University of Kent

CO600 - Tech Report

Project Apiary

Author:
Bradley Jones
Jack Fletcher
John Davidge
Sam Betts

Supervisor:
Ian Utting

April 3, 2014

CO600 - Technical Report - Project Apiary

Bradley Jones
University of Kent
School of Computing
bj59@kent.ac.uk

Jack Fletcher
University of Kent
School of Computing
jpf4@kent.ac.uk

John Davidge
University of Kent
School of Computing
jd389@kent.ac.uk

Sam Betts
University of Kent
School of Computing
sab50@kent.ac.uk

***Abstract*—In this paper we will discuss the motivations for and the development of Project Apiary - an end-to-end solution for real-time monitoring and analysis of log data in distributed computer systems. In the development of Project Apiary we explore the many problems associated with the gathering, storage and analysis of schema-less data, and seek to define user-friendly approaches for processing, and visualising this data. We also explore methods allowing the system to prompt the user to act on important events as they happen, rather than discovering them in an after-the-fact analysis.**

I. Introduction

Data centres produce vast quantities of data in the form of application and machine logs. The larger the data centre and the more complex its applications, the larger the volume of data produced. With the industry moving towards cloud computing, data centres are becoming ever larger and being used for a wider variety of applications, often by thousands or even tens of thousands of users simultaneously. The logs produced by services running on the machines which comprise the data centre contain a wide range of system[1] and application-specific data which can be analysed to gain valuable insight into the operation of the data centre. However, this data is usually unstructured and unorganized, making broad meaningful analysis a non-trivial task. When combined with the sheer volume of data being produced by ever larger data centers, the gathering and analysis of this data becomes an ever more difficult task.

Solutions already exist to combat this problem (as detailed in the background section below), but many of these are closed-source and expensive, and in our opinion those that are open-source do not currently tackle every aspect of the problem in a complete end-to-end solution. Given our background in open-source software we were motivated to explore the development of an open-source project which covered every component necessary for a user to gain meaningful insight into the workings of their data centre.

II. Project Background

The four of us have been working together as a team since the beginning of our Year In Industry at Cisco Systems in San Jose, California. During our time there we gained valuable experience of working with both large and small distributed computer systems - mainly focused in the world of open-source cloud computing. In the course of our work we were exposed to various tools for the monitoring of these distributed systems, but in all cases found them wanting in some respect. Of the tools with which we came into contact we believe these to be the most important:

A. Splunk

Splunk[1] is probably the closest to an industry standard when it comes to analysis of distributed systems. It is a large, enterprise-grade product capable of solving most of the problems we set out to solve with Apiary. The problem with Splunk is that it is expensive to use and is largely closed-source. This does not fit with our philosophy on software transparency and is prohibitive for those who cannot afford to pay.

B. Elasticsearch ELK

The Elasticsearch ELK[2] stack (Elasticsearch, Logstash, and Kibana) is conceptually very similar to Apiary. Its open-source and it consists of many components working together to solve separate parts of the overall problem. A major difference between Apiary and ELK is in the way that the components communicate with each other. ELK uses a REST[2] API[3] to communicate over HTTP, whereas Apiary makes use of a distributed message bus. We believe that this solution is better for scalability and security.

III. Aims

The aim of Apiary was to produce an open-source platform for the harvesting, storage, processing, and visualisation of machine data (logs etc) from large scale distributed systems, and expose this using a real time, interactive web front end. In order to ensure that our project was successful, we set out a series of clear defined goals before starting the project.

- Scalability - During our research we found that typical log rates for data centre applications could run into terabytes per hour. This meant that we would need to be able to handle more data than a single appliance could store, and still maintain an acceptable level of service. For this reason building a system

[1] System Data might include sources like CPU load, RAM usage and network traffic.

[2] Representational state transfer
[3] Application programming interface

that was easily horizontally scaled was of paramount importance.
- Real Time - We decided quite early on that we would be able to provide a richer, more interactive service by building a system that operated on a message based, real time system. This would allow us to push events from the data center all the way to the browser without implementing costly polling techniques.
- Simple Configuration - A problem that we identified with a lot of open-source software stacks is that they are often very difficult to configure, and come with hundreds of configuration options. This can make DevOps[4] very difficult, and just finding the optimal set of configurations can be a task unto itself. For this reason we wanted to keep configuration options to a minimum, and automatically detect configuration for as much of the stack as we possibly could.
- Alert System - One component that we noticed most of the existing solutions don't have, is any kind of event alert system. The user cannot be expected to be monitoring the system at all times, so we decided that we should be able to push triggered alerts directly to the users mobile device.
- User Friendly UI - Another thing we noticed about a lot of the existing solutions out there, particularly those that were open-source, was that UI[5] was often treated as a second class citizen to functionality. We wanted to ensure that our UI was simple, easy to understand, and provided rich data visualisations that allowed the user to gain the most from their data.
- Powerful Query System - Our project has a wide range of applications, and in order to support them all we needed to ensure that the query language could support a wide range of simple and complex operations, whilst remaining fast and real time.

Use cases for a system like this include live alerting of machine failures, or heavy network load; analyzing browsing and buying patterns in e-commerce websites from web server log data; and identifying network abuse by indexing data from firewalls and intrusion detection systems.

IV. Architecture and Design

In this section we will discuss the specific solutions that we built in order to achieve our goals, the motivations behind our design decisions, and the technical challenges we overcame. Our project ended up being divided into a series of independent components; Bee, our data agent; Hive, responsible for storage and processing of data; Queen, our web front end; and finally, our iOS application.

The key design decision was to build our system in a Process Oriented manner. This meant all components were broken down into small, scalable single task components that did not maintain state. All state was stored in a central, scalable database, and all communication was performed using a scalable RabbitMQ[3] messaging exchange. Designing the system in this way meant that we could leverage these technologies to provide a reliable, horizontally scalable, and real time system.

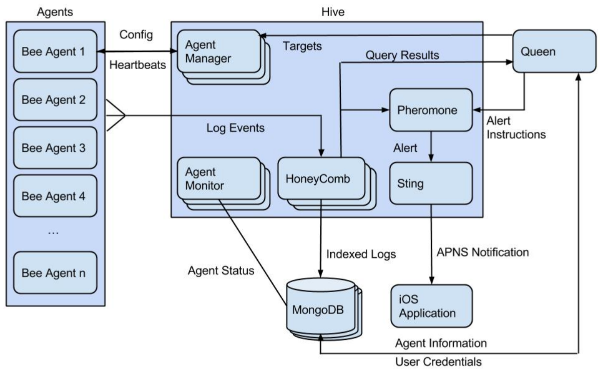

Figure 1. Design of Apiary Architecture

The reason for choosing RabbitMQ over other communication paradigms and technologies, such as REST, is that it provides asynchronous message passing, with worker queues and publish/subscribe models built in. Worker queues provide automatic distribution of messages between Apiary components of the same type, giving Apiary load balancing, and therefore configurationless scaling, for free. The Publish subscribe model allows Apiary components to subscribe to real time messages required for live events. It also maintains queues messages in the case of congestion or component failure, ensuring no data is lost.

For the database we opted for MongoDB[4]. We decided this was the best option for the project because of the necessity to have very flexible data structures. MongoDB stores collections of JSON objects instead of having rigid tables, allowing for metadata and optional fields to be added dynamically to objects; a feature that we specifically wanted for log entries. Due to the lack of relations in MongoDB it is also known to scale very well, and because MongoDB is JSON based like the rest of our stack, we would avoid data consistency issues as no conversion would need to be performed between components.

V. Bee

Bee is our data-harvesting agent, which is installed on all monitored machines and forwards log data into Hive. We designed Bee to be as lightweight and easy to set up as possible. This is because we wanted to ensure that Bees did not interfere with the normal operation of the machine it was monitoring, and for the initial installation task to be as trivial as possible.

For this reason Bee has only 1 configuration option - the IP of your central RabbitMQ cluster. After that all other configuration is either automatically discovered from the host machine (MAC address, hostname, etc), or received from Hive after an initial handshake over the RabbitMQ exchange. After this, Hive can request that this particular

[4]Development-Operations - a software development method that helps organizations rapidly produce and deploy applications.

[5]User Interface

instance of Bee listens to and forwards changes to any file on that particular machine.

Bee is based on NodeJS[5], as we needed it to be multi-platform, and the event-driven nature of its framework lends itself well to file monitoring, as well as responding to incoming RabbitMQ messages. It is designed using a Model-View-Controller (MVC) framework, with a series of actions for different commands from Hive.

VI. Hive

Our core, and most complex component Hives role is to manage data agents (Bees), store incoming log data, and provide complex querying against that data. In order to achieve scalability Hive is made up of 6 subcomponents, with a simple framework in place for extending this with extra components should the user require additional functionality. All components communicate over the RabbitMQ message exchange.

Given our background with OpenStack[6] (written in Python) and the abundance of bindings and support for many of the other technologies we wanted to employ, Python was the obvious language choice for Hive. Python also provided a clean multiprocessing library which was crucial for a number of the Hive subcomponents. We chose to implement a common MVC stack for all our components, along with a JSON protocol that would be shared across the whole stack. This allowed us to rapidly prototype and add new components to Hive as all the communication management and database access code had been broken out into a reusable library.

VII. Hive Subcomponents

A. Agentmanager and Agentmonitor

These were the first of the Hive subcomponents to be built, the purpose of Agentmanager and Agentmonitor are to configure and keep track of our Data Agents (Bee).

Agents are, as described earlier in the paper, installed with no configuration. Agentmanagers first job is to handshake new Agents entering the environment and provide them a unique identifier which is attached to data transmissions to the rest of Hive. After the initial bootstrapping of an Agent, Agentmanager then receives periodic heartbeats from the Agents. This heartbeat is used by AgentMonitor to detect the status of an Agent in the case of an unclean disconnection, error, network congestion, or other problem with the host machine.

The final task performed by these two components is to provide the Agents with a series of Data targets, in the case of the file watching Agent, this allows a user to remotely set the files that the Agent is monitoring. The API allows this to be done in bulk, with many targets, across many Agents, to prevent repetitiveness. Agent status responses (success, fail) are then collated and returned in a labeled list.

Maintaining the real time aspect of the whole application required the addition of events to Agentmanager and Agentmonitor. Whenever a significant change happens on either component; such as, a new agent handshakes into the environment, or an agent is flagged as dead, the component where this change happened publishes an event object onto a known message exchange, which other services - if they are listening - can use to perform any tasks they have which are relevant to a change in the Agent Manager without the need for polling. Subscription to this exchange is completely open, optional and will not alter the overall behavior of the program.

The reason for splitting this component into both Agentmanager and Agentmonitor, is because the tasks performed by each scale differently. Agentmanager needs to handle a constant stream of Agent heartbeats, which get more frequent, as more Agents are added. In comparison Agent Monitors task does not increase in complexity as fast, and because of this, it does not need to be scaled as aggressively. This configuration of split services provides the greatest flexibility for host utilisation at large scale, as you can scale very specific parts of the whole application.

B. Honeycomb

Honeycomb is our most complicated component as this is where all the storage and processing of collected data takes place. Honeycomb follows the standard hive component structure, and uses a common MVC stack for inbound requests. It uses a subscriber queue to digest the stream of data it receives as fast as possible without having to respond to each request, which would slow the process down.

Honeycomb uses our common model structure (see Common) to verify the validity of all received data before it is saved to the database. There is also an additional step in the Honeycomb save function to perform indexing for every data entry. After some research we settled on using PyLucene[8] as a python wrapper for Lucene[9] to handle indexing. Lucene is an open-source plain text analysis and search tool developed by the Apache Software Foundation. Whenever data is saved it is passed to Lucene for indexing so that we can utilize their powerful query syntax, without having to apply schema to the contents of the data.

As well as storing the data, Honeycomb also exposes an API for searching it. These searches can be one of two types; a one-off query, or a recurring query. One-off queries are simply a remote procedure call (RPC) containing the query. The response is written into a JSON object and passed back to the requesting process via the message bus. This is great for individual tasks, but in order to facilitate real time updates we needed to introduce recurring queries. Recurring queries require a little more setup as they need to be run concurrently alongside the main Honeycomb thread. Since Honeycomb has to remain stateless for scalability, it should not locally maintain any reference to these background threads, as this would mean that a second instance of Honeycomb wouldnt be able to interact with the new query.

In order to solve this issue we use the message bus again, spawning a wrapper process which provides an API to the thread in the background, and stores the key to communicate with that wrapper process in the database.

The recurring query thread isnt complex, simply running the lucene query then waiting a second before running it again. The thread also maintains a list of any previously discovered log IDs, and will only output if there has been a change, or if it has been asked to send the next set of results. Output for these background queries is pushed onto a uniquely named fanout exchange on the message bus. This allows multiple processeses to receive copies of the output from a single query, and also ensures that if no one is listening the output is simply thrown away instead of clogging up a queue on the message bus.

C. Pheromone

Using the recurring query API in Honeycomb, Pheromone places a layer of intelligence on top of the query to filter out specific results and trigger alerts based on user-defined conditions. Alerters are created through the Pheromone API. These alerters take trigger cases, and search parameters they need to test for an alert, and a custom message that is included when an alert is transmitted. Pheromone currently only has the one type of alerter - it will fire an alert if a query gets a certain number of matches during a given timescale. However it is not limited to just this one alert type, as the design patterns used for the alerters allows for easy addition of new ones in the future.

Pheromone uses python's multiprocessing library to start background tasks similar to how Honeycombs recurring query tasks operate. When a request comes in Pheromone starts a new alerter in the background, however unlike the Honeycomb workers there is no unique output exchange, as alerts are always pushed onto a known message bus routing key. This allows other services, for example Sting, to listen for them and perform all necessary tasks they have relative to the alert.

D. Sting

Pheromone alerts are a very powerful tool, but the user is not always going to be sitting in front of their computer when an alert is triggered. In fact, the very nature of user-defined alerts generally makes the times when they will occur rare and/or unpredictable, and so we must be able to inform the user of an alert as reliably as possible. By building a native iOS[10] app as described later in this paper, we are able to take advantage of the Apple Push Notification Service (APNS)[11]. APNS is a service made available by Apple to all iOS application developers for sending small text-based messages to iOS devices using your mobile application. In many ways this is similar to the standard Short Message Service (SMS), but with the key differences that the message is sent specifically to the application on the device, and more importantly - is free. Many third-party services exist for sending automatically triggered SMS messages (for example, Twilio[12]), but these services charge several pence per message, which could become very expensive in a large system with many users and alert conditions.

When Pheromone pushes an alert onto the message bus, Sting picks it up and looks up the user to which the alert is registered. For every iOS device registered to that user (multiple iPhones, iPads, etc.) Sting will generate an APNS push notification using the alert text specified by the user and the Device ID associated with the device. The messages are then sent using the APNS API - typically taking up to 5 seconds to arrive. This system ensures that a user will always receive potentially critical updates about the state of their system as and when they happen, giving them the time to act on them before it may be too late.

E. Common

Though not technically a subcomponent in itself - Common is the base layer underlying almost all of the hive subcomponents. Any code that was duplicated and was reusable was moved into Common to maintain the DRY[6] programming style. Common therefore became the home for our MVC stack, including the parent classes associated with writing controllers and routers. Common also includes drivers for generic database access, meaning that the underlying database technology could be changed fairly trivially if necessary. The current model uses PyMongo[7] to hook into MongoDB.

The Base program pulls all the these parts together into an application that runs but needs extending to provide any real functionality. It handles the loading of configuration files and makes sure they are accessible throughout the rest of the program. It also spawns the worker threads required to listen on the message bus. Both a subscriber queue and worker queue are listened to in order to distinguish between RPC (remote procedure call) messages which always require a response, and purely informative messages - which require action but no response. Base also sets up and handles all the logging throughout the program, ensuring that it gets written to an intuitively named file and in a common format that makes issues easier to find.

VIII. Queen

Our front-end component, also built on the NodeJS framework pulls all of the components of Hive together to provide a real-time, visually rich experience.

We used the Express[13] library for NodeJS to build an MVC framework for the UI to keep in line with our design style throughout the rest of the project. This has lead to a clean, modular codebase that has very little duplication of code. The most important decision designing Queen was to ensure that all data transferred to the front-end would happen in real-time. In order to enable this we took advantage of the socket.io[14] library to provide access to web-sockets that allow client(front-end) to server(Queen back end server) communication.

Like the rest of our components Queen uses RabbitMQ for inter-component communication. The flow for communication with other components is described here using a user performing a search as an example:

1) The search terms are passed from the UI as a JSON object back to Queens server via a web-socket.

[6]Dont Repeat Yourself

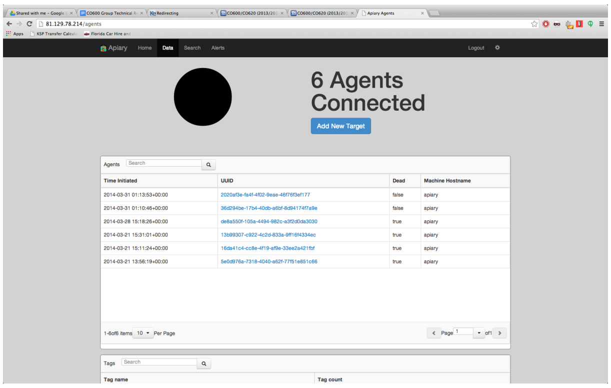

Figure 2. Capture of Queens Data page

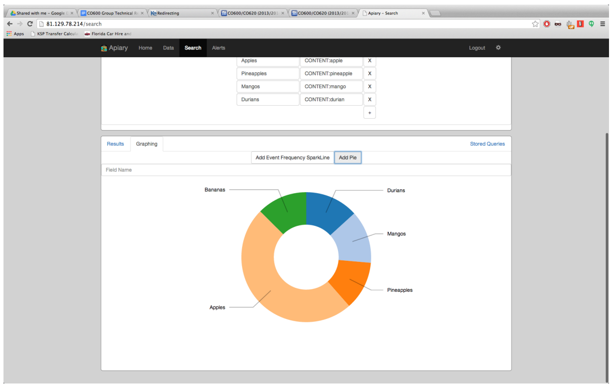

Figure 3. Capture of Queens Search page

2) Queen server then passes this data onto the appropriate Hive component via RabbitMQ, using either a remote procedure call or publish/subscribe model as appropriate.
3) The Hive component (in this case Honeycomb) will then return the data to Queen server.
4) The exact data that is required for the front-end is extracted and passed back up to the UI via a web-socket.

If we request data using a publish/subscribe model, then every time the Queen server receives data on the subscribe queue it can push that data back up to the front-end using a web-socket. This means that the UI always has quick, live access to the data making our UI responsive, and real time.

Queen enables multiple users to all be using the service at once, so that users can store and retrieve queries and alerts that the user wants to access next time they use the system. Also a list of devices associated to the user is stored for our Alerts system. As with other components MongoDB is used as our database for this information.

Just as we did with the Bee agents we wanted to keep Queen configuration as simple as we could. The configuration options as such are kept to just the IP address of the RabbitMQ server, and the IP address of the mongo server.

Rich visualisations were another key goal in designing Queen, due to our experience using the d3js[15] visualisation library we naturally decided to use this. It has enabled us to use visualisations such as sparkline graphs to map event rates in the system and pie charts to compare search results. From a human computer interaction (HCI) point of view this allows the user to process their data in a much richer format, allowing them to make valuable and meaningful analysis.

Key to being able to graph data, was being able to infer schema onto our logs and break queries down into fields. For this we came up with a system whereby a query could be broken into subqueries, and each of these subqueries could represent a member of a field. This allowed us to then take the results, partition them, and graph the various members against one another. The subqueries are made independently, and the results collated and returned to the browser. This is what allows us to really gain value from otherwise unorganised log data, as you can now start to make meaningful analysis.

Due to the quantity of data that can come back from search results it was important not only to visualise this in graph form but also allow users to view the actual log entries live in the browser. In order to do this we feed all data results into a datagrid from the FuelUX[16] library which provides paginated in-browser tables with column sorting.

IX. iOS Application

In the real world a user will not always be sitting in front of their computer when an important event occurs in the Apiary framework. However, it is likely that they will be carrying a modern smartphone, which will have a reasonably consistent WiFi or cellular data connection. Due to our GUI being entirely web-based a user could access everything they need to know using their smartphones web browser, but by creating a native application we can enhance the user experience and provide additional functionality. A small but useful function is the ability to save the IP address or URL of the particular Queen instance to hook into, as well as the users login credentials for that instance, saving a user the time of configuring the application every time they use it.

The greatest argument for a native application is the ability to exploit the powerful Apple Push Notification System. As described above in the Sting section, APNS can be used to send the user real-time updates on the status of their computer system at anytime, as long as they have a cellular/WiFi data connection. When a user logs into Apiary using the iOS application, their unique Apple Device ID is registered against their user account to be used by Sting for alert notifications.

X. Future Work

A. Future Agents

Very early on in our design we decided that our Agents system should be flexible enough to accommodate a variety of agent types, as not all schemaless data comes from text log files. We had the following Agents planned, but due to time constraints were not able to implement them, and felt that our time would be better spent focusing on processing techniques and UI.

First of all, given our OpenStack background, we would have liked to have built an Agent that was integrated with OpenStack Ceilometer[17], the monitoring and metrics component of OpenStack. This would have worked in much the same way, but rather than have the user define a file, they would need to define a Ceilometer Meter to start watching. After this event processing would have worked in exactly the same way.

Another consideration was to build an Agent, or extend Bee so that it could tap into popular logging frameworks. Many applications have now started to move away from traditional log files, and instead pump data into logging frameworks which can maintain slightly more structure about a log. Examples include LogBack[18], or Log4j[19].

Furthermore, other open-source projects could be extended to work with our system, such as FluentD[20], which has a plugin system that would allow us to easily build in Apiary support.

B. Hive - Timemachine and Intelligence

Two future Hive components we considered but were not within the time scale of the project were Timemachine and Intelligence.

Timemachine would be designed to pull data from the database according to a query, and then push results onto a message queue in the same order and format that they originally sent from the agents. A typical use case for this would be an application that requires a replicated stream of historical data for playing back events as they happened. This historical playback could be used to analyse system faults to determine what went wrong.

Intelligence is much more complex and has potentially many more use cases. The idea behind Intelligence is to provide an API for doing more advanced, deferred data processing, such as Map Reduce tasks with the log data stored in Honeycomb. Software like Hadoop[21] would allow for more in depth analysis than is currently possible with the Lucene implementation in Honeycomb, but would not work in real-time. The scope of Intelligence could potentially be extended to include machine learning which we hope would compliment Pheromone with dynamically generated alerts.

XI. Conclusion

To conclude, we set out to design and build an end-to-end solution for real-time monitoring and analysis of log data, this we feel we have achieved successfully. We had several goals which sculpted our design and thought processes for the project, and we believe we have met them with the solution we have built. At the beginning of the project we separated our goals into the six major categories that we felt were most important for a system of this type. Addressing them individually, we can see how our design decisions helped us achieve each one.

A. Scalability

Splitting the Hive into several components communicating via RabbitMQ has given us industry-proven levels of scalability at zero cost and with very little implementation overhead. The use of MongoDB has also given us a database backend which has been tested to destruction in all corners of the tech industry[22].

B. Real Time

RabbitMQ and web sockets have been used to create truly real time communication between the many components of Apiary and the web front end. Expensive polling loops are not required to keep the user up to date with the state of their system.

C. Simple Configuration

Designing components such as the Bee to do the majority of its configuration automatically based on data passed from the central Hive has made deploying Apiary across a distributed system a much more straightforward task than many open-source projects. The inclusion of installation scripts has also contributed to this goal significantly.

D. Alert System

Exploiting the real time nature of our message bus infrastructure and the recurring query API exposed by Honeycomb has allowed us to create the Pheromone and Sting components. These components work together with the iOS application to provide the user with real time feedback regardless of their location.

E. Powerful Query System

By exploiting the advanced text search language that Lucene provides and combining this with fielding in Queen we have built a system that allows the user to filter, and infer schema onto otherwise unorganised data. Lucene's query language is expansive, and we ourselves are yet to fully explore all of the features it provides. Data can be searched and fielded on a wide range of parameters, from content, tag, hostname, log time, and many more.

F. User Friendly UI

Leveraging the power of our query system, our user interface allows the use to create rich data visualisations from query results, allowing them to make valuable and meaningful analysis. Our interface is web based, real-time, and easy to use.

Following the points above, we feel that we have have built a valuable tool, that allows the user to gain real value from data that would otherwise be locked away, and difficult to process. Compared to other projects in the market today we sit at a relatively early stage of development, simply due to timescale, however we feel that our implementation brings a number of novel design decisions that give us an edge in certain applications. No other project employs our system of using message based communication across the whole stack, which brings a myriad of benefits. It allows us to do real time, easy reliable scaling, and deal with heavy loads without scaling. Our fielding system is also unique, as other systems have opted for complex, expensive, log analysis algorithms. Our system allows ultimate flexibility, and keeps overheads down.

In future endeavours we might take slightly different approaches to some tasks based on a series of issues that we hit during the development of Apiary. We would probably try to avoid using NodeJS in certain parts of our stack. This is simply due to NodeJSs infancy, we found in a number of cases that it was difficult to debug, due to the lack of tools, lack of documentation, and lack of predictability in the JavaScript specification. Also, we would spend a little more time planning the communication protocols, as we had to make a number of changes, at one point switching from XML to JSON. This took a significant amount time to refactor, and could have been avoided with more consideration.

Finally, given more more resources, we would have like to have done some extremely large scale testing. As it stands we were only able to distribute instances across the machines that we personally owned, which allowed to to confirm that scaling worked, but did not demonstrate how far it could be pushed, in our testing each component was instanced no more than 3 or 4 times.

Acknowledgment

The authors would like to thank the following people for their assistance during the project:

Ian Utting - http://www.cs.kent.ac.uk/people/staff/iau
Debojyoti dutta - Cisco Systems, San Jose, CA
Lew Tucker - Cisco Systems, San Jose, CA

References

[1] Splunk - http://www.splunk.com

[2] Elasticsearch ELK - http://www.elasticsearch.org

[3] RabbitMQ - https://www.rabbitmq.com

[4] MongoDB - https://www.mongodb.org

[5] NodeJS - http://nodejs.org

[6] OpenStack - https://www.openstack.org

[7] PyMongo - http://api.mongodb.org/python/2.7rc0/

[8] PyLucene - http://lucene.apache.org/pylucene/

[9] Lucene - http://lucene.apache.org

[10] iOS - https://developer.apple.com/devcenter/ios/index.action

[11] Apple Push Notification Service - https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html

[12] Twilio - https://www.twilio.com

[13] Express - http://expressjs.com

[14] Socket.io - http://socket.io

[15] d3js - http://d3js.org

[16] FuelUX - http://exacttarget.github.io/fuelux

[17] Ceilometer - https://wiki.openstack.org/wiki/Ceilometer

[18] Logback - http://logback.qos.ch

[19] Log4j - http://logging.apache.org/log4j/2.x/

[20] FluentD - http://fluentd.org

[21] Hadoop - http://hadoop.apache.org

[22] Datastax Corporation (2013), Benchmarking Top NoSQL Databases, http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf