Device Simulator with Akka

@maxxhuang

@maxxhuang

Engineer @ Ruckus Wireless

Desperate to jump into Scala world

Agenda

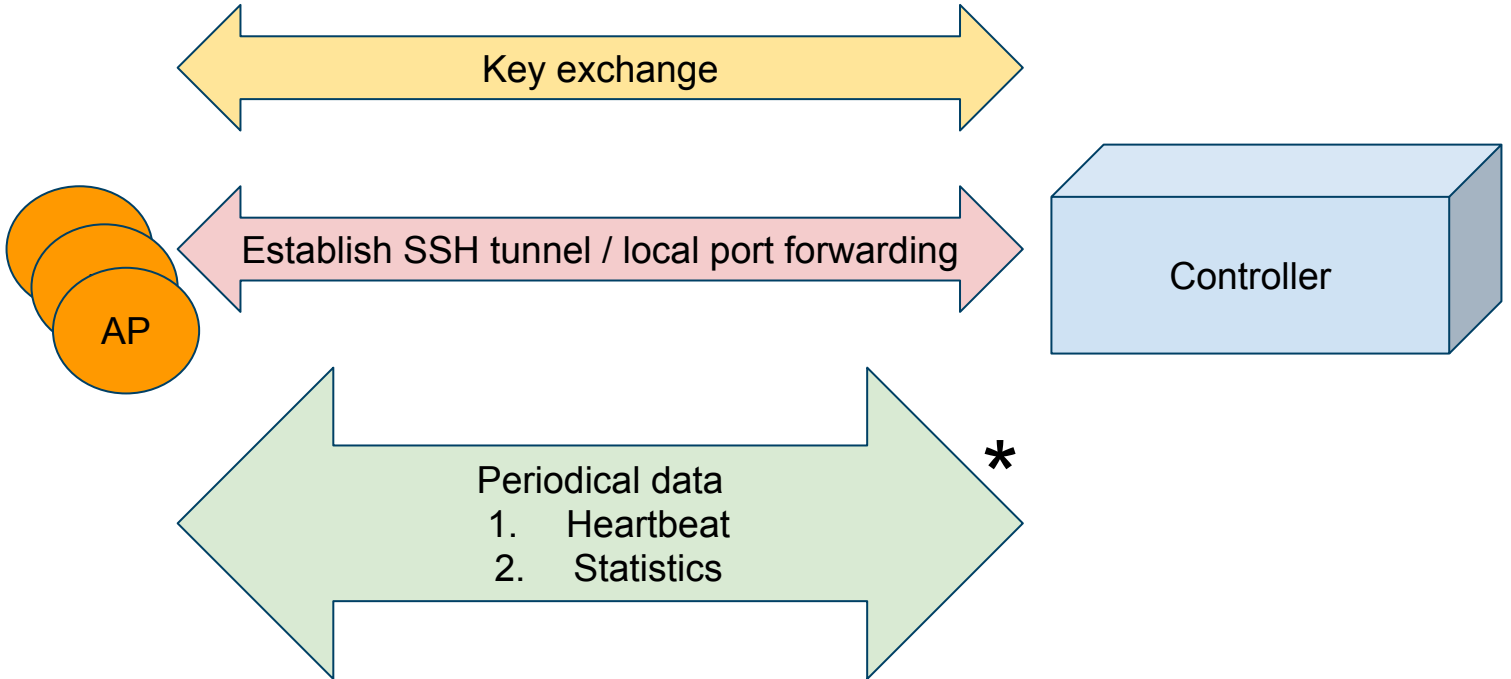
Architecture / Design of Device Simulator

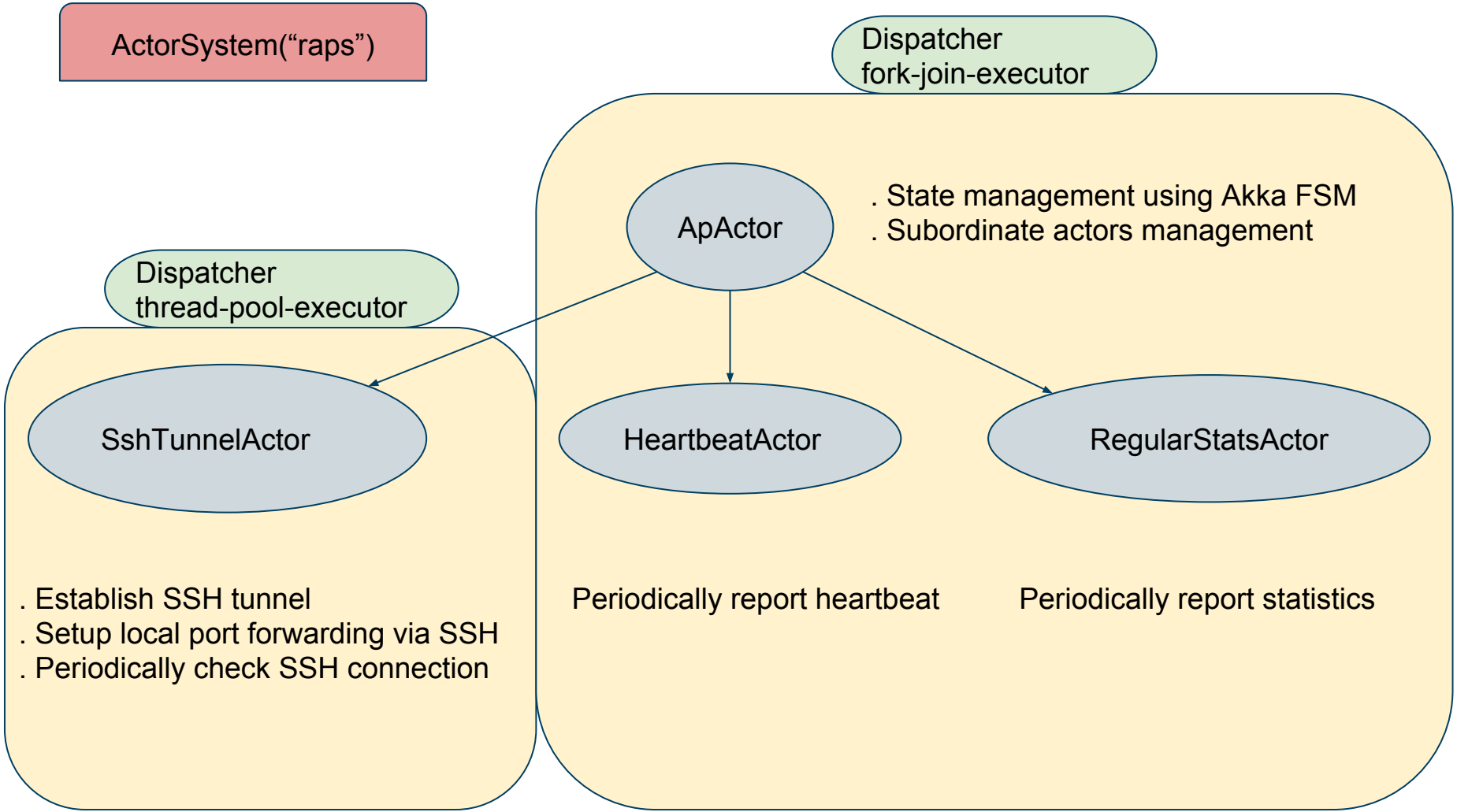
Troubleshooting

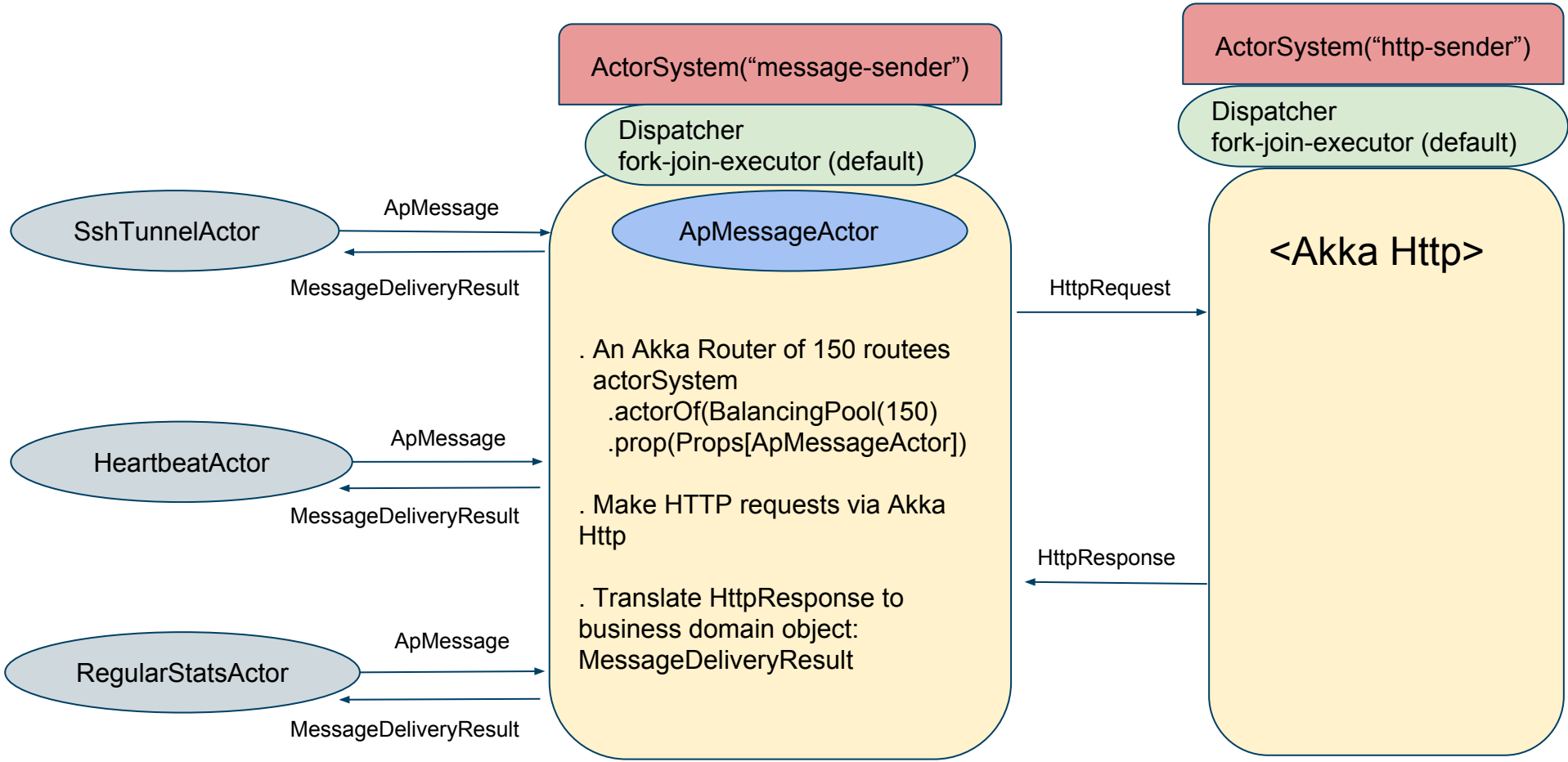
Monitoring

Performance Tuning

Architecture




ActorSystem(“raps”)
Dispatcher
fork-join-executor
ApActor
. State management using Akka FSM
. Subordinate actors management
Dispatcher
thread-pool-executor
SshTunnelActor
HeartbeatActor
RegularStatsActor
. Establish SSH tunnel
. Setup local port forwarding via SSH
. Periodically check SSH connection
Periodically report heartbeat
Periodically report statistics


ActorSystem(“message-sender”)
Dispatcher
fork-join-executor (default)
ApMessageActor
SshTunnelActor
ApMessage
MessageDeliveryResult
HeartbeatActor
ApMessage
MessageDeliveryResult
RegularStatsActor
ApMessage
MessageDeliveryResult
. An Akka Router of 150 routees
actorSystem
.actorOf(BalancingPool(150)
.prop(Props[ApMessageActor])
. Make HTTP requests via Akka Http
. Translate HttpResponse to business domain object: MessageDeliveryResult
HttpRequest
HttpResponse
ActorSystem(“http-sender”)
Dispatcher
fork-join-executor (default)
<Akka Http>

ApActor

- Manage a device’s lifecycle
- Delegate tasks to subordinate actors
  - SSH tunnel -> SshTunnel
  - Heartbeat -> HeartBeatActor
  - Regular stats -> RegularStatsActor
- Use Akka FSM as state machine

```
class ApActor(initApData: ApData)
  extends Actor with FSM[SimApState, ApData]

  startWith(DoDiscovery, initApData)

  when(DoSshTunnel) {
    case Event(SshTunnelEstablished(sshTunnelActor, localTunnelIp, localTunnelPort)data: ApData) =>
      val newApData = this.doSshTunnelHandler.handleTunnelEstablished(...)
      goto(DoRegularStats) using(newApData)
  }
  ...
  when(DoRegularStats) {
    case Event(ApConfigDownload(payload, latestSeqNumber, configPollingActor)data: ApData) =>
      val newApData = this.doConfigDownloadingHandler.downloadAllConfigs(...)
      goto(DoConfigDownloading) using newApData
  }

  whenUnhandled {
    case Event(forbidden: Forbidden, data: ApData) =>
      goto(DoDiscovery) using data.copy(state =0)

    case Event(HeartbeatLost, data: ApData) =>
      goto(DoSshTunnel)
  }

  onTransition {
    case _ -> DoSshTunnel =>
      val sshTunnelActor = this.doSshTunnelHandler.establishSshTunnel(nextStateData,self, context)
  }

  initialize()
}
```

Specify initial state/data

Event handling in each state

Event handling in each state

Unhandled event goes here

State entry action

Start state machine

FSM

The FSM (Finite State Machine) is available as a mixin for the Akka Actor and is best described in the Erlang design principles

A FSM can be described as a set of relations of the form:

State(S) x Event(E) -> Actions (A), State(S')

These relations are interpreted as meaning:

If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.

- *Akka Documentation*
  http://doc.akka.io/docs/akka/current/scala/fsm.html

FSM

- class ApActor extends Actor with **FSM[SimApState, ApData]**
- Event is a wrapper of incoming message and current state data

```
when(DoSshTunnel) {
 case Event(SshTunnelEstablished(...), data: ApData) =>
   val newApData =
this.doSshTunnelHandler.handleTunnelEstablished(...)
   goto(DoConfigPolling) using(newApData)
}
```

- Use '**goto ([next state]) using [next state data]**' or '**stay using [next state data]**' for state trasition with new state data

SshTunnelActor

- Use **Apache SSHD** for SSH tunnel connection and local port forwarding
  - Apache SSD is based on Apache Mina, a NIO client/server framework library
- Responsibility
  - Establish SSH connection & local port forwarding at startup
  - Check SSH connection every 10 seconds
  - Notify ApActor to re-establish SSH tunnel if SSH connection is lost

```
class SshTunnelActor(sshTunnelContext: SshTunnelContext) extends Actor {
  ...
  override def preStart(): Unit = {
   initTunnel()
   context.system.scheduler.schedule(10 seconds, 10 seconds, self, CheckConnection)
  }

  override def postStop(): Unit = closeSshConnection()

  override def receive: Receive = {
    case CheckConnection =>
      checkSshConnection() match {
        case Failure(e) =>
          sshTunnelContext.apActor ! ReestablishSshTunnel
          context.stop(self)

        case Success(_) =>
      }
  }
  ...
}
```

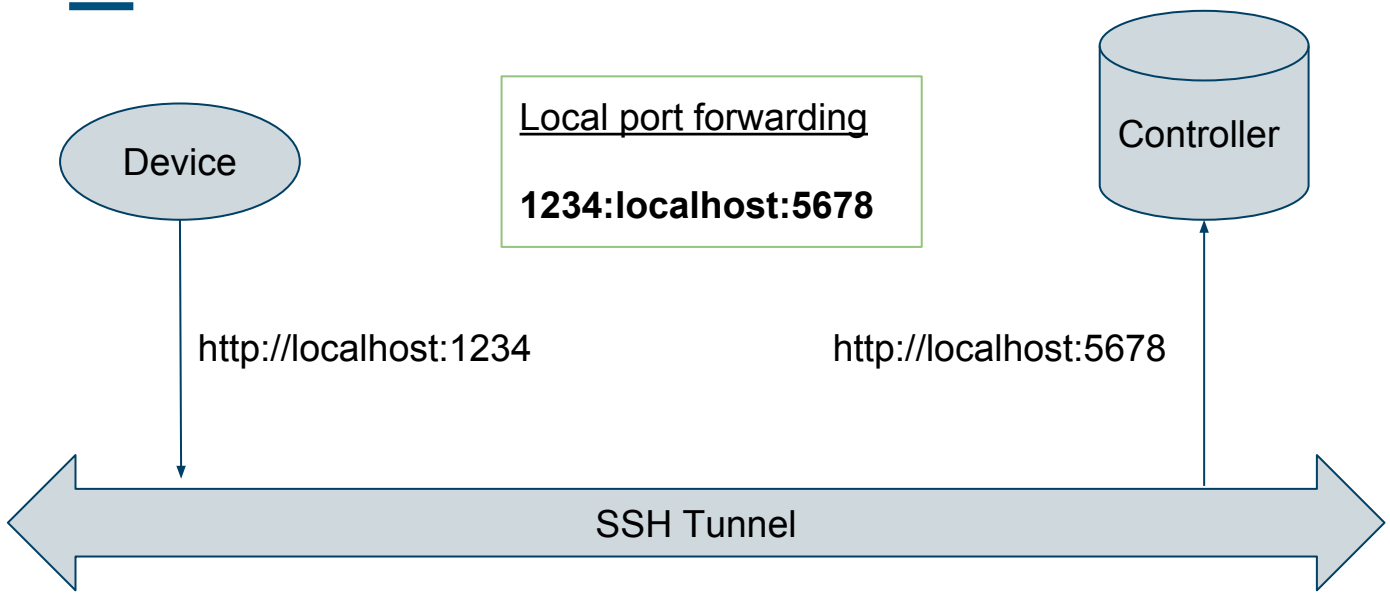
SshTunnelActor

- SSH connection establishment
  - A slow, one-time job
  - Occupy thread for a while at startup
- SSH connection checking
  - A fast, recurring task
- Use a dedicated Dispatcher for SshTunnelActor

Is it worth creating 2 different types of actors to perform connection establishment and connection checking separately?

```
ssh-tunnel-actor-dispatcher {
 type = Dispatcher

 executor = "thread-pool-executor"

 thread-pool-executor {
   core-pool-size-min = 50
   core-pool-size-max = 100
 }

 throughput = 1
}
```

Regular Stats over SSH Tunnel / Local Port Forwarding



SshTunnelActor

A set of threads are created (by Apache SSHD) for each device establishing SSH connection with local port forwarding.

For each device, Apache SSHD sets up a NIO server listening on a random port to serve requests via local port forwarding.

It was observed that 1000 devices would consume up to 8000 threads.

HeartbeatActor / RegularStatsActor

SshTunnelActor along with other subordinate actors share a common pattern in actor implementation.

```
class SubordinateActor extends Actor {
  // schedule a periodical notification to DoSomething
  def preStart = ...

  def receive = {
    case DoSomeThing => // do something
  }
}
```

```
class HeartbeatActor(heartbeatContext: HeartbeatContext) extends Actor with ActorLogging {

  override def preStart: Unit = {
    context.system.scheduler.schedule(0 seconds,
      heartbeatContext.intervalInSecond seconds, self, ReportHeartbeat)
  }

  override def receive: Receive = {

    case ReportHeartbeat =>
      val heartbeatMessage = ...

      heartbeatContext.httpMessageSender.sendMessage(heartbeatMessage, self)

    case result: MessageDeliveryResult =>

      if (result.successful)
        log.debug("{} sent heartbeat successfully.",
          heartbeatContext.apData.apMac)
      else
        log.error("{} failed to send heartbeat: {}",
          heartbeatContext.apData.apMac, result.content)
  }

}
```

```
class RegularStatsActor(regularStatsContext: RegularStatsContext) extends Actor {

  override def preStart(): Unit = {
    context.system.scheduler.schedule(0 second,
      regularStatsContext.intervalInSecond seconds, self, ReportRegularStats)
  }

  override def receive: Receive = {
    case ReportRegularStats =>
      sendRegularStatsMessage()
  }

  private def sendRegularStatsMessage(): Unit = {
    sendRegularData1()
    sendRegularData2()
    sendRegularData3()
  }

}
```

ApMessageActor

- An Akka Router of 150 routees
  - actorSystem.actorOf(BalancingPool(150).prop(Props[ApMessageActor])
- Send message to controller via Akka Http
- Translate HttpResponse to domain object:

```
sealed trait MessageDeliveryResult {
  def successful: Boolean
  def content: String
  def correlationId: CorrelationId
}

case class HttpResult(...) extends MessageDeliveryResult {}

case class DeliveryFailure(...) extends MessageDeliveryResult {...}

case class Forbidden(...) extends MessageDeliveryResult {}
```

Future.foreach does not handle failure case; it just ignore it
Future.onComplete() should be used to handle both success and failure cases.

```
class ApMessageActor extends Actor {
  override def receive: Receive = {
    case Envelope(message, reporter) =>
      HttpSender.send(message, reporter).onComplete {
        case Success(HttpResponse(statusCode, headers, entity, protocol)) =>
      ...
          entity.toStrict(extractPayloadTimeout).foreach { strictEntity =>
            val result =
              if (statusCode == StatusCodes.OK)
                HttpResult(statusCode, strictEntity.contentType, strictEntity.data,
message.correlationId)
              else if (statusCode == StatusCodes.Forbidden)
                Forbidden(strictEntity.data, message.correlationId)
              else {
                val errorMessage = s"Http Status Code: $statusCode, Message:
${strictEntity.data.utf8String}"
                DeliveryFailure(new RuntimeException(errorMessage), message.correlationId)
              }

            reporter ! result
          }

        case Failure(e) =>
          reporter ! DeliveryFailure(e, message.correlationId)
      }
}
```

ApMessageActor

- ApMessageActor’s tasks are eazy and simple, so default Dispatcher should be adequate.
  - Sending messages to Akka Http asynchronously
  - Receiving and translating HttpResponse to MessageDeliveryResult
- Most responses from controller are assumed to be small. ResponseEntity (Akka Http) is loaded in memory by converting it to HttpResponse.Strict: entity.toStrict([timeout]) without leveraging Akka Http’s streaming nature.

Akka HTTP

- 3 flavors of http client APIs
  - Connection-Level Client-Side API
  - Host-Level Client-Side API
  - Request-Level Client-Side API
- We take the simplest approach: request-level API
  - Http().singleRequest()

Start the engine

1000 devices

Lots of “Too many open files” popped up in the log

Each network socket consumes a file descriptor. When the limit of max file descriptor is reached, the process can not access any more files or network sockets.

Modify **/etc/security/limits.conf** to allow for more file descriptors.

```
maxx            soft    nofile          204800
maxx            hard    nofile          204800
```

Restart the engine

Very slow to establish SSH tunnels

Observing the thread dump, most threads doing SSH connection are stuck at SecureRandom

```
"raps-ssh-tunnel-actor-dispatcher-24" #152 prio=5 os_prio=0 tid=0x00007ff5c8033000 nid=0x17e6 waiting
for monitor entry [0x00007ff5a77b7000]
  java.lang.Thread.State: BLOCKED (on object monitor)
  at sun.security.provider.NativePRNG$RandomIO.implGenerateSeed(NativePRNG.java:426)
  - waiting to lock <0x00000006c946e370> (a java.lang.Object)
  at sun.security.provider.NativePRNG$RandomIO.access$500(NativePRNG.java:329)
  at sun.security.provider.NativePRNG.engineGenerateSeed(NativePRNG.java:224)
  at java.security.SecureRandom.generateSeed(SecureRandom.java:533)
  ...
  at org.apache.sshd.client.SshClient.setUpDefaultClient(SshClient.java:758)
  ...
  at com.ruckuswireless.raps.actor.MinaSshTunnelActor.initTunnel(MinaSshTunnelActor.scala:101)
```

SecueRandom

- By default, SecureRandom reads /dev/random for entropy generating in generateSeed() or nextBytes()
- /dev/random is a blocking device. On the other hand, /dev/urandom is non-blocking.
- The difference in general
  - /dev/random: blocking, more secure
  - /dev/urandom: non-blocking, less secure
- Specify entropy gathering device in $JRE_HOME/lib/security/java.security
  - securerandom.source=file:/dev/urandom
- Reference
  - http://stackoverflow.com/questions/137212/how-to-solve-performance-problem-with-java-securerandom
  - http://leaver.me/2015/06/30/SecureRandom%E7%AC%AC%E4%B8%80%E6%AC%A1%E7%94%9F%E6%88%90%E9%9A%8F%E6%9C%BA%E6%95%B0%E9%9D%9E%E5%B8%B8%E6%85%A2/

Let’s hit the road…again

All 1,000 devices connected to the controller with SSH tunnel established, and started reporting regular stats

But, dozens to hundreds of devices kept going up and down. This is the consequence resulting from the failure of sending heartbeats to controller.

Error messages swarmed in the log.

- AB:CD:EF:00:01:EA failed to responded heartbeat: No elements passed in the last 1 minute.
- Tcp command [Connect(127.0.0.1:47098,None,List(),Some(10 seconds),true)] failed
- [o.a.s.c.session.ClientSessionImpl] - Disconnecting(ClientSessionImpl[sshtunnel@/x.x.x.x:22]): SSH2_DISCONNECT_PROTOCOL_ERROR - User session has timed out idling after 600000 ms.

We need more information to figure out what is going on with the simulator.

Kamon

The Open Source tool for monitoring applications running on the JVM.

- Key features
  - Provide a clean and simple API for recording metrics and trace information for any application running on the JVM.
  - Bytecode instrumentation modules that automatically measure and trace your application. We have modules for Scala, Akka, Spray and Play!
  - Ships with several reporting backends (including StatsD, Datadog and New Relic) and allows you to create your own reporters in a breeze.
- Kamon documentation kttp://kamon.io

Kamon

To monitor the application using Kamon, the following has to be set up:

1. Use Kamon metrics APIs to manually or Kamon built-in modules(kamon-akka, kamon-system-metric…) to automatically collect metrics of the application.
2. Install and set up servers (**statsd**, datadog...) that can receive metrics reported by Kamon (kamon-statsd, kamon-datadog)
3. For statsd, additionally install and set up **Graphite** to store metrics data and **Grafana** for visualizing metrics.

Kamon

It’s a lot of setup and configuration to be able to “see” the Kamon metrics rendered in a graph presented in a dashboard.

Kamon provides a docker image to save us from all the setup work. https://github.com/kamon-io/docker-grafana-graphite

Kamon in our case

kamon-system-metrics and kamon-akka are used for metrics collecting.

In addition to built-in metrics collecting, we manually collect http-response time and http-error count.

Kamon configuration is based on the one from https://github.com/muuki88/activator-akka-kamon, so that I don't need to write the configuration from scratch.

Basically it's automatically collecting metrics from system resources (cpu memory, networking traffic), all actors, routers and dispatchers.

```
object ApMessageActor {
  case class Envelope(message: ApMessage, reporter: ActorRef)

  val httpResponseTimeHistogram = Kamon.metrics.histogram("http-response-time")
  val httpErrorCounter = Kamon.metrics.counter("http-errors")

}

class ApMessageActor extends Actor {

  override def receive: Receive = {

    case Envelope(message, reporter) =>
      ...
      val startTime = System.currentTimeMillis()

      HttpSender.send(message, reporter).onComplete {
        case Success(HttpResponse(statusCode, headers, entity, protocol)) =>
          ...
          GroupApMessageDeliveryActor.httpResponseTimeHistogram.record(System.currentTimeMillis() -
startTime)
          ...
        case Failure(e) =>
          ...
          GroupApMessageDeliveryActor.httpResponseTimeHistogram.record(System.currentTimeMillis() -
startTime)
          GroupApMessageDeliveryActor.httpErrorCounter.increment()
      }

  }
```

```
kamon.metric {
  # Time interval for collecting all metrics and send the snapshots to all subscribed actors.
  tick-interval = 10 seconds // Kamon reports metrics to backend (statsd) every 10 seconds

  disable-aspectj-weaver-missing-error = false

  # Specify if entities that do not match any include/exclude filter should be tracked.
  track-unmatched-entities = yes

  filters {
    akka-actor {
      includes = ["*/user/*"]
      excludes = [ "*/system/**", "*/user/IO-**", "*kamon*" ]
    }

    akka-router {
      includes = ["*/user/*"]
      excludes = []
    }

    akka-dispatcher {
      includes = ["**"] // includes = ["*/user/*"]
      excludes = []
    }

    trace {
      includes = [ "**" ]
      excludes = [ ]
    }
  }
}
```

Carry on with the journey

After a few minutes, the docker ran out of disk space and finally crashed.

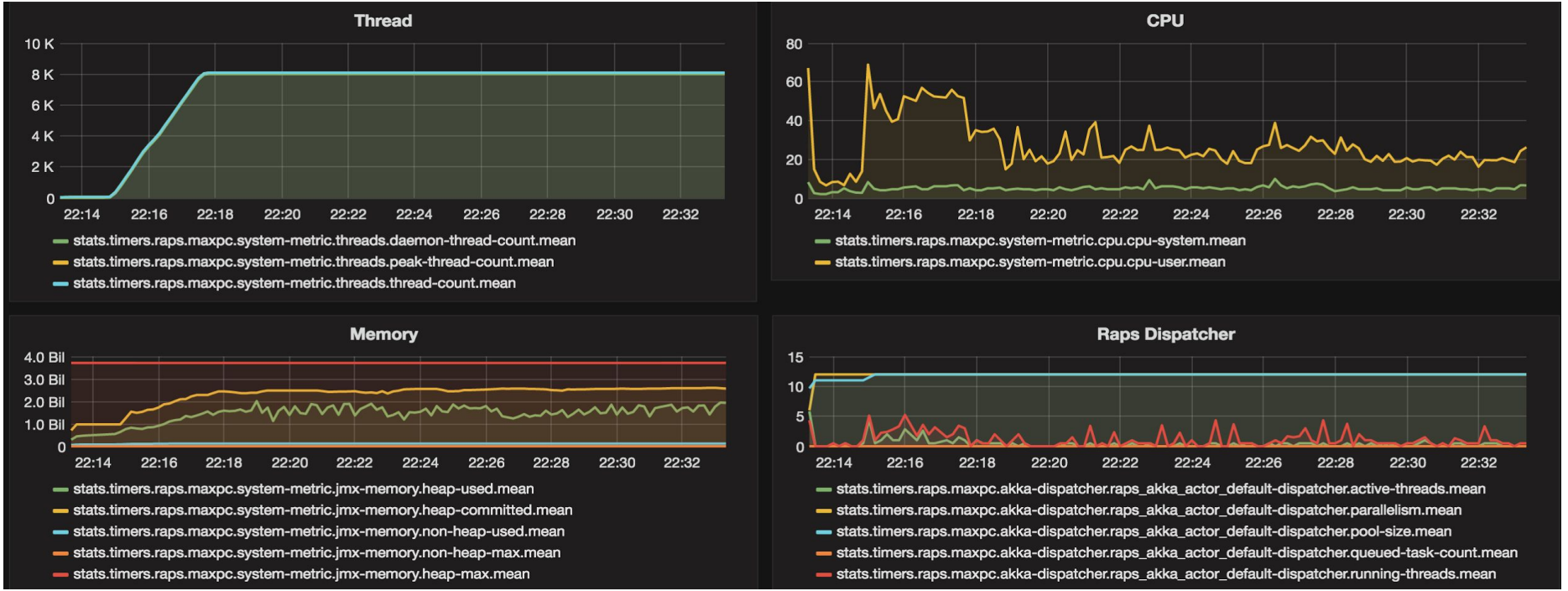
Suspect the metrics produced by actors (4000 for 1000 devices) overwhelmed the docker container.

Being able to examine metrics from 4000 actors does not give you much insight. The information is just too much.

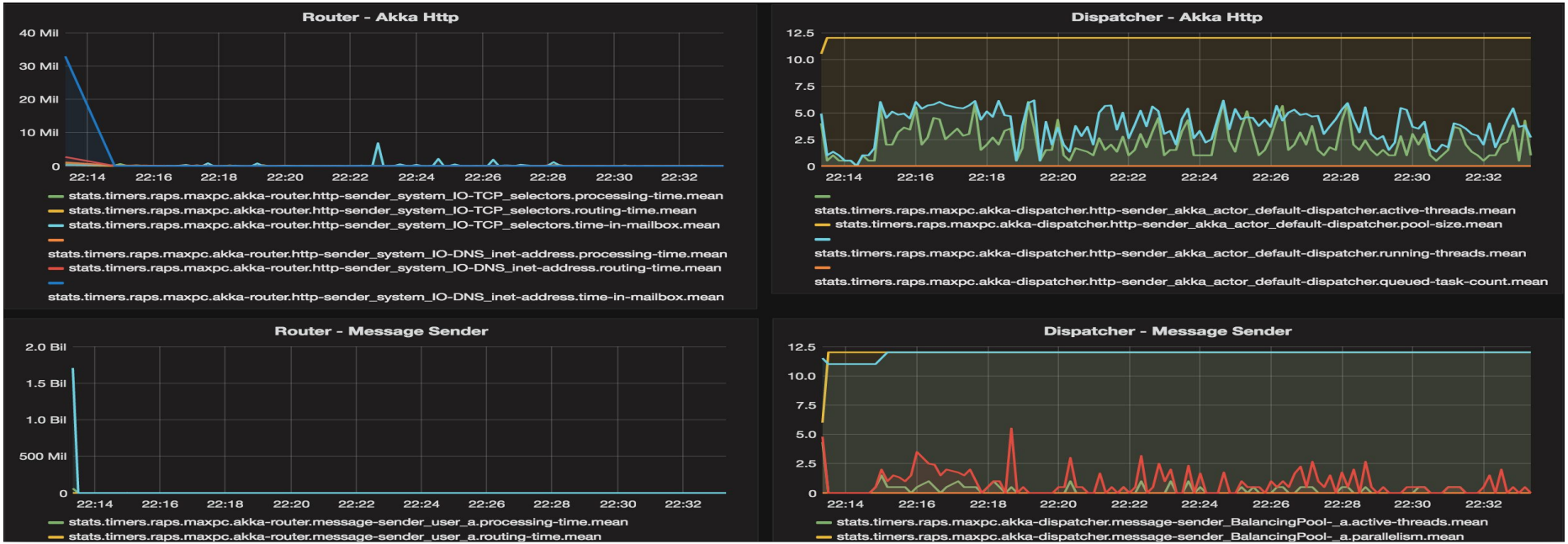
There is a plan to provide actor-group metrics.
https://github.com/kamon-io/Kamon/issues/101

Disable akka actor metrics to avoid data flooding the docker container.
kamon.metric.filters.akka-actor.includes = []

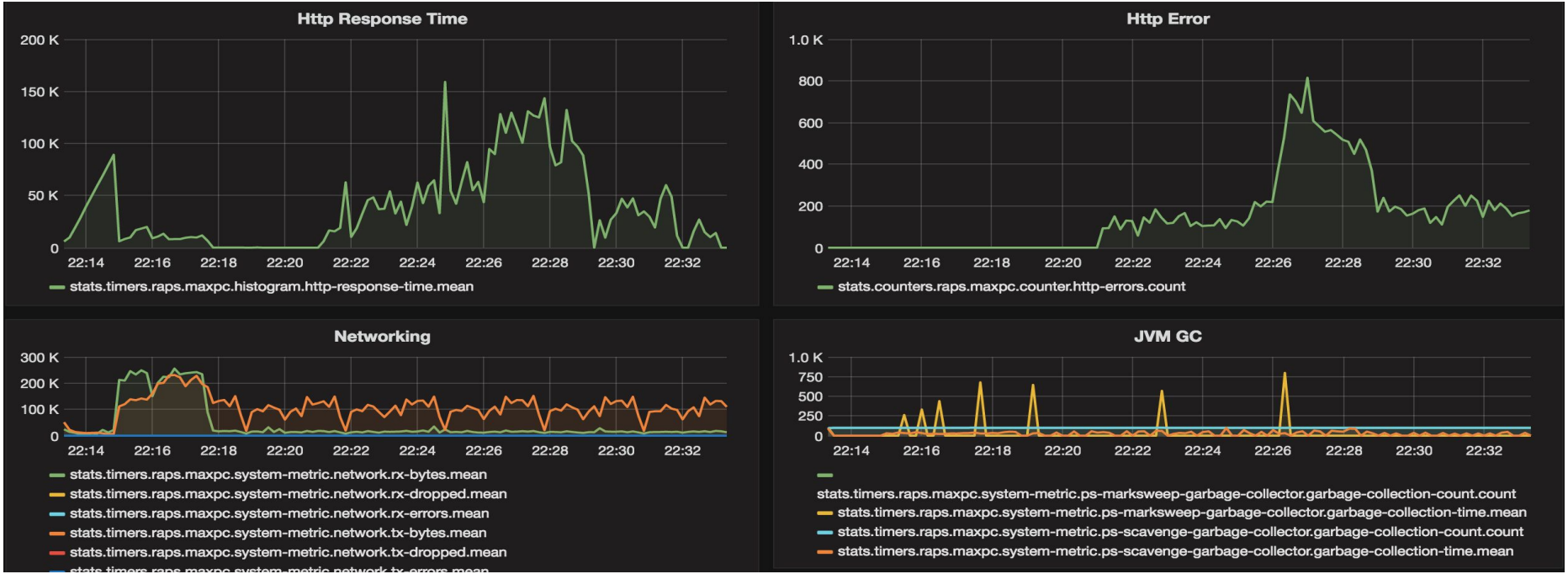
Move on...



- Thread count is a constant 8000 after all devices established SSH tunnels.
- CPU usage is under 30%
- Memory usage is around 1.8G (1.8 billion bytes)
- Thread usage of the dispatcher (for ApActor, HeartbeatActor, RegularStatsActor) is low.



- Akka Http
  - Except for the time of Akka Http initialization, message processing-time and time-in-mailbox is kept under 1 milli-second.
  - Thread count usage is under 7
- ApMessageActor
  - Except for the time of Akka Http initialization, message processing-time and time-in-mailbox is less than 1 milli-second.
  - Thread count less than 3 on average



- From 22:21, it took tens of seconds to finish a http request. The worst cases are up to 150 seconds (150 K milli-seconds)!
- From 20:21, Akka Http errors emerged. The worst cases are 800 errors per 10 seconds. (Kamon reports aggregated error count every 10 seconds and then resets the counter)

What do we have so far…

1. Prolonged http response time
2. Lots of http errors
3. Akka Http seemed not busy:
    a. message processing-time / time-in-mailbox is low
    b. thread usage is normal
4. ApActors and subordinate actors are not busy either.

Maybe the SSH connection is lost, and we are not aware of that!

We had "[o.a.s.c.session.ClientSessionImpl] - Disconnecting(ClientSessionImpl[sshtunnel@/x.x.x.x:22]): SSH2_DISCONNECT_PROTOCOL_ERROR - User session has timed out idling after 600000 ms." in the log.

But we did not see any ssh-reconnecting logs from SshTunnelActor.

It seems that SshTunnelActor failed to catch SSH disconnection.

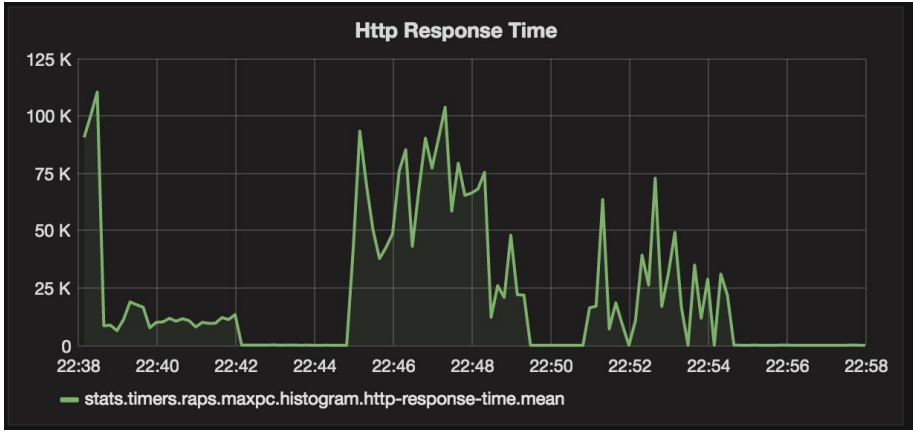
The naive SSH connection checking does not work:
org.apache.sshd.client.SshClient.isOpen()

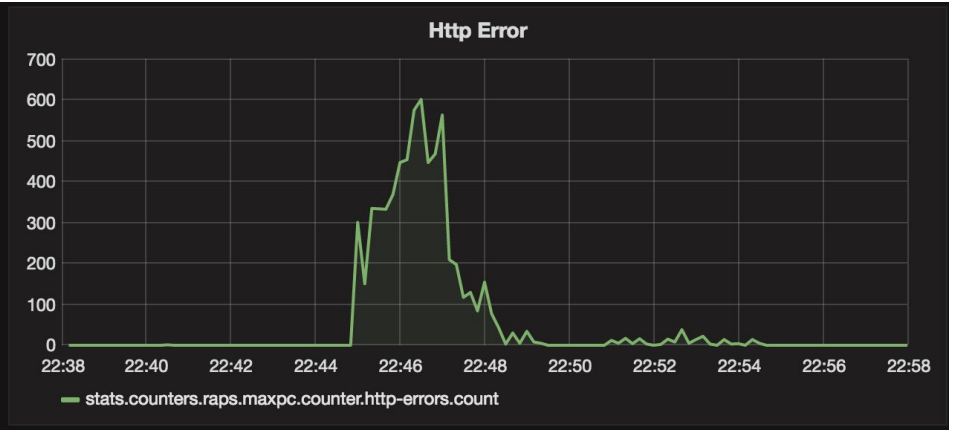
I tried adding a PortForwardingEventListener to catch SSH disconnection but it does not work either.

Last resort is to passively reconnect SSH connection on the failure of heartbeat sending.

```
class HeartbeatActor(heartbeatContext: HeartbeatContext) extends Actor with ActorLogging {

  override def receive: Receive = {

    case ReportHeartbeat => ...
    case result: MessageDeliveryResult =>

      if (result.successful) ...
      else {
        // ApActor, on receiving HeartbeatLost, will go to ssh-tunnel state to rebuild SSH connection
        heartbeatContext.apActor ! HeartbeatLost
        context.stop(self)
      }
  }

}
```

Head toward the dawn





Unstable during the period of time (22:45 - 22:48). Hundreds of devices might be considered disconnected due to the failure of heartbeat reporting.

Slightly unstable during the period of time (22:48 - 22:55). Only several devices are seen going up and down.

No more ssh-session-timeout errors:
“[o.a.s.c.session.ClientSessionImpl] - Disconnecting(ClientSessionImpl[sshtunnel@/x.x.x.x:22]): SSH2_DISCONNECT_PROTOCOL_ERROR - User session has timed out idling after 600000 ms.”

It’s getting a lot better.

But...let’s be honest. We just mitigated the issue (heavily), not solved it.

What’s with the bump of http-errors during (22:45 - 22:48)?

Error messages are still observed

- AB:CD:EF:00:01:EA failed to responded heartbeat: No elements passed in the last 1 minute.
- Tcp command [Connect(127.0.0.1:47098,None,List(),Some(10 seconds),true)] failed

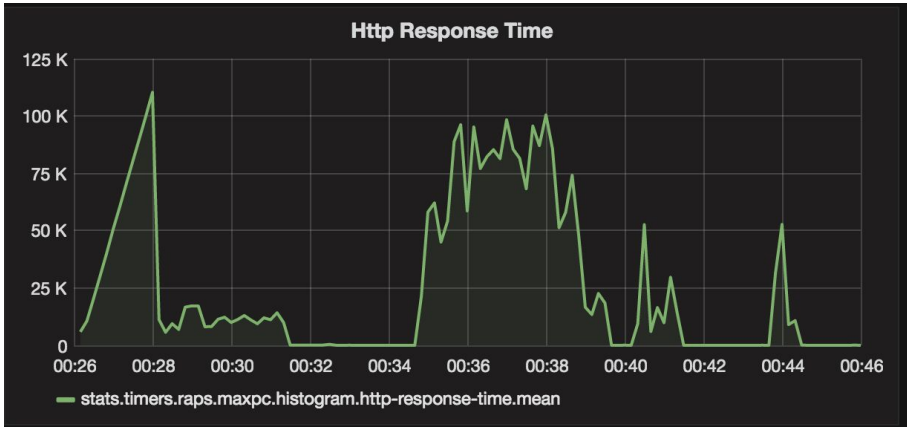
“Tcp command [Connect(127.0.0.1:47098,None,List(),Some(10 seconds),true)] failed”

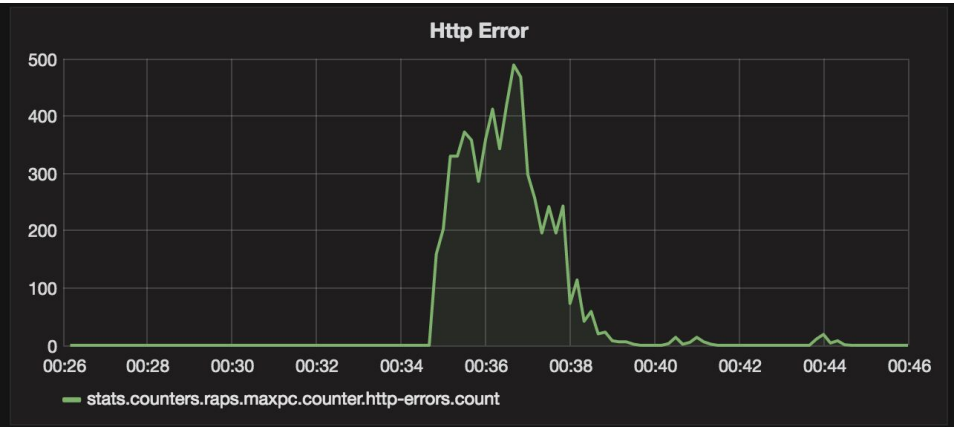
This complaint came from Akka HTTP failing to make a connection within 10 seconds, which is the default value of “connecting-timeout”.

Let’s extend the timeout to see if we can reduce this kind or errors.

akka.http.host-connection-pool.client.connecting-timeout = 30s

Almost there!





Out of luck!
Things did not change a bit!

Another clue is the error message:
AB:CD:EF:00:01:EA failed to respond heartbeat: No elements passed in the last 1 minute.

This is the value of “idle-timeout”, after which, Akka Http will close this idle connection (pool).

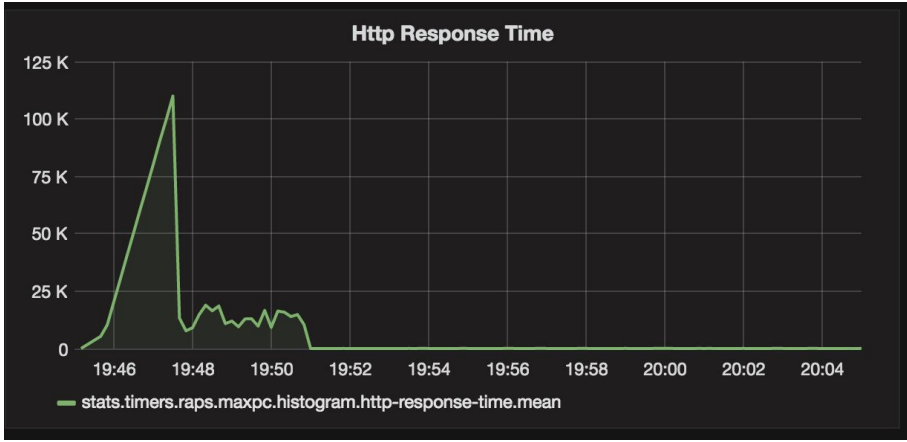
It occurred to me that each device, after SSH connection is created would pause 1 minutes before it starts reporting regular stats.

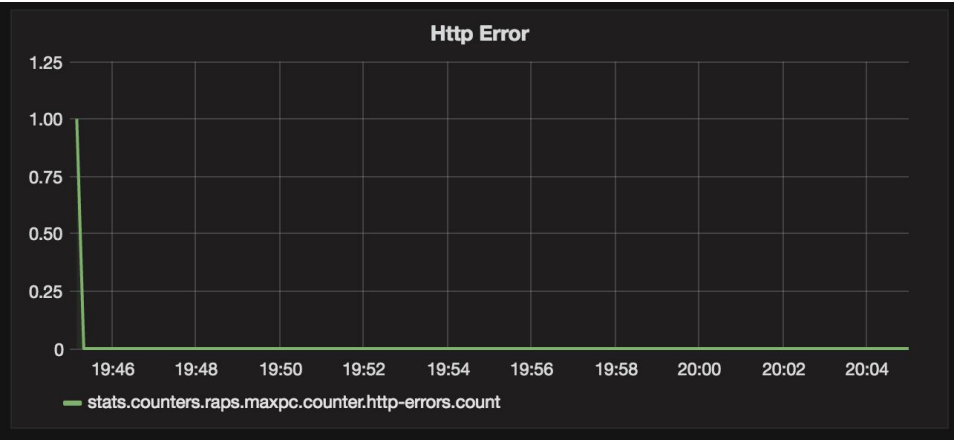
Let’s double the timeout in an attempt to avoid this kind of errors.

```
 # The time after which an idle connection pool (without pending requests)
 # will automatically terminate itself. Set to `infinite` to completely disable idle timeouts.
akka.http.host-connection-pool.idle-timeout = 120s

 # The time after which an idle connection will be automatically closed.
 # Set to `infinite` to completely disable idle timeouts.
akka.http.host-connection-pool.client.idle-timeout = 120s
```

akka.http.host-connection-pool.idle-timeout is the setting that matters in this case. The other one (akka.http.host-connection-pool.client.idle-timeout) does not solve the problem.

Finally...

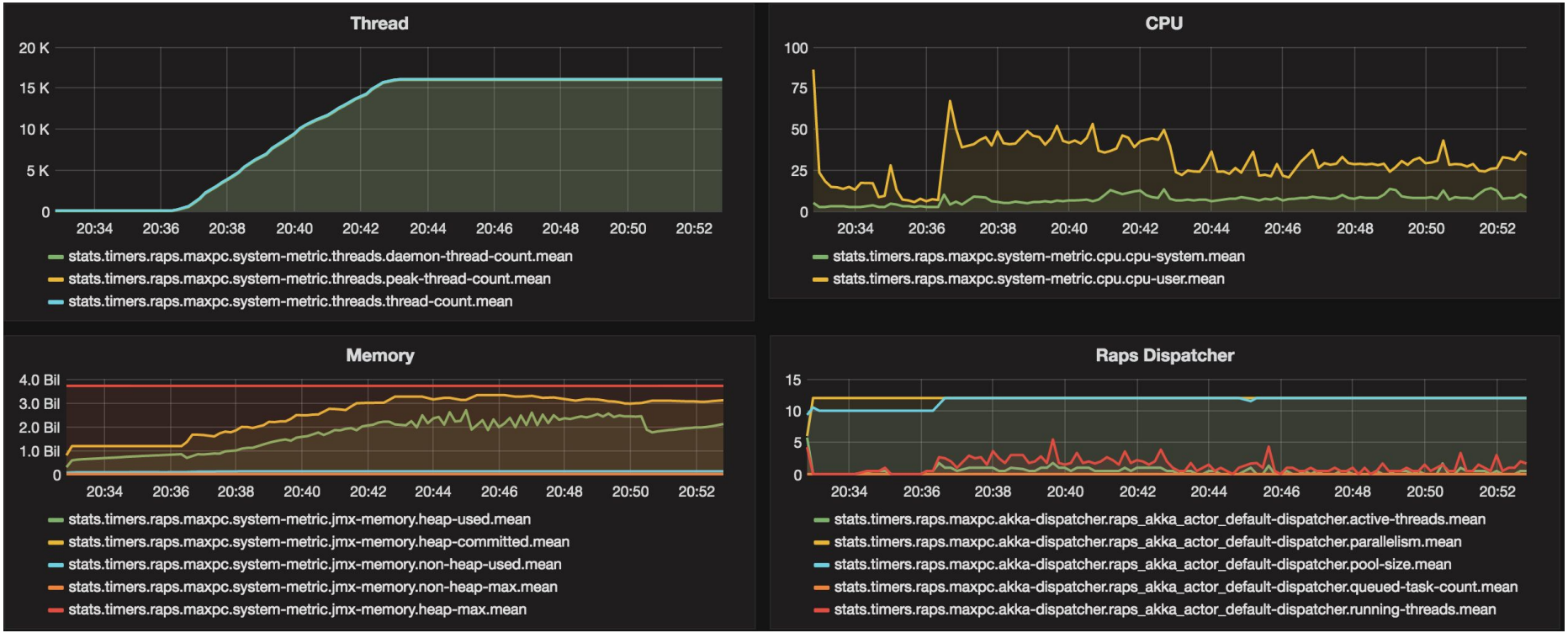




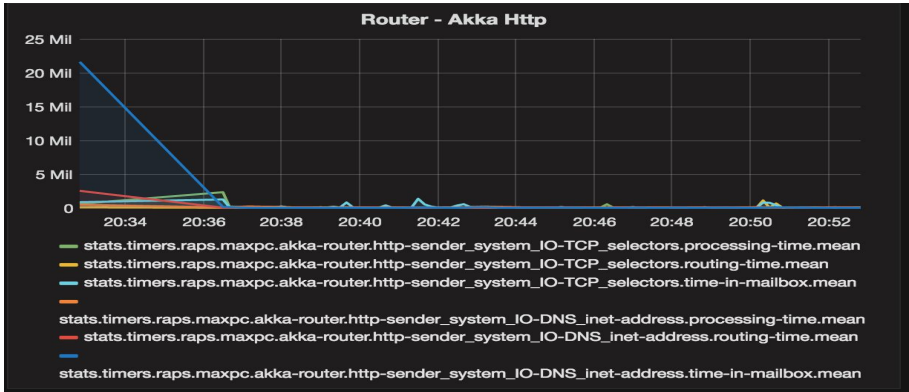
Works like a charm!

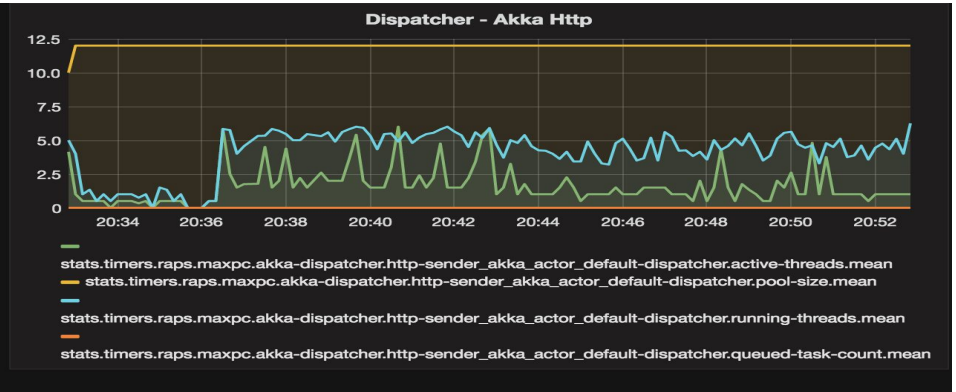
All 1000 devices were connected to the controller and steadily reported heartbeats / regular stats

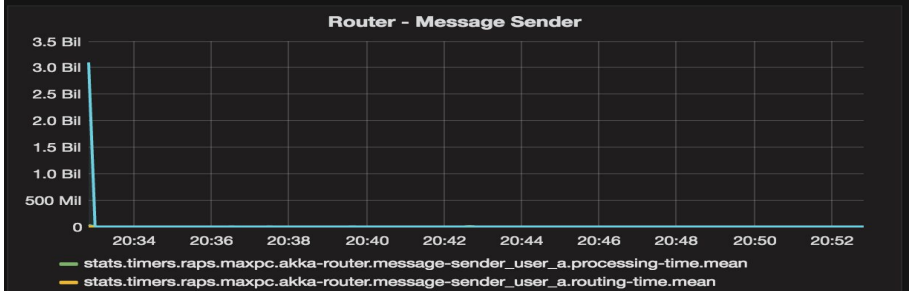
What about 2000 devices?

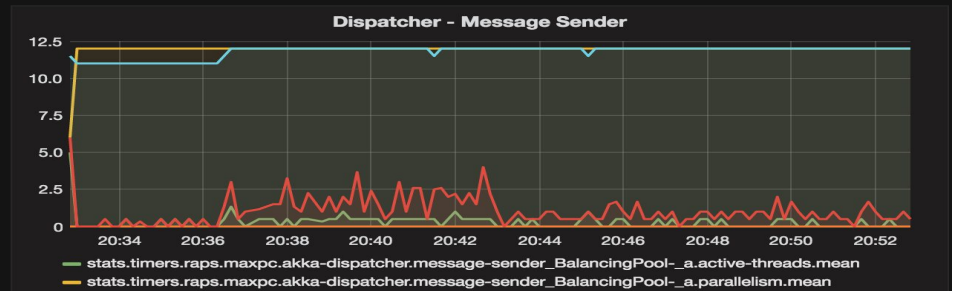


- Thread count climbs up to 16,000
- CPU usage slight increases 5%
- Memory roughly increases 500 MB
- Thread usage of Raps dispatcher does not seem to change

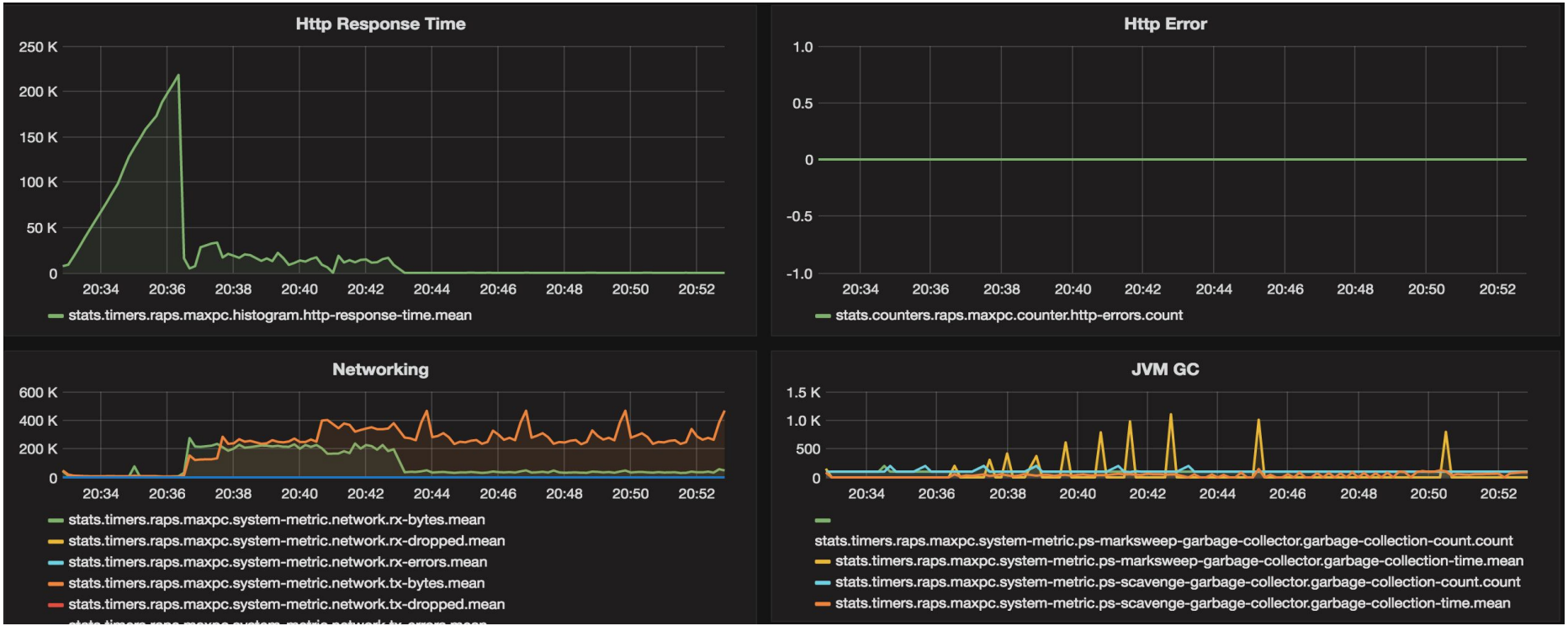








- Akka Http
  - Message processing-time and time-in-mailbox does not change.
  - Thread usage does not change.
- ApMessageActor
  - Message processing-time and time-in-mailbox does not change.
  - Thread usage does not change.



- Except for the time of Akka Http initialization, http-response time is still quick.
- No http errors!
- Networking traffic increases.
- GC activities increases.

Thank you!