Architecture of Large-Scale Systems

By Dr. Taghinezhad

Mail:
a0taghinezhad@gmail.com

https://ataghinezhad.github.io/

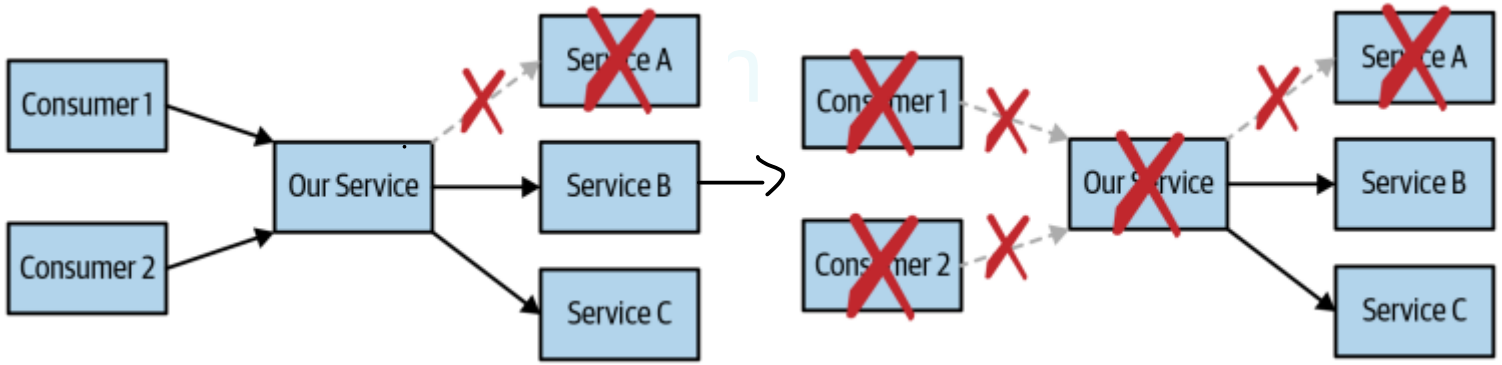
Chapter 5:Dealing with Service Failures

Introduction to Service Failures in Microservice Systems

- **Microservice architectures** introduce several interdependent services, which makes the system vulnerable to service failures.
- **Challenge:** The more services involved, the higher the chances of failure, and any failure could cascade to dependent services, potentially destabilizing the entire application.
- **Objective:** To develop **strategies** to mitigate cascading failures and ensure application stability.

Cascading Service Failures

- **Cascading failure** occurs when the failure of one service triggers failures in dependent services, propagating throughout the system.
- **Example Scenario of a Cascading Failure (Figure):**
  - **Our Service** depends on **Service A, Service B, and Service C**.
    - **Consumer 1 and Consumer 2** depend on **Our Service**.
    - If **Service A** fails, **Our Service** may fail. This, in turn, can cause both **Consumer 1** and **Consumer 2** to fail.
  - **Impact:** A single failure can disrupt multiple components if left unchecked.
- **Key Problem:**
  - **High dependency** between services can make the entire system fragile.
  - A small fault can ripple through the system, affecting **business processes and customer experiences**.



Techniques to Handle Service Failures

- **Developing Predictable, Understandable, and Reasonable Responses** Aim to respond predictably to failures, without propagating the chaos upward in the service chain.
- **Predictable Response**
  - **Definition:**
    A predictable response is a **planned and controlled outcome** for specific failure conditions.
  - **Example of Predictable Responses:**
    •**For a division by zero request:**
      •Predictable Response: "Error: Invalid operation"
      •**Unpredictable Response:** Returning garbage values (e.g., 50000000000).
    •When **dependencies fail**, your service should generate **planned error messages**, not unpredictable outputs.

Why Predictability is Critical:

- Prevents cascading failures by ensuring upstream services handle your service gracefully.
- Avoids injecting **inconsistent or invalid data** into business processes, maintaining data integrity.
- A predictable error message (e.g., “Service Unavailable”) **informs the client** of the issue, allowing it to take appropriate actions.

Understandable Response

•**Definition:**

An understandable response follows the **agreed-upon contract or format** defined by your API.

•**API Contract:**

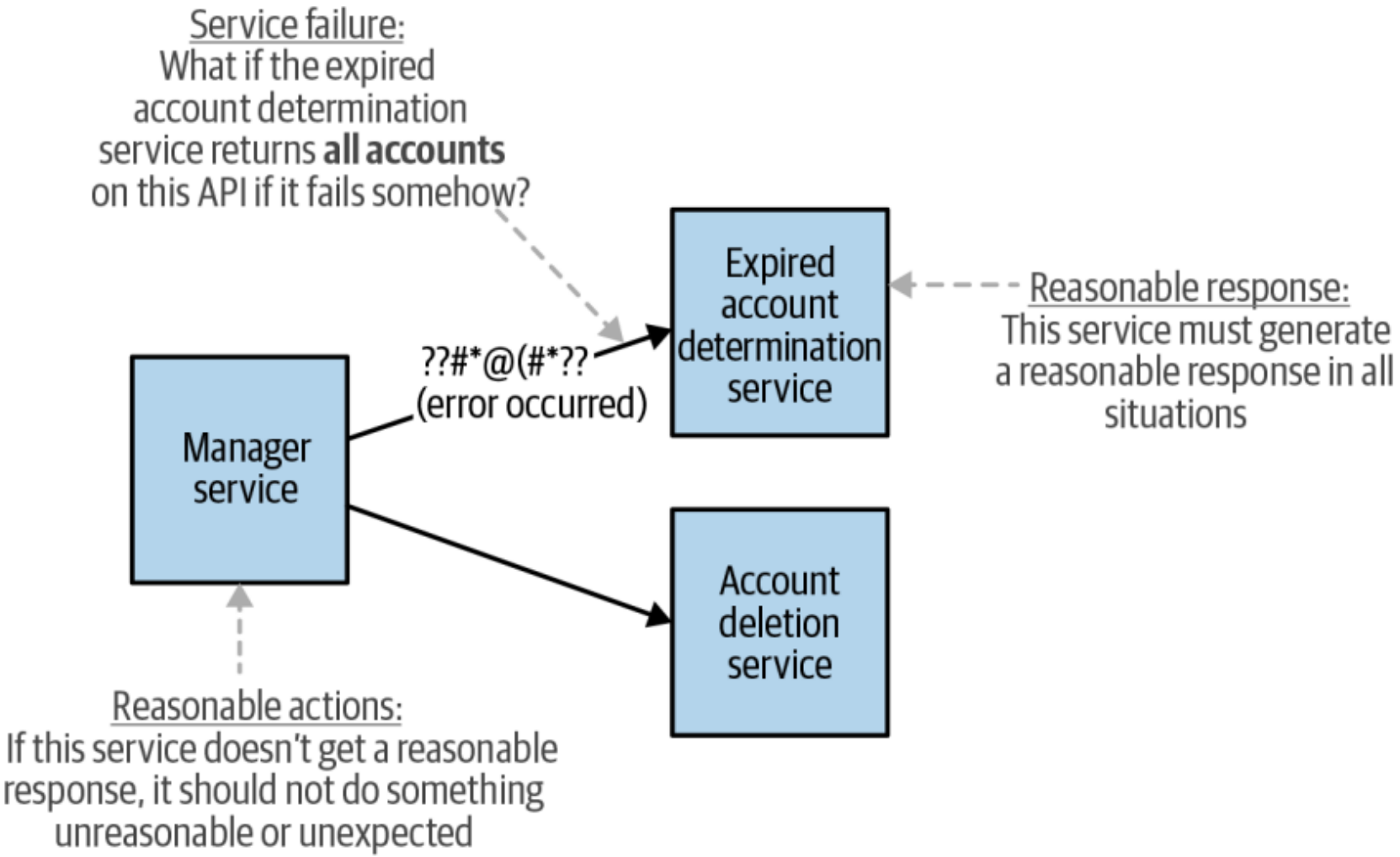
•Clearly defines what responses are expected, even in failure scenarios.

•**Example:** If a service fails to deliver an output, it could return "Error: Unable to process request“ rather than breaking the API contract.

- **Best Practices:**
- Do not violate the API contract due to misbehaving dependencies.
- Ensure your API supports **graceful degradation** by providing fallback responses when needed.

Understandable Response

For example, a service might call an "expired account" service to retrieve a list of accounts ready for deletion, then proceed to delete each account.



Example

- **Reasonable Response**
- **Definition:**
  A reasonable response accurately reflects the **situation of the service**.

Examples of Reasonable Responses:

•If asked to perform **"3 + 5"**:

  •Reasonable Response: **"8"**

•If a dependency fails:

  •Reasonable Response: **"Sorry, I couldn’t complete your request. Please try again later."**

Unreasonable Response:

•Returning "red" or other irrelevant data for a mathematical operation request.

Why Reasonable Responses Matter:

- Prevents damaging side effects:
  - Example: A service asked for **expired accounts** should return "None" if it encounters an error, not a list of **all accounts in the system**.
  - **Impact of an Unreasonable Response:**
    - If all accounts were returned mistakenly, a downstream service might delete all user accounts, causing catastrophic damage.

Techniques for Handling Dependency Failures

1.Graceful Degradation:

- If a dependent service fails, **degrade functionality** without crashing the whole service.
- **Example:** An e-commerce website continues to display product pages, but disables the checkout feature temporarily when the payment service is down.

2.Circuit Breaker Pattern:

- Prevents continuous requests to a failing service by **cutting off communication temporarily**.
- **Example:** If Service A fails, the circuit breaker stops Service B from making further requests, preventing further cascading failures.

3.Fallback Mechanisms:

- Provide **alternative responses** when dependencies fail.
- **Example:** If a weather service fails, return "Weather data is currently unavailable" instead of crashing the service.

4.Timeouts and Retries:

- Implement **timeouts** to avoid waiting indefinitely for a response from a failing service.
- Use **limited retries** to handle temporary failures.

Concept	Definition	Examples	Best Practices
Cascading Failure	Failure of one service causes others to fail	Service A's failure leads to Our Service's failure	Use circuit breaker and fallback mechanisms
Predictable Response	Planned response to failure scenarios	"Service Unavailable" for downtime	Avoid unpredictable garbage responses
Understandable Response	Follows agreed API contract	"Error: Invalid request" for bad input	Ensure API always returns meaningful errors
Reasonable Response	Reflects the service's real state	"Please try again later" during failure	Avoid returning irrelevant data (e.g., "red")
Graceful Degradation	Maintain partial functionality	Product pages shown without checkout	Degrade service functionality selectively
Circuit Breaker Pattern	Temporarily cut off requests to a failing service	Stop retries after a set limit	Prevent cascading failures
Timeouts and Retries	Limit wait time for responses	Retry a few times after failure	Avoid infinite wait loops

Determining and Handling Failures in Microservice Architecture

- **Introduction to Failure Detection**
- In a microservice-based system, determining when a dependency is failing is crucial to prevent cascading service failures and ensure application stability. However, different types of failures require different detection techniques, and each type of failure demands a tailored response.

Failure Modes in Microservices

- Failures can manifest in several ways, ordered from easiest to hardest to detect:

1. **Garbled Response**
    1. The response is in an unrecognizable or corrupted format (e.g., syntax errors).
    2. Immediate detection is possible since the data is unreadable.
2. **Response Indicated a Fatal Error**
    1. A response contains information about a critical failure within the service (e.g., invalid request or internal processing error).
    2. This failure typically occurs within the service logic, not the communication layer.
3. **Unexpected Results**
    1. The operation executes successfully, but the data returned is incorrect or unexpected.

Failure Modes in Microservices

4. Result Out of Expected Bounds

1. The data is in the correct format but falls outside the reasonable or expected range.
2. Example: If the number of days since January 1 is returned as 843, it's out of a plausible range.

5. No Response

- The request is sent, but no response is received.
- Possible causes: network issues, service crash, or service outage.

6. Slow Response

1. The response arrives but takes longer than expected, possibly due to network congestion or resource exhaustion.
2. It is the hardest to handle since timing can be inconsistent.

Handling Detected Failures

- Different failure modes call for specific strategies to maintain stability and usability.
  - **3.1 Graceful Degradation:** If a service dependency fails, perform partial operations to maintain functionality.
    - **Example:** Display cached or limited data if a backend service is unavailable.
  - **3.2 Graceful Backoff:** When failure renders continued operation impossible, find an alternative, less impactful way to respond.
    - **Example:** Redirect users to a maintenance page with relevant messaging instead of a generic error.
  - **3.3 Fail as Early as Possible:** If a request is certain to fail, halt further processing immediately to save resources.
    - **Example:** Check for invalid inputs (like division by zero) at the earliest step to prevent complex failures later.

Handling Detected Failures

- **Why Fail Early?**
  1. **Resource Conservation:** Prevent unnecessary operations and API calls that consume resources.
  2. **Improved Responsiveness:** Immediate failure lets users or calling services react faster.
  3. **Simplified Error Handling:** Early detection makes issues easier to diagnose and resolve.

Handling Detected Failures (Cont.)

- **4. Customer-Caused Errors and Service Limits**
  - Validate user input early to prevent resource-intensive operations based on invalid requests.
    - **Example:** A service that retrieves accounts should cap requests at a reasonable limit (e.g., 5,000 accounts). If the request exceeds the limit, return an error immediately.
- **5. Service Limits**
  - Define service limitations clearly in your API contract.
    - **Example:** If the service can handle only a limited number of items per request, this limit should be documented and enforced.

Handling Detected Failures (Cont.)

- **5. Detecting and Responding to Slow Responses**
- **Challenges:** Determining how slow is “too slow” can be difficult.
- **Mitigation Strategies:**
  - Use **timeouts** and **circuit breakers** to prevent endless waiting.
  - Monitor response times to identify bottlenecks and adjust policies proactively.

Conclusion

- Handling service failures effectively requires:
  - **Proactive failure detection** using a variety of techniques.
  - **Appropriate error handling strategies** such as graceful degradation, backoff, and early failure.
  - **Communication and limits** defined in service contracts to prevent customer-caused errors.
  - By combining these strategies, a microservice-based application can maintain stability even in the face of failures, ensuring predictable, understandable, and reasonable service behavior.