September 15th, 2016

Finagle + Java

A love story (✿ ♥‿♥)

@mnnakamura

hi, I’m Moses Nakamura

Twitter lives on the JVM

When Twitter realized we couldn’t stay on a Rails monolith and continue to scale at the rate we were scaling, we decided to make the leap to the JVM.

Since that time, we haven’t looked back. The open source community, the runtime, the standard library, the tools, the stability, and the performance have been excellent since our switch.

Scaling an Organization

Asynchronous Everything*

Performance

Twitter Couldn’t Scale

- Teams getting in each other’s way
- Debugging scaling problems
- Debugging in general
- Low cap for maximum throughput
- Meh latency at scale

Half of these were architectural.
The other half were runtime.

Architectural Problems

Microservices

The problems we got were less bad than what we started with¯_(ツ)_/¯

Teams getting in each other’s way

- Managing deploys in a monolith
- Cross-team mutable state
- Cognitive load from fast-moving abstractions

~~Teams getting in each other's way~~

- Each team controls their own destiny
- Your team owns their mutable state
- You own your abstractions (except for shared library abstractions) and negotiate APIs with callers and receivers

Debugging scaling problems

- Hard to know which team should work on a throughput degradation
- Tragedy of the commons
- If one team gets capacity, every team gets capacity

~~Debugging scaling problems~~

- If your service breaks, it’s your fault
- The buck stops with you
- If another team gets capacity, you still don’t have extra leeway

Debugging in general

- Logs are hard to use
- Too many changes per-deploy
- Inconsistent tracing
- Inconsistent metrics

~~Debugging in general~~

- Your logs are only for your service
- Deploys usually only have a handful of changes, and you understand them all (except for shared libraries)
- All services use a shared library so tracing is uniform
- All services use a shared library so metrics are uniform

New (but different!!) Problems

- Small pockets of weird
- Moves work to library owners
- Network traffic
- Duplicated code
- Protocol migration
- Security
- Reliability

Runtime Problems

JVM

Pretty much a strict improvement

Low cap for maximum throughput

- Profiling is hard, and not that useful
- Synchronous networking model
- Single-threaded
- Rails is not geared for scale

~~Low cap for maximum throughput~~

- Great profiling tools
- Asynchronous network model (hi netty!)
- Multi-threaded
- Java has many users who care about throughput

Meh latency at scale

- Difficult to parallelize network requests
- Ruby’s garbage collector wasn’t great
- Concurrency model not that fine-grained
- Lots of magic
- Ruby is interpreted

~~Meh latency at scale~~

- Asynchrony makes it easy to parallelize requests
- Rock-solid garbage collectors
- Java Memory Model
- Less magic*
- Java is compiled, and JITted

Type Safety

Doesn’t really fit in anywhere but it’s increasingly useful

Scaling an Organization

Asynchronous Everything*

Performance

Your Server as a Function*

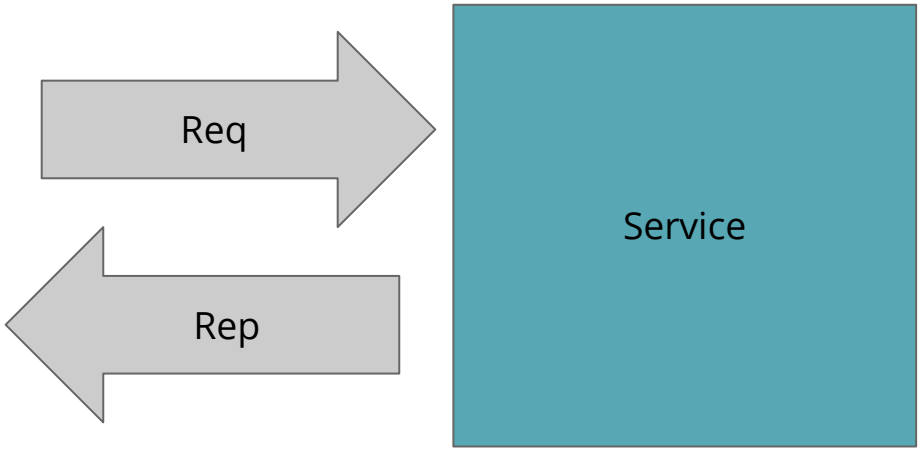
- Futures
- Services
- Filters

Futures

- Similar to CompletableFutures
- Represents an asynchronous value
- Stack-safe, efficient
- Models failures explicitly

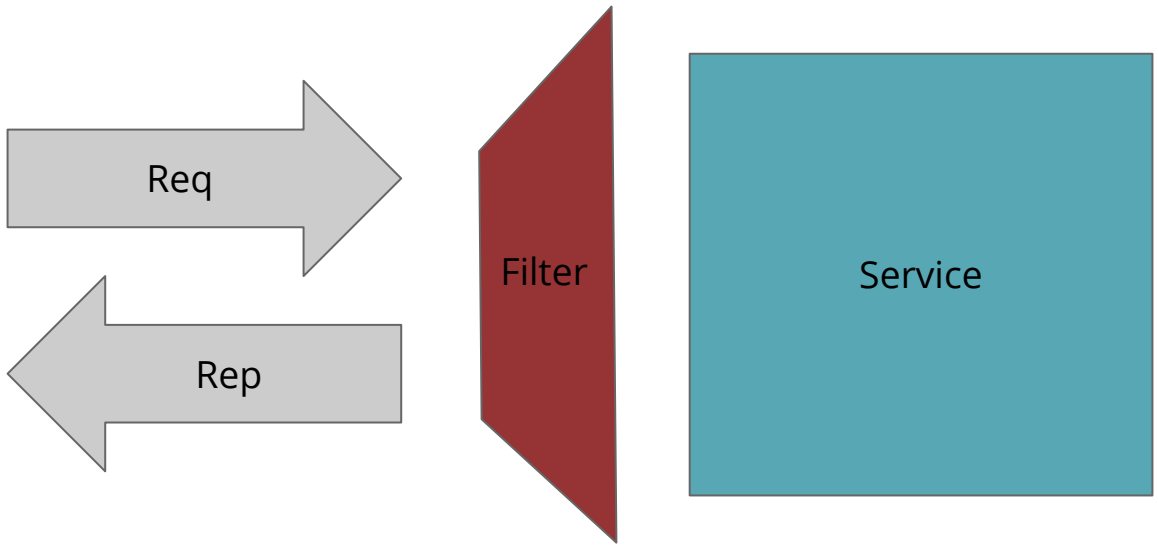
Services

- Req => Future<Rep> + Closable
- Represents an async RPC



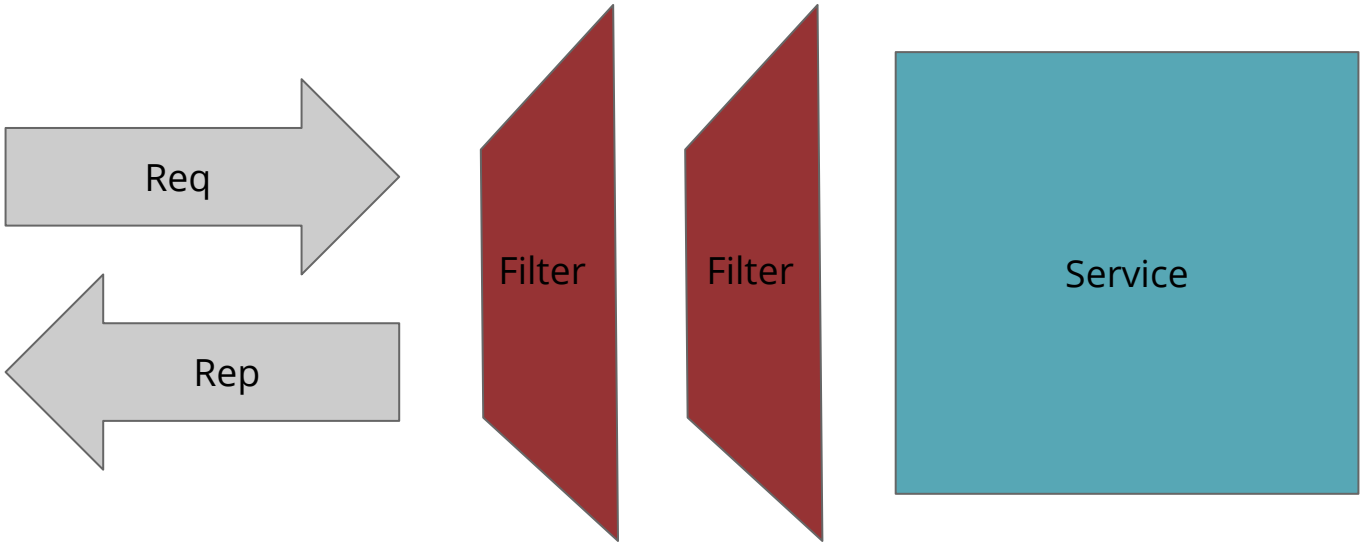
Filters

- (Req1, Service<Req2, Rep2>) => Future<Rep1>
- Lets us wrap and modify services



Filters

- (Req1, Service<Req2, Rep2>) => Future<Rep1>
- Lets us wrap and modify services



Clients

- Services we import off the network
- Load balancing
- Connection pooling
- Naming
- Failure Detection

Servers

- Services we export to the network
- Circuit breakers
- Nacking
- Draining

Asynchronous Programming

- API we expose for concurrent programming
- Most engineers do concurrent programming but don’t need to learn the JMM
- Teach engineers to make data dependencies explicit
- Solves many of the problems with synchronous RPC

Asynchronous Programming Perils

- Breaks profiling
- Stack traces are unreadable
- Closures add GC pressure

Cheap Stack Traces

- Can be used as a cheap way of doing profiling
- We could use it to stitch together a meaningful asynchronous stack trace
- Throwing exceptions gets cheaper

Second Class Functions

- Many functions don’t close over any variables
- We don’t need a full object if egress is 0

Scaling an Organization

Asynchronous Everything*

Performance

Performance wins

- JIT
- GC tuning
- Thread affinity
- Java.util.concurrent
- Jmh
- Excellent profilers
- Eclipse MAT!
- Metrics!

Performance pangs

- Garbage collection throughput
- Garbage collection tail latency
- Asynchronous programming
- Network IO
- Scala

Finer control over memory management

- Choose between stack and heap
- Value objects! (JEP 169)
- Better tools for understanding memory layout
- Better tools for understanding where we’re spending time in GC (Twitter JVM helps)

Network IO

- Synchronization in stdlib network API
- Synchronization in direct buffers
- Too coarse control over off-heap memory (eg even off-heap ends up managed by GC)

questions?