Robust Microservices

Objectives

Key objectives of this chapter:

- Techniques and patterns for making Microservices Architecture robust

1.1 What Can Make a Microservices Architecture Brittle?

- Distributed transactions
- Multiple service interactions
- Neglecting to introduce mechanisms to
  - Support fault-tolerance
  - Contain failure propagation
  - Establish system-wide monitoring and actionable alerting
  - Support automation
- Generally, not designing for Test and Failure

1.2 Making it Resilient – Mechanisms

- The Anti-Corruption Layer concept of DDD aids in designing for failure by creating a layer that contains failures

- Use other applicable design patterns and stick to basic interaction hygiene:
  - Asynchronous communication, the circuit breaker pattern, configured time-outs, etc.
- System fault-tolerance / resilience can be easily achieved by scaling services
  - A failed service can be quickly replaced by a newly provisioned service instance
    - Services must be stateless

1.3 Techniques and Patterns for Making Your Microservices Robust

- Common techniques and patterns in this area include:
  - Choosing between fail fast / quiescent mode (hibernating / suspending the in-flight transactions in hope for service recovery)
  - Timeouts / Retries
  - Bulkhead Pattern
  - The Circuit Breaker Pattern

1.4 Fail Fast or Quiesce?

- Depending on the nature of the client-service interaction, you need to define criteria by which you can decide between
  - Failing Fast
    - So that you can minimize the potential negative impact of a failed transaction

- Quiesce
  - Stop receiving new requests, pause / put on hold the execution of in-flight transactions in one way or another in hope to gain control back when you can unquiesce the transactions put on hold or let in new connections
  - Quiescing is a form of graceful service degradation
- When undecided - talk to your business users

1.5 Synchronous Communication Timeouts / Retries

- Configurable timeouts help establish the client's tolerable interval of waiting for server response
  - Usually it involves the tweaking of the SOCKET TIMEOUT parameter of the TCP/IP or HTTP client library used to make the call
  - You can document this metric in the service-level SLA
- Retries
  - Different delays and number of retries can be used depending on the call's nature
    - You can use time delays between retries in seconds modeled after the Fibonacci series (1,1,2,3,5,8 …) or such like schemes
    - Randomly chosen values from an interval, e.g. between 2 and 5 seconds
  - Retries can be combined with the timeout parameter

Notes:

Here is a snippet of Java client socket code setting the socket timeout to 5 seconds (5,000 milliseconds):

```
Socket socket = new Socket(serverName, port);
```

```
socket.setSoTimeout(5*1000);
// use the socket to connect to the server
```

If the timeout expires, a java.net.SocketTimeoutException is raised; however, the Socket object still remains bound to the server and ready to accept another client call.

1.6 Asynchronous Communication Timeouts / Retries

- This technique is well suited for the UI layer in your app's web pages
- It works as follows:
  - JavaScript code running in the web page makes a call to the server and designates an AJAX-based asynchronous communication call-back configured with the timeout interval
    - Some accurate progress meter can be shown while the user is waiting or working in other parts of the UI
- If no response is received within the timeout interval, the user is presented with some sort of “Please try again later ...” message
- **Note:** Asynchronous calls must be idempotent on retries

1.7 In-Class Discussion

- What, in your opinion, are some of the arguments in favor / against of fail fast vs quiesce?
- Would you consider timeouts and retries as some form of quiescing?
- Who would have the final say in deciding which route to take?

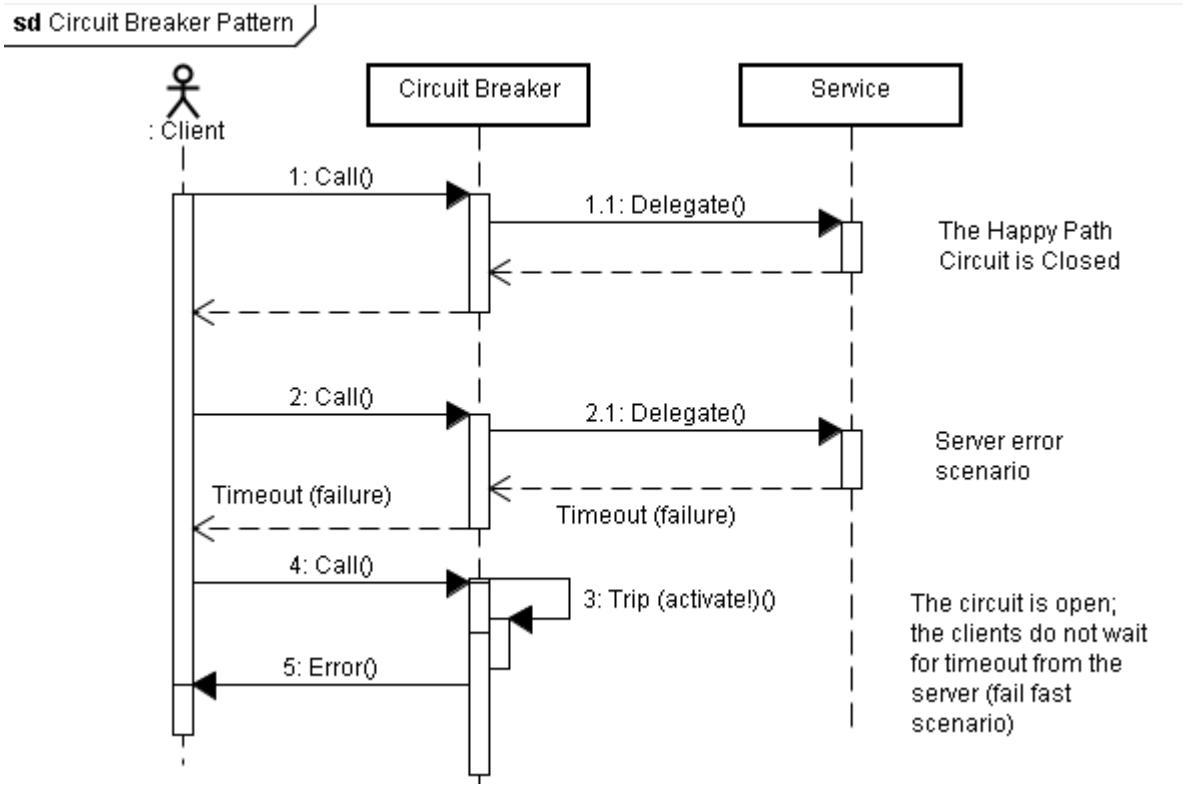
1.8 The Circuit Breaker Pattern

- The circuit breaker (CB) pattern prevents cascading failures when one failing system may triggers a sequence of failures of dependent services (the domino effect)
- At heart of the pattern is the CB component that acts as a “smart” proxy for the downstream service preventing clients from being affected by the service failures
- On the happy path, the CB component transparently relays client's calls to the downstream service and forwards back the response
- Should the “smart” CB detect a problem connecting to the downstream service, it immediately or after a number of (failed) calls returns control to the client with an error code and opens the “circuit” – the client is now disconnected from the service
- CB proceeds to poll the failed service for health status using some sort of configurable policy (the number of retries, etc.), closing the circuit when the service is back up again
- CB can also be re-purposed to guard downstream services from being overwhelmed with client calls, throwing a *Server Busy* type of exception (similar to 5xx HTTP error codes) on behalf of the downstream services

Notes:

Circuit breaker is an electrical term. According to Wikipedia, *"A circuit breaker is an automatically operated electrical switch designed to protect an electrical circuit from damage caused by excess current from an overload or short circuit. Its basic function is to interrupt current flow after a fault is detected.* ***Unlike a fuse,*** *which operates once and then must be replaced, a circuit breaker can be reset (either manually or automatically) to resume normal operation."*

1.9 The Circuit Breaker Pattern Diagram



1.10 The Bulkhead Pattern

- Close to the Anti-Corruption Layer concept from DDD
- It is all about strong microservices isolation from one another using a

variety of implementations so that the failed microservice is contained and not allowed to impact the rest of the services

- For example, the load balancer that checks the health of the clustered microservices it fronts can stop forwarding client requests to the server identified as failed effectively taking it out of the pool of available servers
- Creating local micro databases is another technique that supports the Bulkhead pattern
- Asynchronous communication style offers a natural process isolation through decoupling
- In some cases, you may want to explore the possibility of creating separate connection pools for different clients which may further help with application scalability, fault tolerance, and application isolation

1.11 Factor IX of the 12 App Methodology

- IX. Disposability
  - Maximize robustness with fast startup and graceful shutdown (via containers and small deployment footprint)
- Microservices deployed in containers fully embrace this factor and can be recycled within seconds without negatively affecting user experience

1.12 Feature Enablement

- Isolated service failures should not shut down your business
- Your application should allow your clients to continue using the unaffected business features
  - Your app design should allow for feature enablement / disablement without code re-deployment via centralized management / configuration capability

- This feature can be viewed as manual / on-demand circuit breaker

1.13 Designing for Test and Failure

- Adopt the maxim: service failure is a norm
- Have a clear picture of the impact service failures would have on your Microservices Architecture
- Microservices interactions must be thoroughly tested in production-like environments
  - Use service mockups, stubs, emulators, where needed
- Designing for Test is supported by the Test-Driven Development (TDD) practice that aids in creating better application design (e.g. test cases help identify and test service interfaces and client interactions)
- QA plans should include test cases for recoverable and unrecoverable service / system failures; client impact should be gauged
  - Service degradation decisions must be made here: fail fast, or quiesce, or use cached results, or re-route requests around faulty parts of the system
    - Engage business for assessing the client impact

Notes:

Fail fast is like hanging up on clients; quiescing is like placing them on hold.

Quiescing can be naturally implemented using reactive programming style.

1.14 Making Microservices Testable

- Traditional end-to-end integration testing is not quite efficient for microservices

- Why?

- The common practice is to establish interaction contracts (pacts) between clients and service providers
  - Pacts (contracts) set expectations of the client that should not be broken by the service
    - **Note:** In SOAP services contracts are in the form of WSDL files
- This approach helps with testing services and clients in isolation

1.15 Test for Failure

- You need to know how your application would behave in the face of failures
- So you need tooling for simulating service disruption in order to establish the level of resiliency and tolerance for failure of your application
  - For example, Netflix uses a specialized tool called Chaos Monkey that induces failures of services to test applications' resilience [https://github.com/Netflix/chaosmonkey]

Notes:

Netflix runs its business on Amazon Web Services (AWS) infrastructure and uses microservices architecture style for designing and building their applications. To test their (micro) services, they use a tool called Chaos Monkey that deliberately causes failures in the system to see how resilient the system is to such failures.

1.16 Continuous Testing and Integration

- Development of discrete software components must go hand-in-hand with the development and application of appropriate unit tests as prescribed by the Test-Driven Development (TDD) process

- Integration of software components must start as early as possible (even though the work on components may not yet been fully completed) and conducted frequently (sometimes, several times a day)
  - This process is known as the Continuous Integration (CI) agile practice which helps with catching integration problems early

1.17 Continuous Release and Deployment

- To support reliable continuous releases, the deployment process must be automated; any failed deployments must be rolled back in its entirety in an atomic operation without affecting applications currently running in production
- Deployment parameters, such as average deployment time, size of the deployment bundle, etc., must be recorded and kept for reference

1.18 SLAs

- A Service-Level Agreements (SLA) is a commitment of a service provider to its clients for the promised performance
- An SLA lists a number of metrics (that you measure at run-time), including
  - Uptime:
    - Used for measuring service availability, time to recover from an outage, etc.
  - Number of supported concurrent clients
  - Average response times
    - Sometimes expressed as percentages, e.g. 90% of calls processed under 2 seconds
  - Throughput

- e.g. TPS

- SLAs metrics are compared with the real-time values during service monitoring

1.19 Where and What to Monitor

- Microservices monitoring should be done in various areas:
  - Infrastructure
    - Number of active containers, health of the availability zone (when in cloud), etc.,
  - Host (container, machine)
    - CPU, I/O, uptime
  - Service
    - Response time per application interface
    - Used heap size
    - **Note**: Usually, the uptime of a service deployed in a container is the same as that of the container

1.20 Logging and Monitoring

- Detect failures early by having real-time visibility into the distributed systems' health
- When in production, systems of distributed applications must be placed under exhaustive real-time monitoring control and logging
  - Log everything; monitor relentlessly against service SLAs;
  - Actionable alerting must be put in place
    - Alerts should be based on such run-time metrics as CPU utilization,

number of requests, number of exceptions/errors, response time, etc.

- Those alerts can be used to trigger scaling out / in actions

- Complete postmortem analysis should be possible though logging
  - Log messages should assist in reconstructing the timeline of the system impacting events
  - Unsubstantiated finger pointing, if occurs at the postmortem meeting, must be treated as a “smell” of the overall design failure

- **Notes:**

Ideally, log messages should also help business to see, in real time, the dynamics of KPIs, like detecting trends; counting new customers acquired, clients making opt-out decisions, etc.

1.21 Summary

- In this module we reviewed techniques and patterns for making Microservices Architecture-based solutions robust and resilient to failures