http://bit.ly/2Fj5pj6

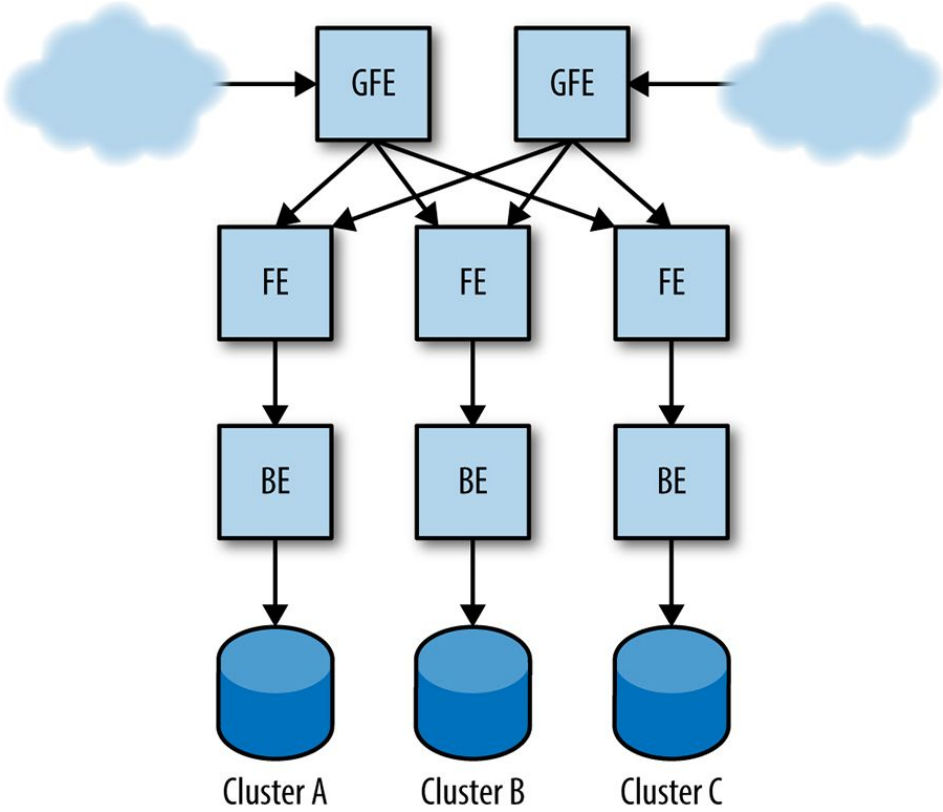
Addressing Cascading Failures
處理連鎖故障

(SRE Ch22)

ccshih
iiiccshih@gmail.com

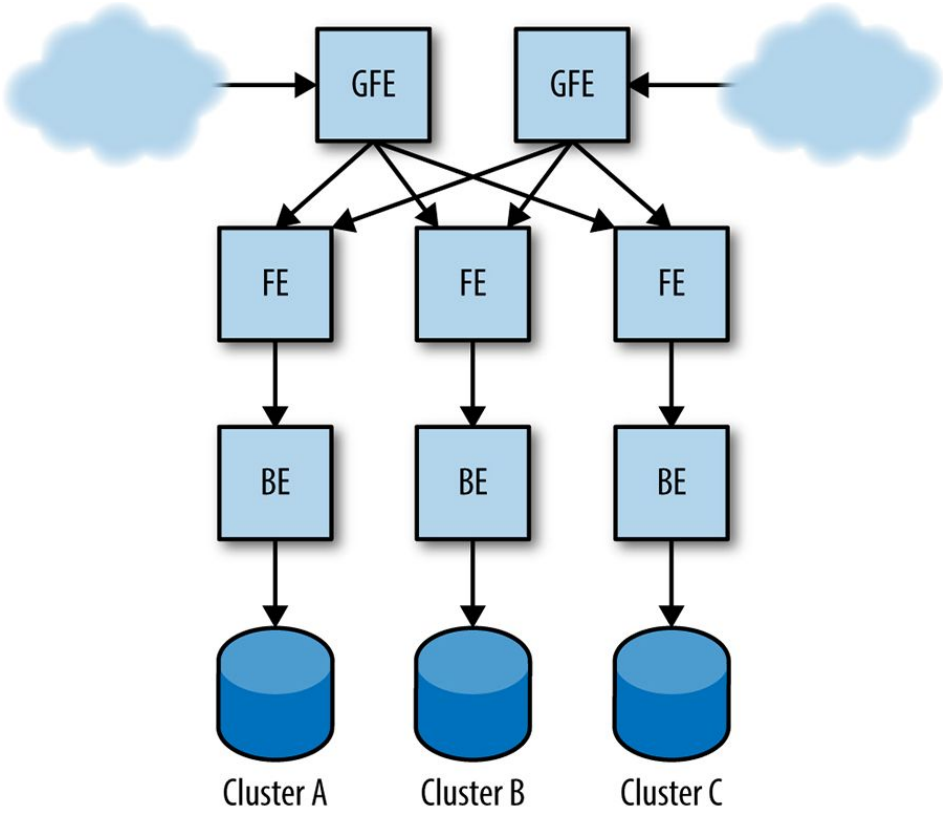
Cascading Failures (連鎖故障)

- 整體系統的一部分故障，導致系統其他部分更容易故障(會傳染的故障)
- 以右圖為例
  - 假設所有 FE 目前都接近滿載
  - 掛了一個 FE，可能會讓其他 FE 因過載而故障
  - 當 FE 故障後，GFE?
- 分散式系統的日常



直覺解法？

- 避免接近滿載
  - 保持閒置容量
  - Cloud Auto-scale
- 避免故障擴散
  - 限流



大綱

- 連鎖故障的原因
- 避免伺服器過載
- 慢啟動與冷快取
- 連鎖故障的觸發時機
- 針對連鎖故障的測試方式
- 連鎖故障的立即處置
- 結論

1. 連鎖故障的原因

- 過載 (Server Overload)
  - 如前例，掉了一些 Instance，導致其他的 Instance 過載，然後連鎖故障
  - 許多連鎖故障都與過載有直接或間接關係
- 資源耗盡 (Resource Exhaustion)
  - 一般的症狀：延遲 (latency) 增加，錯誤率上升，低品質的結果
  - CPU, Memory, Thread, File Descriptor 耗盡有不同的副作用
- 上下游間的蔓延

當 CPU 耗盡時，會發生...

許多的惡性循環！

- 處理中的請求增加 (in-flight request ↑)
  - 同時處理的請求量↑
  - 佔用更多的運算資源，也可能佔用更多的下游資源
- 請求在 Queue 中等待
  - 增加延遲時間
  - Queue 佔用更多 Memory
- Thread Starvation
  - 需要的 Lock 被長時間綁架，Thread 只能 wait
- 被 Watchdog 重啟
- 因為超過客戶要求的 RPC deadline，而被 Retry 轟炸
  - 不斷做白工 (Think Lean)
- 不斷 Context-switch 導致 cache 不能發揮效果

當 Memory 耗盡時，會發生...

- 行程被賜死
- Garbage Collection (GC)
  - GC Death Spiral：GC → 耗 CPU → 處理中的請求增加 → 耗 Memory → GC
- 擠壓應用程式快取空間
  - 增加下游負擔

當 Thread 耗盡時，會發生...

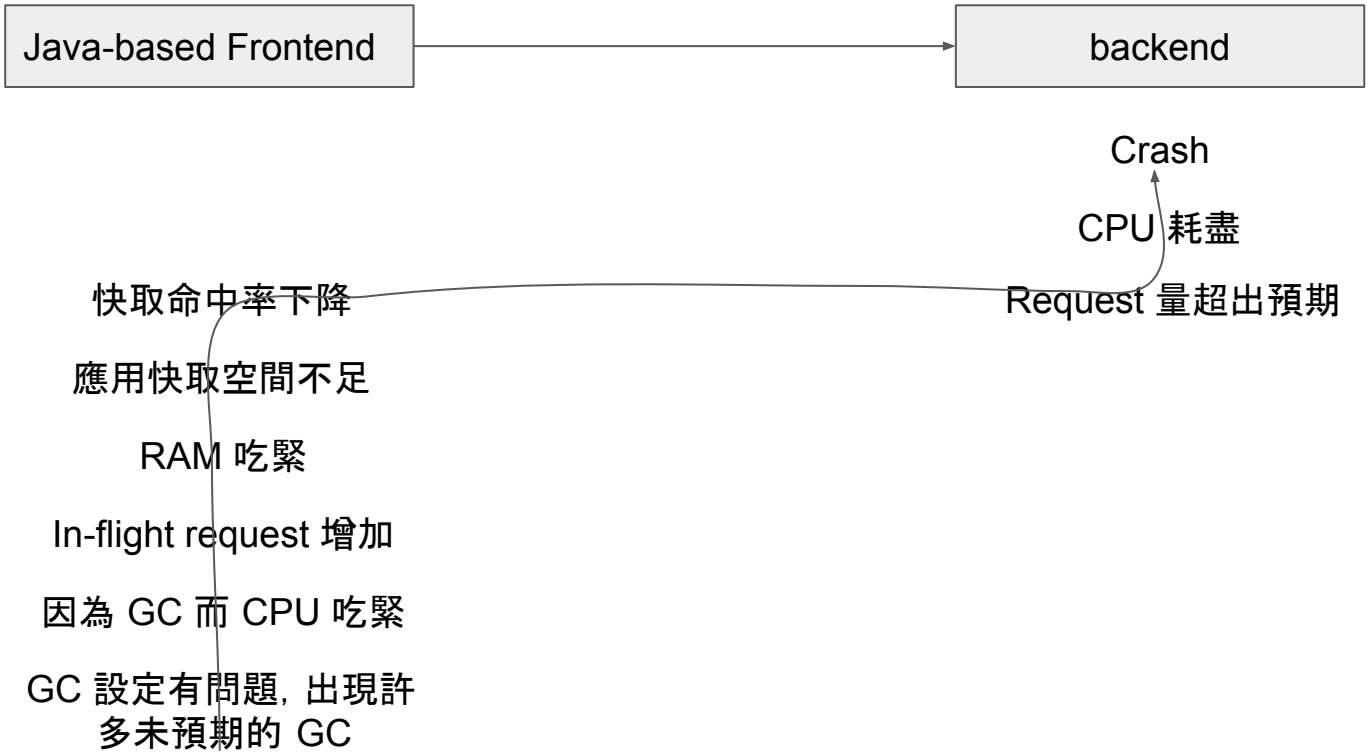
- 錯誤，讓 health check 失敗
- 若增加 Thread，會消耗 Memory

當 File descriptor 耗盡時，會發生...

- 無法建立 connection，讓 health check 失敗

故障循相依關係蔓延，難以追查根因

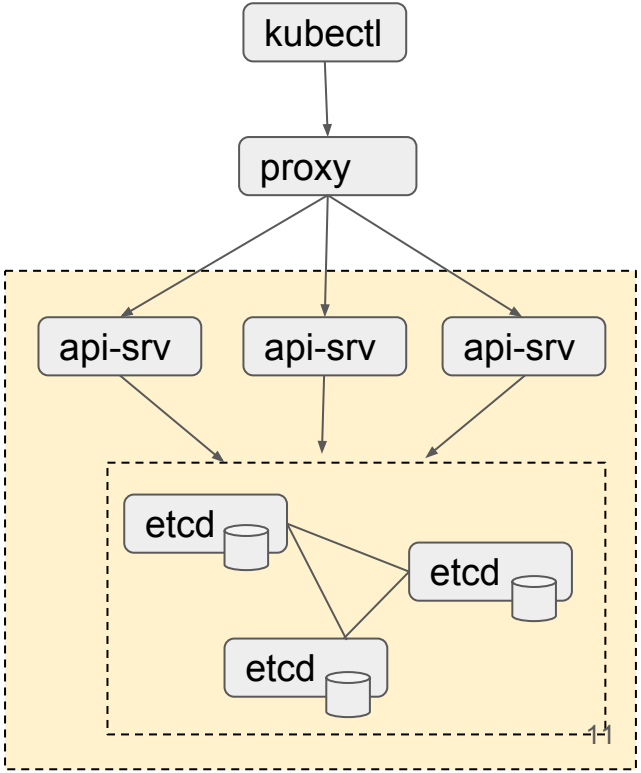
＊ 若跨不同單位，更難追 查



Case Study

- **問題徵兆:**
  - `kubectl` 有時候會發生 timeout. (p.s. `kubectl -v=6 可以顯示所有API 細節指令)
- **嘗試解決方式:**
  - 一開始以為是 `kube-apiserver` 服務器過分忙碌, 試著增加 proxy 來做 replica 來幫忙 load balance
  - 但是超過 10 個備份 master 的時候, 他們發現問題一定不是因為 kube-apiserver 無法承受. 因為 `GKE 透過一台 32-core VM 就可以乘載 500台`
- **原因:**
  - 扣除掉這些原因, 開始要懷疑 master 上剩下的幾個服務. (etcd, kube-proxy)
  - 於是開始試試看來調整 `etcd`
  - 透過使用 datadog 來調查 etcd 的吞吐量, 發現有異常延遲 (latency spiking ~100 ms)
  - 透過 Fio 工具來做效能評估, 發現只用到了 10% 的IOPS(Input/Output Per Second), 由於寫入延遲 (write latency 2ms), 造成整個效能被拖累.
  - 試著把 SSD 從網路硬碟變成每台機器有個 local temp drive (依舊是 SSD)
  - 結果從 ~100ms —> 200us

資料來源: [好文介紹] Scaling Kubernetes to 2,500 Nodes

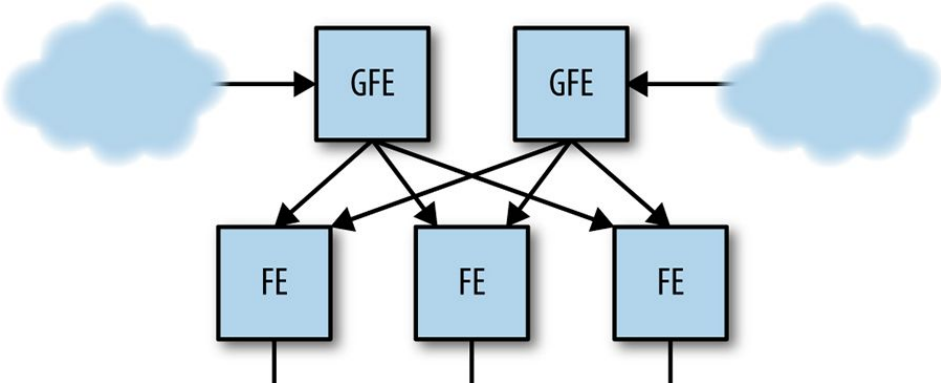


被洶洶洪流沖到再起不能

- 假設一個服務的最大承載量為 100 QPS，因無法承載上游 110 QPS 而開始連鎖故障
- 此時若上游降為 90 QPS，仍無法抑止連鎖故障。因為當連鎖故障發生時，最大承載量已遠小於 100 QPS
- 若連鎖故障下，最大承載量降為正常狀況下的 10%，則上游需降為低於 10 QPS，才能使系統穩定，然後有機會回復
- 會有多少比例的伺服器能在連鎖故障下正常運作呢？取決於：
  - 啟動時間 (Think Cloud-native)
  - 暖機時間
  - 存活時間

上游 LB 錯認狀態，在忙碌時引發連鎖故障

- 可提供服務，但上游 LB卻以為不行
  - 嚴謹的 Health Check
- 承載量不足，因過載而發生連鎖故障

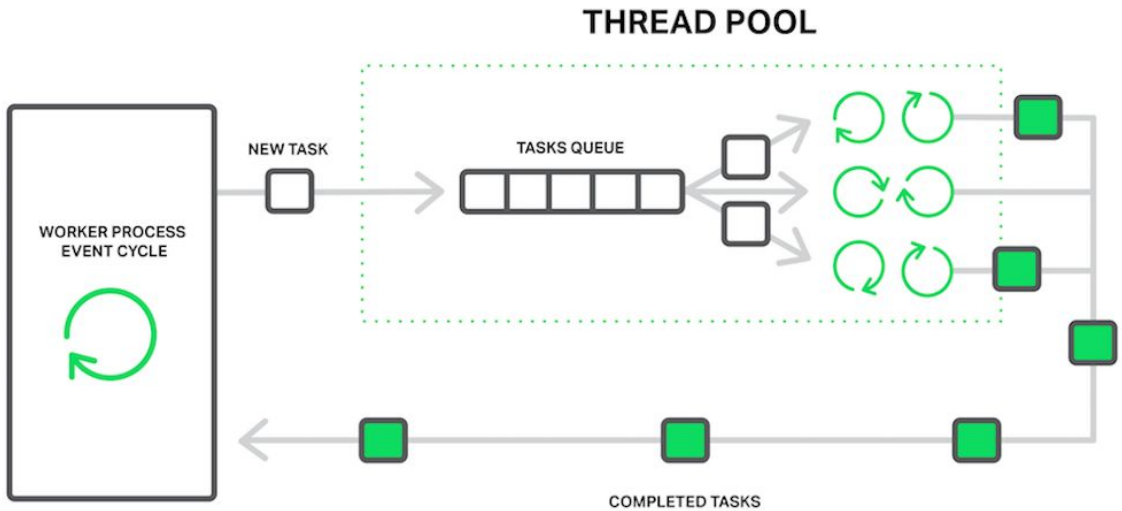


2. 如何避免過載

- 事先壓測來確認承載上限，測試過載時會發生的故障模式
- 在過載時，為節省資源，可以提供較差的結果
- 自己限流
- 在更上游限流
- 先做好容量規劃 (capacity planning)
  - 必要但不可完全依賴

Queue 的用途

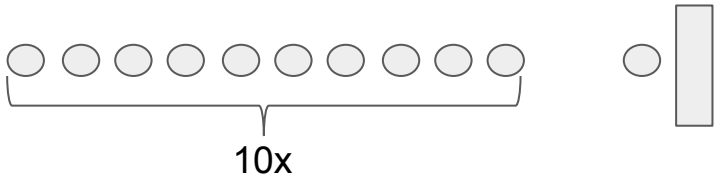
- 若採用 Thread-per-request 處理方式，通常會有個 Queue 檔在 Thread Pool 前，作為緩衝 (Think 銀行)



Source: https://www.nginx.com/blog/thread-pools-boost-performance-9x/

Queue 容量越大越好嗎？

- Queue 容量越大，可以收容越多的緩衝
- 但排隊的會不開心，因為要等很久
  - 假設 Queue 容量是 Thread pool size 的 10 倍
  - 假設 Thread 處理一個 request 平均要 0.1 秒
  - 假設 Queue 滿了，一個 request 的處理時間要 1.1 秒 (11倍)



- 等很久這問題，對分散式系統尤其嚴重 (後論 Deadline)

Queue 容量的建議

- 流量穩定的話，Queue 容量設定在 Thread pool size 的 50% 或更少
- Queue 滿了就直接回絕客戶請求
  - Gmail 常常連 Queue 都不用，若過載就直接 failover 給其他 server 處理

流量拋棄 (Load shedding)

- 避免 Server 被沖垮
- 可能的策略
  - Per-task throttling：限制場內人數
  - 從 FIFO 換成 LIFO 或 Controlled-delay algorithm：客戶不等了就不用浪費時間
    - 與 Deadline (後述) 搭配佳
  - 客戶分級
- 每一個元件都要掐流量嗎？還是只要掐住上游？

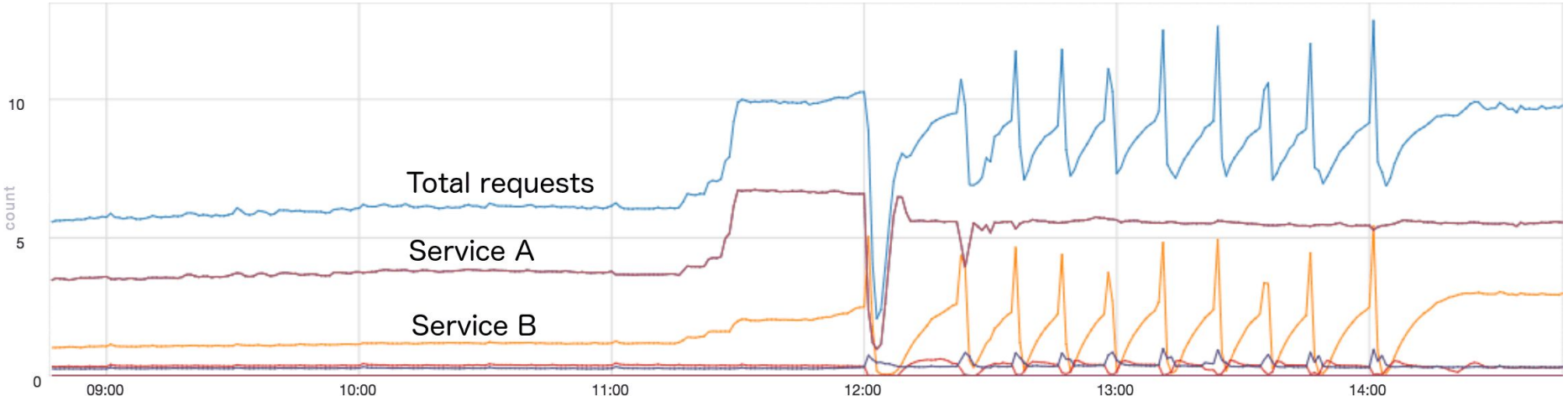
優雅降級 (Graceful Degradation)

- 應用若允許，也可以考慮優雅降級
  - 應視為罕見狀況，常觸發時須報警
  - 這段理論上應該不常執行到，需注意要多測試及演練
- 規劃優雅降級的開關條件
  - 不要設定複雜的自動啟動條件
  - 能容易關閉 (分散式系統)

重試 (Retries)

- 在過載時，重試導致惡性循環：過載 → 客戶重試 → 過載更嚴重
  - 就算是負載緩慢增長也會導致
- 在這兩個狀況下尤其嚴重，需要大幅降低上游流量才能解決
  - 需花很多資源處理
  - 本來就不穩定

Naive Retries Consider Harmful



http://www.evanjones.ca/retries-considered-harmful.html

如何處理重試

- 前述避免過載的策略
- Exponential Backoff
- Jitter
- 限制重試次數
- Retry Budget @ Server
- 上游的重試要更加謹慎，因為會在下游放大
- 某些錯誤不需重試 (ex: 400)
- 監控重試的比率

Restart policy

kubernetes

Containers that are restarted by the kubelet are restarted with an exponential back-off delay (10s, 20s, 40s …) capped at five minutes, and is reset after ten minutes of successful execution. As discussed

GRPC

Deadline

- 由客戶設定
- 不要做客戶不會用到的東西 ⇒ 不要把運算時間花在已知會超過客戶 Deadline 的工作上
- 對 Server 來說
  - 太短的 Deadline → 敏感，放棄有用的半成品，重工
  - 太長的 (或沒有) Deadline→ 不必要的鎖住資源，做虛功

- Deadline 需考量網路延遲
- 對一些不重要的請求額外設定 deadline
- Deadline 有時候非強制性，例如當運算很昂貴時

Deadline & Cancellation Propagation

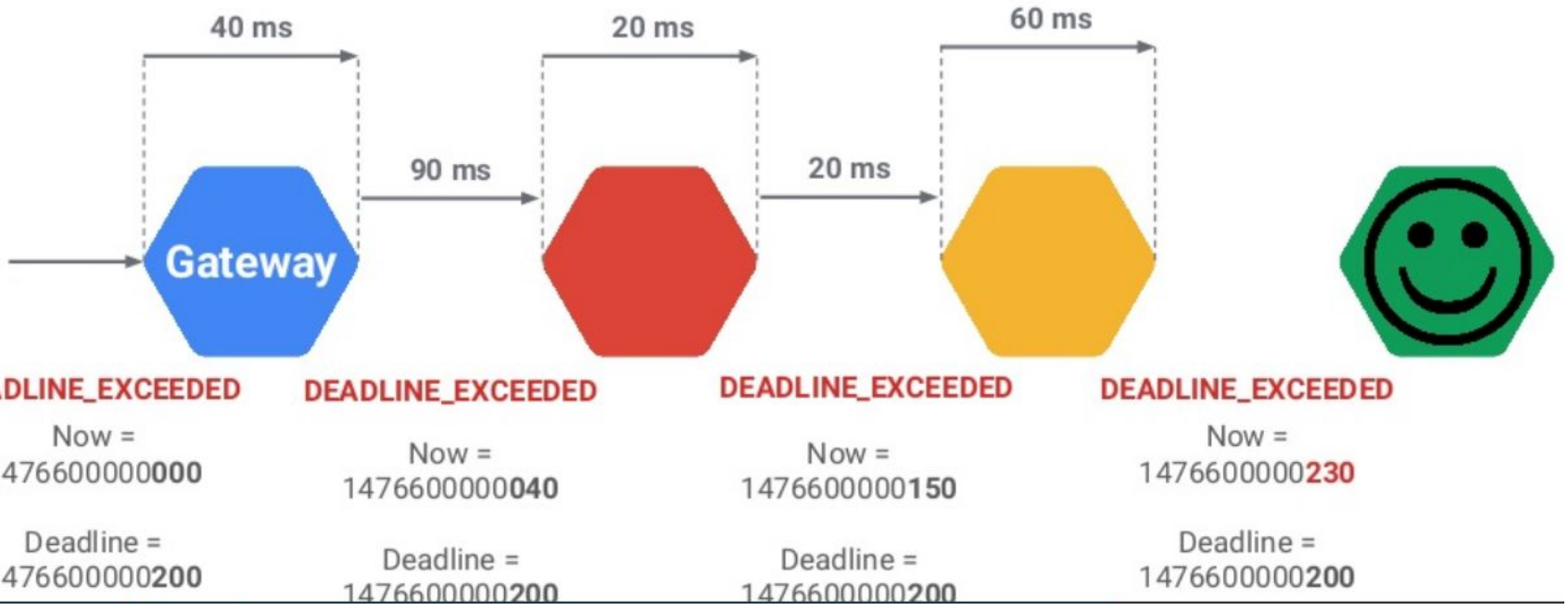
https://www.slideshare.net/borisovalex/enabling-googley-microservices-with-grpc-at-devoxx-france-2017

p.60-86

gRPC Deadline Propagation

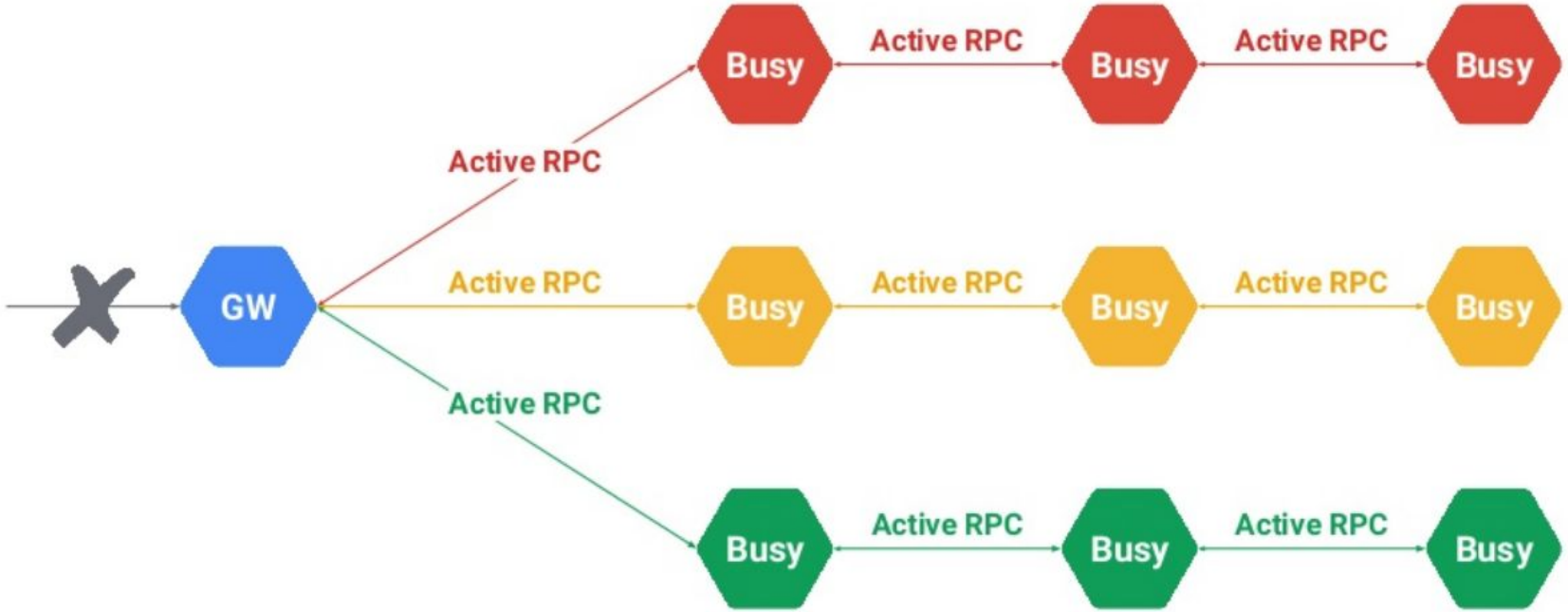
Source: https://www.slideshare.net/borisovalex/enabling-googley-microservices-with-http2-and-grpc

```
withDeadlineAfter(200, MILLISECONDS)
```



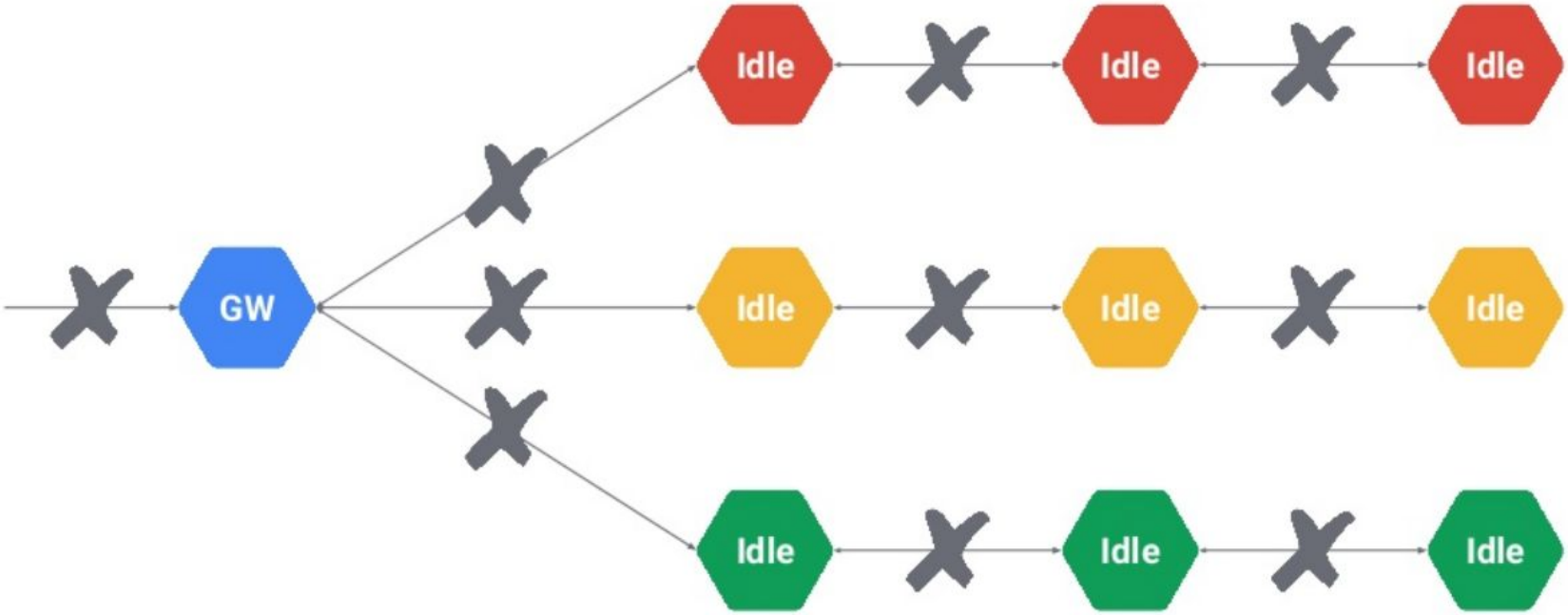
Cancellation?

Source: https://www.slideshare.net/borisovalex/enabling-googley-microservices-with-http2-and-grpc



Cancellation Propagation

Source: https://www.slideshare.net/borisovalex/enabling-googley-microservices-with-http2-and-grpc



Hedged Request needs cancellation propagation

- Hedged Request: 先送一個請求到 Server A，當 Server A 在 p95 時間內未回應時，再送請求至 Server B，當其中一個請求有回覆時，取消另外一個請求

延遲的雙峰分布 Bimodal Latency

- 處理方式是 Thread-per-request (且沒有 Queue)，**1000 QPS**，正常的處理時間是 100 ms，需要 100 個 Thread 滿載處理
- Thread Pool 配置了 1000 個 Thread，所以 Thread Util = 10%
- 假設有 **5% 的請求有問題**，會讓處理的 thread 卡住
- 假設 Deadline 設定為剩下 100 sec (**1000x**)，所以如果 thread 接到有問題的請求，會被卡住 100 sec
- 因為許多 Thread 被卡住，Thread Util 升至 100% (1000 個 Thread)
- Avg Throughput < 1000 / Avg Latency
- Avg Latency = 5% * 100s + 95% * 100ms = 5.095s
- Avg Throughput < 1000 / 5.095 = **196 QPS**
- **超過 80.4% 的請求收到錯誤**

延遲的雙峰分布 Bimodal Latency

	原本 (無問題時)	5% 問題請求, **1000x deadline**
(資源投入) 滿載使用所需的 Thread 數量	100個	1000個
(產出) Throughput	1000 QPS	< 196 QPS
Error Rate	0%	> 80.4%

- 太保守的 Deadline，會在發生問題時佔用太久資源，讓問題更形嚴重

延遲的雙峰分布 Bimodal Latency

	原本 (無問題時)	5% 問題請求, **1000x deadline**	5% 問題請求, **100x deadline**
(資源投入) 滿載使用所需的 Thread 數量	100個	1000個	1000個
(產出) Throughput	1000 QPS	< 196 QPS	950 QPS
Error Rate	0%	> 80.4%	5 %

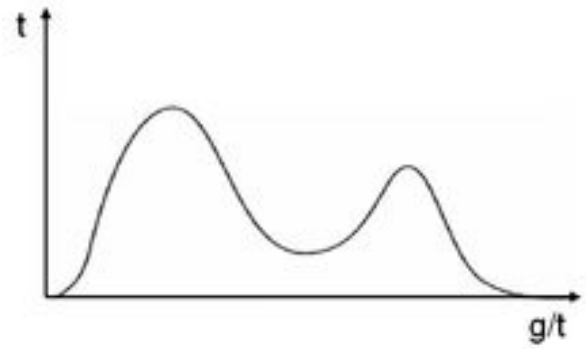
延遲的雙峰分布 Bimodal Latency

900% 的保留產能太多了，實際一點，假設只有 20% 的保留產能

	原本 (無問題時)	5% 問題請求， **1000x deadline**	5% 問題請求， **100x deadline**
(資源投入) 滿載使用所需的 Thread 數量	100個	120個	120個
(產出) Throughput	1000 QPS	< 23 QPS	< 201 QPS
Error Rate	0%	> 99.7%	> 79.9 %

延遲的雙峰分布 Bimodal Latency

- Deadline 設定 不要與 正常回應時間 差距太大
- 若已確定無法完成請求，fail fast!



- 只看平均延遲的話，看不出 Bimodal Latency
- 要看 Latency 的分佈
- 分艙或是對客戶限流，避免資源被少數獨佔

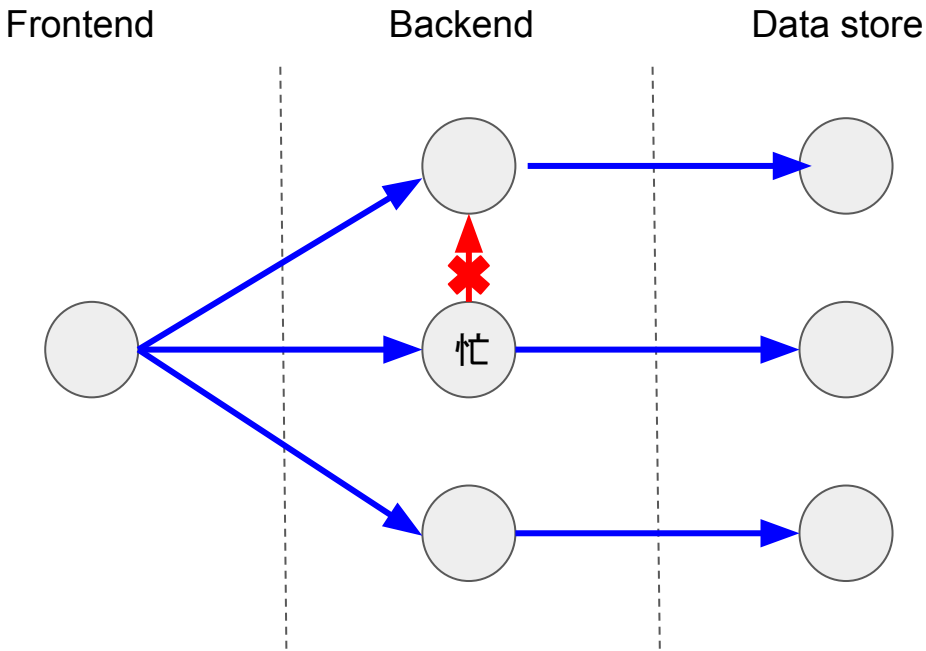
3. 慢啟動與冷快取

- 慢啟動範例
  - 建立連線池
  - 虛擬機執行期優化 (Java)
- 冷快取情境
  - 新增伺服器
  - 伺服器維護後回歸
  - 重啟伺服器
- 避免冷快取
  - 設置獨立的快取叢集, Memcached/Redis cluster, In-Memory Data Grid
  - 會增加一個 RPC Call

慢啟動與冷快取

- 快取的種類
  - Latency Cache：只是用來降低 Latency，Nice-to-have
  - Capacity Cache：用來撐住流量，Must-have
  - Availability Cache
- 如果服務極度依賴快取 (ex: capacity cache)，請考慮
  - 保留服務閒置產能來應付快取失效狀況
  - 採用避免過載的技巧
  - 不要瞬間放流量進來，讓快取有時間因應

永遠往後呼叫，不要在同一層中互相呼叫



- 避免分散式死鎖
- 避免在高負載時加劇負擔
- 讓前端能獲得後端負載資訊 (back-pressure)
- 簡化依賴關係，有利於佈署與調整

4. 連鎖故障的觸發時機

- 行程死亡
- 行程更新
- 換版
- 計畫停機
- 上游請求方式的改變
  - 請求內容
  - 請求頻率
  - LB Policy
- Over-provision 碰上高負載
- 自然增長

5. 針對連鎖故障的測試方式

- 目的
  - 了解：整體系統崩潰臨界點？哪些是脆弱的服務瓶頸？崩潰的症狀？
  - 崩潰下的反應：是否能成功降級？回復的臨界點？資料寫入是否會錯亂？
  - 回復到正常狀況下的反應：是否能正常恢復？
- 壓測
  - 突來流量 和 漸增流量 都要測
- 故障測試
  - Chaos Enginnering：隨機關掉 Server & Region
  - 有信心的話，在生產環境實施小規模故障測試

針對連鎖故障的測試方式

- 了解大客戶的請求方式
- 確保不會被非關鍵性的下游服務影響

6. 連鎖故障的立即處置

- 增加運算資源
  - 有時候會無效

- 暫時停用 Health Check
  - 正常狀況下可行的Health Check, 可能在連鎖故障時會過度嚴苛
  - 分辨兩種 Health Check
    - Liveness: 確認已進入啟動狀態 (for Cluster Management)
    - Readiness: 確認可提供服務 (for Load Balancer)

連鎖故障的立即處置

- 重啟伺服器
  - 在這些狀況下有用
    - GC 的死亡螺旋
    - 資源被一些沒有 Deadline 的請求持續佔用
    - 被 Deadlock 鎖死
  - 但要注意
    - 在啟動過程中，會加重其他伺服器的負擔，可能引發另一波連鎖故障
    - 如果根因是 Cold Cache，重啟反而雪上加霜
  - 執行時
    - Canary this change, and make it slowly

連鎖故障的立即處置

- Drop Traffic
  - 大絕
  - 掐到目前能處理的流量 → 讓大部分伺服器回復 → 讓更多流量進來
  - 但若根因沒有解決，只能治標
- 進入降級模式
- 停用批次作業
- 阻擋造成問題的請求

Example

Summary of the October 22, 2012 AWS Service Event in the US-East Region

- 症狀：10am, 出現 EBS performance degrade or get stuck → 11am, 蔓延到 AZ 中的大部分 Storage
- 處置：
  - 11:10 drop traffic
  - 11:35 自動回復開始運作
  - 13:40 回復 60%, 但尚未找到原因
  - 15:10 找到根因
  - 16:15 幾乎復原
- 問題分析：
  - EBS Storage Server 上有 data collection agent, 雖然重要但不需即時處理、且可以容忍不齊全的資料。之前幾天, 有一台 Data Collection Server 被置換, 對應到的 DNS Record 也被調整, 但沒有同步到所有下游的 DNS Server, 因此有些 EBS Storage Server 會持續找已被置換掉的那台 Data Collection Server
  - 在 Storage Server 上還奘有 Report Agent, 找不到 Data Collection Server 的動作觸發了 Report Agent 的一個 bug, 導致 Report Agent 的 Memory Leak
  - Memory Leak 過多, 影響到正常服務
  - 因為是監控整個 EBS Storage Server 的 Memory 用量, 所以沒有發現 Report Agent 的 Memory Leak
- 困難：
  - 大量的服務器重啟, 影響到問題的追查

https://aws.amazon.com/tw/message/680342/

7. 結論

- 事先了解
  - 系統的臨界點
  - 系統在連鎖故障時的症狀
  - 哪些服務可能是問題根源
- 預防
  - 快速失敗
  - 限流
  - 降級方案
  - 適當的 Deadline
  - 容量規劃
- 治療