Odporność i niezawodność

Projektowanie na awarie · Wzorce stabilności · Chaos Engineering

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Fallacies of Distributed Computing — 8 fałszywych założeń
- Projektowanie na awarie (Design for Failure)
- Wzorce odporności: Timeout, Retry, Circuit Breaker, Bulkhead, DLQ
- Graceful degradation
- Rate limiting i throttling
- Load balancing — algorytmy i health checks
- Autoskalowanie — HPA, VPA, KEDA
- Chaos Engineering
- ☁ Managed resilience w chmurze

Fallacies of Distributed Computing

8 fałszywych założeń (Deutsch 1994, Gosling 1997)

Każdy projektant systemów rozproszonych w końcu się na nich sparzy

1. Sieć jest niezawodna
2. Opóźnienie wynosi zero
3. Przepustowość jest nieskończona
4. Sieć jest bezpieczna
5. Topologia się nie zmienia
6. Jest jeden administrator
7. Koszt transportu wynosi zero
8. Sieć jest jednorodna

Peter Deutsch (Sun Microsystems, 1994) sformułował 7 z nich. James Gosling dodał ósme w 1997.

Dlaczego wciąż aktualne po 30 latach?

Więcej systemów rozproszonych niż kiedykolwiek

Mikroserwisy, serverless, edge computing, AI inference — każde wywołanie to sieć.

Współczesne pułapki:

- **Fallacy #1 w mikroserwisach**: 1 request = 10 wywołań sieciowych = 10 szans na awarię
- **Fallacy #2 w multi-region**: cross-region latency 150ms, nie 0.5ms
- **Fallacy #5 w Kubernetes**: topologia zmienia się co minuty (autoscaling, rolling deploy)
- **Fallacy #7 w chmurze**: cross-AZ data transfer = realne pieniądze

A. Rotem-Gal-Oz — „Fallacies of Distributed Computing Explained“ (2006)

Projektowanie na awarie

Awaria to norma, nie wyjątek

Design for Failure

W systemie rozproszonym **coś zawsze jest zepsute**. Projektuj tak, jakby każdy komponent mógł w każdej chwili przestać działać.

Łańcuch awarii (Nygard, *Release It!*):

- Serwis C zwalnia → serwis B czeka → wątki B się kończą → serwis A nie dostaje odpowiedzi → użytkownik widzi błąd
- Jedna wolna zależność może **zatopić cały system**

Najgorsza awaria to nie crash — to spowolnienie

Serwis odpowiadający w 30s jest gorszy niż martwy. Martwy serwis daje natychmiastowy błąd. Wolny serwis blokuje wątki i zasoby wszystkich wywołujących.

Wzorce odporności

Timeout — nie czekaj w nieskończoność

Zasada

Ustaw timeout na **każdym** wywołaniu zdalnym — nawet cross-process na tej samej maszynie.

Jak dobrać timeout?

- Zmierz p99 latencji normalnego ruchu
- Timeout = p99 + margines (np. 2×)
- Zbyt krótki → fałszywe alarmy
- Zbyt długi → wątki blokują się za długo

Brak timeoutu = katastrofa

RMI (Java) domyślnie nie ma timeoutu. Nygard opisuje przypadek, w którym zablokowane wątki kumulowały się przez godziny, aż system padł całkowicie.

M. Nygard — Release It!, *2nd ed. (2018), rozdz. 5: „Timeouts“*

Retry + Exponential Backoff + Jitter

Przejściowe awarie rozwiązują się same

Network blip, pod restart, chwilowe przeciążenie ⟶ retry po chwili zwykle zadziała.

Naiwny retry = thundering herd

Stały interwał (co 500ms, 3 razy) ⟶ 1000 klientów retryuje w **tym samym momencie** ⟶ serwis, który się właśnie podnosił, pada ponownie.

Rozwiązanie (AWS Builders' Library, Marc Brooker):

- **Exponential backoff**: 500ms ⟶ 1s ⟶ 2s ⟶ 4s (mnożnik 2×)
- **Jitter**: losowy offset ±20% desynchronizuje klientów
- **Cap**: maksymalny czas backoff (np. 30s)
- **Limit**: max liczba prób (np. 3–5), potem odpuść

AWS Builders' Library — „Timeouts, retries, and backoff with jitter"

Retry — pułapki

Retries are selfish (AWS)

Każdy retry zużywa dodatkowy czas serwera. Przy masowym retryowaniu **sam retry staje się atakiem DDoS** na własną infrastrukturę.

Warunki bezpiecznego retry:

- Operacja musi być **idempotentna** (nawiązanie do wykładu 3)
- Retry tylko na błędy **przejściowe** (timeout, 503) — nie na 400/404
- Ogranicz liczbę prób **i** łączny czas
- Dodaj jitter do **wszystkiego**: timerów, cron jobów, scheduled tasków

Wskazówka z AWS

Jitter nie tylko do retry — dodawaj go do wszystkich okresowych zadań. Rozprasza szczyty ruchu i ułatwia skalowanie downstream services.

Circuit Breaker — ochrona przed kaskadą

Circuit Breaker (Nygard, *Release It!*)

Wzorzec z elektrotechniki. Gdy zależność zawodzi, **przerwij obwód** — przestań do niej dzwonić. Daj jej czas na regenerację.

Trzy stany:

Stan	Zachowanie	Przejście
Closed	Normalny przepływ, awarie zliczane	Próg awarii przekroczony ⟶ Open
Open	Żądania natychmiast odrzucane	Po timeout ⟶ Half-Open
Half-Open	Kilka próbnych żądań przepuszczone	Sukces ⟶ Closed, porażka ⟶ Open

Biblioteki: **Resilience4j** (Java), **Polly** (.NET), **opossum** (Node.js)

Bulkhead — izolacja zasobów

Bulkhead (grodzie wodoszczelne)

Izoluj zasoby tak, żeby awaria jednej zależności **nie wyczerpała** zasobów współdzielonych z innymi. Nazwa od grodzi w statkach — zalanie jednego przedziału nie topi statku.

Warianty:

Wariant	Mechanizm	Przykład
Thread pool	Osobna pula wątków per zależność	Serwis płatności: max 20 wątków
Semaphore	Limit równoczesnych wywołań	Max 50 concurrent calls do serwisu X
Connection pool	Osobna pula połączeń per serwis	Osobny pool DB dla krytycznych operacji

M. Nygard — Release It!*, 2nd ed. (2018), rozdz. 5: „Bulkheads“*

Dead Letter Queue — izolacja wadliwych wiadomości

DLQ — nie pozwól, by jedna wiadomość zablokowała resztę

Wiadomość, która nie może być przetworzona (*poison message*) po wyczerpaniu prób trafia do osobnej kolejki — DLQ. Reszta strumienia płynie dalej.

Przepływ:

- Consumer przetwarza wiadomość
- Awaria ⟶ retry z backoff (max N prób)
- Po wyczerpaniu ⟶ wiadomość trafia do DLQ
- Operator analizuje, naprawia bug, wykonuje **replay**

DLQ to kolejka triage, nie cmentarz

Bez alertów i procedury obsługi DLQ to martwa strefa. Monitoruj: głębokość DLQ, wiek najstarszej wiadomości.

AWS SQS: `RedrivePolicy` z `maxReceiveCount` — broker sam przenosi.

Kafka: brak natywnego DLQ — implementacja przez error handler + topic `.dlq`.

Graceful degradation — częściowa odpowiedź > brak odpowiedzi

Zasada

Gdy niekrytyczna zależność zawodzi, **serwuj to co masz** zamiast zwracać 500.

Sytuacja	Bez degradacji	Z graceful degradation
Serwis rekomendacji padł	Strona produktu = 500	Strona bez sekcji „Polecane“
Cache Redis niedostępny	Timeout na każdym requeście	Pomiń cache, idź do DB (wolniej)
Serwis cen zwraca błąd	Brak koszyka	Pokaż ostatnie znane ceny (cached)

Kluczowe: rozróżnij zależności **krytyczne** (bez nich nie można obsłużyć requestu) od **niekrytycznych** (można pominąć).

Wzorce odporności — jak się łączą

Warstwy ochrony

1. **Timeout** — nie czekaj w nieskończoność (pierwsza linia)
2. **Retry + backoff + jitter** — obsłuż przejściowe awarie
3. **Circuit Breaker** — odetnij trwale zepsutą zależność
4. **Bulkhead** — ogranicz blast radius awarii
5. **DLQ** — izoluj wadliwe wiadomości z przetwarzania
6. **Graceful degradation** — serwuj częściowe wyniki

Żaden wzorzec sam nie wystarczy

Timeout bez circuit breakera = powtarzane wolne wywołania.

Retry bez jittera = thundering herd.

Circuit breaker bez graceful degradation = użytkownik widzi błąd.

Stosuj razem jako spójną warstwę odporności.

Rate limiting i throttling

Algorytmy rate limiting

Algorytm	Burst	Pamięć	Precyzja	Najlepszy do
Token Bucket	Tak	O(1)	Dobra	Publiczne API (AWS, Stripe)
Leaky Bucket	Nie	O(n)	Dobra	Wygładzanie ruchu
Fixed Window	Granica	O(1)	Słaba	Proste filtry, internal
Sliding Window Log	Nie	O(n)	Dokładna	Płatności, auth
Sliding Window Counter	Nie	O(1)	98%	Miliony kluczy API

Token Bucket — najpopularniejszy

Mechanizm

Kubełek z max pojemnością. Tokeny uzupełniają się stałym tempem. Każde żądanie zużywa jeden token. Brak tokenów = odrzucenie.

Dwa parametry:

- **Capacity** (max burst) — np. 100 żądań
- **Refill rate** (średni limit) — np. 10/s

Przykład

capacity=100, refill=10/s. Burst 100 żądań natychmiast ⟶ OK. Potem: max 10/s. Po 5s bez ruchu: kubełek znów pełny (50 tokenów).

Używany przez: **AWS API Gateway**, **Stripe**, **Kong**.

Rozproszony rate limiter

Problem

20 podów API, użytkownik trafia losowo na różne. Lokalny rate limiter pozwoli na 20× limit.

Rozwiązanie: **Redis + Lua scripts** (atomowe operacje):

- `INCR` + `EXPIRE` dla fixed window
- Sorted set z timestampami dla sliding window log
- Lua script = atomowa operacja na Redisie

Gdy rate limiter padnie — fail-open czy fail-closed?

Fail-open: przepuść ruch ⟶ ryzyko przeciążenia.
Fail-closed: zablokuj ruch ⟶ ryzyko niedostępności.
Większość systemów: **fail-open** z alertem.

Odpowiedź HTTP: `429 Too Many Requests` + `Retry-After` header.

A. Xu — System Design Interview, *rozdz. 4: „Design a Rate Limiter“*

Load balancing

Algorytmy load balancingu

Algorytm	Jak działa	Kiedy stosować
Round Robin	Rotacja kolejno po serwerach	Serwery identyczne, requesty jednorodne
Weighted RR	Więcej ruchu do mocniejszych serwerów	Różne pojemności (mixed hardware)
Least Connections	Do serwera z najmniejszą liczbą połączeń	Requesty o zmiennym czasie trwania
Least Response Time	Do serwera z najkrótszym ostatnim czasem	Latency-sensitive workloady
Consistent Hashing	Hash klucza ⟶ serwer na pierścieniu	Cache affinity, session stickiness

Google SRE (rozdz. 20): przejście z Least-Loaded na **Weighted Round Robin** dało lepszy rozkład obciążenia.

Google SRE Team — SRE Book, *rozdz. 19–20*

Health checks i L4 vs L7

Health checks

Active: LB pinguje serwery co N sekund (`/health`).

Passive: monitoring realnego ruchu (za dużo 5xx ⟶ wyłącz).

Best practice: oba jednocześnie.

L4 vs L7

L4 (Transport): routing po IP+port. Szybkie, nie widzi HTTP. Np. AWS NLB.

L7 (Application): routing po URL, headerach, cookies. SSL termination. Np. AWS ALB, Envoy.

Typowy błąd

Jeden load balancer = single point of failure. Redundancja LB (active/passive pair, anycast DNS) jest krytyczna.

Autoskalowanie

HPA, VPA, KEDA — trzy wymiary skalowania

Cecha	HPA	VPA	KEDA
Kierunek	Horyzontalny (repliki)	Wertykalny (CPU/RAM)	Horyzontalny (eventy)
Metryki	CPU, memory, custom	Historia użycia	65+ zewn. źródeł
Scale to 0	Nie (min=1)	Nie	**Tak**
Workload	Stateless web, API	Right-sizing	Kolejki, batch, eventy

Nie mieszaj HPA i VPA na tej samej metryce!

HPA skaluje repliki po CPU + VPA zmienia requesty CPU = pętla sprzężenia zwrotnego.

Bezpieczny pattern: HPA na custom metryce (RPS), VPA w trybie Off (tylko rekomendacje).

Złota zasada: scale up szybko, scale down wolno

Asymetryczne skalowanie

- **Scale up**: krótkie okno stabilizacji, agresywne progi
- **Scale down**: `stabilizationWindowSeconds` ≥ 300s, ostrożne progi
- Nagły spadek ruchu nie oznacza, że nie wróci za chwilę

Produkcyjny pattern (2025/26):

Warstwa	Rola
HPA	Skaluje repliki na podstawie RPS / CPU
VPA (Off mode)	Rekomenduje resource requests (aplikowane w CI)
KEDA	Skaluje async workerów od 0 na podst. głębokości kolejki
Cluster Autoscaler	Dodaje nody gdy pody są w stanie Pending

KEDA — Kubernetes Event-Driven Autoscaler · keda.sh

Chaos Engineering

Netflix Simian Army (2011)

Filozofia

„Najlepszy sposób na uniknięcie awarii to ciągłe ich wywoływanie.“ — Netflix

Narzędzie	Co robi
Chaos Monkey	Losowo zabija instancje produkcyjne w godzinach pracy
Chaos Gorilla	Symuluje awarię całej Availability Zone
Chaos Kong	Symuluje awarię całego regionu AWS
Latency Monkey	Wprowadza sztuczne opóźnienia w komunikacji

Open-source od 2012. Chaos Monkey wymusił, by **każdy** serwis Netflixa przetrwał utratę instancji bez wpływu na użytkowników.

Netflix Tech Blog — „The Netflix Simian Army“ · netflixtechblog.com

Principles of Chaos Engineering

principlesofchaos.org

1. **Zdefiniuj steady state** — co oznacza „normalnie“? (RPS, error rate, p99)
2. **Wprowadzaj realistyczne zdarzenia** — kill instancji, packet loss, latency injection
3. **Eksperymentuj w produkcji** — staging nie odwzorowuje rzeczywistości
4. **Automatyzuj** — ciągłe eksperymenty, nie jednorazowe
5. **Minimalizuj blast radius** — zacznij od małego procenta ruchu

GameDay

Zorganizowane, ograniczone czasowo ćwiczenie: inżynierowie celowo wprowadzają awarie i obserwują zachowanie systemu. Amazon prowadzi GameDay od 2003. Buduje *muscle memory* zespołu na incydenty.

P. Alvaro, K. Andrus et al. — „Automating Failure Testing Research at Internet Scale“ (2016)

Case study: Design a Rate Limiter

Design a Rate Limiter (A. Xu, rozdz. 4)

Decyzje projektowe

- **Gdzie umieścić?** API Gateway (edge) vs middleware vs dedykowany serwis
- **Jaki algorytm?** Token Bucket dla publicznych API, Sliding Window Counter dla high-cardinality
- **Jak zbudować rozproszony?** Redis + Lua scripts (atomowe operacje)

Co gdy rate limiter sam padnie?

- **Fail-open**: przepuść ruch, zaakceptuj ryzyko przeciążenia (częstszy wybór)
- **Fail-closed**: zablokuj ruch, zaakceptuj niedostępność
- W obu przypadkach: **alert** + **automatyczny fallback**

Kluczowy trade-off: precyzja (Sliding Window Log, O(n) pamięci) vs skalowalność (Token Bucket, O(1) pamięci).

A. Xu — System Design Interview, *rozdz. 4*

Managed resilience w chmurze

Managed load balancing i autoscaling

Dostawca	Usługa	Cechy
AWS	ELB (ALB / NLB)	L7 content-based routing, L4 ultra-high throughput
AWS	Auto Scaling Groups	EC2 autoscaling, target tracking, predictive scaling
GCP	Cloud Load Balancing	Globalny L7 LB, anycast IP, auto-SSL
Azure	Front Door + LB	Globalny L7 + regionalny L4, WAF wbudowany
CNCF	KEDA	Event-driven autoscaling K8s, 65+ scalerów, scale-to-zero

Managed rate limiting i chaos engineering

Dostawca	Usługa	Cechy
AWS	WAF + Shield	Rate limiting na edge, DDoS protection
AWS	Fault Injection Service	Managed chaos engineering (kill, latency, throttle)
GCP	Cloud Armor	WAF, rate limiting, adaptive protection
Azure	Chaos Studio	Fault injection na Azure resources
CNCF	LitmusChaos	Kubernetes-native chaos engineering

Zasada

Nie buduj własnego rate limitera ani chaos platformy — chyba że masz **bardzo specyficzne** wymagania. Managed services obsługują edge cases, których sam nie przewidzisz.

Serwis A wywołuje serwis B, który wywołuje serwis C.

Serwis C zaczyna odpowiadać z opóźnieniem 30 sekund zamiast 100ms.

Jak ta awaria propaguje się w górę łańcucha?

Które wzorce (timeout, circuit breaker, bulkhead) zadziałają — i w jakiej kolejności?

Podsumowanie

Kluczowe wnioski

1. **8 fallacies** — sieć zawodzi, opóźnienie nie jest zerowe, topologia się zmienia
2. **Design for Failure** — awaria to norma, projektuj warstwy ochrony
3. **Timeout + Retry + Circuit Breaker + Bulkhead** = spójna warstwa odporności
4. **DLQ** — izoluj poison messages, nie blokuj strumienia
5. **Rate limiting**: Token Bucket dla API, Sliding Window Counter dla skali
6. **Load balancing**: dobieraj algorytm do workloadu, nie do przyzwyczajenia
7. **Autoscaling**: HPA + VPA (Off) + KEDA + Cluster Autoscaler = pełny stack
8. **Chaos Engineering** — testuj awarie zanim one przetestują ciebie
9. ☁ **Managed services** — ALB, WAF, KEDA, Fault Injection Service

Źródła i lektury

- M. Kleppmann — *DDIA*, rozdz. 8: „The Trouble with Distributed Systems“
- M. Nygard — *Release It!*, 2nd ed. (2018)
- Google SRE Team — *SRE Book*, rozdz. 18–22
- A. Xu — *System Design Interview*, rozdz. 4: „Design a Rate Limiter“
- AWS Builders’ Library — „Timeouts, retries, and backoff with jitter“
- Netflix Tech Blog — „The Netflix Simian Army“
- Principles of Chaos Engineering — principlesofchaos.org
- KEDA — keda.sh

Następny wykład

Wykład 7: Migracja i ewolucja architektury

Strangler Fig · Branch by Abstraction · Anti-Corruption Layer · Feature Flags