Characterizing Retry Policies for Microservice Applications

Alan Song (PRIMES CS)

Mentor: Lisa (Yueying) Li
What are Microservices?
These applications need a lot of fault tolerance!
Fault Tolerance Measures

Client

Server
How do retry mechanisms work?
Retry Storms
Metastable Failures

- Stable
- Vulnerable
- Metastable

Dependencies:
- Increasing Load from Stable to Vulnerable
- Trigger from Vulnerable to Metastable
- Recovery from Metastable to Stable
- Sustaining Effect from Metastable to Metastable
Retries are by far the most common sustaining effect!
Retry Policies
# gRPC retry options

The following table describes options for configuring gRPC retry policies:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxAttempts</td>
<td>The maximum number of call attempts, including the original attempt. This value is limited by GrpcChannelOptions.MaxRetryAttempts which defaults to 5. A value is required and must be greater than 1.</td>
</tr>
<tr>
<td>InitialBackoff</td>
<td>The initial backoff delay between retry attempts. A randomized delay between 0 and the current backoff determines when the next retry attempt is made. After each attempt, the current backoff is multiplied by BackoffMultiplier. A value is required and must be greater than zero.</td>
</tr>
<tr>
<td>MaxBackoff</td>
<td>The maximum backoff places an upper limit on exponential backoff growth. A value is required and must be greater than zero.</td>
</tr>
<tr>
<td>BackoffMultiplier</td>
<td>The backoff will be multiplied by this value after each retry attempt and will increase exponentially when the multiplier is greater than 1. A value is required and must be greater than zero.</td>
</tr>
<tr>
<td>RetryableStatusCodes</td>
<td>A collection of status codes. A gRPC call that fails with a matching status will be automatically retried. For more information about status codes, see Status codes and their use in gRPC. At least one retryable status code is required.</td>
</tr>
</tbody>
</table>
Testbed Setup

DeathStarBench [hotel-reservation]

Blueprint (Anand et. al)

wrk2 workload generator
Retry Mechanism Implementation

Fixed (Power-of-d) Retry:
- Launch a constant of $d$ copies of request without delay

Fixed-interval Retry with Max Attempts
- Launch a retry with constant delay and max attempts

Exponential Backoff Policy (with random jitter)
- Launch retries with exponentially increased time interval

Implemented by us
Results: Limitations of Fixed Retry

1) Lack of reasoning about causes for timeouts

Amplification with bad retry policy (max thpt = 3000)

Max Retries = 10

SLO Latency
Results: Limitations of Fixed Retry

Max Retries = 2

Tail less amplified - higher throughput (3500)
For 3000 RPS, all requests are under SLO lat

Takeaway: the retry limit for a fixed retry policy needs to be load-aware.
Results: Exponential Retry

Takeaway: exponential retry is more resilient than fixed retry under load-increase triggers
Telemetry

Dependency graph to identify percentage of retries

Latency graph to color retried requests

Call span of different tiers of services to identify triggers
Future Work

1) Study the relationship between retry policies and rate limiting policies

2) Study retries caused by different triggers like capacity degradation

3) Expand beyond simple retry policies by exploring learning-based retry mechanisms
1) Microservices is an important software architecture that demands high fault tolerance.

2) Retry mechanisms, meant to improve fault tolerance, inadvertently sustain metastable failures—failures that persist even when a trigger is removed.

3) We study the relationship between retry policies and performance of a microservice application operating under duress.
Results: Limitations of Exponential Retry

Exp. Retries ()

Tail less amplified - higher throughput (3500)
For 3000 RPS, all requests are under SLO lat

Takeaway: fixed retry policy needs to be load-aware.
Example of Metastable Failure

Time

Normal Case
System is Vulnerable

Cache
Capacity = 3000 req/s

DB

3000 req/s

Cache Fails

Cache
Capacity = 300 req/s

DB

6000 req/s
Including Retries

Cache Recovers
System is in a Metastable State

Cache
Capacity = 3000 req/s

DB

6000 req/s
Including Retries

Retry needs to be system-context aware
How do Retries Cause Metastable Failures?

Initial

Capacity

Retries

Load

Retries

Trigger

Capacity degradation

Load increase

Trigger Removed

Capacity restored
Metastable Failures in Serverless

- **Case #2: Load Increase (Retries) Leading to Sustaining SLO Violations**

  1. Given a function running at the normal load
  2. When there's resource contention leading to capacity degradation, due to cold starts or threshold cap, SLO violations happen
  3. The common strategy for function end-users trigger retries which in turn result in increased load and more cold-started containers
  4. Even after the resource contention is gone, SLO violations still exist

However, resource contention can be transient so instead of creating new containers (cold starts), a better move is to do load shedding.

Sol: A better controller that can differentiate transient or sustaining contention to avoid metastable failures.
Load

Capacity

Time
Results: Limitations of Fixed Retry

Max Retries = 2

Tail less amplified - higher throughput (3500)
For 3000 RPS, all requests are under SLO lat
However, may behave worse for lower load

Takeaway: fixed retry policy needs to be load-aware.
Limitations of Retry

Latency by Percentile Distribution

Latency (milliseconds)

Percentile

wrk_qps1000_retry_2.txt  wrk_qps1000_retry_base.txt  wrk_qps2000_retry_base.txt  wrk_qps2000_retry_2.txt
A Familiar Experience
What causes metastable failures?

Vulnerable State

Triggers

Load Spike

Capacity Decrease