**Multi-cluster Kubernetes**

**Introduction**

This project extends traditional Kubernetes to a multi-cluster environment, providing an interface for several different clusters to communicate with and update each other. Clusters are separated into regional groups that represent a geographic locality (ex: the US region, or the Europe region). Once the multi-cluster extension is running on each cluster, a user can input the number of service replicas desired in a region. Any cluster can accept an update from a user, making the system very decentralized and easy to access. This change is then scheduled to specific clusters in the region, and the new global state is propagated throughout the global multi-cluster system. When the relevant clusters receive the updated global configuration, they update their deployments to reflect the rule input by the user.

**Software Structure**

**Global Configuration:**

The global configuration of the multi-cluster Kubernetes system is described by a pointer to the GlobalConfigLocal struct (which is defined in configHelper.cpp); each cluster has a GlobalConfigLocal pointer that describes the current global state. The GlobalConfigLocal struct consists of a vector of pointers to RegionConfigLocal structs, which each describe a region in the multi-cluster system that services can be scheduled to. The RegionConfigLocal struct consists of a regionName as an identifier (ex: “us”) and a vector of pointers to ClusterConfigLocal structs, which each describe a cluster within the region. Each ClusterConfigLocal struct consists of a clusterName as an identifier (ex: “us-east”) and a vector of pointers to ServiceConfigLocal structs, which each describe a service running on the cluster. Each ServiceConfigLocal struct consists of the service name (ex: “nginx”) and how many copies (replicas) of the service are running on the cluster. This hierarchical design provides a simple interface for the current global configuration of a multi-cluster system.

**Scheduler:**

The scheduler (defined in scheduler.cpp) operates using a simple algorithm that serves as a proof-of-concept for multi-cluster global scheduling. The current global configuration is input into the scheduler function, along with a regionName and the number of copies of a service that should be running in that region after scheduling. Upon receiving this, the scheduler goes through each service desired in the new region and counts the number of copies that are already running in the region. The scheduler will then start adding or removing copies of that service (one at a time) in clusters within the region, until the desired number of services in the region is achieved. During scheduling, the global configuration variable is updated so that after all scheduling is complete, the new global configuration can be used to modify the deployments running on each Kubernetes cluster.

**Local Updater:**

Once the global configuration is updated, each cluster must update the deployments running on it according to the new global configuration it has received. This is done through the updateLocal function (defined in updateLocal.cpp), which takes in the regionName and clusterName of the calling cluster and the global configuration that includes the services/deployments the cluster should be running. Upon identifying the portion of the global configuration that involves the calling region/cluster, the local updater uses Python scripts to call the Kubernetes API and create, update, or remove deployments. For a new deployment, the API sets up a default configuration that sets the deployment to listen on port 8080 (assuming a valid Docker container of the deployment exists).

**Analysis**

This project has many areas it needs improvement in to function as a robust piece of production code. For example, the scheduling algorithm is simple and will load more services to the first clusters within the region’s cluster vector; there is no proper load balancing as services are added/removed. Also, there are no options to configure the deployment of a new service upon its creation, which would involve a more complex data structure to describe the global configuration of the system. Other edge cases (such as what if the Docker container for a deployment has a name different from <deployment>:latest?) are also not covered, as the goal was to get the basic system up and running for demonstration. Furthermore, the project should have been programmed in Go instead of C++. Programming in Go would have better integrated the additions to the standard Kubernetes environment and would have removed the use of Python scripts for accessing the Kubernetes API.

Further work is also needed to implement request forwarding among clusters within a region. Upon receiving a request for a service it is not currently running, a cluster should be able to forward that request to another cluster in the region that is running that service, acting as a proxy. This would strengthen the regional organization of the system and provide useful multi-cluster functionality. Some early ideas for this software were considered by the team but, again, ensuring a working demo was made a larger priority.