Lab 3 Report

Anush Choudhary

November 8, 2025

GitHub Repository: https://github.com/Anush2712/Distributed-systems-Lab-3.git

1 Introduction

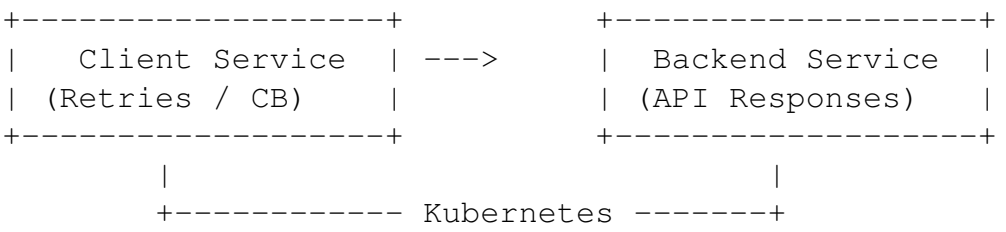
Modern distributed applications rely on microservices that communicate over a network, and these communications may fail due to latency, timeout, overload, or service crashes. To enhance fault tolerance, reliability mechanisms such as **retry logic**, **circuit breakers**, and **chaos testing** are utilized. This lab demonstrates reliability techniques in a Kubernetes-based microservices setup.

2 System Architecture

The project consists of:

- Backend Service (REST API returning JSON)
- Client Service consuming the backend service
- Kubernetes deployments managing both services
- Circuit breaker + retry logic inside client
- Chaos testing to simulate backend failure

Architecture Diagram



3 Kubernetes Setup

3.1 Backend Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: my-backend:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
  - port: 5000
    targetPort: 5000
```

Explanation: This is a Kubernetes manifest of our microservice backend. It describes one replica of our backend container and ports it out to 5000 to allow other services access it. Fault tolerance The Deployment ensures that Kubernetes automatically restarts the backend pod in case it fails. Service component provides the backend to be accessible within the cluster by a stable internal name rather than a dynamic IP address.

3.2 Client Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
spec:
  replicas: 1
  selector:
    matchLabels:
      app: client
  template:
```

```
    metadata:
      labels:
        app: client
    spec:
      containers:
      - name: client
        image: my-client:latest
```

Explanation: This topology implements the client microservice. It is also a single-pod Kubernetes Deployment. As the client does not accept traffic, but just consumes the backend, then it does not require a Service. Kubernetes also makes sure that the client pod remains running and recovers it in case of a failure. The client code will constantly make calls to the backend to exhibit the behavior of a retry and circuit-breaker.

4 Code Implementation

4.1 Backend Service

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route("/data")
def get_data():
    return jsonify({"message": "Backend␣Running␣   "})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Explanation: Simple Flask backend API is run by this Python script. It opens one of the endpoints with a response of a small JSON message indicating that the service is alive. This is a real microservice that will enable us to test how the client will behave in case of normal and failure conditions. Running this in Kubernetes can be used to simulate a microservice architecture in reality.

4.2 Client Service with Circuit Breaker + Retry

```
import requests, time

FAIL_THRESHOLD = 3
OPEN_TIME = 10

fail_count = 0
circuit_open = False
open_timestamp = 0

def call_backend():
    global fail_count, circuit_open, open_timestamp
```

```
    if circuit_open:
        if time.time() - open_timestamp < OPEN_TIME:
            print("Circuit open... skipping call")
            return
        else:
            circuit_open = False
            fail_count = 0
            print("Retrying backend...")

    try:
        r = requests.get("http://backend:5000/data", timeout=2)
        print("Success:", r.json())
        fail_count = 0

    except requests.exceptions.RequestException:
        fail_count += 1
        print(f"Backend failed | Fail count = {fail_count}")

        if fail_count >= FAIL_THRESHOLD:
            circuit_open = True
            open_timestamp = time.time()
            print("Circuit Breaker OPEN")

while True:
    call_backend()
    time.sleep(3)
```

Explanation: The client keeps attempting to make calls to the backend service and logs the reply. A failure is counted on the client in case the backend is unreachable. Upon three consecutive failures, it goes into a circuit breaker state during which it temporarily ceases to make requests. This helps in avoiding overloading the backend in the event of a downed backend. The client then attempts again and recovers normal operation provided the backend has recovered, otherwise, the client then attempts to restart the process again after a cool-down period. The patterns of microservice resilience are illustrated in this code.

5 Experiments & Outputs

5.1 Normal Operation

Expected Log Output:

```
Success: {'message': 'Backend Running '}
Success: {'message': 'Backend Running '}
```

5.2 Chaos Test – Backend Failure

Simulate backend crash:

```
kubectl delete pod -l app=backend
```

Explanation: This command will purposely remove the backend pod to imitate the failure of a service. Kubernetes will restart the pod automatically, yet when it is unavailable, the client service should keep running, and it has to gracefully handle the outage. This test will enable us to see the action of retry and circuit-breaker behavior.

Observed Output:

```
Backend failed | Fail count = 1
Backend failed | Fail count = 2
Backend failed | Fail count = 3
Circuit Breaker OPEN
Circuit open... skipping call
Circuit open... skipping call
```

5.3 Service Recovery

Restart backend:

```
kubectl rollout restart deployment backend
```

Output:

```
Retrying backend...
Success: {'message': 'Backend Running '}
```

Explanation: This command deletes the pod that provides the backend service on purpose in order to simulate a service failure. The pod will be automatically recreated by Kubernetes, but with it being down, the client service will need to keep working and deal with the failure gracefully. The test gives us a chance to test the behavior of retry and circuit-breaker.

Under healthy conditions of the backend, the client manages to be fed and display successful messages. The circle circuit breaker logic and attempt to restart do not take place since there are no mistakes. This assures that the system is functional in the normal operating conditions.

On failure of the backend, the client begins to make repeated attempts. Once the circuit breaker has tried three unsuccessful attempts, it opens thus disallowing further calls after some time. This is a stress-free behavior showing robust failure processing. After the backend is brought online, the client resumes normally.

6 Observations

Mechanism	Purpose	Result
Retries	Handle transient failures	Works, but adds latency
Circuit Breaker	Avoid hitting dead service	Enabled graceful fallback
Chaos Testing	Validate resilience	Client recovered automatically

7 Conclusion

This lab demonstrated reliability patterns essential to microservices:

- Circuit breakers stop cascading failures.
- Retry logic helps handle temporary glitches.
- Chaos testing ensures real-world robustness.
- Kubernetes restarts failed services automatically.

The implemented architecture successfully tolerated backend crashes and recovered gracefully, proving the effectiveness of resilience design in microservice systems.