Concurrency Anti-patterns

Blake Matheny
@bmatheny

www.tumblr.com/jobs

Tumblr Basics

- ~1.5B HTTP requests/day
- Peak of around 50k requests/second (more on some services)
- Core app in PHP (+memcache, MySQL, Gearman, redis)
- Backend services in Scala (from C), starting in mid 2011
- Scala services primarily use Finagle and Thrift
- Some Play applications using Akka

7 Anti-patterns (distributed system deadly sins)

- Lust - I want all the abstractions
- Gluttony - Appetite for client connections
- Greed - All CPU time all the time
- Sloth - When not to be lazy
- Wrath - Angry process termination
- Envy - That var looks nice, can I have it?
- Pride - Why mess with perfection?

Lust - I want all the abstractions

Issues

- Combining several concurrency abstractions is hard
- Until Scala 2.10, you had at least 4 options
  - Scala futures
  - Akka futures
  - Java futures
  - Twitter futures
- Can lead to hard to spot bugs
- Serious semantic differences in implementations

Code Smell (rediscover)

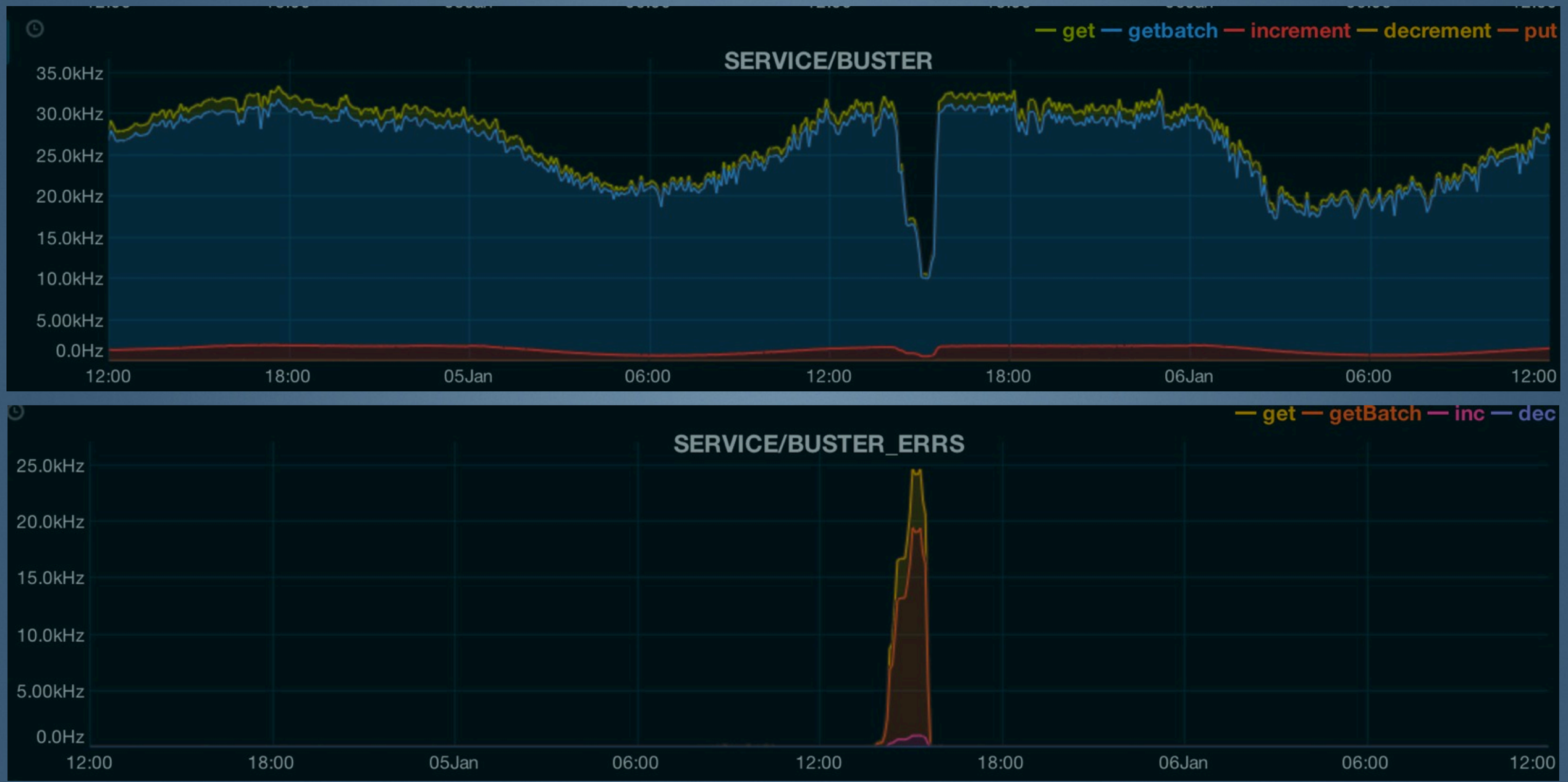
```
object TwitterToAkka {

  implicit def TwitterToAkka[T](f:
TwitterFuture[T])(implicit t: ExecutionContext):
AkkaFuture[T] = {

    val p = AkkaPromise[T]()
    f.onSuccess{v => p.complete(Right(v))}
     .onFailure{err => p.complete(Left(err))}
    p

  }
}
```

Gluttony - Appetite for Clients

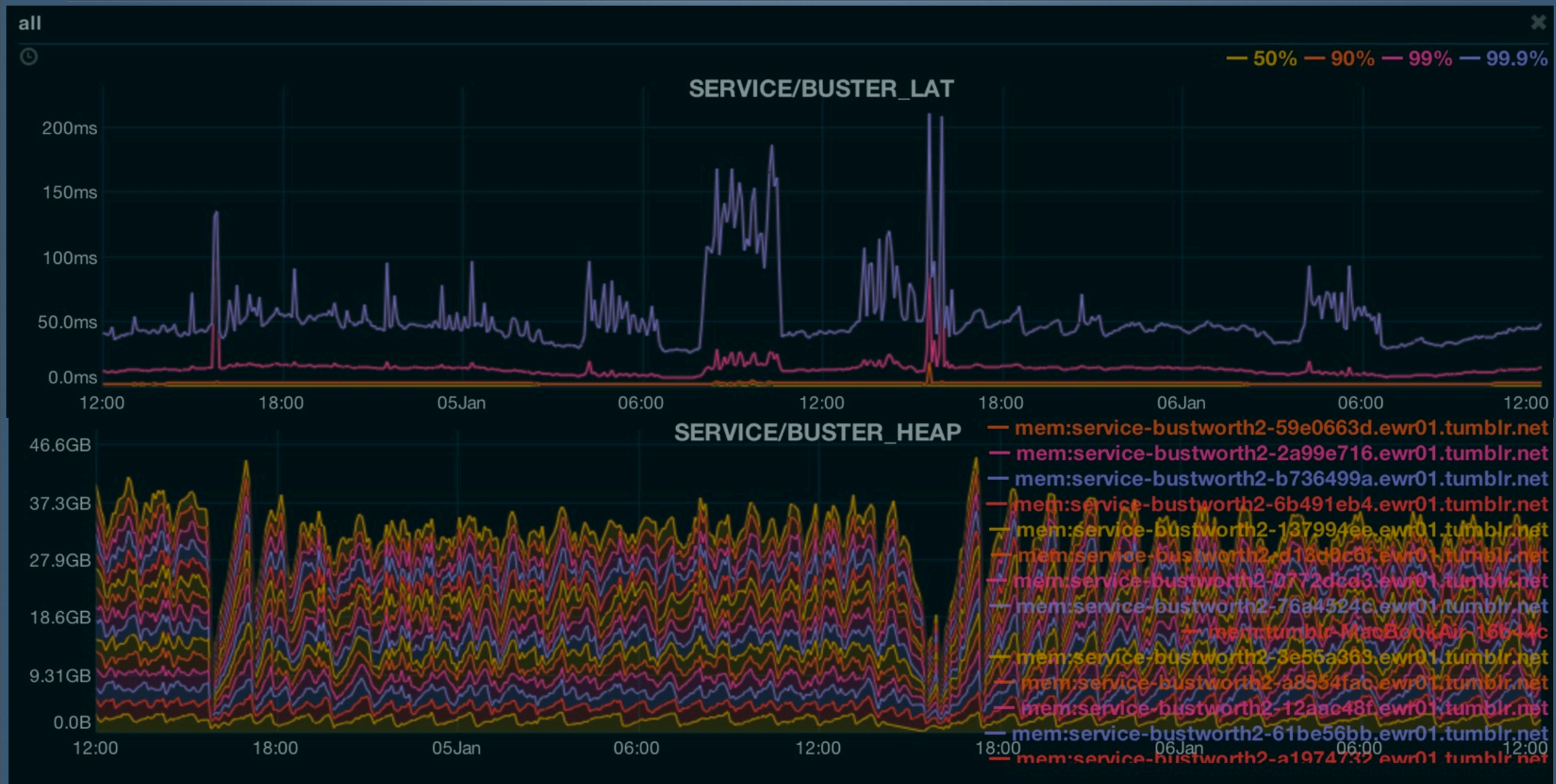
Code Smell (Buster)

```
(code not found)
```

Symptoms of Gluttony



Symptoms of Gluttony



Gluttony - Appetite for Client Connections

Issues

- Constrained by memory, CPU, network
- You can't handle an infinite number of connections
- Would you rather fall over (or reduce performance), or refuse the connection?
- Each abstraction has a different way of specifying how to apply back pressure

Fixes

- Apply back pressure
- Do a tiny bit of capacity planning and set upper bounds
- Load/stress test using both slow clients and slow backends/resources

Greed - All CPU time all the time

Issues

- Blocking in non-blocking code
- Can be harder to spot when mixing abstractions
- Depending on workload you can be limited by number of available CPUs

Fixes

- Use map, foreach, or other abstractions available to you
- Use onSuccess/onFailure with Finagle (good for logging, stats after the fact)

Code Smell (Oscar)

```
val results = Future(Values(reps.map { f =>
  f() match { // same as future.get()
    case Values(values) =>
      values.length match {
        case x if x > 0 =>
          hits += 1
        case _ =>
      }
      values
}.flatten))
```

Sloth - When not to be lazy

Issues

- Improper connection caching
  - Stale connection handles
  - Shutdown and closing connections
- Lazy evaluation in concurrent context
  - Class initialization problems
  - Can end up blocking
  - Performance

Code Smell (service-util)

```
type Client = Service[Command,Reply]

val cache: mutable.ConcurrentMap[String, Client] =
  new ConcurrentHashMap[String, Client]

def apply(host: ClientConfig): Client = {
  cache.getOrElseUpdate(host.hostPort, {
    /* setup a new Client here */
  })
}
```

Code Smell (Collins)

```
Caused by: java.lang.ExceptionInInitializerError: null
```

Wrath - Angry process termination

Issues

- How do you shutdown a process?
- Backend connection handles left open
- Non-daemon threads causing you to kill -9
- Various abstractions handle shutdowns differently

Fixes

- Stop accepting new connections
- Drain any awaiting queued work
- Drain your existing connection queue
- Close backend connections

Code Smell (Wentworth)

```
type Client = Service[StrictKeyCommand, Reply]
type WriteQueue = BlockingQueue[StrictKeyCommand]

class RedisWriter(
  redis: Client, writeQueue: WriteQueue
) extends Runnable {
  def run {
    while (true) {
      drainQueue()
    }
  }
}
```

Envy - That var looks nice, can I have it?

Code Smell

```
// taken from Phillip Haller http://goo.gl/s62qu
var state = 0
def receive = {
  case Request(x) =>
    future {
      handleRequest(x, state)
    }
  case ChangeState(newState) =>
    state = newState
}
```

Envy - That var looks nice, can I have it?

Issues

- Sometimes you need/want mutable state
- You can't completely hide from the Java memory model
- AtomicRef, STM, etc can obscure the obvious

Fixes

```
// taken from Phillip Haller http://goo.gl/s62qu
var state = 0
def receive = {
  case Request(x) =>
    // state safely published
    val currentState = state
    future {
      handleRequest(x, currentState)
    }
  case ChangeState(newState) =>
    state = newState
}
```

Code Smell

```
// taken from Phillip Haller http://goo.gl/s62qu
var state = 0
def receive = {
  case Request(x) =>
    future {
      // the value of state in a closure?
      handleRequest(x, state)
    }
  case ChangeState(newState) =>
    state = newState
}
```

Pride - Why mess with perfection?

Code Smell (refrog)

```
def expire(key: ChannelBuffer): Future[Unit] = {
  if (shouldRandomlyRun()) {
    client.zCount() flatMap { stored =>
      if (stored < count)
        client.zRemRangeByRank().unit
      else
        client.zRemRangeByScore().unit
    }
  } else Future.Unit
}
```

Pride - Why mess with perfection?

Issues

- Production code is imperfect
- Instrumentation provides insight into prod issues
- Again, different abstractions == different methods
- If it's live without instrumentation, it's too late

Fixes

- Ideal if your protocol handlers have built in stats
- Collection, Aggregation, Submission of Stats
- Distributed tracing, dapper style

Code Smell (refrog)

```
def expire(key: ChannelBuffer): Future[Unit] = {
  if (shouldRandomlyRun()) { // does this ever run?
    client.zCount() flatMap { stored =>
      if (stored < count) // Count which op runs
        client.zRemRangeByRank().unit
      else // error handling/counting?
        client.zRemRangeByScore().unit
    }
  } else Future.Unit
}
```

Lessons learned

1. There are many concurrency abstractions available, use the appropriate tool for the job
2. Subtle differences in implementations can make bugs very hard to spot
3. More abstraction, more overhead (time to learn, time to fix, time to run)
4. Avoid blocking, most abstractions provide methods for working with futures
5. Most load/stress tests use fast clients. In reality, client speed has huge variability.

Questions?

Blake Matheny: bmatheny@tumblr.com
www.tumblr.com/jobs