May15-17

Advisor: Dr. Mitra

Lighthouse Final Document

Client: Workiva

Version 1.1

Caleb Brose, Chris Fogerty, Nick Miller, Rob Sheehy, Zach Taylor

April 28, 2015

CONTENTS

Contents 1
1. Introduction 3
Project Definition 3
Project Goals 3
Definitions of Common Terms and Abbreviations 3
2. Deliverables 4
Lighthouse 4
Harbor 4
Beacon 4
Documentation 4
3. System Level Design 5
System Requirements 5
Use Cases 5
Functional Requirements 5
Non-Functional Requirements 7
Functional Decomposition 8
High Level Decomposition 8
Harbor 9
Lighthouse 10
Beacons 10
4. Detailed Description 11
Interface Specifications 11
Lighthouse API 11
Streaming 11
Lighthouse and Beacon 11
Harbor 13
Deployments and Applications 13
Using Applications 14
Design and Implementation 14
Technical Issues 16
Why Do Beacons Exist? 16
Authentication 16
Streaming 16
5. Testing 17

6. Conclusion 18
7. Appendix I: User Manual 19
Local Development 19
Software Requirements 19
Pulling the Source 19
Installing Dependencies 19
Building 19
Running 19
Testing 20
Playing Around 20
Production Environment 21
Preferred Operating System 21
Setup Beacon 21
Setup Lighthouse 22
8. Appendix II – Previous Versions 23
Addition of Beacons 23
Database Paradigms 24
Beacon Implementation 25
Application Management 25
9. Appendix III: Other Considerations 26
Single Page Application Architecture 26
AngularJS and Go 26
Requirements Gathering 26
Future Work 27
10. Appendix IV Table of Tables 28
11. Appendix V Table of Images 28

1. INTRODUCTION

Project Definition

Within the past year, Docker has emerged as an execution and deployment environment for distributed applications. It supports standardized application installation and execution via "containers" built from a common template that can run on a variety of host platforms. In the past, the efforts to standardize distributed and scalable applications were focused on the virtual machine, which was replicated across the desired host platforms. Docker removes the overhead of hosting and running the virtual machine, and instead utilizes the native system resources, while maintaining the desired isolation one would like to see between multiple applications running on one system. Docker has achieved near native performance on application startup and shutdown, meaning new applications can be deployed and scaled as quickly as possible.

Project Goals

Our goal with Lighthouse is to provide a Docker host management system, capable of interfacing with a variety of cloud providers including, but not limited to, Google Compute Engine, and Amazon Web Services. Lighthouse will allow users to authenticate with their target Docker host platform, monitor the state of their application instances, and deploy new containers from one centralized interface, significantly reducing the time it requires to deploy and monitor applications across multiple cloud providers. By the project completion, our goal is to have built a system that is efficient, modular, and secure.

Definitions of Common Terms and Abbreviations

Term	Definition
Docker	An open source platform used for creating, packaging, and shipping, and running applications.
Image	A read only file system used by Docker which contains applications and other files.
Container	A "running" image which is writeable. Containers are primarily used to run the applications that are being developed.
Registry	An image storage service.
GCE ***Google Compute Engine***	A virtual machine (VM) hosting service by Google which may host Docker instances.
AWS ***Amazon Web Service***	A virtual machine hosting service by Amazon which may host Docker instances.
JSON ***JavaScript Object Notation***	A data exchange format which is the primary mode of communication between the frontend and backend applications.
Beacon	A portion of the Lighthouse Project ecosystem. It lives in the cloud and provides a single access point for Docker instances in the cloud.

Lighthouse	The centralized backend of the Lighthouse Project ecosystem. It provides an API to access all the Docker systems owned by an organization.
Harbor	The front-end web application built specifically for use with the Lighthouse Project. It provides a responsive interface to Docker.

Table 1 Common terms and their definitions

2. DELIVERABLES

The Lighthouse Project consists of the following modules.

Lighthouse

The Lighthouse system is capable of interfacing with an existing instance of Docker. It communicates with Docker instances via Docker's REST API, and it is capable of:

- Routing all valid Docker API calls through itself to the appropriate Docker instance.
- Authenticating users, using per-instance permissions, and blocking unauthorized requests.
- Extending Docker's functionality by intercepting requests to Docker and calling internal functions, such as logging and history.
- Perform batch operations over a collection of Docker instances.

Harbor

Harbor is a web application capable of utilizing Lighthouse to manage Docker instances. It uses JavaScript, AngularJS, and Oboe to:

- Perform basic container management actions such as Create, Stop, Remove, Update, and Pause
- View past containers run in the Docker instance. That is, view a history of each application running on the instance.
- Log in and out of the backend application to authenticate frontend users.
- Stream the progress of batch operations to the user.

Beacon

The beacon module resides within the cloud provider's network. This gives it unique access to the provider's network-internal APIs for discovering other servers being rented by the user. The beacon module can:

- Detect what provider network it is running on, in order to select the correct API.
- Detect Docker instances owned by the user and running within the same network.
- Redirect requests from the Lighthouse central back-end to the relevant Docker instance.

The beacon module also includes its own API, so support for new cloud providers can be added with ease.

Documentation

The Lighthouse API is well documented in order to accommodate users that wish to automate actions, create their own frontend applications, or extend Harbor with new functionality. This documentation can be found on GitHub, under the Lighthouse organization.

3. System Level Design

System Requirements

Use Cases

For a successful implementation of our **minimal viable product**, we've identified two major actors in the Lighthouse system: **release manager**, and **IT admin**.

Release Manager Functions

1. Log in / Authenticate
2. View currently deployed containers
3. Deploy and start a new container
4. Rollback a container deploy to a previous version

IT Administrator Functions

1. Log in / Authenticate
2. Initialize provider with authentication credentials
3. Create and delete system users
4. View provider statistics and analytics

Functional Requirements

- Docker addresses
    - Description
        - The IP address that are hosting Docker daemon publicly on port 2375.
    - Users
        - Administrators
    - Actions
        - add
        - remove
        - edit
- Docker Images
    - Description
        - Each Docker daemon has a set of images referenced by name or UID.
    - Users
        - Developers/Administrators
    - Actions
        - add
        - remove
        - edit
- Docker Containers
    - Description
        - Each Docker daemon has a set of running containers referenced by name or UID.
        - Containers are spawned from images pre-existing inside a Docker daemon.
    - Users

        - Developers/Administrators
    - Actions
        - start
            - Arguments
                - exposed port
                - command
                - environment variables
        - stop
- Docker Analysis
    - Description
        - Should be able to view running containers in real time.
    - Users
        - Developers/Administrators
    - Actions
        - view
            - logs
            - CPU usage
            - memory consumption
- Authentication
    - Description
        - Given an email and password all users can be denied or granted access to the webapp.
    - Users
        - Everyone
    - Actions
        - login
            - Arguments
                - Email
                - Password
        - logout
- Authentication Accounts
    - Description
        - Admins should be able to control user accounts.
    - Users
        - Administrators
    - Actions
        - add
        - remove
        - edit

Non-Functional Requirements

Security System (should be secure and not allow unauthorized control)

- Protect against
    - XSS attacks
    - session hijacking
    - unauthenticated requests
    - exposure of sensitive container/server data
- Only use HTTPS to mitigate various communication security exploits.

Code Quality (should be easy to maintain and understand)

- Git
    - Commit comments should be clear and concise, standard commit conventions
- Style Guides
    - GO
        - style guide
        - effective tips
    - JavaScript
        - style guide
- Code reviews must include at least 2 other team members to be verified for master merge.

Functional Decomposition

High Level Decomposition

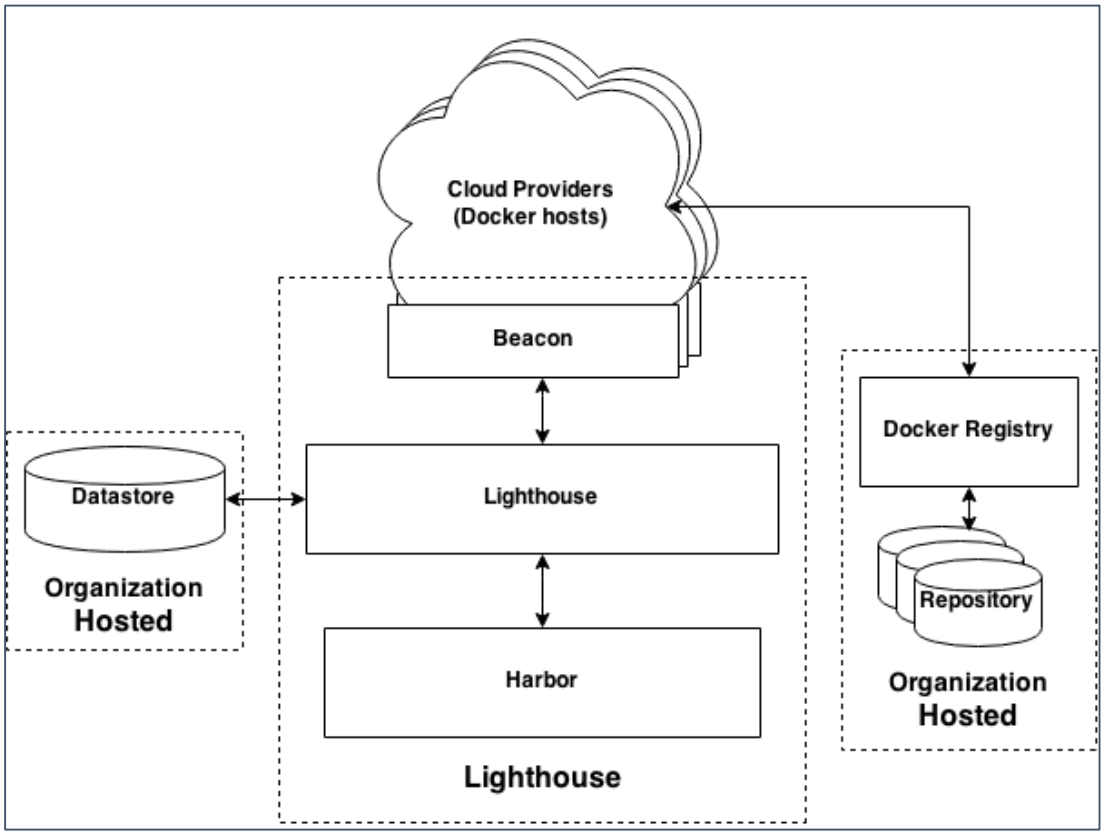

Figure 1 System Block Diagram

The overall architecture of Lighthouse is built upon the common client-server pattern. Users interact with the system via a "single page" web application that's loaded once into their browser, with subsequent system requests made via a REST API over HTTPS, utilizing JSON as the data interchange format.

Requests received from the web interface are routed through authentication and API control. Our API is implemented as a superset of the standard Docker Remote API, allowing users to send specific Docker requests to their target platform, as well as add new platforms, users, and other infrastructure configuration.

Host providers manage the actual startup and application runtime, but are controlled via the Lighthouse main controller. Provider interfaces are installed on each provider as part of the Lighthouse application and define a common interface for communication between the host network and Lighthouse.

Docker images are hosted on the Docker registry provided by the organization using Lighthouse. Host providers are instructed to pull new images from the registry on application deployment.

Harbor

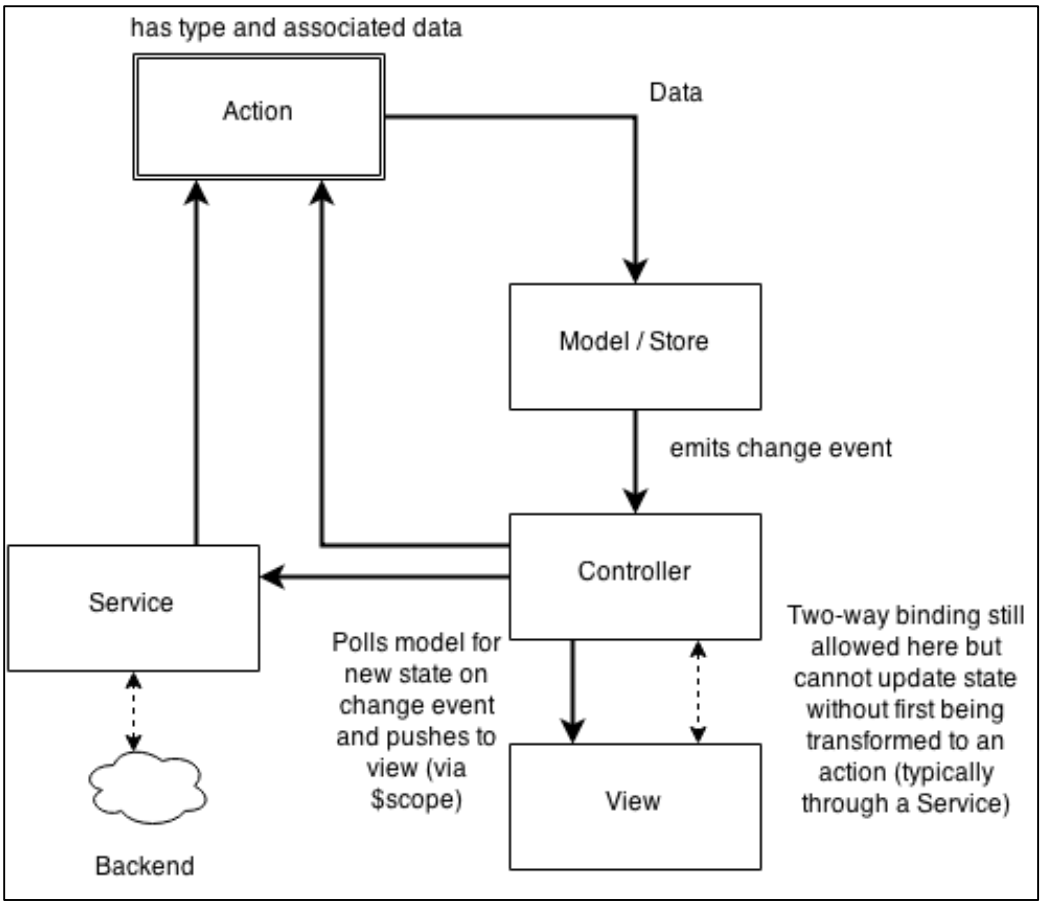

Figure 2 Harbor block diagram

Harbor's goal is to present the user with information and provide the user with a simple UI to manage their array of Docker installations. Using the API exposed by Lighthouse, Harbor renders the response from Lighthouse into HTML and displays it to the user. Harbor must know how to handle any errors in communicating with the Lighthouse server, manage user sessions (for logging in and out), and generate API requests to Lighthouse.

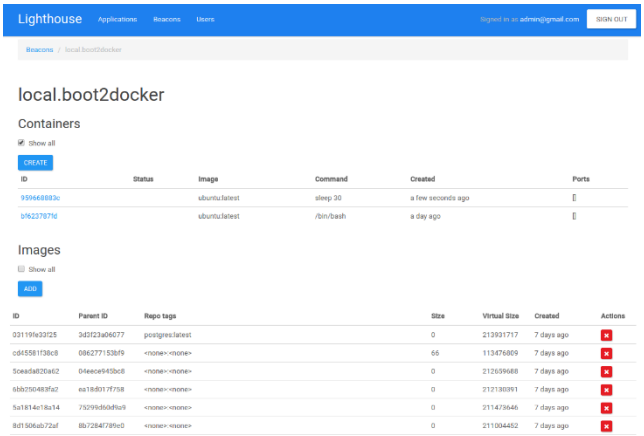

Figure 3 An example of Harbor's UI

Lighthouse

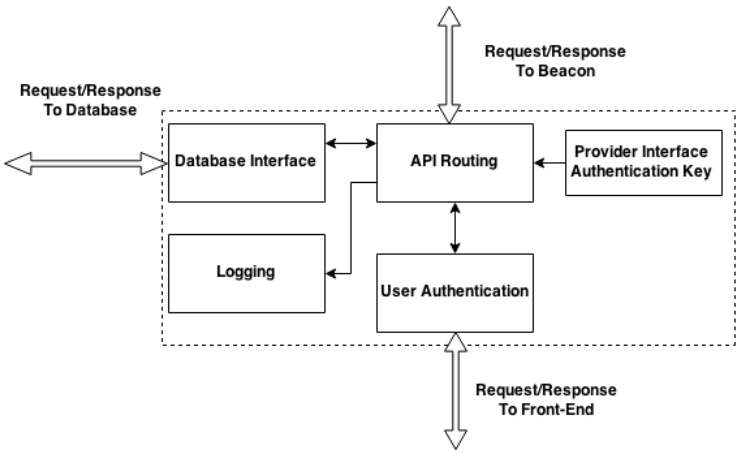

Figure 4 Lighthouse block diagram

It is Lighthouse's job to authenticate users, cache Docker hosts from the provider interface, and provide features on top of what Docker already provides, such as rolling back deployments. Lighthouse exposes a REST API (described below) that anyone can build off. While we use it as a way to communicate with the front-end, the API allows for users to build easily build automation into their systems. The back-end communicates with provider interface(s) in order to discover any cloud instance that is owned by the user and running Docker. It must also elegantly handle errors by returning a sane response to the client, and log any errors or events that pop up.

Beacons

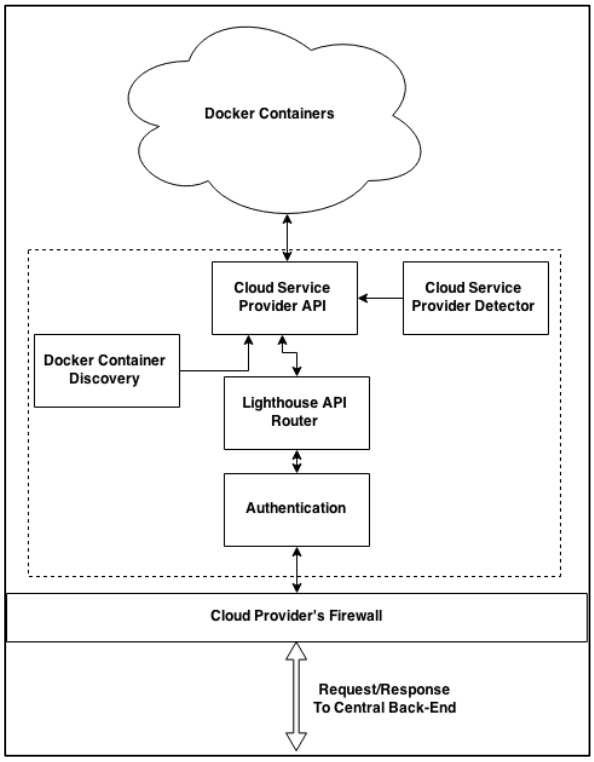

Figure 5 Beacon block diagram

The beacon is installed within each cloud segment the user wants to access. It allows us to use a single API on the back-end to locate relevant cloud instances across all providers. By being in the relevant cloud, the beacon is granted access to privileged APIs, allowing it to discover other instances owned by the same account.

4. Detailed Description

Interface Specifications

Lighthouse API

The Lighthouse module provides a REST API as an interface between the UI and the back end logic. This API exposes endpoints that allow clients to control Docker, authenticate users, modify users and permissions, interact with beacons, and manage applications. The full API documentation is a bit large to fit here, but can be found on GitHub, under the Lighthouse organization.

Streaming

Lighthouse's core functionality revolves around the batch processing of large deployments. As the number of deployments scales up, so does the latency of the processing. The combination of multi-threaded processing and incremental status updates becomes critical functionality as the instance count in a deployment reaches into the hundreds.

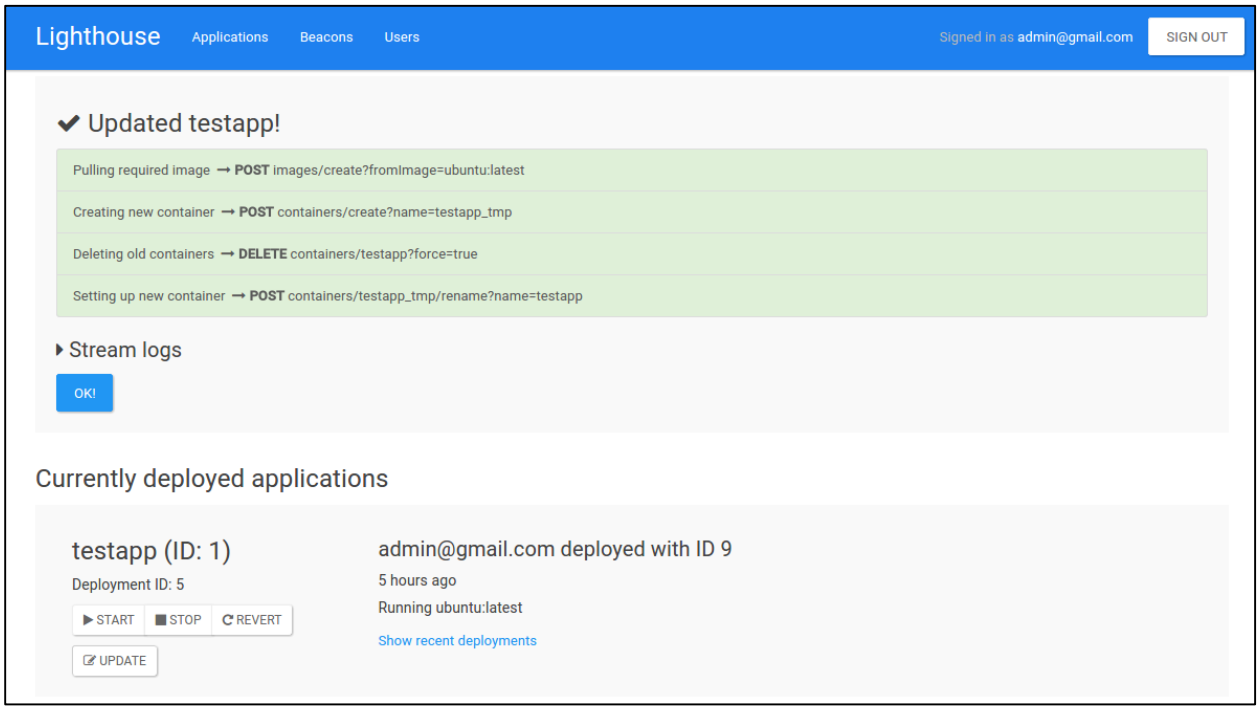

Figure 6 Streaming UI in Harbor

Lighthouse and Beacon

To successfully build out a stream oriented API, our design required coordination between each major component. Lighthouse, acting as the core mediator between each component in a request, takes two approaches to handling response streams. The first is focused on Docker requests.

Incoming Docker requests are routed to the appropriate Beacon instance, which, in turn, forwards the request onto the targeted Docker instance. The open TCP connections are read from and written to in pre-defined byte segments, instead of the entire response. The diagram below illustrates the connection relationship and read/write characteristics.

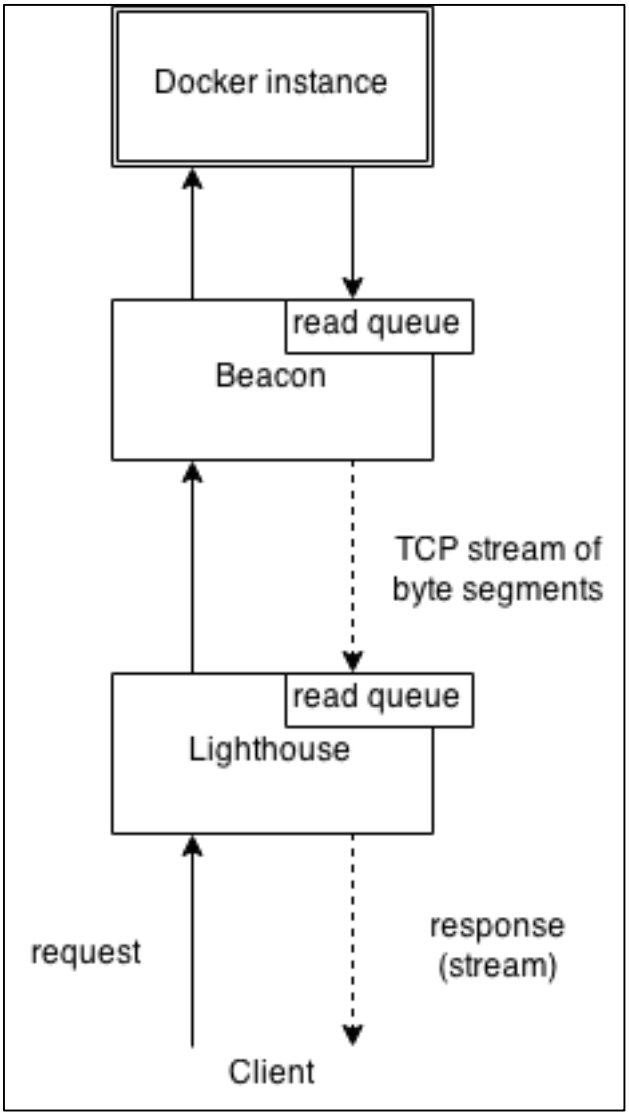

Figure 7 Streaming concept diagram

The implementation for the read/write portion of a component in the stack above is relatively straightforward. A pre-defined byte segment is read from the response buffer and relayed to the next client as a byte segment of the same size. See Appendix IV.

One important aspect to consider is the nature of the listening client. A nice property of this implementation is that it will work for any client expecting either a streaming response or blocking response. A blocking client will not process its response until all incoming bytes have been stored and the connection closed. In this case, the underlying transport layer (i.e. TCP) seamlessly manages the incoming buffer, passing it to the application layer (i.e. client) once the response is complete. Therefore,

all Docker responses from instances are treated as if they were streams, since clients gracefully handle either case.

Lighthouse's second approach is focused on the application API. Here, incremental status objects as described the Applications section are written to the response stream and flushed immediately. The end result here is the same as in Docker responses.

Harbor

Harbor's internal services handle asynchronous requests and responses by utilizing the underlying browser API. Asynchronous HTTP requests (commonly referred to as Ajax) have been prevalent for quite some time, and many Javascript libraries take advantage of this, providing a clean interface to develop against.

```
oboe({
            method: 'POST',
            url: _url(request),
            body: request.body || {}
        })
        .start(function () {
            // process stream start
        })
        .node('{Status}', function (status) {
            // process status object
        })
        .fail(function (error) {
            // process failure
        });
```

Figure 8 Stream Processing code example

In Harbor's case, it utilizes oboe.js, to process and dispatch internal events as the incoming streaming is processed. Oboe's API model revolves around the knowledge of the stream structure (as a JSON data segment) to trigger state specific processing. An example of this processing is given above, for the application deploy stream.

Effectively, oboe watches the incoming JSON stream for specific attributes (in this case, a specific key, "Status") and calls the registered callback function for that event. Harbor can then perform its own asynchronous data processing (i.e. update the UI) to show the incremental deploy status.

Docker image pull streams are handled in a similar manner, but instead, a progress bar is updated during the callback. Ultimately, the consideration of streaming APIs and incremental status updates sets up an environment for more responsive UIs and a smoother user experience. Streaming functionality can be further exploited to show real-time network statistics and instance resource load.

Deployments and Applications

Application management is one of Lighthouse's key features in terms of extending Docker's existing functionality. When a developer thinks of their application, they are probably imagining one object running in one location. This abstraction works perfectly well for writing applications, but a release manager does not have this luxury. When a new version of an application is created, the release manager has to think of the application as many objects running across many machines, these machines

are in fact the instances of Docker which run the application. The actual process of updating an application is both time consuming, tedious, and error prone as the release manager must interface with each of the potentially hundreds of instances to manually perform the update one at a time. With our application management features, we provide a layer on top of this which allows the release manager to think of the application like a developer, as a single entity.

Using Applications

Creating an Application

There's a first time for everything and the same is true for applications. In order to get an application up and running, all the user has to do is provide a set of Docker instance which will actually be running the application and the set of instructions telling the application what to do. From here, Lighthouse handles all the busy work of telling each instance to start running the new application.

Updating an Application

Of course, no application ever has a single version and some applications will take off faster than others. To accommodate, we let the user update their application in a very painless manner. To update the code running in the application, the user only has to provide a new set of instructions just like they did when they created the application. Applications can be scaled up or down in an equally painless way, the user only has to tell Lighthouse any instances they would like to add or remove from an application and Lighthouse will handle the rest.

Reverting an Application

Unfortunately, mistakes are a very real event and when a bug in the code is bringing an application down it needs to be corrected as quickly as possible. Lighthouse can be instructed to rollback applications any number of stages, all the way back to the original version. This makes the rollback process painless as the user can go back to a stable version without even having to enter a command.

Design and Implementation

Database Design

Applications are stored in two tables, one storing the applications themselves and another which contains the different versions of the application which we call deployments.

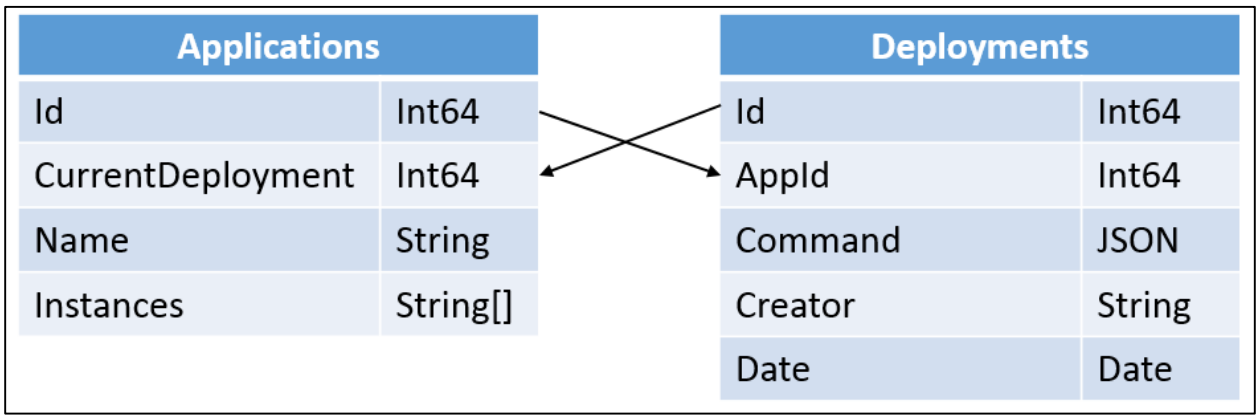

Figure 9 Deployment and Application database design

This design allows us to quickly get general information about an application by simply querying the Applications table and lets us perform new deployments by appending a new row to the Deployments table with the new command.

Batch Processing

The key feature of application management is sending requests to many instances all at once, we do this through batch processing. For Lighthouse, a batch process needs only a list of instances to communicate with and the message it needs to tell them. From here the batch process will send requests to each instance in parallel which will make the entire process much faster. To keep the user informed, the process will create an update stream which is used to report successes, failures, and any administrative data the user would need to know. If any errors were to occur during the process, a rollback procedure is performed to ensure that all instances are always in the same state as one another. There are certainly cases where a single request from the user will need to send multiple requests to each instance (an application update for example). When multiple requests are needed, each request waits for the one before it to complete before starting.

Concurrent request processing is easy to implement by using Go's goroutine and channel functionalities. Channels operate as thread-safe queues and goroutines are easy to make threads which each carry out one request to a single instance. The goroutines carry out their function and then report their status to the master thread via channels. The master thread simply spawns the thread and waits for them to end.

Streaming Responses

For all of Lighthouse's application management functions which require batch operations, the response to the client is in the form of a stream which is constantly updated. The reason for this is to provide the client with real-time updates of what is happening server-side rather than having to wait for all processing to complete (which could take several minutes) before reporting anything. These updates have the form:

```
{
   Status    ("OK", "Warning", "Error", "Starting", "Complete", "Finalized")
   Method    ("GET", "POST", "PUT", "DELETE")
   Endpoint  ("containers/create", ...)
   Message   ("", "<Warning Reason>, "<Endpoint>", ...)
   Code
   Instance  ("myInstance.com:5000/v1.12")
   Item      (0 to Total)
   Total
}
```

Figure 10 Streaming response, general form

The issue is that when Lighthouse sends out requests in a batch operation, there isn't really any way of knowing in what order they will complete (due to things like request latency or processing power of the destination instance). Lighthouse does, however, know when it creates new batch request and when that request has completed. For this reason, the update stream is formatted as a series of blocks which begin with an update with status "Starting" and end with an update with status "Complete". The blocks in turn wrap a series of updates relaying responses from individual instances which will tell the client whether the block completed successfully or not. Because many different requests will return a stream, each requiring a different number of blocks, there must be a way for the client to know that the request

is complete. To accommodate, Lighthouse sends a single update with status “Finalized” when the request is fully completed, regardless of success or failure. The client should only consider the request complete once this update is received. An example of an entire update stream is shown below.


```
{"Status":"Starting", "Method":"DELETE", "Endpoint":"containers/my_app",
"Message":"Deleting old container", "Code":0, "Instance":"", "Item":0,
"Total":4}

{"Status":"OK", "Method":"DELETE", "Endpoint":"containers/my_app",
"Message":"", "Code":404, "Instance":"instance0.com", "Item":0, "Total":4}

{"Status":"Warning", "Method":"DELETE", "Endpoint":"containers/my_app",
"Message":"No such container", "Code":404, "Instance":"instance2.com",
"Item":2, "Total":4}

{"Status":"Error", "Method":"DELETE", "Endpoint":"containers/my_app",
"Message":"", "Code":400, "Instance":"instance1.com", "Item":1, "Total":4}
{"Status":"Error", "Method":"DELETE", "Endpoint":"containers/my_app",
"Message":"host unreachable", "Code":500, "Instance":"instance3.com",
"Item":3, "Total":4}

{"Status":"Complete", "Method":"DELETE", "Endpoint":"containers/my_app",
"Message":"Deleting old container", "Code":0, "Instance":"", "Item":4,
"Total":4}

{"Status":"Finalized", "Method":"", "Endpoint":"", "Message":"", "Code":0,
"Instance":"", "Item":0, "Total":0}
```


Figure 11 Update Stream example

Technical Issues

Why Do Beacons Exist?

Beacons came about as a result of our desire to make it as easy as possible to get the Lighthouse ecosystem up and running. We realized that many organizations may already have Docker swarms set up, and it would be tedious to add hundreds of Docker instances to Lighthouse manually. At the same time, we were searching for a solution to make it easy for Lighthouse to communicate with hundreds of Docker instances. This would also be tedious to set up if each Docker instance required its own static (and public-facing) IP address. The solution we designed was the Beacon module.

Authentication

This issue arose in the middle of March 2015. Everything was coming together, and we were about to get user management features online, when two team members caused the Lighthouse authentication state and the Harbor authentication state to get out-of-sync. This caused issues where Harbor indicated to the user that they were logged, but Lighthouse had no recollection of this session. The solution was to generate a notification on the server to automatically inform the client of its authentication state.

Streaming

The idea for streaming status happened over a code jam, when we realized Harbor had no way of indicating how far an operation (especially a batch operation) had completed before failing. It was also

not possible to view status, progress, or logs without manually refreshing the page. This contradicted our goal of having a quick response time for user interactions.

This issue was especially challenging due to the nature of HTTP responses. While the protocol supports streaming, the response (and stream) needed to work its way along three hops to the final Docker instance, requiring a lot of coordination from Lighthouse and Beacon.

5. Testing

The Lighthouse testing and code quality process consists of multiple steps that ensure a high quality product is being created, while also allowing rapid development and continuous release. We use a test-driven development process to continuously deliver tested code throughout our development of Lighthouse. This gives the project auditability and a solid set of standards.

Since we use Git and GitHub, our development process revolves around pull requests. These are requests to merge new code into the master branch of our repositories. Once a developer has completed their task and is ready for it to be reviewed, they create a pull request. An example is here: https://github.com/lighthouse/lighthouse/pull/50. On this pull request, developers can view the file additions and deletions, discuss and review the code, and ensure that certain checks have been met. This methodology allows developers to review other's code and approve. This leads to a very high quality product.

On each pull request a certain set of checks must be completed. Our process requires the following for new code to be merged: the full test suite must pass, code coverage must not drop below a certain threshold, and one of two other developers must sign off on it. We use two services to assist with this process. We use Travis CI (https://travis-ci.org) to run the full test suite on every commit and report to GitHub whether they passed or failed. Coveralls (https://coveralls.io) helps us track code coverage for the test suite. If the coverage decreases by a certain threshold (around 6%) a failure is indicated on the pull request. This ensures that we meet a testing standard and that new code includes tests. The following screenshot shows what that checklist looks like.

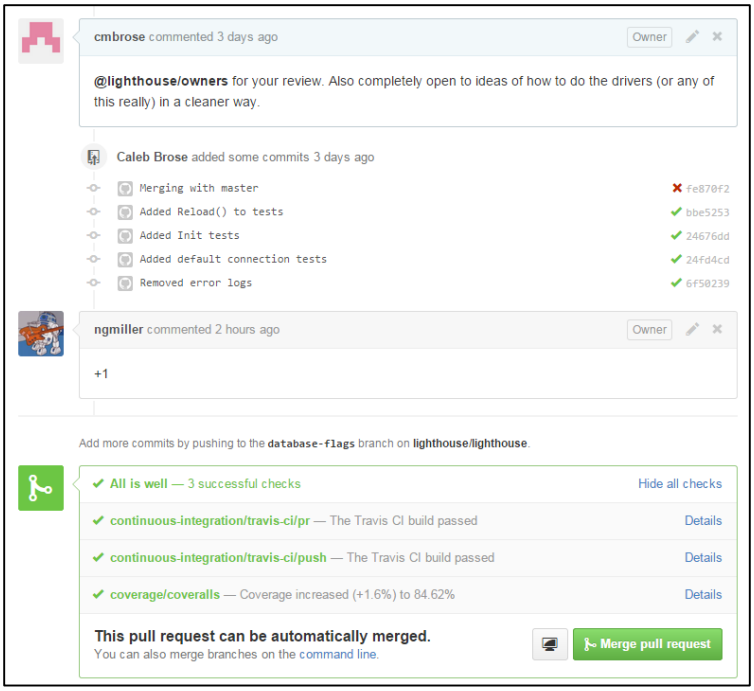

Figure 12 Automated Testing checklist

6. Conclusion

With the advent of Docker, there has been a torrent of new technical challenges surrounding the technology. Now that Docker has landed and has become an important and accepted industry standard, the foundation has been created. Some of the new challenges will be around building on top of this technology, particularly management, security, and development of best practices.

Lighthouse's main purpose is to tackle the management of the Docker platform. Our main goal is to make management of this platform simple, efficient, and secure. By communicating frequently with potential stakeholders, making the project open, and using standard and modern technologies, we will be able to accomplish the goals outlined in this document, which will allow Lighthouse to be a successful and useful tool for Docker management.

7. Appendix I: User Manual

Local Development

Software Requirements

The following items are required for building and running in a development environment. Make sure this software is installed on your hardware before you do anything else!

- Docker (note) install boot2docker for OSX and windows operating systems
- Node
- Go
- Git

Pulling the Source

Once you have the software installed, it's time to clone the source code.

- Harbor git clone git@github.com:lighthouse/harbor.git
- Lighthouse go get -t github.com/lighthouse/lighthouse
- Beacon go get -t github.com/lighthouse/beacon

Installing Dependencies

As soon as you've pulled everything it's time to install the project dependencies.

- Harbor
    - npm install -g bower
    - bower install
    - npm install -g gulp
    - npm install -g karma-cli
    - npm install
- Lighthouse
    - docker pull postgres:latest

Building

Now that dependencies are installed, it's time to build each project.

- Harbor
    - gulp build --gopath
- Lighthouse
    - go install github.com/lighthouse/lighthouse
- Beacon
    - go install github.com/lighthouse/beacon

Running

Once we've compiled everything, it's time to get down to the nitty gritty of running everything.

- Lighthouse
    - cd $GOPATH/src/github.com/lighthouse/lighthouse
    - docker run -p 5432:5432 -d postgres:latest
    - $GOPATH/bin/lighthouse --databases-reload

- Beacon
    - $GOPATH/bin/beacon -token foobar -h 0.0.0.0:5001 -driver local

Testing

Now for everyone's favorite part, testing.

- Harbor
    - npm test
- Lighthouse
    - go test github.com/lighthouse/lighthouse/... -cover
- Beacon
    - go test github.com/lighthouse/beacon/... –cover

Playing Around

Now that everything is happily running, let's do some magic.

- Visit localhost:5000/login
- Enter in your credentials
    - (Default) Email: admin@gmail.com
    - (Default) Password: admin

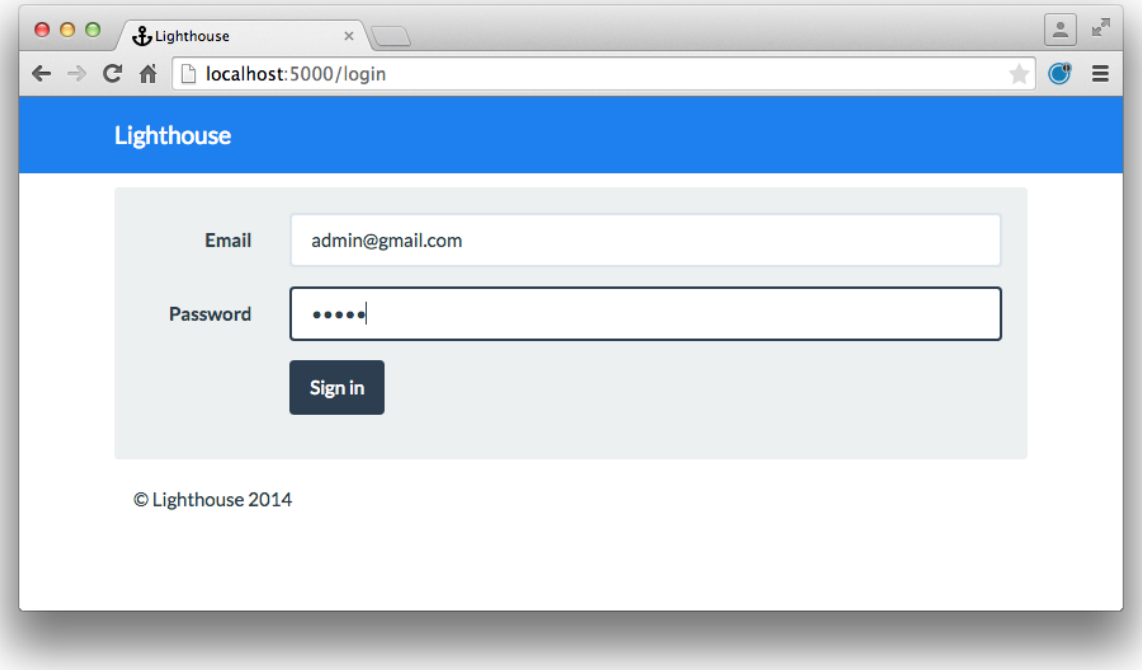

Figure 13 Logging in

- Click "Add Beacon" and administer the following information
    - Alias: Local Beacon
    - Address: 127.0.0.1:5001
    - Token: foobar

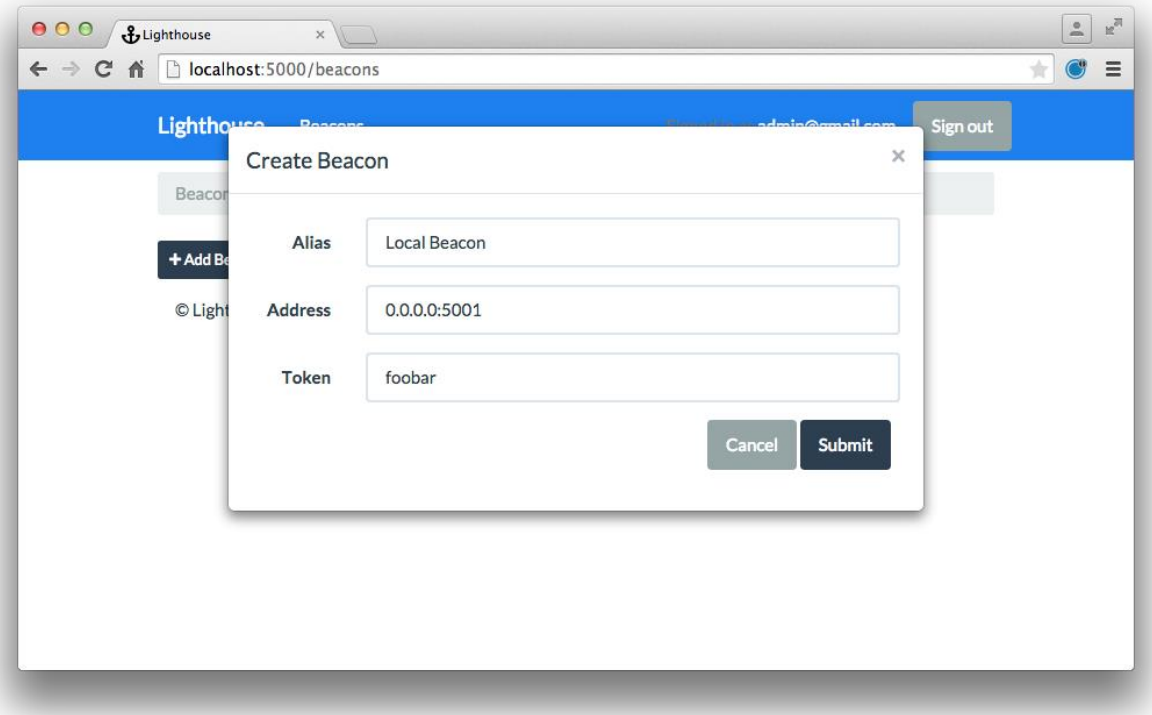

Figure 14 Connecting to a Beacon

Production Environment

Preferred Operating System

For all of these projects we recommend using any Linux distribution that runs Docker as the OS. Throughout the project we favored CoreOS.

Setup Beacon

Right now Beacon is configured to run on two cloud providers out of the box. Those two cloud providers include Google Compute Engine and Digital Ocean. Refer to the GitHub project read me for setting up Google Compute Engine and Digital Ocean for Beacon. Otherwise Beacon offers the option of allowing a configuration file to define which VMs are available in a network. For information on Beacon configuration files read this.

- Pull Image
  - docker pull lighthouse/beacon:latest
- Basic Run
  - docker run -d -p 5000:5000 lighthouse/beacon:latest -h 0.0.0.0:5000
- Using SSL
  - docker run -d -p 5000:5000 -v /path/to/cert/dir:/certs lighthouse/beacon:latest -key /certs/server.key -pem /certs/server.pem -h 0.0.0.0:5000
- Force Driver
  - docker run -d -p 5000:5000 lighthouse/beacon:latest -driver config -h 0.0.0.0:5000
- Use Custom Authentication Token

- Docker run -d -p 5000:5000 lighthouse/beacon:latest -token imasecret -h 0.0.0.0:5000

Setup Lighthouse

- Pull Image
  - docker pull lighthouse/lighthouse:latest
  - docker pull postgres:latest
- Start Database
  - docker run --name postgres-image -d postgres:latest
- Basic Run
  - docker run -t -i --rm -p 5000:5000 --link postgres-image:postgres lighthouse/lighthouse:latest
- Load Credentials Into Database From Config File
  - docker run -t -i --rm -p 5000:5000 --link postgres-image:postgres lighthouse/lighthouse:latest -databases-reload

8. Appendix II – Previous Versions

During development, the design of Lighthouse was constantly reevaluated as we gained a better and better grasp of the problem domain. There were very few modifications that would be noticed at a high level view, the most prominent being the addition of Beacons. Instead most design changes were hidden inside the lower parts of the technology stack, examples of this being database paradigms, Beacon implementations, and Application management.

Addition of Beacons

Early in design we intended for there to be only one end-to-end backend system which would have all the functionality now provided by Lighthouse as well as the abstractions provided by Beacon. This would be achieved by having the provider drivers intercept outgoing requests which applied to the systems they managed and applying any additional headers or formatting that was required to access that system. The figure below illustrates this early vision of Lighthouse. This model was updated to the current version for two primary reasons. The first was simply that the interception management would have added a lot of complexity to Lighthouse's pipeline. The second, and more important, reason was that this model was not completely feasible as many cloud providers will have firewalls which would have blocked the requests coming from Lighthouse. By installing Beacons inside the provider itself, we allow Lighthouse to bypass this firewall in a secure manner.

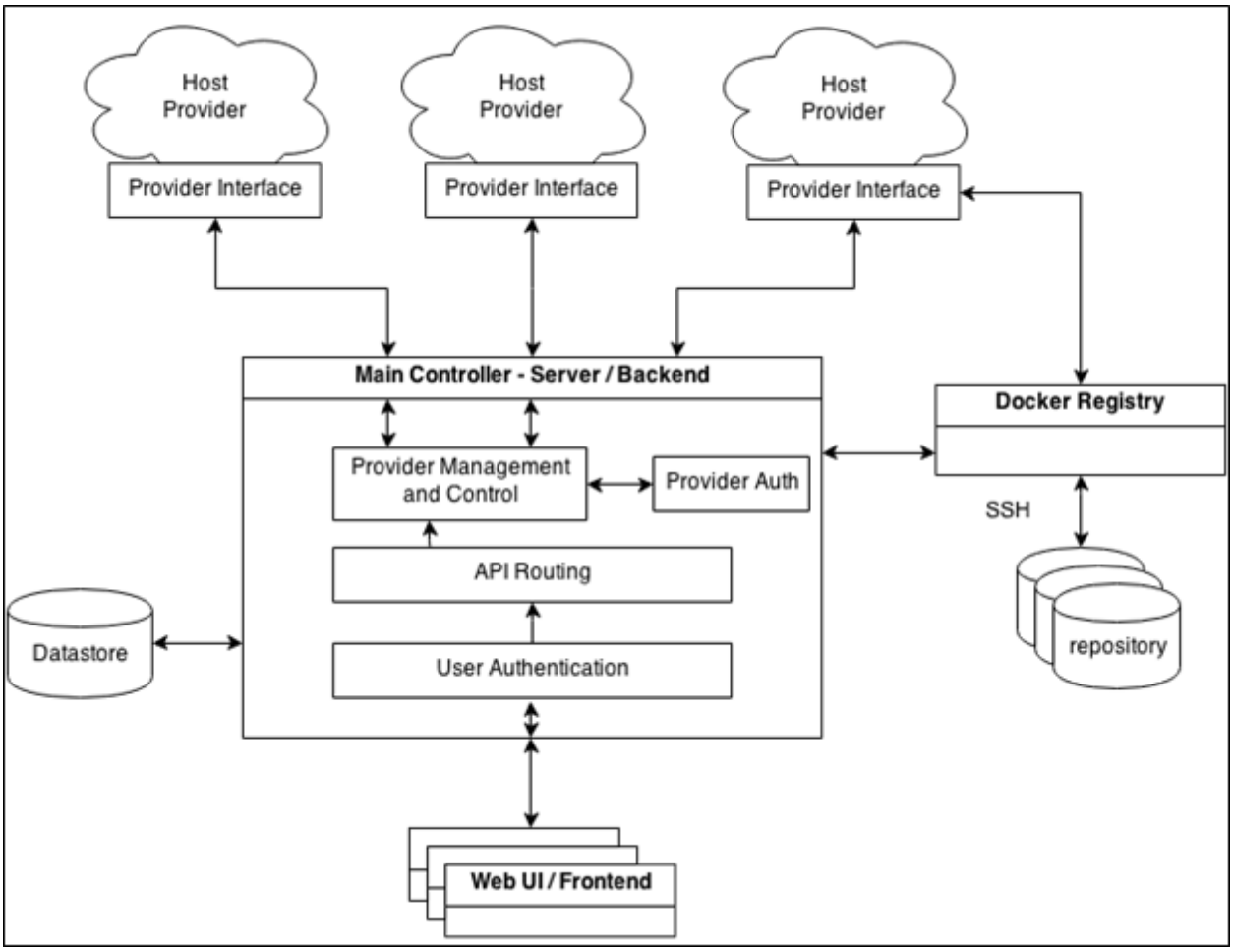

Figure 15 First version with Beacons

Database Paradigms

From verifying users to deploying applications, a substantial amount of Lighthouse's operations rely on databases. For this reason, we developed a very extensive databases package which abstracted things like query generation and result extraction into one concise interface. This helped to both improve codebase readability and reduce testing overhead. However, because the environment of Lighthouse was constantly being updated and this package was no exception.

At creation, all tables that Lighthouse used were simple key-value lookups where the key was always some string and the value was a JSON object. The benefit of this implementation was just its simplicity, there was only one column to query by, one column that could be updated, etc. This method worked very well for things like users which would only need to be extracted by their emails. However, when development reached the implementation of Beacons, we saw our first object that needed to be looked up by more than one value (Beacon address or instance address) and as a result, we needed a more sophisticated architecture.

The second iteration of the databases package saw the addition of schemas. Here an object would be added to the database by extracting each of its fields as a column. This method was incredibly powerful as we could store and lookup any of our objects by any of its values. The tables below shows how the actual storage inside the database changed using our table of Docker instances as an example.

Key	**Value**

"127.0.0.1/v1.12"	{
	"InstanceAddress" : "127.0.0.1/v1.12",
	"BeaconAddress" : "127.0.0.1:5001",
	"Name" : "myInstance",
	"CanAccessDocker" : true
	}

Table 2 Key-Value Storage

InstanceAddress	**BeaconAddress**	**Name**	**CanAccessDocker**
"127.0.0.1/v1.12"	"127.0.0.1:5001"	"myInstance"	true

Table 3 Schema Storage

Obviously this method is clearer and has a much smaller storage overhead. Its implementation is substantially more complicated, but the benefits far outweigh this complexity and through unit testing we are confident that the implementation is correct.

The final paradigm shift in databases was due to a desire to add support for new databases. Currently Lighthouse only supports Postgres databases, but this is far less strict now than at it was at conception. At some point, the digital rubber must meet the road and real queries and execs must be generated. Originally this was done inside the databases package itself and the package created strings specific to the Postgres syntax. If the user's database did not support this syntax then Lighthouse was simply not going to work with it. Of course this isn't a behavior most users would appreciate, we needed a way to

be able to generate database specific strings. This led to the creation of what we call Compilers. These are drivers for a specific database that are capable of generating strings for their particular database. With this method, the user need only specify the Compiler to use and the databases package will load it and operate normally. This was a much more modular design than we had originally created.

Beacon Implementation

As already mentioned, Beacons we not originally part of our end-to-end pipeline. Once they were however, there was some discussion about what their actual duty would be. Because the ancestor of Beacon, the provider driver had been developed inside Lighthouse, it was partially tied to some of that functionality. Our decision was how much of that functionality would migrate with Beacons and how much would remain in Lighthouse. One proposal was to have "heavy" Beacons and make Lighthouse "light." What this means is that each Beacon would be capable of most of the functionality that Lighthouse currently provides and Lighthouse itself would mostly handle routing. This method did help us abstract some functionality to simplify our implementations, but was not very feasible as it would have required that each Beacon have access to its own database which was far too much overhead to have to be replicated across the many cloud providers the user may have. Instead our implementation uses "light" Beacons which provide routing for a "heavy" Lighthouse, essentially the opposite of this proposal.

Application Management

The concept of Applications is not implemented by Docker and as a result their design was one of the most complicated aspects of the project. The initial concept was intercepting Container management requests that passed through Lighthouse and to group them by name into Applications. With this method, the frontend system would be largely in charge of performing the batch functionality required of Applications. At a high level, this works, but when the development process is considered there are issues, the most obvious of which being that Applications are only loosely connected as Containers that share a name. Consider developers who use Docker and create many containers for their own local tests. It is very possible that these developers would be giving their containers names and we would very likely see Applications being created that weren't true Applications because of name collisions. When we realized this, the Applications infrastructure was modified to the stand-alone architecture that it is now.

9. Appendix III: Other Considerations

The development process over the course of our project certainly proved to be challenging. Early design choices forced us into certain conceptual models that played a large role in later work, and learning new frameworks and languages in the process only added to the complexity of the work involved.

Single Page Application Architecture

As outlined in the Technical Issues section above, the single-page application architecture with Harbor posed some interesting design challenges with regards to session management. While we were able to build out a design around the architecture, hindsight suggests a different approach should have been taken to ensure session management was central to the initial design process, considering the multi-user and collaborative nature of Lighthouse.

With that said, there have been numerous benefits to our decision to use a single-page architecture. Client-side template rendering is extremely responsive and increases the perception of speed as the user navigates through the application. Strict client-side data management means that other pages can be navigated to without transient data such as streaming updates having to persist in the slower browser cache or on the server. Wrapping the entire front-end into a unified framework also helped to ease the development process, allowing us to effectively treat Harbor as a separate project in and of itself, by placing it in a separate version control repository. This means that Lighthouse and Harbor can have independent release cycles and two development branches can be tested simultaneously with minimal friction.

AngularJS and Go

Coming into the initial development effort, not everyone on our team had experience with the entire technology stack we had settled upon. While this is certainly a common trend in professional development teams (especially with new hires), it's important to take these factors into consideration when ramping up the development momentum. The relative immaturity of AngularJS and Go meant certain issues were met too late in the development process and reverting the choice to use the framework and/or language was not an option. Fortunately, the online communities surrounding these technologies are quite large and provided us with a platform and learning and troubleshooting not many environments have.

Requirements Gathering

One of the major takeaways from our development process was considering the importance of the initial requirements gathering process. Complexity in software is generally considered a rule, not an exception, with the end goal being to minimize both domain modeling complexity and accidental complexity in the feature set.

Initial meetings with our client, Workiva, were beneficial in providing us with a small set of initial features to begin developing, but, ultimately, the direction of the project was left to our discretion. While the freedom to develop an open-ended project has its benefits, there is a certain amount of benefit to restriction as well. Small, isolated software components allow for modular integration into larger systems and tend to be more stable pieces of software. The work we've done on Lighthouse has only solidified the understanding of a need for a rigid and comprehensive requirements gathering process.

Future Work

As the community surrounding Docker and isolated, container-based deployments grows, so will the use cases for Lighthouse. Even without considering the future changes to distributed application development, the feature set of Lighthouse can certainly be increased in the meantime. Two major features that come to mind are network monitoring for a specific container and resource monitoring for a Docker instance as whole.

Lighthouse and the related components (Harbor and Beacon) are licensed as open source under the Apache2 license, allowing developers to contribute and expand on the project in the future.

10. Appendix IV Table of Tables

Table 1 Common terms and their definitions 4
Table 2 Key-Value Storage 24
Table 3 Schema Storage 24

11. Appendix V Table of Images

Figure 1 System Block Diagram 8
Figure 2 Harbor block diagram 9
Figure 3 An example of Harbor's UI 9
Figure 4 Lighthouse block diagram 10
Figure 5 Beacon block diagram 10
Figure 6 Streaming UI in Harbor 11
Figure 7 Streaming concept diagram 12
Figure 8 Stream Processing code example 13
Figure 9 Deployment and Applicaiton database design 14
Figure 10 Streaming response, general form 15
Figure 11 Update Stream example 16
Figure 12 Automated Testing checklist 18
Figure 13 Loggin in 20
Figure 14 Connecting to a Beacon 21
Figure 15 First version with Beacons 23