COMPX529-20A: Assignment 1

Individual Assignment
Submit on Moodle
Deadline: Aug 14, 6pm
Weight: 15%

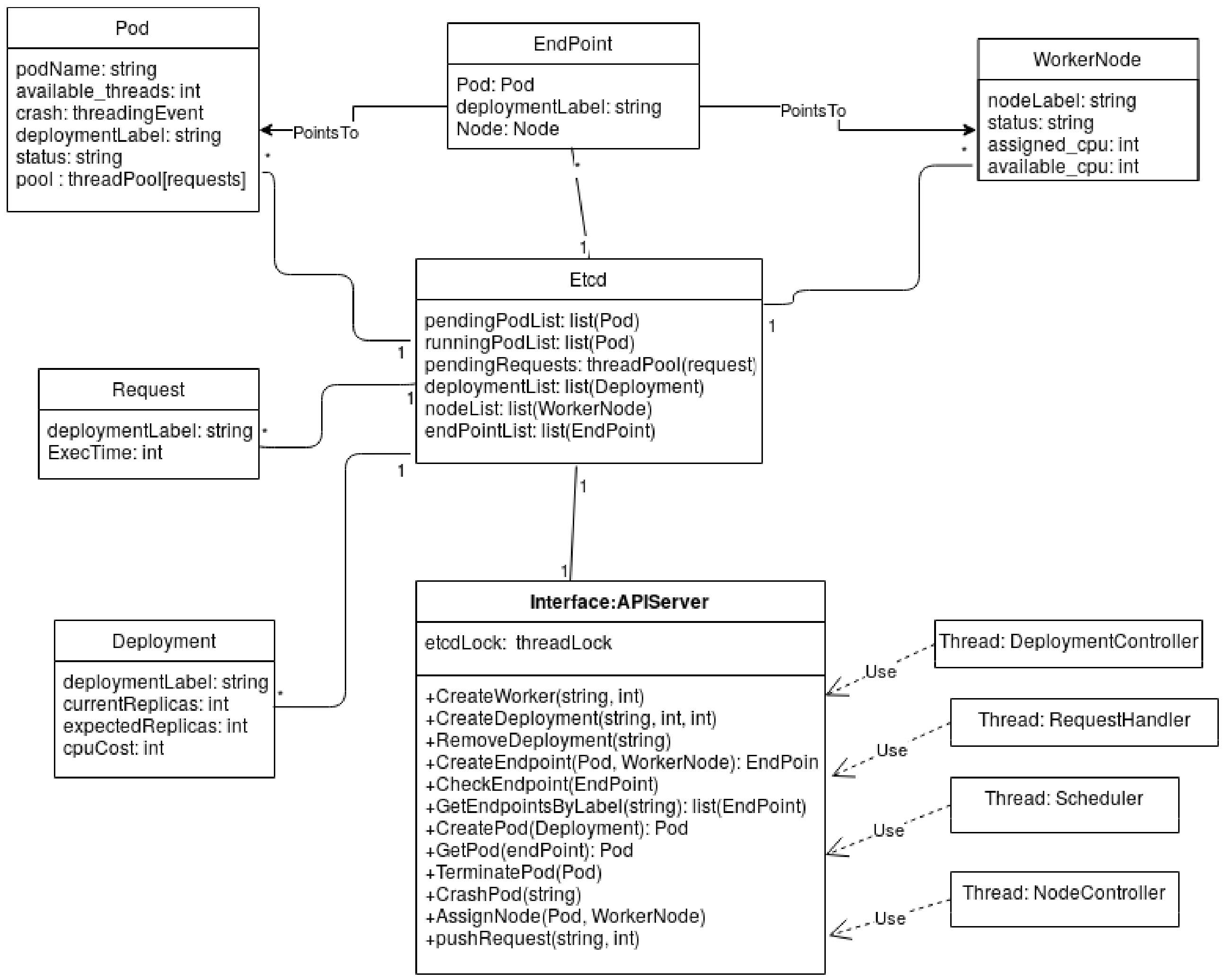
The Kubernetes (K8s) architecture deploys application containers across physical worker nodes in the form of pods. It is a self-healing architecture that dynamically keeps track of the status of its cluster-wide resources. Your task is to **implement and test a simulation of a functional Kubernetes cluster in Python**. You will be provided with a series of Python files that declare component classes and some basic methods for which you must define expected behaviour by adding the remaining code. Your simulator will be of a basic cluster with deployed services whose workload is cpu bound.

Your application will read in a file that will generate events line by line driven by instructions in the file. Instructions will be formatted as follows:

```
AddNode [NodeLabel] [TotalCPUs]
Deploy [Name] [RequestedCPUsPerReplica] [Expected Replicas]
DeleteDeployment [DeploymentName]
ReqIn [Label] [Deployment] [ExecTime]
CrashPod [DeploymentName]
Sleep [ms]
```

AddNode, adds a node to the cluster. **Deploy** indicates that a new service needs to be added, whereas **DeleteDeployment** removes a service. After a service is deployed, the system will create a number of pods and place them on the nodes of the cluster. These pods will be handling incoming requests, which are fed in the system by **ReqIn**. However, a pod might crash by **CrashPod**. Finally, **Sleep** indicates the amount of milliseconds the simulator should wait before executing the next command—-in the meantime, the cluster threads might be working to complete their various tasks.

The classes for you to implement are outlined in the UML diagram below:



A Deployment describes a group of Pod replicas deployed across WorkerNodes on your cluster. It will have a label, along with an expected number of Pods to be deployed on the system and a parameter for the number of threads that each Pod will be allocated.

A Pod is the scaling unit for your simulated workload. Each pod will have a pool of threads, simulating assigned cores in a real world context. When requests are handled, they will utilise a thread and sleep it for a given time duration in seconds. If a Pod crashes it will not be able to handle any requests and the crash event will be set, interrupting running threads.

A WorkerNode represents physical machines in a Kubernetes cluster. It has a deploymentLabel and has assigned and available cpu values representing its thread capacity.

EndPoint objects keep track of which Pods relate to which Deployments and are deployed on which Nodes.

Etcd is the database of the cluster and will contain:

- A list of Deployment objects.
- A a list of WorkerNode objects

- A list of EndPoint objects.
- A list of pending pod objects to be deployed.

The DeploymentController thread checks the current number of pods against the expected number associated with each deployment. It will create or destroy pod instances as required. New pods are given a “Pending” label and placed in the Etcd pendingPod list for scheduling. Pods to be deleted are given a “Terminating” label and are removed from their associated node once their active requests have been completed.

The NodeController thread checks the information on each Worker stored in etcd. It will update EndPoints if they are no longer valid, and will restart any failed pods found deployed on a given node.

The Scheduler thread checks for any pending pods that are waiting to be deployed and determines an appropriate node on which it can be housed. The pod is then transferred to the runningPodList in etcd and an endpoint object is created to reflect the deployment location. If there are no nodes available, the pod will remain pending.

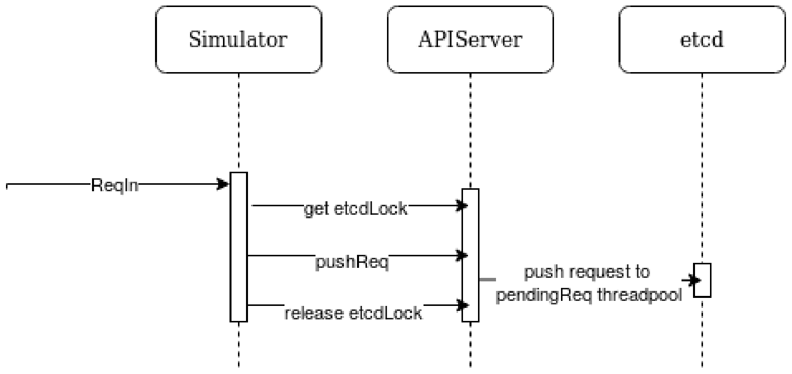
The RequestHandler will periodically check the pendingRequest queue and engage one of the threads within the pool of a Pod associated with the given Deployment. Requests will be distributed to Pods on a first-fit basis.

All of these threads interact with the data stored within Etcd using the methods defined within the APIServer.

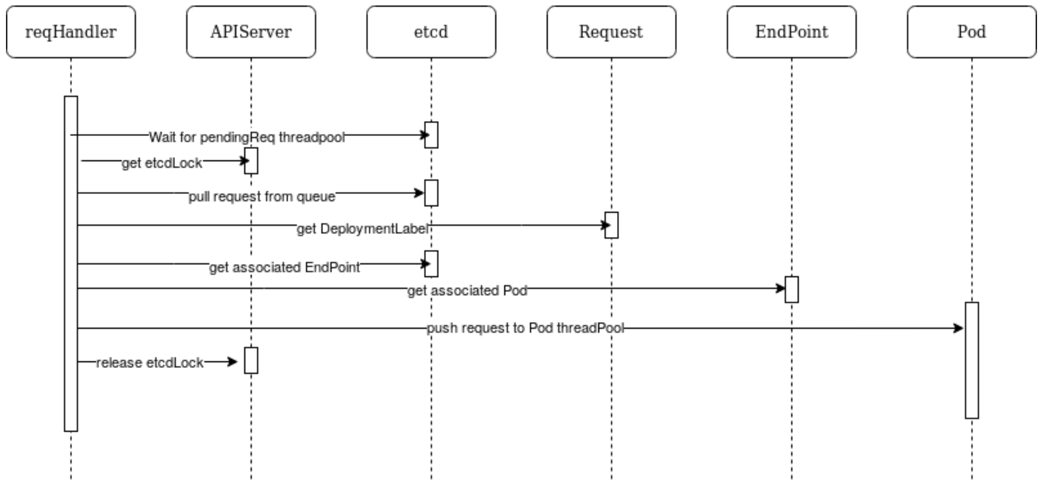
You will be provided with stub files for all of the relevant classes, along with a runSimulator file that will use threads to invoke API methods.

Your task will be to define the behaviour of these methods and implement the necessary control loops within the running threads. Then test the overall system by using three generated trace files we provide capturing the various metrics produced by the system over time.

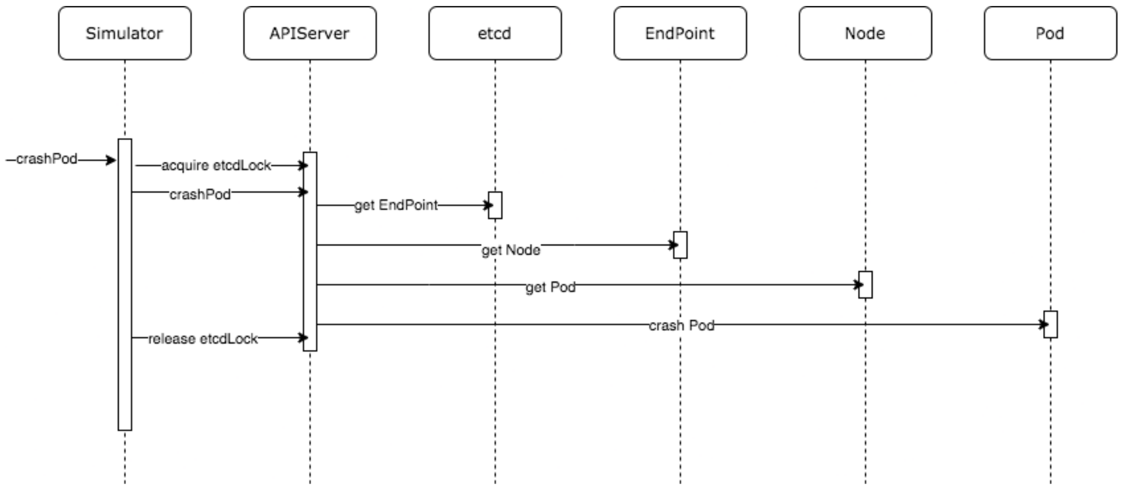
Some examples of the way in which information is handled within the simulator can be seen on the following page.



ReqIn (Part 1/2): A Request coming into the system



ReqIn (Part 2/2): The request handler engaging a Pod



CrashPod: A Pod being crashed

Deliverables:

1. Fill in the remaining code without making changes to the architecture of the solution. If a change to the architecture (classes, methods, fields) is needed, document your decision and illustrate the new architecture clearly. We have provided you with sample code to get you started; don't rely on it though as we haven't tested it :-)
2. Testing evidence that your simulator works (you can use the tracefiles of Step 3 or you can build your own smaller test cases).
3. Analysis of the results of running evaluation experiments for three different tracefiles (include figures, graphs, tables and appropriate discussion). For all three simulations, use the value of 5 for all cluster manager thread delays, as is specified in the runSimulator file. We will provide you a tracefile generator. You will need to input your student id in it and set the configurations to the following options for each of the three scenarios:
    a. Scenario A: ID = YourStudentId and Seed = 1.
    b. Scenario B: ID = YourStudentId and Seed = 2.
    c. Scenario C: ID = YourStudentId and Seed = 3.

4. A zip file containing all your work. Ensure your write any reports on a document processor, e.g., Word, and submit them as pdf.

*****End of Assignment 1*****