Probably DMCTP can't dump static linked programs and programs, which call syscall directly. """ The run-time overhead of DMTCP is essentially zero. When there is no checkpoint or restart in process, DMTCP code will run only within DMTCP wrappers around certain less frequently used system calls. Examples of such wrappers are wrappers for open(), getpid(), socketpair(), etc. """ DMTCP doesn't support namespaces, so it can not dump Linux Containers. DMTCP virtualizes PID-s in user-space, actually a task is restored with another pid. It may be prefered in some cases. I'm not sure that DMCTP can restore anonymous shared memory correctly. Probably DMTCP can't restore TCP connections, pending signals, zombies, signalfd, file locks, epoll, etc.

- Full checkpointing: Minimize the net utilization and have the minimun failure risk but the downtime is equal to the duration of the migration.

- Pre-copy: the memory data are transfered while the checkpoint is still running. The memory is copied on the target node using iterations while the process keeps running
- Post-copy: Inverting the pre-copy we have that the process is freezed and the only the necessary things to keep the process running are migrated, to re-run the process with the minimun downtime.

Related Work

Motivation

- **Co-locating Containerized Workload Using Service Mesh Telemetry**: The cloud-native architecture and container-based technologies are revolutionizing how online services and applications are designed, developed, and managed by offering better elasticity and flexibility to developers and operators. However, the increasing adoption of microservice and serverless designs makes application workload more decomposed and transient at a larger scale. Most existing container orchestration systems still manage application workload based on simple system-level resource usage and policies manually created by operators, leading to ineffective application-agnostic scheduling and extra management burden for operators.
  In this work, we focus on workload placement for containerized applications and services and argue for the integration of application-level telemetry for profiling application status and co-locating application workload. To avoid extra performance overhead and modifications to existing applications, we propose to use the telemetry collected by service mesh to model the application communication patterns with a graph-based representation. By applying a graph partitioning algorithm, we create co-location groups for application workload that minimize cross-group communication traffic to improve the overall application performance, i.e., response time. Our preliminary experiments with a realistic online e-commerce application show that our solution can reduce the average response time by up to 58% compared to the default Kubernetes scheduler.

- **NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh**: Container technology has revolutionized the way software is being packaged and run. The telecommunications industry, now challenged with the 5G transformation, views containers as the best way to achieve agile infrastructure that can serve as a stable base for high throughput and low latency for 5G edge applications. These challenges make optimal scheduling of performance-sensitive containerized workflows a matter of emerging importance. Meanwhile, the wide adoption of Kubernetes across industries has placed it as a de-facto standard for container orchestration. Several attempts have been made to improve Kubernetes scheduling, but the existing solutions either do not respect current scheduling rules or only considered a static infrastructure viewpoint.To address this, we propose NetMARKS - a novel approach to Kubernetes pod scheduling that uses dynamic network metrics collected with Istio Service Mesh. This solution improves Kubernetes scheduling while being fully backward compatible. We validated our solution using different workloads and processing layouts. Based on our analysis, NetMARKS can reduce application response time up to 37 percent and save up to 50 percent of inter-node bandwidth in a fully automated manner. This significant improvement is crucial to

Kubernetes adoption in 5G use cases, especially for multi-access edge computing and machine-to-machine communication.

- **Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes**: Novel applications will require extending traditional cloud computing infrastructure with compute resources deployed close to the end user. Edge and fog computing tightly integrated with carrier networks can fulfill this demand. The emphasis is on integration: the rigorous delay constraints, ensuring reliability on the distributed, remote compute nodes, and the sheer scale of the system altogether call for a powerful resource provisioning platform that offers the applications the best of the underlying infrastructure. We therefore propose Kubernetes-edge-scheduler that provides high reliability for applications in the edge, while provisioning less than 10% of resources for this purpose, and at the same time, it guarantees compliance with the latency requirements that end users expect. We present a novel topology clustering method that considers application latency requirements, and enables scheduling applications even on a worldwide scale of edge clusters. We demonstrate that in a potential use case, a distributed stream analytics application, our orchestration system can reduce the job completion time to 40% of the baseline provided by the default Kubernetes scheduler.

- **Improving microservice-based applications with runtime placement adaptation**: Microservices are a popular method to design scalable cloud-based applications. Microservice-based applications (µApps) rely on message passing for communication and to decouple each microservice, allowing the logic in each service to scale independently. Complex µApps can contain hundreds of microservices, complicating the ability of DevOps engineers to reason about and automatically optimize the deployment. In particular, the performance and resource utilization of a µApp depends on the placement of the microservices that compose it. However, existing tools for µApps, like Kubernetes, provide minimal ability to influence the placement and utilization of a µApp deployment.
  In this paper, we first identify the runtime aspects of microservice execution that impact the placement of microservices in a µApp. We then review the challenges of reconfiguring a µApp based on these aspects. Our main contribution is an adaptation mechanism, named REMaP, to manage the placement of microservices in an µApp automatically. To achieve this, REMaP uses microservice affinities and resource usage history. We evaluate our REMaP prototype and demonstrate that our solution is autonomic, lowers resource utilization, and can substantially improve µApp performance.

Similar solutions to similar problems

- **Migrating Pods in Kubernetes**: Since its release in 2014, Kubernetes has become the leading container orchestration platform. Its ability to provide a consistent interface across different public cloud providers, private clouds, as well as bare metal deployments is one of the main reasons for its popularity. When attempting to rely solely on Kubernetes for deployment, it may be necessary to deploy applications that have not been designed for cloud environments. Kubernetes considers resources, especially Pods, to be ephemeral and requires multiple instances of applications to be deployed if high-availability should be guaranteed. If a node has to be taken down for maintenance, Pods are simply killed and restarted somewhere else. When doing so with applications that do not support high-availability, this will lead to downtime. For virtual machines, this problem has been solved

with live migration, which allows moving virtual instances to another host without turning them off. Popular container runtimes like Docker support checkpoint and restore mechanisms, which would allow implementing (live) migration for containers, and therefore Kubernetes, as well. The increasing use of edge computing enables additional use cases for migration. For example, workloads could be migrated between different edge nodes to minimize latency for moving clients in 5G networks. Supporting Pod migration would strengthen Kubernetes' suitability for deploying on the edge. This thesis analyzes how Pod migration can be integrated into Kubernetes. It argues that Pods should be cloned instead of attempting to reschedule them. By deleting the old Pod, the result is identical to an actual migration. The introduction of a MigratingPod resource is proposed to properly reflect the migration in the API, with a migration controller handling the migration itself. A prototype was implemented to evaluate the proposed design, which demonstrates that the approach is viable and integrates well with existing components. A benchmark shows that the chosen strategy to create full checkpoints is sufficient to migrate small workloads with reasonable downtimes. To achieve full network transparency and to be able to migrate larger Pods, faster migration strategies like pre- and post-copy need to be implemented by container runtimes and Kubernetes itself.

- **Jelastic**: To ensure the smooth and seamless cloud regions' interaction, the process of migration between them should cause zero application downtime and require zero configurations afterward. The live migration option suits perfectly for this purpose.
  At Jelastic, it is implemented with the help of CRIU (Checkpoint/Restore In Userspace) as a part of the leveraged OpenVZ container-based virtualization technology, and P.Haul - specially developed Python-powered mechanism on top of CRIU, intended for live migration of containers and memory-touching processes inside.
  This allows to freeze a running environment and save the data it is currently operating with to a hard drive (i.e. make a checkpoint). As a result, an application can be easily transferred to a different location, where it will be restored in the same state it was frozen with and continue working as before. Herewith, such a relocation takes 10-15 seconds in average, during which all the incoming requests are queued, so users may only experience a brief single-time delay in the server response. Add the advanced traffic load balancing solution to it, and you will be able to migrate a bunch of containers without redeploy and downtime as well.
  In a combination with the approach of composing a cluster from multiple hardware sets, this technology becomes especially efficient. In order to prove this, let's consider a couple of real use cases - e.g. with live migration across data centers of such competitive vendors as Microsoft Azure and Amazon Web Services.
  Another good example is migration between regions of different cloud infrastructure vendors. This allows to achieve the high availability across multiple clouds and ensures disaster recovery through the unification of DevOps workloads deployment. For example, you can instantly cope with any performance issues, like maintenance works on your current region or temporary load burst, through simple evacuation of all your containers to another DC.
  One more benefit here is the ability to relocate applications according to the current business requirements and, in such a way, gain the most sufficient quality/price/SLA rate - for example, less performance region can be used as a Private Cloud for the development/testing zone, while another Public one provides the sufficient capacity for

production. Smooth migration ensures all these changes can be performed without bothering about possible downtimes and profit losses.

- **Migrating Deep Learning Data and Applications among Kubernetes Edge Nodes**: Many current IoT applications deployed at the edge use deep learning (DL) in their real-time processing and analytics. Not only inference but also training is moving to edge devices. DL application and dataset migration among these devices are mandatory for scenarios like node failure, user mobility or when nodes need to collaborate (e.g., distributed training). Container technologies and Kubernetes (K8s) are being increasingly adopted to manage infrastructure at the edge. Unfortunately, there is no built-in mechanism in K8s to support migration of stateful containers between its cluster nodes. The K8s cluster's master node generally launches a new fresh container in another node to replace the failed one. While there is an existing mechanism for migrating a Pod between K8s nodes, there is no past work investigating the migration of DL datasets and containerized DL applications among K8s cluster nodes. In this paper, we present our 1) comprehensive study on the effectiveness and limitations of existing checkpointing mechanisms for containerized DL applications and 2) our comparative performance study of several approaches in migrating DL datasets and applications in a K8s cluster. Our results show that migrating states of DL applications and restoring them from their previous states enables faster recovery (reducing training time by 10 to 73 percent) than re-running these models from the beginning regardless of the percentage of epochs that have completed. Additionally, our experimental results show that transferring a dataset between K8s workers using the K8s persistent volume with kubectl cp is generally suitable and efficient. However, when network latency is high, using our customized middleware with a feedback controller to migrate data in parallel can speed up total migration time compared to the K8s's persistent volume approach alone.

- **Enabling Live Migration of Containerized Applications Across Clouds**: Live migration, the process of transferring a running application to a different physical location with minimal downtime, can provide many benefits desired by modern cloudbased systems. Furthermore, live migration between different cloud providers enables a new level of freedom for cloud users to move their workloads around for performance or business objectives without having to be tied down to any single provider. While this vision is not new, to-date, there are few solutions and proof-of-concepts that provide this capability. As containerized applications are gaining popularity, we focus on the design and implementation of live migration of containers across cloud providers. CloudHopper, our proof-of-concept live migration service for containers to hop around between Amazon Web Services, Google Cloud Platform, and Microsoft Azure is evaluated using a common web-based workload. CloudHopper is automated and supports pre-copy optimization, connection holding, traffic redirection, and multiple interdependent container migration. It is applicable to a broad range of application use cases.

- **EVPN/SDN Assisted Live VM Migration between Geo-Distributed Data Centers**: Live Virtual Machine (VM) migration has significantly improved the flexibility of modern Data Centers (DC). However, seamless live migration of a VM between geo-distributed DCs faces several challenges due to difficulties in preserving the network configuration after the migration paired with a large network convergence time. Although SDN-based approaches can speed up network convergence time, these techniques have two limitations. First, they

typically react to the new topology by installing new flow rules once the migration is finished. Second, because the WAN is typically not under SDN control, they result in sub-optimal routing thus severely degrading the network performance once the VM is attached at the new location. In this paper, we identify networking challenges for VM migration across geo-distributed DCs. Based on those observations, we design a novel long-haul VM migration scheme that overcomes those limitations. First, instead of reactively restoring connectivity after the migration, our SDN-based approach proactively restores flows across the WAN towards the new location with the help of EVPN and VXLAN overlay technologies. Second, the SDN controller accelerates the network convergence by announcing the migration to other controllers using MP-BGP control plane messages. Finally, the SDN controller resolves the sub-optimal routing problem that arises as a result of migration implementing a distributed anycast gateway. We implement our approach as extensions to the OpenDaylight controller. Our evaluation shows that our approach outperforms existing approaches in reducing the downtime by 400 ms and increasing the application performance up to 12 times.

- **Stateful Container Migration in Geo-Distributed Environments**: Container migration is an essential functionality in large-scale geo-distributed platforms such as fog computing infrastructures. Contrary to migration within a single data center, long-distance migration requires that the container's disk state should be migrated together with the container itself. However, this state may be arbitrarily large, so its transfer may create long periods of unavailability for the container. We propose to exploit the layered structure provided by the OverlayFS file system to transparently snapshot the volumes' contents and transfer them prior to the actual container migration. We implemented this mechanism within Kubernetes. Our evaluations based on a real fog computing test-bed show that our techniques reduce the container's downtime during migration by a factor 4 compared to a baseline with no volume checkpoint.

- **Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes**: Container technology has become a very popular choice for easing and managing the deployment of cloud applications and services. Container orchestration systems such as Kubernetes can automate to a large extent the deployment, scaling, and operations for containers across clusters of nodes, reducing human errors and saving cost and time. Designed with "traditional" cloud environments in mind (i.e., large datacenters with close-by machines connected by high-speed networks), systems like Kubernetes present some limitations in geo-distributed environments where computational workloads are moved to the edges of the network, close to where data is being generated/consumed. In geo-distributed environments, moving around containers, either to follow moving data sources/sinks or due to unpredictable changes in the network substrate, is a rather common operation. We present MyceDrive, a stateful resource migration solution natively integrated with the Kubernetes orchestrator. We show that geo-distributed Kubernetes pod migration is feasible while remaining fully transparent to the migrated application as well as its clients, while reducing downtimes up to 7x compared to state-of-the-art solutions.

- **Stateful Container Migration employing Checkpoint-based Restoration for Orchestrated Container Clusters**: Recently, container-leveraged cloud computing technology is being adopted in a wide range of ICT areas. For a more effective management of diversified containers over multiple machine nodes, several container orchestrators including

Kubernetes have been utilized in production cloud environments. Due to reliability reasons, stateful containers in an orchestrated container cluster are sometimes required to be migrated to other clusters. Typically, a storage-based migration approach is commonly adopted despite its considerable downtime, which also requires the application-level implementation of storage components for state save and restore. Thus, in this paper, we propose a checkpoint-based approach of stateful container migration for orchestrated container clusters. By checkpointing and restoring in-memory states of processes that are running in the containers with CRIU (Checkpoint/Restore in Userspace) feature, container migration can be realized without the application-level implementation of storage components. We then verify the proposed approach at a PoC(Proof-of-Concept) level by managing the associated migration procedure with the container orchestrator.

- **Proactive Stateful Fault-Tolerant System for Kubernetes Containerized Services**: Recently, the development of Kubernetes (K8s) containerization platform has enabled cloud-based, lightweight, highly scalable, and agile services in both general and telco use-cases. Ensuring high availability, reliable and continuous containerized services is a major requirement of service providers to provide fault-tolerance, transparent service experiences to end-users. To satisfy this requirement, fault prediction and proactive stateful service recovery features must be applied in cloud systems. Prior proactive failure recovery approaches mostly focused on either improving fault prediction performance based on different machine learning time series forecasting techniques or optimizing recovery service placement after fault prediction. However, a mechanism that enables stateful containerized service migration from the predicted faulty node to the healthy destination node has not been studied. Service migration in previous proactive works is only simulated or performed by virtual machine (VM) migration techniques. In this paper, we propose a proactive stateful fault-tolerant system for K8s containerized services that pipelines a Bidirectional Long Short-Term Memory (Bi-LSTM) fault prediction framework and a novel K8s stateful service migration mechanism for service recovery. Experimental results show how the Bi-LSTM model improved prediction performance against other time-series forecasting models used prior proactive works. We then combined the Bi-LSTM fault prediction framework with both the default K8s and our stateful migration mechanisms. The comparison between these two proactive systems proves our system efficiency in terms of reducing Quality of Service (QoS) violation percentage and service recovery time.

- **MS2M: A message-based approach for live stateful microservices migration**: In the last few years, the proliferation of edge and cloud computing infrastructures as well as the increasing number of mobile devices has facilitated the emergence of many novel applications. However, that increase of complexities also creates novel challenges for service providers, for example, the efficient management of interdependent services during runtime. One strategy is to reallocate services dynamically by migrating them to suitable servers. However, not every microservice can be deployed as stateless instances, which leads to suboptimal performance of live migration techniques. In this work, we propose a novel live migration scheme focusing on stateful microservices in edge/cloud environments by utilizing the underlying messaging infrastructure to reconstruct the service's state. Not only can this approach be applied in various microservice deployment scenarios, experimental evaluation results also show a reduction of 19.92% downtime compared to the stop-and-copy migration method.

- **Migrating Pods with Containerized Applications Between Nodes in the Same Kubernetes Cluster Using Cloudify**: Cloudify team decided to use Kubernetes running on OpenStack to demonstrate how to migrate containerized applications between nodes. This solution allows users to define a specific node for deployment of an application. The idea is to avoid service disruption by migrating pods from one node to another, each with a containerized application, within the Kubernetes cluster. The user provides:
    - Node1 label
    - Node2 label
    - External IP address to be exposed
      On Node1, Cloudify deploys two pods, each with containerized applications:
    - App container in Pod1
    - DB container in Pod2
      The containers are grouped and exposed as a Kubernetes Service at `http://<ip-address>:8080`.
      The Kubernetes pods (with their containers) are moved to Node2 using a custom workflow operation with Cloudify.