From delayed polling to real-time telematics with RabbitMQ and Erlang

Natalya Arbit and Brett Cameron
March 2014

Abstract

Lucid Logistics (http://www.lucidlogistics.com/, a subsidiary of Quake Global, http://quakeglobal.com/) provides end-to-end solutions for global GPS monitoring, management, and tracking of remote assets via the Iridium Low Earth Orbit (LEO) satellite network. One of the biggest challenges faced by Lucid Logistics has been distributing data immediately to customers as it arrives from the satellite provider. Historically data has been stored in a backend database and made available to customers in a somewhat generic fashion via user interfaces, email notifications, and reports. However, more customers now want to receive raw data feeds for their assets in near-real-time that they can process and act upon using their own specific set of business rules in order to best optimize their particular unique business operations. Exposing the data for customers to consume via a web service was not a viable solution as this would require customers to poll for new data on some schedule, which would not only make inefficient use of resources but would also fail to address the real-time data requirement. Instead, extending the existing solution to better meet the demand for near-real-time data feeds required some type of publish/subscribe mechanism that allowed customers to subscribe to data feeds and would push data to subscribers in real-time as it was received from the satellite provider. Additionally, the solution needed to utilize protocols that could be readily accommodated by all customers. These requirements led Lucid Logistics to PubSubHubBub (http://code.google.com/p/pubsubhubbub/), an open webhook-based protocol for distributed publish/subscribe communication over HTTP(s), initially designed to extend the Atom and RSS protocols for data feeds. This in turn led Lucid Logistics to RabbitHub (https://github.com/tonyg/rabbithub), a powerful Erlang-based implementation of the PubSubHubBub protocol that operates as a RabbitMQ plugin. Not only was RabbitHub ideally suited to Lucid Logistics' needs in terms of providing the necessary protocol support, but because it is implemented as a RabbitMQ plugin, the resultant solution is also able to leverage and benefit from a myriad of RabbitMQ and Erlang features, including built-in security mechanisms, SSL support, message and message queue persistence, flexible routing topologies via exchanges, scalability and performance, and fault-tolerance through clustering.

In this talk the speakers will discuss how RabbitHub, RabbitMQ, and Erlang have been used to enhance the Lucid Logistics solution to provide customers with data in near-real-time. The PubSubHubBub protocol will be discussed and the operation of the RabbitHub plugin will be examined in the context of how it leverages various Erlang/OTP features and the RabbitMQ plugin architecture to provide a simple yet powerful HTTP-based interface to RabbitMQ's messaging infrastructure. The solution affords considerable flexibility and provides numerous opportunities for additional development of the Lucid Logistics message processing ecosystem, including providing data to customers via other protocols supported by other RabbitMQ plugins (MQTT, AMQP, STOMP, Web Sockets), and closer integration with the old database-based solution. These and other ideas will be briefly discussed.

About us - Natalya

Natalya Arbit is a senior developer with close to 15 years of professional experience spanning a variety of industries. She has spent the last 6 years working in telematics with a focus on supporting SaaS multi-tenant web portals using the Microsoft stack of technologies. She holds a Master of Science degree in Geology from University of Minnesota, Duluth, and a Bachelor of Science degree in Economics from the Russian State Geological Prospecting University.

About us - Brett

Brett Cameron currently works as a senior software architect with HP's corporate Cloud Services group, focusing on the design and implementation of message queuing and related integration services for customers and for internal use. Brett lives in Christchurch, New Zealand and has worked in the software industry for over 20 years. In that time he has gained experience in a wide range of technologies, many of which have long since been retired to the software scrapheap of dubious ideas. Brett has been involved in the research and development of low-latency and highly scalable messaging solutions for the Financial Services sector running on HP platforms and as a consequence of this work he has participated in several interesting Open Source projects involving Erlang and Erlang-based software products such as RabbitMQ. He is responsible (or should that be irresponsible) for porting various Open Source solutions (including Erlang, RabbitMQ, and most of Riak) to HP's legacy OpenVMS operating system. Brett holds a doctorate in chemical physics from the University of Canterbury, and still maintains close links with the University, delivering guest lectures and acting as an advisor to the Computer Science and Electronic and Computer Engineering departments on course structure and content. In his spare time Brett enjoys listening to music, playing the guitar, and drinking beer.

AGENDA

- **Introduction**
- The challenge
- The solution
- Current status and potential future developments
- Conclusions/summary
- Questions

Introduction/background

The challenges and issues faced by the solution provider:

- Distributing data immediately as it arrives from the satellite provider
- Avoiding polling mechanisms!
- Providing lightweight message format
- Spooling data if end user is temporarily unable to receive it
- Providing secure transfer of data
- Guaranteeing channel privacy in a multi-tenant environment

Solution:

- RabbitMQ and RabbitHub… powered by Erlang

Lucid Logistics

- Lucid Logistics (http://www.lucidlogistics.com/) provides end-to-end solutions for global GPS monitoring, management, and tracking of remote assets via the Iridium Low Earth Orbit (LEO) satellite network
  - Worldwide customer base
  - Multiple industries
    - Oil and Gas, Mining and Heavy Equipment, Logistics, SCADA and Fixed Assets, Government and NGOs
  - End-to-end solutions (hardware and software)
  - Developed the first secure GPS all-satellite tracking system with guaranteed data delivery (May 2013) and over-the-air configuration and firmware updates
- Customers use GPS monitoring to manage vehicle fleets and marine vessels as well as non-mobile, fixed assets like compressors and generators
- Lucid Logistics software solutions provide customers with various telematics data
  - Asset location
  - Emergency notifications
  - Vehicle driver behavior (excessive speeding, idling, …)
  - Asset health data
    - Provided by J1939/J1708 integration to read various sensor data

Satellite telemetry/telematics

- Telematics is essentially any integrated use of telecommunications and informatics:
  - Sending, receiving, and storing information via telecommunication devices in conjunction with affecting control on remote objects
  - The integrated use of telecommunications and informatics, for application in vehicles and with control of vehicles on the move
  - Includes but is not limited to GPS technology integrated with computers and mobile communications technology in automotive navigation systems

Satellite telemetry/telematics

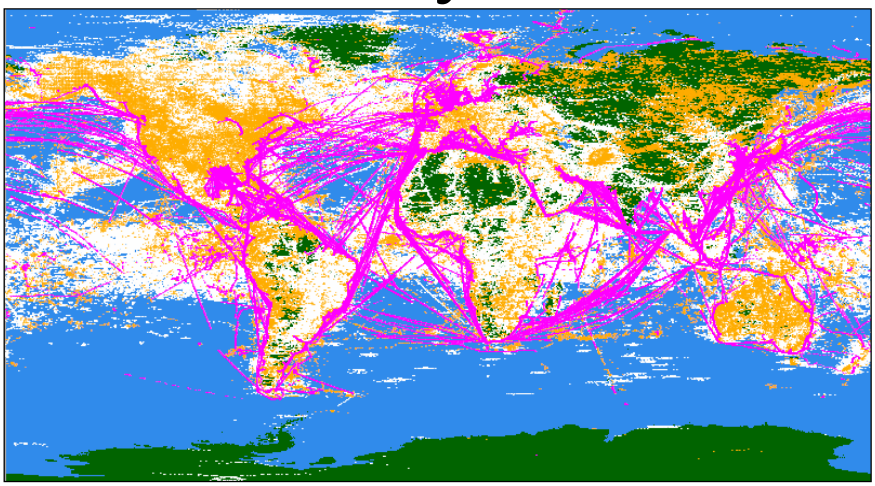
Specifics of Satellite Communication on the Iridium Network:

- Constellation of 66 active satellites used for worldwide voice and data communication
- Unique mesh design - covers the entire Earth, including poles, oceans and airways; no dead spots
- Extremely low (within 30 seconds) message latencies
- Expensive air time

Solution provider strategies:

- Utilize Short Burst Data (SBD) service - sending and receiving short data bursts, less than 2 kilobytes at a time
- Utilize internally developed algorithms to produce highly compacted messages. This approach allows Lucid Logistics to provide extremely competitive, cost-effective solutions.

Satellite telemetry/telematics



Note: * One week plot of M2M/SBD session, voice call and high-speed data origination points for the week of 7/10/11 to 7/16/11 (commercial traffic only)

AGENDA

- Introduction
- **The challenge**
- The solution
- Current status and potential future developments
- Conclusions/summary
- Questions

The challenge

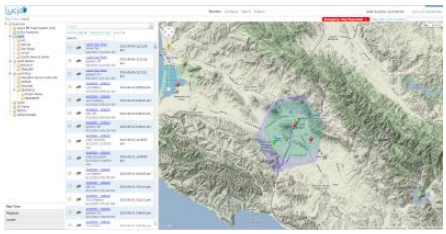
Message stream:

- Consists of compact 10 to 20-byte long messages
- Each message is unpacked according to internally developed proprietary protocol
- Light load with messages arriving at configured interval anywhere between 1 minute and 24 hours depending on end use scenario
- Load increases as customer base requiring direct data feed grows
- Individual unpacked messages become heavier with growing number of monitored items making traditional reporting slower
- Can't put more details without running into risk of Natalya being fired!

The challenge

- Historically data has been stored by Lucid Logistics in a backend database and made available to customers in a somewhat generic fashion
  - Web-based user interface
  - Email notifications
  - On-demand and scheduled reports
  - This model is recognized as the industry standard in asset tracking and management

- More and more customers are now wanting to receive raw data feeds in near-real-time
  - Allows them to process and act on data using their specific business rules
  - Facilitates better optimization of business operations
  - Allows them to utilize an existing infrastructure and mix feeds from multiple solution providers



AGENDA

- Introduction
- The challenge
- **The solution**
- Current status and potential future developments
- Conclusions/summary
- Questions

If at first you don't succeed...

- Initial SOAP-based solution
- The problem with polling
- Pub/sub approach
- PSHB, RabbitHub, RabbitMQ, and Erlang

15

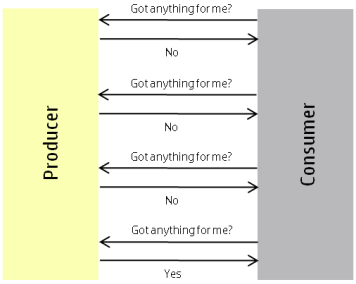
SOAP-based solution

- An initial approach aimed at providing data to customers more effectively was to provide a SOAP-based web service
- The interface was exposed to customers with the assumption that they would poll for updates on a defined schedule
- The results of this approach were not pretty
  - Resources wasted on both sides
  - Strict policies had to be implemented and enforced to prevent excessive resource drain
  - Customers were not happy because the feed was nowhere near real-time

16

The problem with polling

- Polling wastes resources
- It is pretty much impossible to schedule polling correctly, even with fancy stuff like adaptive rate control
  - Polling frequently wastes client and server resources
  - Polling infrequently is nice to servers, but clients may get unacceptably delayed information
  - If the frequency of updates is known and reasonably constant an optimal polling frequency can usually be determined
    - But this is rarely the situation
    - And clients need to abide by the rules
- ...but polling *is* easy to implement!



There are pros and cons...

- A disadvantage of the push model is that the pushing process potentially needs to handle consumers becoming unavailable
  - Polling handles the case where either side is temporarily down
    - If the server is down, the consumer keeps asking
    - If the consumer is down, the server patiently waits
    - But just how far will this scale?
    - What if consumers need to receive updates in near-realtime (a relative term I suppose)?

As will be discussed later, courtesy of RabbitMQ, the final solution readily addresses these failure scenarios.

- Dealing with acknowledgements is an issue either way and generally comes down to good client/consumer design

The number of Internet-connected devices that currently poll for updates is terrifying!

There are pros and cons…

- But any which way you look at it polling wastes resources
  - Wasted CPU cycles
  - Wasted network bandwidth
  - Wasted energy
  - Wasted time
  - … unhappy customers

Clearly another method of enhancing the Lucid Logistics system was required…

A better approach

- Extending the present solution to better meet the demand for near-real time data feeds required some type of pub/sub mechanism
  - No polling!
- A webhook-based solution was considered a simple and effective approach to integration
  - HTTP(S) is ubiquitous
  - Firewall-friendly
    - Well, mostly… but more on that topic later
  - …
- *Enter PubSubHubbub (or PSHB, or PuSH)…*

PubSubHubbub

PubSubHubbub

- See http://code.google.com/p/pubsubhubbub/
- Devised by Brad Fitzpatrick (http://bradfitz.com/) and Brett Slatkin (http://www.crunchbase.com/person/brett-slatkin)

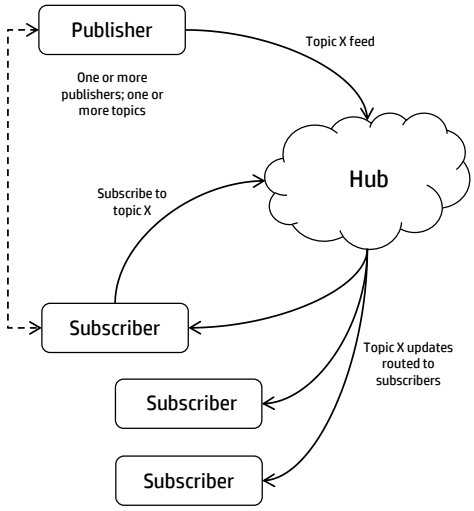
- An open web-hook-based protocol for distributed publish/subscribe communication over HTTP(S)
  - Initially designed to extend the Atom (and RSS) protocols for data feeds
  - Extended to support any data type that can be exchanged via HTTP

- Subscribers can get near-instant notifications (via webhook callbacks) when a topic to which they are registered is updated
  - Pushed HTTP notifications without requiring subscribers to expend resources on polling for changes
  - Improves upon the typical situation where a client periodically polls the feed server at some (possibly arbitrary) interval

PubSubHubbub

Key components

- Publishers
  - Publish feeds with their content
  - Include hub forwarding information in feeds (optional)
  - POST updates to hubs
- Subscribers
  - Accept the feeds forwarded by hubs
  - POST subscription requests to hubs and confirm them
  - Provide endpoint URLs to which hubs can post updates
- Hubs
  - Receive subscription requests from subscribers and verify them
  - Provide endpoint URLs to which publishers can POST updates
  - POST updates to subscribers
  - May query publishers for updates



The PubSubHubbub protocol

High-level protocol overview

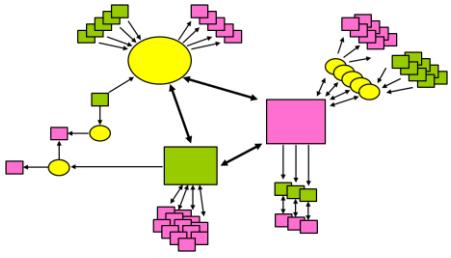
- A resource URL (a "topic") declares its hub server(s) in HTTP headers via links
  - Publishers delegate content distribution to hubs
  - Hubs can be run by the publisher of the resource, or can be a community hub that anyone can use
- A subscriber can:
  - Initially fetch the resource URL as normal from the publisher
    - If the response declares any hubs in its headers, the subscriber can then avoid polling of the publisher URL and can instead register with the designated hub(s) and subscribe to updates
  - Or go straight to the relevant hub (next point)
- The subscriber subscribes to the topic of interest using determined or previously known hub details, supplying the hub with a callback URL
  - Subscription requests are verified via a simple callback challenge/response mechanism
  - Subscriptions may expire after a subscriber-specified time or they can be indefinite
- Publishers post new content to the relevant hub(s)
- The hub efficiently multicasts the new content out to all registered subscribers

23

The PubSubHubbub protocol

Some additional points to note

- Subscribers need to run a web-accessible server so that hubs can directly provide them (via HTTP POST) with new content
  - PSHB is a server-server protocol
  - It won't work too well if the consumer is a PC or workstation sitting behind a firewall or a NAT device
- The subscription process uses a verification of intent mechanism to prevent abusive subscriptions
  - Basically just a simple challenge/response check
- A validation mechanism allows for subscriptions to private or protected web resources
- Subscribers can optionally share a secret token with the hub
  - The token will be used by the hub to compute an HMAC key that will be sent to the subscriber
  - The subscriber can then easily verify the hub identity



24

PubSubHubBub

So…

- The PSHB protocol looked like an excellent solution to Lucid Logistics problem
- Several implementations…
  - Google and Superfeedr offer a public open hubs that anyone can use
    - https://pubsubhubbub.appspot.com/
    - http://pubsubhubbub.superfeedr.com/
  - WordPress has a PSHB implementation handling notifications of blog updates
    - See http://en.blog.wordpress.com/2010/03/03/rub-a-dub-dub-in-the-pubsubhubbub/

- But these could not be used by Lucid Logistics
- Additionally, these implementations are geared towards Atom/RSS feeds and/or require data to be formatted as XML

- *Enter RabbitHub (love it when a plan comes together)…*

RabbitHub

- An Erlang implementation of the PubSubHubbub protocol that operates as a RabbitMQ plugin
  - Initially created in 2009 by Tony Garnock-Jones (one of the original developers of RabbitMQ)
  - See https://github.com/tonyg/rabbithub

- Looking at it very simplistically, RabbitHub implements a simple webhook-based pub/sub mechanism that provides an HTTP-based interface to RabbitMQ
  - Gives every AMQP exchange and queue hosted by the broker two URL's:
    - One to use for delivering messages to the exchange or queue
    - One to use to subscribe to messages forwarded on by the exchange or queue
  - Subscriptions supply a callback URL to RabbitHub that is used to deliver messages via HTTP POST
    - Subscription details stored in Mnesia
  - Provides a generally useful, easy to use, and efficient means of interacting with RabbitMQ in a (sort of) RESTful manner

RabbitHub

- The biggest advantage of RabbitHub over other PSHB implementations is the support for arbitrary message format
- Because RabbitHub is written in Erlang as a plugin to RabbitMQ it also benefits from a myriad of RabbitMQ and Erlang features
  - Relatively low memory footprint
  - Built-in security
  - SSL support (required enhancement to RabbitHub)
  - Message and message queue persistence
    - If a consumer goes away their messages can be queued up or placed in a dead-letter queue
  - Flexible routing topologies via exchanges
  - Administrative tools
  - Fault tolerance and clustering
  - Scalability
  - Performance
  - …

RabbitHub does not implement any of the Atom-specific parts of the PSHB protocol, including the "ping" operation that tells a PSHB hub to re-fetch content feeds; however such functionality is not required by Lucid Logistics.

27

But of course nothing's ever that simple…

- RabbitMQ had evolved considerably since RabbitHub was originally written, and the code needed updating to work with current RabbitMQ versions
  - RabbitHub makes considerable use of RabbitMQ internal APIs, some of which had evolved/changed
  - There were also a few Erlang/OTP and Mochiweb version issues to be dealt with

- Other changes and enhancements that have been made include
  - Use the OTP httpc library to POST messages to subscribers
    - HTTPS/SSL and proxy support
  - Permit specification of message delivery mode in query string when publishing messages (message durability)
  - Replaced boot-step-style plugin start-up code
  - Rebar-based build procedure
    - Much easier than using the RabbitMQ Public Umbrella for plugin development
    - Can easily do plugin builds on Windows using Cygwin
    - Found the above mind-blowingly easy after trying to build officially unsupported version of RabbitMQ Public Umbrella with unsupported version of Erlang/OTP on Ubuntu 12.04 and 13.10 for four days (Natalya)

28

Use the OTP httpc library

```
deliver_via_post(#rabbithub_subscription{callback = Callback},
                 #basic_message{routing_keys = [RoutingKeyBin | _],
                                content = Content0 = #content{payload_fragments_rev = PayloadRev}},
                 ExtraHeaders) ->
    case catch mochiweb_util:urlsplit(Callback) of
        {_Scheme, _NetLoc, _Path, ExistingQuery, _Fragment} ->
           #content{properties = #'P_basic'{content_type = ContentTypeBin}} =
               rabbit_binary_parser:ensure_content_decoded(Content0),
           PayloadBin = list_to_binary(lists:reverse(PayloadRev)),

           C = case ExistingQuery of
                   "" -> "?";
                   _ -> "&"
               end,

           URL = Callback ++ C ++ mochiweb_util:urlencode([{'hub.topic', RoutingKeyBin}]),

           case httpc:request(post, {URL,
                                     [{"Content-length", integer_to_list(size(PayloadBin))},
                                      {"Content-type", case ContentTypeBin of
                                                           undefined -> "application/octet-stream";
                                                           _ -> binary_to_list(ContentTypeBin)
                                                       end},
                                      {"X-AMQP-Routing-Key", binary_to_list(RoutingKeyBin)} | ExtraHeaders],
                                     [], PayloadBin}, [], []) of
               {ok, {{_Version, StatusCode, _StatusText}, _Headers, _Body}} ->
                   if
                       StatusCode >= 200 andalso StatusCode < 300 ->
                           {ok, StatusCode};
                       true ->
                           {error, StatusCode}
                   end;
               {error, Reason} ->
                   {error, Reason}
            end;
        _ ->
            {error, invalid_callback_url}
    end.
```

The original implementation used a nifty but simple hand-rolled HTTP client that did not support SSL or proxy servers.

Use the OTP httpc library

- Using the OTP httpc library meant that HTTPS could be used to POST messages to subscribers and HTTP(S) proxy support could be easily facilitated

```
%% rabbitmq.config
%%
[
    ...
    ...
    {rabbithub, [
        {http_client_options, [
            {proxy,{{"10.1.1.1",8080}, ["localhost"]}},
            {https_proxy,{{"10.1.1.1",8080},["localhost"]}}
        ]}
    ]}
].
```

Example configuration file entry. The same proxy server has been specified for both HTTP and HTTPS, and the proxy server will not be used for requests to localhost.

```
%% rabbithub:start/2
%%
    ...
    HttpOpts = get_env(http_client_options, []),
    ok = httpc:set_options(HttpOpts),
    ...
```

Message delivery mode

- Specify message delivery mode by including `hub.persistmsg=true|false` in query string when POSTing messages to RabbitHub
  - Non-persistent is the default
  - Target queue must be durable if `hub.persistmsg=true`

```
extract_message(ExchangeResource, ParsedQuery, Req) ->
    RoutingKey = param(ParsedQuery, "hub.topic", ""),
    DeliveryMode = get_msg_delivery_mode(param(ParsedQuery, "hub.persistmsg", "0")),
    ContentTypeBin = case Req:get_header_value("content-type") of
                         undefined -> undefined;
                         S -> list_to_binary(S)
                     end,
    Body = Req:recv_body(),
    rabbit_basic:message(ExchangeResource,
                         list_to_binary(RoutingKey),
                         [{'content_type', ContentTypeBin},{'delivery_mode',DeliveryMode}],
                         Body).
```

For example:

```
$ curl -v -d "Hello World!"
http://guest:guest@localhost:15670/endpoint/x/amq.direct?hub.topic=foo&hub.persistmsg=true
```


Plugin startup

Replaced older-style RabbitMQ boot-step code…

```
-rabbit_boot_step({?MODULE,
                   [{description, "RabbitHub"},
                    {mfa, {rabbithub, setup_schema, []}},
                    {mfa, {rabbit_sup, start_child, [rabbithub_sup]}},
                    {mfa, {rabbithub_web, start, []}},
                    {mfa, {rabbithub_subscription, start_subscriptions, []}},
                    {requires, routing_ready}]}).
```

… to conform to newer RabbitMQ plugin architecture model (also deals to a conflict with the Mochiweb wrapper)

```
start(_Type, _StartArgs) ->
    setup_schema(),     %% uses mnesia to store subscription details
    ssl:start(),

    %% Sort out HTTP client options
    HttpOpts = get_env(http_client_options, []),
    ok = httpc:set_options(HttpOpts),
    {ok, Opts} = httpc:get_options(all),

    {ok, Pid} = rabbithub_sup:start_link(),
    rabbithub_web:start(),
    rabbithub_subscription:start_subscriptions(),

    rabbit_log:info("RabbitHub started~n"),
    rabbit_log:info("RabbitHub HTTP client options:~n~p~n", [Opts]),
    {ok, Pid}.
```

```
{application,rabbithub,
         [{description,"rabbithub"},
          {vsn,"3.2.1"},
          {modules,[rabbithub,rabbithub_auth,
                    rabbithub_consumer,
                    rabbithub_pseudo_queue,
                    rabbithub_subscription,
                    rabbithub_subscription_sup,
                    rabbithub_sup,
                    rabbithub_web,simple_httpc]},
          {registered,[]},
          {mod, {rabbithub, []}},
          {env,[{listener,[{port,15670}]},
                {default_username,"guest"}]},
          {applications,[kernel,stdlib,
                         crypto,rabbitmq_web_dispatch]}]}.
```

Rebar-based build

All too easy! I like rebar...

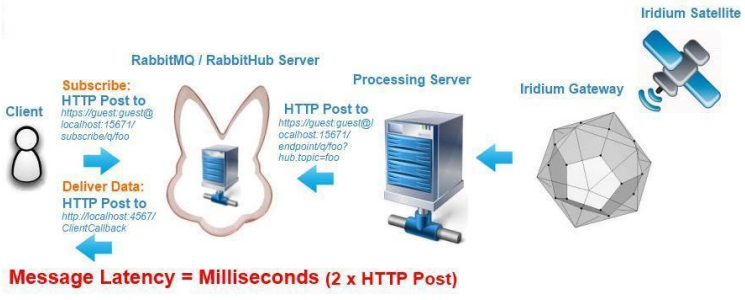
```
$ git clone https://github.com/brc859844/rabbithub
$ cd rabbithub
$ ./rebar get-deps
$ ./rebar compile
```

```
%% rebar.config
{
  deps, [
    {rabbit_common, ".*", {git, "https://github.com/brc859844/rabbit_common.git", {tag, "rabbitmq-3.2.1"}}}
  ]
}.

{erl_opts, [
  debug_info,
  compressed,
  report,
  warn_export_all,
  warn_export_vars, warn_shadow_vars, warn_unused_function, warn_deprecated_function, warn_obsolete_guard, warn_unused_import
]}.
```


The resultant solution

- It was a simple task to fold the PSHB-based solution into the existing Lucid Logistics message delivery stack
- New data is securely POSTed to subscribers' callback URLs as soon as it comes in
- Messages pushed to subscribers in JSON format
- The overhead of polling is eliminated
- Message latencies go from minutes to milliseconds
- RabbitMQ/RabbitHub and Processing Server can be clustered for HA and scalability



Subscribing to a feed with RabbitHub

- Subscribing to a feed (topic) is straightforward:

```
$ curl -d \
"hub.mode=subscribe&hub.callback=http://localhost:4567/sub&hub.topic=foo&hub.lease_seconds=600" \
http://guest:guest@localhost:15670/subscribe/x/amq.direct
```

  - Any messages published to the exchange "`amq.direct`" with routing key "`foo`" will be forwarded by RabbitHub to the callback URL `http://localhost:4567/sub`
  - This subscription will expire after 600 seconds
  - Subscriptions can be to a queue or to an exchange
    - When subscribing to an exchange RabbitHub creates an exclusive queue for the subscriber and binds it to the exchange using the supplied binding details (namely the specified topic)
    - It is possible to subscribe to any RabbitMQ exchange type
  - Messages can be published using other protocols supported by RabbitMQ (AMQP, STOMP, MQTT, ...)
    - RabbitHub does not care how messages get into queues from which it is servicing subscribers

Management tasks

RabbitMQ Management HTTP API was used to integrate with Lucid Logistics backend

- Automates user and queue related administrative functions
- Creates/deletes a queue per customer each with individual credentials to support multi-tenancy
- New customer administrative setup takes less than a minute

AGENDA

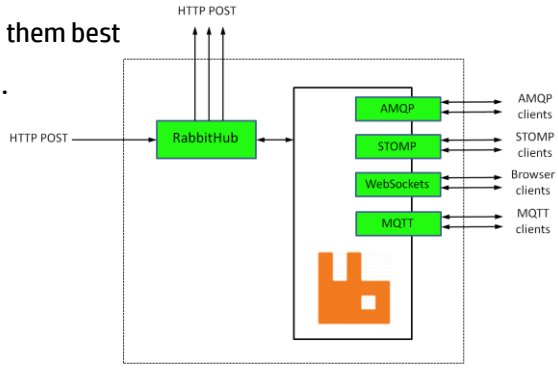
- Introduction
- The challenge
- The solution
- **Current status and potential future developments**
- Conclusions/summary
- Questions

Current status

- The implementation is in production
- Currently services a limited number of customers that agreed to participate in a pilot
- API access is requested by prospects in majority of sales calls especially related to large opportunities

Potential future developments

- Recently started using RabbitMQ for satellite messaging queues
- Other uses within the Lucid Logistics message processing infrastructure are being considered, e.g. dealing with message processing errors
- Multi-protocol messaging hub
  - Customers can use whichever protocol suites them best
    - AMQP, MQTT, STOMP, WebSockets, HTTP, …
    - No need to change application architecture



AGENDA

- Introduction
- The challenge
- The solution
- Current status and potential future developments
- **Conclusions/summary**
- Questions

Summary

- Introduction of real-time business requirements often means that existing application architectures need to be re-evaluated and changed
  - Instead of allowing external services to poll and query the data store at will via SOAP and face data delays, the architecture needed to push data to subscribers in real-time
  - By applying Open Source Erlang-based technologies, Lucid Logistics has developed powerful pub/sub-based architecture and are now able to deliver business-critical data to their customers in near-realtime.

    The development team also immediately saw big potential behind RabbitMQ and benefits of using it during various stages of message processing in the existing infrastructure.
  - Lucid Logistics was able to build an enterprise-level solution with all the capital and operating expense benefits of Open Source software.

    The team also found that RabbitMQ had an active, extensive community to lean on far superior to Google's PUSH.

AGENDA

- Introduction
- The challenge
- The solution
- Current status and potential future developments
- Conclusions/summary
- **Questions**


Thank you