COMP41720 Distributed Systems: Architectural Principles LAB 3: Designing for Resilience and Observability

Duration: Week 7-8 (culminating in a lab submission by end of Week 8)

1. Lab Overview This lab focuses on equipping you with the **fundamental architectural principles and practical techniques** for designing resilient distributed systems. You will gain hands-on experience implementing various **resilience patterns** to enable applications to gracefully handle failures, which are inevitable in distributed environments. Furthermore, you will use **chaos engineering** principles to proactively test your system's fault tolerance, moving beyond merely getting the code to work, to understanding *why* certain architectural decisions lead to more robust systems.

2. Learning Objectives By the end of this lab, you will be able to:

- **Understand common failure modes** in distributed systems, such as network issues, process crashes, and slow responses.
- **Implement key resilience patterns**, including Circuit Breakers, Retries, and Backoff strategies, within a distributed application.
- **Apply chaos engineering tools** to simulate failures (e.g., network partitions, node shutdowns) in a controlled environment.
- **Analyze and compare system behavior** with and without implemented resilience patterns, identifying the practical benefits and trade-offs in terms of system availability, performance, and fault tolerance.
- **Reason architecturally** about designing systems that can gracefully handle partial failures and ensure reliability.

3. Introduction & Context In distributed systems, failures are a given. As the course emphasizes, "it's not a question of if something will eventually break, but when". Proactive design is crucial to build systems that are not just scalable but also resilient. This involves anticipating failure modes and incorporating patterns that allow the system to degrade gracefully or recover automatically, rather than crashing entirely.

This lab moves beyond simple communication and data management to address the **"hard parts" of software architecture** – dealing with unpredictable failures and ensuring continued operation. You will apply the principle of **trade-off analysis** by observing how different resilience patterns impact your system's properties like availability and performance, reinforcing the idea that there are "no best practices," only the "least worst combination of trade-offs" for a given scenario.

4. Tools & Environment To effectively demonstrate resilience, your setup should involve at least two communicating components deployed in a distributed manner.

- **Client Application Language** (Choose ONE):
  - Python, Java (modern versions recommended, not Java 8 as per feedback), Node.js, Go, or any language you are comfortable with that has good support for client-side resilience patterns.
- **Deployment Environment:**
  - **Kubernetes:** Recommended for deploying your distributed application, as it provides an excellent platform for simulating failures and managing

distributed components. You can use a local Kubernetes cluster (e.g., Minikube, Kind, Docker Desktop's Kubernetes) or a cloud-based one.
    - **Docker:** For containerizing your application components, which is essential for deployment on Kubernetes.
- **Resilience Libraries/Frameworks:**
    - **Circuit Breaker:** Research and choose a library that provides circuit breaker functionality for your chosen language (e.g., Resilience4j for Java, Tenacity for Python, Polly for .NET, or implement a basic one conceptually).
    - **Retry & Backoff:** Most modern HTTP clients or language SDKs offer built-in retry mechanisms, often with configurable backoff strategies (e.g., Exponential Backoff, Jitter as seen in AWS SDKs).
- **Chaos Engineering Tool:**
    - **Chaos Toolkit:** A powerful, open-source tool for defining and executing chaos experiments.
    - **Chaos Monkey / Gremlin:** (Optional, if you have access or wish to explore).

5. Lab Tasks & Experiments

Part A: Setup & Baseline Distributed Application

1. **Simple Distributed Application:**
    - **Design and implement a minimal distributed application** consisting of at least two services: a `ClientService` and a `BackendService`. The `ClientService` should make synchronous requests (e.g., HTTP REST calls, gRPC) to the `BackendService`. You can build on your Lab 1 RESTful services if appropriate, or create a new simple interaction.
    - The `BackendService` should have a basic endpoint that occasionally introduces **simulated delays or failures** (e.g., HTTP 500 errors, network timeouts) to allow for testing resilience patterns.
2. **Kubernetes Deployment:**
    - **Containerize** both your `ClientService` and `BackendService` using Docker.
    - **Deploy your application on a Kubernetes cluster.** Ensure both services are accessible and can communicate. Document your Kubernetes manifests (e.g., Deployment, Service YAML files).
3. **Baseline Test:**
    - Perform initial tests to confirm your application functions correctly under normal conditions.
    - **Observe the impact when the `BackendService` fails or becomes slow** *without* any resilience patterns in place (e.g., client blocking, timeouts, errors propagating directly). Document these observations.

Part B: Implementing Resilience Patterns

1. **Circuit Breaker Implementation:**
    - **Integrate a Circuit Breaker pattern into your `ClientService`** for calls made to the `BackendService`.
    - Configure the circuit breaker with parameters such as failure threshold, a duration to wait before attempting to half-open, and a maximum number of concurrent requests allowed when half-open.
    - **Experiment:**

        - Trigger enough failures (from `BackendService`) to cause the circuit breaker to **open**. Observe the `ClientService`'s behavior (e.g., fast-failing, returning a fallback response, not even attempting the call).
        - After the configured `wait` duration, observe the circuit breaker attempting to **half-open** and allowing a limited number of requests through.
        - Verify the circuit **closes** if successful calls resume, or re-opens if failures persist.
        - **Document observations and analyze the trade-offs**: How does the circuit breaker improve availability and protect the `ClientService`? What are the implications for data freshness or user experience?
2. **Retries with Exponential Backoff and Jitter:**
    - **Implement retry logic with an exponential backoff strategy and jitter** within your `ClientService` for calls to the `BackendService`. This is used for transient failures that might resolve themselves.
    - **Experiment:**
        - Configure the `BackendService` to return **transient failures** (e.g., HTTP 429 Too Many Requests, or intermittent 500s).
        - Observe the `ClientService` automatically retrying the requests with increasing delays.
        - Demonstrate how jitter helps prevent a "thundering herd" problem of synchronized retries.
        - **Document observations and analyze the trade-offs**: When is this pattern appropriate versus a circuit breaker? What are the potential impacts on backend load, latency, and system stability?

Part C: Chaos Engineering Experiment

1. **Chaos Engineering Setup:**
    - Choose a chaos engineering tool (e.g., Chaos Toolkit).
    - **Define a chaos experiment** targeting your Kubernetes-deployed `BackendService`.
    - **Recommended Experiment:** Simulate a **network partition** that prevents communication between your `ClientService` and `BackendService`, or a **node failure/shutdown** of the node running your `BackendService` pod.
2. **Execute & Observe:**
    - Execute the chaos experiment **while your `ClientService` is active**.
    - **Observe the system's behavior** in detail, especially how your implemented resilience patterns react.
    - **Collect metrics or logs** from both services and the Kubernetes cluster to support your observations (e.g., circuit breaker state, retry counts, error rates, pod status).
3. **Analysis & Justification:**
    - **Compare the observed behavior** to what you would expect *without* the resilience patterns (refer to your baseline test).
    - **Provide a detailed architectural analysis** of how the resilience patterns enabled your system to continue functioning (or fail gracefully) despite the injected fault.
    - **Relate your findings back to architectural characteristics** like availability, fault tolerance, and responsiveness. Justify *why* these patterns are crucial for

robust distributed system design, considering their costs (e.g., complexity, potential for increased latency in some cases).

6. Deliverables

1. **Lab Report (PDF format):**
    - **Introduction:** Briefly state the lab's purpose, your chosen technologies, and the application's basic functionality.
    - **Setup & Configuration:** Detail your application components, Docker images, and Kubernetes deployment (including YAML manifests). Provide a clear diagram of your deployed system.
    - **Resilience Experiments (Parts B & C):**
        - For each experiment (Circuit Breaker, Retries, Chaos Engineering):
            - Describe the **specific configuration** of the pattern/tool (e.g., circuit breaker thresholds, retry logic parameters, chaos experiment YAML).
            - **Document your observations** vividly (e.g., client service logs, service behavior during failure, recovery process). Use screenshots, log snippets, or charts as evidence.
            - **Crucially, provide a detailed analysis of the architectural trade-offs.** Justify *why* you would choose these specific resilience strategies for different failure types or business requirements. Link your observations directly to core distributed systems principles like the CAP Theorem, availability, performance, and fault tolerance.
    - **Conclusion:** Summarize your key learnings about designing for resilience, any unexpected observations, and the overall impact of applying these architectural patterns.
2. **Source Code Repository Link:** Provide a link to your code repository (e.g., GitHub, GitLab) containing all application code, Dockerfiles, and Kubernetes manifests used for your experiments.

7. Assessment Criteria Your lab will be assessed based on the following:

- **Correctness and Completeness of Implementation (40%):**
    - Successful setup and deployment of the distributed application on Kubernetes.
    - Accurate implementation of Circuit Breaker and Retry/Backoff patterns.
    - Successful execution and demonstration of the chaos engineering experiment.
- **Depth of Analysis and Justification (40%):**
    - **Clear and insightful analysis of architectural trade-offs** for each resilience pattern and chaos experiment.
    - Strong justifications for design choices, linking observations to distributed systems principles (e.g., graceful degradation, system stability under stress).
    - Effective use of observations and evidence to support analytical claims.
- **Clarity and Organization of Report (10%):**
    - Well-structured, concise, and easy-to-understand report.
    - Effective use of diagrams and visual aids.
- **Code Quality and Readability (10%):**
    - Clean, well-commented code that is easy to understand and reproduce.

8. Tips for Success

- **Start Early:** Setting up Kubernetes and experimenting with chaos tools can have a learning curve. Give yourself ample time for setup and troubleshooting.
- **Read Documentation:** Familiarize yourself with the documentation for your chosen resilience libraries and chaos engineering tool.
- **Focus on the "Why":** Beyond getting the code to work, ensure you deeply understand *why* certain resilience patterns are chosen, *why* certain behaviors occur during failures, and *why* different architectural choices yield different trade-offs. This is central to the course.
- **Incremental Builds:** Implement and test each resilience pattern individually before combining them or introducing chaos.
- **Utilize Support:** Teaching Assistants (TAs) and instructors are available for questions during lab sessions and office hours. Do not hesitate to ask for clarification on vague instructions or technical challenges. We are working to provide clearer instructions and better support based on past feedback.
