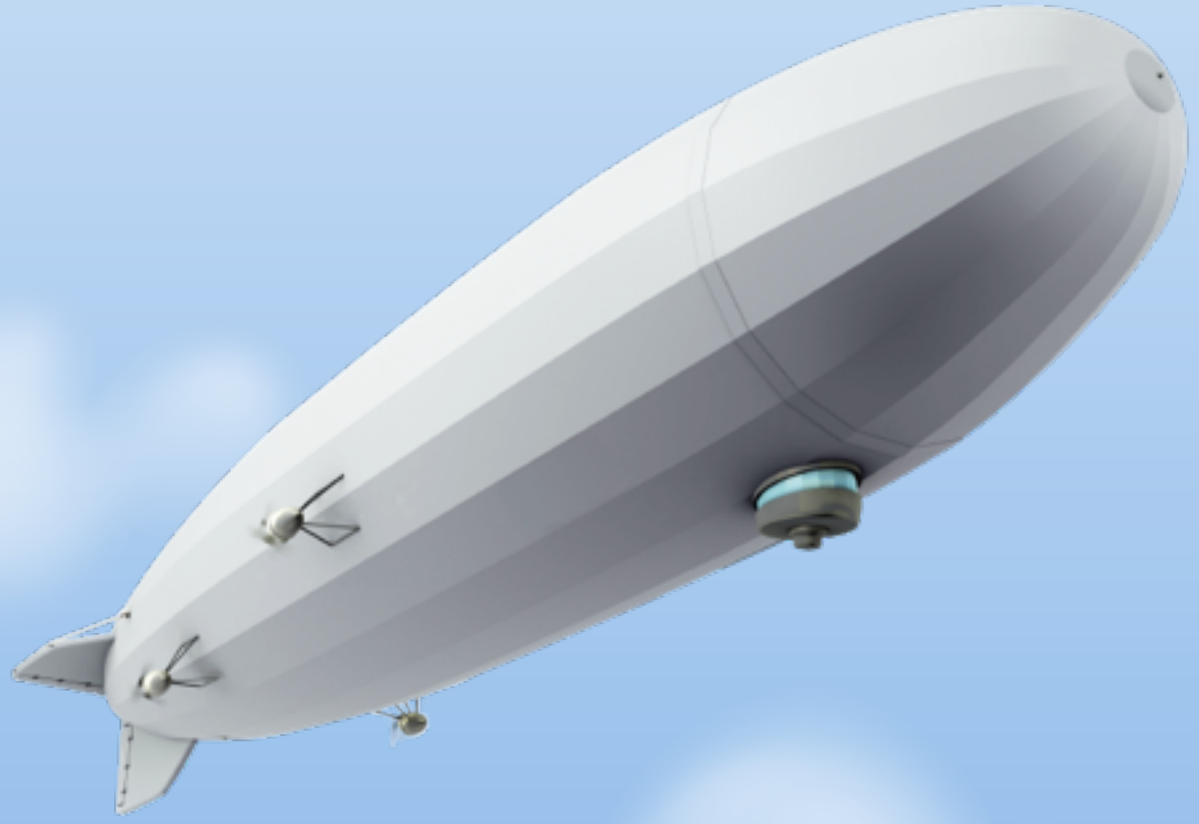
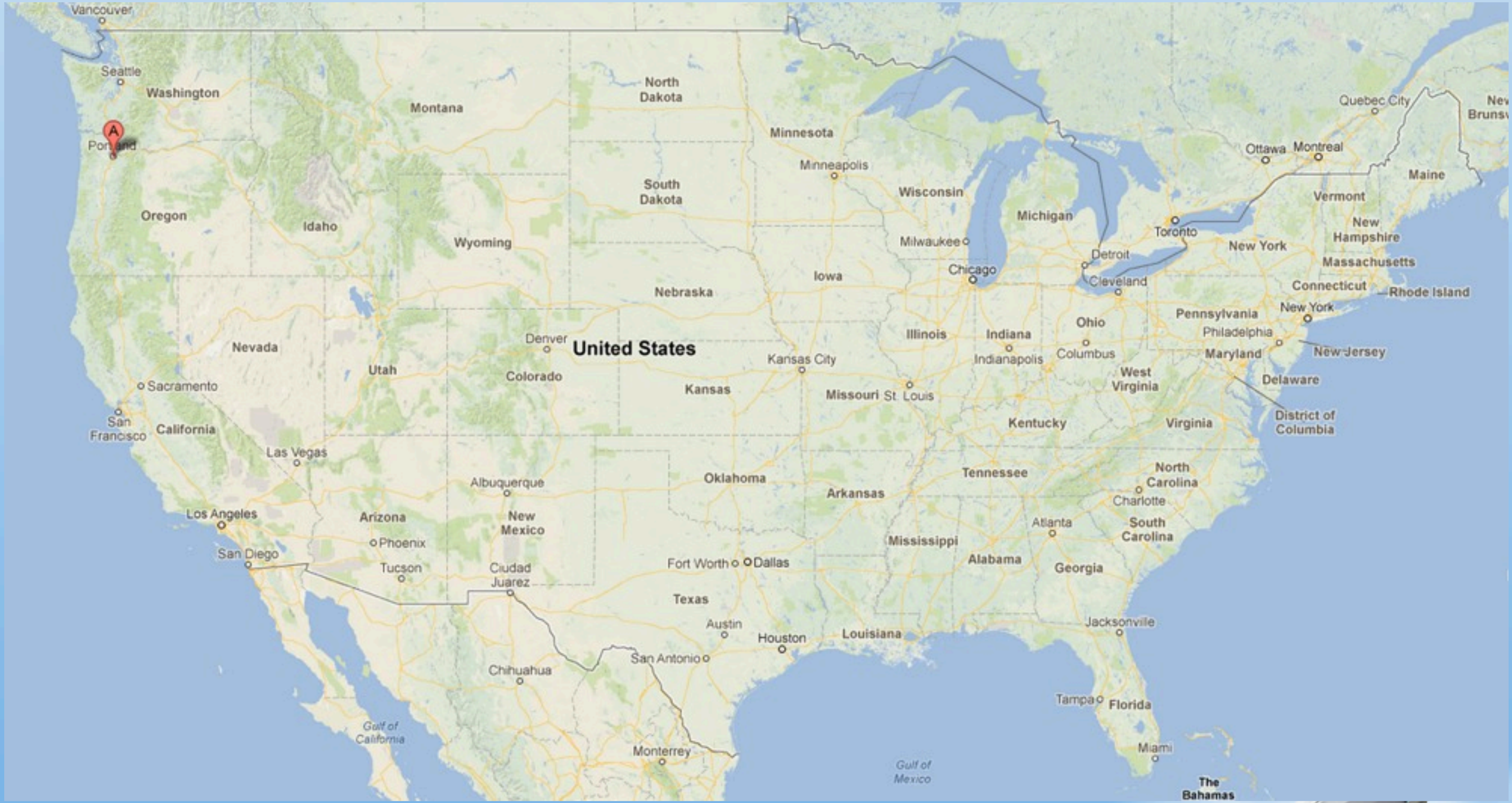


High Performance Network Programming on the JVM

GeeCON, May 2013

Erik Onnen

About Me


Vancouver
Seattle
Washington
A
Portland
Oregon
Idaho
Montana
North Dakota
South Dakota
Wyoming
Minnesota
Minneapolis
Wisconsin
Michigan
Milwaukee
Chicago
Iowa
Nebraska
Nevada
Utah
Denver
United States
Colorado
Kansas
Kansas City
Missouri
St. Louis
Illinois
Indiana
Indianapolis
Ohio
Columbus
Detroit
Cleveland
Toronto
Ottawa
Montreal
Quebec City
New Brunsw
Maine
Vermont
New Hampshire
New York
Massachusetts
Connecticut
Rhode Island
New York
Pennsylvania
Philadelphia
New Jersey
Maryland
Delaware
District of Columbia
West Virginia
Virginia
Kentucky
Sacramento
San Francisco
California
Las Vegas
Albuquerque
Oklahoma
Arkansas
Tennessee
North Carolina
Charlotte
South Carolina
Atlanta
Georgia
Los Angeles
Arizona
Phoenix
New Mexico
San Diego
Tucson
Ciudad Juarez
Fort Worth
Dallas
Mississippi
Alabama
Texas
Austin
Houston
Louisiana
Jacksonville
San Antonio
Chihuahua
Tampa
Florida
Gulf of California
Monterrey
Miami
Gulf of Mexico
The Bahamas

About Me

- Vice President, Architecture at Urban Airship
- Most of my career biased towards performance and scale
- Java, C++, Python in service oriented architectures

In this Talk

- Terminology and Key Theorems
- Foundations for this talk (WTF is an “Urban Airship”?)
- Networked Systems on the JVM
- Choosing a framework
- Critical learnings
- Q&A

Lexicon

What makes something “High Performance”?

Lexicon

What makes something “High Performance”?

- Low Latency - I initiate an action with a service, how long does that take
- Throughput - how many of those operations can I drive through my architecture at one time?
- Scalability - how far can we push one service, how does it fail
- Productivity - how quickly can I create a new operation? A new service?
- Sustainability - when a service breaks, what’s the time to RCA

Key Theorems

Key Theorems

- Programming language can have a material impact on runtime performance - it matters at scale

Key Theorems

- Programming language can have a material impact on runtime performance - it matters at scale
- Writing code is often the easy part of a developer’s job

Key Theorems

- Programming language can have a material impact on runtime performance - it matters at scale
- Writing code is often the easy part of a developer’s job
- Virtualized servers are often the victim of egregious crimes against networking and system throughput (e.g. ec2)

Key Theorems

- Programming language can have a material impact on runtime performance - it matters at scale
- Writing code is often the easy part of a developer’s job
- Virtualized servers are often the victim of egregious crimes against networking and system throughput (e.g. ec2)
- Async I/O for all the things isn’t always the best way to maximize throughput from your servers

Key Theorems

- Programming language can have a material impact on runtime performance - it matters at scale
- Writing code is often the easy part of a developer’s job
- Virtualized servers are often the victim of egregious crimes against networking and system throughput (e.g. ec2)
- Async I/O for all the things isn’t always the best way to maximize throughput from your servers
- Deviations in any of these can lead to more CoGS (bad for startups)

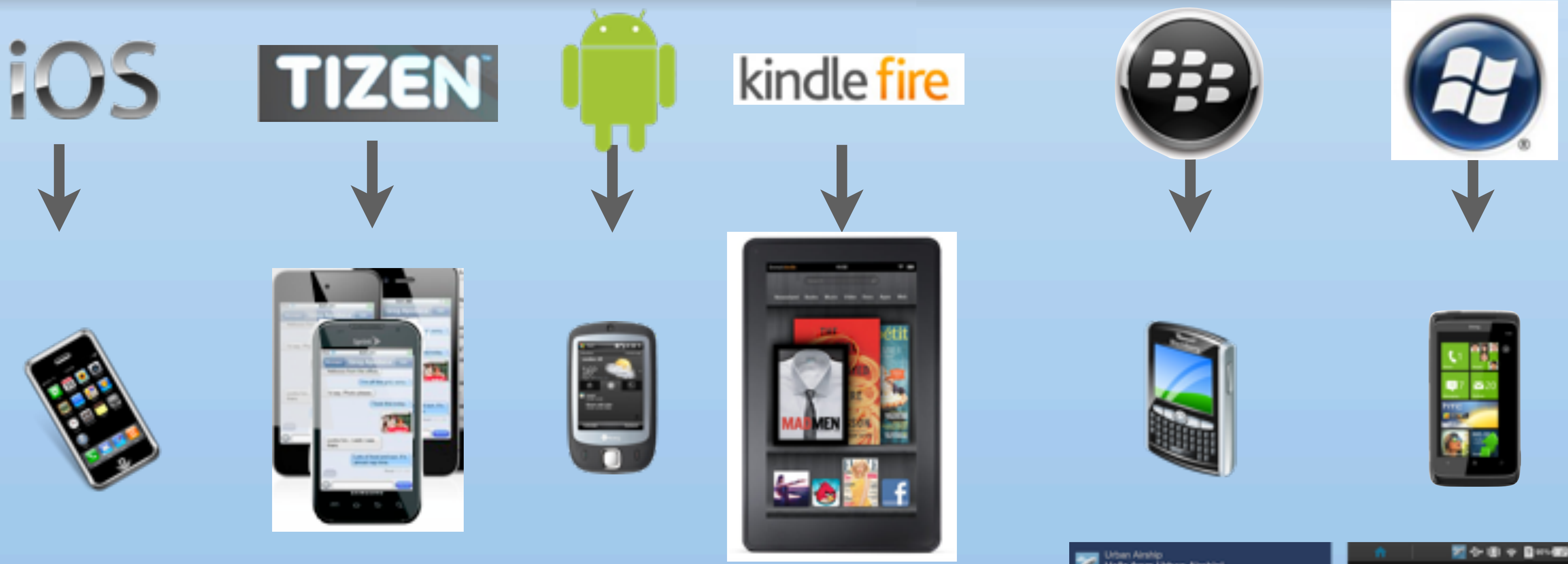
Key Theorems

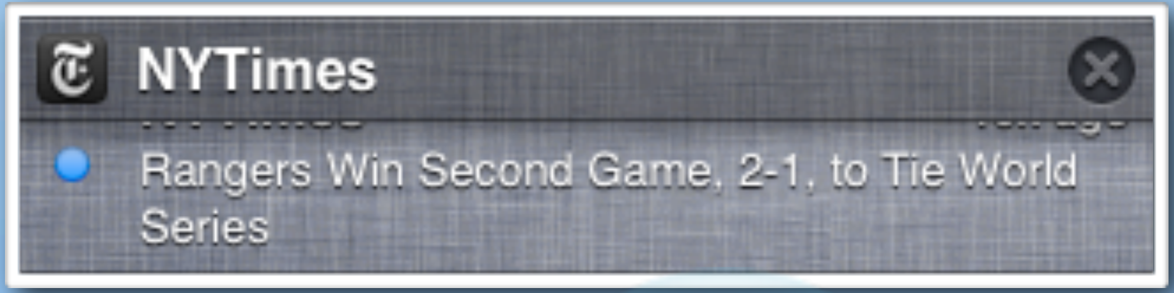
- Programming language can have a material impact on runtime performance - it matters at scale
- Writing code is often the easy part of a developer’s job
- Virtualized servers are often the victim of egregious crimes against networking and system throughput (e.g. ec2)
- Async I/O for all the things isn’t always the best way to maximize throughput from your servers
- Deviations in any of these can lead to more CoGS (bad for startups)
- Mobile makes all of these harder

WTF is an Urban Airship?

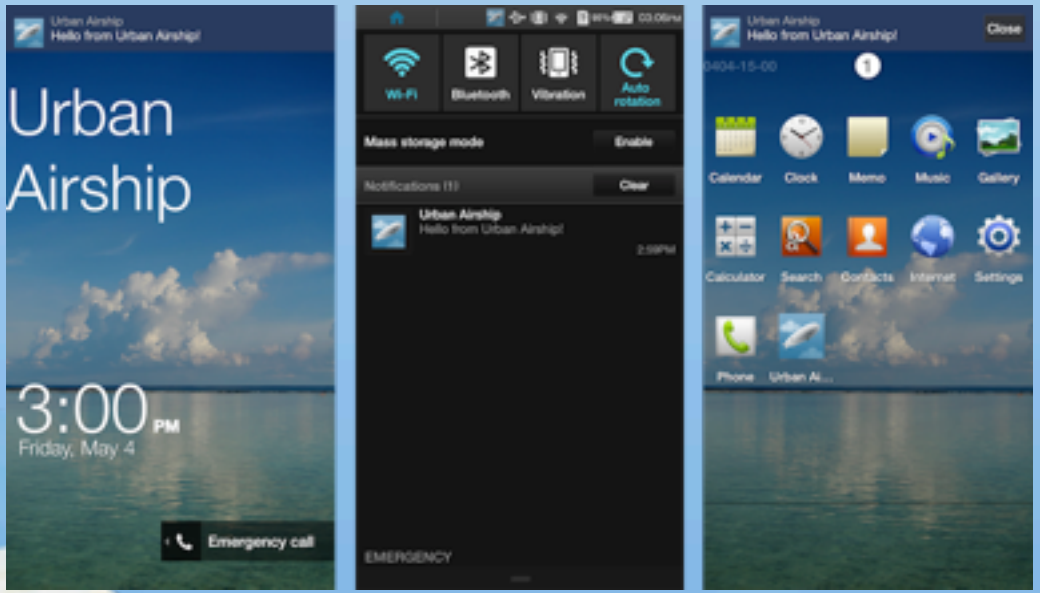
- Fundamentally, an engagement platform
- Buzzword compliant - Cloud Service providing an API for Mobile
- Unified API for services across platforms for messaging, location, content entitlements, digital wallet assets
- SLAs for throughput, latency
- Heavy users and contributors to HBase, ZooKeeper, Cassandra

WTF is an Urban Airship?




NYTimes

Rangers Win Second Game, 2-1, to Tie World Series


Urban Airship
Hello from Urban Airship!

Urban Airship

3:00 PM
Friday, May 4

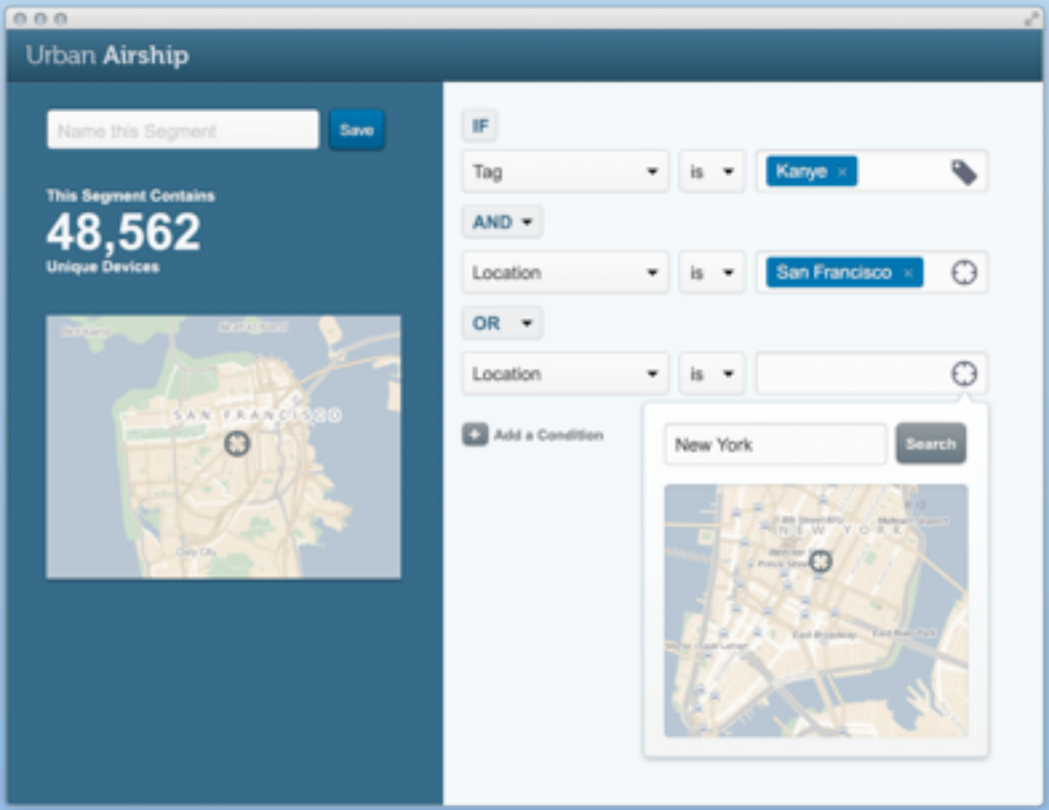
Emergency call

Urban Airship
Hello from Urban Airship!
2:59PM

Urban Airship
Hello from Urban Airship!

What is Push?

- Cost
- Throughput and immediacy
- The platform makes it compelling
  - Push can be intelligent
  - Push can be precisely targeted
- With great power comes great DoS flood



This Segment Contains
48,562
Unique Devices

IF
Tag is Kanye
AND
Location is San Francisco
OR
Location is
New York

How does this relate to the JVM?

- We deal with **lots** of heterogeneous connections from the public network, the vast majority of them are handled by a JVM
- Ingress:
  - 28K HTTPS requests handled every second
  - > 20 million devices connected at any one time
- Internally:
  - Millions of operations per second across our LAN
  - > 20 billion operational metrics a day

Life in Interesting Times

- Fundamentally, SSDs are changing how we think about developing for the JVM
- Similarly, the cost of RAM has made 256GB memory a practical thing but harder to make good use with JVM
- These concerns are not “Big Data”

Distributed Systems on the JVM

- Platform has several tools baked in
  - HTTP Client and Server
  - RMI (Remote Method Invocation) or better JINI
  - CORBA/IIOP
  - JDBC
- Lower level
  - Sockets + streams, channels + buffers
  - Reader/Writer for text
  - Java5 brought NIO which included Async I/O

Distributed Systems on the JVM

- Java 7 brought Asynchronous(Server)SocketChannel
    - Thread pool-backed buffered connect, reads, writes
    - Nicer abstraction than dealing with buffered offsets, spurious wake-up manually
- Fundamentally, the JVM suffers from lowest common denominator problems with the NIO/NIO.2 abstractions

Synchronous vs. Async I/O

Synchronous vs. Async I/O

- Synchronous Network I/O on the JRE
  - Sockets (InputStream, OutputStream)
  - Channels and Buffers
- Asynchronous Network I/O on the JRE
  - Selectors (async)
  - Buffers fed to Channels which are asynchronous
  - Almost all asynchronous APIs are for Socket I/O
- Can operate on direct, off heap buffers
- Offer decent low-level configuration options

Synchronous vs. Async I/O

- Synchronous I/O has many upsides on the JVM
  - Clean streaming - good for moving around really large things
  - Sendfile support for MMap'd files (FileChannel::transferTo)
  - Vectored I/O support
  - No need for additional SSL/TLS abstractions (except for maybe Keystore cruft)
  - No idiomatic impedance for RPC

Synchronous vs. Async I/O

Synchronous vs. Async I/O

- Synchronous I/O - doing it well

Synchronous vs. Async I/O

- Synchronous I/O - doing it well
  - Buffers all the way down (streams, readers, channels)

Synchronous vs. Async I/O

- Synchronous I/O - doing it well
  - Buffers all the way down (streams, readers, channels)
    - Minimize trips across the system boundary

Synchronous vs. Async I/O

- Synchronous I/O - doing it well
  - Buffers all the way down (streams, readers, channels)
    - Minimize trips across the system boundary
    - Minimize copies of data

Synchronous vs. Async I/O

- Synchronous I/O - doing it well
  - Buffers all the way down (streams, readers, channels)
    - Minimize trips across the system boundary
    - Minimize copies of data
    - Vector I/O if possible

Synchronous vs. Async I/O

- Synchronous I/O - doing it well
  - Buffers all the way down (streams, readers, channels)
    - Minimize trips across the system boundary
    - Minimize copies of data
    - Vector I/O if possible
    - MMap if possible

Synchronous vs. Async I/O

- Synchronous I/O - doing it well
  - Buffers all the way down (streams, readers, channels)
    - Minimize trips across the system boundary
    - Minimize copies of data
    - Vector I/O if possible
    - MMap if possible
  - Favor direct ByteBufffers and NIO Channels

Synchronous vs. Async I/O

- Synchronous I/O - doing it well
  - Buffers all the way down (streams, readers, channels)
    - Minimize trips across the system boundary
    - Minimize copies of data
    - Vector I/O if possible
    - MMap if possible
  - Favor direct ByteBufffers and NIO Channels
  - Manage timeout expectations

Synchronous vs. Async I/O

Synchronous vs. Async I/O

- Async I/O
  - On Linux, implemented via epoll as the “Selector” abstraction with async Channels
  - Async Channels feed buffers, you have to tend to fully reading/writing them (addressed in Java 7)
- Async I/O - doing it well
  - Again, favor direct ByteBuffers, especially for large data
  - Consider the application - what do you gain by not waiting for a response?
  - Avoid manual TLS operations

Sync vs. Async - FIGHT!

Async I/O Wins:

Sync vs. Async - FIGHT!

Async I/O Wins:

- Server with large numbers of clients

Sync vs. Async - FIGHT!

Async I/O Wins:

- Server with large numbers of clients
- Only way to be notified if a socket is closed without trying to read it

Sync vs. Async - FIGHT!

Async I/O Wins:

- Server with large numbers of clients
- Only way to be notified if a socket is closed without trying to read it
- Large number of open sockets

Sync vs. Async - FIGHT!

Async I/O Wins:

- Server with large numbers of clients
- Only way to be notified if a socket is closed without trying to read it
- Large number of open sockets
- Lightweight proxying of traffic

Sync vs. Async - FIGHT!

Async I/O Loses:

Sync vs. Async - FIGHT!

Async I/O Loses:

- Context switching, CPU cache pipeline loss can be substantial overhead for simple protocols

Sync vs. Async - FIGHT!

Async I/O Loses:

- Context switching, CPU cache pipeline loss can be substantial overhead for simple protocols
- Not always the best option for raw, full bore throughput

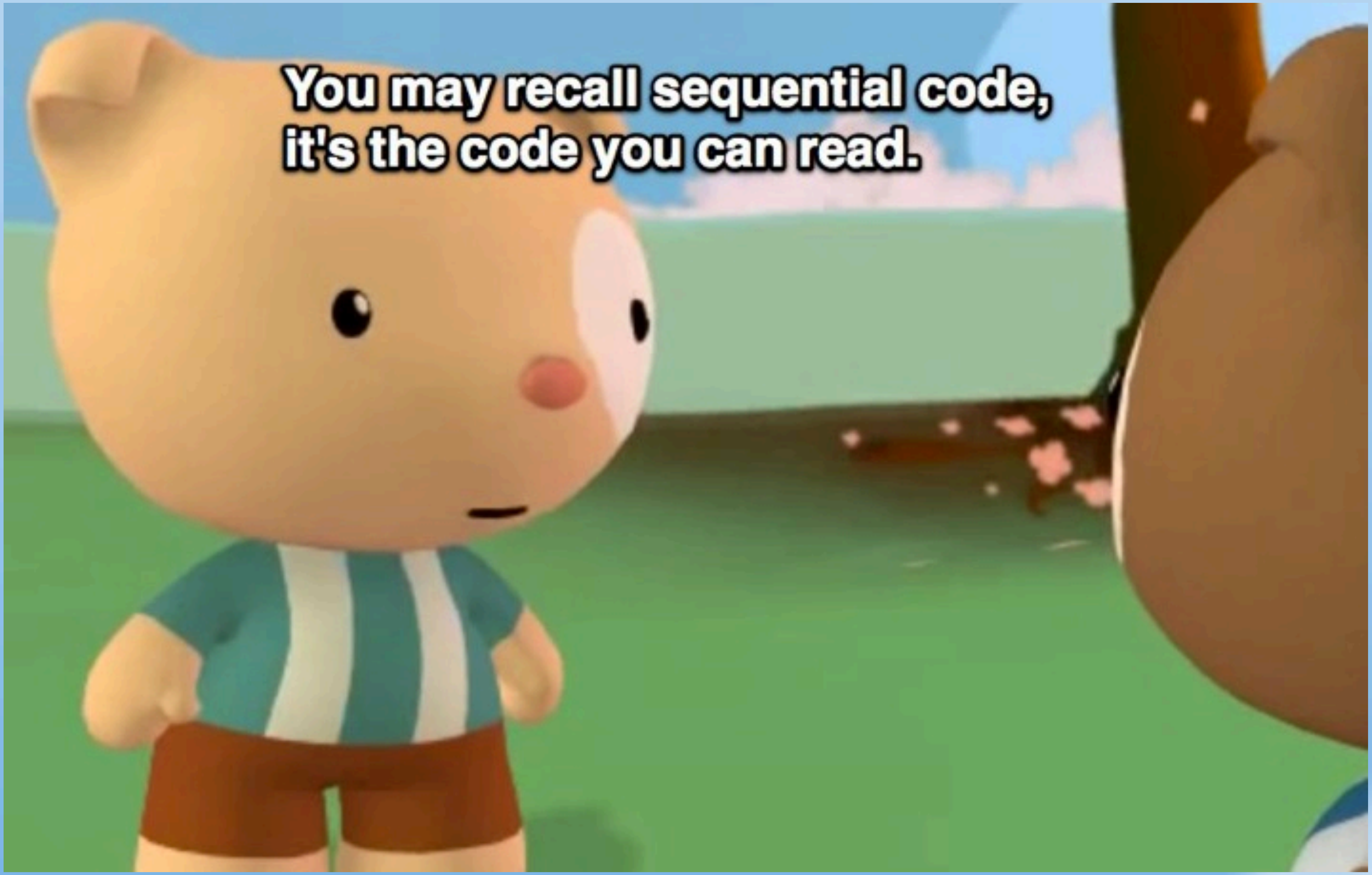
Sync vs. Async - FIGHT!

Async I/O Loses:

- Context switching, CPU cache pipeline loss can be substantial overhead for simple protocols
- Not always the best option for raw, full bore throughput
- Complexity, ability to reason about code diminished

Sync vs. Async - FIGHT!

Async I/O Loses:



http://www.youtube.com/watch?v=bzkRVzciAZg&feature=player_detailpage#t=133s

Sync vs. Async - FIGHT!

Sync I/O Wins:

Sync vs. Async - FIGHT!

Sync I/O Wins:

- Simplicity, readability

Sync vs. Async - FIGHT!

Sync I/O Wins:

- Simplicity, readability
- Better fit for dumb protocols, less impedance for request/reply

Sync vs. Async - FIGHT!

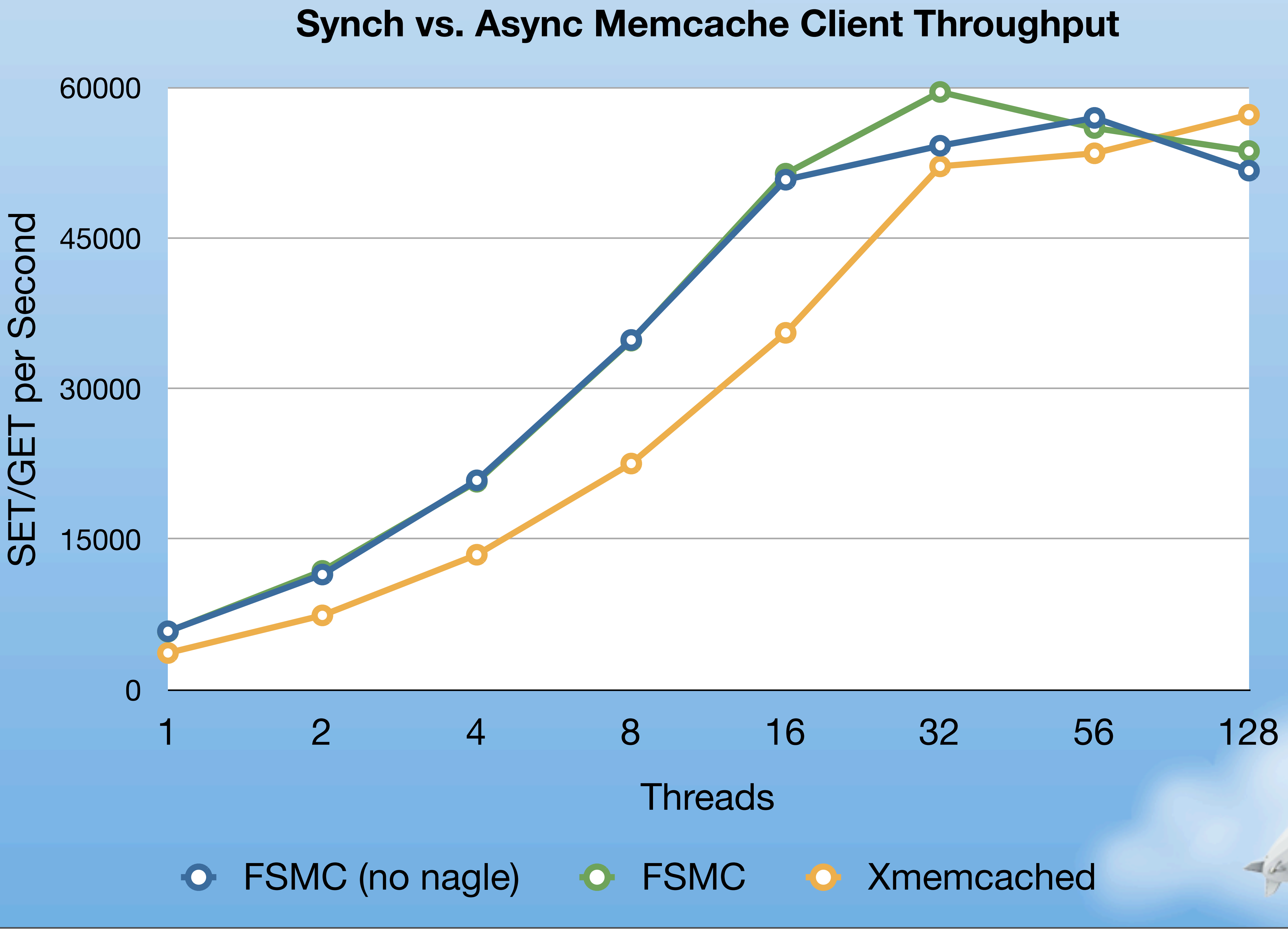
Sync I/O Wins:

- Simplicity, readability
- Better fit for dumb protocols, less impedance for request/reply
- Squeezing every bit of throughput out of a single host, small number of threads

Sync vs. Async - Memcache

- UA uses memcached heavily
- memcached is an awesome example of why choosing Sync vs. Async is hard
- Puts **always** should be completely asynchronous
- Reads are fairly useless when done asynchronously
- Protocol doesn’t lend itself well to Async I/O
- For Java clients, we experimented with Xmemcached but didn’t like its complexity, I/O approach
- Created FSMC (freakin’ simple memcache client)

FSMC vs. Xmemcached



FSMC vs. Xmemcached

```
FSMC:
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.97  143.825726      11811     12177      2596 futex
  0.01    0.014143          0    402289           read
  0.01    0.011088          0    200000           writev
  0.01    0.008087          0    200035           write
  0.00    0.002831          0     33223           mprotect
  0.00    0.001664         12       139           madvise
  0.00    0.000403          1       681           brk
  0.00    0.000381          0      1189           sched_yield
  0.00    0.000000          0       120        59 open
  0.00    0.000000          0        68           close
  0.00    0.000000          0       108        42 stat
  0.00    0.000000          0        59           fstat
  0.00    0.000000          0       124         3 lstat
  0.00    0.000000          0      2248           lseek
  0.00    0.000000          0       210           mmap
```

```
14:37:31,568  INFO [main]
[com.urbanairship.oscon.memcache.FsmcTest] Finished
800000 operations in 12659ms.

real  0m12.881s
user  0m34.430s
sys   0m22.830s
```

```
Xmemcached:
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 54.87  875.668275       4325    202456           epoll_wait
 45.13  720.259447        454   1587899    130432 futex
  0.00    0.020783          3      6290           sched_yield
  0.00    0.011119          0    200253           write
  0.00    0.008682          0    799387         2 epoll_ctl
  0.00    0.003759          0    303004    100027 read
  0.00    0.000066          0      1099           mprotect
  0.00    0.000047          1        81           madvise
  0.00    0.000026          0        92           sched_getaffinity
  0.00    0.000000          0       126        59 open
  0.00    0.000000          0       148           close
  0.00    0.000000          0       109        42 stat
  0.00    0.000000          0        61           fstat
  0.00    0.000000          0       124         3 lstat
  0.00    0.000000          0      2521           lseek
  0.00    0.000000          0       292           mmap
```

```
14:38:09,912  INFO [main]
[com.urbanairship.oscon.memcache.XmemcachedTest]
Finished 800000 operations in 18078ms.

real  0m18.248s
user  0m30.020s
sys   0m16.700s
```

A Word on Garbage Collection

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC
- A good citizen will create lots of ParNew garbage and nothing more

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC
- A good citizen will create lots of ParNew garbage and nothing more
  - Allocation is near free

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC
- A good citizen will create lots of ParNew garbage and nothing more
  - Allocation is near free
  - Collection also near free if you don’t copy anything

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC
- A good citizen will create lots of ParNew garbage and nothing more
  - Allocation is near free
  - Collection also near free if you don’t copy anything
- Don’t buffer large things, stream or chunk

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC
- A good citizen will create lots of ParNew garbage and nothing more
  - Allocation is near free
  - Collection also near free if you don't copy anything
- Don't buffer large things, stream or chunk
- When you must cache:

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC
- A good citizen will create lots of ParNew garbage and nothing more
  - Allocation is near free
  - Collection also near free if you don’t copy anything
- Don’t buffer large things, stream or chunk
- When you must cache:
  - Cache early and don’t touch

A Word on Garbage Collection

- Any JVM service on most hardware has to live with GC
- A good citizen will create lots of ParNew garbage and nothing more
    - Allocation is near free
    - Collection also near free if you don’t copy anything
- Don’t buffer large things, stream or chunk
- When you must cache:
    - Cache early and don’t touch
    - Better, cache off heap or use memcached

A Word on Garbage Collection

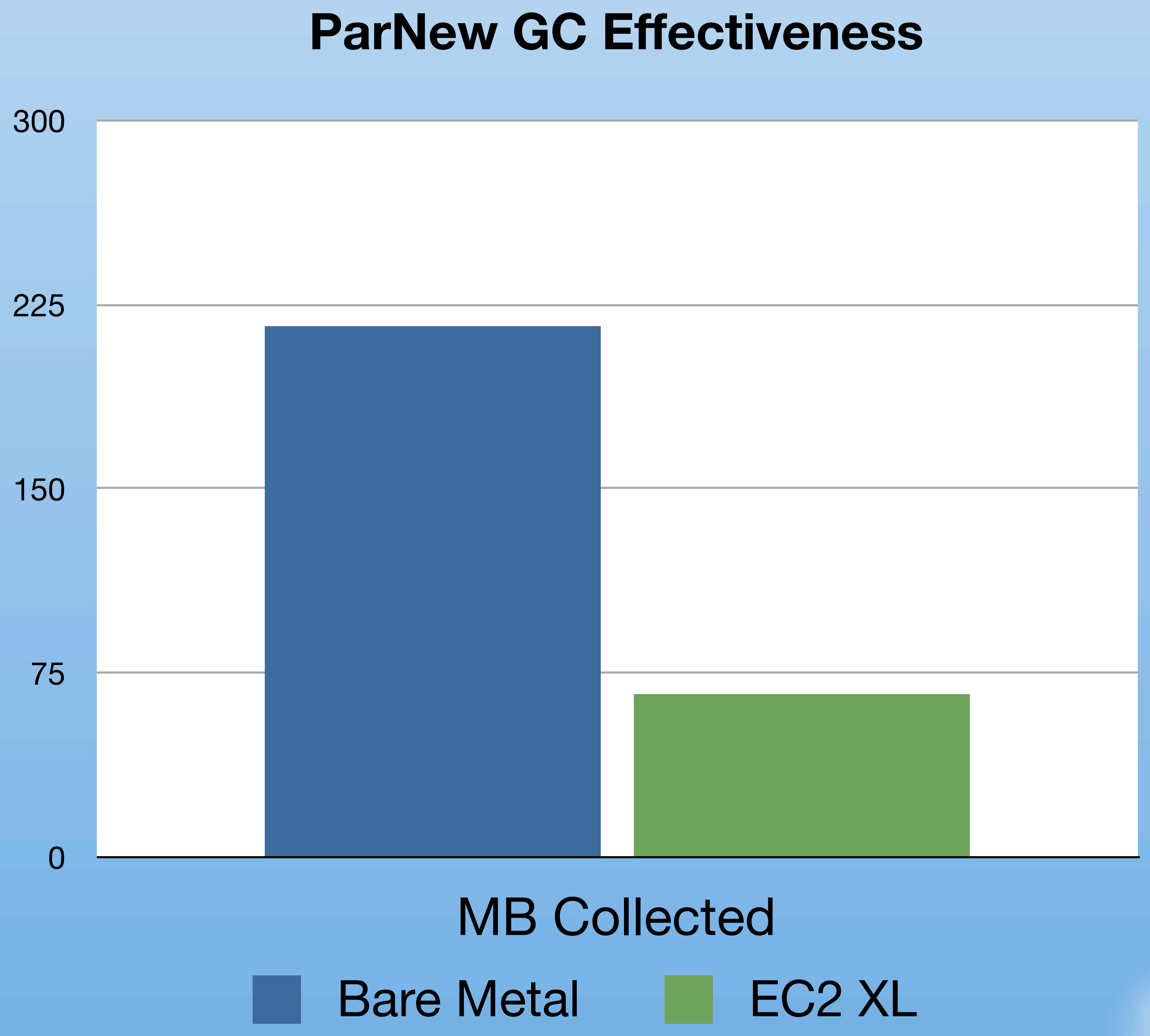
A Word on Garbage Collection

GOOD

A Word on Garbage Collection

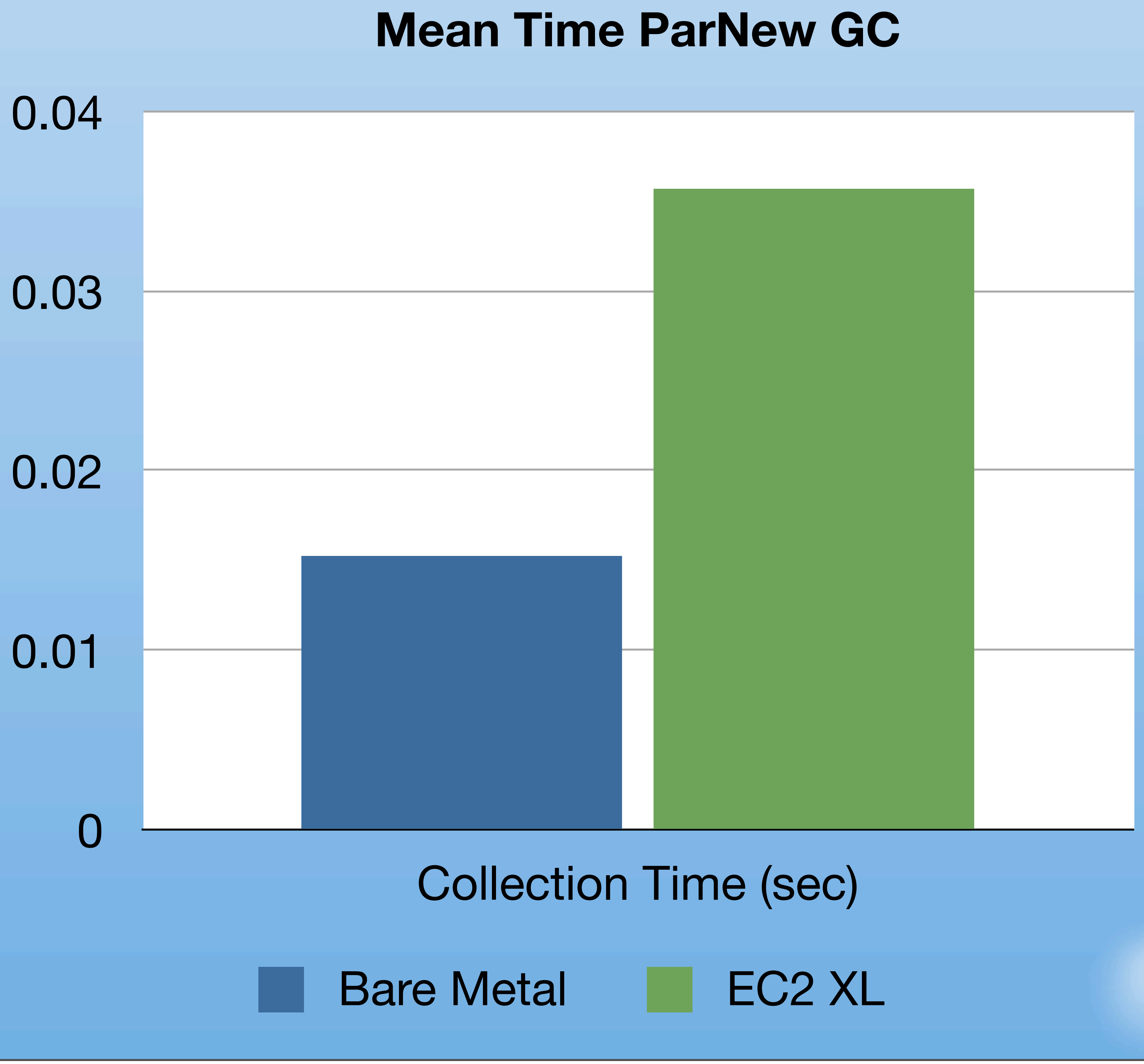
About EC2...

When you care about throughput, the virtualization tax is high



About EC2...

When you care about throughput, the virtualization tax is high



How we do at UA

- Originally our codebase was mostly one giant monolithic application, over time several databases
- Difficult to scale, technically and operationally
- Wanted to break off large pieces of functionality into coarse grained services encapsulating their capability and function
- Most message exchange was done using beanstalkd after migrating off RabbitMQ
- Fundamentally, our business is message passing we need to do that efficiently

Choosing A Framework

Choosing A Framework

- All frameworks are a form of concession

Choosing A Framework

- All frameworks are a form of concession
- Nobody would use Spring if people called it “Concessions to the horrors of EJB”

Choosing A Framework

- All frameworks are a form of concession
- Nobody would use Spring if people called it “Concessions to the horrors of EJB”
- Understand concessions when choosing, look for:

Choosing A Framework

- All frameworks are a form of concession
- Nobody would use Spring if people called it “Concessions to the horrors of EJB”
- Understand concessions when choosing, look for:
  - Configuration options - how do I configure Nagle behavior? Socket buffer sizes?

Choosing A Framework

- All frameworks are a form of concession
- Nobody would use Spring if people called it “Concessions to the horrors of EJB”
- Understand concessions when choosing, look for:
    - Configuration options - how do I configure Nagle behavior? Socket buffer sizes?
    - Metrics - what does the framework tell me about its internals?

Choosing A Framework

- All frameworks are a form of concession
- Nobody would use Spring if people called it “Concessions to the horrors of EJB”
- Understand concessions when choosing, look for:
    - Configuration options - how do I configure Nagle behavior? Socket buffer sizes?
    - Metrics - what does the framework tell me about its internals?
    - Intelligent logging - next level down from metrics

Choosing A Framework

- All frameworks are a form of concession
- Nobody would use Spring if people called it “Concessions to the horrors of EJB”
- Understand concessions when choosing, look for:
  - Configuration options - how do I configure Nagle behavior? Socket buffer sizes?
  - Metrics - what does the framework tell me about its internals?
  - Intelligent logging - next level down from metrics
  - How does the framework play with peers?

Frameworks - DO IT LIVE!

Frameworks - DO IT LIVE!

- Our requirements:

Frameworks - DO IT LIVE!

- Our requirements:
  - Capable of > 100K requests per second in aggregate across multiple threads

Frameworks - DO IT LIVE!

- Our requirements:
  - Capable of > 100K requests per second in aggregate across multiple threads
  - Simple protocol - easy to reason about, inspect

Frameworks - DO IT LIVE!

- Our requirements:
  - Capable of > 100K requests per second in aggregate across multiple threads
  - Simple protocol - easy to reason about, inspect
  - Efficient, extensible wire format - Google Protocol Buffers

Frameworks - DO IT LIVE!

- Our requirements:
  - Capable of > 100K requests per second in aggregate across multiple threads
  - Simple protocol - easy to reason about, inspect
  - Efficient, extensible wire format - Google Protocol Buffers
  - Compostable - easily create new services

Frameworks - DO IT LIVE!

- Our requirements:
  - Capable of > 100K requests per second in aggregate across multiple threads
  - Simple protocol - easy to reason about, inspect
  - Efficient, extensible wire format - Google Protocol Buffers
  - Compostable - easily create new services
  - Support both sync and async operations

Frameworks - DO IT LIVE!

- Our requirements:
  - Capable of > 100K requests per second in aggregate across multiple threads
  - Simple protocol - easy to reason about, inspect
  - Efficient, extensible wire format - Google Protocol Buffers
  - Compostable - easily create new services
  - Support both sync and async operations
  - Support for multiple languages (Python, Java, C++)

Frameworks - DO IT LIVE!

- Our requirements:
  - Capable of > 100K requests per second in aggregate across multiple threads
  - Simple protocol - easy to reason about, inspect
  - Efficient, extensible wire format - Google Protocol Buffers
  - Compostable - easily create new services
  - Support both sync and async operations
  - Support for multiple languages (Python, Java, C++)
  - Simple configuration

Frameworks - DO IT LIVE!

Frameworks - DO IT LIVE!

- Requirements:

Frameworks - DO IT LIVE!

- Requirements:
  - Discovery mechanism for finding/discarding services

Frameworks - DO IT LIVE!

- Requirements:
  - Discovery mechanism for finding/discarding services
  - Application congestion control combined with clear responsibility contracts

Frameworks - DO IT LIVE!

- Requirements:
  - Discovery mechanism for finding/discarding services
  - Application congestion control combined with clear responsibility contracts
- Optional:

Frameworks - DO IT LIVE!

- Requirements:
  - Discovery mechanism for finding/discarding services
  - Application congestion control combined with clear responsibility contracts
- Optional:
  - Adaptive load balancing

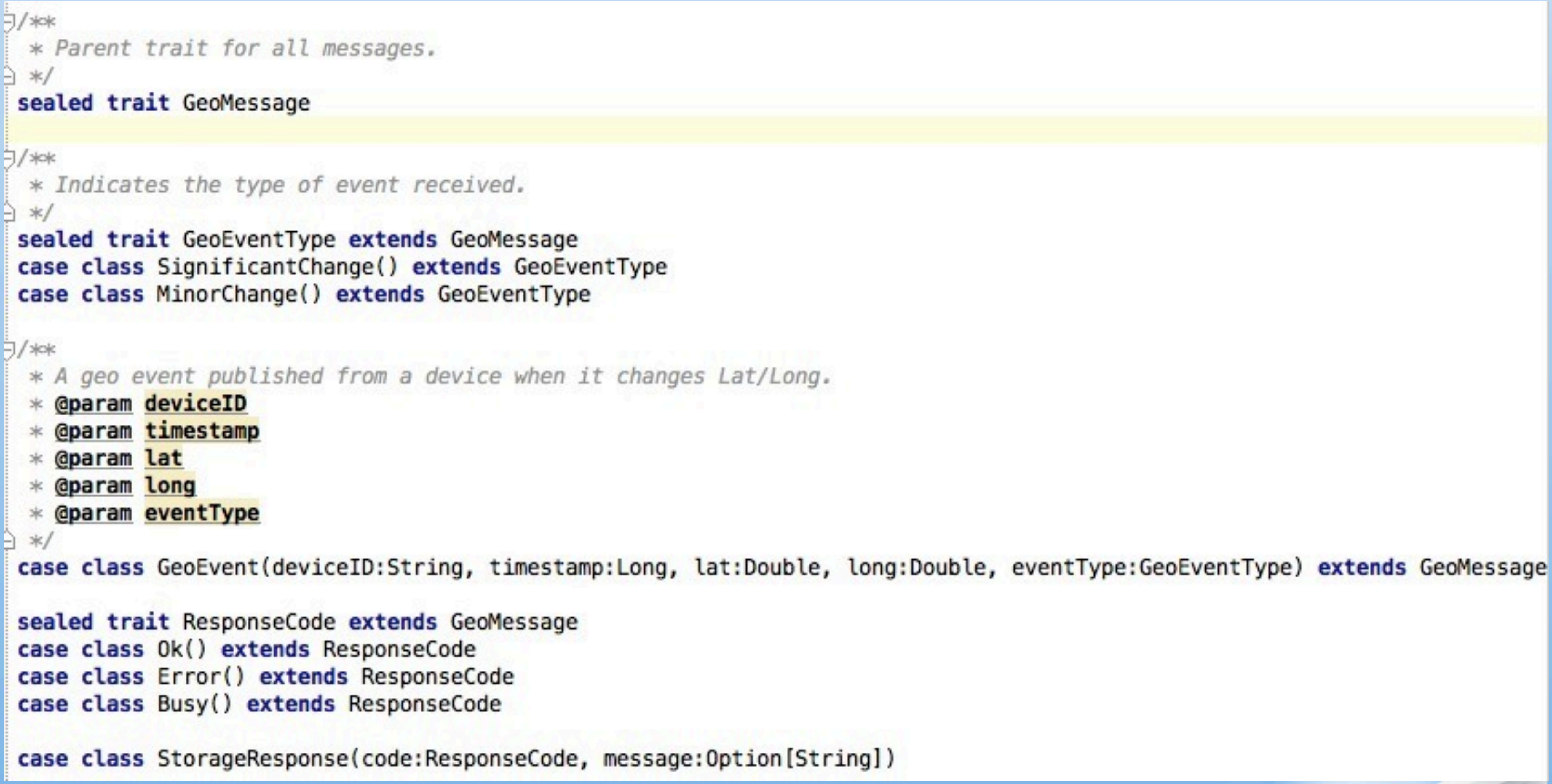
Frameworks - DO IT LIVE!

- Requirements:
  - Discovery mechanism for finding/discarding services
  - Application congestion control combined with clear responsibility contracts
- Optional:
  - Adaptive load balancing
  - Automated network partition recovery

Frameworks - Akka

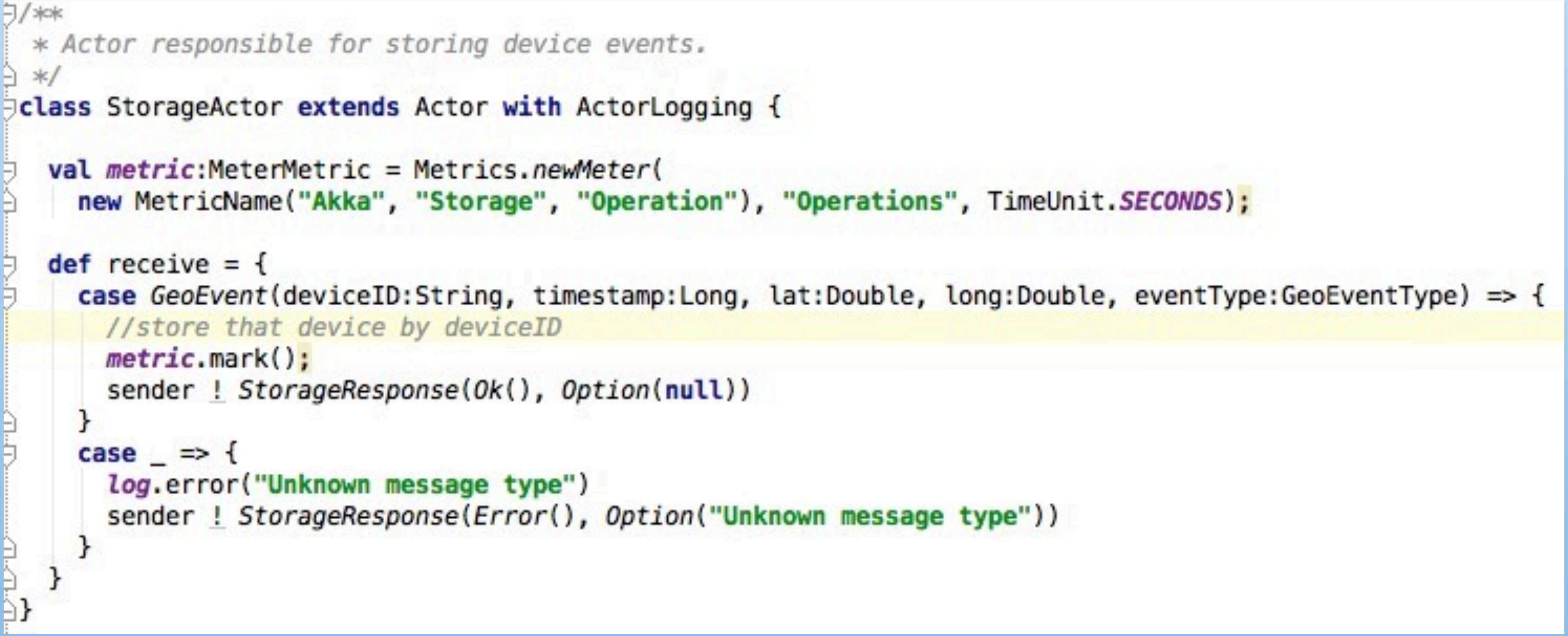
- Predominantly Scala platform for sending messages, distributed incarnation of the Actor pattern
- Message abstraction tolerates distribution well
- If you like OTP, you'll probably like Akka

Frameworks - Akka



```
/**
 * Parent trait for all messages.
 */
sealed trait GeoMessage

/**
 * Indicates the type of event received.
 */
sealed trait GeoEventType extends GeoMessage
case class SignificantChange() extends GeoEventType
case class MinorChange() extends GeoEventType

/**
 * A geo event published from a device when it changes Lat/Long.
 * @param deviceID
 * @param timestamp
 * @param lat
 * @param long
 * @param eventType
 */
case class GeoEvent(deviceID:String, timestamp:Long, lat:Double, long:Double, eventType:GeoEventType) extends GeoMessage

sealed trait ResponseCode extends GeoMessage
case class Ok() extends ResponseCode
case class Error() extends ResponseCode
case class Busy() extends ResponseCode

case class StorageResponse(code:ResponseCode, message:Option[String])
```

Frameworks - Akka



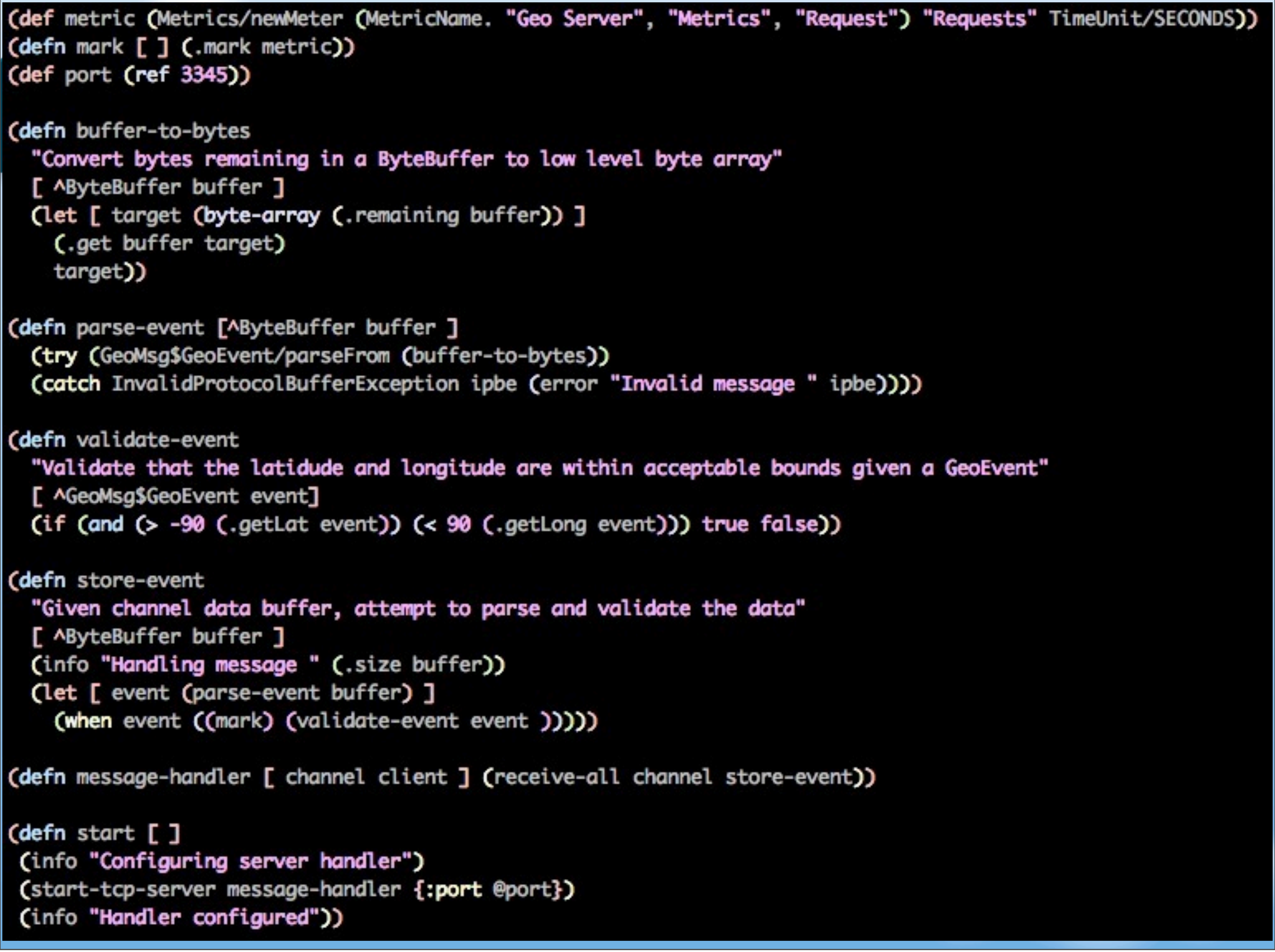
```scala
/**
 * Actor responsible for storing device events.
 */
class StorageActor extends Actor with ActorLogging {

  val metric:MeterMetric = Metrics.newMeter(
    new MetricName("Akka", "Storage", "Operation"), "Operations", TimeUnit.SECONDS);

  def receive = {
    case GeoEvent(deviceID:String, timestamp:Long, lat:Double, long:Double, eventType:GeoEventType) => {
      //store that device by deviceID
      metric.mark();
      sender ! StorageResponse(Ok(), Option(null))
    }
    case _ => {
      log.error("Unknown message type")
      sender ! StorageResponse(Error(), Option("Unknown message type"))
    }
  }
}
```

Frameworks - Akka

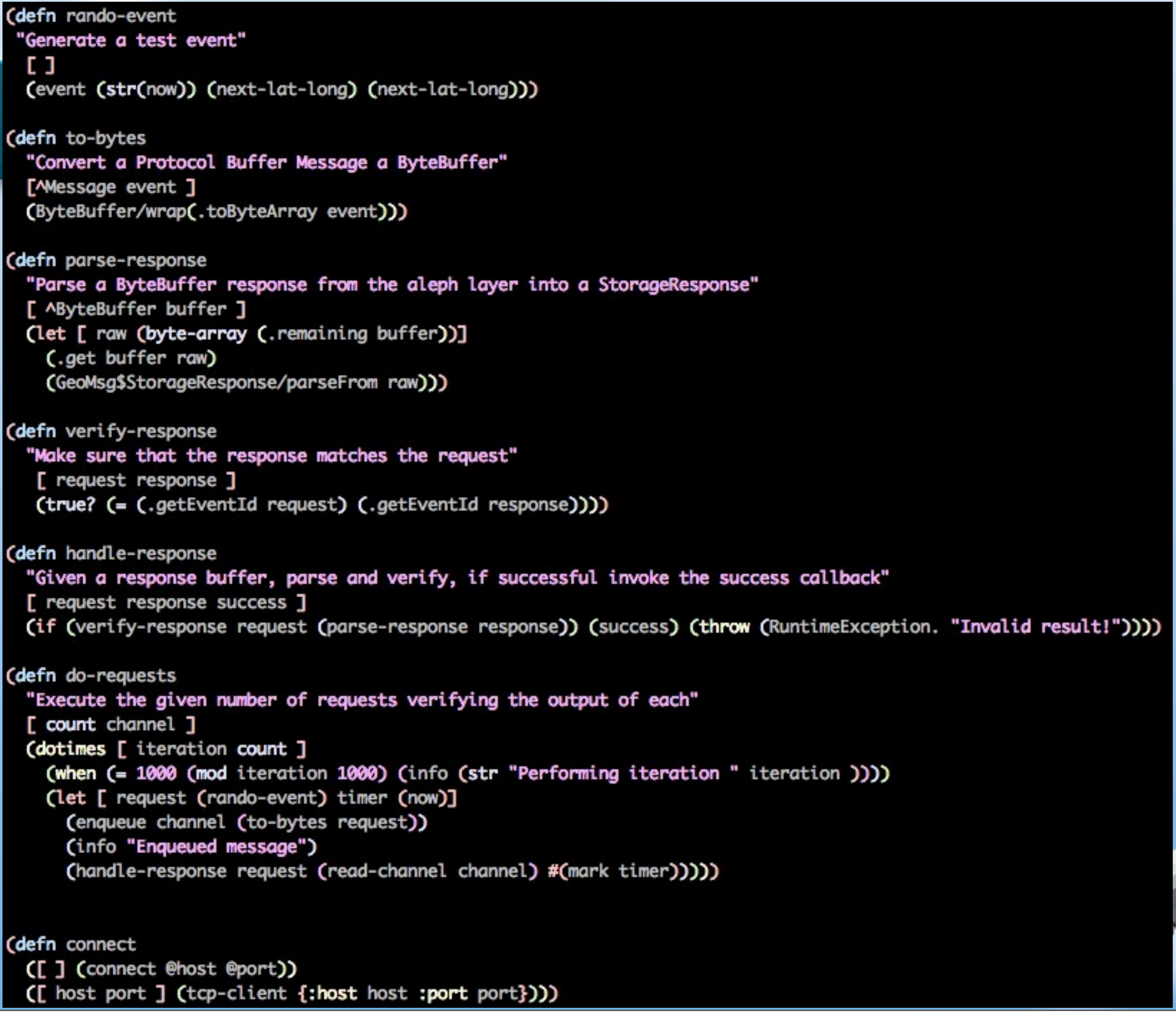
- Cons:
  - We don’t like reading other people’s Scala
  - Some pretty strong assertions in the docs that aren’t substantiated
  - Bulky wire protocol, especially for primitives
  - Configuration felt complicated
  - Sheer surface area of the framework is daunting
  - Unclear integration story with Python
  - Don’t want Dynamo for simple RPC

Frameworks - Aleph

- Clojure framework based on Netty, Lamina
- Conceptually funs are applied to channels to move around messages
- Channels are refs that you realize when you want data
- Operations with channels very easy
- Concise format for standing up clients and services using text protocols



```
(def metric (Metrics/newMeter (MetricName. "Geo Server", "Metrics", "Request") "Requests" TimeUnit/SECONDS))
(defn mark [ ] (.mark metric))
(def port (ref 3345))

(defn buffer-to-bytes
  "Convert bytes remaining in a ByteBuffer to low level byte array"
  [ ^ByteBuffer buffer ]
  (let [ target (byte-array (.remaining buffer)) ]
    (.get buffer target)
    target))

(defn parse-event [^ByteBuffer buffer ]
  (try (GeoMsg$GeoEvent/parseFrom (buffer-to-bytes))
  (catch InvalidProtocolBufferException ipbe (error "Invalid message " ipbe))))

(defn validate-event
  "Validate that the latidude and longitude are within acceptable bounds given a GeoEvent"
  [ ^GeoMsg$GeoEvent event]
  (if (and (> -90 (.getLat event)) (< 90 (.getLong event))) true false))

(defn store-event
  "Given channel data buffer, attempt to parse and validate the data"
  [ ^ByteBuffer buffer ]
  (info "Handling message " (.size buffer))
  (let [ event (parse-event buffer) ]
    (when event ((mark) (validate-event event )))))

(defn message-handler [ channel client ] (receive-all channel store-event))

(defn start [ ]
 (info "Configuring server handler")
 (start-tcp-server message-handler {:port @port})
 (info "Handler configured"))
```



```clojure
(defn rando-event
 "Generate a test event"
  [ ]
  (event (str(now)) (next-lat-long) (next-lat-long)))

(defn to-bytes
  "Convert a Protocol Buffer Message a ByteBuffer"
  [^Message event ]
  (ByteBuffer/wrap(.toByteArray event)))

(defn parse-response
  "Parse a ByteBuffer response from the aleph layer into a StorageResponse"
  [ ^ByteBuffer buffer ]
  (let [ raw (byte-array (.remaining buffer))]
    (.get buffer raw)
    (GeoMsg$StorageResponse/parseFrom raw)))

(defn verify-response
  "Make sure that the response matches the request"
   [ request response ]
   (true? (= (.getEventId request) (.getEventId response))))

(defn handle-response
  "Given a response buffer, parse and verify, if successful invoke the success callback"
  [ request response success ]
  (if (verify-response request (parse-response response)) (success) (throw (RuntimeException. "Invalid result!"))))

(defn do-requests
  "Execute the given number of requests verifying the output of each"
  [ count channel ]
  (dotimes [ iteration count ]
    (when (= 1000 (mod iteration 1000)) (info (str "Performing iteration " iteration )))
    (let [ request (rando-event) timer (now)]
      (enqueue channel (to-bytes request))
      (info "Enqueued message")
      (handle-response request (read-channel channel) #(mark timer)))))


(defn connect
  ([ ] (connect @host @port))
  ([ host port ] (tcp-client {:host host :port port})))
```

Frameworks - Aleph

- Cons:
  - Very high level abstraction, knobs are buried if they exist
  - Channel concept leaky for large messages, unclear how to stream
  - Documentation, tests

Frameworks - Netty

- **The** preeminent framework for doing Async Network I/O on the JVM
- Netty Channels backed by pipelines on top of lower level NIO Channels
- Pros:
  - Abstraction doesn’t hide the important pieces
  - The only sane way to do SSL with Async I/O on the JVM
  - Protocols well abstracted into pipeline steps
  - Clean callback model for events of interest but optional in simple cases - no death by callback

Frameworks - Netty

- Cons:
  - Easy to make too many copies of the data
  - Some old school bootstrap idioms
  - Writes can occasionally be reordered
  - Failure conditions can be numerous, difficult to reason about
  - Simple things can feel difficult - UDP, simple request/reply
  - Sync timeout implementation heavy-handed

Frameworks - DO IT LIVE!

Frameworks - DO IT LIVE!

- Considered but passed:

Frameworks - DO IT LIVE!

- Considered but passed:
  - PB-RPC Implementations

Frameworks - DO IT LIVE!

- Considered but passed:
  - PB-RPC Implementations
  - Thrift

Frameworks - DO IT LIVE!

- Considered but passed:
  - PB-RPC Implementations
  - Thrift
  - Twitter’s Finagle

Frameworks - DO IT LIVE!

- Considered but passed:
  - PB-RPC Implementations
  - Thrift
  - Twitter’s Finagle
  - Akka

Frameworks - DO IT LIVE!

- Considered but passed:
  - PB-RPC Implementations
  - Thrift
  - Twitter’s Finagle
  - Akka
  - ØMQ

Frameworks - DO IT LIVE!

- Considered but passed:
  - PB-RPC Implementations
  - Thrift
  - Twitter's Finagle
  - Akka
  - ØMQ
  - HTTP + JSON

Frameworks - DO IT LIVE!

- Considered but passed:
  - PB-RPC Implementations
  - Thrift
  - Twitter’s Finagle
  - Akka
  - ØMQ
  - HTTP + JSON
  - ZeroC Ice

Frameworks - DO IT LIVE!

Frameworks - DO IT LIVE!

- Ultimately implemented our own using combination of Netty and Google Protocol Buffers called Reactor

Frameworks - DO IT LIVE!

- Ultimately implemented our own using combination of Netty and Google Protocol Buffers called Reactor
- Discovery (optional) using a defined tree of versioned services in ZooKeeper

Frameworks - DO IT LIVE!

- Ultimately implemented our own using combination of Netty and Google Protocol Buffers called Reactor
- Discovery (optional) using a defined tree of versioned services in ZooKeeper
- Service instances periodically publish load factor to ZooKeeper for clients to inform routing decisions

Frameworks - DO IT LIVE!

- Ultimately implemented our own using combination of Netty and Google Protocol Buffers called Reactor
- Discovery (optional) using a defined tree of versioned services in ZooKeeper
- Service instances periodically publish load factor to ZooKeeper for clients to inform routing decisions
- Rich metrics using Yammer Metrics

Frameworks - DO IT LIVE!

- Ultimately implemented our own using combination of Netty and Google Protocol Buffers called Reactor
- Discovery (optional) using a defined tree of versioned services in ZooKeeper
- Service instances periodically publish load factor to ZooKeeper for clients to inform routing decisions
- Rich metrics using Yammer Metrics
- Core service traits are part of the framework

Frameworks - DO IT LIVE!

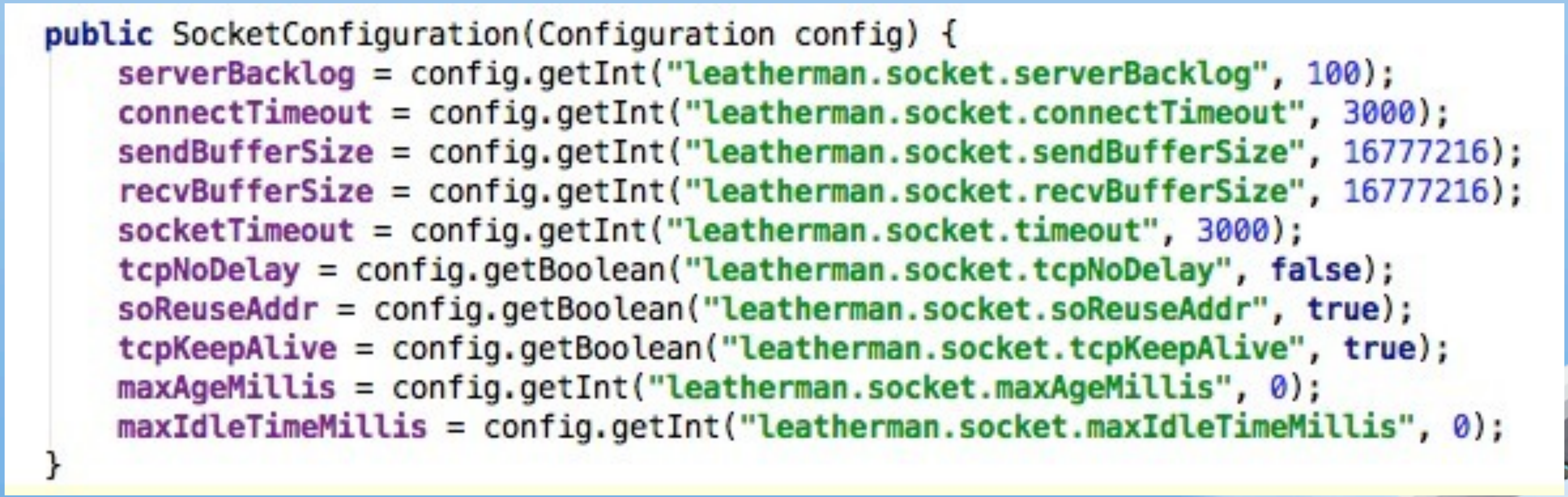
- Ultimately implemented our own using combination of Netty and Google Protocol Buffers called Reactor
- Discovery (optional) using a defined tree of versioned services in ZooKeeper
- Service instances periodically publish load factor to ZooKeeper for clients to inform routing decisions
- Rich metrics using Yammer Metrics
- Core service traits are part of the framework
- Service instances quiesce gracefully

Frameworks - DO IT LIVE!

- Ultimately implemented our own using combination of Netty and Google Protocol Buffers called Reactor
- Discovery (optional) using a defined tree of versioned services in ZooKeeper
- Service instances periodically publish load factor to ZooKeeper for clients to inform routing decisions
- Rich metrics using Yammer Metrics
- Core service traits are part of the framework
- Service instances quiesce gracefully
- Netty made UDP, Sync, Async. easy

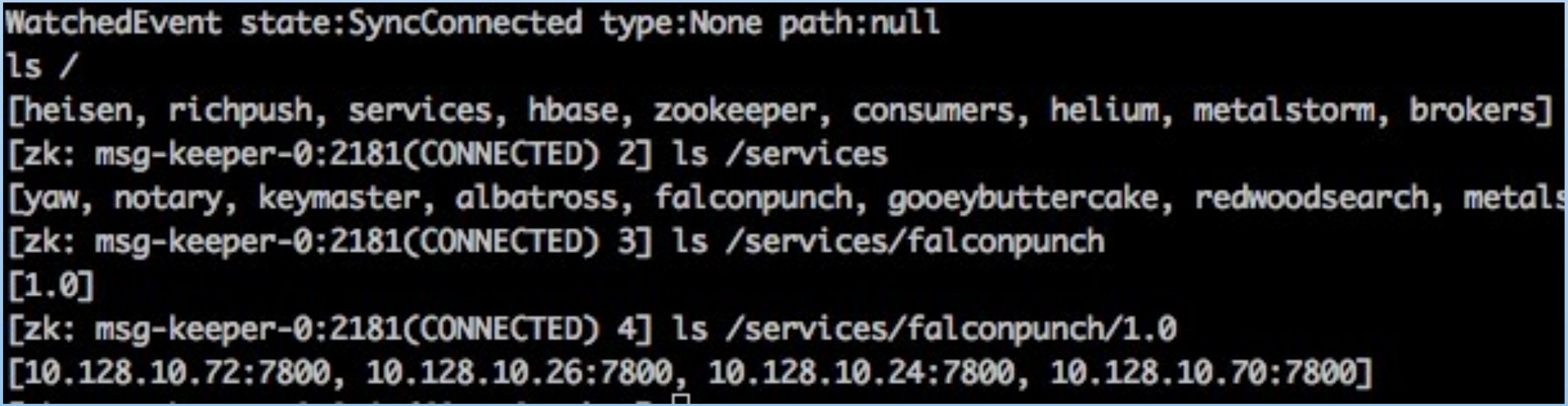
Frameworks - DO IT LIVE!

- All operations are Callables, services define a mapping b/t a request type and a Callable
- Client API always returns a Future, sometimes it’s already materialized
- Precise tuning from config files



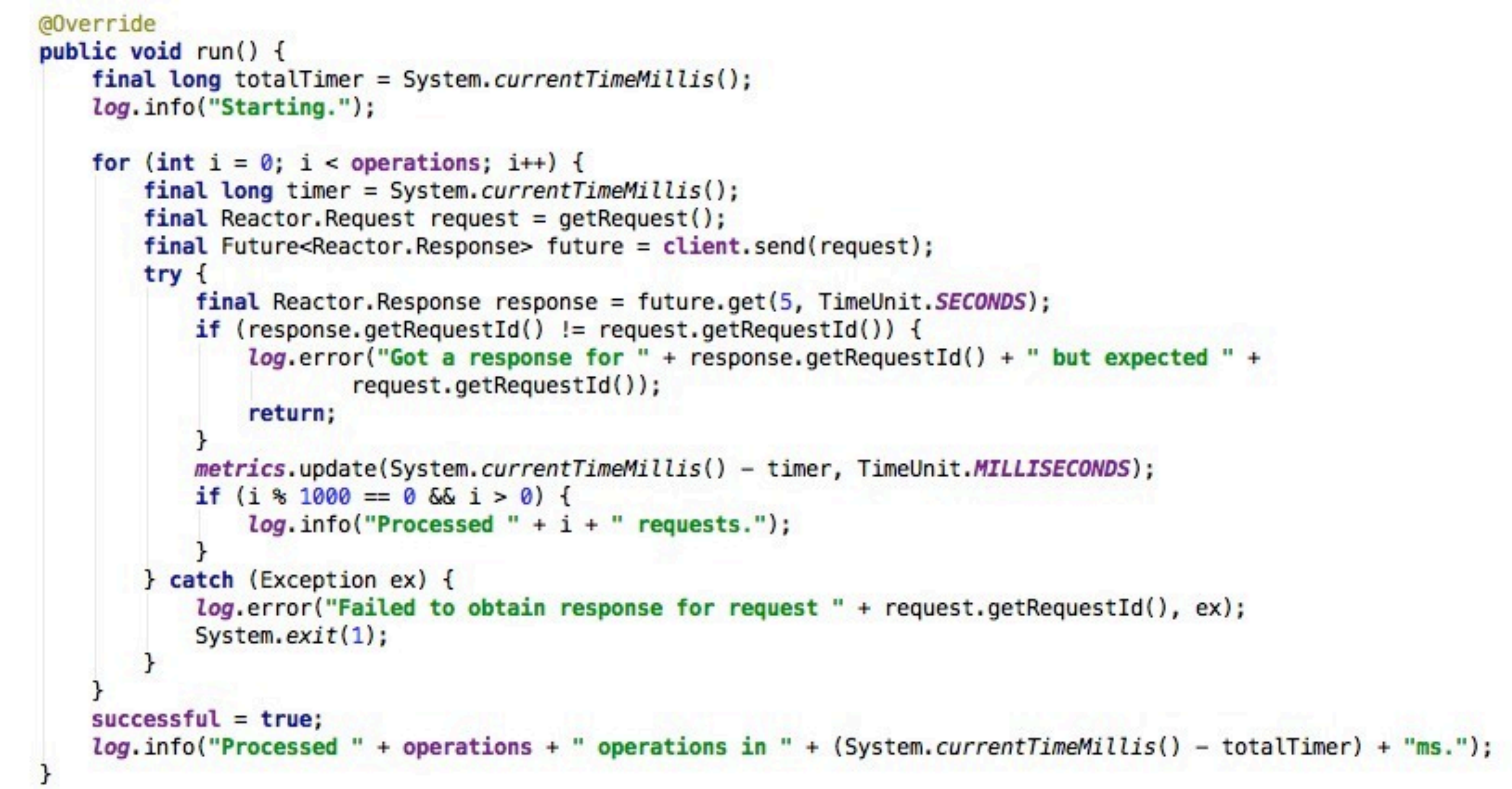
```
public SocketConfiguration(Configuration config) {
    serverBacklog = config.getInt("leatherman.socket.serverBacklog", 100);
    connectTimeout = config.getInt("leatherman.socket.connectTimeout", 3000);
    sendBufferSize = config.getInt("leatherman.socket.sendBufferSize", 16777216);
    recvBufferSize = config.getInt("leatherman.socket.recvBufferSize", 16777216);
    socketTimeout = config.getInt("leatherman.socket.timeout", 3000);
    tcpNoDelay = config.getBoolean("leatherman.socket.tcpNoDelay", false);
    soReuseAddr = config.getBoolean("leatherman.socket.soReuseAddr", true);
    tcpKeepAlive = config.getBoolean("leatherman.socket.tcpKeepAlive", true);
    maxAgeMillis = config.getInt("leatherman.socket.maxAgeMillis", 0);
    maxIdleTimeMillis = config.getInt("leatherman.socket.maxIdleTimeMillis", 0);
}
```

What We Learned - In General



```
WatchedEvent state:SyncConnected type:None path:null
ls /
[heisen, richpush, services, hbase, zookeeper, consumers, helium, metalstorm, brokers]
[zk: msg-keeper-0:2181(CONNECTED) 2] ls /services
[yaw, notary, keymaster, albatross, falconpunch, gooeybuttercake, redwoodsearch, metals
[zk: msg-keeper-0:2181(CONNECTED) 3] ls /services/falconpunch
[1.0]
[zk: msg-keeper-0:2181(CONNECTED) 4] ls /services/falconpunch/1.0
[10.128.10.72:7800, 10.128.10.26:7800, 10.128.10.24:7800, 10.128.10.70:7800]
```

Frameworks - DO IT LIVE!



```java
@Override
public void run() {
    final long totalTimer = System.currentTimeMillis();
    log.info("Starting.");

    for (int i = 0; i < operations; i++) {
        final long timer = System.currentTimeMillis();
        final Reactor.Request request = getRequest();
        final Future<Reactor.Response> future = client.send(request);
        try {
            final Reactor.Response response = future.get(5, TimeUnit.SECONDS);
            if (response.getRequestId() != request.getRequestId()) {
                log.error("Got a response for " + response.getRequestId() + " but expected " +
                        request.getRequestId());
                return;
            }
            metrics.update(System.currentTimeMillis() - timer, TimeUnit.MILLISECONDS);
            if (i % 1000 == 0 && i > 0) {
                log.info("Processed " + i + " requests.");
            }
        } catch (Exception ex) {
            log.error("Failed to obtain response for request " + request.getRequestId(), ex);
            System.exit(1);
        }
    }
    successful = true;
    log.info("Processed " + operations + " operations in " + (System.currentTimeMillis() - totalTimer) + "ms.");
}
```

What We Learned - In General

What We Learned - In General

- Straight through RPC was fairly easy, edge cases were hard

What We Learned - In General

- Straight through RPC was fairly easy, edge cases were hard
- ZooKeeper is brutal to program with, recover from errors

What We Learned - In General

- Straight through RPC was fairly easy, edge cases were hard
- ZooKeeper is brutal to program with, recover from errors
- Discovery is also difficult - clients need to defend themselves, consider partitions

What We Learned - In General

- Straight through RPC was fairly easy, edge cases were hard
- ZooKeeper is brutal to program with, recover from errors
- Discovery is also difficult - clients need to defend themselves, consider partitions
- RPC is great for latency, but upstream pushback is important

What We Learned - In General

- Straight through RPC was fairly easy, edge cases were hard
- ZooKeeper is brutal to program with, recover from errors
- Discovery is also difficult - clients need to defend themselves, consider partitions
- RPC is great for latency, but upstream pushback is important
- Save RPC for latency sensitive operations - use Kafka

What We Learned - In General

- Straight through RPC was fairly easy, edge cases were hard
- ZooKeeper is brutal to program with, recover from errors
- Discovery is also difficult - clients need to defend themselves, consider partitions
- RPC is great for latency, but upstream pushback is important
- Save RPC for latency sensitive operations - use Kafka
- RPC less than ideal for fan-out

What We Learned - In General

- Straight through RPC was fairly easy, edge cases were hard
- ZooKeeper is brutal to program with, recover from errors
- Discovery is also difficult - clients need to defend themselves, consider partitions
- RPC is great for latency, but upstream pushback is important
- Save RPC for latency sensitive operations - use Kafka
- RPC less than ideal for fan-out
- PBs make future replay trivial

What We Learned - TCP

What We Learned - TCP

What We Learned - TCP

- RTO (retransmission timeout) and Karn and Jacobson’s Algorithms

What We Learned - TCP

- RTO (retransmission timeout) and Karn and Jacobson’s Algorithms
  - Linux defaults to 15 retry attempts, 3 seconds between

What We Learned - TCP

- RTO (retransmission timeout) and Karn and Jacobson's Algorithms
  - Linux defaults to 15 retry attempts, 3 seconds between
  - With no ACKs, congestion control kicks in and widens that 3 second window exponentially, thinking its congested

What We Learned - TCP

- RTO (retransmission timeout) and Karn and Jacobson's Algorithms
  - Linux defaults to 15 retry attempts, 3 seconds between
  - With no ACKs, congestion control kicks in and widens that 3 second window exponentially, thinking its congested
  - Connection timeout can take up to 30 minutes

What We Learned - TCP

- RTO (retransmission timeout) and Karn and Jacobson’s Algorithms
  - Linux defaults to 15 retry attempts, 3 seconds between
  - With no ACKs, congestion control kicks in and widens that 3 second window exponentially, thinking its congested
  - Connection timeout can take up to 30 minutes
  - Devices, Carriers and EC2 at scale eat FIN/RST

What We Learned - TCP

- RTO (retransmission timeout) and Karn and Jacobson’s Algorithms
  - Linux defaults to 15 retry attempts, 3 seconds between
  - With no ACKs, congestion control kicks in and widens that 3 second window exponentially, thinking its congested
  - Connection timeout can take up to 30 minutes
  - Devices, Carriers and EC2 at scale eat FIN/RST
  - Our systems think a device is still online at the time of a push

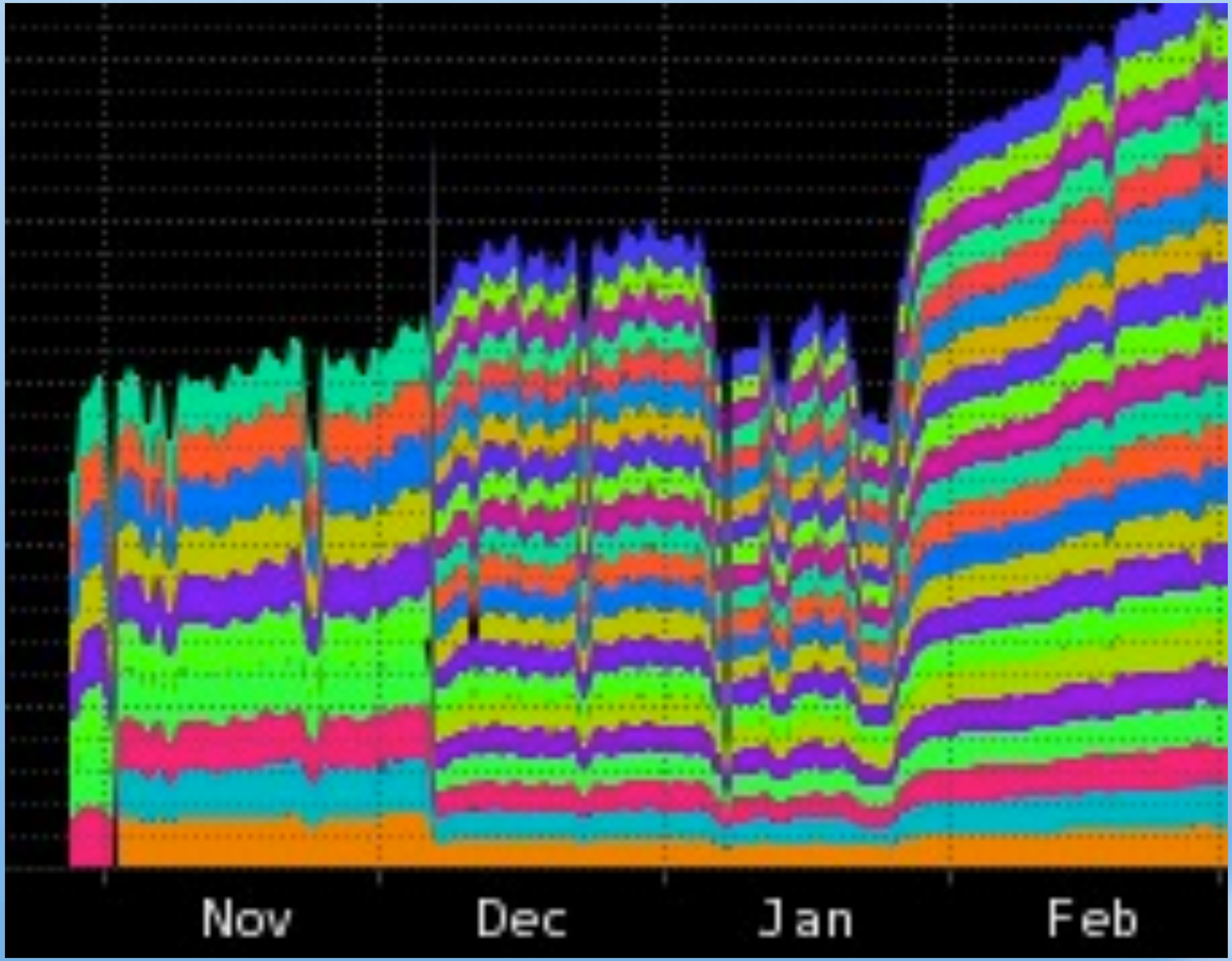
What We Learned - TCP

What We Learned - TCP

- After changing the RTO

What We Learned - TCP

- After changing the RTO



What We Learned - TCP

What We Learned - TCP

What We Learned - TCP

- Efficiency means understanding your traffic

What We Learned - TCP

- Efficiency means understanding your traffic
- Size send/recv buffers appropriately (defaults way too low for edge tier services)

What We Learned - TCP

- Efficiency means understanding your traffic
- Size send/recv buffers appropriately (defaults way too low for edge tier services)
- Nagle! Non-duplex protocols can benefit significantly

What We Learned - TCP

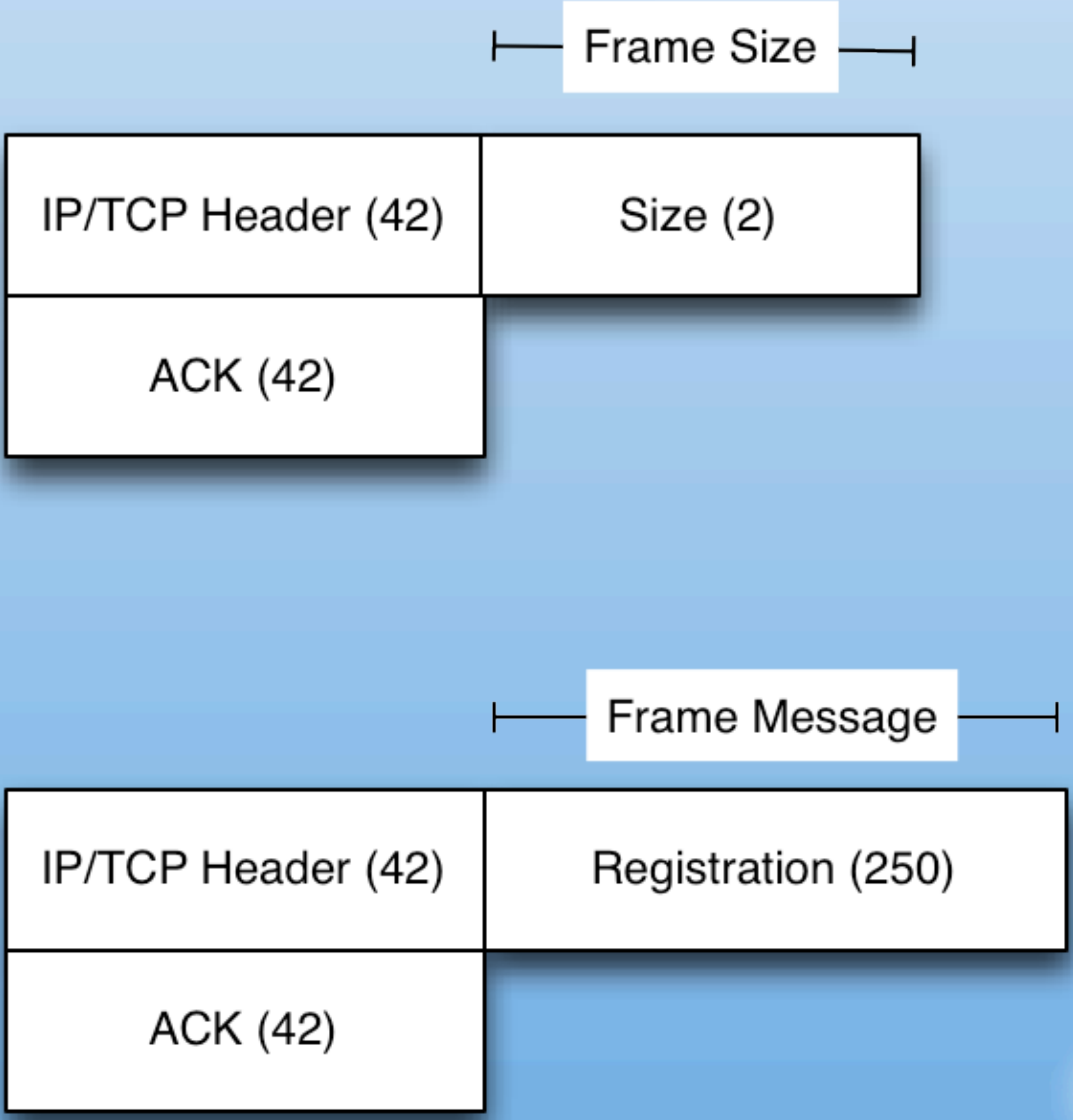
- Efficiency means understanding your traffic
- Size send/recv buffers appropriately (defaults way too low for edge tier services)
- Nagle! Non-duplex protocols can benefit significantly
- Example: 19K message deliveries per second vs. 2K

What We Learned - TCP

- Efficiency means understanding your traffic
- Size send/recv buffers appropriately (defaults way too low for edge tier services)
- Nagle! Non-duplex protocols can benefit significantly
- Example: 19K message deliveries per second vs. 2K
- Example: our protocol has a size frame, w/o Nagle that went in its own packet

What We Learned - TCP

What We Learned - TCP



Frame Size

IP/TCP Header (42)

Size (2)

ACK (42)

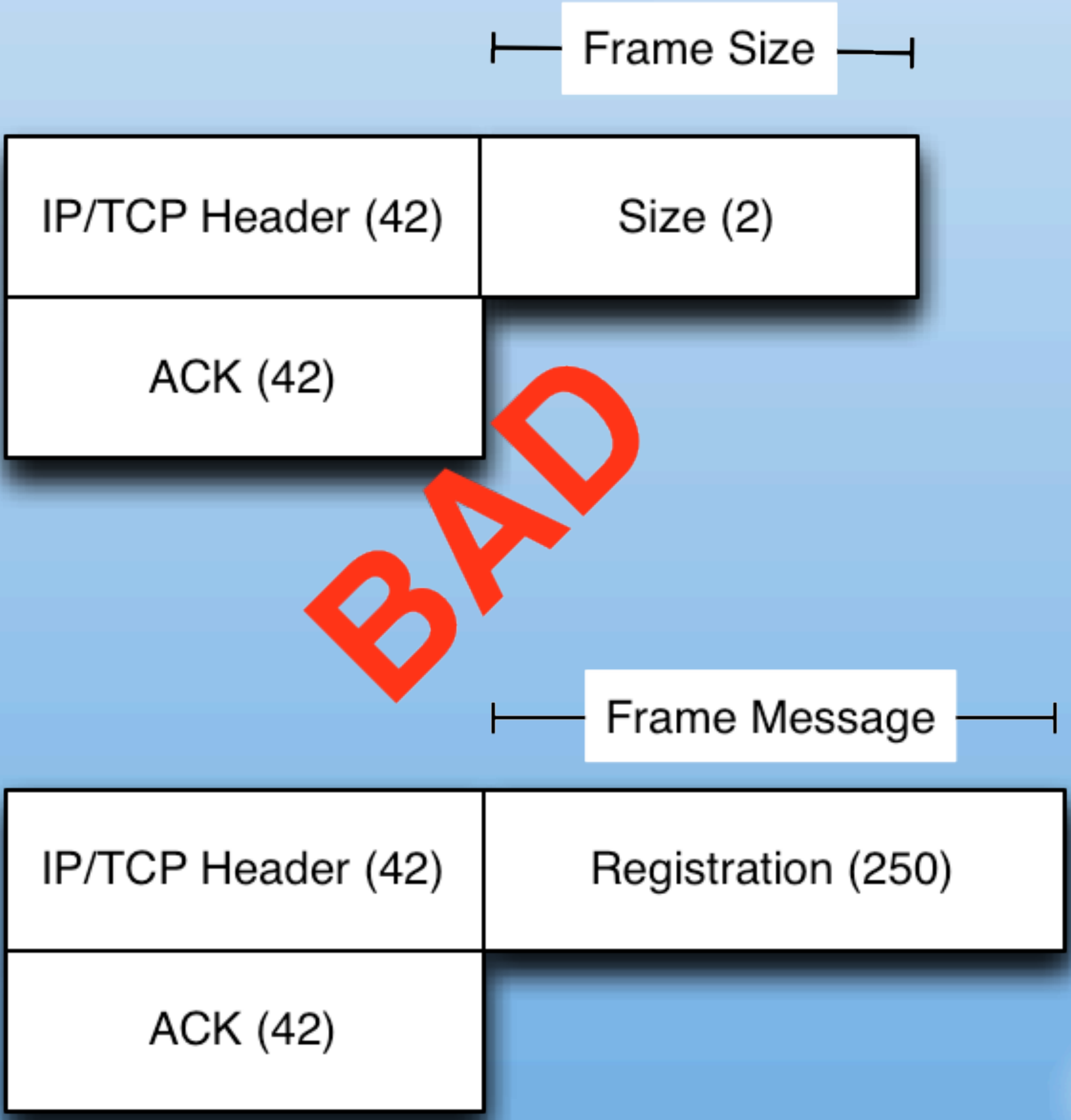
Frame Message

IP/TCP Header (42)

Registration (250)

ACK (42)

What We Learned - TCP



Frame Size

IP/TCP Header (42)

Size (2)

ACK (42)

BAD

Frame Message

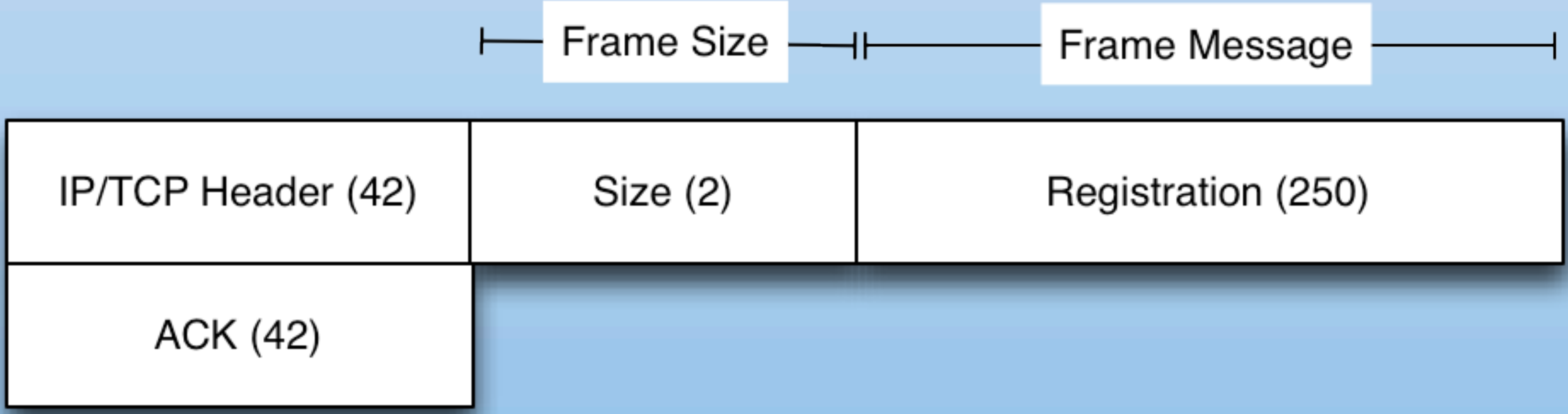
IP/TCP Header (42)

Registration (250)

ACK (42)

What We Learned - TCP

What We Learned - TCP



Frame Size

Frame Message

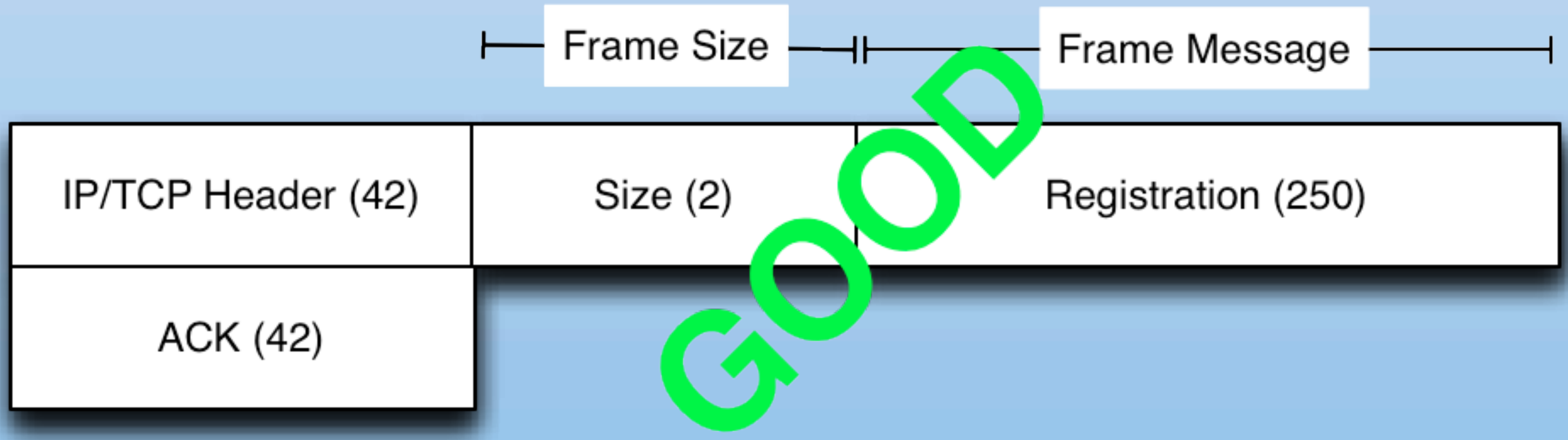
IP/TCP Header (42)

Size (2)

Registration (250)

ACK (42)

What We Learned - TCP



Frame Size

Frame Message

IP/TCP Header (42)

Size (2)

Registration (250)

ACK (42)

Saves 84 bytes, 1 round trip

What We Learned - TCP

What We Learned - TCP

What We Learned - TCP

- Don’t Nagle!

What We Learned - TCP

- Don’t Nagle!
  - Again, understand what your traffic is doing

What We Learned - TCP

- Don’t Nagle!
    - Again, understand what your traffic is doing
    - Buffer and make one syscall instead of multiple

What We Learned - TCP

- Don’t Nagle!
  - Again, understand what your traffic is doing
  - Buffer and make one syscall instead of multiple
  - High-throughput RPC mechanisms disable it explicitly

What We Learned - TCP

- Don’t Nagle!
  - Again, understand what your traffic is doing
  - Buffer and make one syscall instead of multiple
  - High-throughput RPC mechanisms disable it explicitly
  - Better mechanisms not accessible to JVM directly

What We Learned - TCP

- Don’t Nagle!
  - Again, understand what your traffic is doing
  - Buffer and make one syscall instead of multiple
  - High-throughput RPC mechanisms disable it explicitly
  - Better mechanisms not accessible to JVM directly
  - See also:

What We Learned - TCP

- Don’t Nagle!
  - Again, understand what your traffic is doing
  - Buffer and make one syscall instead of multiple
  - High-throughput RPC mechanisms disable it explicitly
  - Better mechanisms not accessible to JVM directly
  - See also:
    - http://www.evanjones.ca/software/java-bytebuffers.html

What We Learned - TCP

- Don’t Nagle!
  - Again, understand what your traffic is doing
  - Buffer and make one syscall instead of multiple
  - High-throughput RPC mechanisms disable it explicitly
  - Better mechanisms not accessible to JVM directly
  - See also:
    - http://www.evanjones.ca/software/java-bytebuffers.html
    - http://blog.boundary.com/2012/05/02/know-a-delay-nagles-algorithm-and-you/

About UDP...

About UDP...

About UDP...

- Generally to be avoided

About UDP...

- Generally to be avoided
- Great for small unimportant data like memcache operations at extreme scale

About UDP...

- Generally to be avoided
- Great for small unimportant data like memcache operations at extreme scale
- Bad for RPC when you care about knowing if your request was handled

About UDP...

- Generally to be avoided
- Great for small unimportant data like memcache operations at extreme scale
- Bad for RPC when you care about knowing if your request was handled
- Conditions where you most want your data are also the most likely to cause your data to be dropped

About SSL/TLS

About SSL/TLS

- Understand the consequences - complex, slow and expensive, especially for internal services

About SSL/TLS

- Understand the consequences - complex, slow and expensive, especially for internal services
- ~6.5K and 4 hops to secure the channel

About SSL/TLS

- Understand the consequences - complex, slow and expensive, especially for internal services
- ~6.5K and 4 hops to secure the channel
- 40 bytes overhead per frame

About SSL/TLS

- Understand the consequences - complex, slow and expensive, especially for internal services
- ~6.5K and 4 hops to secure the channel
- 40 bytes overhead per frame
- 38.1MB overhead for every keep-alive sent to 1M devices

About SSL/TLS

- Understand the consequences - complex, slow and expensive, especially for internal services
- ~6.5K and 4 hops to secure the channel
- 40 bytes overhead per frame
- 38.1MB overhead for every keep-alive sent to 1M devices

TLS source: http://netsekure.org/2010/03/tls-overhead/

We Learned About Carriers

We Learned About Carriers

- Data plans are like gym memberships

We Learned About Carriers

- Data plans are like gym memberships
- Aggressively cull idle stream connections

We Learned About Carriers

- Data plans are like gym memberships
- Aggressively cull idle stream connections
- Don’t like TCP keepalives

We Learned About Carriers

- Data plans are like gym memberships
- Aggressively cull idle stream connections
- Don’t like TCP keepalives
- Don’t like UDP

We Learned About Carriers

- Data plans are like gym memberships
- Aggressively cull idle stream connections
- Don’t like TCP keepalives
- Don’t like UDP
- Like to batch, delay or just drop FIN/FIN ACK/RST

We Learned About Carriers

- Data plans are like gym memberships
- Aggressively cull idle stream connections
- Don’t like TCP keepalives
- Don’t like UDP
- Like to batch, delay or just drop FIN/FIN ACK/RST
- Move data through aggregators

About Devices...

About Devices...

- Small compute units that do exactly what you tell them to

About Devices...

- Small compute units that do exactly what you tell them to
- Like phone home when you push to them...

About Devices...

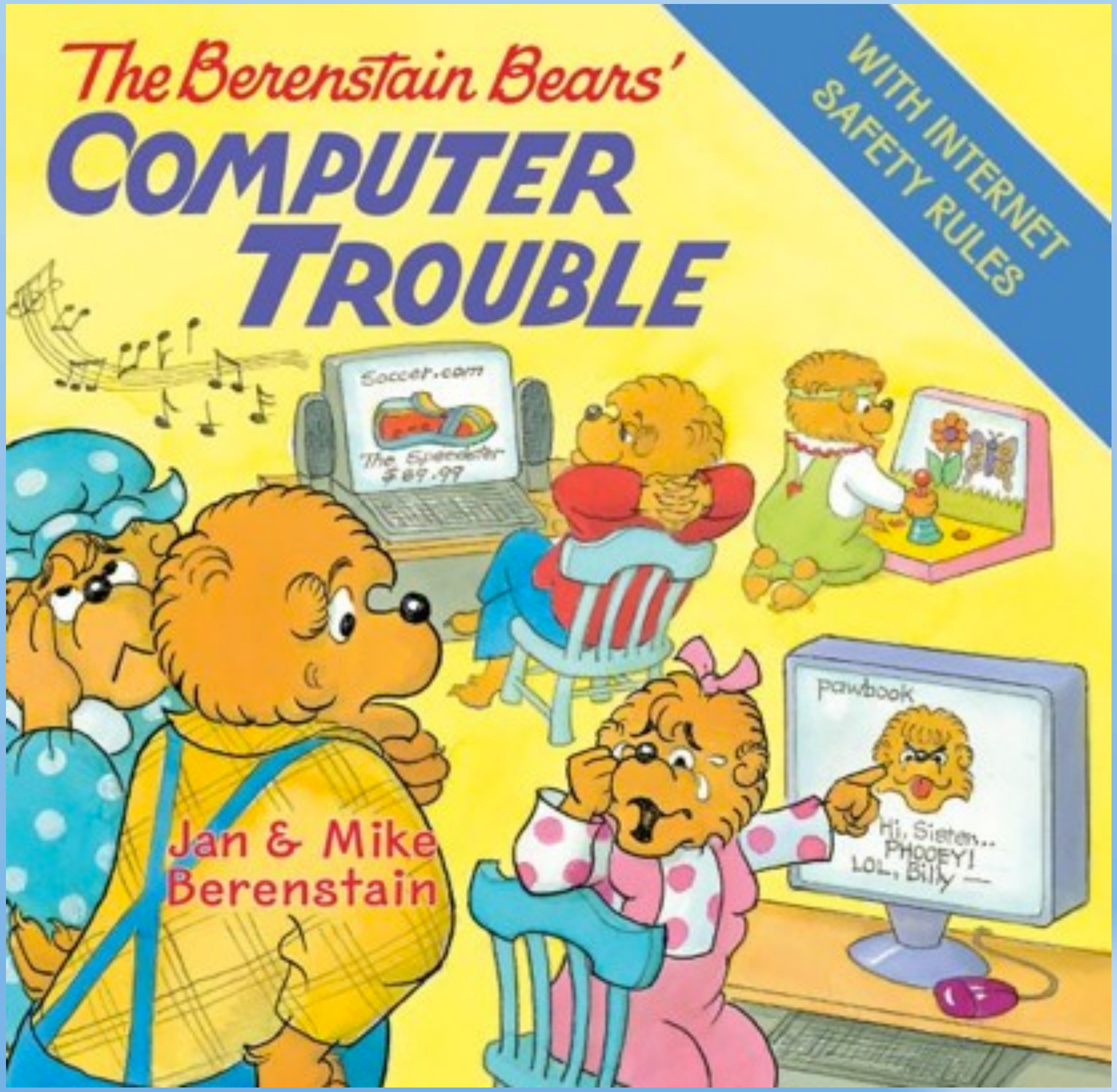
- Small compute units that do exactly what you tell them to
- Like phone home when you push to them...
- 10M at a time...

About Devices...

- Small compute units that do exactly what you tell them to
- Like phone home when you push to them...
- 10M at a time...
- Causing...

About Devices...

- Small compute units that do exactly what you tell them to
- Like phone home when you push to them...
- 10M at a time...
- Causing...



About Devices...

About Devices...

- Herds can happen for many of reasons:

About Devices...

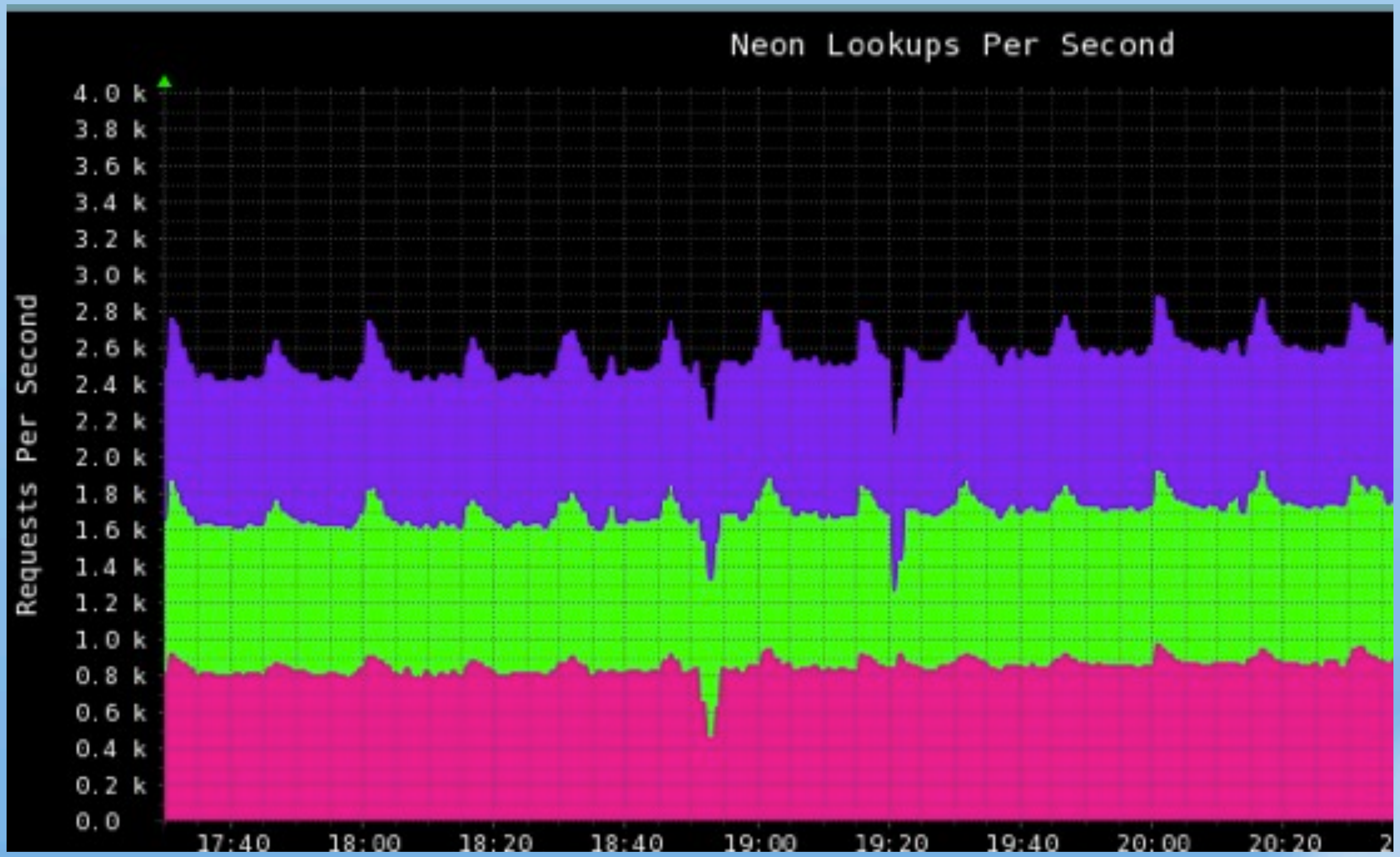
- Herds can happen for many of reasons:
  - Network events

About Devices...

- Herds can happen for many of reasons:
  - Network events
  - Android imprecise timer

About Devices...

- Herds can happen for many of reasons:
  - Network events
  - Android imprecise timer



About Devices...

About Devices...

- By virtue of being a mobile device, they move around a lot

About Devices...

- By virtue of being a mobile device, they move around a lot
- When they move, they often change IP addresses

About Devices...

- By virtue of being a mobile device, they move around a lot
- When they move, they often change IP addresses
  - New cell tower

About Devices...

- By virtue of being a mobile device, they move around a lot
- When they move, they often change IP addresses
    - New cell tower
    - Change connectivity - 4G -> 3G, 3G -> WiFi, etc.

About Devices...

- By virtue of being a mobile device, they move around a lot
- When they move, they often change IP addresses
  - New cell tower
  - Change connectivity - 4G -> 3G, 3G -> WiFi, etc.
- When they change IP addresses, they need to reconnect TCP sockets

About Devices...

- By virtue of being a mobile device, they move around a lot
- When they move, they often change IP addresses
  - New cell tower
  - Change connectivity - 4G -> 3G, 3G -> WiFi, etc.
- When they change IP addresses, they need to reconnect TCP sockets
- **Sometimes** they are kind enough to let us know

About Devices...

- By virtue of being a mobile device, they move around a lot
- When they move, they often change IP addresses
  - New cell tower
  - Change connectivity - 4G -> 3G, 3G -> WiFi, etc.
- When they change IP addresses, they need to reconnect TCP sockets
- **Sometimes** they are kind enough to let us know
- Those reconnections are expensive for us and the devices

We Learned About EC2

We Learned About EC2

- EC2 is a great jumping-off point

We Learned About EC2

- EC2 is a great jumping-off point
- Scaling vertically is very expensive

We Learned About EC2

- EC2 is a great jumping-off point
- Scaling vertically is very expensive
- Like Carriers, EC2 networking is fond of holding on to TCP teardown sequence packets

We Learned About EC2

- EC2 is a great jumping-off point
- Scaling vertically is very expensive
- Like Carriers, EC2 networking is fond of holding on to TCP teardown sequence packets
- vNICs obfuscate important data when you care about 1M connections

We Learned About EC2

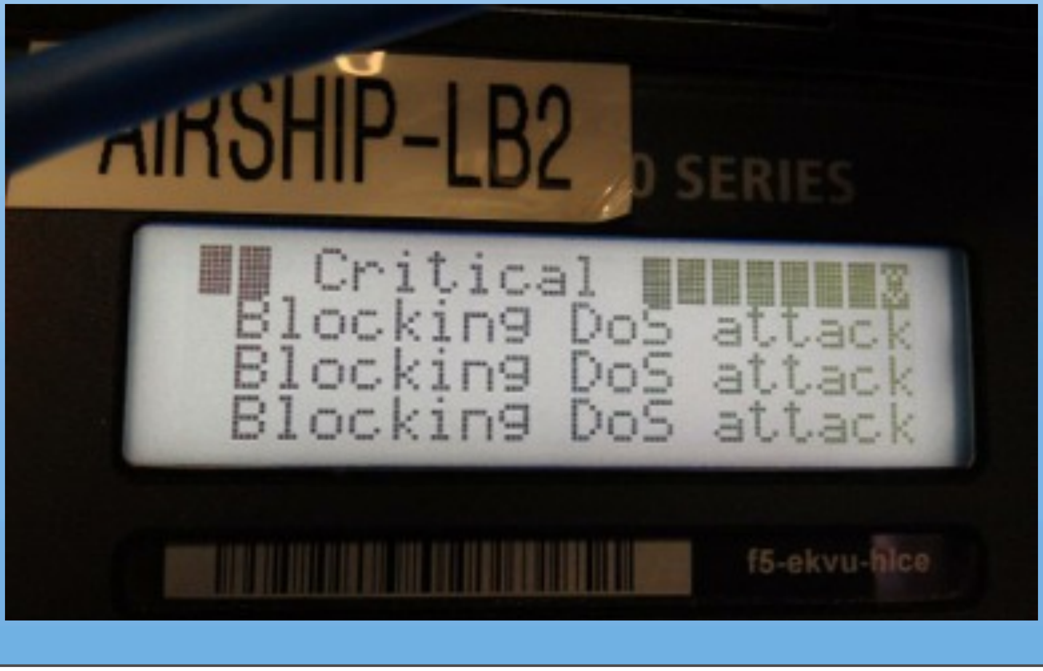
- EC2 is a great jumping-off point
- Scaling vertically is very expensive
- Like Carriers, EC2 networking is fond of holding on to TCP teardown sequence packets
- vNICs obfuscate important data when you care about 1M connections
- Great for surge capacity

We Learned About EC2

- EC2 is a great jumping-off point
- Scaling vertically is very expensive
- Like Carriers, EC2 networking is fond of holding on to TCP teardown sequence packets
- vNICs obfuscate important data when you care about 1M connections
- Great for surge capacity
- Don’t split services into the virtual domain

Lessons Learned - Failing Well

- Scale vertically **and** horizontally
- Scale vertically but remember...
  - We can reliably take one Java process up to 990K open connections
  - What happens when that one process fails?
  - What happens when you need to do maintenance?



```
Critical
Blocking DoS attack
Blocking DoS attack
Blocking DoS attack
```

Thanks!

- Urban Airship http://urbanairship.com/
- Me @eonnen on Twitter or erik@urbanairship.com
- We’re hiring! http://urbanairship.com/company/jobs/

Additional UA Reading

Additional UA Reading

- Infrastructure Improvements - http://urbanairship.com/blog/2012/05/17/scaling-urban-airships-messaging-infrastructure-to-light-up-a-stadium-in-one-second/

Additional UA Reading

- Infrastructure Improvements - http://urbanairship.com/blog/2012/05/17/scaling-urban-airships-messaging-infrastructure-to-light-up-a-stadium-in-one-second/
- C500K - http://urbanairship.com/blog/2010/08/24/c500k-in-action-at-urban-airship/