1. ETCD COORDINATOR

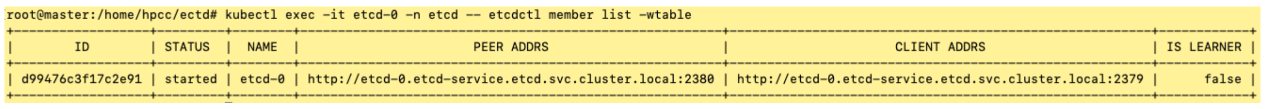
etcd is a lightweight, distributed key-value store designed for storing configuration data and coordinating distributed systems. Originally developed by CoreOS (now part of Red Hat), etcd provides a reliable way to maintain consistent state across clusters of machines through the Raft consensus algorithm. It serves as the central database for service discovery, configuration management, and leader election in modern cloud-native infrastructures. etcd acts as a “brain” of distributed system in which it keeps track of the cluster overall state, ensuring that all components share the same, up-to-date information, even in the face of node failures.

The following YAML file defines a Kubernetes deployment for etcd and it sets up a single-node etcd instance by default. To deploy a multi-node etcd cluster, simply increase the value of the “replicas” field and update the “–initial-cluster” argument to include all member nodes with the corresponding stable DNS names retrieved from headless service, separated by commas.

```yaml
# file: etcd.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: etcd
---
apiVersion: v1
kind: Service
metadata:
  name: etcd-service
  namespace: etcd
spec:
  clusterIP: None
  publishNotReadyAddresses: true
  selector:
    app: etcd
  ports:
  - name: client
    port: 2379
  - name: peer
    port: 2380
  - name: metrics
    port: 8080
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: etcd
  namespace: etcd
spec:
  serviceName: etcd-service
  replicas: 1
  podManagementPolicy: Parallel
  selector:
    matchLabels:
      app: etcd
  template:
    metadata:
      labels:
        app: etcd
    spec:
      containers:
      - name: etcd
        image: quay.io/coreos/etcd:v3.6.0
        ports:
        - containerPort: 2379
          name: client
        - containerPort: 2380
```

```
     name: peer
    - containerPort: 8080
     name: metrics
    env:
    - name: POD_NAME
     valueFrom:
      fieldRef:
       fieldPath: metadata.name
    command: ["/usr/local/bin/etcd"]
    args:
    - --name=$(POD_NAME)
    - --data-dir=/data
    - --listen-peer-urls=http://0.0.0.0:2380
    - --listen-client-urls=http://0.0.0.0:2379
    - --advertise-client-urls=http://$(POD_NAME).etcd-service.etcd.svc.cluster.local:2379
    - --initial-advertise-peer-urls=http://$(POD_NAME).etcd-service.etcd.svc.cluster.local:2380
    - --initial-cluster=etcd-0=http://etcd-0.etcd-service.etcd.svc.cluster.local:2380
    - --initial-cluster-state=new
    - --initial-cluster-token=etcd-etcd
    - --listen-metrics-urls=http://0.0.0.0:8080
    volumeMounts:
    - name: etcd-data
     mountPath: /data
 volumeClaimTemplates:
 - metadata:
   name: etcd-data
  spec:
   accessModes: ["ReadWriteOnce"]
   storageClassName: local-path
   resources:
    requests:
     storage: 1Gi
```

To verify that etcd has been deployed successfully, execute the command shown below. If the deployment is correct, the output should appear as illustrated in Figure 1.



```
root@master:/home/hpcc/ectd# kubectl exec -it etcd-0 -n etcd -- etcdctl member list -wtable
+------------------+---------+--------+--------------------------------------------------------+--------------------------------------------------------+------------+
|        ID        | STATUS  |  NAME  |                       PEER ADDRS                       |                      CLIENT ADDRS                      | IS LEARNER |
+------------------+---------+--------+--------------------------------------------------------+--------------------------------------------------------+------------+
| d99476c3f17c2e91 | started | etcd-0 | http://etcd-0.etcd-service.etcd.svc.cluster.local:2380 | http://etcd-0.etcd-service.etcd.svc.cluster.local:2379 |      false |
+------------------+---------+--------+--------------------------------------------------------+--------------------------------------------------------+------------+
```

Figure 1: CMD: kubectl exec -it etcd-0 -n etcd -- etcdctl member list -wtable.

etcd manages configuration data as a key-value store, where each key is organized hierarchically similar to a file path in a filesystem. etcd supports a wide range of value types, including strings, JSON objects, and serialized data. The following example demonstrates how to store a JSON file as the value of an etcd key.

```
# put value to a key
kubectl exec -it etcd-0 -n etcd -- etcdctl put <KEY> "$(cat <FILE_NAME>.json)"

# read value from a key
kubectl exec -it etcd-0 -n etcd -- etcdctl get <KEY>
```

The following code loads the JSON config from etcd and automatically updates it in real time whenever changes occur. The etcd3 can be installed via “pip install etcd3” and the endpoint url of etcd can be retrieved similar to Kafka.

```
import time
import json
import etcd3    # version 0.12.0

etcd = etcd3.client(host='etcd-0', port=2379)
CONFIG_KEY = "/config/monitor"
value, _ = etcd.get(CONFIG_KEY)
config_value = json.loads(value) if value else {}

def watch_config_key(watch_response):
    global config_value
    for event in watch_response.events:
        if isinstance(event, etcd3.events.PutEvent):
            config_value = json.loads(event.value.decode('utf-8'))

def main():
    watch_id = etcd.add_watch_callback(CONFIG_KEY, watch_config_key)
    try:
        while True:
            print(f"Using config: {config_value} to do something...")
            time.sleep(5)
    except KeyboardInterrupt:
        print("Stopping watcher...")
        etcd.cancel_watch(watch_id)

if __name__ == "__main__":
    main()
```

Heartbeat technique is commonly used in distributed systems to monitor the health of nodes. In etcd, a heartbeat can be implemented by periodically writing a value to a specific key that is associated with a time-to-live (TTL). Once the TTL expires, the key is automatically removed if no new heartbeat is received (node dead). The following code demonstrates how to send a heartbeat signal to the etcd server at regular intervals.

```
import time
import json
import etcd3    # version 0.12.0

etcd = etcd3.client(host='etcd-0', port=2379)
HEARTBEAT_KEY = "/monitor/heartbeat/node-1"
LEASE_TTL = 5  # seconds

def send_heartbeat():
    lease = etcd.lease(LEASE_TTL) # create a lease with TTL
    print(f"Lease created with TTL {LEASE_TTL} seconds, ID: {lease.id}")

    try:
        while True:
            data = json.dumps({"status": "alive", "ts": time.time()})
            etcd.put(HEARTBEAT_KEY, data, lease=lease) # put heartbeat key with lease attached
            print("Heartbeat sent")
            lease.refresh() # refresh the lease before it expires to keep key alive
            time.sleep(LEASE_TTL / 2)   # sleep less than TTL to ensure refresh before expiry
    except KeyboardInterrupt:
        print("Stopping heartbeat")
        lease.revoke()  # optional: revoke lease and delete key immediately

if __name__ == "__main__":
    send_heartbeat()
```

The following code runs on the server side to monitor the heartbeat signals from all nodes in a system, allowing it to detect active or failed nodes in real time.

```
import etcd3    # version 0.12.0

etcd = etcd3.client(host='etcd-0', port=2379)
KEY_PREFIX = "/monitor/heartbeat/"

def on_heartbeat_event(watch_response):
    for event in watch_response.events:
        key = event.key.decode('utf-8')
        node_id = event.key.decode('utf-8').split('/')[-1]

        if isinstance(event, etcd3.events.PutEvent):
            value = event.value.decode('utf-8')
            print(f"[+] Node {node_id} alive -> {value}")
        elif isinstance(event, etcd3.events.DeleteEvent):
            print(f"[-] Node {node_id} dead (key removed)")

def monitor_heartbeats():
    watch_id = etcd.add_watch_prefix_callback(KEY_PREFIX, on_heartbeat_event)

    try:
        while True:
            pass # doing main function
    except KeyboardInterrupt:
        print("Stopping watcher...")
        etcd.cancel_watch(watch_id)

if __name__ == "__main__":
    monitor_heartbeats()
```

2. EXERCISES

In this exercise, students are asked to integrate previous monitoring tools with etcd. In particular, suppose this tool has to monitor n nodes where n where n corresponds to the number of group members, along with one centralized server. Each monitoring agent operates based on the configuration defined by the server and periodically sends heartbeat signals. In particular, the flow for integrating etcd is:

- Each node collects monitoring data and sends heartbeat signals to etcd using the key format "/monitor/heartbeat/<HOSTNAME>".
- Each node receives and updates its configuration in real time through the key format "/monitor/config/<HOSTNAME>". Since configuration is accessed by multiple threads within the monitor agent, **synchronization** mechanisms must be taken into account, e.g. employing reader-writer lock.
- The centralized server aggregates monitoring data and heartbeat statuses from all nodes to maintain a global view of system health.

Note: The configuration should follow the format shown below, though students may design their own structure. However, the custom format should be backward compatible, as students are required to demonstrate dynamic configuration update behavior.

```
{
    "interval": 10,
    "metrics": ["cpu", "memory", "disk read", "disk write", "net in", "net out"]
}
```

**** Submission:** This course does not require individual submissions. Students are encouraged to work collaboratively in groups to complete the assigned tasks. However, each group will be required to present and demonstrate all aspects of their work during the final class session. The final presentation will be purely an engineering demonstration and no written report is required.