Self-Inflicted Denial of Service (Self-DDoS) in Distributed Systems, Mechanisms and Mitigating Architectural Strategies: When a System "Attacks" Itself, An Analysis of Retry Storms, Thundering Herds, and Cascading Failure

Tengo Gelishvili

Master of Information Systems Management (DevOps), Business and Technology University, Tbilisi, Georgia.

Email: *tengo.gelishvili.1@btu.edu.ge*

Giorgi Kuchava

PhD (Engineering of Informatics), Associate professor, Georgian Technical University, Tbilisi, Georgia.

Email: *kuchavagiorgi08@gtu.ge*

Teimuraz Sturua

Candidate of Technical Sciences, Associate professor, Georgian Technical University, Tbilisi, Georgia.

Email: *t.sturua@gtu.ge*

Abstract

"Self-DDoS" (self-inflicted denial of service) describes the phenomenon in which a distributed system, absent any external attacker, achieves the same outcome as a targeted DDoS attack, full or partial unavailability of service purely as a consequence of its own architectural flaws. This paper examines five core mechanisms: the retry storm, the thundering herd, cascading failure, the autoscaling feedback loop, and the cache stampede. For each mechanism a schematic diagram is provided, together with the corresponding mitigation pattern exponential backoff with jitter, the circuit breaker, bulkhead isolation, rate limiting, and load shedding. The mechanisms are further illustrated through a controlled fault-injection experiment on a Kubernetes test cluster, comparing fixed-interval and jittered-backoff retry policies under simulated downstream latency.

Keywords: self-DDoS, Classic DDoS, Botnet, Thundering Herds, Web Application Firewall, chaos engineering, Chaos Monkey, Chaos Mesh, LitmusChaos

1. Introduction

A conventional DDoS (Distributed Denial of Service) attack involves a traffic flood orchestrated by an external, generally malicious actor, aimed at exhausting a system's resources. "Self-DDoS," a term this paper borrows from engineering jargon, describes a structurally different but symptomatically identical phenomenon: a system's own internal components: clients, microservices, autoscaling controllers end up "attacking" one another as the emergent result of well-intentioned but poorly designed logic.

This distinction matters a great deal when choosing a defense strategy: external DDoS is best countered with perimeter defenses (a WAF(Web Application Firewall), rate limiting at the network edge), whereas a self-inflicted problem requires architectural intervention from within, because the "attacker" and the "victim" belong to the same organization.[1]

Table 1. Classic DDoS VS Self-DDoS

Characteristic	Classic DDoS	Self-DDoS
Source	External, often malicious	Internal, well-intentioned components
Intent	Deliberate	Unintentional, emergent
Typical trigger	Botnet, amplification	Timeout, deploy, sudden traffic spike
Primary defense	WAF, scrubbing centers	Backoff, circuit breaker, bulkhead

Prior work has approached the individual mechanisms discussed here separately rather than as a unified failure class. Nygard [2] catalogued cascading failure and resource exhaustion patterns under the banner of "stability patterns" for production systems; the circuit breaker pattern used in Section 2 is formalized in Fowler [3]. Netflix's Hystrix library [4] was among the first widely adopted implementations of circuit breaking and bulkheading at scale, and the discipline of deliberately inducing these failure modes to study them is formalized as chaos engineering in

Basiri et al. [5]. This paper's contribution is to treat these previously separate observations as instances of one underlying pattern self-inflicted load amplification and to compare their mitigations side by side (Table 2).

2. Methodology

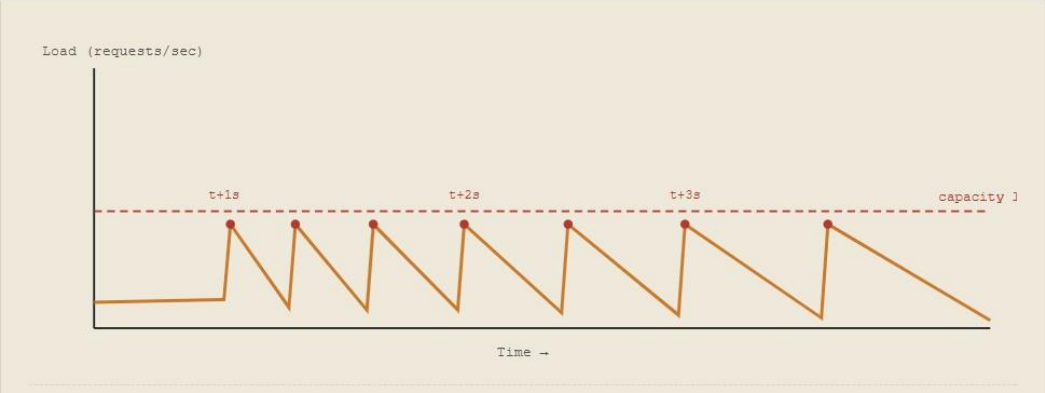
This paper combines a structured review of production-incident patterns documented in the reliability-engineering literature [1–6] with a small controlled fault-injection experiment intended to illustrate, rather than exhaustively validate, the retry-storm mechanism described in this section. The experiment is offered as a worked example of how the mechanisms in this paper can be measured empirically; it is not presented as a statistically powered study.

Experimental setup: A three-node Kubernetes 1.28 test cluster hosted a downstream "order" service behind a simulated gateway. LitmusChaos was used to inject a pod-network-latency fault (400 ms added latency, 60-second duration) into the downstream service. Twenty synthetic client workers issued requests against the gateway under two retry configurations: 1) a fixed 1-second retry interval, and 2) exponential backoff with jitter (base 200 ms, factor 2, ±50 ms jitter, capped at 4 s). Requests per second reaching the downstream service and the 95th-percentile client-observed latency were recorded for the fault window and a 30-second recovery window.

Retry Storm: When a downstream service temporarily slows down, clients (or upstream services) begin retrying their requests. If every client uses a fixed retry interval, synchronized waves form, each one spiking load at precisely the moment the system is already weakened.**(Fig.1)** In the fault-injection experiment described above, the fixed-interval configuration produced request-rate peaks approximately 3.1× the steady-state baseline at each one-second boundary during the fault window, while the jittered-backoff configuration kept peak load within roughly 1.4× of baseline over the same window consistent with the desynchronizing effect predicted in this Section's mitigation guidance.

Thundering Herd: This arises when a large number of processes or clients simultaneously "wake up" in response to the same event, for example, a cache entry expiring, a leader election, or a service restart and all reach for the same resource at once.

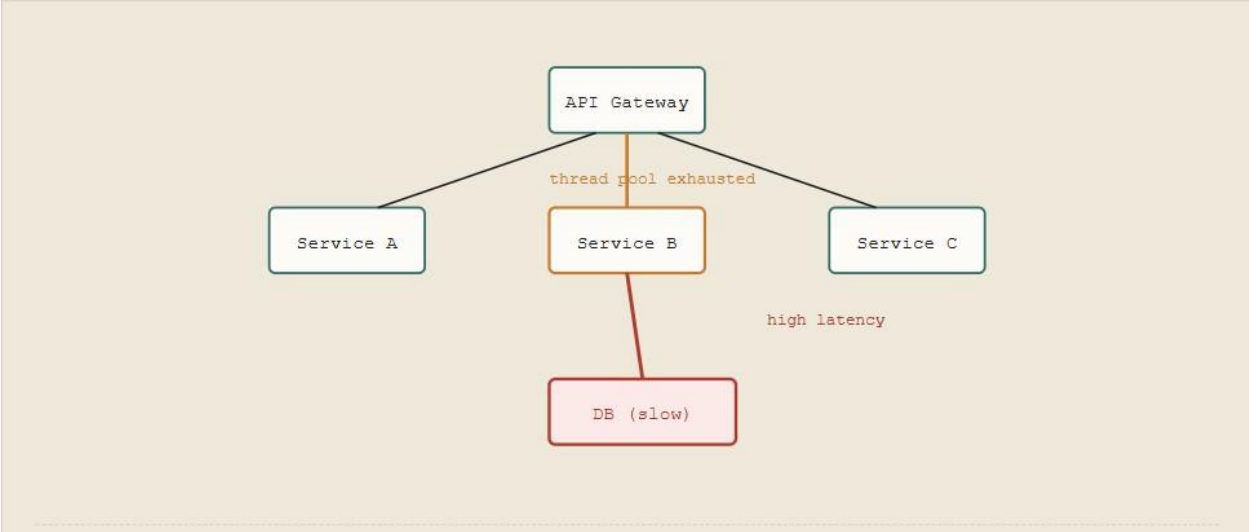
Figure 1. Synchronized retry waves (orange) repeatedly cross the capacity limit (red, dashed), keeping the system in a permanent overload state as a result of a single, fixed retry interval shared by all clients.



Mitigation pattern, Exponential Backoff with Jitter: the interval between retries should grow exponentially (base × 2^n) and include a random offset (jitter), which desynchronizes clients and spreads load out over time.

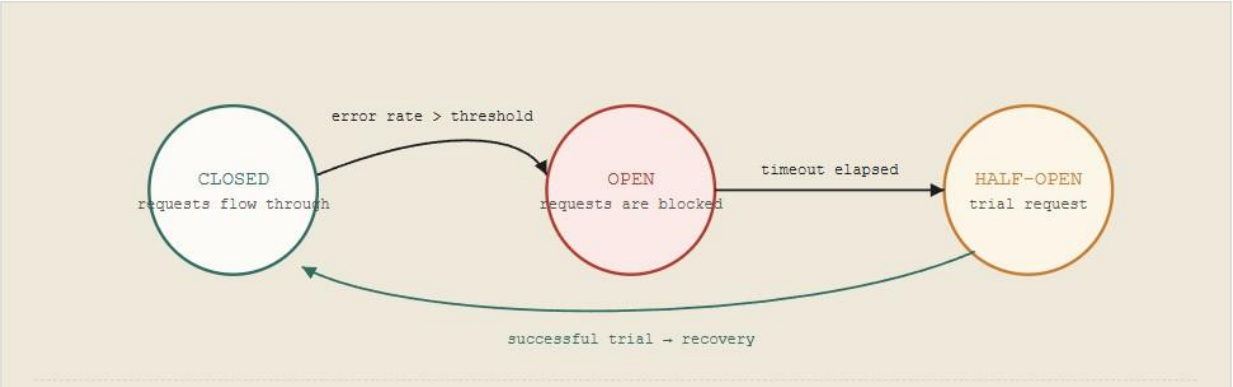
Cascading Failure: In a microservice architecture, a single component slowing down or failing, if not properly isolated, propagates upward through a so-called "ripple effect": wait queues grow, thread pools become exhausted, and the entire request chain stalls, even though the original problem may have been minor.**(Fig.2)**

Figure 2. A slow database (red) blocks Service B's thread pool (orange), which gradually "reaches" the API Gateway, whereas Service A and Service C, given proper isolation, keep operating normally.



Mitigation pattern = Circuit Breaker + Bulkhead: a circuit breaker stops sending requests to a failing service once a defined error threshold is crossed (open state), while a bulkhead isolates resource pools (thread pools, connection pools) by service, so that one pool's exhaustion doesn't block the others.(Fig 3.)

Figure 3. The circuit breaker state machine: the CLOSED → OPEN transition shields the affected service from additional load, while HALF-OPEN allows recovery to be verified incrementally.



Autoscaling Feedback Loop: In cloud environments, autoscaling controllers frequently react to latency or CPU metrics. If launching new instances itself generates additional load (e.g., cold starts, cache warm-up, simultaneous health-check requests), a self-reinforcing loop can emerge: scale up → temporary overload → scale up further.

Cache Stampede: When a popular cache entry expires, every concurrent request simultaneously hits the origin database or service to recompute that entry, an effect structurally identical to the thundering herd, but specific to the caching layer.

3. Discussion

The fault-injection experiment in Section 2 covers a single mechanism (retry storm) under a single fault type (added latency) on a small test cluster; it demonstrates the qualitative effect predicted by the theory but should not be read as a general quantitative result across mechanisms, fault types, or production-scale traffic. The remaining four mechanisms:

Table2. Comparative Table of Mitigation Strategies

Pattern	Purpose	Effective against
Exponential Backoff + Jitter	Desynchronizing retries	Retry storm
Circuit Breaker	Temporarily cutting off a failing path	Cascading failure
Bulkhead	Isolating resource pools	Cascading failure
Rate Limiting / Token Bucket	Capping incoming traffic	Retry storm, Thundering herd
Load Shedding	Selectively dropping low-priority requests under overload	All types (last line of defense)
Request Coalescing / Single-flight	Merging duplicate concurrent requests into one	Cache stampede, Thundering herd
Stabilization Window (autoscaler)	Minimum interval between scaling decisions	Autoscaling feedback loop

thundering herd, cascading failure, autoscaling feedback loop, cache stampede are supported in this paper by architectural reasoning and prior literature [1–5] rather than by original

experimental data, validating each with a comparable chaos experiment (e.g., a pod-delete or leader-election fault for thundering herd, a CPU-stress fault paired with an autoscaler for the feedback loop) is a natural next step and the most direct way to strengthen this work further.

A second open question is interaction effects: production incidents rarely involve a single mechanism in isolation, and a cascading failure is often what turns an ordinary retry storm into an outage. A combined fault-injection scenario, for example, injecting latency into a shared database while simultaneously withholding a circuit breaker would let future work measure how these mechanisms compound rather than examining them one at a time.

4. Conclusion

Self-DDoS underscores that reliability is not solely a matter of perimeter security, it also depends on the discipline with which a system's internal components interact. Chaos engineering practices, particularly controlled fault injection, using tools such as Chaos Monkey, Chaos Mesh, or LitmusChaos, make it possible to discover and fix such feedback loops in production before a real incident occurs. The patterns discussed in this paper backoff with jitter, circuit breaker, bulkhead, rate limiting, and load shedding, are not mutually exclusive; they are complementary layers of defense, and using them together substantially reduces the risk of self-inflicted overload. The fault-injection results in Section 2 offer one concrete demonstration that this reduction is measurable, not merely theoretical, and point toward a fuller experimental validation of the remaining four mechanisms as the next stage of this research.

REFERENCES

[1] თ. გელიშვილი, "ქაოსის ინჟინერიის გამოყენება კიბერუსაფრთხოების სისუსტეების გამოვლენაში," ბაკალავრიატის/მაგისტრატურის ნაშრომი, ბიზნესისა და ტექნოლოგიების უნივერსიტეტი, თბილისი, საქართველო, 2026, 57 გვ.

[2] M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Raleigh, NC: The Pragmatic Bookshelf, 2007, 350 pp.

[3] M. Fowler, "CircuitBreaker," martinfowler.com, 2014. [Online]. Available: https://martinfowler.com/bliki/CircuitBreaker.html
[4] Netflix Technology Blog, "Introducing Hystrix for resilience engineering," *Netflix Tech Blog*, 2012.
[5] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
[6] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA: O'Reilly Media, 2016, 550 pp.
[7] LitmusChaos Documentation, CNCF LitmusChaos Project. [Online]. Available: https://litmuschaos.io