Mastering Chaos: Achieving Fault Tolerance with Observability-Driven Prioritized Load Shedding

Building fault-tolerant, performant and cost-efficient applications with the Aperture open source project

Harjot Gill
CEO, FluxNinja
@harjotsgill

Hardik Shingala
Engineer, FluxNinja
@HDKShingala

@hardik-shingala
Introduction

- **Harjot Gill**
  - Co-founder and CEO @ FluxNinja
    - Founded in 2021
    - Based in the San Francisco Bay Area
    - Announced Aperture open source project in late 2022
  - Dedicated 10+ years building tooling for DevOps and SREs
  - Previously, Co-founder and CEO @ Netsil (Acquired by Nutanix in 2018)
    - Microservices observability start-up, spin-off from University of Pennsylvania
    - Pioneered low-friction API observability: stream-processed packets to reconstruct APIs
    - Mapping complex microservices applications

- **Hardik Shingala**
  - Software Engineer @ FluxNinja
  - 5+ years of experience in cloud native infrastructure products
Metastable failures

Little’s law conundrum: The inevitability of overloads
Little’s law and overloads

\[
L = \lambda W
\]

- \( L \) = Requests in-flight
- \( \lambda \) = Average Throughput
- \( W \) = Average Response time

Every service has an inherent concurrency limit. For a service to remain stable, concurrent requests must be limited.
Availability degrades rapidly

An overload on a service often kicks-off a chain reaction causing an application wide outage...
Cascading failure

1. Degraded database that experiences increased latency.

2. Latency spreads through API call chains triggers a misconfigured circuit breaker at service 2

3. Errors returned from Service 2, impact Service 1 - a large blast radius
Death spiral

1. Nodes 1 and 2 degrade and are replaced by new nodes which are not ready for traffic.

2. The load-balancer redirects all requests to Node 3, risking its degradation as well.

→ API call not in use
→ High latency API call
Retry storm

1. Degraded database that experiences increased latency.

2. Latency propagates via API call chains, prompting retries in Services 2, 3, and 5.

3. Retries put even greater pressure on the database.

→ Healthy API call
→ High-error API call
→ Unhealthy API call
Retry storm: permanent overload

System is in a state of permanent overload

Capacity (rps) and Load (rps)

Load < Capacity
All good!

Capacity

Load

Retry storm

Temporary reduction in capacity leading to a slight overload

Capacity restored to the original level

System is in a state of permanent overload
Metastable failures

1. The system operates in both stable and vulnerable states as load fluctuates.

2. A trigger (e.g. bad deployment, user surge) can transition the system from vulnerable to a metastable state.

3. High load sustains even after initial trigger is removed (permanent overload state).

Metastable Failures in the Wild, Huang et al.
Common triggers

- Insufficient capacity allocation
- Service upgrades that introduce performance-regressions due to bugs
- Unexpected traffic spikes during new product launches or sales promotions
- Slowdowns in upstream services or third-party dependencies
- Retry storm after a temporary failure
- Cache failure leading to higher load on database
- Subset of servers going offline causing excess load on remaining servers

*Metastable failures are unpredictable, yet very common in modern applications*
Mitigation strategies
Building indestructible applications
Local countermeasures are ineffective

Circuit breaking
- Typically implemented in service proxy (e.g. Envoy)
- Localized view between service instances (e.g. error rates)
- Rejects all requests when it “trips”
- Hard to configure the “tripping” threshold as some services are more tolerant to errors
- Client-side technique - does not offer service protection

Static rate-limiting
- Typically implemented as a per-user limit
- Does not offer service protection as the per-user limit is not per-service limit

Reactive auto scaling
- Typically scale workers based on resource consumption (e.g. CPU or memory)
- Can be slow as services need time to warm-up, do discovery, establish database connections and so on
- Bottleneck typically shifts elsewhere
- Expensive to absorb transient traffic spikes

Local countermeasures are often slow, inadequate and ineffective
Mitigation with adaptive load shedding

Normal load

- Requests
- Responses
- Service
- Normal latency

Overload

- Requests
- Responses
- Service
- Normal latency
- Shed excess load

Little’s law
\[ L = \lambda W \]

\( L \) = Requests in-flight
\( \lambda \) = Average Throughput
\( W \) = Average Response time

Service remains **stable** by shedding excess load
Availability degrades gracefully
Requirements for adaptive load shedding

- Determining the ideal load in a constantly changing environment
  - Setting the limit too low can result in rejected requests and wasted capacity
  - Setting the limit too high can lead to slow and unresponsive servers
- Observability: Real-time, global visibility into the state of the entire system
  - Detect overload at databases but load shed at the gateway services
- Controllability: Continuously tracking and correcting system state variables
  - PID controller based closed-loop system
  - Congestion control and active queue management algorithms: TCP BBR, AIMD (Additive increase, multiplicative decrease), CoDel
- Interaction with other control systems with similar goals:
  - Auto scaling
  - Load balancing
Requirements for prioritization

- Optimize user experience and business value: prioritize on attributes such as API endpoints, user types, origin service
- Prioritization and fairness algorithms
  - Token and leaky buckets
  - Network schedulers: weighted-fair queueing
  - Probabilistic dropping
- Estimating the cost (tokens) of admitting different types of requests
  - Tokens = Estimated latency?
  - Tokens = Query complexity?
Global load management with Aperture

Controlling the flux: Observability meets Controllability
Aperture overview

- Open source platform for observability-driven load management
- Programmable through declarative policy language expressed as a control circuit graph
- Common policies are packaged as high-level “blueprints”
  - Load scheduling & workload prioritization
  - Quota enforcement
  - Load ramping
  - Auto scaling
- Layered on top of existing stack
  - SDKs: Java, Go, Python etc.
  - Service Mesh: Istio etc.
  - API Gateways and proxies: Nginx, Kong
Aperture architecture
Adaptive load scheduler
Service protection based on feedback loop
Observability-driven approach

Latency baseline and track deviation

Observe (overload confirmation)

Used Connections

Scheduler
Gateway
Cart
Search
Inventory
Database

Latency
CPU/Mem

Used Connections

Healthy Service
Vulnerable Service

Healthy API Call
High Latency API Call
Observe

App infrastructure

3rd Party DB
Adaptive load scheduling policy

Service latency is queried periodically

Exponential moving average establishes a baseline

Load scheduler corrects deviation from baseline by shedding load
Load scheduler policy component

Policy is expressed as a control "circuit" composed of components

Signals flow between components through ports and the circuit is evaluated periodically

Workloads are defined by matching labels and assigning priorities

Selectors determine agents where this scheduler will be configured
Adaptive load scheduler insertion

1. Service checks with Aperture Agent before serving each request or a feature

2. Rego (OPA) based classifier uses OpenTracing baggage headers to label requests

3. Weighted-fair queuing scheduler enforces prioritization and fairness
Workload prioritization with Aperture

<table>
<thead>
<tr>
<th>Client Service</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cart</td>
<td>255</td>
</tr>
<tr>
<td>Search</td>
<td>150</td>
</tr>
<tr>
<td>Recommendations</td>
<td>10</td>
</tr>
</tbody>
</table>

- Token Bucket
  - Fill rate
  - Tokens?
  - Ok or wait

- Weighted Fair Queueing Scheduler
  - Scheduled requests
  - Dropped requests

- Cart
- Search
- Inventory
Global quotas
Enforcing precise limits
Global quotas

Want to maintain RPS per service?

Don't want external partners to abuse a service?

Honor 3rd party limits?

Gateway

Cart

Inventory

Checkout

Database

Vulnerable Service

Stable Service

App infrastructure

External Service

Gateway

Cart

Inventory

Search

Checkout

RPS: 15k

RPS: 25k

RPS: 10k

RPS: 30k

RPS: 10k

RPS: 30k

RPS: 25k

RPS: 40k

RPS: 40k

RPS: 50k

RPS: 50k

RPS: 40k

RPS: 30k

RPS: 15k

RPS: 15k

RPS: 25k

RPS: 15k
Global quotas

- Service protection
  - When max capacity is known (load testing)
  - Allocate/enforce exact quotas (rps) with other services
- Managing external API rate limits
  - External services such as OpenAI, GitHub, DynamoDB etc. have rate limits. Clients must honor the limit in order to prioritize requests
  - Control costs by preventing accidental overuse
- Preventing abuse
  - Rate-limit external clients based on per-user or per-device quotas
Global quotas in Aperture

1. Distributed token buckets using consistent hashing on labels.

2. Agents take tokens from the owner Agent for the label.

- Aperture provides consistent-hashing based global token buckets
- High performance compared to centralized Redis based system
- Smooth load compared to fixed window rate limiting
- Lazy sync (optional) for even lower latencies
- Schedule (prioritize) requests when capacity is reached
Quota scheduler policy component

Quotas are expressed as -
- Bucket capacity (for allowing bursts) - e.g. 500 requests
- Fill amount and interval - e.g. 25 request per second
- Label key - Buckets are created for each key/value pair, e.g. users, services, API keys

Workloads are defined by matching labels and assigning priorities

Selectors determine agents where this scheduler will be configured
Aperture in FluxNinja ARC
Protecting PostgreSQL by scheduling GraphQL APIs
Protecting PostgreSQL
Without Aperture

Latency when load is normal

Latency spikes when load has increased

High acceptance rate with normal load

Low acceptance rate with high load
With Aperture

Latency when load is normal

Latency is normal even when load has increased

Acceptance rate at normal load

Higher rate for high priority requests
Q & A

• Aperture project on GitHub: https://github.com/fluxninja/aperture
• Aperture Docs: https://docs.fluxninja.com/docs
• Early access to FluxNinja ARC: https://app.fluxninja.com/sign-in