Circuit breaker mechanism for Microservices

Kunal Grover

This isn’t something new right?

- Netflix Hystrix
- https://martinfowler.com/bliki/CircuitBreaker.html

Pop Quiz

You have 20 microservices in your single endpoint application. You monitor P99 numbers for each microservice. What is the user side number you are effectively monitoring?

Basic principles for design with microservices

!! DESIGN FOR FAILURES !!

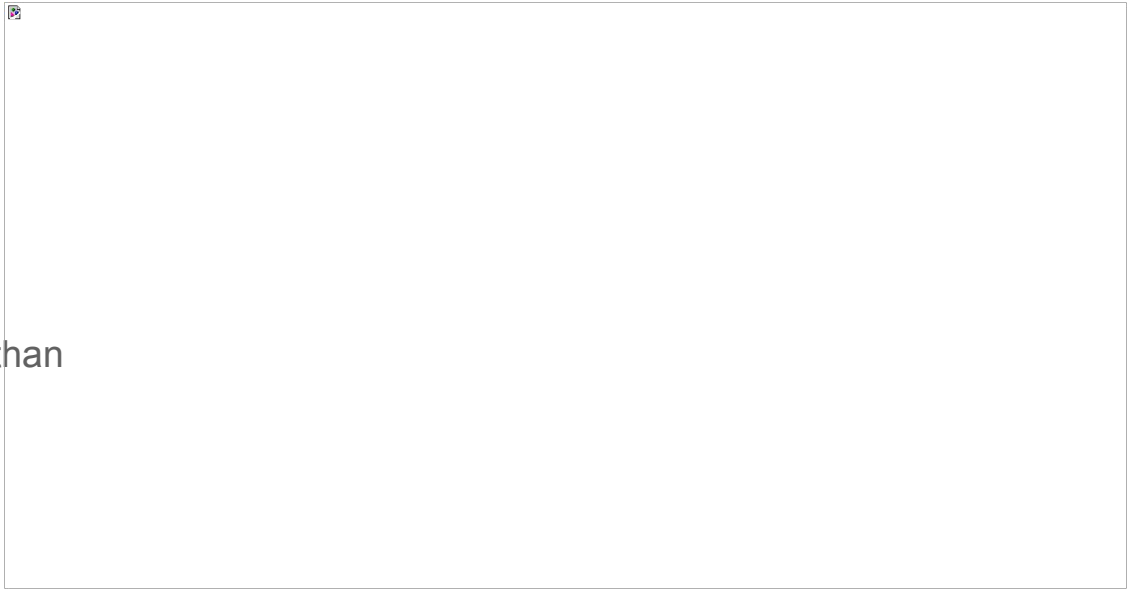
- Don’t fail if your microservice goes down
- Application should not forever wait for microservice
- Contain and Isolate failures(*)
- Respect the service when it is slow(*)
- Fail fast - Recover fast(*)

Typical multi-threaded concurrent webapp

Issues with the architecture

- Service latency average = 100ms
- Timeout = 1s, Retry = 1
- RPS: 100
- 100 application threads

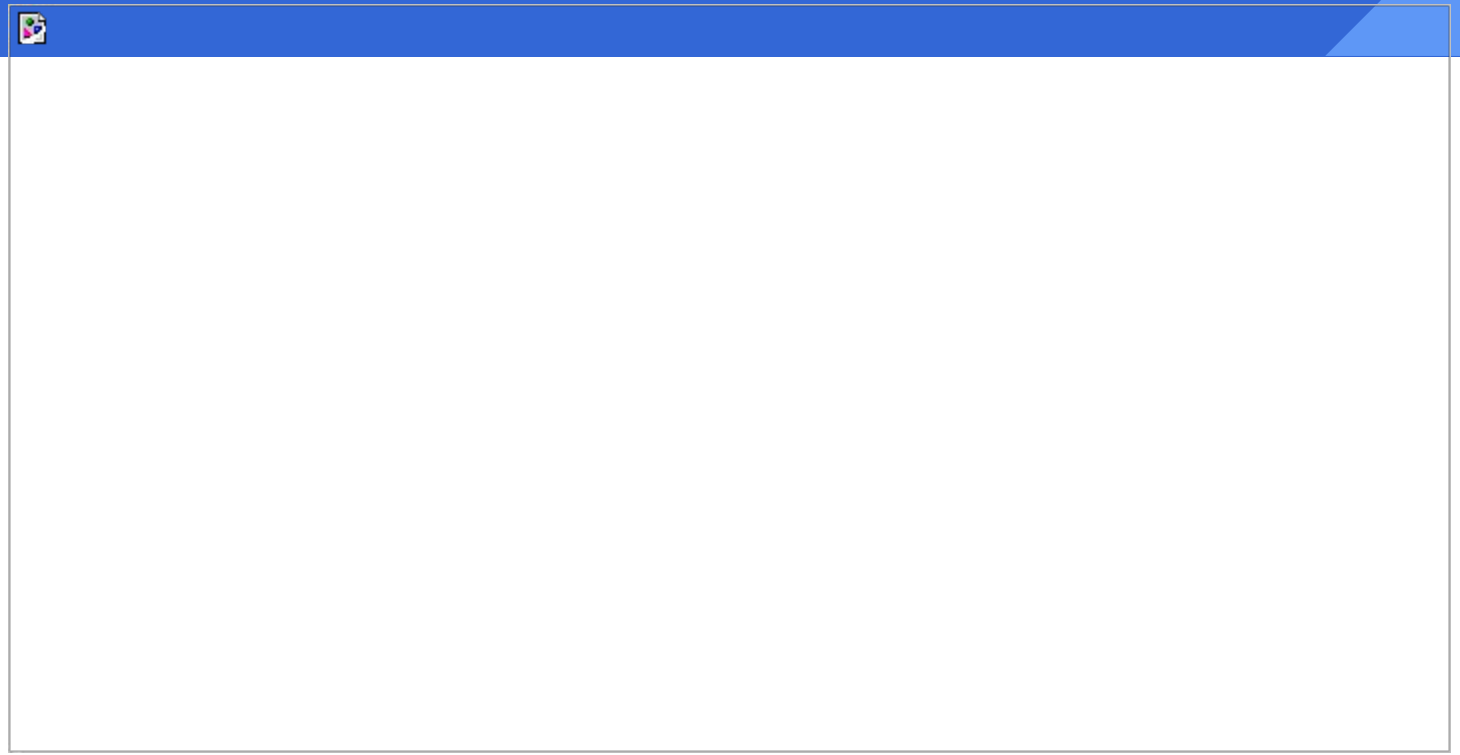
- Average latency from server? More than 2 second
- Requests received by dependent service? 100RPS
- RPS served: 50



Issues with the architecture

- Contain and Isolate failures
  - One service failure can consume entire thread pool
  - All requests get affected
- Respect the service when it is slow
  - You are bombarding your service which is already down
- Fail fast - Recover fast
  - We fail slow after each request
  - Might never recover due to backpressure on the application

Failure resilience



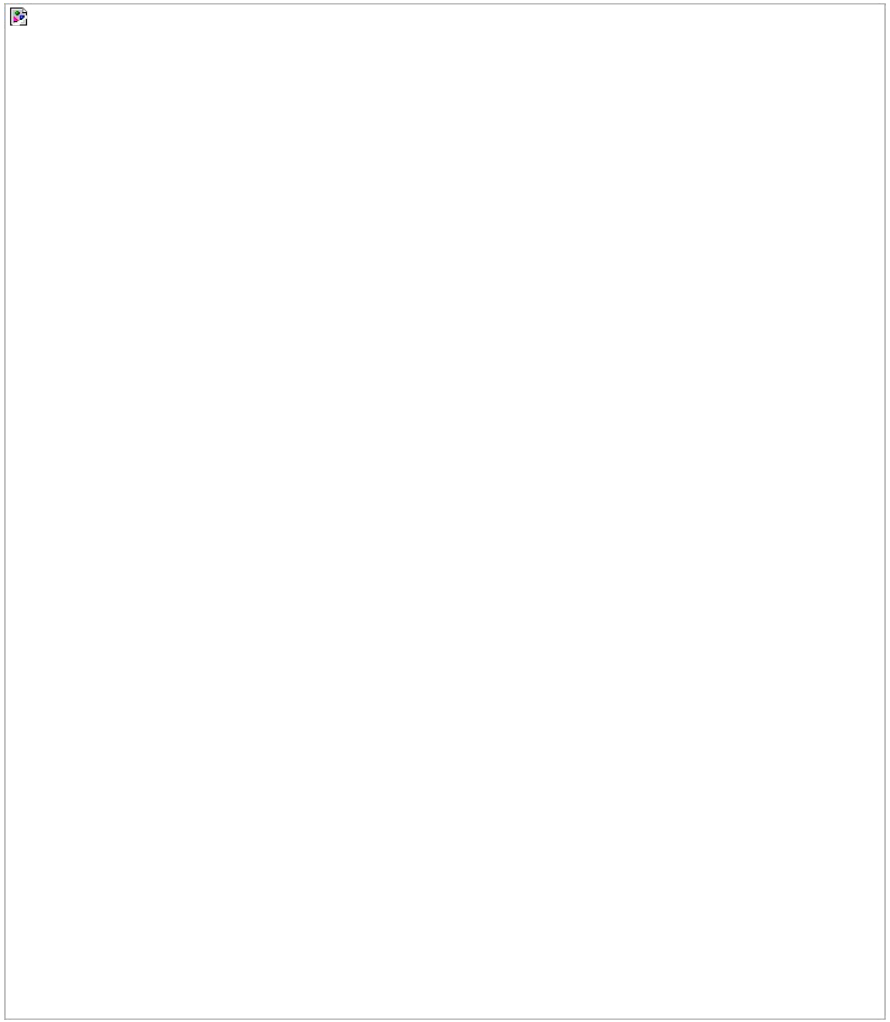
- Threads? 100 * 0.1 = 10 threads.
- Queue size: Let's start with 10

Case: Service is down.

- **How many requests get affected?**
  10 per 2 seconds (served after 2 sec)
- **How fast do we recover?**
  Maximum 2 seconds of service being up!

Can we do better?

- CompletableFuture timeouts run independant from thread timeouts
  Smaller application timeouts!



Coding it

- Timeouts coming up in Java9, doing timeouts with Java8 is tricky: https://crondev.wordpress.com/2017/01/23/timeouts-with-java-8-completablefuture-youre-probably-doing-it-wrong/
- Retries should be simple: https://gist.github.com/kunalgrover05/b960643679a4417eac0db240d8b6f352

Coding it

```
interface RetryRunner {
    CompletableFuture timeoutWrapper(CompletableFuture future);
    CompletableFuture retryWrapper(CompletableFuture future);
    void runAsync(CompletableFuture future);
}

class CircuitBreaker implements RetryRunner {
    CircuitBreaker(Executor executor);
    ... Implement other methods
}
```

Coding it

Timeouts:

```
CompletableFuture timeoutWrapper(CompletableFuture task) {
    timeoutFuture = new CompletableFuture();
    scheduler.schedule(timeoutFuture.completeExceptionally(new
    TimeoutException()), 100, TimeUnit.MILLIS);
    return CompletableFuture.anyOf(task, timeoutFuture);
}
```

Retries:

```
CompletableFuture retryWrapper(CompletableFuture task) {
    return task.thenCompose(/** another future **/);
}
```

Testing it!

- IPTables FTW!
- Drop all incoming packets from your microservices, see how your application behaves.
- Monitor thread pools, number of failures, number of retries

Monitoring ideas

- Thread Pools: Active threads/Queue size
- Percentage of rejections happened? Determine your entire customer experience
- P99 number of retries? If you are retrying more than 1% of times, you might be over burdening the service.

Caveats

- Logging is hard
- Exception handling