Design and Implementation of a Simple Cloud/Cluster

Due: Check My Courses

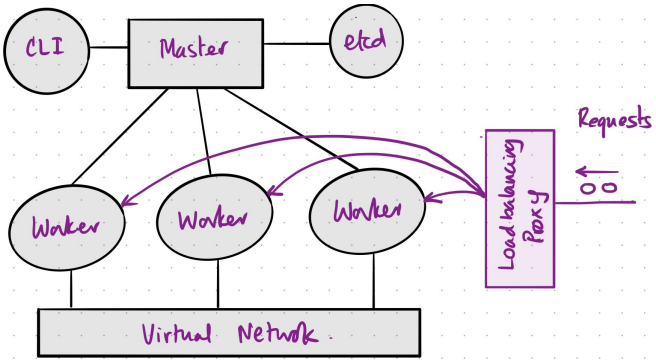
Objective

To design and implement a simple cloud (more precisely a cluster of Docker containers) with the following features.

1. *Manager-worker cluster architecture*: The manager node is responsible for managing the workers that actually run the workload. We would have a single manager in the simple prototype you are developing. Still, we want to tolerate manager failures.
2. *Scalable system*: A system that can be manually scaled up or down. That is the number of workers dedicated to an application should be adjustable by the user.
3. *Autoscaling*: In addition to the above manual scaling, we want auto-scaling. In auto-scaling, we can set maximum limits and the manager is responsible for selecting the best operating point (number of workers) for an application.
4. *Multiple applications*: We need to support multiple applications at the same time.
5. *Load balancing proxy*: The cloud system could run different applications at the same time. We need a load balancer to distribute the requests for the different applications to the backends for the applications.

Project Overview

The following diagram shows the overall architecture of a simple cloud. We have a master node that is responsible for the overall resource management. The master node is also the point of control for the simple cloud. That means the master node needs to have a command line interface so that we can issue commands to control the operation of the cloud. For example, we could scale up/down the number of workers allocated to an application, launch (start) an application, or un-launch (stop) and application. The master node is going to store its state in a fault tolerant data store. We want to use something like **etcd** for that purpose.



The workers are containers that are meant to run the application. A worker would run a specific instance of an application. Depending on the application loading requirements (in terms of throughput), the user launching the application would have requested a given number of replicas. Or, the number of replicas would be scaled up or down depending on the requirements set by the user and loading situation. The loading situation could be measured by the number of requests coming into a replica. For example, a web server could service a fixed number of requests per second, if we have fewer replicas, the web server replica will be operating beyond an acceptable loading level. So, we need to create another replica and spread the requests over more replicas thus reducing the amount of loading in a single replica.

The worker containers would be networked with each other using a Linux bridge or OpenVSwitch. You can select one of them for your implementation.
We can have many replicas serving a particular application. For example, in the above web server example there would be many web servers in the cloud. When a request comes in, we need to route the request to one of the web server replicas. That is the job of the load balancer and proxy. The load balancer/proxy module hides the fact that the application is actually deployed on many containers. With the load balancer, it looks like there is one larger server. Also, the load balancer can hide the downtimes of the different containers provided the load balancer itself is informed about the health status of the replicas.

Descriptions of Components

We provide additional descriptions on the components of the cloud architecture. The design sketched here can be slighted adjusted, but a drastic simplification should be avoided. For example, the manager as described here has a CLI. You can replace it with a design that provides a Web UI to the manager.

Manager: This is a central component of the cloud/cluster system. We are using a single instance of the manager in this design. It is important to make it fault tolerant. The way we want to implement fault tolerance is to keep all the state outside the manager in a fault tolerant data store like **etcd**. This way the manager could restart at any given time and still everything will be running as it should. You need to test this in your implementation by crashing the manager and checking that the manager is able to perform its functions without any problems.

You need to implement the following or equivalent commands in the manager: start a worker with a specific command, stop a worker, list all workers, scale the number of workers running a given command, and other commands you prefer to have.

The command that you want to run is a Linux executable program or command. In the simple cloud, all commands that you want to execute are running inside containers (workers). If the command you want to execute is a custom executable program, you need to copy that into the container and then start it. For example, if you want to start **nc** (**netcat**) you don't need to copy the **nc** executable file into the container because the container would have a copy already. You can just start it with the given arguments.

Worker: These are containers running the application instances with the arguments supplied by the user. You need to create as many worker instances as needed to support the application. The initial command that is launching the application could specify the number of workers. Alternatively, we could have the number of workers adjusted by the user after the application is launched. The adjustments are done by the scaling commands. Yet another alternative is to automatically scale up/down to meet the performance goals.

The manager observes the workers using a command like **docker stats** (using docker stats we can gather the resource usage levels of the different containers). The manager can use the resource usage values to implement the auto-scaling schemes.

Proxy: The proxy could be implemented using HAProxy, Trafeik, or similar systems. This is essentially a load balancer that acts as a façade to the applications that are hosted in the different containers. The easiest is to deploy the proxy in a container. You can configure the proxy so that it points to the containers running the applications. It is important that your design allows the number of application containers to change at runtime. So, the proxy should not depend on a static configuration file to know about the application containers. Luckily, information about the backends can be dynamically supplied to many popular proxies such as Trafiek.

Network bridge: This is another important element of the cluster. It provides the connectivity and also could play a role in interconnecting the proxy to the application containers depending on your choice of the proxy.

Other Implementation Notes

You need to get started early with this project. This handout is NOT completely specifying the system. Several important parameters are undefined and left for your design. For example, what parameters the manager should handle is left for you to decide.

Here are the important milestones for the project.

1. Demonstrating the system is fault tolerant for manager failure. You need to demonstrate that the workers are not affected by manager failures. We should be able to restart a failed manager and the system should continue without any issues.
2. Manual scaling of application deployments. We should be able to scale up/down the number of workers running a particular application.
3. The simple cloud should be able to host many applications at the same time. That is, we should be able to have workers running for different applications without interfering with each other. In particular, the load balancer or proxy should be able to route the requests for an application to the containers that host that particular application.
4. Auto-scaling of the containers allocated for a given application is another important aspect of the design. **This is also the hardest part of the project**.

Grading Scheme

1. Completing milestone: 1 up to 60%
2. Completing milestones: 1 & 2: up to 80%
3. Completing milestones: 1, 2, & 3 up to 100%
4. Completing milestones: 1, 2, 3, & 4 up to 120%