Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

HubSpot Software Engineering Internship

Stefano Lupo
14334933

August 6, 2018

An Internship Report submitted in partial fulfillment
of the requirements for the degree of
MAI (Computer Engineering)

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism ‘Ready Steady Write’, located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: ______________________ Date: ______________________

Acknowledgements

I would like to thank everyone that I worked with on the Email-Sending-Infrastructure team at HubSpot, especially the team’s technical lead Michael O’Brien who provided answers to the many questions that I had throughout the course of my internship.

I would also like to thank my academic liaison Prof. Owen Conlan for the guidance he has given in regards to the academic side of the internship.

Contents

List of Figures

List of Tables

List of Code Listings

Nomenclature

S_a	The set of available IP addresses
S_e	The set of existing IP addresses
S_c	The set of chosen IP addresses
S_f	The final set of IP addresses
n	The number of IP addresses to choose
t	The total number of IP addresses to return
n_{hb}	The number of host bits specified by a CIDR IP addresses
n_{ab}	The number of address bits specified by a CIDR IP addresses
n_{hosts}	The number of hosts implied by a CIDR IP address
’	Appended to variables to imply the depth of recursion
b	The bucket size used
b^*	The number of IPs required to fill a particular bucket

1 Introduction

1.1 What is HubSpot?

"HubSpot is inbound marketing and sales software that helps companies attract visitors, convert leads, and close customers." (1)

Over the past few decades, there has been a monumental shift in how consumers think about, shop for and purchase products. Consumers are now far more skeptical towards traditional marketing campaigns, for example via TV or print advertisements. Thanks to the near endless amount of information available on the internet, consumers are much more informed about the products or services they are purchasing and place their trust in credible third party reviews. In response to the modern consumer, a modern marketing strategy must be put in place.

HubSpot is built with one core belief in mind - *inbound marketing*. Inbound marketing is a strategy in which potential customers are *attracted* to a website (or other establishment) naturally, prior to attempting to convert them into buying customers. The key concept here is that customers must be able to obtain something of value prior to making a purchase. This attracts potential customers to the website, provides them with something they need and gives them an insight into the quality of the product or service on offer. For example, a website of a company selling musical instruments may write a blog post detailing what one should look for when purchasing their first guitar. For a modern consumer looking to purchase a guitar for the first time, it is extremely likely that they will look online for some information and a comprehensive guide on what to look for when purchasing a guitar would be the perfect resource. While reading this article, the consumer could be provided with attractive CTAs (call-to-actions) which, for example, bring them to a list of suggested guitars for beginners. Thus a potential customer has been naturally *attracted* to the companies online store, after acquiring some knowledge on exactly what they are looking for. This is a stark contrast to the traditional approach to marketing in which the potential customer is simply shown an add for a guitar. This overall concept of inbound marketing is succinctly summarized by the inbound marketing ethos - *don't interrupt buyers, attract them* (1).

1.2 The HubSpot Platform

HubSpot offers a comprehensive suite of products to help businesses employ an inbound marketing strategy, all of which are contained within a single platform.

1.2.1 Customer Relationship Management (CRM)

At the core of the HubSpot platform, is the HubSpot CRM. Simply put, a CRM is a single data source consisting of **all** of the data relating to a given customer. For example, it contains simple properties such as their name, age and address, but can also contain (in HubSpot's CRM) more complex information such as their last purchase, the last time they were contacted by a sales rep etc. Almost all of HubSpot's products aim to build *personalized* experiences for their customers and the data required to drive these personalized experiences often comes from the CRM. As a CRM is such an integral part of a modern business' daily operations, HubSpot offers their CRM for free.

1.2.2 Sites

HubSpot also provides a comprehensive website builder allowing users to build custom websites using a drag-and-drop style editor. This hugely simplifies the process of building a website, requiring zero knowledge of web development. These websites can be fully database driven, allowing the experience to be tailored to individual users. This product also includes a host of tools such as forms, CTAs, blogging and many others. All of the pages will be fully responsive and render perfectly on devices of any screen size. These sites can be hosted with HubSpot, further simplifying the website setup process.

1.2.3 Email Marketing

One of HubSpot's most popular tools is the marketing email product. Marketing emails can be built using an intuitive drag-and-drop editor and can be personalized on a per-subscriber basis, backed by HubSpot's CRM. Emails can be sent to arbitrarily complex subsets of subscribed customers (for example, customers who previously made a purchase, but haven't done so in 6 months). All marketing email that is sent can gather detailed analytics on how subscribers interact with the email, allowing HubSpot's customers to gain a deeper understanding of their customers. Similarly, email content can be optimized using A/B testing backed by machine learned models. A/B testing creates multiple versions of the same email, for example with different fonts and colours. A test send on a small sample of the target subscribers is performed, half of the sample receiving version A and half of the sample receiving version B. Key analytics such as open-rates and click rates are examined in order to attempt to ascertain which email will perform better on average. The better version of the email can then be sent to the remaining subscribers, maximizing the effectiveness of the marketing emails sent.

1.2.4 Service Hub

Modern customer's demand a high level of support should they encounter any issues when using a product or service. Service Hub is a modern tool to enable support staff to provide the high level of customer support that is necessitated today. This product provides a set of tools to enable customers to get in touch with the *who* they need to, *when* they need to. It offers a plug and play messaging system for websites, allowing customers to chat with support staff in real time. It also offers a single tool for support staff which consolidates all customer conversations into one place (e.g. emails, support tickets, comments and even some social media support), allowing support staff to easily stay on top of their work.

1.3 The Email Sending Infrastructure Team

The Email Sending Infrastructure team is one six teams working on HubSpot's email products. These teams vary in purpose from creating the drag-and-drop email builder, gathering analytics on emails sent, ensuring that no spam email is sent through HubSpot and finally, the Email Sending Infrastructure team's purpose, sending the actual final email. Considering the fact that sending email is quite an old and well established mechanism, at first glance, one might wonder why an entire team is needed solely for maintaining the infrastructure for sending the emails. However as HubSpot continues to grow, the number of emails that need to be sent continues to grow. Currently, HubSpot sends upwards of 60 - 70 million marketing emails per day on behalf of their customers. At this scale, sending email becomes a non-trivial task that must be carefully managed, requiring a high emphasis on efficiency at each step.

There are some major challenges with sending email at this scale, primarily stemming from the fact that SMTP, the Secure Mail Transfer Protocol (see section 3.2.2), was developed over 30 years ago. This protocol was designed with (relative to the modern day) very small packet sizes in mind. As such, it is a very *chatty* protocol, requiring several round trips between client and server, which can cause the SMTP conversation required to send even a small email take time on the order of hundreds of milliseconds, or even seconds. Thus, sending millions of emails a day, each taking a (relatively) substantial amount of time, highlights the difficulties faced by the Email Sending Infrastructure team. Another challenge of sending email is the variation in recipient servers. Anyone can very easily setup an email server on any machine. As one might expect, attempting to send 100 emails to a small desktop machine on a home network is very different to sending them a high performance mail server in one of Google's data centers. Thus, the Email Sending Infrastructure team's systems must be able to handle sending to both types of servers efficiently.

In today's computing world of multicore processors and high concurrency, the solution to the problem is to asynchronously send the emails, allowing the system to continue to perform useful work while waiting on responses from servers. However as with all concurrent solutions, this adds an extra layer of complexity to the system. System resources, especially the heap, where all of the in-flight emails will be stored, must be carefully managed. Another common issue faced is that recipient servers often apply heavy rate limiting to the IPs sending the emails, or to the SMTP domain from which the email originates. This is done to ensure the recipient servers are not overwhelmed by large spikes of emails from a single domain (a common occurrence when spammers compromise legitimate email accounts). Thus asynchronously sending emails requires distributed rate limiting through distributed semaphores, all of which must be carefully balanced in order to maximize throughput, while maintaining a good relationship with the recipient servers.

The main project owned by the Email Sending Infrastructure team is an internal Mail Transfer Agent (MTA). MTAs are systems responsible for sending email on behalf of clients. This internal MTA began development in 2016 and has been an integral part of HubSpot's email products ever since. The choice to develop an internal MTA may seem odd at first, given that email has been around for decades and many *Mail Transfer Agents as a Service (MTAaaS)* already exist. There were several key reasons for developing an internal MTA. The first was to learn as much information about the email sending process as possible and to relay this information back to HubSpot email marketing customers, allowing them to create the best email marketing campaigns they can. Another key reason is that HubSpot is responsible for sending these marketing emails on behalf of their customers and cannot simply blame a third party should the MTAaaS perform poorly. Finally it also provides flexibility and allows HubSpot's email products to grow and change overtime as the business dictates.

The Email Sending Infrastructure team is a relatively new team whose main focus is taking ownership of all of the code used in HubSpot for the email sending process (including the internal MTA previously discussed). The team saw 25x growth in the number of emails sent through their MTA in 2017 and managed to become **more** reliable while scaling, enabling HubSpot's email products to grow without constraints.

2 Team Management and Team Process

As with any type of team, it is critical that the team is managed appropriately. However, *appropriately* doesn't imply that the team should be managed *strictly*. The Email Sending Infrastructure team are a very self motivated team and thus required no micro management. The day to day managerial tasks fall to the tech-lead (TL). The TL is an experienced engineer who is officially responsible for the success of the team. The TL of Email Sending Infrastructure frequently liaises with other teams and senior staff in order to maintain alignment between the goals of the company as a whole and the goals of the team. The Email Sending Infrastructure team also recently received a project manager (PM). This role is responsible for working with customers and ensuring the team is prioritizing what will have the most impact on the customers and business as a whole. In the case of the Email Sending Infrastructure team, neither the PM nor the TL were concerned with the day-to-day operations of a given engineer on the team. Each of the team members is sufficiently interested and invested in the success of the team that trust is placed in each of the team members to pull their weight. This management strategy aligns directly with HubSpot's ethos of *"use good judgment"*. HubSpot aims to hire the type of employees that can be entirely trusted and believes in letting their employees work in whatever way they are most comfortable, which is typically the way in which they are the most productive.

However, no matter the management style, a solid *team process* is paramount for the success of the team. This involves having well defined methods for managing tasks and monitoring the productivity, morale and competency of the team as a whole. Some of the critical parts of the team process of the Email Sending Infrastructure team are outlined in the following sections.

2.1 GitHub Enterprise and Issue Tracking

HubSpot's entire engineering department runs on GitHub Enterprise (2). This is an online platform which provides code hosting, comprehensive version control through

Git (3), tracking issues and managing pull requests.

A common method of keeping track of what needs to be done inside of a software development team is creating *issues*. An issue is typically a paragraph or two long, outlining a task that the team should work towards to create, fix or improve upon something. A good issue will have context as to **why** the issue is important, **who** the issue impacts (negatively or positively) and the overall impact of the issue (for example is it critical or should other more pressing issues be prioritized).

Internally, HubSpot is a very open company and a lot of teams work very closely together. As such it is perfectly acceptable (and encouraged) that someone from outside of the team create an issue to inform the team of something they think the team should know. Sometimes, the request contained in the issue is actually already in place - for example a request for a feature which already exists. An engineer on the team will typically respond to the issue and can close the issue once a desirable outcome has been reached.

Critically, issues provided a comprehensive and searchable history of what the team has been working on. Issues often result in a piece of code being (re)written and a pull request submitted. GitHub offers a mechanism for cross referencing issues and pull requests so the issue which incentivized the pull request (and the pull request which resolves the issue) are forever searchable in the Git history of the project. This is extremely useful for engineers who wish to gain some broader context on a piece of the system, allowing them to read the context contained in the pull request and associated issue.

2.2 Task Management

In order to maximize productivity and maintain a healthy workload, a system must be put in place which dictates how team members are assigned units of work. All units of work were defined in GitHub issues (see section 2.1). The method used by the Email Sending Infrastructure team, was to split tasks up into the following five stages:

Backlog	This contains tasks which are of low priority and can be deferred until a later date
Design	This contains tasks which require a considerable amount of planning before being undertaken. Tasks in this stage often result in discussions among the team about how best to tackle the task.
Ready	Tasks which have enough detail and context, that they are ready to be

	undertaken. More complex tasks that were once in the design category land here once all of the corresponding details have been decided upon. These are tasks which engineers on the team should choose as a next task upon completion of a task.
In-Progress	This contains tasks which are currently being worked on (usually) by a single engineer.
Completed	Tasks which have been completed this week.

All of the tasks are managed through an online platform called Waffle (4) which presents a *"Waffle Board"* which displays the tasks in each of the above stages. The team concludes each week with a meeting in which the board is inspected and tasks are moved to new stages as appropriate. This provides a great mechanism for monitoring the team's productivity and ensuring that the team is focused on the most important tasks.

2.3 Communication

The main method for communication among employees in HubSpot is through Slack (5). Slack is a messaging platform which allows users to create and join channels. At HubSpot, most teams have a well known channel in which all members of the team are present in. People with questions or comments can simply join the Slack channel and send a message. Slack channels and messages are well indexed allowing users to search by keyword to find what they are looking for. This is extremely useful for looking for answers to questions that have likely been answered before, or to find the appropriate Slack channel to ask a question in.

HubSpot also has a number of channels in which engineers can ask for help. For example there is a channel specifically for Java questions where Java experts and beginners can help one and other out. These Slack channels are also very useful to follow in order to pick up random pieces of information. Often times, questions that are asked stimulate interesting conversations, allowing engineers to gain a deeper insight into what was initially asked.

2.4 Proactive Operations Reviews

Operations Reviews (Ops Reviews for short) are meetings in which the health of the system over the past time period is examined. For the Email Sending Infrastructure team, these are conducted weekly. HubSpot has a PagerDuty service which allows alerts to be set up based on the system's current health. If one of these alerts triggers, the on-call engineer on the team receives a notification informing them of the alert. These are critical alerts that must be looked into immediately and rectified. The main purpose of the weekly ops review is to examine the alerts that triggered during the previous week. Any alerts who's root cause remains unknown at this point gets a GitHub issue created and is assigned to an engineer to investigate. Sometimes alerts can trigger prematurely, for example if a threshold value is too low and should be safely increased. In this case, an issue is created to tweak the alert to be less strict. The point of the alerts is to inform an engineer of critical problems with the system. Thus, the rate of false-positives should be kept to a minimum, reducing the overall number of alerts that trigger, while ensuring that alerts that do trigger are in fact critical to the health of the system.

This has been a key part of the Email Sending Infrastructure team's success. There is no brushing issues with the system under the rug. Any critical issues that arise are investigated and reported on. This hugely increases the stability of the system over time.

2.5 Code Walkthroughs

As discussed in section 1.3, the Email Sending Infrastructure team is a relatively new team and has inherited a large portion of the code base they own. As such, engineers on the team have likely not yet encountered certain parts of the system. Each week, an engineer from the team hosts a code walkthrough. The engineer chooses a part of the system or a piece of code that they have never seen before and studies what it's purpose is and how it is implemented. The code walkthrough is a 30 minute meeting in which the engineer presents what they have learned about the part of the system they studied to the entire team. This allows other team members to gain an insight into a part of the system they have likely never seen before, without the need to spend time digging through it themselves. The engineer hosting the code walkthrough rotates in a round-robin fashion, allowing the team as a whole to gain more of an insight into a system they inherited as the weeks go by.

2.6 Critical Situation Post Mortem

Depending on the team and product, HubSpot managers and tech leads have a set of parameters for what constitutes a critical situation (crit sit). For the Email Sending Infrastructure team, a system issue that effects a significant portion of customers' email sending capabilities constitutes a crit sit. Should a crit sit arise, once the system is stable again, there is a well defined, formal procedure that must be followed. This procedure involves documenting the events that lead to the crit sit, the root cause of the crit sit, the remedial action and measures that should be put in place to prevent the situation from arising in the future. The team typically also has a meeting with a senior manager of the company in which the above procedure is run through.

Critically, there is never any blame assigned during a critical situation. The key reasoning behind the crit sit procedure is to identify the issue, learn from the root cause and put preventative measures in place. The crit sit procedure is a valuable tool used in HubSpot and forces teams to evaluate problems in their systems and to improve the system's stability over time.

2.7 Daily Standups

The Email Sending Infrastructure team also has a daily standup meeting. This meeting is extremely brief but helps bring the team together at least once a day. This allows the engineers to discuss what they are currently working on, allowing team members to offer advice and have some context when the inevitable pull request is submitted. Daily standups have proved very useful as often times, more experienced team members will help identify a potential issue to watch out for when tackling a certain problem. This can help less experienced engineers avoid spending hours on an implementation only to realize a certain complexity was overlooked.

3 Software Development at HubSpot

Software development has been a core part of HubSpot since day one. As such, the overall development process and methodology has undergone several iterations. Critically, at each iteration, the development process is reevaluated and the experience gained over the past number of years is capitalized upon. This has lead to a streamlined and simple development process which is easy to learn, allowing both new hires and temporary interns to get up to speed quickly. HubSpot also has several *platform* teams solely dedicated to providing easy to use integrations of powerful tools which can be cumbersome to configure on a per project basis. These teams contribute to what's known as a the overall *platform as a service (PaaS)*. Product development teams can then use this platform to get access to these powerful tools, without needing to spend time on configuration.

As all of HubSpot is built on the same technology stack and all projects adhere to a common project structure, any developer can drop into any project owned by another team and immediately know where to search for what they are looking for. This has the direct benefits of allowing engineers to fix bugs they encounter when using other teams' projects. One of HubSpot's core beliefs when it comes to software engineering is that everyone should contribute what they can, instead of passing the blame to other teams. Most engineers are busy with new tasks and challenges and fixing a bug in code that belongs to a fellow engineer is substantially more productive for **both** teams than filing an issue and waiting for it to be resolved.

3.1 Technologies in use at HubSpot

As discussed, all of HubSpot is built using a set of technologies. This set of technologies is continuously growing as needs change and as new technologies become available. Typically any given team and/or project will use a subset of the technologies on offer. This section outlines some of the core technologies used by the Email Sending Infrastructure team.

3.1.1 Java 8

Java 8 is the second to newest release of Java. Java as a language has been around since the mid 1990s and has been under continuous development since then. As such, it has become a very stable and reliable language, used by nine million developers world wide (6). At HubSpot, Java 8 is used for the development of all backend services. The architecture employed is one of micro-services. This allows for rapid development and deployment of many loosely coupled services. The services are extremely modular and are often used by a variety of teams and projects.

Java is an object oriented language which provides a variety of concepts to aid in system design. Classically, the core of object oriented programming is that entities (objects) should model one *real* entity and one only. Client's can then use these objects in their own classes, abstracting away all of the complex implementation logic of the imported class from the client. Any information required by the object can be encapsulated within the object itself allowing the object to provide a simple interface to it's users, consisting of a number of methods, which if named correctly, provide a succinct description of what the method does which lines up exactly with what the client thinks the method should do.

Java 8 was released in March of 2014 and provides several new and extremely powerful features, many of which are used on a daily basis at HubSpot. Aside from all of the core functionality contained in Java, a subset of the most interesting and useful features it provides are outlined below.

Lambda Expressions

Often times it is useful to create classes which wrap a piece of logic or code. Similarly, systems often require the execution of a piece of code in response to a certain event (e.g. running some code in response to a mouse click). Traditionally in java this was accomplished by writing a manual *functional interface*. These functional interfaces were simple interfaces which provided a contract containing a single method. The class in question can then be passed an instance of an object which implements the functional interface and can thus invoke the method defined in the interface when appropriate. An example of this using traditional Java is outlined in code listing 3.1 which contains code for a button that can be clicked and can have on-click methods associated with it. Prior to Java 8, this code was cumbersome to write requiring a custom interface to be created, implemented by a concrete class and its implementation instantiated and passed to the class which manages the events.

```java
public class Button {
   private OnClickRunner onClickRunner;
   private String buttonName;

   public Button(OnClickRunner onClickRunner,
              String buttonName) {
      this.onClickRunner = onClickRunner;
      this.buttonName = buttonName;
   }

   public void click() {
      onClickRunner.runOnClick(buttonName);
   }

   public static void main(String[] args) {
      Button button = new Button(new ButtonClickLogger(), "SEARCH");
      button.click();
   }
}

interface OnClickRunner {
   void runOnClick(String buttonName);
}

class ButtonClickLogger implements OnClickRunner {
   @Override
   public void runOnClick(String buttonName) {
      System.out.println("Button " + buttonName + " was clicked.");
   }
}
```

Code Listing 3.1: onClick Listener without Lambda Expression

This code can be written in a much more concise form by using Java 8's new lambda expressions. The JDK now provides the most common functional interfaces which can be used in place of custom functional interfaces. For example the `Consumer<T>` functional interface, defines a single `accept` function which takes an argument of type `T` and returns nothing (it *consumes* the argument), which is exactly what the `OnClickRunner` in code listing 3.1 defined. The `Button` class can be refactored to use a `Consumer<String>`, allowing the logic of the `OnClickRunner` to be directly specified through a lambda expression. An example of this is shown in code listing 3.2 and the lambda expression can be seen on line 16.

```java
public class Button {
   private Consumer<String> onClickConsumer;
   private String buttonName;

   public Button(Consumer<String> onClickConsumer,
              String buttonName) {
      this.onClickConsumer = onClickConsumer;
      this.buttonName = buttonName;
   }

   public void click() {
      onClickConsumer.accept(buttonName);
   }

   public static void main(String[] args) {
      Button button = new Button(
            name -> System.out.println("Button " + name + " was clicked"),
            "SEARCH"
      );

      button.click();
   }
}
```

Code Listing 3.2: onClick Listener with Lambda Expression

As seen in code listing 3.2, the use of lambda expressions greatly simplifies the code. Several functional interfaces exist to enable the use of lambda expressions for functions with differing signatures. For example `BiConsumer<T, U>` represents a function which takes two arguments of type `T` and `U` respectively and returns nothing, or the more generic `Function<T, R>` which takes an argument of type `T` and returns a value of type `R`.

Streams

Another extremely powerful feature of Java 8 is the new Stream API. This allows for the bulk processing of collections through map/reduce like operations. Performing arbitrary data manipulation on a collection of Java objects is extremely common. Typically this could be accomplished using a simple loop. However this often requires the introduction of several local variables which can add excessive noise to code. Of course, some operations are still better suited to a simple loop, particularly if the data transformation function has side effects, in which case it is impossible to use streams. However it is widely considered good practice to minimize side effects of functions in order to maintain simplicity and making heavy use of streams is a great way to remind developers not to introduce side effects and to in general, minimize the amount of state required by a class. The Java 8 Stream API is fluent, allowing for arbitrarily complex stream operations to be chained together. Streams are also evaluated lazily, minimizing the amount of work to be done and can be parallelized internally by the API, providing excellent performance. The Java 8 Stream API consists of three types of operations:

Initial stream call	The `stream()` call can be invoked on any object that implements the `Collection` interface. This call returns a `Stream<T>` (where `T` is the type of the objects in the collection) allowing subsequent intermediate and terminal operations to be invoked on the returned stream.
Intermediate Operations	These are the operations which perform the data transformation. There are a variety of intermediate operations provided such as `sort` which sorts the objects in the stream, `filter` which filters objects in the stream according to some predicate and `map` which maps an arbitrary function over each object in the stream. As mentioned, the Stream API is fluent, meaning multiple intermediate operations can be chained together (for example filtering the objects and then sorting them).
Terminal Operations	The is the final operation which describes how the data should be reduced to a single object (which may be a collection). Common terminal operations are `max` which returns the maximum of the objects in the stream, `findFirst` which returns the first object the stream encounters that matches a given predicate and `collect` which defines how the objects in the stream should be collected into a collection (for example collecting the objects into a set would remove any duplicates)

Comparing code listing 3.3 and code listing 3.4, the clarity of the code produced using the Streams API can be seen. The code listings shows two approaches to a piece of logic which returns the length of each of the strings (in ascending order), without white space, and which don't contain the word *owl*. Although this is a toy example, several benefits of the Streams API can be seen. The code using streams (code listing 3.4) reads like the steps of an recipe, clearly stating what is performed at each step. However the code using the traditional `for` loop (code listing 3.3), requires `if` statements and redundant local variables, distracting the programmer from the core steps of the algorithm. Streams also provide the benefits of immutability and parallelism for free.

```java
List<String> words = Arrays.asList(" cats", "dogs ", "OWLS", "frogs");
 List<Integer> results = new ArrayList<>();
 for (String word : words) {
    word = word.trim();
    word = word.toLowerCase();
    if (!word.contains("owls")) {
       results.add(word.length());
    }
 }
 results.sort(Comparator.naturalOrder());
 return results;
```

Code Listing 3.3: Batch Processing without Streams

```java
List<String> words = Arrays.asList(" cats", "dogs ", "OWLS", "frogs");
return words.stream()
  .map(String::trim)
  .map(String::toLowerCase)
  .filter(word -> !word.contains("owls"))
  .map(String::length)
  .sorted()
  .collect(Collectors.toList());
```

Code Listing 3.4: Batch Processing with Streams

Completable Futures

Asynchronous programming is present in most modern systems. In the early days of Java, this was accomplished by the JDK through abstractions at the thread level. This required careful tracking of the state of threads by the programmer. Concurrent programs are incredibly difficult to reason about and thus concurrency is one of the most challenging aspects of modern software engineering and is the source of a huge number of bugs. However, the benefits of concurrent programming are extremely obvious, essentially making it a necessity in modern systems. Thus any abstractions that can aid in reducing the number of things a programmer must keep track of will be beneficial. In Java 8's case, this abstraction is the `CompletableFuture` API. This API allows for programmers to perform asynchronous tasks by specifying a `Supplier<U>`, a function which takes no arguments and returns (supplies) a value of type `U`, which will return a value at some point in the future. Thus what's returned from this call is not an instance of type `U`, but a `CompletableFuture<U>`, that is, an object that at some point in the future will contain an instance of type `U` (provided no errors occur).

The programmer may also specify an `Executor` (7) which is essentially an object (e.g. threadpool) capable of running tasks on a thread other than the thread in the current context. If no `Executor` is specified, the task is automatically submitted to Java's work stealing `ForkJoinPool` (8).

The API provides a simple `get` method for blocking until the return value is present (or throws an exception). More interestingly however, it also provides several methods to specify subsequent processing of the return value when it returns, allowing the program to continue doing useful work while waiting on the result. A subset of these methods are the following (9):

thenApply This method is used to supply a function that should be run upon completion of the underlying `CompletableFuture`. The result of the `CompletableFuture` will be passed as the sole argument to this function. This function is free to return any type and in doing so, sets the type associated with the underlying `CompletableFuture`.

thenCompose This method is very similar to `thenApply` except it is used when the desired function to be run is also asynchronous (that is, it too returns a `CompletableFuture`). This behavior could technically also be handled by `thenApply` (as it is free to return any value), but this would cause the return type of the parent `CompletableFuture` to itself be a `CompletableFuture`. The benefit of `thenCompose` is that it contains logic to unwrap this nested `CompletableFuture`, allowing the return type of the parent to remain a `CompletableFuture<T>` even if the function passed to `thenCompose` itself returns a `CompletableFuture<T>`.

thenAccept This method is used when the return type of the `CompletableFuture` only needs to be used in the registered callback and is not used outside of the supplied function. As such, this method takes a `Consumer<T>` as its argument, that is, a function which takes `T` as a parameter and does not return anything.

Finally, all of these methods contain corresponding methods in which an `Executor` is also specified. This allows the callbacks to be run on the specified `Executor` (thread pool).

3.1.2 Guice - Dependency Injection

Google's Guice (10) is a dependency injection framework for Java. Dependency injection is a software pattern which abstracts away the actual construction (instantiation) of objects from the user. As a system grows and abstractions are built atop of one and other, simply instantiating an object can be come tedious and difficult. Traditionally, in order to instantiate an object, all of its unconditionally required dependencies must be passed to the constructor of the object. Otherwise the object could be left in an inconsistent state. Thus in order to instantiate an object `O`, all of its dependencies, for example `X, Y and Z` must be provided to `O`'s constructor. Thus the client of `O` must first instantiate instances of `X, Y and Z` before it can use `O`. However `X` may have it's own set of dependencies and the problem simply gets worse and worse. An example of the difficulties this can cause (based on the example in Guice's documentation (11)) is shown in code listing 3.5. The code shown has to recursively create each of the dependencies for each of it's dependencies which can quickly get out of hand for large systems.

```java
public class PizzaOrderingService {
  private SqlTransactionLogger transactionLogger;
  private AccountsDao AccountsDao;
  private PayPalCustomerBiller customerBiller;

  public PizzaOrderingService() {
    SqlCredentials sqlCredentials = new SqlCredentials("dbName",
        "password");
    SqlConnection sqlConnection = new SqlConnection(sqlCredentials);
    PayPalCredentials payPalCredentials = new
        PayPalCredentials("accountId", "password");
    this.transactionLogger = new SqlTransactionLogger(sqlConnection);
    this.accountsDao = new AccountsDao(sqlConnection);
    this.customerBiller = new PayPalCustomerBiller(payPalCredentials);
  }
}
```

Code Listing 3.5: Pizza Ordering Service without Dependency Injection

Traditionally, this problem was somewhat helped (but not entirely solved) by the using the factory pattern (12). Depending on the implementation, this can ease the pain of getting access to objects the client depends on, by allowing the factory to contain the logic for the instantiation of the class' dependencies. However this still requires the manual implementation of the factory classes themselves, leaving much to be desired.

Dependency injection solves this problem by allowing fully formed instances of dependencies (e.g. a `PaypPalCustomerBiller`) to be *passed* to the client, removing the need for the client to instantiate the object themselves. Client's simply ask for their dependencies to be injected into their constructor. Guice provides this functionality by simply annotating the constructor with the `@Inject` annotation. This (along with a some other boilerplate) informs Guice that this class should have it's dependencies injected into the class' constructor. Guice accomplishes this by building an arbitrarily complex dependency graph at run time. When a class needs a certain dependency, the graph can be examined in order to figure out how to instantiate that dependency.

However Guice does need a starting point - dependencies can't just be injected for every single class without providing some initial classes in which to build upon. In the example above, these base classes would be the `SqlCredentials` and `PayPalCredentials` classes. These classes should not have any injected dependencies. In this example these could simply read the credentials needed from a file. Guice allows us to add vertices to the dependency graph by *providing* objects using the `@Provides` annotation. Thus both `SqlCredentials` and `PayPalCredentials` would need to be provided using `@Provides`. The resulting object graph from the code above is show in figure 3.1 - notice the classes at the bottom of the hierarchy are **provided**.

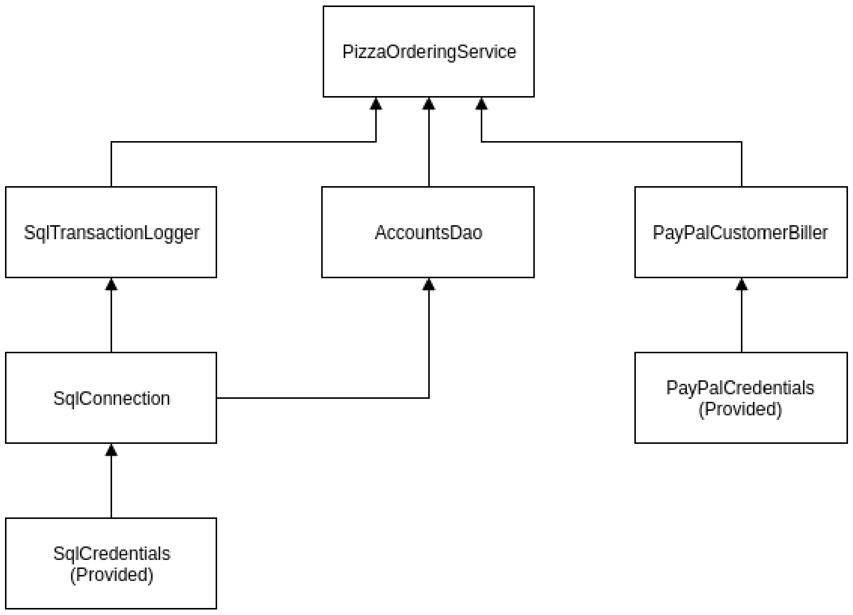


Figure 3.1: Dependency Graph Built by Guice

This leads to extremely simple code for the `PizzaOrderingService` in which the logic is entirely separated from the dependency management.

```java
public class PizzaOrderingService {
 private SqlTransactionLogger transactionLogger;
 private PayPalCustomerBiller customerBiller;
 private AccountsDao accountsDao;

 @Inject
 public PizzaOrderingService(SqlTransactionLogger transactionLogger,
                             PayPalCustomerBiller customerBiller,
                             AccountsDao accountsDao) {
   this.transactionLogger = transactionLogger;
   this.customerBiller = customerBiller;
   this.accountsDao = accountsDao;
 }
}
```

Code Listing 3.6: Pizza Ordering Service with Dependency Injection

Aside from abstracting away the dependency instantiation, dependency injection also has the added benefit in that all of a class' dependencies become part of the class' constructor signature. There are no required dependencies that are buried within the implementation logic. This makes the code much simpler to test. The `PayPalCustomerBiller` used by the class is essentially hardwired into the code that does not use dependency injection (code listing 3.5), meaning there is no way to test this class without actually billing a (fake) customer. However as the `PayPalCustomerBiller` used by the dependency injected code (code listing 3.6) is injected into the constructor, a mock of this object (which skips the actual billing, but returns results as if it actually billed a customer) can be passed to the class and used for testing purposes.

3.1.3 Immutables - Immutability for Java

Immutability is a programming paradigm in which once an object is created, it may never been changed. At first glance this sounds like a bad idea which will result in redundant object creation, but the positives strongly outweigh the negatives. The primary benefit of immutability is that the programmer is **guaranteed** that any object they hold a reference to, will never be changed. This concept is closely tied to the concept of writing *pure* functions. These are functions which have no external side effects. That is, they take in arguments and return a value, but do not change the input arguments (or any other state contained in the program) in any way.

These benefits are best highlighted through sample code. Consider the case of a simple website where each login attempt by a user should be logged to a database table in order to detect a hacker trying to crack a password by repeatedly trying to login as the same user. For obvious reasons, this table should not store the user's password (in case of real login attempts), so this should be omitted from the object being logged to the database. An example **`LoginRequest`** is shown in code listing 3.7. A **`LoginAttemptLogger`** class is written to handle logging these attempts to the database and is shown in code listing 3.8. The **`logLoginAttempt`** method handles stripping the password from the **`LoginRequest`** and writing it to the database.

```java
class LoginRequest {
  String username, password, ip;

  public LoginRequest(String username, String password, String ip) {
    // Initialize fields..
  }

  public void setPassword(String password) {
    this.password = password;
  }
}
```

Code Listing 3.7: An Example `LoginRequest`

```java
class LoginAttemptLogger {
  LoginRequestDao loginRequestDao;

  @Inject
  public LoginAttemptLogger(LoginRequestDao loginRequestDao) {
    this.loginRequestDao = loginRequestDao;
  }

  public void logLoginAttempt(LoginRequest loginRequest) {
    // Don't log the users password to the database
    loginRequest.setPassword("");
    loginRequestDao.writeToTable(loginRequest);
  }
}
```

Code Listing 3.8: An Example `LoginAttemptLogger` Implementation

However this style of code is a recipe for disaster. The **`LoginRequest`** is *mutable* and the **`logLoginAttempt`** method contains a side effect in that it sets the password of the **`LoginRequest`** to an empty string. Some perfectly reasonable client code is shown in code listing 3.9 in which the client logs the login request to the database and subsequently tries to log in. In this case, no user will ever be able to login as all of the passwords of the login requests are always mutated to be an empty string. Thus, having `LoginRequest` as a mutable object causes a critical bug that will not be caught until runtime. The client login code may check and find that the `LoginRequest` contains a password prior to logging the login attempt to the database. However when it goes to finally login, it never will.

```java
public boolean loginClient(LoginRequest loginRequest) {
  logLoginAttempt(loginRequest);
  Account account = accountsDao.getAccount(loginRequest.username);
  return account.password == loginRequest.password;
}
```

Code Listing 3.9: Perfectly Reasonable Client Login Code

The solution to this problem is to create a new `LoginRequest` without the user's password and log that to the database. This can be done inside of the client login code (defensive copying) before passing the `LoginReuqest` to the `logLoginAttempt`. However if mutators are provided, it is extremely likely that they will be used somewhere in the code. Thus the best solution is simply to not provide them at all - make the object entirely immutable. This in turn requires some clunky code inside of the client login method (see code listing 3.10), but avoids the issue caused by the side effect of `logLoginAttempt`.

```java
public boolean immutableLoginCLient(LoginRequest loginRequest) {
  LoginRequest loginRequestCopy = new LoginRequest(
    loginRequest.username,
    loginRequest.password,
    loginRequest.ip);
  logLoginAttempt(loginRequestCopy);
  Account account = accountsDao.getAccount(loginRequest.username);
  return account.password == loginRequest.password;
}
```

Code Listing 3.10: Logically Correct Login Code with Extra Boilerplate

The Immutables (13) provides a framework for auto generating fully immutable object implementations in Java. These implementations provide extremely useful functionality such as implementing builders and providing methods for updating the fields of an object in an immutable way. The immutable data structure is defined using an interface (annotated with `@Value.Immutable`) which contains the getter methods for each desired fields of the object. A class which implements this interface in an immutable way is then auto generated by the framework and can is then used in place of the interface. An example of the interface used for **`LoginRequest`** is shown in code listing 3.11.

```java
@Value.Immutable
interface LoginRequestIf {
  String getUsername();
  String getPassword();
  String getIp();
}
```

Code Listing 3.11: Interface for `LoginRequest` using Immutables Framework

The implementation of this interface generated by the framework then provides a **`withFieldName`** method for each of the fields defined, allowing a new instance of the object with the updated fields to be obtained with a single method call as shown in code listing 3.12. This solves the problem of mutating the **`LoginRequest`** that the client holds a reference to and drastically simplifies working with immutable objects in Java. This framework is used extensively at HubSpot and is a major contributor to the simplicity of writing code without bugs at the company.

```java
public void logLoginAttempt(LoginRequest loginRequest) {
  // Don't log the users password to the database
  loginRequestDao.writeToTable(loginRequest.withPassword(""));
}
```

Code Listing 3.12: The `LoginAttemptLogger` using the Immutables Framework

3.1.4 Kafka - Streaming Platform

Kafka (14) is a horizontally scalable, fault tolerant, distributed streaming platform used to read and write streams of data in real time. It is an extremely high performance system and is used extensively by the Email Sending Infrastructure team as the primary data pipeline. Kafka runs on it's own cluster and stores streams of records inside categories known as Kafka *topics*. Kafka provides two key APIs that are used at HubSpot - one for producing records to a given Kafka topic and one for consuming records from a specific Kafka topic. Kafka is used extensively inside of the team's internal pipeline, but also as an interface between teams. For example, the teams responsible for generating the full HTML body of an email to be sent on behalf of a customer can produce this ready to be sent email to a specific Kafka topic. Kafka consumers owned by the Email Sending Infrastructure team are subscribed to this topic and thus pick up these records and can perform the send of the email. This entirely decouples the work done by the two teams. The upstream teams simply put messages onto Kafka to be handled elsewhere. An obvious alternative to using Kafka would be to simply expose a REST endpoint to which the message is POSTed to. However simple HTTP would struggle to support the volume of requests (upwards of 25M emails per day) seen by the Email Sending Infrastructure team. As mentioned, Kafka is horizontally scalable, meaning the number of nodes on the Kafka cluster can simply grow as the number of messages published to Kafka increases. Similarly on the consumer side, the number of deployed consumers subscribed to a given topic can simply be increased in order to meet the increased number of records published to the topic.

Kafka also provides another layer of granularity - partitions. Each Kafka topic is segmented into a number of partitions. Each partition is (at any given time) owned by exactly one Kafka consumer, but each Kafka consumer may own multiple partitions. This leads to an interesting case when choosing the number of partitions to use for a given topic. Ideally, the number of partitions should be a highly composite number (15). These are numbers which are divisible by many other numbers, for example 24 which is divisible by 2, 3, 4, 6, 8 and 12. To illustrate why, consider the case where 9 partitions are used - the work load is only equally distributed if 1, 3 or 9 consumers are used. Thus if 3 consumers isn't enough, scaling to five consumers means four consumers will be consuming from two partitions and one will only be consuming from one partition. Using a highly composite number of partitions allows for more flexibility when choosing the number of consumers, while still balancing the load across the consumers. Kafka can also handle rebalancing the workload, by redistributing the partitions when the number of consumers changes.

When messages are produced to a Kafka topic, a decision must be made on which partition to assign the message to. This can be done intelligently with load balancing in mind, or simply using a round robin. Partitions can also be replicated in order to provide fault tolerance.

An example of a Kafka topic with two consumers is shown in figure 3.2.

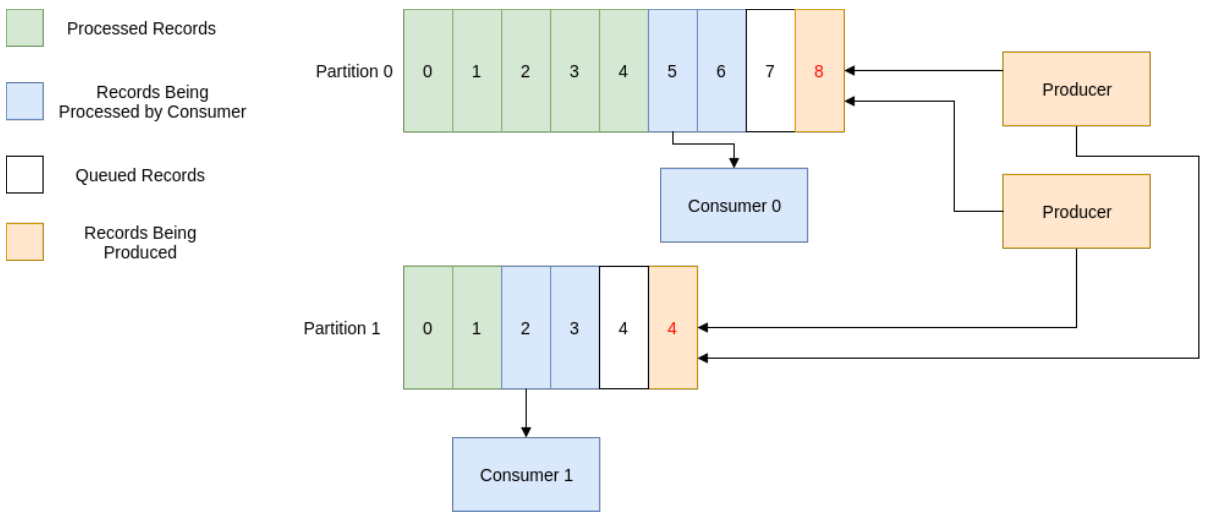


Figure 3.2: Kafka Producers and Consumers

The current situation is outlined as follows:

- Consumer 0 has successfully processed (and committed) messages 0 to 4 and is currently processing messages 5 and 6 of partition 0
- Consumer 1 has successfully processed (and committed) messages 0 and 1 and is currently processing messages 2 and 3 of partition 1
- Message 7 in partition 0 and message 4 in partition 1 have both been successfully produced and committed to the log, but not yet processed
- Message 8 in partition 0 and message 4 in partition 1 will be the next messages produced by the producers.

The main purpose of Kafka partitions is to allow records on a single topic to be processed in parallel. As each partition is only owned by a single consumer, the model required for keeping track of what messages have been processed in a certain partition is very simple. However since a consumer may own multiple partitions, the configuration can be tweaked to ensure consumers are always busy. Partitions also aid in keeping records on a single partition isolated. If a bad record appears in a particular partition and the consumer gets stuck trying to process it, only that partition will will be impacted.

Each message published to a Kafka partition gets an incremental id known as an *offset*. Consumers are responsible for managing their offset in the message stream. Thus consumers can be configured to start from any offset, which can even allow for reprocessing of data should the need arise. This has been useful in the past at HubSpot when a bug has caused emails to fail to send. The consumers can have their offsets reset to when the bug first surfaced and the emails will be resent.

However this is a delicate process and is only used as a last resort. The offsets for consumers 0 and 1 in figure 3.2 would be 4 and 1 respectively.

A key concept to understand about Kafka is that consumers are presented with batches of records, of a configurable size. The batch size in figure 3.2 is two. The records inside the batch may be processed out of order, but the batch of records is considered completed **only when every record in the batch has been processed**. In Kafka, only an entire batch of messages can be marked as processed. For example consider consumer 1 in figure 3.2. Should the consumer succeed to process message two, but fail to process message three, Kafka must be notified of the failure to process the batch of messages (or will notice a timeout) and the entire batch will be retried.

Another interesting concept in Kafka is consumer groups. Consider the case where two entirely separate sets of consumers (running different code) need to read from the same Kafka topic. Both sets of consumers should be able to process every message that is published to the topic and this is what consumer groups provides. Without consumer groups, another consumer could be set up in order to read from the Kafka topic, but as discussed, it would only acquire a certain number of partitions and thus miss some messages (and steal messages from existing consumers). The solution is to specify the consumer group that each running consumer belongs to. In this case, since there are two sets of consumers, both doing different things, there should be two consumer groups. Kafka will then treat all of the consumers in each consumer group as if there are no other sets of consumers reading from the Kafka topic. This feature is best understood by considering the two edge cases:

- If all consumers are in the *same* consumer group, Kafka simply load balances the partitions across the number of consumers. Each consumer will see a *subset* of the records published to the kafka topic, depending on the partitions owned.
- If all consumers are in *different* consumer groups, each consumer will have its own offset in **all** of the partitions. That is, the consumer responsible for partition zero in consumer group A may have a current offset of 100, while the consumer responsible for partition zero in consumer group B may have a current offset of 0. This is essentially, publish-subscribe (pub-sub) behavior. All of the consumers will simply see all of the records and the rate of consumption of a certain consumer has no impact on the rate of consumption of another consumer.

Continuing with the previous example of having all of the emails to be sent through HubSpot contained on a Kafka topic, the following consumer groups could be set up in order to both send every email that appears on the topic and to bill the customer for each email they send. This is shown in figure 3.3

Sending CG This consumer group would contain all of the Email Sending Infrastructure team's consumers. There would likely be a lot of consumers in this consumer group in order to keep up with the time consuming process of actually sending the emails on behalf of the customers. These consumers would need to be as efficient as possible and the delta between the current offset (the index of the most recently produced message) and the oldest offset (the index of the oldest message still being processed) for each partition would likely be carefully monitored to ensure the consumers are not falling behind

Billing CG This consumer group would perhaps simply read the account number of the customer sending the email and bill that customer. This consumer group would be considerably less time critical and would likely only need to perform some light weight task for each message on the Kafka topic.

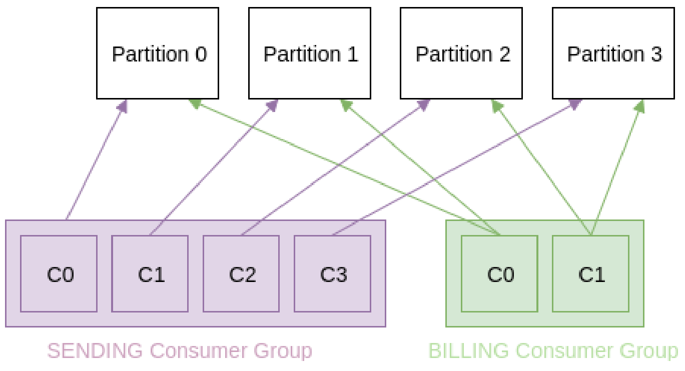


Figure 3.3: Partition Assignments with Multiple Consumer Groups

A final note of interest regarding Kafka is that it provides an *at-least-once* guarantee that messages will arrive at consumers for processing. If a single message in a batch of messages cannot be processed for whatever reason, the entire batch will be retried. Thus external idempotency logic is required in order to obtain an *exactly-once* processing guarantee. The Email Sending Infrastructure team uses a simple HBase table in order to lock each email that is to be sent, prior to actually sending it.

3.1.5 Hadoop - Big Data Processing

As with most modern companies, HubSpot's platform generates a monumental amount of data. This data can be leveraged to gain extremely useful insights into how the platform is used, how certain systems are performing and what should be targeted by engineers working on the product. As discussed in section 3.1.4, Kafka is used throughout HubSpot to provide high performance data streams. This is ideal for things like moving an email through the entire pipeline from design to sending. However, the contents of Kafka topics are often also persisted (for example for 30 days) to Amazon Web Service's cloud storage facility, Amazon S3 (16) . Persisted Kafka topics provide a huge amount of data which is typically critical to a system's operations if Kafka is used in the first place.

Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models (17). Often times the size of the datasets persisted to S3 would be too large to hold on disk of a typical machine, and certainly not in memory. Hadoop is horizontally scalable allowing for more and more machines to be added to a cluster as demand grows. Hadoop is used extensively at HubSpot for processing these huge volumes of data by writing Hadoop jobs. These are jobs which make use of the Hadoop framework and run on a cluster created specifically for Hadoop jobs. These jobs can be scheduled and used to periodically hydrate some other data store, or on demand and used when a certain question needs to be answered.

An example of a scheduled job used by the Email Sending Infrastructure team is one which aggregates all of the sending statistics for a given HubSpot email sending account. This is used to hydrate an SQL table, which backs an internal dashboard which is used to gain insights into the system. Large customers can potentially send hundreds of thousands, or occasionally millions, of emails in a given day. As discussed, all of these emails are contained in Kafka and persisted to S3. In order to extract the useful information, these files need to be indexed and aggregated. Several key values can then be derived from this data such as the percentage of emails sent from this account that were delivered first time, the breakdown of the reasons emails sent from this account were rejected (e.g. the IP has a bad reputation) and many others.

Hadoop offers a large number of benefits. Hadoop is fault tolerant - for example if a mapper node dies and produces no output, the corresponding map process will be repeated on another node. Hadoop make use of the *Hadoop Distributed File System (HDFS)* (18) which provides data replication and support for huge datasets. One of the key benefits of using HDFS that Hadoop leverages is that it can attempt to assign work to nodes based on the data they **already have**. This limits the costly operation of moving data around the network. This is succinctly summarized by Hadoop's famous design ethos - *"bring the processing to the data"*.

The main user facing API contained in Hadoop, is Hadoop Map-Reduce. Map-reduce is a programming model for highly parallelizable data processing. The map-reduce methodology consists of four key phases - *split, map, shuffle, reduce*. As an example, consider a dataset which is a corpus of quotes of the form: *"Learning never exhausts the mind - Leonardo da Vinci"*. An example task on this dataset would be to calculate the number of quotes contained by each of the authors in the corpus. This would be implemented using map-reduce in the following manner:

Split The initial step would be to split the corpus into chunks for processing on different nodes in the cluster. As discussed, Hadoop does this intelligently by trying to assign chunks for processing to worker nodes who already contain that data. This phase would output a single record for each quote in the corpus.

Map This is the beginning of the useful processing of the data. The purpose of the map phase is to output a key-value pair. In this example, the key would be the name of the author and the value would simply be one, as each entry in the corpus is a single quote. Thus the map phase would likely require a regular expression to parse the name of the author from the quote.

Shuffle The *shuffle* phase is responsible for grouping the output key value pairs produced by the map phase by key. This forms a list of the values that are associated with each key from the map phase. In this example, a possible output of the shuffle phase would be:
`Leonardo da Vinci -> [1, 1, 1]`, indicating that three entries with the key “Leonardo da Vinci” were found and each had a corresponding value of 1.

Reduce The final phase is the *reduce* phase. The input to the reduce phase is the output of the shuffle phase - a key and a collection of values associated with that key. As the name suggests, the reduce phase is responsible for reducing this collection of values down to a single value. In this example, the single value output of the reduce phase would simply be the number of elements in the collection of values, indicating the number of quotes found by the particular author.

3.1.6 Protobuf - Structured Data Serialization

Protocol Buffers (Protobuf) are Google’s language-neutral, platform-neutral, extensible mechanism for serializing structured data (19). Protobuf is similar in concept to XML or JSON, but produces data which is 3 to 10 times smaller in serialized form (20). Protobuf also offers many powerful features such as enums, typing, default values and nested messages.

Protobuf works by defining the desired data structures in a special .proto file. Compilers exist for a large number of languages (e.g. Java, C++, Python etc) which compile these .proto files into classes usable inside the language. For example the Java Protobuf compiler generates immutable Java objects with a variety of useful features such as builders and methods for serializing and deserializing to and from immutable byte arrays. The serialization / deserialization process is extremely fast and is especially useful in conjunction with Kafka (see section 3.1.4), which operates solely on byte arrays. An example of a Protobuf message for a person is shown in code listing 3.13. This example shows the rich features such as optional fields (email), enums (PhoneType), nested messages (PhoneNumber) and arrays of values (PhoneNumber).

```
message Person {
  required string name = 1;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Code Listing 3.13: A Protobuf Representation of a Person

3.1.7 gRPC - Remote Procedure Calling

gRPC is a cross language remote procedure calling framework (21). It serves as an alternative to the current defacto standard of having services communicate using REST endpoints over HTTP/1.1. gRPC operates by defining a service and specifying the methods that can be called remotely, along with their parameters and return types (22). gRPC uses Protobuf (see section 3.1.6) as it's IDL (interface definition language). That is, the defined methods (by default) accept as parameters, and return, Protobuf messages. As with Protobuf, compilers exist for a variety of languages which generate the client side and server side code for their respective languages. Clients of the service can then remotely invoke these methods on objects they hold known as *stubs*. The methods are then remotely invoked on the server. To the client, the procedure call appears no different to any other procedure call as all of the remote networking is handled by the framework.

gRPC is built upon HTTP/2 which offers many new powerful features. One of the main advantages of HTTP/2 over HTTP/1.1 is multiplexing which allows multiple HTTP requests to be made (and responses received) asynchronously over a single TCP connection. The enables the gRPC framework to provide four types of RPC invocations:

1. The first type of call is a simple *unary* call. In this case, the client sends a single request to the server, who handles the request and returns a single response. This is equivalent to a typical HTTP/1.1 request and response.
2. The second type of call is *server side streaming*. In this case, the client sends a single request to the server and the server responds with a stream of messages. This is useful in cases where the response may be a large collection of items. The client may not need **all** of the items before it can begin processing. Thus the server can respond with a stream, allowing the client to begin reading messages from the stream as they become available.
3. The third type of call is *client side streaming*. This is the exact opposite to server side streaming and is useful in cases where the client must send a lot of data, but the server can begin processing that data before it has all been received.
4. The final type of call is *bidirectional streaming*. As expected, this is when both the client and server communicate using streams. These streams are entirely independent from one and other, allowing the client / server to read messages in any order.

The Email Sending Infrastructure team uses gRPC at the final stage of the email sending pipeline. The last step is to get the email to the servers which are responsible for sending the email to the recipient using SMTP (see section 3.2.2). As discussed in section 1.3, SMTP email sends can take on the order of seconds. This leads to a very asynchronous system which causes issues if using standard REST endpoints over HTTP/1.1.

gRPC is used in this case instead of Kafka as particular emails **must** be sent from particular IP addresses for SPF to pass (see section 3.2.1 for more information on SPF). Certain email sending servers will be bound to certain public IP addresses, meaning certain emails **must** be sent from certain servers. If Kafka was used for the final step, there is no guarantee that the consumer who owns the partition the email ends up in will be bound to the correct IP address, meaning they would not be able to send the email. gRPC can be used to connect to the server who is **known** to be bound to the correct IP address. This could also accomplished using a simple REST endpoint, but as discussed, this causes performance issues.

3.2 Email Specific Technologies

3.2.1 Domain Name System (DNS)

Although not entirely email specific, DNS is extremely critical for the Email Sending Infrastructure team. Aside from teams working on the HubSpot development platform (that is teams working on the platform which engineers develop HubSpot products on), most other teams need not concern themselves with DNS at all. However as DNS is such an important component of email, DNS records must be carefully maintained. For email, there are five key DNS record types, outlined below.

A A records are used extensively in DNS. These records define the IP address that is associated with a particular domain. For example, the IP address of the web server which hosts the website *hubspot.com* can be obtained by performing a DNS lookup for the A record associated with *hubspot.com*. This address translation is done every time a user visits a website and enables the use of memorable, human friendly domain names instead of raw IP addresses.

MX MX (Mail Exchanger) records are similar to A records except they define the **hosts** which can accept mail for a given domain. For example, if an email is to be sent to *billing@hubspot.com*, the email sender must know where to send that email. Thus an MX lookup is performed for *hubspot.com*, the domain in which the account associated with the email address *billing@hubspot.com* resides on. This typically returns a host name, for example, *mx.hubspot.com*. Thus a subsequent A record lookup for *mx.hubspot.com* is required to determine the exact IP address(es) of the server(s) which can accept email for *billing@hubspot.com*.

SPF SPF (Sender Policy Framework) records are used to authorize a set of IP addresses which a domain intends to send email from. The friendly-from address of an email (the sender's address) is simply a plain text header contained in the email. This means that anyone could potentially send email as someone else by spoofing the friendly-from address. SPF records provide part of the mechanism which solves this problem. A domain can declare a list of IP addresses it intends to send email from using an SPF record. Thus when an email is received from *stefano@example.com*, the SPF record associated with *example.com* can then be fetched and if the source IP address of the email does not match one of the declared IP addresses, the email may be blindly accepted, tentatively accepted or dropped entirely. An example SPF record would be `v=spf1 ip4:1.2.3.4/30 -all`. The first component (`v=spf1`) indicates the SPF version in use, the second component (`ip4:1.2.3.4/30`) is a CIDR representation of the IP addresses this domain authorizes (see section 4.2.1) and the final component (`-all`) indicates that email sent from all other IP addresses except the ones defined here should be dropped.

PTR PTR records are the inverse of A records and are used to determine the host associated with a particular IP address.

DKIM DKIM (DomainKeys Identified Mail) records provide another level of authentication. DKIM uses asymmetric-key cryptography to digitally sign emails, verifying that the email was sent by an authorized sender (provided only authorized senders hold the private key). The public component of DKIM keys are contained under the `_domainkey` sub domain and are prepended with a DKIM *selector* indicating which DKIM key was used. The *DKIM-Signature* header is included in the email, an example of which is shown in code listing 3.14.

```
DKIM-Signature: v=1; a=rsa; d=example.com; s=dk1; q=dns/txt;
h=from:to:subject:date;
bh=UOJhdjksJHSDAasdhUYHQO12SaDKJAHDSAPashd8ojhdsafdui;
b=IUASHB89FHNasdjlkasjlkHONJKSDNohjndjhsdkajJHjsdsJHD;
```

Code Listing 3.14: An Example DKIM signature

The most important DKIM tag-value pairs are outlined below (23):

v This indicates the DKIM version in use

a This indicates the algorithm used to generate the signature.

s This is the selector prefix and is used to pick the public key to use from those available at the `_domainkey` subdomain.

h This contains the list of headers which were hashed to generate the *b* tag

b	This contains the Base-64 encoded hash of the the headers listed in *h*
bh	This contains the hash of the body of the email
d	This contains the domain in which the corresponding public key can be found

The corresponding public key can be found by performing a TXT lookup of `<s>._domainkey.<d>`, or in the above case `dk1._domainkey.example.com`. The value of these records also contain tag-value pairs, such as *k* which indicates the cryptography system used and *p* which contains the public key itself. Once the public key has been obtained, the inverse of the signing operation can be applied to ensure the email came from an authorized sender.

3.2.2 Simple Mail Transfer Protocol (SMTP)

SMTP is the protocol used for sending email to a mail server. SMTP is quite an old protocol, with the first RFC being published in 1982 (24). For an email to be sent from one machine to another, a connection must be opened up between the mail server responsible for the sender's email and the mail server responsible for the recipient's email. SMTP defines the way in which messages should be exchanged between these two servers. Most mail servers today support Extended SMTP (ESMTP), which is outlined in RFC 1869 (25). This defines a format for indicating which extensions a given SMTP server supports.

SMTP contains 6 main steps to send email:

EHLO The first SMTP command of interest is `EHLO <SMTP domain>`. This is the first message expected by the server. In turn, the SMTP server will respond with a 250 OK message and list all of the extensions it supports.

MAIL FROM: The sender then specifies the mail from address to be used for the email (who the email was sent from) using `MAIL FROM: <from_email_address>`. Again, provided the server accepts the email address as valid, the server should return a 250 OK.

RCPT TO: The sender then specifies the email address that the SMTP server should deliver the email to using `RCPT TO: <recipient_email_address>`. Provided the SMTP server accepts responsibility for this recipient address (it expects to

	receive email for this address), the server will again respond with a 250 OK. Otherwise, the server will reject the message and terminate the connection.
DATA	The sender then indicates that it wishes to begin sending the data contained in the email message, by sending the `DATA` command. The server will then return with a 354, along with some instruction on how to terminate the message, indicating that the client may begin to send the message.
<content>	The sender then proceeds to send the content of the email message line by line, followed finally by a specific message indicating that the content is completed. This termination message is a single period character on its own line. The server should again respond with a 250 OK once the content has been received.
QUIT	Finally, the sender issues a `QUIT` command. The server will respond with a 221 message and the session may be safely terminated.

As the name suggests, SMTP is quite a simple protocol. Many libraries exist which handle SMTP messaging exchanges. However as this needs to happen for every single email that is sent through HubSpot, the Email Sending Infrastructure team implemented their own Java high performance and open source SMTP library, NioSmtpClient (26). This SMTP client supports several of the SMTP extensions such as TLS which provides encryption and SMTPUTF 8 which supports sending UTF8 characters over SMTP. It also allows connections to be recycled and can automatically determine most efficient sending method based on the EHLO response from the server.

3.3 System Management and Health Monitoring

While working on projects of this magnitude, bugs and issues are an inevitability. The amount of traffic seen by these systems compounds any small issues or bugs present in the system. As such, it is critical to have systems in place which monitor the health of the system and inform the team of any potential issues with the system. Whats more, these issues must be continuously examined and remedial action must be taken where applicable. The Email Sending Infrastructure team makes use of several tools and methods for monitoring the health of their systems, a subset of which are outlined in the following sections.

3.3.1 Log4j2

Log4j2 (27) is a Java framework which is provides facilities for logging to different log levels and advanced log filtering (for example with regular expressions). This provides an excellent facility for understanding why systems are behaving unexpectedly in production. A common pattern is to insert log messages to a low priority log level (e.g. DEBUG) which describe the state of the system or the code path taken. Typically when the system is behaving normally, a higher priority log level is set (e.g. INFO) meaning these finer grained log messages are skipped. However, should an issue arise, the log level can then be easily switched to the lower priority temporarily to get a more detailed insight into why the system is misbehaving. This pattern allows detailed log messages to be produced only when they are needed, reducing the amount of noise present in the logs. The framework also provides the ERROR level which can log error messages as uncaught exceptions without killing the currently executing thread.

3.3.2 Sentry

Sentry (28) is an online platform which logs uncaught exceptions that arise during program execution. This greatly simplifies the task of finding out the reason for a system fault or failure without the need to trawl through pages of log files. Sentry logs the full stack trace associated with an exception, the time of occurrence and other pieces of meta data such as the name of the deployable. It uses this data to monitor the occurrences of particular exceptions over time, provides facilities for opening and closing GitHub issues and most importantly, to send an email to all those subscribed to the project (e.g. the Email Sending Infrastructure team in this case) when an exception occurs. Sentry proves to be extremely useful at deploy time. Obviously when deploying new code to production servers, one must be sure that the changes did not cause the system to enter an unhealthy state. Provided the code is well written and that unexpected exceptions that occur are not silently swallowed, Sentry can be monitored at deploy time in order to help provide the engineer with confidence that the deployed changes were non breaking. Sentry also provides support for integrating into Log4j2 (see section 3.3.1). Sentry can monitor ERROR level log messages that are produced by Log4j2 and subsequently log these error messages to the Sentry platform. As an engineer this combination of tools is extremely useful for indicating that the system has found an issue, without killing the thread. This is ideal in cases where some work has been done and the system has encountered a critical error, but does not need to be restarted. This mitigates the need to repeat the work, but still informs the team that an error has occurred by logging an exception to Sentry.

3.3.3 SignalFX

SignalFX (29) is a online tool for recording metrics and data visualization. Gathering and analysing metrics is a key part of ensuring the health of a complex system. When things start to fail, SignalFX is the first place engineers look to. SignalFX essentially allows data to be dumped to the cloud and for complex graphs and charts to be rendered in real time using this data. SignalFX supports creating dashboards consisting of many of these graphs and charts. The Email Sending Infrastructure team has several of these dashboards, each encapsulating a single part of the system. When problems inevitably arise, the team can examine these dashboards in order to try and isolate and find the problem. Once remedial action has been taken, the dashboards can be monitored to ensure the desired effect takes place. Being able to see these metrics in close to real time is incredibly useful for diagnosing faults with the system. It also serves as an excellent mechanism for finding parts of the system that can be improved. For example, critical code paths can be timed and the results logged to SignalFX. This allows accurate inferences to be made about parts of the system and allows the team to target parts of the system which need improvement.

SignalFX also allows for detectors to be configured. For example, a chart could be created monitoring the number of emails sent in the past minute by each IP address responsible for sending emails. A detector can then be put in place to detect if this number falls below a certain threshold. Detectors can be configured to alert via email and Slack when they trigger, allowing engineers to be notified.

SignalFX also supports complex data aggregations. This is beneficial as it decouples calculations that need to be done on metrics from the code which the metrics are monitoring. Performing any sort of data manipulation in performance critical code paths is obviously not desirable when every millisecond counts. Instead, metrics gathering is as simple as producing the data to SignalFX. The complex graphs and charts can then be configured inside of the SignalFX tool, entirely independent to the production code.

4 Tasks Undertaken

Throughout the course of the internship, a variety of tasks were undertaken and completed. For the sake of brevity a small subset of the most interesting of these tasks are outlined below.

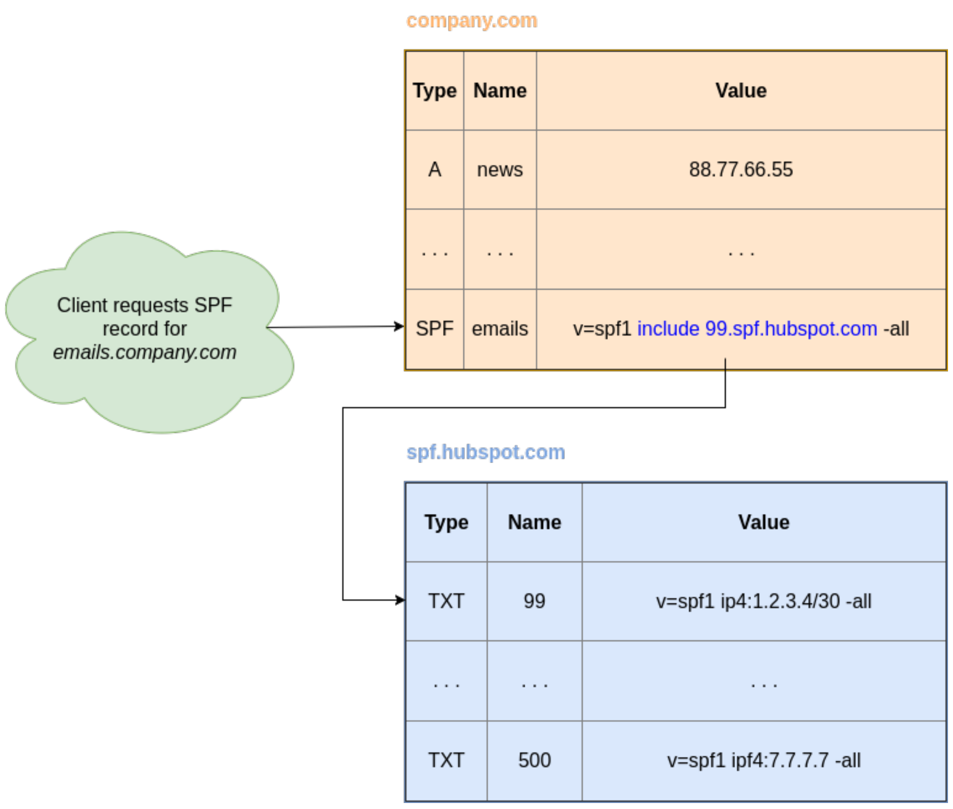
4.1 Rebuilt DNS Management System

DNS plays an important part in sending emails. As discussed in section 3.2.1, there a several DNS records that must be maintained for a server to send email. As HubSpot aims to offer as seemless an experience as possible to it's customers, it attempts to take care of as much of the DNS settings as possible on behalf of the customer. Ultimately however, if a customer want's HubSpot to send their marketing emails through an SMTP domain of `emails.company.com`, these DNS records must be obtainable at that domain as discussed in section 3.2.1.

As things change at HubSpot, it is quite likely that these DNS records would need to change over time. For example, if the customer should add another dedicated IP address to their account, this would have to be added to their SPF record. At first glance this would require HubSpot's customers to be frequently changing their DNS records, something most customer's would not be overly comfortable with. The solution to this problem, as with so many problems in computer science, is indirection.

The Domain Name System supports the concept of including other DNS records, even from entirely different domains. This is exactly the behavior that lets HubSpot manage their customers' DNS on their behalf. For example, instead of `emails.company.com` having the SPF record which contains their allowed sending IPs and having to change it should their sending IPs change, they can simply setup this record as a pointer to another DNS record, a record on HubSpot's domain. Clients who require the SPF record for for `emails.company.com` will be informed to use the record contained on HubSpot's domain instead. An example of this is shown in figure 4.1, where 99 is the customer's HubSpot identification number. A similar setup exists for customers' DKIM records.

Thus the customer only ever needs to set up the DNS pointers to HubSpot's domain once. Once that is done, HubSpot can control the actual values of those DNS records on behalf of the customer.



Type	Name	Value
A	news	88.77.66.55
. . .	. . .	. . .
SPF	emails	v=spf1 include 99.spf.hubspot.com -all

Type	Name	Value
TXT	99	v=spf1 ip4:1.2.3.4/30 -all
. . .	. . .	. . .
TXT	500	v=spf1 ipf4:7.7.7.7 -all

Figure 4.1: DNS Includes for *emails.company.com*, HubSpot Customer 99.

Initially, all of the DNS records that the customers include were created at various points in the code. There was no definition for exactly what DNS records should be available and where they should be available for a given account. Some of the DNS record creation was coupled to the code that was responsible for creating new customer accounts, but as time went on, different DNS records were required, resulting in redundant DNS records being created. There were also several cron jobs (jobs which run on a schedule) responsible for checking the current status of the DNS records and attempting to adjust them as necessary. There was also no unit tests for any of the code which synced DNS records. Some of these jobs concluded that different values should exist for the same DNS record, resulting in the jobs competing with each other and updating DNS records every time they ran. It was also unclear what DNS changes were required if accounts were updated. This led to the creation of an entirely new DNS management system.

4.1.1 Requirements for the Management System

Prior to beginning to rebuild the system, an analysis of the current system was required to ensure all of the required functionality contained in the old system would be included in the new system. Another important step was to consider what the *ideal* system would look like. There were two main options to consider:

1. Create all of the DNS records required when the corresponding customer account was created. As discussed previously, it is likely that the value of these DNS records would change over time. Thus, this would also require periodic cron jobs in order to compare the current state of a customer's DNS records against what the system believes their DNS records should be and to update them accordingly. The downside to this approach was having the required DNS records defined and maintained in two separate places - once during account creation and once during the DNS record synchronization.
2. Abstract away all DNS record creation from the account creation process and write the periodic cron jobs in such a way that they can also create brand new DNS records as necessary. The downside of this approach is that there would be a period of time in which the accounts would be created, but have invalid DNS records, which could be problematic if not managed appropriately.

After some consideration, it became clear that option two was the better choice. The period of time in which the accounts would be created but have invalid DNS records could be managed rather simply. Provided the jobs were written correctly, this approach would have the benefit that it could sync DNS records for arbitrarily complex entities. For example, initially the customer's account was all that was required in order to keep their DNS records in sync. However as time went on, other types of DNS records also needed to be maintained, which could not necessarily be deterministically generated from just a customer's account. An example of this is IP addresses which are owned by HubSpot, but not yet in use by any customer. These IP addresses should have specific DNS records in place. However by definition, these IP addresses are not yet assigned to any accounts. Thus the new system should also be able to sync DNS records based on IP addresses. In the past the solution to this problem would have been to simply create yet another periodic cron job to maintain those DNS records, making it even more difficult for an engineer to know where in the code a given DNS record is maintained.

Another important concept are DNS *zones* (sub domains). For easier organization and management of a large number of DNS records, a top level domain (TLD) can be further sub-categorized into zones. In the previous example domain `emails.company.com`, the TLD is `company.com` and a zone on that TLD could be created for `emails`. The purpose of DNS zones is to allow finer grained control for a domain with a large number of DNS records. For example, a customer could setup all of the records required for HubSpot's email marketing on a single zone, the `emails` zone, isolated from all of the domain's other records (e.g. for their website, internal email etc).

The requirements of the new system were determined to be the following:

1. It can sync DNS records for arbitrary entities (Accounts, IP Addresses etc)
2. The code for generating the values of specific DNS records should be decoupled from the code which actually runs the job. This allows for comprehensive unit testing of the code which generates the records.
3. It should only make external requests to update records which are no longer valid.
4. It should group the records required for a given entity by zone, allowing for easier record lookups and updates

4.1.2 Implementation of the System

The first task was to decide how the code which builds the DNS records should be organized. It was imperative to have this code be as clear and concise as possible. The DNS records themselves are hosted on various providers (e.g. Cloudfare), but the Platform team in HubSpot provides a simple client for working with these records (creating records etc). A `DnsRecordBuilder<T>` interface was defined and is shown in appendix A code listing A.1. This interface defines the following methods:

- `Multimap<String, RecordRequest> buildSpfRecords(T entity)`
  - This method will generate the SPF records required for a given entity of type `T` (e.g. an account)
  - Equivalent methods are also defined for MX, A and DKIM records
  - The `RecordRequest` object is a Java object whose contents can be passed to the DnsClient library to create / update / delete the given record
  - These methods return Multimaps (which is essentially a `Map<String, Collection<RecordRequest>>`) which use the record's zone as the key

  - All of these methods are defaulted to returning empty multimaps (if they are not overridden)

- `boolean appliesTo(T entity)`
  - This method specifies whether or not this particular implementation of `DnsRecordBuilder` should apply to this entity
  - For example if the entity was an account, this method can be used to specify that only accounts with dedicated IP addresses should use this `DnsRecordBuilder`

- `Map<String,DnsConfiguration> getDnsConfigByZone(T entity)`
  - The `DnsConfiguration` is a Java object which contains a zone name and a list of `RecordRequest` objects (which contains all of the records to be created)
  - This method has a useful default implementation which aggregates all of the multimaps returned by each methods that builds the records of each type (SPF, DKIM, A, MX) into a single multimap. The returned map of DNS configurations by zone can then be built and returned.
  - This is the most common method called on this interface as it invokes all of the other methods, returning the exact DNS configurations (by zone) needed to keep the given entity up to date.

- `Map<String, String> buildPtrRecords(T entity)`
  - The final method defines the PTR records that should be created for this entity, returning a map containing the IP address as the key and the host name corresponding to this IP address as the value
  - As discussed in section 3.2.1, PTR records are used to perform reverse A lookups, defining the host name associated with an IP address.

Implementations of this interface can then be created for a given entity type. For example, a `DefaultAccountDnsRecordBuilder<Account>` was created to encapsulate all of the records that need to be created for every HubSpot customer account. Similarly, a `UnassignedIpDnsRecordBuilder<IpAddress>` was also created to encapsulate all of the DNS records required for an unassigned IP address. The `IpAddress` object contains the status of the IP address (whether its associated with a customer account or not). The `appliesTo` method of this implementation can then check this field and only return DNS records if the `IpAddress` is in fact unassigned. As time went on, more of these `DnsRecordBuilder` implementations were created, encapsulating different DNS requirements. This provides a very clean and extensible way of defining DNS records which are deterministically built from arbitrary entities.

The next step in the implementation was to write an intermediate class which given an entity of type `T`, applies all of the `DnsRecordBuilder`s corresponding to the entity and finds out which records need updating.

A `Map<Class, List<DnsRecordBuilder>>` is created and is used to determine the list of `DnsRecordBuilder`s that need to be applied to a given entity by simply looking up the entity's class in the map. The `getDnsConfigurationsByZone` method defined above is invoked on each of the builders in the list, generating all of the DNS records that need to be present for the given entity.

Finally another piece of logic was written to determine which records need to be updated for the entity. This is done by checking the records against an in-memory cache, a database cache and finally live DNS. This reduces the number of real DNS queries that need to be made. The in-memory cache and database cache would be frequently invalidated to ensure cached DNS records had not become stale. Any records which require updates are then added to a `List<DnsConfiguration>` and returned to the caller to perform the actual updating. This code segment is provided in appendix A code listing A.2.

The final step was to create the cron job which will use the above code to keep the required DNS records in sync. The HubSpot development platform provides a simple means of registering jobs which are to be run on a schedule. The benefits of the extra complexity of the above code are seen in the implementation of the job. The job simply fetches all of the entities of interest (`Account`s, `IpAddress`es etc) from a database. For each of these entities, the job calls the method from the intermediate class as discussed in the previous paragraph using the entity as a parameter. This method returns the list of records which need be updated or created. The job then handles performing the required DNS updates. A simplified version of the job in which the DNS records for all accounts are synced is shown in appendix A code listing A.3.

4.2 CIDR Minimization Algorithm

As discussed in section 3.2.1, SPF records are a vital part of authorization when it comes to sending emails. SPF records are typically used to specify a set of IP addresses that a particular domain may send emails from. An important aspect of SPF records (or more specifically, the underlying TXT record) is that the length of the entire record value (which is a simple string) should be at most 255 characters as per RFC 7208 (30). Given the fact that a particular HubSpot customer may potentially send email over any one of tens of HubSpot owned IPs, this can cause problems.

One of the upgrades customers can avail of is purchasing dedicated IP addresses, which will be used for their email traffic and theirs only. This allows the customer to build up a good IP reputation without the risk of the reputation being harmed by other HubSpot customers who may send lower quality email. HubSpot owns a large number of IP addresses in order to facilitate this. One of the other tasks undertaken during the course of the internship was automating the process of setting up email sending accounts for new dedicated customers. Prior to this automation, one of the decisions that needed to be made by support staff working with customers was which IP address(es) to assign to customers. Some customers have existing IP addresses and IPs should be selected in order to minimize the length of the resulting SPF record that the customer will have. SPF records support CIDR notation (see section 4.2.1) of IP addresses, meaning smart IP selection can save valuable characters in a customer's SPF record. Thus an important step in automating the setting up of customer accounts with dedicated IP addresses was automating the IP address selection, while still minimizing the resulting SPF records.

4.2.1 CIDR Notation

CIDR (Classless Inter-Domain Routing) notation is a compact way to represent an IP address along with its associated subnet mask and network prefix (31). With regards to SPF records, it is useful as it can be used to compactly represent a set of IP addresses. This set consists of the IP addresses of all of the hosts on the sub-network (subnet) specified by the network prefix. This section will primarily discuss CIDR notation for version four (IPv4) IP addresses, though all of the same logic holds for version six (IPv6) IP addresses.

Typically IP addresses are represented as quartet of period separated integers ranging from 0 - 255, for example, $192.168.1.1$. However, this representation is simply employed in order to make reading IP addresses easier for humans. In actuality, version four IP addresses are more simply represented as 32 bit integers. Each of the numbers in the quartet can take on one of 256 values. Thus

$$\begin{aligned} \log_2 256 &= 8 \textit{ bits per element in quartet} \\ 8 \textit{ bits per element} \times 4 \textit{ elements in quartet} &= 32 \textit{ bits} \end{aligned} \tag{1}$$

$192.168.1.1$ could be represented as a 32 bit integer by using 192 as the upper (most significant) 8 bits, 168 as the next 8 bits and so on. CIDR notation contains the IP address in question, followed by a slash and a number, for example $192.168.1.2/31$. CIDR notation partitions the 32 bit representation of the IP address into two pieces - the upper bits make up the network prefix and the remaining bits are used to specify the specific host on that network. The number following the slash denotes the number of bits to use for the network prefix. Thus $192.168.1.3/31$ specifies that 31 out of the 32 bits should be used for the network prefix and all other bits should be zeroed in order to obtain the network address. This implies that the network address is $192.168.1.2$. This is because the least significant byte of this IP address is 3, or $(0000\ \ 0011)_2$ in binary, and the last bit is to be zeroed meaning the last byte of the network address becomes $(0000\ \ 0010)_2$, which is 2 in base-10.

CIDR notation can thus be used to represent a set of IP addresses, provided they are contiguous. The set has a cardinality that is an integral power of 2 and the first IP address in the set lies on a CIDR boundary. The set of IPs $\{192.168.1.0,\ 192.168.1.1\}$ can be represented using CIDR notation as $192.168.1.0/31$. The logic here is that the address of the subnet containing the hosts of interest is provided and the implied set of IPs is the set of all IP addresses of the hosts on that subnet. Thus if a customer owns these two IP addresses, their SPF record can simply contain the CIDR notation value of the subnet which contains the two IP addresses, reducing the number of characters required by almost half. This is due to the fact that $/31$ implies that there is one bit (the last bit) which identifies the host on the subnetwork defined by $192.168.1.0$. This bit can either be a zero or a one, yielding the two possible IP addresses that were started with - $192.168.1.0$ or $192.168.1.1$.

4.2.2 A Simplified Version of the Algorithm

The main objective of the algorithm is summarized as follows (Note if the CIDR post fix is omitted, $/32$ is implied):

Given a set of existing IP addresses S_e (the *existing* set) and a set of available IP addresses S_a (the *available* set), choose a set of n IP addresses S_c (the *chosen* set) from S_a such that the resulting number of characters in the CIDR representation of S_f (the *final* set) is minimized, where

$$S_f = S_e \cup S_c \tag{2}$$

An example scenario in which the algorithm could be used is given in equation 3

$$\begin{aligned} &S_e = \{1.1.1.1,\ \ 1.1.1.2\} \\ &S_a = \{1.1.1.0,\ \ 1.1.1.3,\ \ 1.1.1.4,\ \ 1.1.1.5\} \\ &n = 2 \end{aligned} \tag{3}$$

In this scenario, the algorithm should result in $S_c = \{1.1.1.0,\ 1.1.1.3\}$, resulting in $S_f = \{1.1.1.0,\ 1.1.1.1,\ 1.1.1.2,\ 1.1.1.3\}$ which is represented in CIDR notation as $S_f = \{1.1.1.0/30\}$. Critically, although the set of IPs $\{1.1.1.1,\ 1.1.1.2,\ 1.1.1.3,\ 1.1.1.4\}$ are contiguous, the CIDR representation of these IPs is $\{1.1.1.1,\ 1.1.1.2/31,\ 1.1.1.4\}$ **not** $\{1.1.1.1/30\}$ as first IP does not lie on a CIDR boundary.

Although the algorithm could likely be brute forced by generating every possible set of IP addresses and choosing the one with the fewest characters in the CIDR representation, this algorithm would run in exponential time making it less than ideal.

In order to attempt to gain some deeper insight into the problem, a common mathematical approach was used in which a simpler version of the problem was considered - the case where there is no existing IPs ($S_e = \{\}$). For convenience, a new variable t is also introduced to represent the total number of IPs in the final set (the cardinality of S_f). Thus:

$$t = |S_f| = |S_e| + n \tag{4}$$

The first step of the algorithm requires determining the largest possible CIDR block that could be obtained for a given t. The number of IPs in a CIDR block is related to the number of bits available for representing the hosts on the subnetwork. Thus, the number of IPs in a CIDR block must always be an integral power of 2. This is shown in table 4.1.

The calculation for the number of host bits n_{hb} is shown in equation 5 where n_{ab} is the number of address bits (the value after the / in the CIDR address)

$$n_{\text{hb}} = 32 - n_{\text{ab}} \tag{5}$$

The calculation for the number of hosts on the subnetwork (n_{hosts}) is given by the number of digits that the number of host bits n_{hb} can represent and is shown in equation 6.

$$n_{\text{hosts}} = 2^{n_{hb}} \tag{6}$$

Example Address	**Number of Host Bits n_{hb}**	**Number of IPs in Block n_{hosts}**
1.1.1.0/32	0	1
1.1.1.0/31	1	2
1.1.1.0/30	2	4
1.1.1.0/29	3	8
...	...	...
1.1.1.0/1	31	2^{31}

Table 4.1: The Size of CIDR Blocks as a Function of n_{hb}

Thus, the largest possible block of CIDR IPs for a given t can be obtained by finding the highest integral power of two that is less than or equal to t. For example, if $t = 10$, then 8 would be the largest possible CIDR block as $2^3 = 8$ but $2^4 = 16$.

An importance concept of the algorithm is assigning each IP address to a certain *bucket*. This assignment process needs to know what bucket size to use. The IPs will be placed into the bucket that represents the subnet that they would be contained within for a given number of host bits. Thus, the bucket sized used will be an integral power of 2 aligning with table 4.1. Logically, the presence of a filled bucket indicates that a CIDR block can be formed from the set of IPs contained in that bucket. An example is shown in figure 4.2 using a bucket size of 4. In this case, bucket $1.1.1.0$ is full, meaning the IPs inside it can be used to create the CIDR block of size 4 $\{1.1.1.0/30\}$.

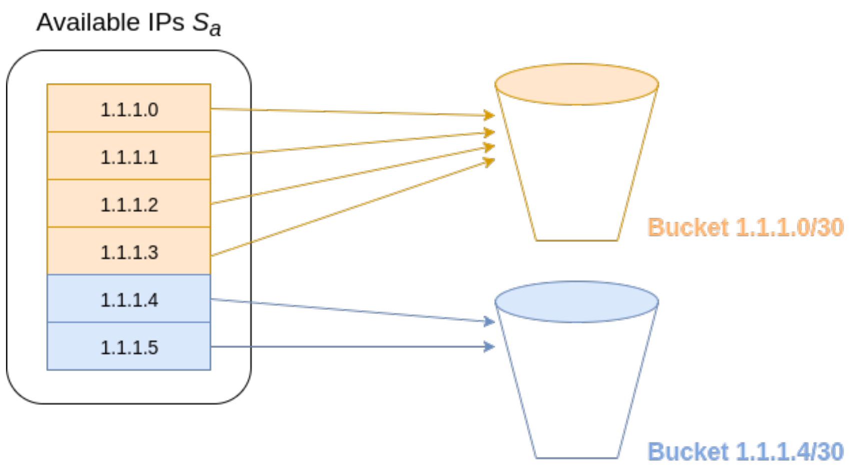


Figure 4.2: Assigning IPs to Buckets with a Bucket Size of 4

As discussed, the bucket size will be an integral power of 2, but an important question is which integral power of 2. The algorithm starts by assigning the IPs to their buckets using a bucket size equal to the largest possible CIDR block that can be obtained for the given t. As discussed previously, this is the largest integral power of 2 that is less than or equal to t.

At this point, the algorithm begins to take shape. Consider the situation represented in figure 4.2 along with a value of $t = 4$. The initial bucket size will be 4 and the presence of a filled bucket $(1.1.1.0/30)$ indicates that a CIDR block of the bucket size can be created. Since the bucket size is equal to the desired number of IPs, the optimal choice is the $1.1.1.0/30$ block. The algorithm can return this block and it will be the optimum set of addresses to choose to minimize the number of CIDR entries required to represent the set of IP addresses.

Next consider the situation represented in figure 4.2 along with a value of $t = 6$. The initial bucket size will still be 4 (as this is the largest integral power of 2 less than or equal to t). Thus, the algorithm will again detect that the $1.1.1.0/30$ bucket is full and select this block of four IPs. However the algorithm must return a total of $t = 6$ IPs and therefore must select a further 2 IPs. The algorithm accomplishes this by making a recursive call. By removing all of the (so far) selected IPs from the set of available IPs (forming S'_a, the *available* set for the next call) and setting the new value of t to be the remaining number of IPs required (t' = 2), a recursive call will simply find the best set of IPs from what is left. This is shown in figure 4.3 in which the recursive call would return $1.1.1.4/31$ as this bucket is full. The selected IPs from each recursive call are then unioned, returning the optimum set of IPs - $\{1.1.1.0/30,\ 1.1.1.4/31\}$

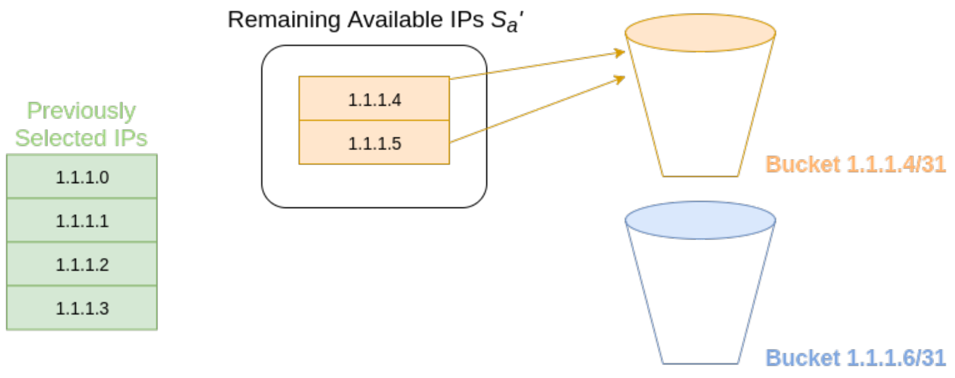


Figure 4.3: Assigning Remaining IPs to Buckets with a Bucket Size of 2

In the case of $t = 7$, the algorithm would proceed as before, with one extra recursive call with $t = 1$. Assuming there were sufficient IPs available (the number of IPs in the diagrams were limited for brevity), the algorithm would have selected the same 6 IPs as before and the final call would be the trivial case of $t = 1$ in which a random IP can be selected. At each stage, the set of IPs returned from the recursive calls can then be unioned with the current call's chosen IPs, and the unioned set returned.

The final possibility to consider is what should be done when no buckets are filled. In this case, the algorithm should not select any IPs at this bucket size. Instead, it should reduce the bucket size to the next highest integral power of 2 and recurse, setting t equal to this reduced bucket size. However, there is another important step, the need for which is best illustrated with an example. A diagram of this situation is shown in figure 4.4

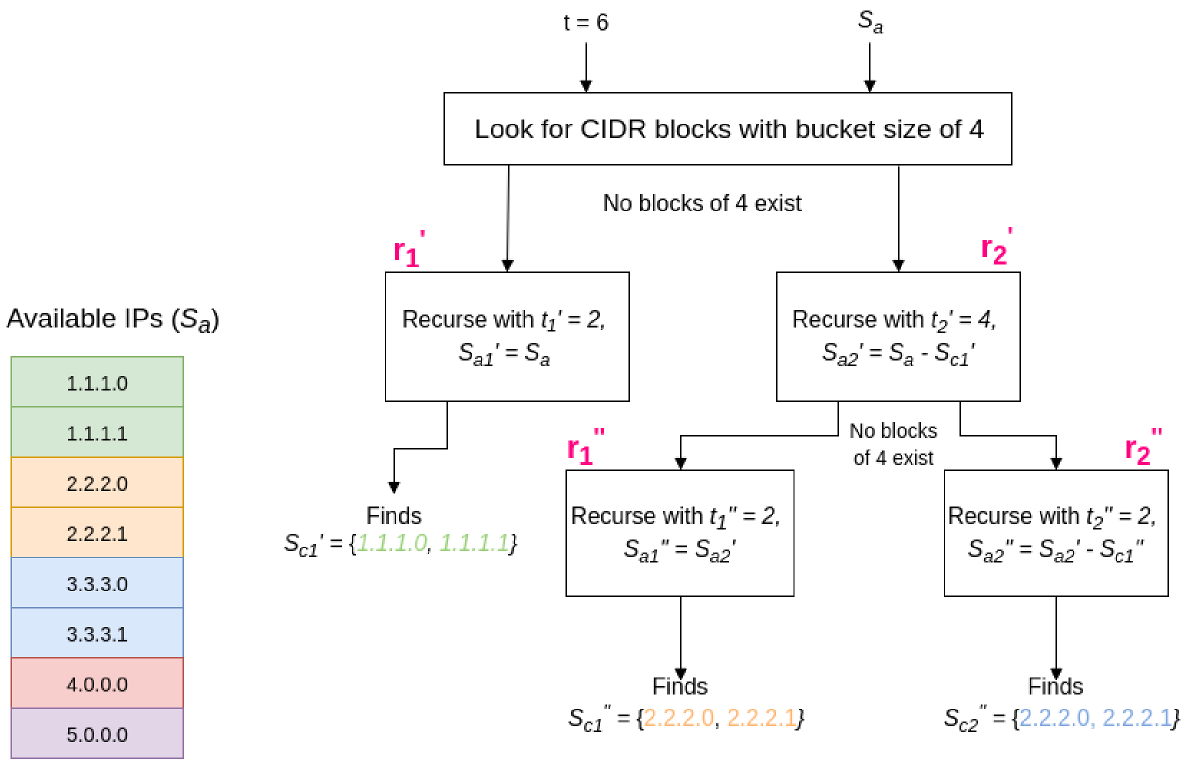
If the initial value for t is 6, the initial bucket size will be 4. If no buckets of size 4 are filled, the algorithm will then recurse. Let this recursive call be denoted as r_1', where $'$ indicates the depth of the recursion. Let the total number of IPs required for this recursive call r_1' be denoted as t_1'. Thus, $t_1' = 2$ (the next largest integral power of 2 as discussed previously) and the set of available IPs is unchanged $S_{a_1}' = S_a$. Let the set of chosen IPs returned from r_1' be denoted as S_{c_1}'.

The initial call required 6 IPs to be returned and r_1' will return the 2 green IPs shown in figure 4.4, $\{1.1.1.0,\ 1.1.1.1\}$. Thus a second recursive call, r_2' is required. This recursive call, r_2' is **not** nested within the first recursive call r_1', but rather is a sibling recursive call (it is at the same depth as r_1'). Let the total number of IPs required for r_2' be denoted as t_2'. t_2' is calculated using the formula given in equation 7. This is simply the remaining number of IPs that need to be chosen after the first recursive call. In this case, $t_2' = 6 - 2 = 4$.

$$t_2' = t - t_1' \tag{7}$$

The set of available IPs that is used for r'_2, S'_{a_2}, is given by equation 8. This is the initial set of all available IPs, with the IPs chosen from the first recursive call omitted. Let the set of IPs returned from r'_2 be denoted as S'_{c_2}.

$$S'_{a_2} = S_a - S'_{c_1} \tag{8}$$



t = 6
S_a
Look for CIDR blocks with bucket size of 4
No blocks of 4 exist
r_1'
r_2'
Available IPs (S_a)
1.1.1.0
1.1.1.1
2.2.2.0
2.2.2.1
3.3.3.0
3.3.3.1
4.0.0.0
5.0.0.0
Recurse with $t_1' = 2$, $S_{a1}' = S_a$
Recurse with $t_2' = 4$, $S_{a2}' = S_a - S_{c1}'$
Finds $S_{c1}' = \{1.1.1.0, 1.1.1.1\}$
r_1''
No blocks of 4 exist
r_2''
Recurse with $t_1'' = 2$, $S_{a1}'' = S_{a2}'$
Recurse with $t_2'' = 2$, $S_{a2}'' = S_{a2}' - S_{c1}''$
Finds $S_{c1}'' = \{2.2.2.0, 2.2.2.1\}$
Finds $S_{c2}'' = \{2.2.2.0, 2.2.2.1\}$

Figure 4.4: Code Path When Algorithm Does Not Find Full Bucket

The second recursive call, r'_2 (as described above), will then look for IPs using a bucket size of 4. Again, none will be found. Thus the algorithm will need to recurse again. Let this call be denoted r''_1. The bucket size for r''_1 will be 2 (the next highest integral power of 2). Thus, r''_1 will find the 2 IPs shown in orange in figure 4.4, $\{2.2.2.0,\ 2.2.2.1\}$. However the second recursive call r'_2 required a total of 4 IPs and only 2 have been selected. Thus just as in the first case, a sibling recursive call, r''_2 is required to choose the remaining IPs. Thus r''_2 will require two more IPs to be chosen and will return the blue set of IPs shown in figure 4.4, $\{3.3.3.0,\ 3.3.3.1\}$. The union of the results of r''_1 and r''_2 will be returned to r'_2 and will contain the required 4 IPs, $\{2.2.2.0/31,\ 3.3.3.0/31\}$. Finally the result of r'_1 and r'_2 will be unioned, returning the desired 6 IPs, $\{1.1.1.0/31,\ 2.2.2.0/31,\ 3.3.3.0/31\}$.

Now that all cases have been considered, the steps of the algorithm (for the simplified case of having no existing IPs) are shown below:

- Set bucket size to be largest integral power of 2 that is less than or equal to t
- Assign all available IPs into buckets using the calculated bucket size
- If a bucket is full
  - Add all of the IPs in the bucket to the set of chosen IPs (S_c)
  - Remove all of the IPs in the bucket from the set of available IPs (forms S'_a)
  - Calculate t' as $t - bucketSize$
  - Recurse using S'_a and t' (if necessary)
- Otherwise
  - Recurse (r'_1) using next biggest integral power of 2 and the initial set of available IPs
  - Recurse again (r'_2) using the remaining number of IPs required and the set of available IPs, excluding those returned from r'_1 (S'_{c_1})
  - Return the union of the results of the two recursive calls.

4.2.3 Extending the Algorithm to Support Existing IPs

Now that a solution to the simplified problem where $S_e = \{\}$ has been formed, the more complex initial problem could now be tackled. However upon closer inspection, there is only one extra point to consider when the set of existing IPs is non empty: **the final set must be a super set of the set of existing IPs**.

The extended algorithm's parameters are changed to the following to aid in the recursion:

- A list of available IPs - S_a
- A list of existing IPs - S_e
- The total number of IPs desired - t
- The bucket size to use - b

The main changes to the algorithm are the following:

- Assign the existing IPs (S_e) to buckets using the provided bucket size
- Assign the available IPs (S_a) to separate buckets using the provided bucket size
- For each of the *existing* IP buckets S_{e_i}

  - Calculate b_i^*, the number of IPs required to fill the bucket (equation 9)
  - If the number of IPs in the corresponding *available* IP bucket (S_{a_i}) equals b_i^*, the union of the two buckets creates a CIDR block and should be added to the currently chosen set of IPs (S_c), as long as it doesn't cause $|S_c|$ to become larger than the total number of desired IPs, t
  - If $|S_c|$ is now equal to t (the total number of IPs required), S_c can be returned.
  - Otherwise, recurse (see below)

$$b_i^* = b - |S_{e_i}| \tag{9}$$

This shows that the only logical change is that the number of IPs required to *fill the existing IP buckets* should be examined, rather than searching for full buckets from the available pool.

As with the simplified case, this method may require some recursion after the above steps. The recursive call in this case is actually simpler, but depends on the current state of the algorithm. As previously mentioned, all of the IPs in the existing set should be contained in the final set. If the algorithm reaches a state where all of the existing IPs have been used, but there is still a number of IPs to be selected, the algorithm can simply fallback to the previous case where $S_e = \{\}$.

Otherwise, the algorithm recurses using the remaining available IPs, the remaining *required* IPs (this is the set of existing IPs that have not yet been chosen), the remaining number of IPs to be chosen and the next bucket size to use which is the next largest integral power of 2.

Consider the following example as shown in figure 4.5.

- $S_e = \{1.1.1.0,\ 2.2.2.0\}$
- $S_a = \{1.1.1.1,\ 1.1.1.2,\ 1.1.1.3,\ 2.2.2.1,\ 2.2.2.2\}$
- $t = 7$

As $t = 7$, the initial bucket size b would be 4 and the IPs would be assigned to buckets as shown in figure 4.5

The algorithm would then proceed in the following manner:

- Examine each existing bucket (S_{e_i}) and the corresponding available bucket (S_{a_i})
  - Calculate b_i^* for $1.1.1.0/30$ as per equation 9, yielding 3

  - Add $1.1.1.0/30$ to S_c as $b_i^* = |S_{a_i}| = 3$
  - Calculate b_i^* for $2.2.2.0/30$ as per equation 9, also yielding 3
  - Skip $2.2.2.0/30$ as $b_i^* > |S_{a_i}|$ $(3 > 2)$

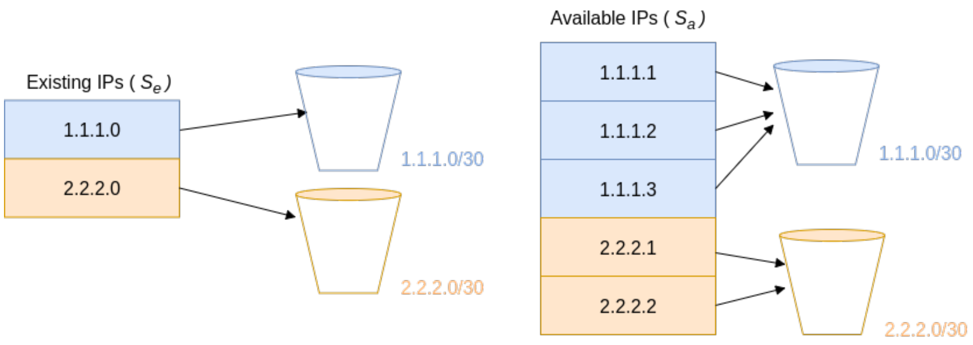


Figure 4.5: Available and Existing IPs Assigned to Buckets of Size 4

Since the total number of IPs required was $t = 7$ and only 4 IPs have been selected, a recursive call is required. As there are still some existing IPs that have not been included, a recursive call is made with the following parameters:

- $S_e' = \{2.2.2.0\}$ - the remaining *required* IPs that have not yet been chosen
- $S_a' = \{2.2.2.1,\ 2.2.2.2\}$ - the remaining *available* IPs
- $t' = 3$ - the remaining *number* of IPs to be selected
- $b' = 2$ - the bucket size should be the next largest integral power of 2 (was 4)

Thus, the IPs will be assigned to buckets of size 2, as shown in figure 4.6.

The algorithm will then proceed as follows:

- Examine each existing bucket (S_{e_i}') and the corresponding available bucket (S_{a_i}')
  - Calculate $b{*}_i'$ for $2.2.2.0/31$ as per equation 9, yielding 1
  - Add $2.2.2.0/31$ to S_c' as $b{*}_i' = |S_{a_i}'| = 1$
  - Skip $2.2.2.2/31$ as $b{*}_i' > |S_{a_i}'| (2 > 1)$

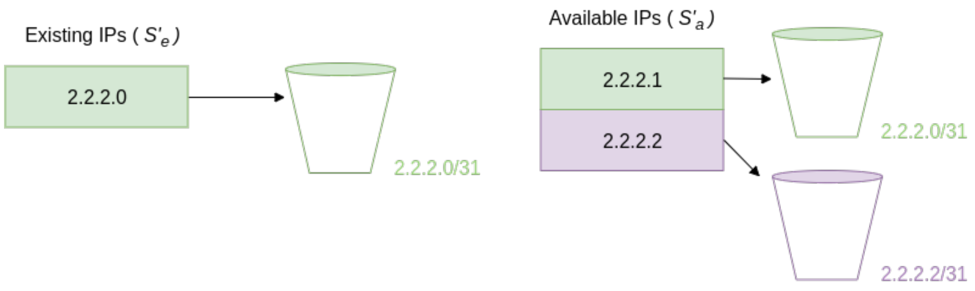


Figure 4.6: Remaining Available and Existing IPs Assigned to Buckets of Size 2

Finally since a total of 7 IPs were required and 6 have now been selected, a final recursive call is required. However all of the existing IPs have now been chosen, meaning the existing IPs argument for the next recursive call will empty. This means the algorithm falls back to the simple case as described earlier and will choose a random final IP (2.2.2.2 in this case).

There are two enhancements that can be made to the algorithm at this point.

1. A better method for choosing a **single** IP from the available IP pool. This shouldn't be done randomly as this could result in large CIDR blocks contained in the available IP pool being broken up unnecessarily. This is described in appendix B.

2. In the case where the existing set of IPs (S_e) is non empty, the previously described method of choosing the highest integral power of 2 that is less than or equal to t for the initial bucket size is not ideal. Given a set of 2 IPs and requiring a total of 4 IPs, the approach would suggest starting with a bucket size of 4. However unless the existing IPs themselves are contiguous, there is no possibility of forming a CIDR block of size 4. This is described in detail in appendix C.

Finally, the entire source code for the algorithm can be found in appendix D

4.2.4 Unit Testing the Algorithm

The final step in the development was to write extensive unit tests to ensure the algorithm operates as expected. Unit testing is an critical part of the software development process at HubSpot and engineers are always encouraged to write code which is as unit testable as possible. As the code is essentially a number of static utility functions, unit testing is quite simple and does not require mocking any complex objects.

The initial task was to come up with an example state to test the algorithm with. The example state contains the set of available IP addresses S_a and the set of existing IP addresses S_e. These should be chosen in such a way that all code paths of the algorithm are seen at some point or another when running the tests. This requires some consideration as for what is contained in S_a and S_e, as well as the total number of IPs required, t, for each of the test cases.

As a starting point, the following sets of IPs were chosen for the default state (this can be changed on a per test basis of course):

$$\begin{gathered} S_a = \{1.1.1.2,\ 1.1.1.3,\ 1.1.1.4,\ 1.1.1.5,\ 1.1.1.6,\ 1.1.1.7,\ 1.1.1.8,\ 1.1.1.9\} \\ S_e = \{1.1.1.0,\ 1.1.1.1\} \end{gathered} \tag{10}$$

Initially however, tests were written for the simplified case where $S_e = \{\}$. The following tests were then implemented:

- The algorithm should choose contiguous blocks when available
  - This was tested by simply requesting that 4 IPs be selected from S_a and ensuring that the resulting set was $\{1.1.1.4/30\}$.
- The algorithm should find contiguous blocks of IPs when a single IP will be left over
  - This was tested by requesting that 5 IPs be selected from S_a
  - The resulting set should contain the block of 4 IP addresses $(1.1.1.4/30)$ and a single other IP
  - This test was added to ensure that the algorithm still selected a block of four instead of trying to find several blocks of 2
- The algorithm should find contiguous IPs recursively

  - This was tested by requesting that 6 IPs be selected from S_a
  - The resulting set should contain the block of 4 IP addresses ($1.1.1.4/30$) as before, but should also select the $1.1.1.2/31$ block for the remaining 2 IP addresses
  - This test was sufficient to assume the algorithm would recursively select CIDR blocks at lower block sizes when available

- The algorithm should always return the correct number of IP addresses
  - This test was implemented by repeatedly invoking the algorithm with t running from 1 to $|S_a|$ and ensuring $|S_c| = t \ \forall \ t$

These tests were sufficient for the simple case where $S_e = \{\}$ and the algorithm was then tested using the more complex case. The following tests were implemented in order to ensure correctness of the algorithm:

- The algorithm should find contiguous blocks when provided with existing IP addresses
  - This was tested by requesting a total of 4 IPs, given the default state as described in equation 10
  - The algorithm should then select $\{1.1.1.2/31\}$ so that when unioned with $S_e = \{1.1.1.0/31\}$, the CIDR block $\{1.1.1.0/30\}$ is formed
- The algorithm should always return the correct number of IPs and contain all of the existing IPs
  - This test was an important test as the algorithm would be useless if it minimized CIDR blocks but didn't guarantee that all of the existing IPs would be present in the resulting set
  - This test was implemented by allowing a variable i to take on values from 1 to $|S_a|$ and by setting $t = |S_e| + i$.
  - This essentially repeatedly runs the algorithm, requesting each possible value from the minimum number of new IPs (1), to the maximum number of new IPs (the number of available IPs)
  - At each iteration, the returned set was checked to ensure the correct number of IPs was returned and that all of the existing IPs were contained in that set.
- The algorithm was also tested to ensure it would look for contiguous blocks both above and below (in the IP address space) the existing IPs provided in order to

try and find CIDR blocks

  - This was tested by requesting a total of 2 IPs and setting $S_e = \{1.1.1.3\}$
  - This tested that the algorithm would find CIDR blocks below the existing IPs provided
  - The algorithm should return $1.1.1.2/31$, indicating it correctly searches below the provided existing IPs
  - A similar test was written where $S_e = 1.1.1.2$ and the algorithm correctly returned $1.1.1.2/31$ indicating that it also looks above the provided existing IP addresses for CIDR blocks

- The algorithm should find CIDR blocks with existing IP addresses that cause a single IP to be left over
  - This was tested by requesting a total of 9 IPs using the default state as shown in equation 10
  - The resulting set should contain the CIDR block of 8 IPs $(1.1.1.0/29)$ and the remaining extra IP $(1.1.1.9)$
- The algorithm should return already grouped existing IPs
  - It is essentially impossible to break up a CIDR block contained inside the set of existing IP addresses, but this test was added to ensure the algorithm correctly identified existing CIDR blocks appropriately.
  - A set of existing IPs $S_e = \{2.2.2.0/31\}$ was used along with an the available set of IPs listed in equation 10 and a total of 4 IPs were requested.
  - The algorithm correctly returned the existing CIDR block as well as another CIDR block of size 2, yielding $\{1.1.1.2/31,\ 2.2.2.0/31\}$
  - This ensured the algorithm selected the existing block of 2 as well as another block of 2, instead of selecting a block of 4 by excluding existing IPs
- The algorithm should handle existing IPs which don't contribute to any CIDR blocks and should fill the remaining number of IPs required with the best CIDR block from the available set of IPs
  - This was tested by setting $S_e = \{2.2.2.0,\ 3.3.3.0,\ 4.4.4.0\}$, using the set of available IPs shown in equation 10 and requesting 5 IPs.
  - The algorithm should then return the three existing IPs along with a CIDR block of size 2 from the set of available IPs

These tests provided a sufficient amount of coverage to provide confidence in the algorithm and its implementation. Several other tests were implemented which were less specific to the algorithm (such as ensuring user input was validated). The algorithm has been running in a production environment for over 3 months and has helped automate an important step that was previously being performed manually by a support team at HubSpot.

4.3 Abstracting Upstream Email Sending to Kafka

While discussing Kafka in section 3.1.4, an idealized case of having all emails that need to be sent through HubSpot simply placed on a Kafka topic was shown. This functionality is supported for emails that can be sent through HubSpot's internal Mail Transfer Agent (MTA) (see section 1.3 for more details), but is not supported for emails using other third party MTAs. The reason for this was HubSpot's email marketing was initially built using third party MTAs, prior to the development of the internal MTA. Prior to this task, email sent through third party MTAs was done *in-process*, inside of an upstream email team's deployable, as all of this code was written prior to the formation of the Email Sending Infrastructure team. This code is entirely to do with email sending and thus should certainly be owned and controlled by the Email Sending Infrastructure team. It also adds a considerable amount of latency to the upstream email team's code, as they must wait for the results of email sends.

The main goal of this task was to decouple the email sending code from the upstream teams deployable. Kafka was the perfect choice to use as the interface between the two teams and the desired setup was for the upstream team to produce the emails to be sent to a given Kafka topic, for those emails to be sent by the Email Sending Infrastructure team's Kafka consumers and for the results of the email send to be produced back to the upstream email team through a second Kafka topic.

The major difficulty associated with rewriting the code in this way was that the results of the email send were no longer obtainable in-process of the upstream teams deployable. The email sending would essentially happen asynchronously and the results would be present in a **different** deployable. Ultimately this is the exact behavior that is desired as the process producing these emails to Kafka should be able to *"fire and forget"* - once it has produced the email to Kafka, it has done it's job.

The main difficulty associated with this task was implementing the solution in a *backwards compatible* way. All of HubSpot's outgoing marketing email would be running through this new code path. Thus it must be rolled out extremely slowly and carefully. This meant the code changes must be made in such a way that either the old in-process email sending, or the new Kafka email sending could be used.

This task was essentially two tasks which were entirely dependent on one and other. One of these tasks was creating the infrastructure required on the Email Sending Infrastructure team's side of things and the other was making use of this infrastructure on the upstream email team's side of things.

4.3.1 The Email Sending Infrastructure Team's Side

This task required the creation of an entirely new project owned by the Email Sending Infrastructure team. This project would now contain all of the code required for sending the emails to both the internal MTA and the third party MTAs. This required substantial research into setting up and configuring Kafka and setting up a project on the HubSpot stack.

As discussed, all changes had to be backwards compatible. This was achieved by writing a new higher level abstraction of a class which sends emails. To the upstream team, they would simply obtain a reference to an object which had a `produceEmailRequest` method. Under the hood, there were several implementations of an interface which contained this method. One of these implementations provided access to the old in-process email senders, while the other implementations provided access to the new Kafka producers which would produce these emails to the new Kafka topic. This abstraction was provided to the to the upstream team, allowing the Email Sending Infrastructure team to control *under what circumstances* and *when* the new Kafka sending pipeline could be used. For example, the Kafka senders could be used a configurable number of times per minute, for particular types of email sends, or disabled entirely.

This extra consideration and layer of abstraction was key in providing a mechanism for safely rolling out changes which could potentially break **every** email that was to be sent through HubSpot.

4.3.2 The Upstream Email Team's Side

The upstream email team in this case were the team responsible for managing all of the data generated around an email to be sent. In this circumstance, their main task was performing all of the required work that was to be done post send attempt. This included writing the email to a variety of places, producing events which lead to customer billing, tracking the number of sends a particular customer had made and many others.

The main issue at this point was some of this *post-send* logic required the **result** of the email send. However as the main purpose of this task was to decouple the email send from their process and reduce latency, by definition, the results of email sends were no longer available. This required careful splitting out of the *post-send* logic into what could be performed in-process and what could not. Any of the logic which was entirely dependent on the result of the email send had to be moved into a Kafka consumer, listening to the results of the email sends, which were produced by the Email Sending Infrastructure team's Kafka consumers who were responsible for sending the email.

Though the task may seem trivial, this was actually one of the most difficult tasks undertaken throughout the course of the internship. A monumental amount of research and consideration was required in order to ensure all edge cases were covered and that the new Kafka sending pipeline would be stable. As discussed, every marketing email sent through HubSpot would run through this code and this code was also responsible for billing customers, so any bugs could have proved extremely costly.

5 Conclusion

In conclusion, the internship provided a fantastic insight into how software is developed in an enterprise environment and provided ample opportunity to learn new concepts and extend knowledge on previously studied topics.

Over the course of the internship, experience was gained working with a variety of new and exciting technologies such as Kafka (section 3.1.4) and Hadoop (section 3.1.5). Knowledge gained from previous modules of the course on fundamental computer science principles, problems and techniques (for example immutability, dependency injection, databases and distributed systems) was put into practice and used to deliver industry standard, well tested software that is used millions of times per day in a production environment.

The internship also provided an opportunity to practice and improve upon softer skills such as time management, time estimation for software development tasks, hosting and contributing to meetings with other engineers and reviewing code submitted by teammates. Although modules aiding in developing these fundamental industry skills are available throughout the MAI program, having the chance to use these skills on a daily basis provides an excellent means of improving these skills in a real enterprise environment.

Bibliography

[1] Inc. HubSpot. What is hubspot?, . https://www.hubspot.com/what-is-hubspot.

[2] Inc. GitHub. Github enterprise - how businesses build software. https://enterprise.github.com/home.

[3] Linus Torvalds. Git. https://git-scm.com/.

[4] CA Technologies. Waffle - developer-first project management for teams on github. URL `waffle.io`.

[5] Slack. Slack for enterprise. https://slack.com/.

[6] Timothy Beneke and Tori Wieldt. Javaone 2013 review: Java takes on the internet of things. 2013. www.oracle.com/technetwork/articles/java/afterglow2013-2030343.html.

[7] Oracle. Java 8 executor documentation, . docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html.

[8] Oracle. Java 8 forkjoinpool documentation, . docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html.

[9] Tomasz Nurkiewicz. Java 8 definitive guide to completablefuture. www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html.

[10] Google. Google's guice dependency injection framework, . github.com/google/guice.

[11] Google. Google's guice dependency injection framework docs, . github.com/google/guice/wiki/Motivation.

[12] Wikipedia. Factory method pattern wiki post. en.wikipedia.org/wiki/Factory_method_pattern.

[13] Open Source Project under Apache License 2.0. Immutables java project homepage. immutables.github.io/.

[14] Apache Software Foundation. Apache kafka homepage, . kafka.apache.org.

[15] Eric W Weisstein. "highly composite number." from mathworld–a wolfram web resource. mathworld.wolfram.com/HighlyCompositeNumber.html.

[16] Inc. Amazon Web Services. Amazon s3. https://aws.amazon.com/s3/.

[17] The Apache Software Foundation. Apache hadoop, . http://hadoop.apache.org/.

[18] The Apache Software Foundation. Hadoop distributed file system architecture guide, . https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[19] Google. Protocol buffers, . https://developers.google.com/protocol-buffers/.

[20] Google. Protocol buffer developers guice, . https://developers.google.com/protocol-buffers/docs/overview.

[21] Inc. Google. grpc homepage, . https://grpc.io/.

[22] Inc. Google. grpc overview page, . https://grpc.io/docs/guides/concepts.html.

[23] Return Path. Dkim signature header detail. https://help.returnpath.com/hc/en-us/articles/222438487-DKIM-signature-header-detail.

[24] Jonathan B. Postel. Rfc 1982, simple mail transfer protocol, 1982. https://tools.ietf.org/html/rfc821.

[25] J. Klensin. Rfc 1869, smtp service extensions, 1995. https://tools.ietf.org/html/rfc1869.

[26] Inc. HubSpot. Niosmtpclient github page, . https://github.com/HubSpot/NioSmtpClient.

[27] The Apache Software Foundation. Log4j2 - java logging framework, . URL `logging.apache.org/log4j/2.x`.

[28] Inc. Functional Software. Sentry - error tracking software. URL `sentry.io`.

[29] SignalFx. Signalfx: Cloud monitoring for the enterprise. URL `signalfx.com`.

[30] Kitterman Technical Services S. Kitterman. Rfc 7208, sender policy framework (spf). tools.ietf.org/html/rfc7208.

[31] Spiceworks Technology Writer Susan Adams. What is cidr notation? (adapted). https://community.spiceworks.com/networking/articles/2495-what-is-cidr-notation.

Appendices

App. A Source Code for DNS Management System

```java
public interface DnsRecordBuilder<T> {
  default Multimap<String, RecordRequest> buildDkimRecords(T entity) {
    return ArrayListMultimap.create();
  }

  default Multimap<String, RecordRequest> buildARecords(T entity) {
    return ArrayListMultimap.create();
  }

  default Multimap<String, RecordRequest> buildMxRecords(T entity) {
    return ArrayListMultimap.create();
  }

  default Multimap<String, RecordRequest> buildSpfRecords(T entity) {
    return ArrayListMultimap.create();
  }

  default Map<String, String> buildPtrRecords(T entity) {
    return Collections.emptyMap();
  }

  boolean appliesTo(T entity);

  default Map<String, DnsConfiguration> getDnsConfigurationByZone(T
      entity) {
    Map<String, DnsConfiguration> dnsUpdatesByZone = new HashMap<>();
    Multimap<String, RecordRequest> allRecordRequestsByZone =
        ArrayListMultimap.create();

    allRecordRequestsByZone.putAll(buildDkimRecords(entity));
    allRecordRequestsByZone.putAll(buildARecords(entity));
    allRecordRequestsByZone.putAll(buildMxRecords(entity));
    allRecordRequestsByZone.putAll(buildSpfRecords(entity));

    for (String zone : allRecordRequestsByZone.keySet()) {
      dnsUpdatesByZone.put(zone, DnsConfiguration.builder()
          .setZone(zone)
          .setRecords(allRecordRequestsByZone.get(zone))
          .build());
    }

    return dnsUpdatesByZone;
  }
}
```

Code Listing A.1: The `DnsRecordBuilder` Interface

```java
public <T> List<DnsConfiguration> buildHubSpotDnsRecordUpdatesForZones(T
    entity) {
  List<DnsConfiguration> hubSpotDnsChanges = new ArrayList<>();
  Set<DnsRecordBuilder<T>> buildersToApply =
      getDnsRecordBuildersToApplyOrThrow(entity);

  for (DnsRecordBuilder<T> dnsRecordBuilder : buildersToApply) {
    Map<String, DnsConfiguration> requiredConfigByZone =
        dnsRecordBuilder.getDnsConfigurationByZone(entity);
    for (Entry<String, DnsConfiguration> entry :
        requiredConfigByZone.entrySet()) {
      String zone = entry.getKey();
      DnsConfiguration config = entry.getValue();
      List<RecordRequest> changedHubSpotDomainRecords =
          dnsClientHelper.findRecordsToUpdate(config.getRecords(), zone);
      hubSpotDnsChanges.add(config.withRecords(changedHubSpotDomainRecords));
    }
  }

  return hubSpotDnsChanges;
}

private <T> Set<DnsRecordBuilder<T>> getDnsRecordBuildersToApplyOrThrow(T
    entity) {
  if (!recordBuildersByClass.contains(entity.getClass())) {
    throw new IllegalArgumentException(String.format("Invalid type given:
        %s", entity.getClass()));
  }

  return recordBuildersByClass.get(entity.getClass());
}
```

Code Listing A.2: Apply `DnsRecordBuilder`s and Find Records That Need Updating.

```java
public class DnsSyncJob extends AbstractJob {
  private final AccountsDao accountsDao;
  private final DnsSyncJobHelper dnsSyncJobHelper;
  private final DnsClientHelper dnsClientHelper;

  @Inject
  public DnsSyncJob(AccountsDao accountsDao,
                    DnsSyncJobHelper dnsSyncJobHelper,
                    DnsClientHelper dnsClientHelper) {
    this.accountsDao = accountsDao;
    this.dnsSyncJobHelper = dnsSyncJobHelper;
    this.dnsClientHelper = dnsClientHelper;
  }

  @Override
  public void run() {
    Collection<Account> accountsToSync = accountsDao.getAllAccounts();
    syncDnsForEntities(accountsToSync);

    // Similar code for Ip addresses and other entities..
  }

  private <T> void syncDnsForEntities(Collection<T> entities) {
    for (T entity : entities) {
      checkEntity(entity);
    }
  }

  private <T> void checkEntity(T entity) {
    List<DnsConfiguration> dnsConfigurationsForZones =
        dnsSyncJobHelper.buildDnsRecordUpdatesForZones(entity);
    for (DnsConfiguration dnsConfiguration : dnsConfigurationsForZones) {
      for (RecordRequest recordRequest : dnsConfiguration.getRecords()) {
        dnsClientHelper.upsertRecord(recordRequest,
            accountDnsConfiguration.getZone());
      }
    }
  }
}
```

Code Listing A.3: A Simplified DNS Sync Job Implementation

App. B CIDR Min. Algorithm - Single IP Choice

When selecting a single IP from the available pool, ideally, the algorithm should choose an *awkward* IP that does not have any free neighboring IPs. Otherwise, the algorithm may break up valuable CIDR blocks by choosing randomly.

This enhancement to the algorithm was added by examining the set of available IPs and attempting to find a bucket containing a single IP. A list of IPs denoted as `remainingIps` is initialized to contain the same IPs as the `availableIps` set. An initial bucket size was defined and the available IPs were assigned to buckets at this size. The bucket with the smallest number of IPs was then selected. If the number of IPs in this bucket was 1, this IP would be returned. Otherwise, the `remainingIps` list was set to the IPs in this bucket and the process would be repeated with a smaller bucket size.

This allows IPs that *may* form CIDR blocks to be kept available for cases when more than 1 IP is requested.

The code for the function is shown in code listing B.1

Note for the sake of the report, the size of the bucket was used for explanatory purposes. However the code makes frequent use of the number of host bits. These are related through equation 1

$$bucketSize = 2^{numHostBits} \tag{1}$$

```java
/**
 * Tries to find an IP that will not break up a contiguous block. This
    works by recursively finding the bucket with the fewest IPs.
 * This repeats until a single IP is found
 * It starts by splitting IPs into buckets using 2 ^
    AWKWARD_IP_STARTING_NUM_HOST_BITS as the initial bucket size
 * If it fails to find one at this block size, it uses the IPs from the
    bucket with the fewest IPs in it,
 * reduces the block size and tries again, ultimately falling back to a
    random IP
 */
static String findSingleAwkwardIp(List<String> availableIps) {
  List<String> remainingIps = new ArrayList<>(availableIps);
  int numHostBits = AWKWARD_IP_STARTING_NUM_HOST_BITS;

  while (true) {
    Multimap<Integer, String> ipsByBucket = getIpsByBucket(remainingIps,
        getMask(numHostBits));

    // Find bucket with least number of IPs
    Collection<String> ipsInBucketWithFewestIps =
        ipsByBucket.asMap().values().stream()
        .sorted(Comparator.comparingInt(Collection::size))
        .findFirst().get();

    if (ipsInBucketWithFewestIps.size() == 1) {
      return ipsInBucketWithFewestIps.iterator().next();
    }

    numHostBits--;
    remainingIps.clear();
    remainingIps.addAll(ipsInBucketWithFewestIps);
  }
}
```

Code Listing B.1: Finding a Single Awkward IP

App. C CIDR Min. Algorithm - Initial Bucket Size

The previously described method for choosing the initial bucket size was choosing the highest integral power of 2 that is less than or equal to t. This method only makes sense for the case where $S_e = \{\}$.

For example, given $S_e = \{1.1.1.0,\ 2.2.2.0\}$ and $t = 4$ the algorithm must choose two IPs ($n = 2$). The described method would suggest starting with a bucket size of $b = 4$. However it would be impossible to generate a CIDR block of size 4 given the existing IPs and a choice of only two IPs, **regardless of the available IPs**.

This extension is handled by a separate function which is called prior to starting the algorithm, to determine the appropriate starting bucket size. This function only requires the total number of IPs desired (t) and the existing set of IPs (S_e) as parameters.

The function starts by assigning IPs to buckets using the largest integral power of 2 less than or equal to t rule. It then calculates n (the number of IPs that need to be chosen from the available pool) and checks if any of the existing buckets require n **or less** IPs to be filled. As only n or less IPs can be selected, if no buckets can be *filled* by selecting n or less IPs, there is no possibility of creating CIDR blocks at this bucket size. This is done without considering the available IPs. Thus even if a bucket *could* be filled by choosing n IPs, it may not be. If no buckets can be filled choosing n or less IPs, the loop continues, checking the next largest bucket size. Otherwise, the function returns the current bucket size.

Note for the sake of the report, the size of the bucket was used for explanatory purposes. However the code makes frequent use of the number of host bits. These are related through equation 1

$$bucketSize = 2^{numHostBits} \tag{1}$$

```java
/**
 * Find the theoretical best case bucket size given a list of existing ips and a total number of ips
 * Eg [54.174.58.0, 54.174.58.100] and a total size of 4:
 * There is no point in starting with a bucket size of 4 as even if we had all the IPs in the world,
 * we cant create a /30 block with those two existing IPs, so start with /31
 */
static int findStartingNumberOfHostBits(List<String> requiredIps, int totalIps) {
  int numHostBits = getMaxNumberOfHostBits(totalIps);

  while (true) {
    int maxBucketSize = getNumberOfHosts(numHostBits);
    int mask = getMask(numHostBits);
    int newIpsToCreate = totalIps - requiredIps.size();

    Multimap<Integer, String> currentIpsByBucket = getIpsByBucket(requiredIps, mask);

    // Ensure that at least one of the potential buckets could be filled by creating newIpsToCreate IPs
    // If we're not adding enough IPs to potentially fill the bucket, theres no point
    if (currentIpsByBucket.asMap().values().stream().anyMatch(ips -> maxBucketSize - ips.size() <= newIpsToCreate)) {
      LOG.debug("Starting with bucket size of {}", numHostBits);
      return numHostBits;
    } else {
      LOG.debug("No point in checking a bucket size of {}", numHostBits);
      numHostBits--;
    }
  }
}
```

Code Listing C.1: Choosing the Initial Bucket Size

App. D CIDR Min. Algorithm - Source Code

```java
public class IpAddressUtils {
  private static final Logger LOG =
      LoggerFactory.getLogger(IpAddressUtils.class);
  private static final int AWKWARD_IP_STARTING_NUM_HOST_BITS = 3;

  public static List<String> getContiguousIpsWithExistingIps(List<String>
      availableIps, List<String> requiredIps, int totalNumIps) {
    return requiredIps.size() == 0 ? getContiguousIps(availableIps,
        totalNumIps)
        : doGetContiguousIpsWithExistingIps(availableIps, requiredIps,
            totalNumIps, findStartingNumberOfHostBits(requiredIps,
            totalNumIps));
  }

  private static List<String>
      doGetContiguousIpsWithExistingIps(List<String> availableIps,
      List<String> existingIps, int totalNumIps, int numHostBits) {
    if (numHostBits == 0) {
      List<String> bestIps = new ArrayList<>(existingIps);
      bestIps.addAll(getContiguousIps(availableIps, totalNumIps -
          existingIps.size()));
      return bestIps;
    }

    int maxBucketSize = getNumberOfHosts(numHostBits);
    int mask = getMask(numHostBits);

    Multimap<Integer, String> requiredIpsByBucket =
        getIpsByBucket(existingIps, mask);
    Multimap<Integer, String> availableIpsByBucket =
        getIpsByBucket(availableIps, mask);

    List<String> ipsToUse = new ArrayList<>();
    Set<String> requiredIpsRemaining = new HashSet<>(existingIps);

    for (Entry<Integer, Collection<String>> entry :
        requiredIpsByBucket.asMap().entrySet()) {
      int bucket = entry.getKey();
      List<String> requiredIpsInBucket = new ArrayList<>(entry.getValue());

      Collection<String> availableIpsInBucket =
          availableIpsByBucket.containsKey(bucket) ?
          availableIpsByBucket.get(bucket) :
          Collections.emptyList();

      int requiredNumIpsToFillBucket = maxBucketSize -
          requiredIpsInBucket.size();

      // If the existing ips will fit into this bucket with the new ips
      if (availableIpsInBucket.size() == requiredNumIpsToFillBucket
          && totalNumIps >= ipsToUse.size() + maxBucketSize) {
        ipsToUse.addAll(availableIpsInBucket);
        ipsToUse.addAll(requiredIpsInBucket);
        requiredIpsRemaining.removeAll(requiredIpsInBucket);
        if (ipsToUse.size() == totalNumIps) {
          return ipsToUse;
        }
      }
    }

    // Ensure no duplicate ips will be used in subsequent searches
    List<String> availableIpsForNextSearch = new ArrayList<>(availableIps);
    availableIpsForNextSearch.removeAll(ipsToUse);

    List<String> finalIps = new ArrayList<>(ipsToUse);
    int numIpsRemaining = totalNumIps - ipsToUse.size();

    // All existing ips have been used, get the best IPs we can for the rest
    if (requiredIpsRemaining.isEmpty()) {
      finalIps.addAll(getContiguousIps(availableIpsForNextSearch,
          numIpsRemaining));
      return finalIps;
    }

    finalIps.addAll(doGetContiguousIpsWithExistingIps(availableIpsForNextSearch,
        new ArrayList<>(requiredIpsRemaining), numIpsRemaining, numHostBits
        - 1));
    return finalIps;
  }

  public static List<String> getContiguousIps(List<String> availableIps,
      int totalNumIps) {
    if (totalNumIps == 1) {
      return Collections.singletonList(findSingleAwkwardIp(availableIps));
    }

    int numHostBits = getMaxNumberOfHostBits(totalNumIps);
    int maxBucketSize = getNumberOfHosts(numHostBits);

    // Find number of ips that would overflow this block
    int extraIpsNeeded = totalNumIps - maxBucketSize;
    int mask = getMask(numHostBits);

    // Try find a contiguous block
    Multimap<Integer, String> ipsByBucket = getIpsByBucket(availableIps, mask);
    int ipsNeededThisBlockSize = totalNumIps - extraIpsNeeded;
    List<String> bestIps = new ArrayList<>();
    for (Entry<Integer, Collection<String>> entry : ipsByBucket.asMap().entrySet()) {
      if (entry.getValue().size() == ipsNeededThisBlockSize) {
        bestIps.addAll(entry.getValue());
        break;
      }
    }

    // Ensure we don't find same ips at next call
    List<String> availableIpsForNextSearch = new ArrayList<>(availableIps);
    availableIpsForNextSearch.removeAll(bestIps);

    if (bestIps.isEmpty()) {
      LOG.debug("Couldn't find a contiguous block of size {}, trying smaller block size.", maxBucketSize);
      int numIpsForFirstSearch = getNumberOfHosts(numHostBits - 1);
      int numIpsForSecondSearch = totalNumIps - numIpsForFirstSearch;

      List<String> bestIpsOne = getContiguousIps(availableIpsForNextSearch, numIpsForFirstSearch);
      bestIps.addAll(bestIpsOne);

      if (numIpsForSecondSearch > 0) {
        availableIpsForNextSearch.removeAll(bestIpsOne);
        List<String> bestIpsTwo =
            getContiguousIps(availableIpsForNextSearch, numIpsForSecondSearch);
        bestIps.addAll(bestIpsTwo);
      }
    } else if (extraIpsNeeded != 0) {
      LOG.debug("{} Extra IPs were required after block size {}, trying next block size", maxBucketSize);
      bestIps.addAll(getContiguousIps(availableIpsForNextSearch,
          extraIpsNeeded));
    }

    return bestIps;
  }

  static String findSingleAwkwardIp(List<String> availableIps) {
    List<String> remainingIps = new ArrayList<>(availableIps);
    int numHostBits = AWKWARD_IP_STARTING_NUM_HOST_BITS;

    while (true) {
      Multimap<Integer, String> ipsByBucket = getIpsByBucket(remainingIps,
          getMask(numHostBits));

      // Find bucket with least number of IPs
      Collection<String> ipsInBucketWithFewestIps =
          ipsByBucket.asMap().values().stream()
          .sorted((x, y) -> x.size() > y.size() ? 1 : -1)
          .findFirst().get();

      if (ipsInBucketWithFewestIps.size() == 1) {
        return ipsInBucketWithFewestIps.iterator().next();
      }

      numHostBits--;
      remainingIps.clear();
      remainingIps.addAll(ipsInBucketWithFewestIps);
    }
  }

  @VisibleForTesting
  static int findStartingNumberOfHostBits(List<String> requiredIps, int
      totalIps) {
    int numHostBits = getMaxNumberOfHostBits(totalIps);

    while (true) {
      int maxBucketSize = getNumberOfHosts(numHostBits);
      int mask = getMask(numHostBits);
      int newIpsToCreate = totalIps - requiredIps.size();

      Multimap<Integer, String> currentIpsByBucket =
          getIpsByBucket(requiredIps, mask);

      // Ensure that at least one of the potential buckets could be
      // filled by creating newIpsToCreate IPs
      // If we're not adding enough IPs to potentially
```

```
      // fill the bucket, theres no point
      if (currentIpsByBucket.asMap().values().stream().anyMatch(ips ->
          maxBucketSize - ips.size() <= newIpsToCreate)) {
        LOG.debug("Starting with bucket size of {}", numHostBits);
        return numHostBits;
      } else {
        LOG.debug("No point in checking a bucket size of {}", numHostBits);
        numHostBits--;
      }
    }
  }

  private static int getMaxNumberOfHostBits(int numIps) {
    int tailBits = 0;
    int toShift = numIps;
    while (toShift >= 2) {
      tailBits ++;
      toShift = toShift >> 1;
    }
    return tailBits;
  }

  private static int getNumberOfHosts(int numHostBits) {
    return (int) Math.pow(2, numHostBits);
  }

  private static int getMask(int numHostBits) {
    return toBigEndian(0xffffffff << numHostBits);
  }

  private static Multimap<Integer, String> getIpsByBucket(List<String>
      ips, int mask) {
    Multimap<Integer, String> ipsByBucket = ArrayListMultimap.create();
    for (String ipString : ips) {
      int ip = getIntFromIp(ipString);
      int bucket = ip & mask;
      ipsByBucket.put(bucket, ipString);
    }

    return ipsByBucket;
  }

  private static int getIntFromIp(String ip) {
    int value = 0;
    String[] pieces = ip.split("\\.");
    for (int i = 0; i < pieces.length; i++) {
      int piece = Integer.valueOf(pieces[i]);
      value += piece << i * 8;
    }

    return value;
  }

  private static int toBigEndian(int littleEndian) {
    byte[] b = ByteBuffer.allocate(4).putInt(littleEndian).array();
    return
        ByteBuffer.wrap(b).order(ByteOrder.LITTLE_ENDIAN).asIntBuffer().get();
  }
}
```

Code Listing D.1: CIDR Minimization Algorithm Source Code