The Schedulers

In this lesson, we will get familiar with the container schedulers.

WE'LL COVER THE FOLLOWING

An Analogy

Picture some young teenagers. After school, they'd go to a courtyard and play soccer. That was an exciting sight. A random number of kids running around the yard without any orchestration. There was no offense and no defense. They'd just run after a ball.

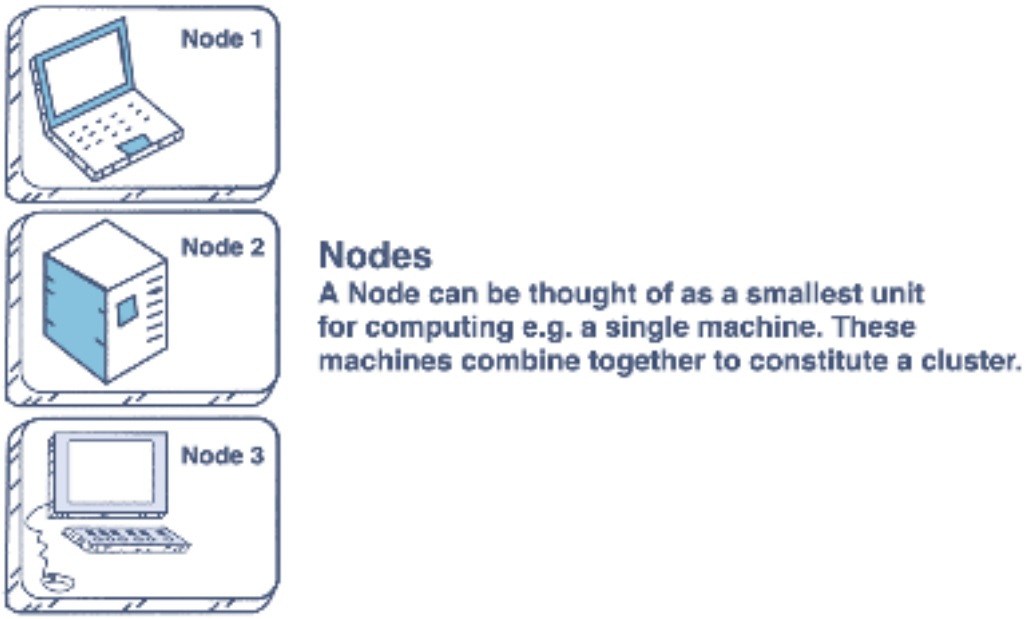
Everyone moves forward towards the ball, someone kicks it to the left, and kids move in that direction, only to start running back because someone kicked the ball again. The strategy was simple. Run towards the ball, kick it if you can, wherever you can, repeat. It's hard to understand how anyone managed to score. It was complete randomness applied to a bunch of kids. There was no strategy, no plan, and no understanding that winning required coordination.

If that was a "real" team, they'd need a coach. They'd need someone to tell us what the strategy is, who should do what, and when to go on the offense or fall back to defend the goal. They'd need someone to orchestrate them.

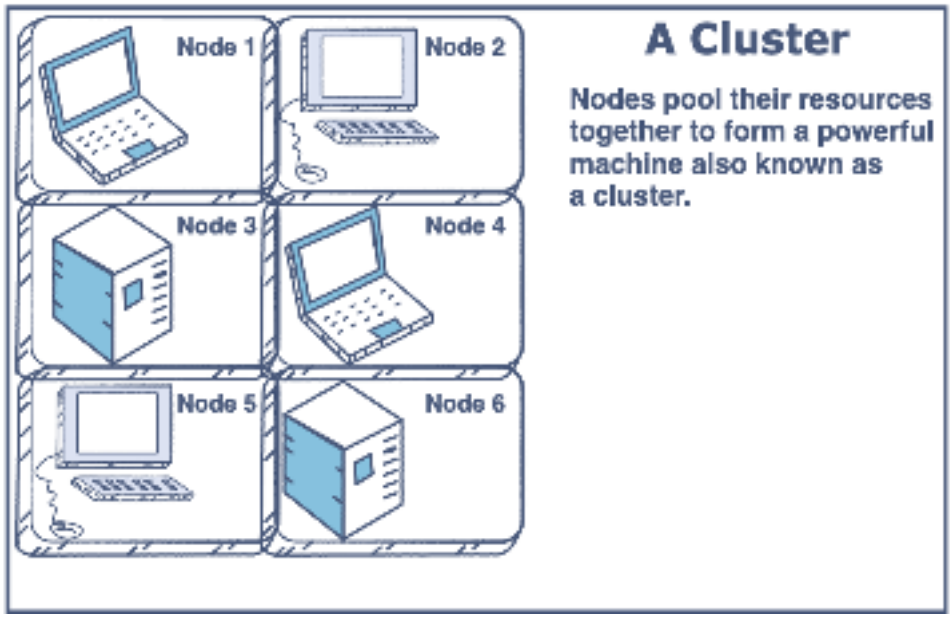
The field (a cluster) had a random number of people (services) with the common goal (to win). Since anyone could join the game at any time, the number of people (services) was continually changing. Someone would be

injured and would have to be replaced or, when there was no replacement, the rest of us would have to take over his tasks (self-healing).

The following illustrations will give you a basic idea of a node and a cluster.



1 of 2



2 of 2

The Relatable Part

Those football games can be easily translated into clusters. Just as the kids

needed someone to tell them what to do (a coach), clusters need something to orchestrate all the services and resources. Both need not only to make up-front decisions, but also to continuously watch the game/cluster, and adapt the strategy/scheduling depending on the internal and external influences. Kids needed a coach and clusters need a scheduler. They need a framework that will decide where a service should be deployed and make sure that it maintains the desired run-time specification.

Why Use Schedulers?

A cluster scheduler has quite a few goals.

- It makes sure that resources are used efficiently and within constraints.
- It makes sure that services are (almost) always running.
- It provides fault tolerance and high availability.
- It makes sure that the specified number of replicas are deployed.
- It makes sure that the desired state requirement of a service or a node is (almost) always fulfilled. Instead of using imperative methods to achieve our goals, with schedulers, we can be declarative.
- We can tell a scheduler what the desired state is, and it will do its best to ensure that our desire is (almost) always fulfilled. For example, instead of executing a deployment process five times hoping that we'll have five replicas of a service, we can tell a scheduler that our desired state is to have the service running with five replicas.

The difference between imperative and declarative methods might seem subtle but, in fact, is enormous. With a declarative expression of the desired state, a scheduler can monitor a cluster and perform actions whenever the actual state does not match the desired. Compare that to an execution of a deployment script. Both will deploy a service and produce the same initial result. However, the script will not make sure that the result is maintained over time. If an hour later, one of the replicas fail, our system will be compromised.

Traditionally, we were solving that problem with a combination of alerts and manual interventions. An operator would receive a notification that a replica

failed, he'd log in to the server, and restart the process. If the whole server is down, the operator might choose to create a new one, or he might deploy the failed replica to one of the other servers. But, before doing that, he'd need to check which server has enough available memory and CPU. All that, and much more, is done by schedulers without human intervention.

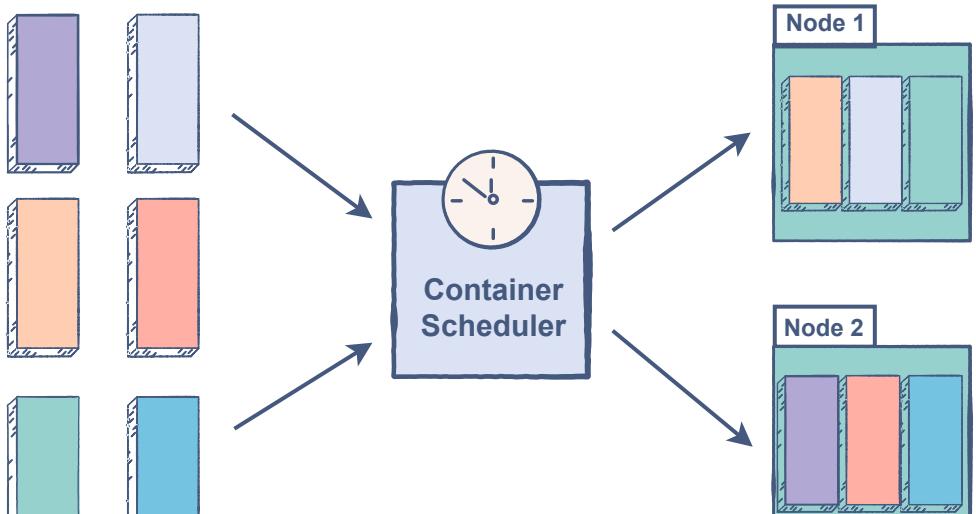
Think of schedulers as operators who are continually monitoring the system and fixing discrepancies between the desired and the actual state. The difference is that schedulers are infinitely faster and more accurate. They do not get tired, they do not need to go to the bathroom, and they do not require paychecks. They are machines or, to be more precise, software running on top of them.

The Container Schedulers

That leads us to container schedulers. How do they differ from schedulers in general?

Container schedulers are based on the same principles as schedulers in general. The significant differences between a scheduler and a container scheduler are:

- They are using containers as the deployment units.
- They are deploying services packaged as container images.
- They are trying to collocate them depending on desired memory and CPU specifications.
- They are making sure that the desired number of replicas are (almost) always running.



All in all, they do what other schedulers do but with containers as the lowest and the only packaging unit. And that gives them a distinct advantage. They do not care what's inside. From a scheduler's point of view, all containers are the same.

Why to Combine Containers and Schedulers?

Containers provide benefits that other deployment mechanisms do not.

- Services deployed as containers are isolated and immutable.
- Isolation provides reliability.
- Isolation helps with networking and volume management. It avoids conflicts. It allows us to deploy anything, anywhere, without worrying whether that something will clash with other processes running on the same server.
- Schedulers, combined with containers and virtual machines, provide the ultimate cluster management nirvana.
- They allow us to combine the developer's necessity for rapid and frequent deployments with a sysadmin's goals of stability and reproducibility.

And all that leads us to Kubernetes...

The next lesson is all about getting the Kubernetes introduced to you.