The Art of Powering the Internet's Next Messaging System

March 27th, 2015

Who are we?

Juan Puig Martínez

- Senior Software Engineer
- 5+ years of Erlang/OTP
- @jpuigm

Mubarak Seyed

- Software Engineer
- Committer - Apache Flume
- Distributed Systems
- @mubarakseyed

Agenda

- What is Layer?
- SPDY
- Message Routing
- Data infrastructure
- Performance
- Challenges and lessons learned
- Q&A

We are building...

The Open Communication Layer for the Internet

Client SDKs that make it dead simple to add messaging and communication features to your application.

What is so hard about communications?

Pretty much everything...

Client Side Storage

Network Transport

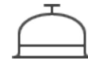
Cloud Storage

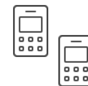
Data Sync

Offline States

Global Infrastructure

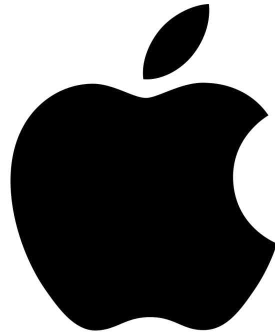
Security


Push Notifications


Multiple Devices

Scalability

… Across all platforms

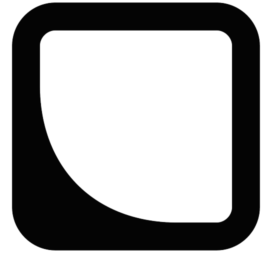


Fundamental building blocks

Maps

stripe

Payments



Communications

You focus on the customer experience.

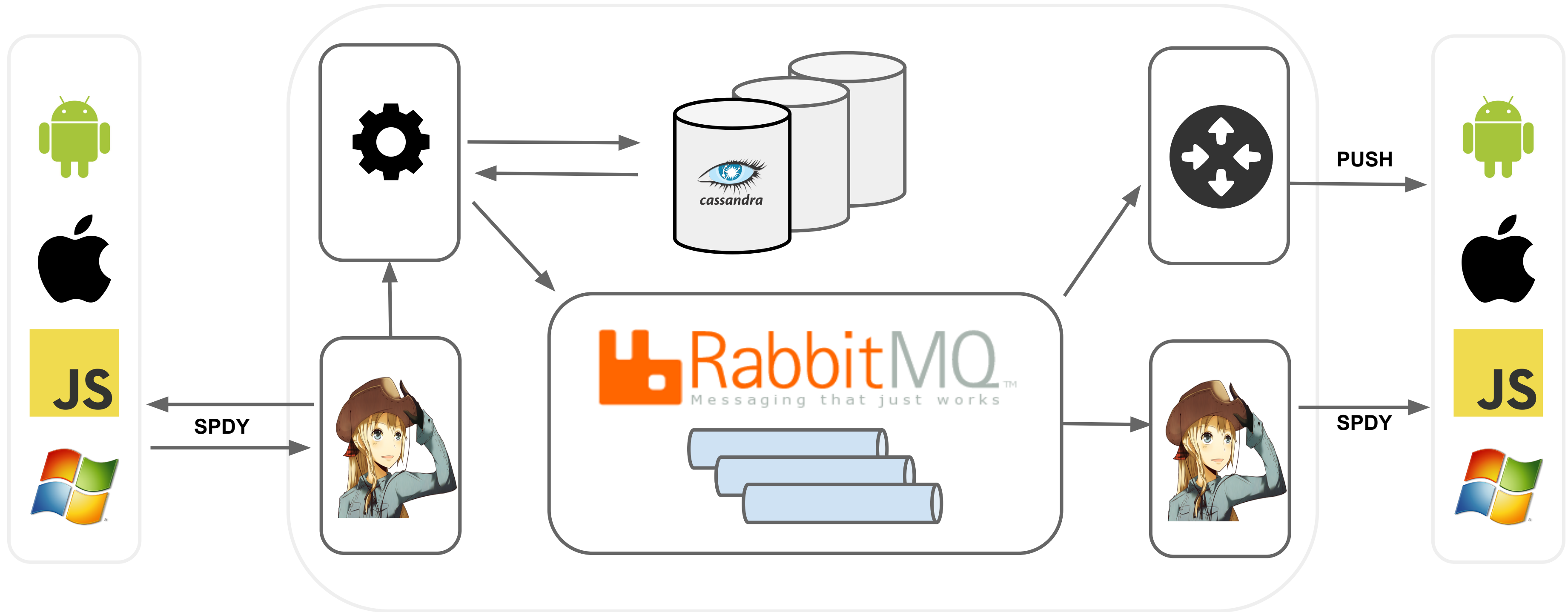
We'll focus on infrastructure, scalability, and security.

Layer – Software Stack

Why Erlang?

- Naturally born for **communications**
- **Fault tolerance**
  - Help us focus on HA using best OTP practices
- **Concurrency model**
  - Handles large number of concurrent lightweight processes
- **Bit syntax**
  - Binary protocol pattern matching

Layer’s backend in a nutshell

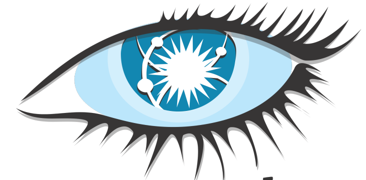


Architecture – Key components

SPDY

Routing

Infra



SPDY and Cowboy

SPDY

- Basis for HTTP/2
  - Connection management
  - Data transfer formats
- Layer relies on SPDY protocol to transport content
  - Latency reduction
    - Compression (headers)
    - Multiplexing

SPDY – A bit of history

- There are a few drafts to date (v1, v2, v3, v3.1 and v4)
- Most implementations are based on v3 and v3.1
- Spring 2013, experimental SPDY support is added to cowboy

- Summer 2013, latest SPDY version (v4) is released
- HTTP/2?

Cowboy + SPDY

- ninenines/cowboy - `cowboy_spdy.erl`
  - Protocol implementation
  - Loop that handles frames, and coordinates replies
- ninenines/cowlib - `cow_spdy.erl`
  - Protocol manipulation
  - Parsing/building of frames, streams, headers and settings

Cowboy + SPDY

ninenines / **cowboy**

Watch 228 | Star 1,999 | Fork 530

History for **cowboy** / src / **cowboy_spdy.erl**

Commits on Feb 16, 2015

- **Merge branch 'add_spdy_record_field_type' of https://github.com/sile/...** — essen authored 29 days ago — e74120b

Commits on Feb 3, 2015

- **Use cowlib master** — essen authored on Feb 1 — 3cede06

Commits on Dec 4, 2014

- **Add typespecs for state record in cowboy_spdy module** — sile authored on Dec 4, 2014 — c0672b8

Commits on Nov 7, 2014

- **Rename 'halt' to 'stop' for better consistency** — essen authored on Nov 7, 2014 — 999dc5b

Commits on Oct 3, 2014

- **Replace some /binary to /bits in binary pattern matching** — essen authored on Oct 3, 2014 — bee5ca8

Commits on Sep 24, 2014

- **Remove the error tuple return value for middlewares** — essen authored on Sep 24, 2014 — c56bada
- **Remove the onrequest hook** — essen authored on Sep 24, 2014 — aa4d86b

Commits on Jul 12, 2014

- **Reply with 400 on header parsing crash** — essen authored on Jul 12, 2014 — 97a3108

Commits on Jul 7, 2014

- **Merge branch 'fix-spdy-parse-frame' of git://github.com/voluntas/cowboy** — essen authored on Jul 7, 2014 — fd423eb

Commits on Jun 28, 2014

- **Fix cowboy_spdy parse frame** — voluntas authored on Apr 20, 2014 — fec3355

Commits on Jun 10, 2014

- **Fix specs and a weird value in cowboy_spdy** — essen authored on Jun 10, 2014 — 7cd3ecc

Cowboy + SPDY

- SPDY defines advanced (optional) features:
  - Server push
    - Multiple replies to a client for a single request
  - Rate limiting
    - `SETTINGS` frame
    - `SETTINGS_MAX_CONCURRENT_STREAMS` parameter
    - Max number of concurrent streams that sender will allow (directional, and defaulted to unlimited)
  - Flow control
    - `WINDOW_UPDATE` frame
    - Integer limiting how many bytes of data sender is allowed to transmit

Cowboy + SPDY

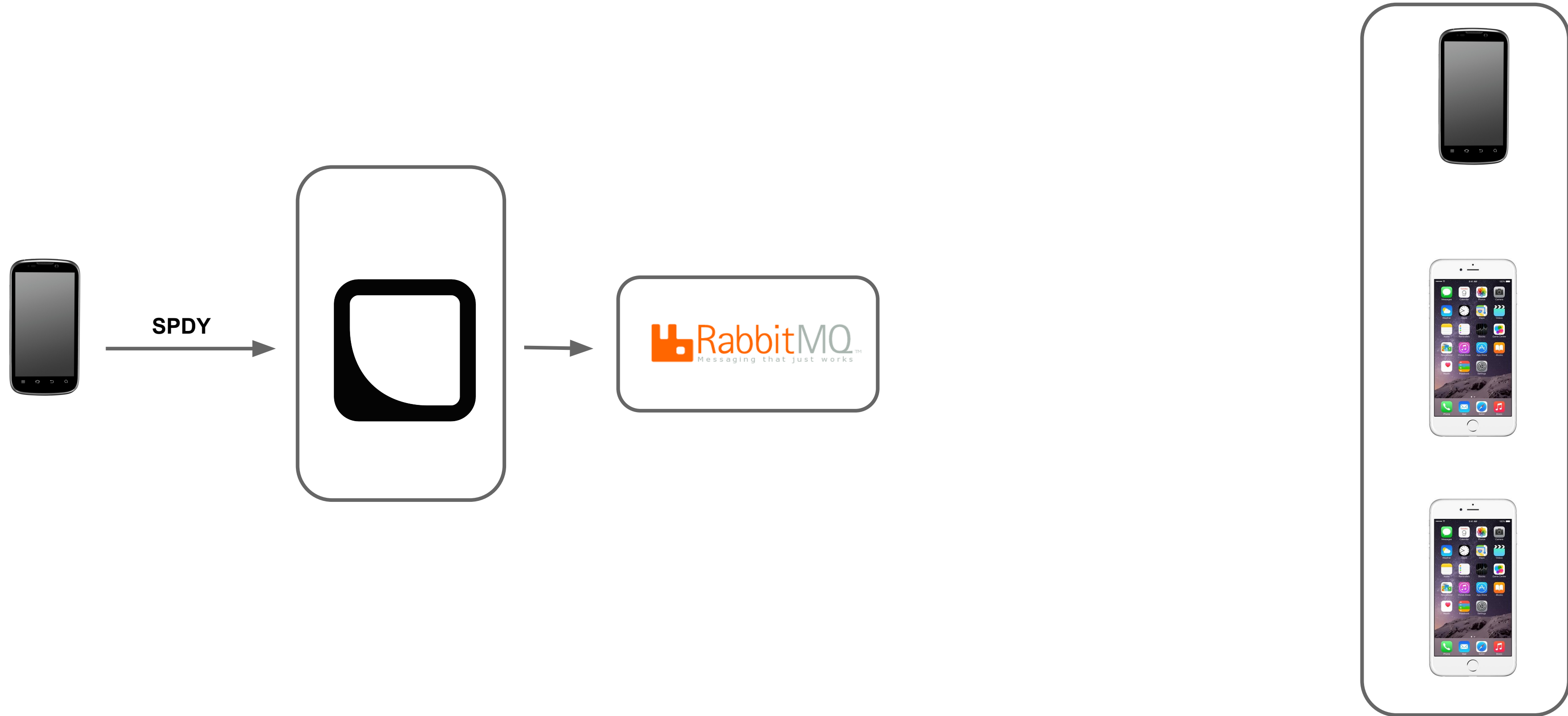
- We forked ninenines/cowboy –> layerhq/cowboy
- Implementation of advanced protocol features (SPDY v.3.1)
  - Server push
  - Flow control
  - Rate limiting (PR submitted)

Message Routing

Layer’s messaging platform publishes every single message to a RabbitMQ broker.

```erlang
send_message(Channel, Exchange, RoutingKey, Headers, Body, DeliveryMode) ->
    UglyHeaders = proplist_to_amqp_headers(Headers),
    amqp_channel:cast(Channel,
                      #'basic.publish'{exchange=Exchange, routing_key=RoutingKey},
                      #amqp_msg{props=#'P_basic'{headers=UglyHeaders,
                                                 delivery_mode=DeliveryMode},
                                payload=Body}).
```

RabbitMQ

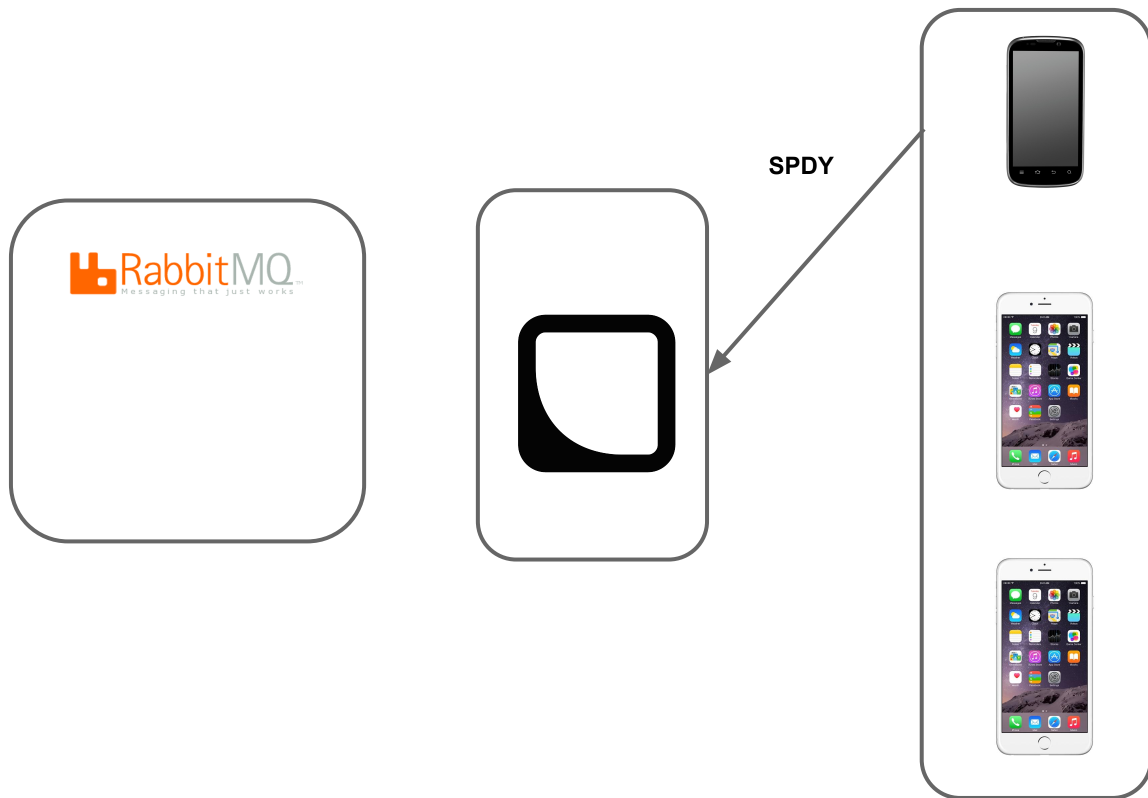


When a device connects to Layer,
a queue for that device is
created.

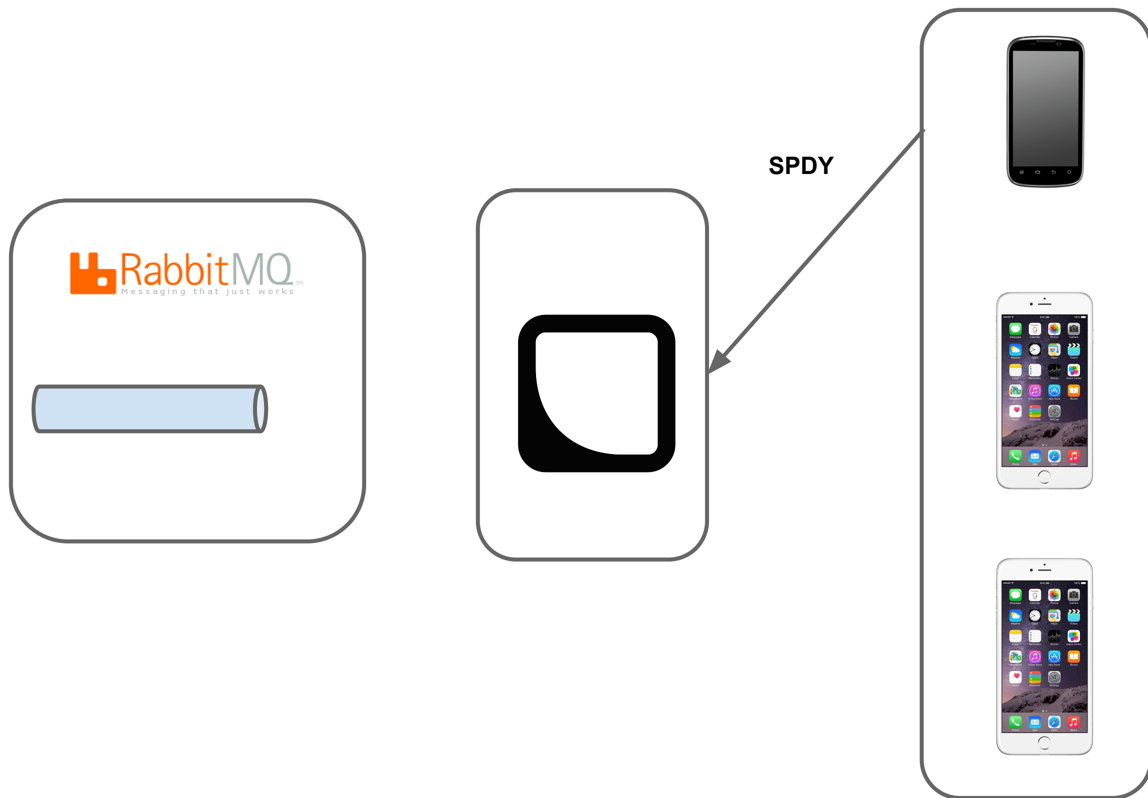
```erlang
declare_queue(Channel, Queue, Args) ->
    Declare = #'queue.declare'{queue=Queue, arguments=proplist_to_amqp_headers(Args)},
    #'queue.declare_ok'{message_count=MessageCount,
                        consumer_count=ConsumerCount} = amqp_channel:call(Channel, Declare),
    {MessageCount, ConsumerCount}.


consume_queue(Channel, Queue, Args) ->
    NoAck = proplists:get_value(no_ack, Args, false),
    Sub = #'basic.consume'{queue=Queue,
                           no_ack=NoAck,
                           arguments=proplist_to_amqp_headers(proplists:delete(no_ack, Args))},
    #'basic.consume_ok'{consumer_tag=ConsumerTag} = amqp_channel:call(Channel, Sub),
    ConsumerTag.
```

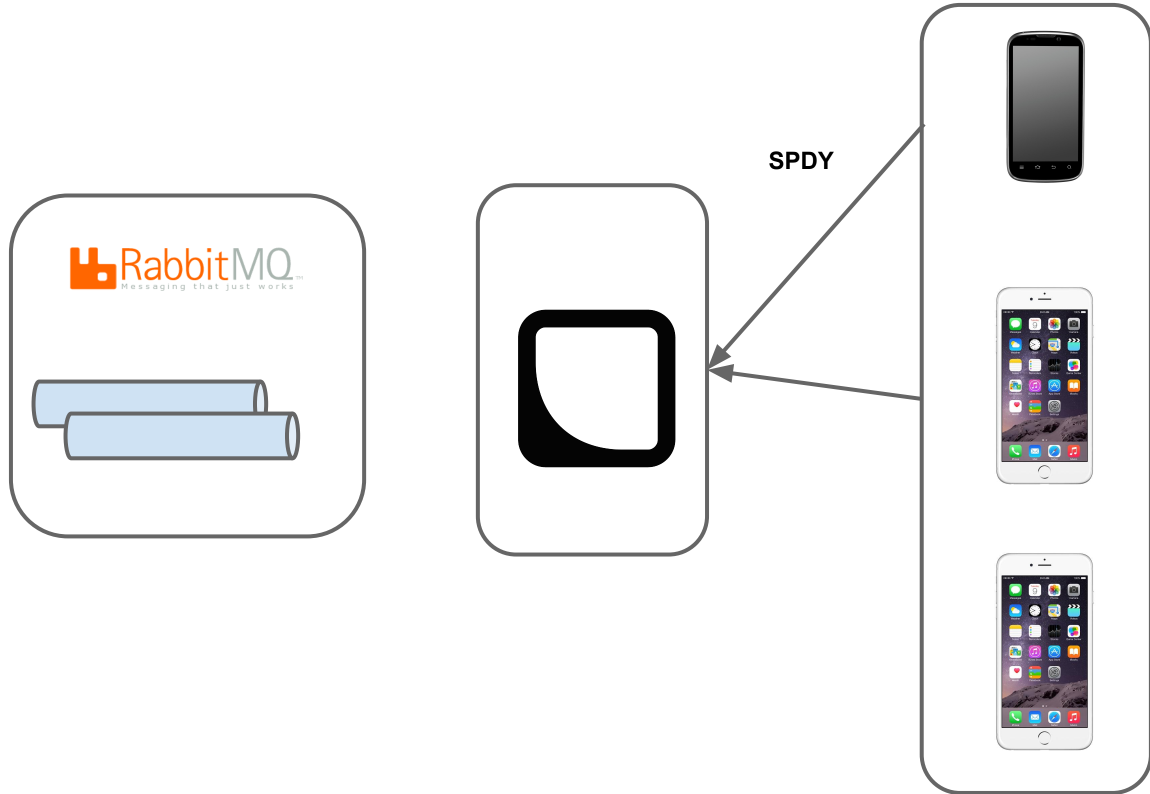
RabbitMQ



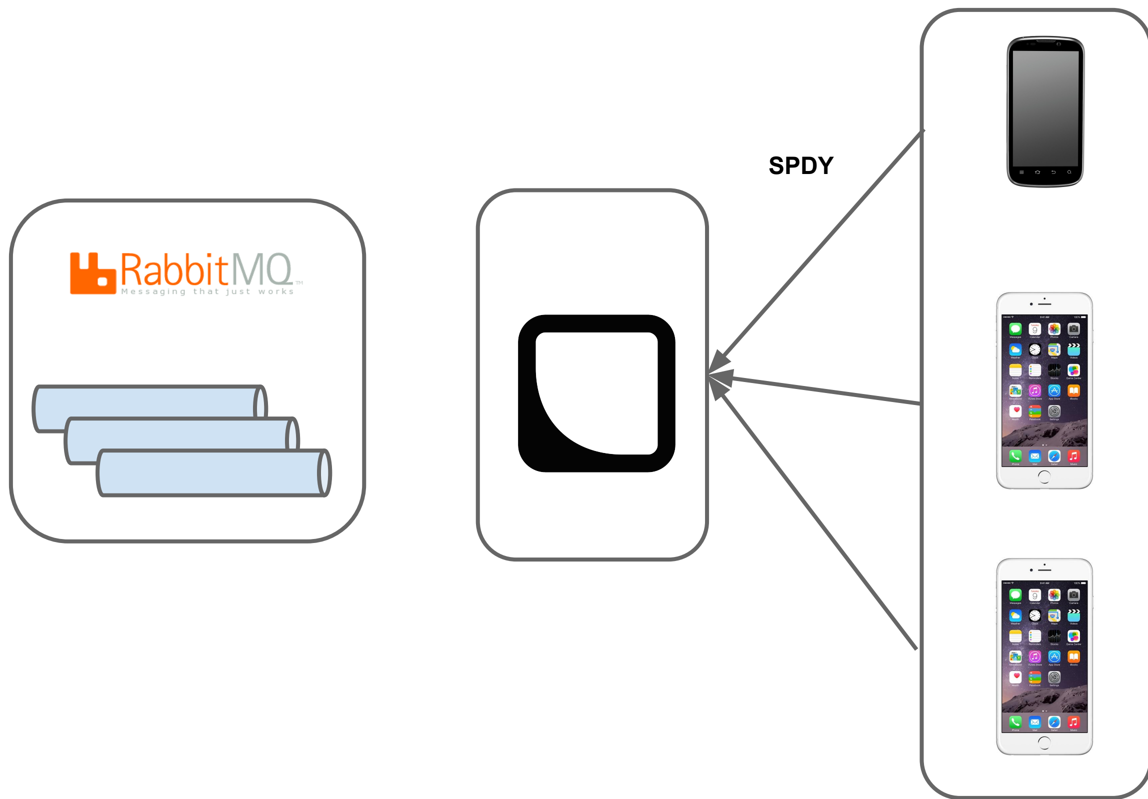
RabbitMQ



RabbitMQ



RabbitMQ



There is a process per connected device that consumes messages from that device’s queue.

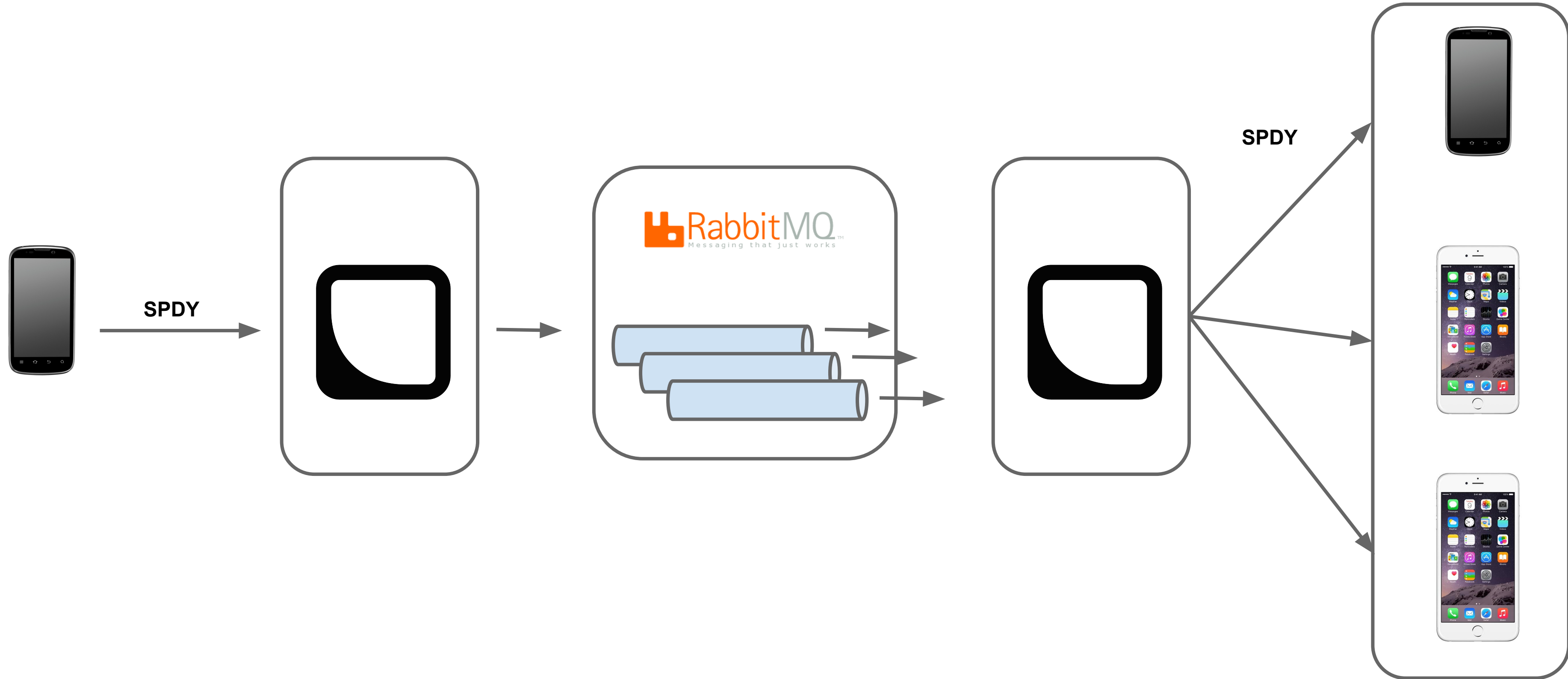
```erlang
% Got a message
handle_info({#'basic.deliver'{delivery_tag=DeliveryTag, consumer_tag=ConsumerTag},
             #amqp_msg{props=#'P_basic'{headers=Headers}, payload=Body}},
            #state{module=Module, substate=SubState}=State) ->
    HeadersProplist = ecu_rabbit:amqp_headers_to_proplist(Headers),
    {ok, NewSubState} = case erlang:function_exported(Module, handle_dequeue, 5) of
        true -> Module:handle_dequeue(DeliveryTag, ConsumerTag, HeadersProplist, Body, SubState);
        false -> Module:handle_dequeue(DeliveryTag, HeadersProplist, Body, SubState)
    end,
    {noreply, State#state{substate=NewSubState}};
```

When that happens,
messages are delivered through
SPDY server push.

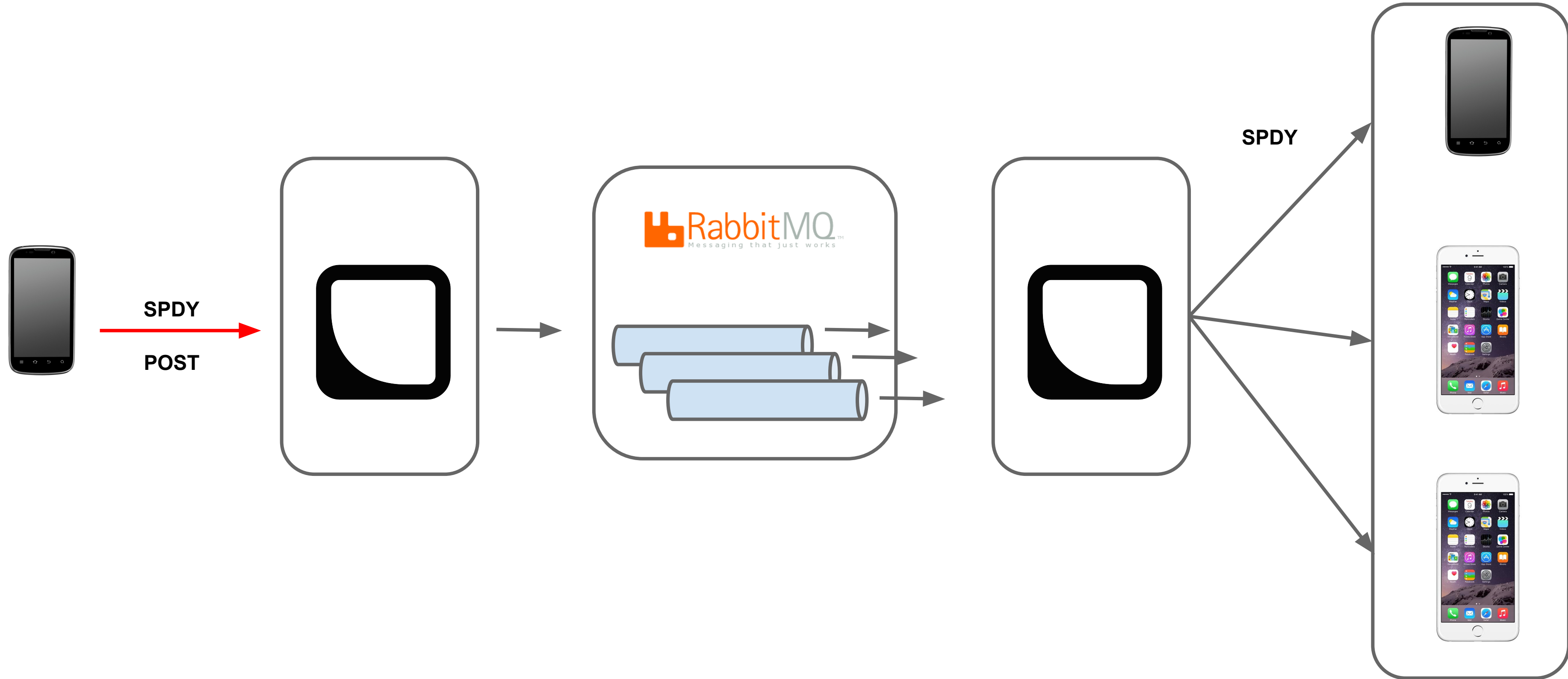
```erlang
spdy_push(AppId, Req, Headers, Body, #state{version_codec=VersionCodec, format_codec=FormatCodec}) ->
    Path = proplists:get_value(path, Headers),
    Reason = proplists:get_value(reason, Headers),
    ContentType = proplists:get_value('content-type', Headers),
    PushBody = case tmc_request:get_codecs_from_header(ContentType) of
        {VersionCodec, FormatCodec} -> Body; % already in the right format
        {OtherVersionCodec, OtherFormatCodec} ->
            {Event, TransientMetadata} = tmc_request:decode(OtherVersionCodec,
                                                            OtherFormatCodec,
                                                            AppId,
                                                            post_event_response,
                                                            Body),
            iolist_to_binary(tmc_request:encode(VersionCodec,
                                                FormatCodec, AppId,
                                                get_event,
                                                {ok, Event, false, TransientMetadata}))
    end,
    Alert = proplists:get_value(alert, Headers),
    lager:info("Sending transport push for ~s event at ~s with alert ~p.",
               [Reason, Path, Alert]),
    exopose:incr([tmc, push, status, 200]),
    cowboy_req:push_reply(200, Path, [{<<"layer-ack">>, <<"Ack">>}], PushBody, Req).
```

Example: let’s assume a 4-device conversation...

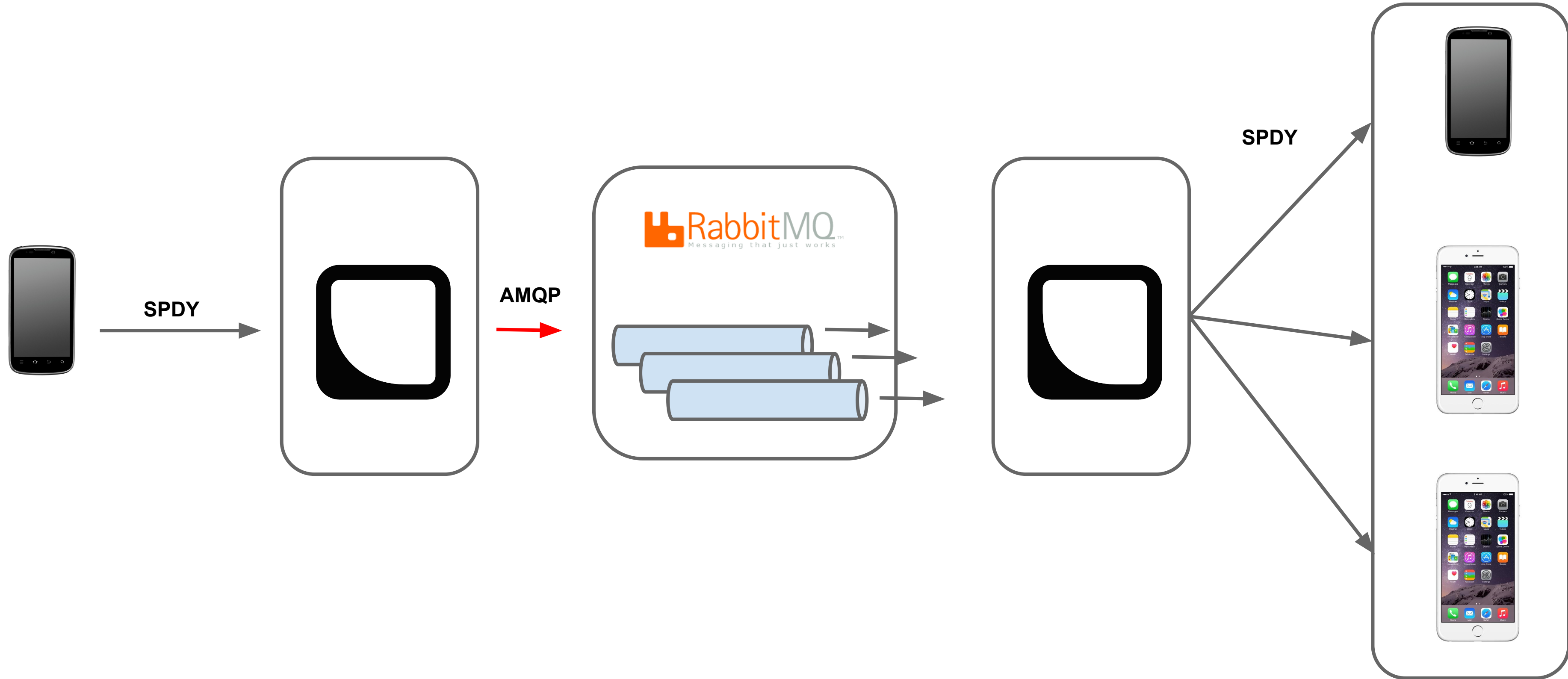
RabbitMQ – Transport pushes



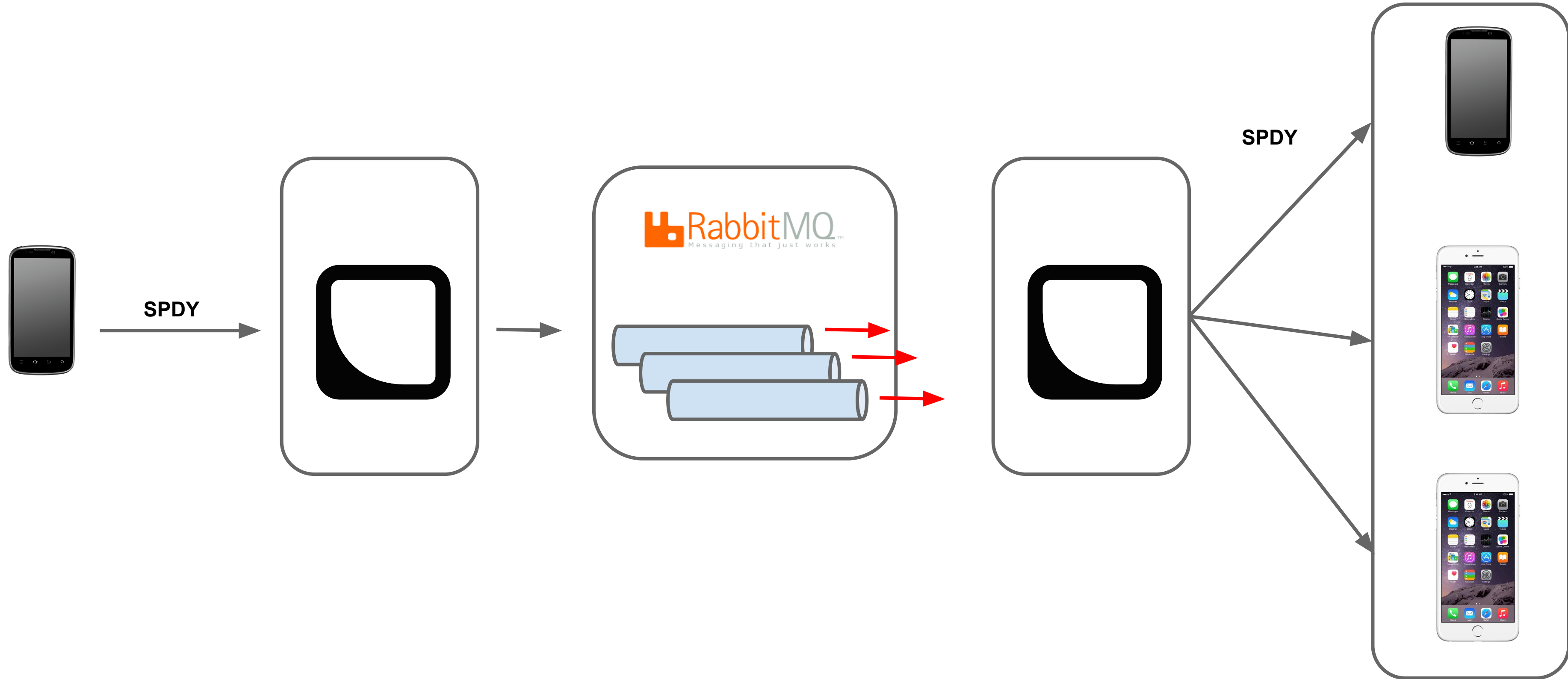
RabbitMQ – Transport pushes



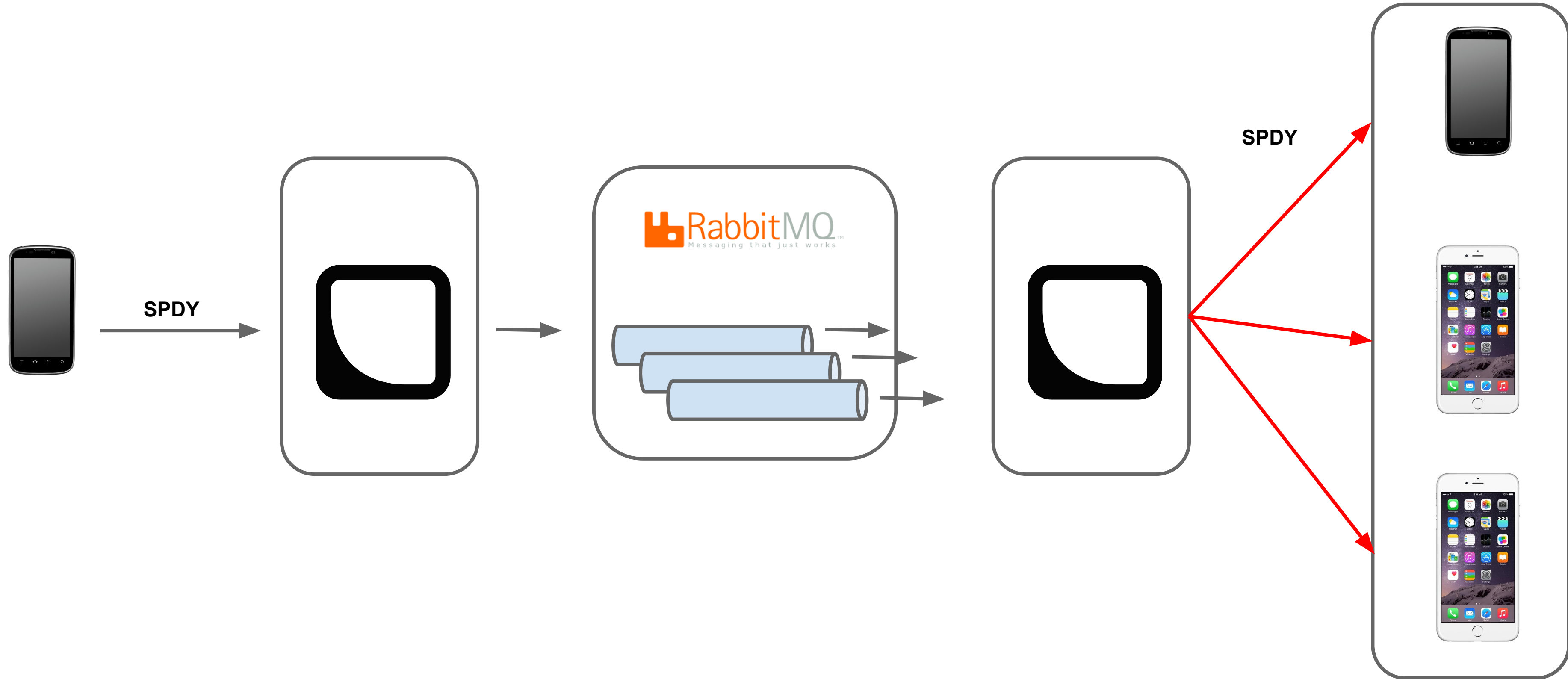
RabbitMQ – Transport pushes



RabbitMQ – Transport pushes



RabbitMQ – Transport pushes

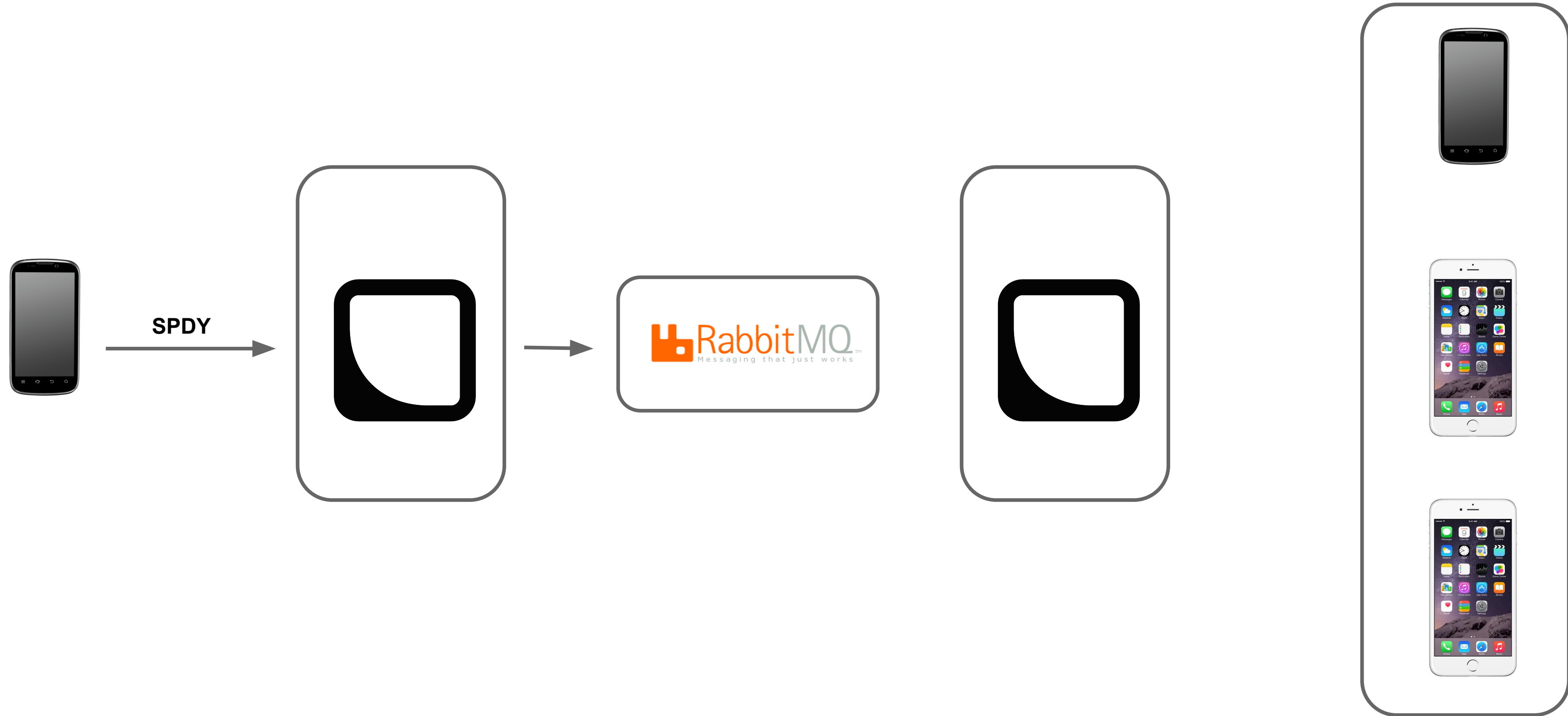


What if target devices

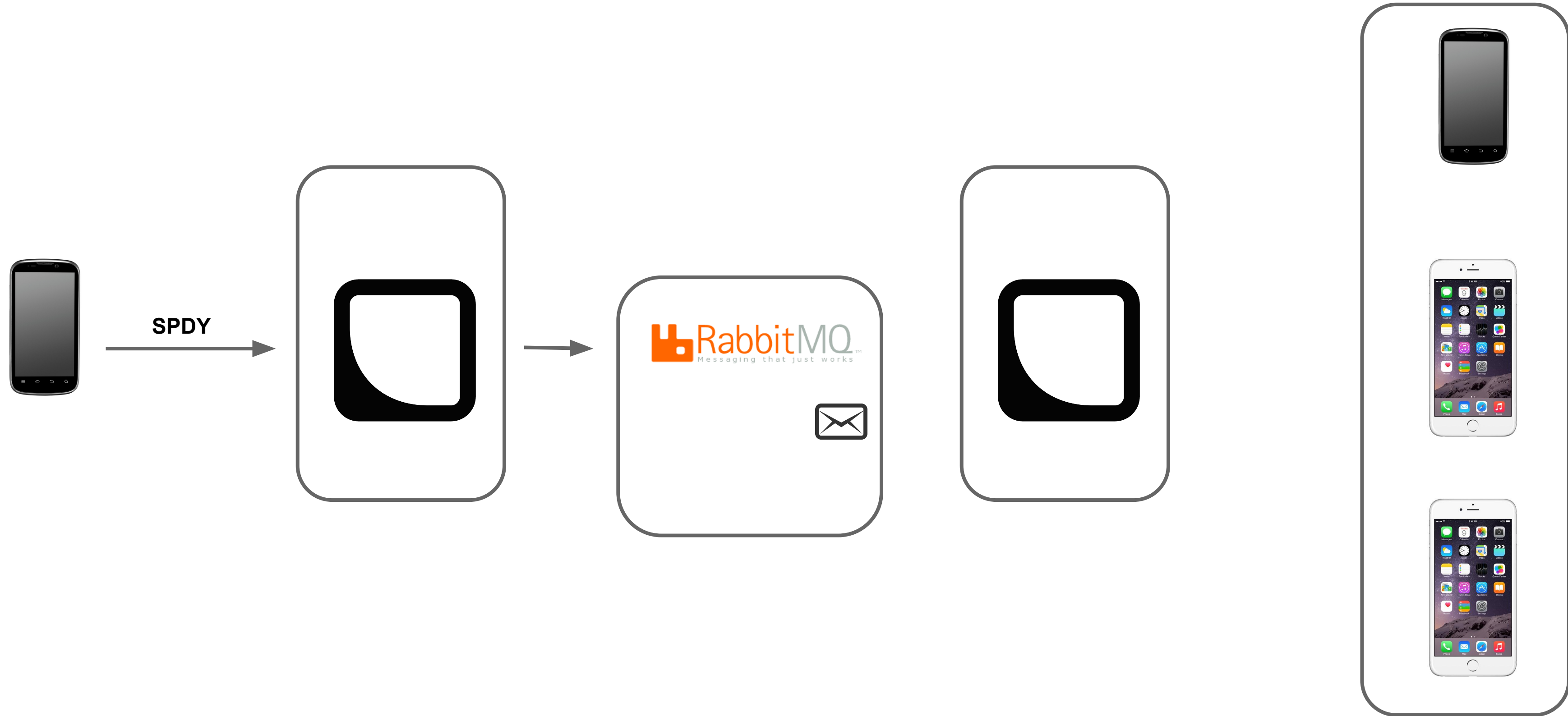
are **not** connected?

There are no queues
for these devices...

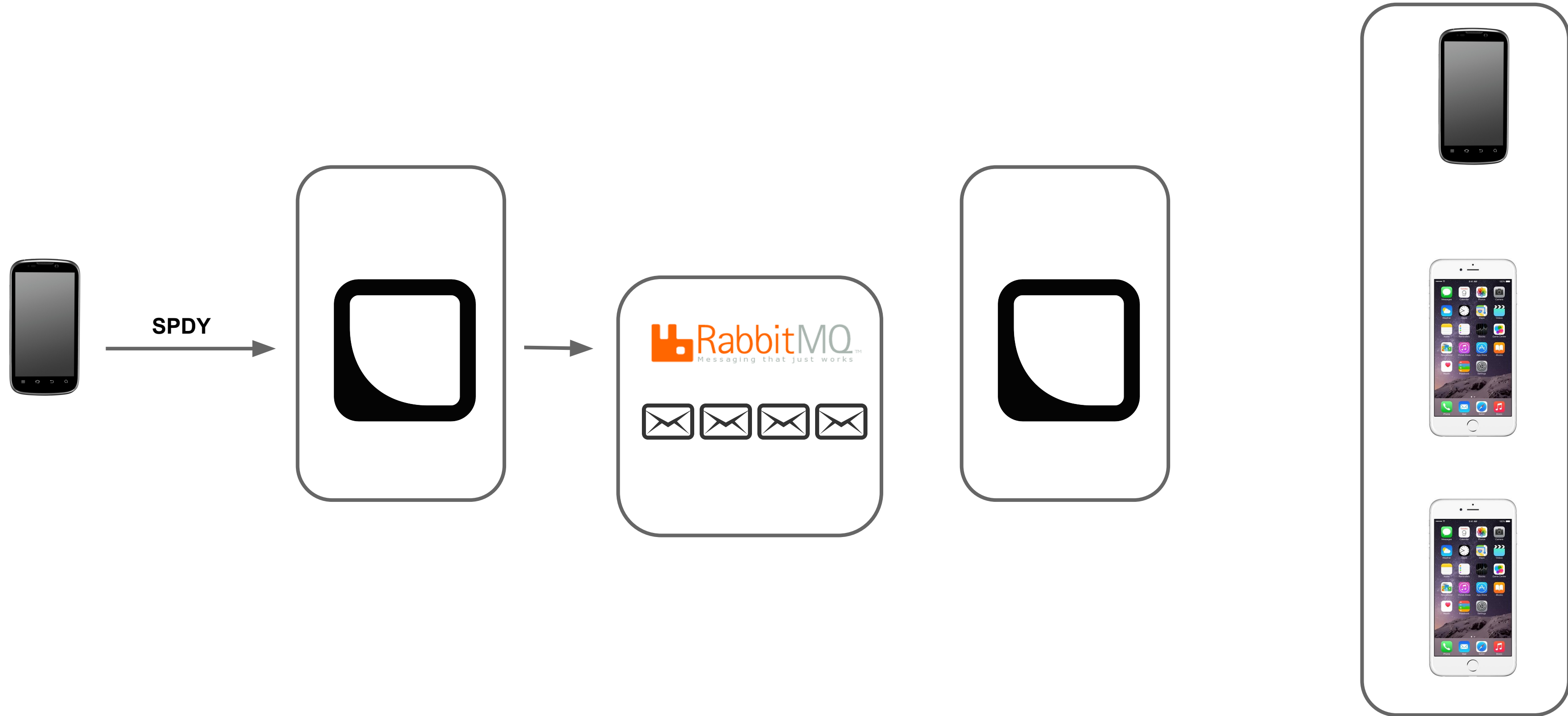
RabbitMQ



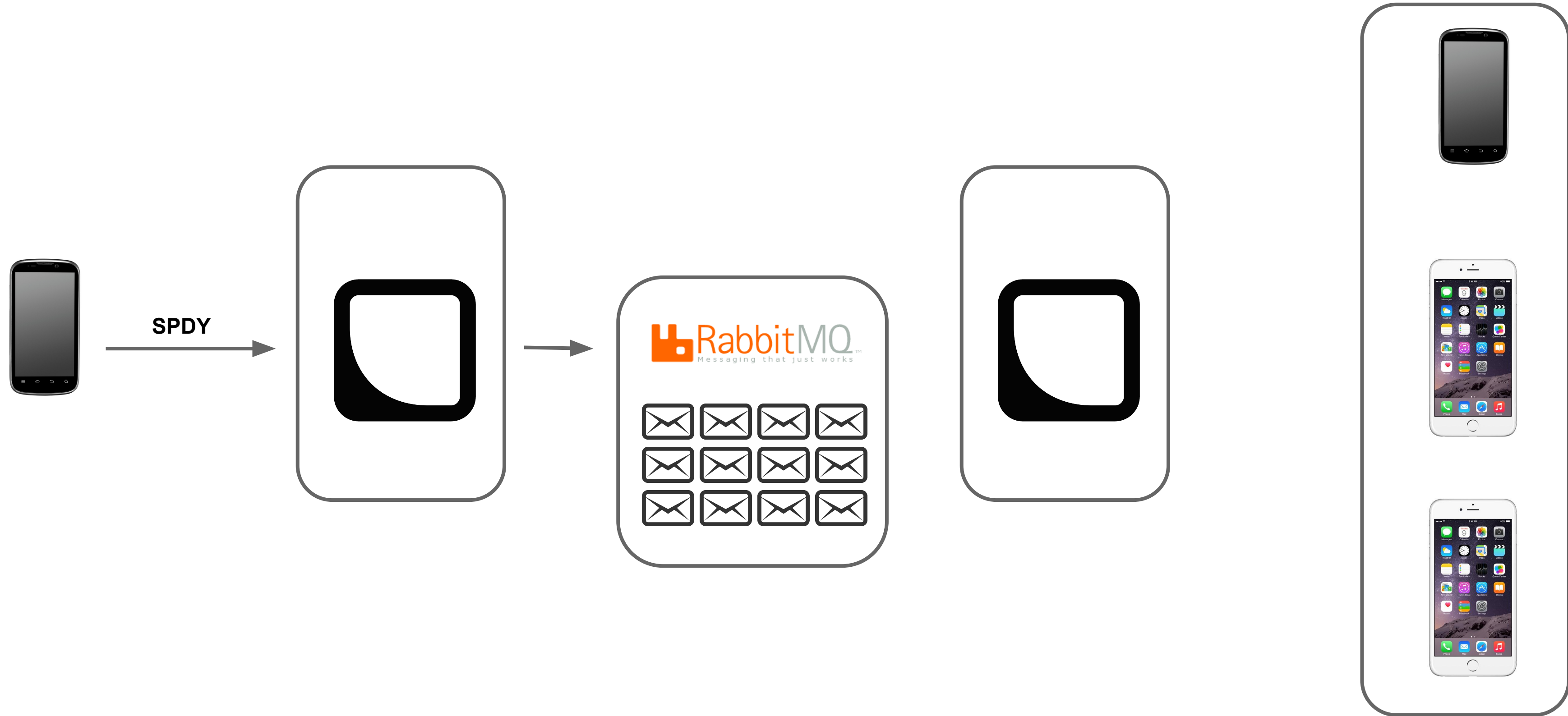
RabbitMQ



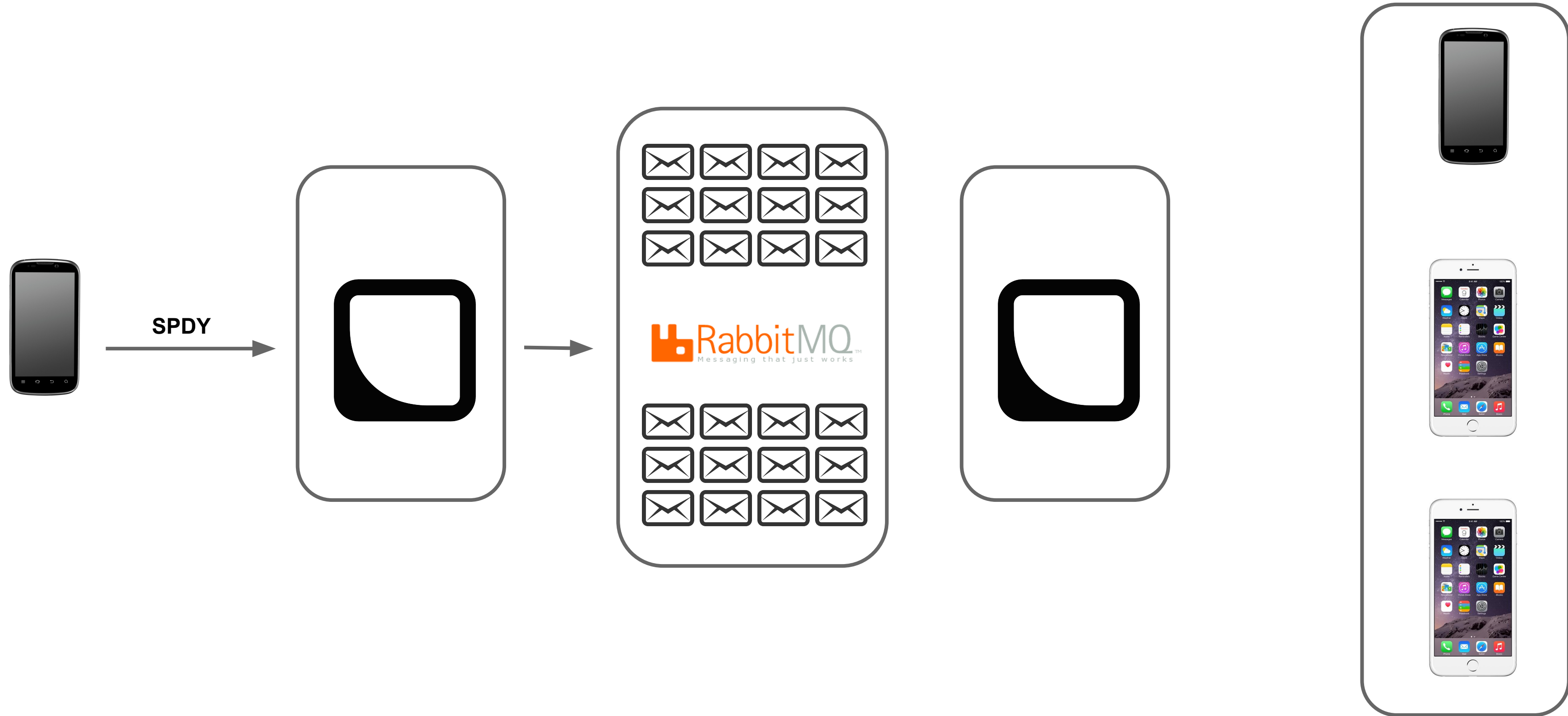
RabbitMQ



RabbitMQ



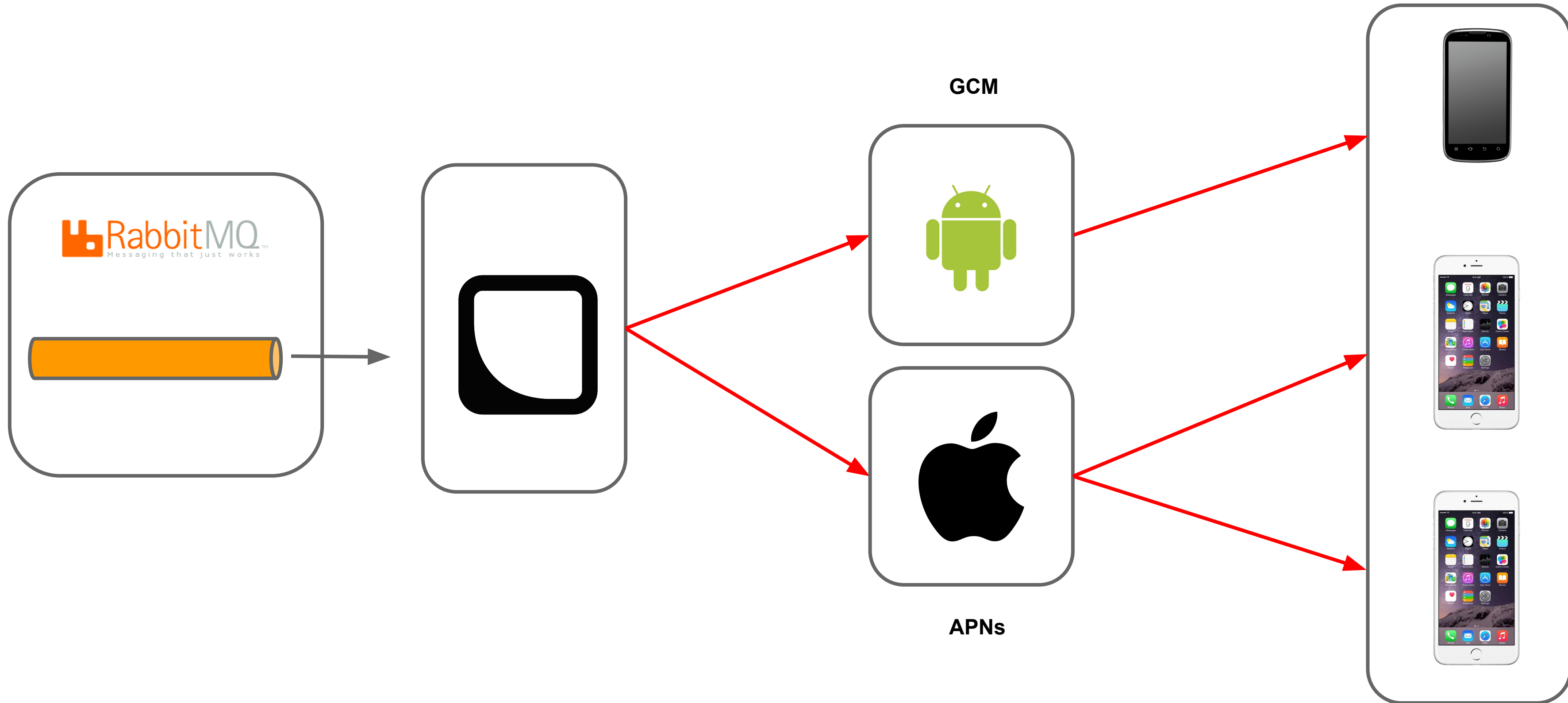
RabbitMQ



...but they need to be notified regardless.

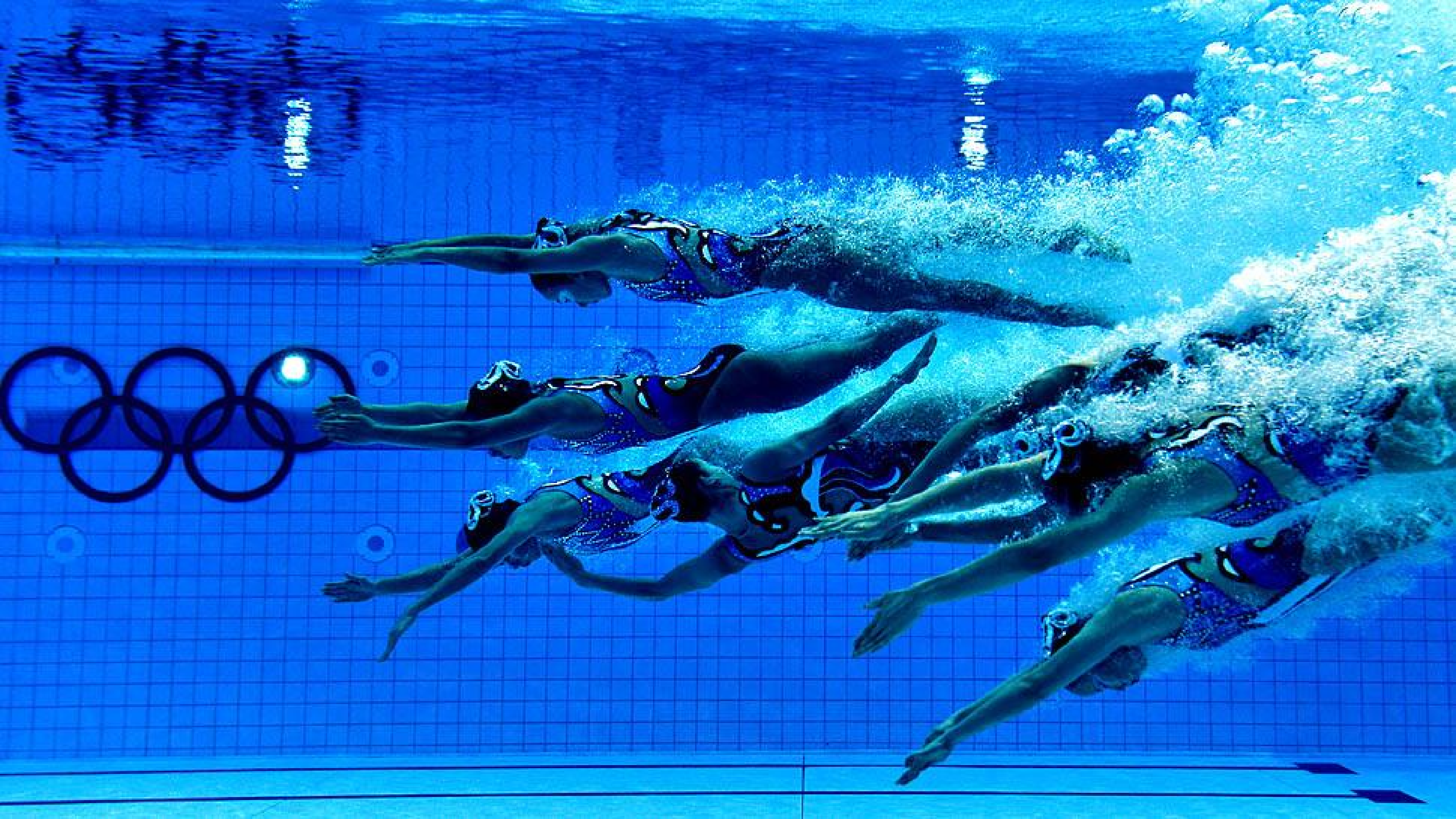
Devices get **platform pushes**, for which there’s a match-all binding queue.

RabbitMQ – Platform pushes



That’s pretty cool, but how do we manage connectivity resources?

5
©Ken Kaminesky

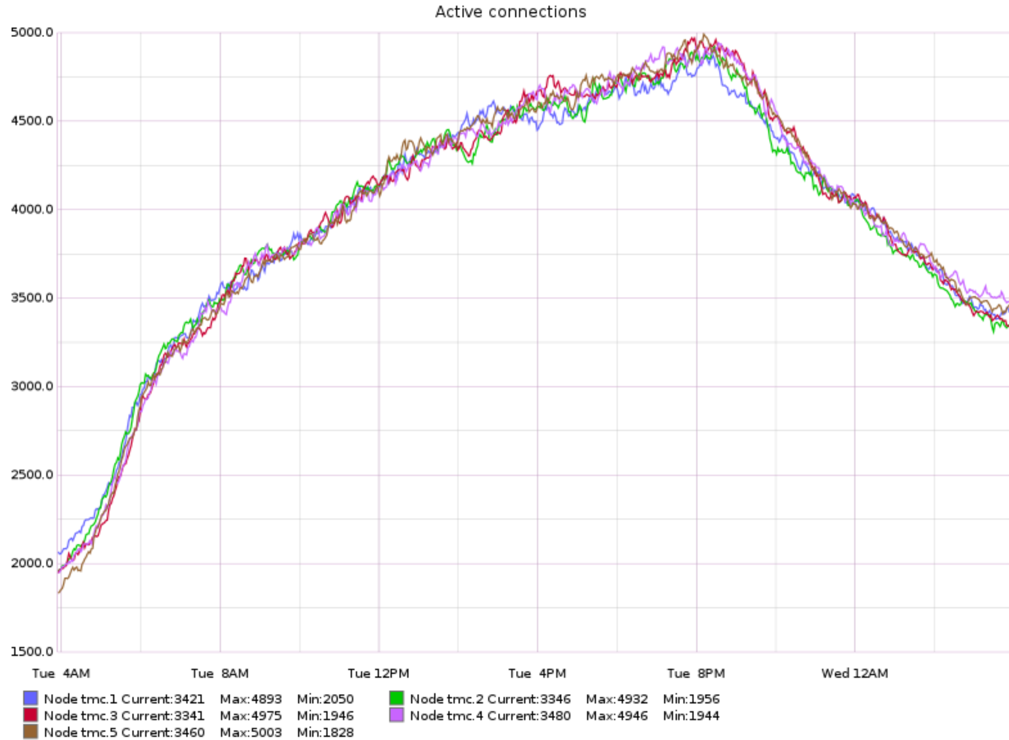


SPDY connections

- `ranch_server.erl` does the job (ninenines/ranch)
- `gen_server` that manages connections, listeners, and ports

```
> ranch_server:count_connections/1
```

SPDY connections



How about AMQP connectivity?

AMQP Connection & Channel Pooling

- 2 types of resources
  - Connections (actual TCP connections)
  - Channels (Lightweight connection)
- 2-layer pooling system
  - `conn 1`
    - `chann 1`
    - `...`
    - `chann N`

```erlang
-spec add_node({string(), inet:port_number()}) -> ok | {error, term()}.
add_node({Host, Port}) ->
    App = gen_server:call(?MODULE, get_app),
    Count = application:get_env(App, ecu_rabbit_connection_count, 2),
    %% Culling connections will kill their channels, so don't.
    pooler:new_pool([{name, connection_pool_name(Host, Port)},
                     {max_count, Count},
                     {init_count, Count},
                     {cull_interval, {0, min}},
                     {start_mfa, {?MODULE, start_connection, [Host, Port]}}]).
```

```erlang
start_connection(Host, Port) ->
    Params = #amqp_params_network{host=Host, port=Port},
    case amqp_connection:start(Params) of
        {ok, ConnectionPid} = Result ->
            %% Tell the gen_server to start a channel pool for the connection.
            ?MODULE ! {connection_started, ConnectionPid},
            %% Monitor the connection so that, when it dies, the gen_server can
            %% remove the channel pool started here.
            monitor(process, ConnectionPid),
            Result;
        Error ->
            Error
    end.

handle_info({connection_started, ConnectionPid}, #state{app=App}=State) ->
    %% Culling channels will kill their connection, so don't.
    pooler:new_pool([{name, channel_pool_name(ConnectionPid)},
                     {group, ?GROUP},
                     {max_count, application:get_env(App, ecu_rabbit_channel_max_count, 65535)},
                     {init_count, application:get_env(App, ecu_rabbit_channel_init_count, 10)},
                     {cull_interval, {0, min}},
                     {start_mfa, {amqp_connection, open_channel, [ConnectionPid]}}]),
    {noreply, State};
```

Data Infrastructure

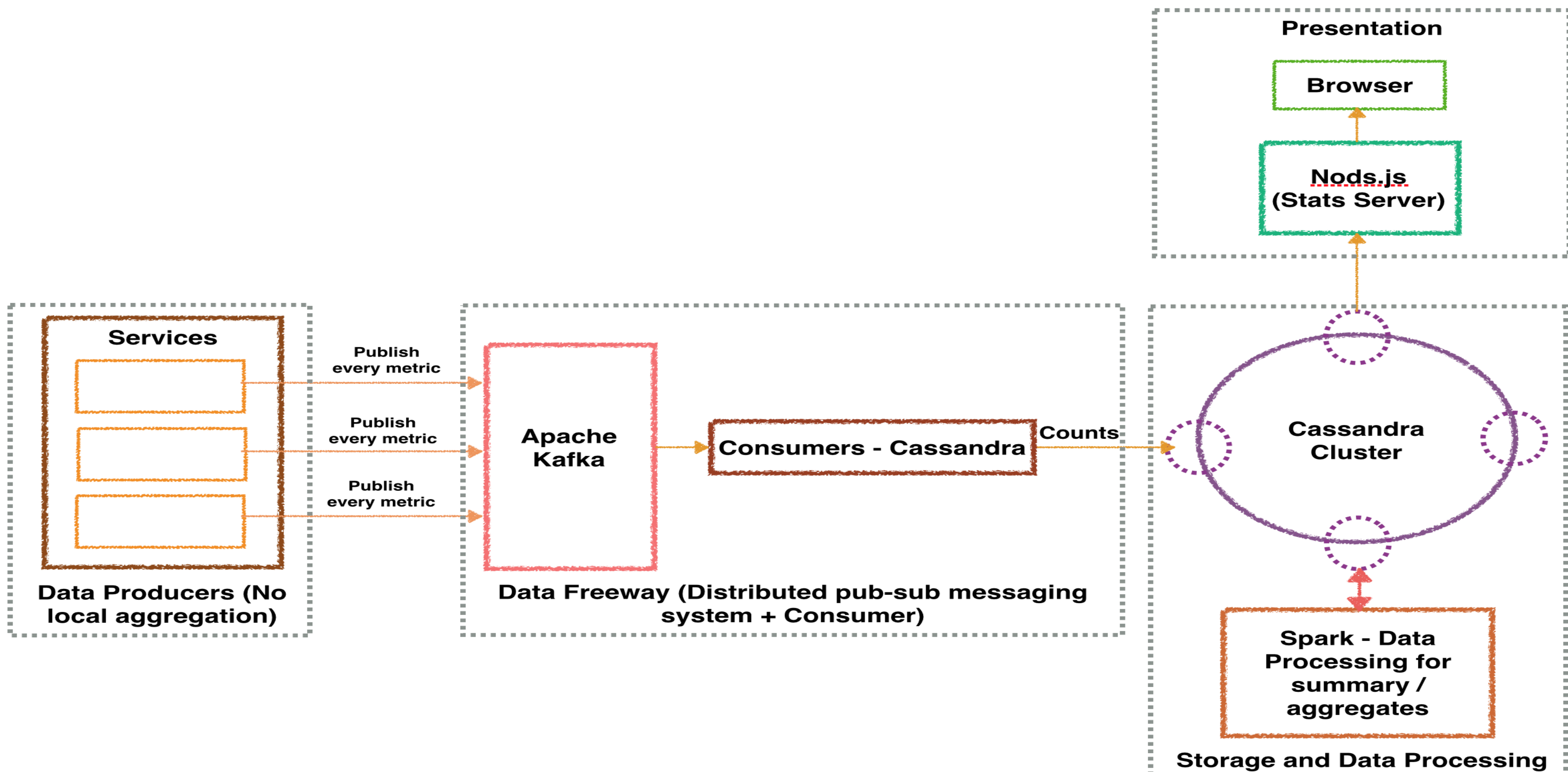
Data Infrastructure

- Apache Cassandra
- Apache Kafka
- ekaf (erlang client for Kafka)
- Apache ZooKeeper
- Apache Spark
- Consumers
- Spark Streaming and SQL

Apache Cassandra

- Online Datastore
  - Apps management
  - Access control
  - Session management
  - Messages and events store
  - Metrics datastore

Data Infrastructure



ekaf, an Erlang client for Kafka

- github.com/helpshift/ekaf
- High-performance
- Producer of events and metrics
- Asynchronous and Batching
- Stream messaging metrics
  - `message_sent, message_delivered, message_read`
  - `active_user, auth_error, push_notifications`
  - `sdk_version, app_metric, user_metric`

Apache Kafka

- Distributed, partitioned, replicated commit-log (Pub-sub)
- Stores messaging events and metrics
- Data freeway
- Multiple topics (multiple partitions per topic)
- Replication Factor = 3
- Application logs (in the works)

Apache Spark

- Lightning fast engine for large-scale data processing
- Aggregation Jobs (MAU, DAU, SDK/Device segregation, Summary, Pipeline for Data service)
- Spark Streaming: Near real-time event processing for analytics to serve customer dashboards
- Spark SQL: Ad-hoc queries for business team
- Tools: Data validation and migration (in the works)

Performance

Performance

http://erlang.org/pipermail/erlang-questions/2015-January/082758.html

[erlang-questions] Garbage Collection, BEAM memory and Erlang memory

Roberto Ostinelli <roberto@widetag.com>
Thu Jan 22 17:33:57 CET 2015

- Previous message: [erlang-questions] cowboy_router weird availability error
- Next message: [erlang-questions] Garbage Collection, BEAM memory and Erlang memory
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Dear List,
I'm having some troubles in pinpointing why a node is crashing due to
memory issues.
For info, when it crashes, it does not produce a crash dump. However I've
monitored live and I've seen the .beam process eat up all memory until it
abruptly exits.

The system is a big router that relays data coming from TCP connections,
into other TCP connections.
I'm using cowboy as the HTTP server that initiates the long-lived TCP
connections.

I've done all the obvious:

   - Checked the States of my gen_servers and processes.
   - Checked my processes mailboxes (the ones with the longest queue have 1
   item in the inbox).
   - My ETS table memory is constant (see below).

I put the system under controlled load, and I can see with
length(processes()). that my process count is stable, always around 120,000.

I check the processes that are using most memory with this call:
```

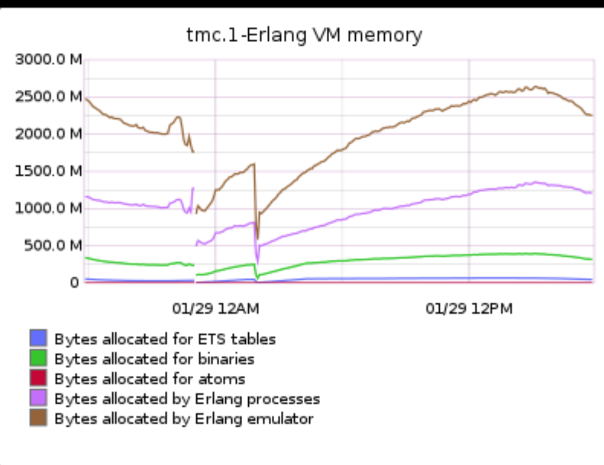
Challenges

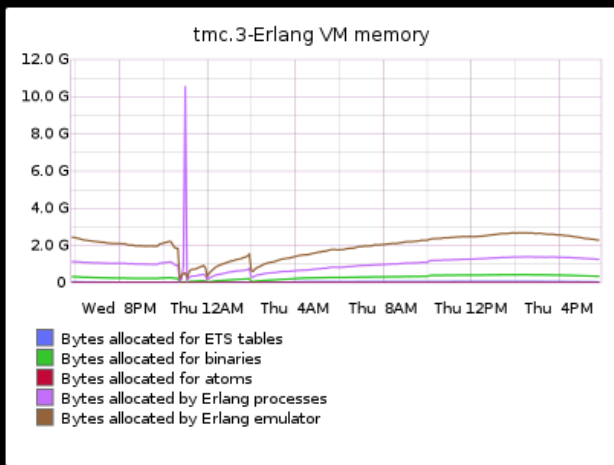
- Memory issues
- Cassandra driver
- Erlang VM shutting down with ‘`reached_max_restart_instensity`’
- `rebar` and reproducible builds, `rebar3`?
- Tradeoffs when forking popular Erlang repos

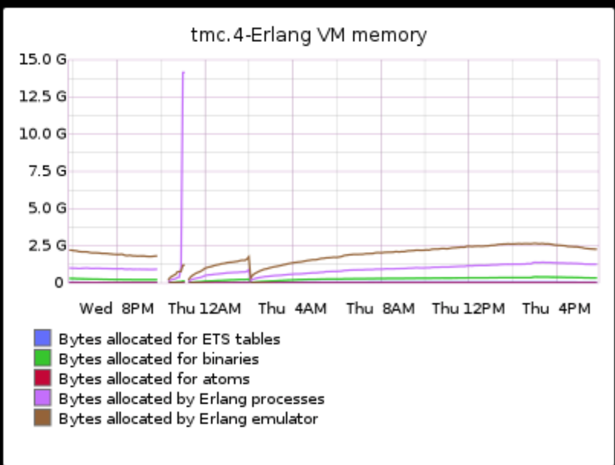
Lessons learned

- Avoid shutdown/max restart frequency copy behavior
  - `supervisor2.erl` by `rabbitmq-server`
- Erlang GC optimizations – Abnormal heap growth
- Monitor pretty much everything!
  - Count HTTP codes, memory, CPU, ...
  - Measure HTTP latencies (`GET, POST, PATCH, ...`)
- Cassandra provides an eventually consistency latency of up to 10ms
  - Requires writing with explicit timestamp (behavior determined by clocks otherwise)
- RabbitMQ & Pooler unexpected behaviors (culling feature)
- `lager - {error_logger_hwm, undefined}`

Lessons learned – Abnormal heap growth

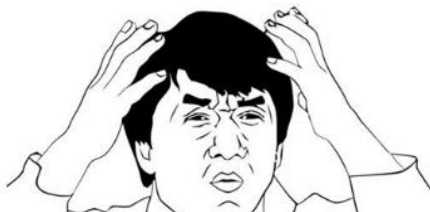




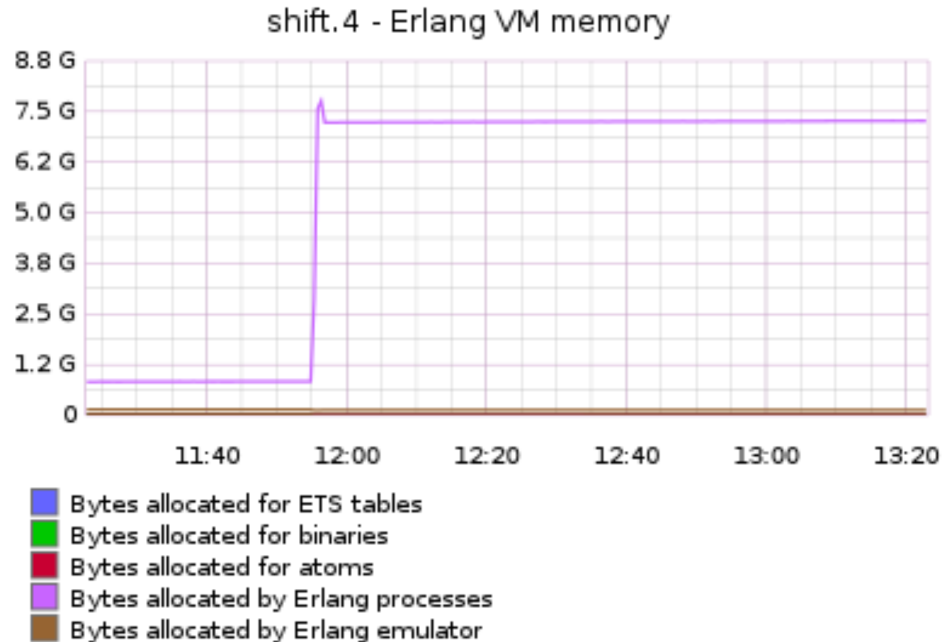


Lessons learned – Abnormal heap growth

- Facts:
  - Huge `error_logger` heap



Lessons learned – Abnormal heap growth



Lessons learned – Abnormal heap growth

- Facts:
  - Huge `error_logger` heap
  - Shrinks to nothing if GC kicks

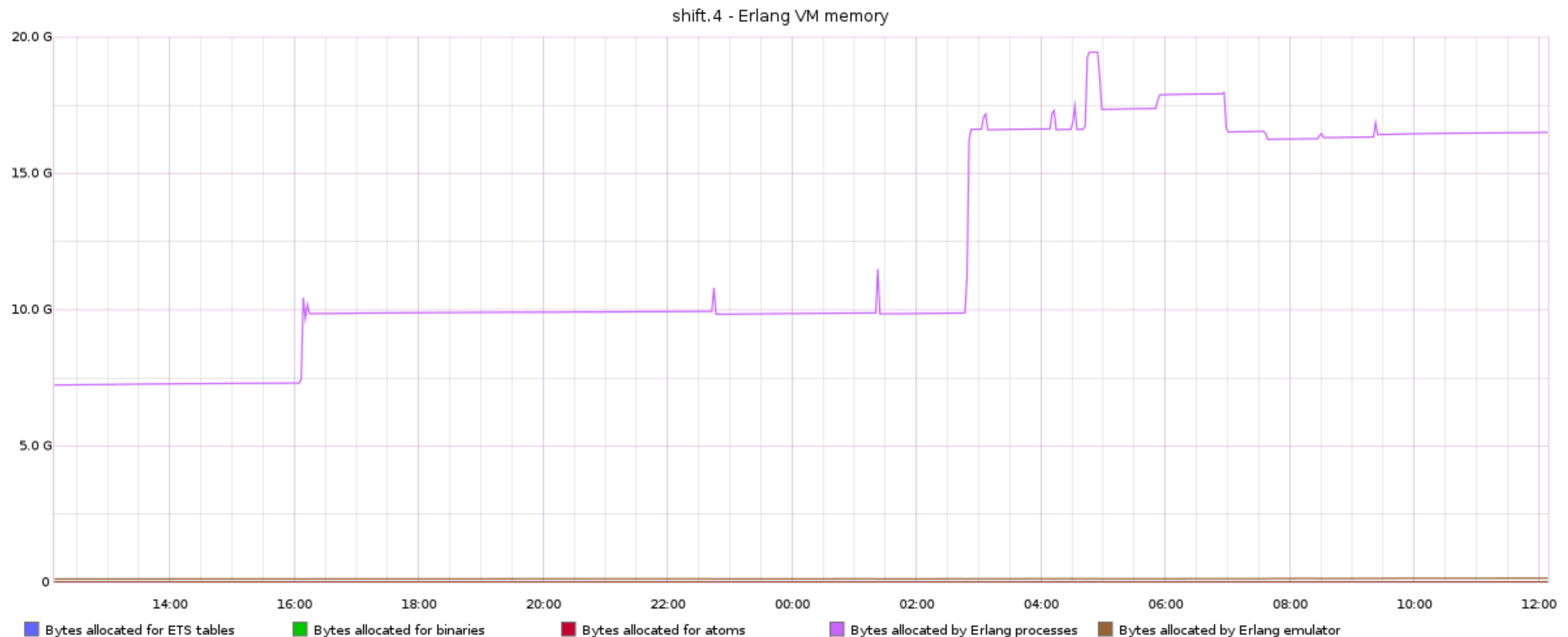
Lessons learned – Abnormal heap growth

- Hypothesis:
  - Large burst of messages to the process, e.g: 100 000
  - If process kicks GC when those messages are in queue, it will make a copy to its heap (This is done as an optimization)
  - The (possibly) bad side effect is that the young heap will have to be grown to fit all messages
  - GC will be triggered which releases all messages as they have been logged leaving the heap very large
  - The heap shrinking algorithm will not kick until a 2nd GC
  - Since the heap is very large, it will take a long time for that second GC to kick in

Lessons learned – Abnormal heap growth

- Outcomes:
  - You end up with an abnormally large heap for `error_logger`.

Lessons learned – Abnormal heap growth



Lessons learned – Abnormal heap growth

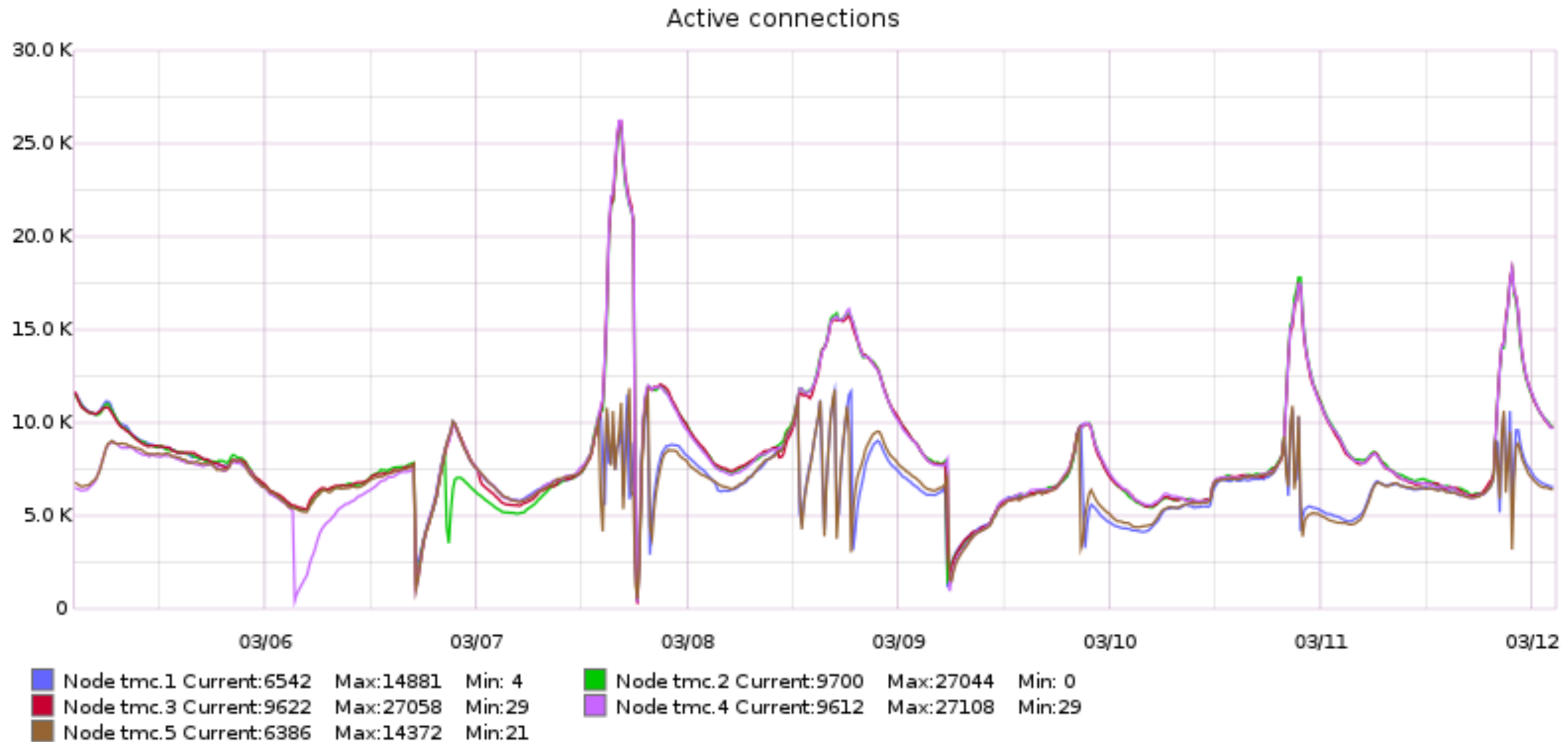
- Solutions:
  - Monitor the process that you know can have this problem
  - Trigger a GC on them when needed

```
> erlang:system_monitor({large_heap, integer() >=0})
```

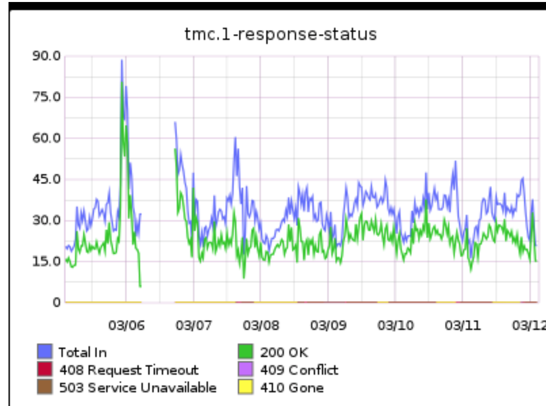
Reference: Exploring Garbage Collection implementation in Erlang (Lukas Larsson, Erlang Solutions).

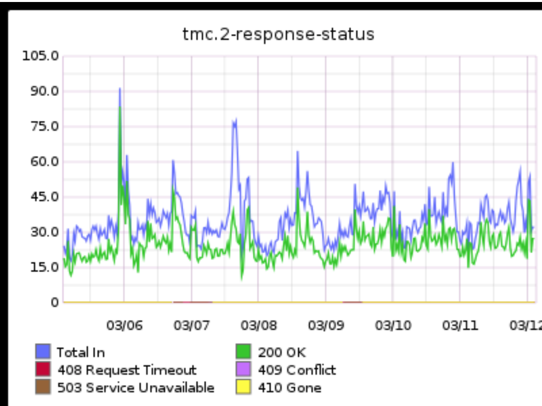
Lessons learned – Abnormal heap growth

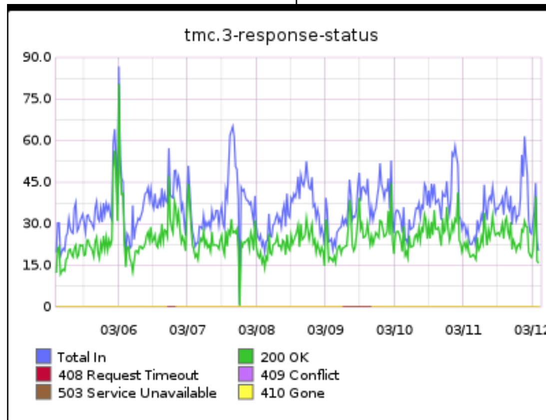
Lessons learned – Monitor your system!

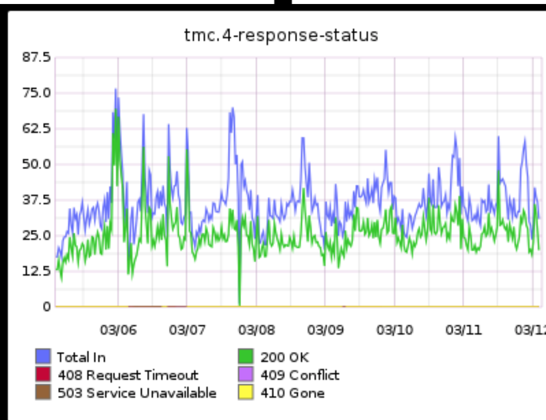


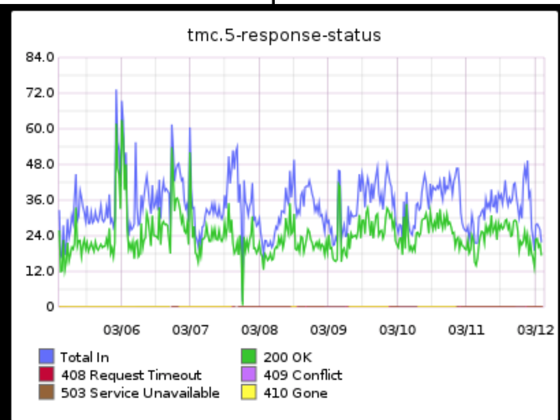
Lessons learned – Monitor your system!



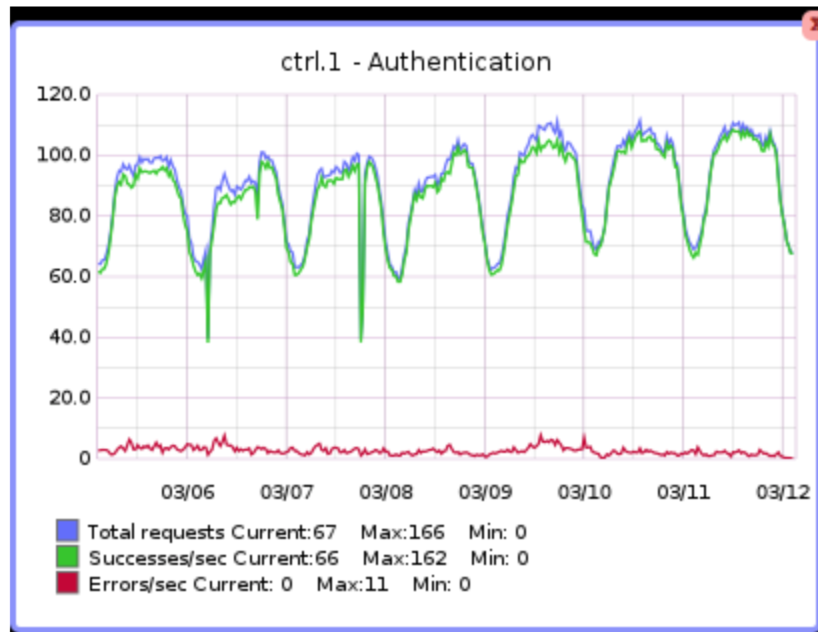


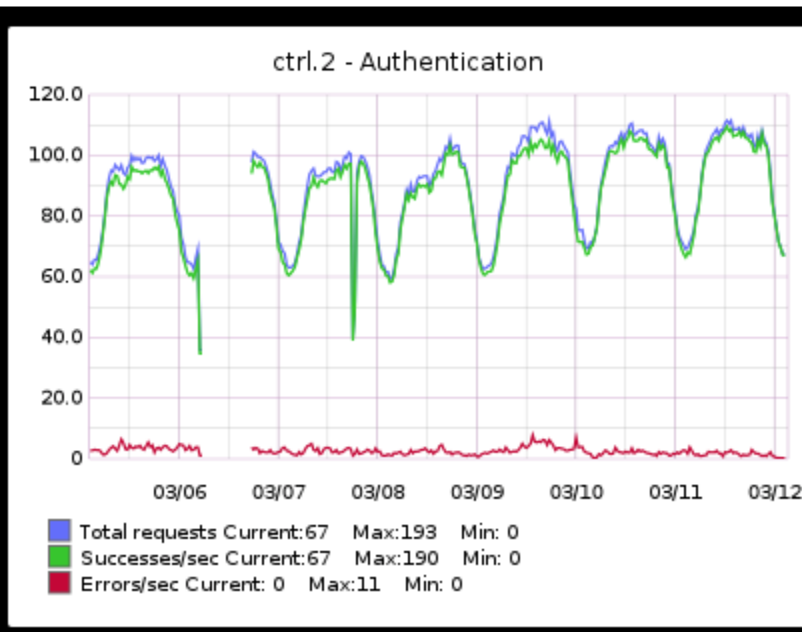




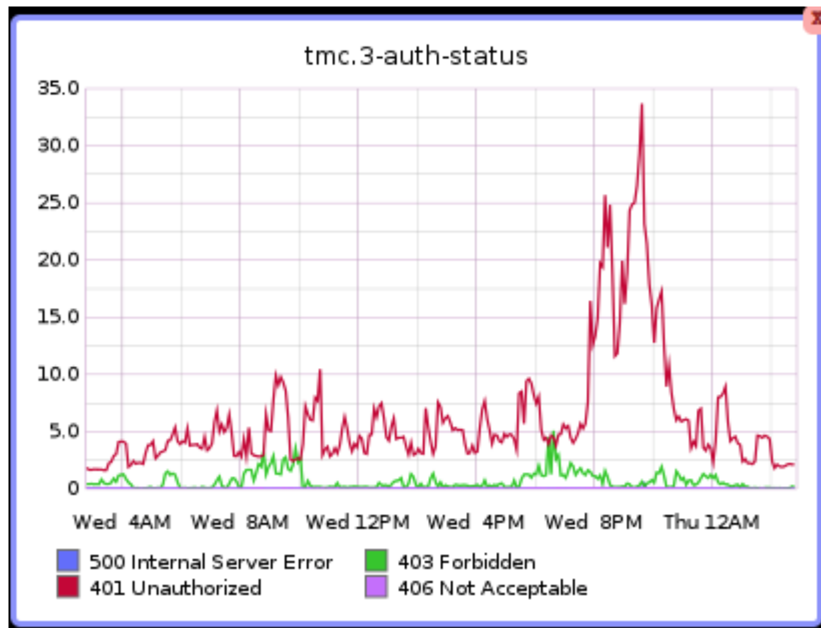


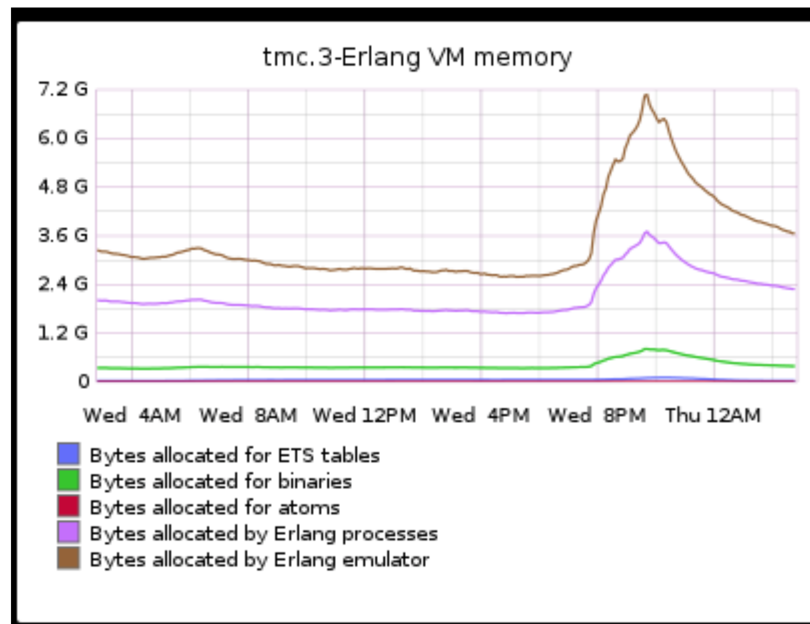
Lessons learned – Monitor your system!





Lessons learned – Monitor your system!





Open source contributions

Merged to upstream:

- github.com/ferd/backoff – Jitter based exponential backoff
- github.com/seth/pooler – Anticipatory growth
- github.com/layerhq/cqerl – Several bug fixes

Layer forks:

- github.com/layerhq/cowboy – SPDY server push, flow control, rate limiting
- github.com/layerhq/gcm_ccs:layer-non-singleton – Protocol upgrades
- github.com/layerhq/apns4erl – Protocol upgrades
- github.com/layerhq/thrift-erl – General improvements
- ...and many more! https://github.com/layerhq

Questions?

www.layer.com