From RoR & the Ruby VM to the JVM

Raffi Krikorian / @raffi
> 2.5E9 “deliveries” a day
Ruby on Rails
Sustainable productivity for web-application development
HAS

→ huge number of concurrent connections
→ lots of I/O
→ few persistent objects
Phasianus versicolor also known as Japanese Pheasant is a bird of the lowlands. Closely related to the Common Pheasant, the cock is distinguished by dark green plumage on breast and mantle. The male has an iridescent violet neck, red bare facial skin and purplish green tail. The female is smaller than male, and has a dull brown plumage with dark spots.
Building a Faster Ruby Garbage Collector

Since late 2009, much of www.twitter.com has run on Ruby Enterprise Edition, a modified version of the standard MRI 1.8.7 Ruby interpreter. At the time, we were working with the REE team to integrate some third-party patches that allowed us to tune the REE collector for long-lived workloads. We knew this was not a perfect choice, because a new runtime (even MRI 1.9x) would introduce compatibility problems, and it also indicated that alternative runtimes are not necessarily faster for our workloads. Nevertheless, the CPU cost of REE remained too high.

To address this problem, we decided to explore options for optimizing the REE collector. We called this effort Project Kji, after the Japanese bird.

INEFFICIENT GARBAGE COLLECTION

Our performance measurements revealed that even after our patches, the Ruby garbage collector uses a significant fraction of the CPU for running the garbage collector on Twitter. This is largely because MRI's garbage collector uses a single heap:

- The garbage collector's naive stop-the-world mark-and-sweep process accesses the entire memory set several times. It first marks all objects in the "root-set" level as "in-use" and then reexamines all the objects to release memory of those not in use. Additionally, the collector suspends the server during every sweep, thereby periodically "freezing" some of the program.
- The collection process is not generational. That is, the collector does not move objects between heaps; they all stay at the same address for the object's lifetime. The resulting fragmented memory extracts a penalty in bookkeeping cost because it can neither be consolidated nor discarded.

We needed to make the garbage collector more efficient but had limited options.
NEEDS...

- ability to handle server workloads
- an efficient language
NEEDS...

- ability to handle server workloads
- an efficient language
First chose fast, functional and expressive, statically typed, concurrent, and beautiful.
NEEDS...

ability to handle server workloads

flexibility in language
NEEDS...

- ability to handle server workloads
- flexibility in language
NEEDS...

- Ability to handle server workloads
- Flexibility in language
- A real concurrency model
IS AN EVENT DRIVEN & REAL-TIME PROBLEM
NEEDS...

ability to handle server workloads

finagle [fəˈnæɡəl]  
verb [ trans. ] informal  
obtain (something) by devious or dishonest means: Ted attended all the football games he could finagle tickets for.  
• [ intrans. ] act in a devious or dishonest manner: they wrangled and finagled over the fine points.

Finagle, from Twitter

Finagle is a network stack for the JVM that you can use to build asynchronous Remote Procedure Call (RPC) clients and servers in Java, Scala, or any JVM-hosted language. Finagle provides a rich set of protocol-independent tools.  

Finagle is written in Scala on top of Netty.
abstract class Service[-Req, +Rep] extends (Req => Future[Rep]) {
  ...

  /**
   * This is the method to override/implement
   * to create your own Service.
   */
  def apply(request: Req): Future[Rep]

  ...
}

val response = service(request)
service(request).onSuccess { response =>
  // compute result from response
  println("got response! " + response)
}
Has a home timeline

- figure out which tweets to show you
- get those tweets
- get the users that authored those tweets
timelineService(userId).andThen {
  tweetIds =>
  tweetService(tweetIds).andThen {
    tweets =>
    val userIds = getUserIds(tweets)
    userService(userIds).onSuccess {
      users =>
      // use users + tweets to produce JSON
    }
  }
}
Finagle's Substrate

- Connection management
- Protocol codecs
- Transient error handling
- Distributed tracing
- Service discovery
- Observability
ServerBuilder()
  .name("ServiceName")
  .reportTo(statsReceiver)
  .tracer(traceReceiver)
  .codec(Http())
  .maxConcurrentRequests(1000)
  .requestTimeout(500.milliseconds)
  .build(Service[Request, Response])
ClientBuilder()
  .cluster(TimelineServiceCluster)
  .hostConnectionCoresize(5)
  .hostConnectionLimit(10)
  .hostConnectionIdleTime(5.seconds)
  .retries(3)
  .timeout(500.milliseconds)
<table>
<thead>
<tr>
<th>Storage / Retrieval</th>
<th>Models / Biz Objects</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeline Storage</td>
<td>Tweet Storage</td>
<td></td>
</tr>
<tr>
<td>Tweet Storage</td>
<td>Social Graph</td>
<td></td>
</tr>
<tr>
<td>User Storage</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Twitter App

Timeline Service
Tweet Service
User Service

Timeline Storage
Tweet Storage
Social Graph
User Storage

Composition
Models / Biz Objects
Storage / Retrieval
SWITCHING TO

DOESN’T IMPLY THAT

IS A MISTAKE
In the final three minutes of the Super Bowl tonight, there were an average of 10,000 Tweets per second.
Madonna's performance during the Super Bowl's halftime show saw an average of 8,000 Tweets per second for five minutes.
The highest Tweets per second #SuperBowl peak came at the end of the game: 12,233. 2nd highest was during Madonna's performance: 10,245.