Contents

List of Figures

List of Tables

1. Introduction (5 Pages)
    1.1. Who are HubSpot - 2
    1.2. The Email Sending Infrastructure Team - 2 (team structure (what a TL is), what they all do, Hubspot small team methodology, cross team work)
    1.3. What the team owns (products)
    1.4. Goals of the team or something - 1
2. HubSpot Methodology (10 Pages)
    2.1. Java MicroServices (unit, integ, accept)- 5
    2.2. React Native / Redux front end - 4
    2.3. Team Process
        2.3.1. Task Management
        2.3.2. Ops Review (System Health)
        2.3.3. Code Walkthroughs
        2.3.4. Inter Team Coordination
3. Technologies Used (26 pages, probably excessive)
    3.1. Java 8 Features (streams, CF, lambdas, Executors) - 5
    3.2. Kafka - 5
    3.3. Hadoop - 3
    3.4. DNS - 3
    3.5. SMTP - 2
    3.6. MySql (InnoDB) - 3
    3.7. HBase (Sync, Idempot, Locks) - 3
    3.8. ZooKeeper (+ Circus) - 2
    3.9. gRPC + Proto
4. Major Projects Undertaken (20)
    4.1. Rebuilt DNS management - 5
    4.2. Auto DNS for dedicated accounts - 5
    4.3. EmailMtaSending Kafka stuff - 5
    4.4. Minimize CIDR IP selection alg - go into detail about this (eg talk about SPF records in email, CIDr notation and diagram out the alg, performance etc)- 3 pages

CHECK THIS - I double we need these headings.. There are a number of chapters that you must have: an introduction; a background or literature review chapter; and

a conclusion chapter. The focus of the other chapters will depend on your specific project.

Introduction

What is HubSpot?

Inbound Marketing All hubspot products

Email at HubSpot

Teams, cusomter POV etc etc

EmailSendingInfra

Stuff, dedicated IPs, poster would probably be useful

Team Management and Team Process

As with any type of team, it is critical that the team is managed appropriately. Blah blah blah..

Issue Tracking

Issue tracking through GitHub, cross team colab, closing issues with PRs,

Task Managemenet

TODO: Link this with above issue tracking 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. 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 amongst the team about how best to tackle the task.

Ready
: Tasks which are sufficiently specified that they can 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.

Communication

Company wide slack, channel per team, support channels, searchable.

Proactive Ops Reviews

Code Walkthroughs

Critical Situation Post Mortums

Daily Standups

1 to 1s

Might not include this

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 soley dedicated to providing easy to use integrations of powerful tools which can be combersome 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.

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.

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 (5). 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. Objects can then be used inside of other classes, abstracting away all of the complex implementation logic of the class in use from the user. 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, coinsisting 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 user 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*. There functional interfaces were simple interfaces which provided a contract containing a single method. The class 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.

Listing 3.1: onClick Listener without Lambda Expression

```
1  public class Button {
2     private OnClickRunner onClickRunner;
3     private String buttonName;
4
5     public Button(OnClickRunner onClickRunner,
6                   String buttonName) {
7        this.onClickRunner = onClickRunner;
8        this.buttonName = buttonName;
9     }
10
11    public void click() {
12       onClickRunner.runOnClick(buttonName);
13    }
```

```
14
15      public static void main(String[] args) {
16          Button button = new Button(new ButtonClickLogger(), "SEARCH");
17          button.click();
18      }
19  }
20
21  interface OnClickRunner {
22      void runOnClick(String buttonName);
23  }
24
25  class ButtonClickLogger implements OnClickRunner {
26      @Override
27      public void runOnClick(String buttonName) {
28          System.out.println("Button " + buttonName + " was clicked.");
29      }
30  }
```

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 our `OnClickRunner` 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.

Listing 3.2: onClick Listener with Lambda Expression

```
1   public class Button {
2       private Consumer<String> onClickConsumer;
3       private String buttonName;
4       public Button(Consumer<String> onClickConsumer,
5                     String buttonName) {
6           this.onClickConsumer = onClickConsumer;
7           this.buttonName = buttonName;
8       }
9
10      public void click() {
11          onClickConsumer.accept(buttonName);
12      }
13
14      public static void main(String[] args) {
15          Button button = new Button(
16                  name -> System.out.println("Button " + name + " was clicked"),
17                  "SEARCH"
18          );
19
20          button.click();
21      }
22  }
```

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 Java collection of objects. 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 shows two approaches to a piece of code which returns the length of each of the strings (in ascending order) without whitespace 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 recipie, 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.

Listing 3.3: Batch Processing without Streams

```
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;
```

Listing 3.4: Batch Processing with Streams

```
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());
```

Completable Futures

Asynchronus programming is present in most if not all 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 challening 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 nescesity 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 asynchronus 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` (6) 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 submited to Java's work stealing `ForkJoinPool` (7).

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. A subset of these methods are the following (8):

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 asynchronus (that is, it too returns a completable future). This behaviour could technically also be handled by /javathenApply (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 supplied function is asynchronus.

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.

Guice - Dependency Injection

Google's Guice (9) 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 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 (10)) 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.

Listing 3.5: Pizza Ordering Service with no Dependency Injection

```
1  public class PizzaOrderingService {
2    private SqlTransactionLogger transactionLogger;
3    private AccountsDao AccountsDao;
4    private PayPalCustomerBiller customerBiller;
5
6    public PizzaOrderingService() {
7      SqlCredentials sqlCredentials = new SqlCredentials("dbName",
           "password");
8      SqlConnection sqlConnection = new SqlConnection(sqlCredentials);
9      PayPalCredentials payPalCredentials = new
           PayPalCredentials("accountId", "password");
10     this.transactionLogger = new SqlTransactionLogger(sqlConnection);
11     this.accountsDao = new AccountsDao(sqlConnection);
12     this.customerBiller = new PayPalCustomerBiller(payPalCredentials);
13   }
14 }
```

Traditionally, this problem was somewhat helped (but not entirely solved) by the using the factory pattern (11). 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 thee concrete class' dependencies. However this still requires the manul implementation of the factory classes themselves, leaving much to be desired.

Dependency injection solves this problem by allowing fully formed instances of dependencies (eg a `CustomerBiller`) 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 classes 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`. 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.

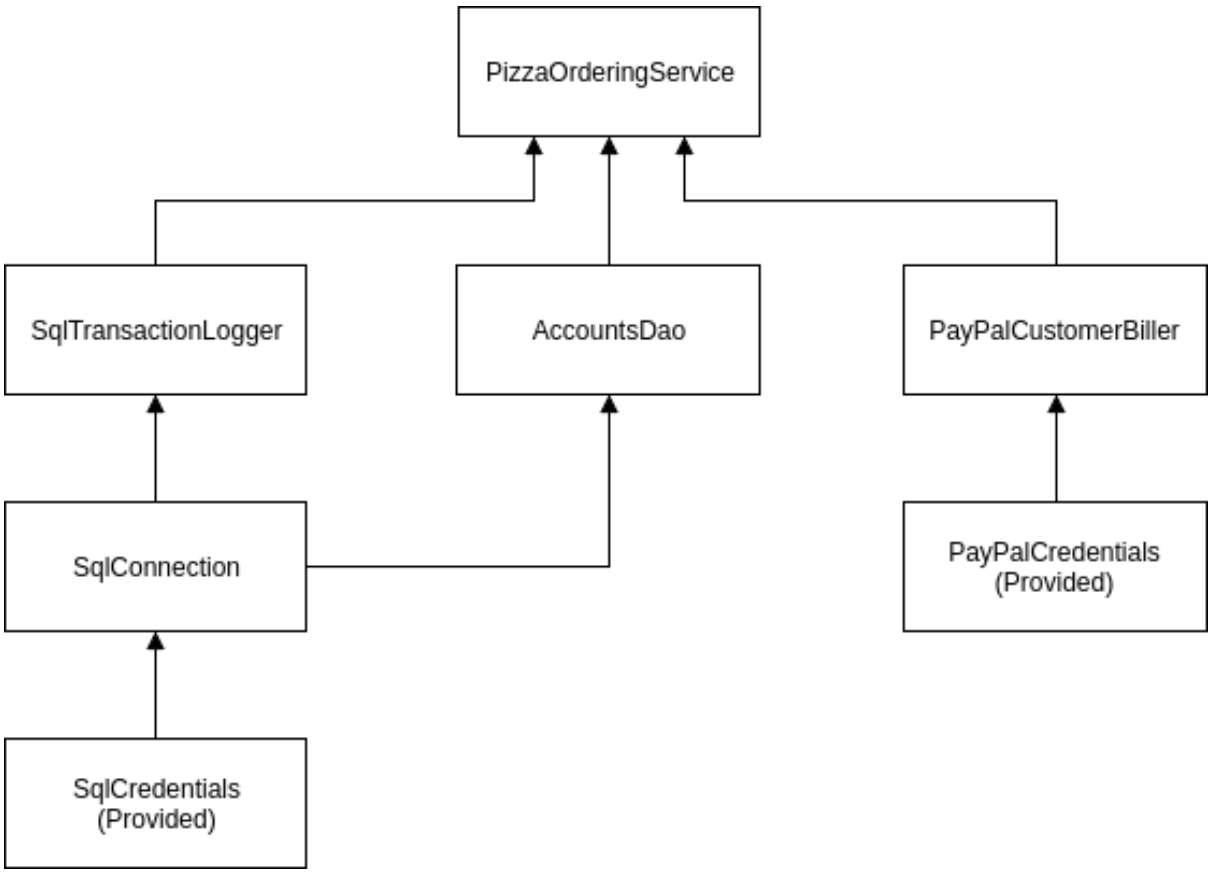

Figure 3.1: Dependency Graph built by Guice

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

Listing 3.6: Pizza Ordering Service using Dependency Injection

```
1  public class PizzaOrderingService {
2    private SqlTransactionLogger transactionLogger;
3    private PayPalCustomerBiller customerBiller;
4    private AccountsDao accountsDao;
5
6    @Inject
7    public PizzaOrderingService(SqlTransactionLogger transactionLogger,
8                                PayPalCustomerBiller customerBiller,
9                                AccountsDao accountsDao) {
10     this.transactionLogger = transactionLogger;
11     this.customerBiller = customerBiller;
12     this.accountsDao = accountsDao;
13   }
14 }
```

Aside from abstracting away the dependency instantiation, dependency injection also has the added benefit in that all of a classes dependencies become part of the classes 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), a mock of this object (which skips the actual billing, but returns results as if it actually billed a customer) can be provided to the class and used for testing purposes.

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 a user. For obvious reasons, this table should not store the user's password (in case of real login attempts), so this should be stripped before logging it 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.

Listing 3.7: An Example LoginRequest

```
1  class LoginRequest {
2    String username, password, ip;
3
4    public LoginRequest(String username, String password, String ip) {
5      // Initialize fields..
6    }
7
8    public void setPassword(String password) {
9      this.password = password;
10   }
11 }
```

Listing 3.8: An Example LoginAttemptLogger Implementation

```
1  class LoginAttemptLogger {
2    LoginRequestDao loginRequestDao;
3
4    public LoginAttemptLogger(LoginRequestDao loginRequestDao) {
5      this.loginRequestDao = loginRequestDao;
6    }
7
```

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

However this style of code is a recipie 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.

Listing 3.9: Perfectly Reasonable Client Login Code

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

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 extremelty 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 avoid the issue caused by the side effect of `logLoginAttempt`.

Listing 3.10: Logically Correct Login Code with Extra Boilerplate

```
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;
}
```

The Immutables (12) provides a framework for autogenerating 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 inter-

face (annotated with `@Value.Immutable`) which contains the getter methods for each desired field 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.

Listing 3.11: Interface Used to Define LoginRequest using Immutables Framework

```
1 @Value.Immutable
2 interface LoginRequestIf {
3   String getUsername();
4   String getPassword();
5   String getIp();
6 }
```

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 exstensively at HubSpot and is a major contributor to the simplicity of writing code without bugs at the company.

Listing 3.12: The LoginAttemptLogger Method using the Immutables Framework

```
1 public void logLoginAttempt(LoginRequest loginRequest) {
2   // Don't log the users password to the database
3   loginRequestDao.writeToTable(loginRequest.withPassword(""));
4 }
```

Kafka - Streaming Platform

Kafka (13) is a horiztonally 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 exstensively 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 exstensively inside of the team's internal pipeline, but also as an interface between teams. For example, the teams responsible for building out and rendering 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 the records published to these topics and can perform the send of the email. This entirely decouples the work done by teams by simply requiring teams to put messages onto Kafka to be handled elsewhere. An obvious alternative to using Kafka would be to simply expose a REST endpoint in 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 workers 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 (14). 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. 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 paritition 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.

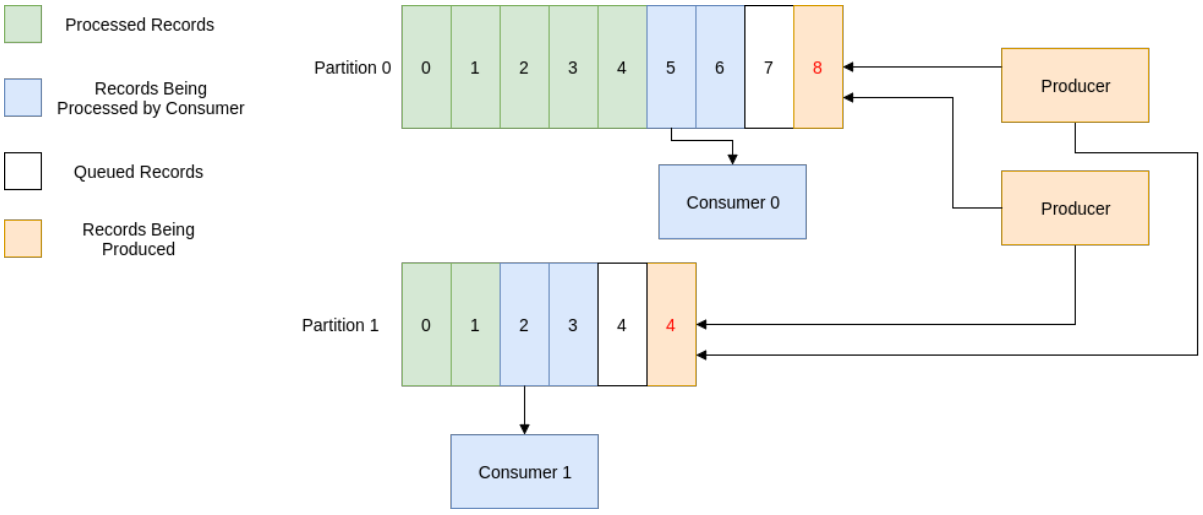

Figure 3.2: Kafka Producers and Consumers

The current situation is outlined as follows:

- Consumer 0 has successfully processed (and commited) messages 0 to 4 and is currently processing messages 5 and 6
- Consumer 1 has successfully processed (and commited) messages 0 and 1 and is currently processing messages 2 and 3

- Message 7 in partition 0 and message 4 in partition 1 have both been successfully produced and commited to the log
- Message 8 in partition 1 and message 4 in parition two will be the next messages produced by the producers.

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 1 and 2 in figure 3.2 would be 4 and 1 respectively.

An 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 0 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 workers (running different code) need to read from the same Kafka topic. Both sets of workers should be able to process every message that is published to the Kafka and this is what consumer groups provides. Without consumer groups, another worker 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 workers). The solution is to specify the consumer group that each running worker belongs to. In this case, since there are two sets of workers, 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 workers reading from the Kafka topic. This feature is best understood by considering the two edge cases:

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

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 each customer for each email that appears on the topic. This is also shown in figure 3.3

SENDING This consumer group would contain all of the Email Sending Infrastructure team's consumers. There would likely be a lot of consumer's 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 would be time critical and thus performance and efficiency would be critical. 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 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.

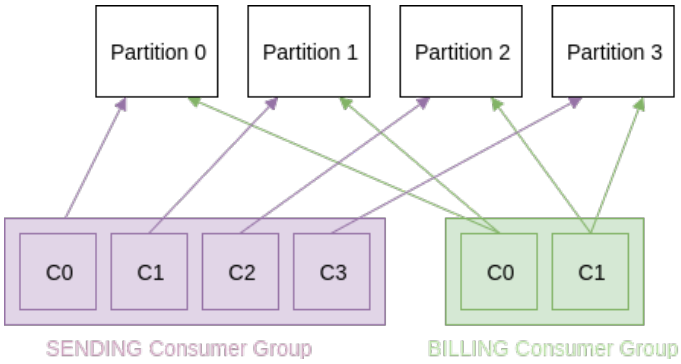

Figure 3.3: Partition Assignments with Multiple Consumer Groups

A final note of interest regarding Kafka is that it provides and *at-least-once* guarantee that messages will arive 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.

MySql

Indexes, liquibase, InnoDB

HBase

ZooKeeper (Circus)

Hadoop

gRPC and Protobuf

Email Specific Technologies

DNS

SMTP

System Management and Health Monitoring

Should this be in technologies used, focus this on how it helps the team manage the system 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 critcal to have systems in place which monitor the health of the system and inform the team of any potential issues with the system. Whatsmore, these issues must be continuously examined and remedial action must be taken where applicable. The Email Sending Infrastructure team made use of several tools and methods for monitoring the health of their systems, a subset of which are outlined below:

Log4j2(1)

Log4j2 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 (eg DEBUG) which describe the state of the system or the code path taken. Typically when the system is behaving normally, a highger priortiy log level is set (eg 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 debug mode which can log error messages as uncaught exceptions without killing the currently executing thread.

Sentry(2)

Sentry 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 occuence and other pieces of meta

data such as the name of the deployable. It uses this data to monitor the occurences 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 (eg the Email Sending Infrastructure 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 the aforementioned Log4j2. Sentry can monitor ERROR level log messages that are produced by Log4j2 and subsequently log these error messages to sentry. 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 enforms the team that an error has occured by logging an exception to Sentry.

SignalFX(3)

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.

Rebuilt DNS Management System

Auto DNS for dedicated IPs

EmailMtaSending Kafka

CIDR Minimization Algorithm

MAKE SURE THIS MAKES SENSE WRT DNS, DISCUSS STRUCTURE OF SPF
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 (15). 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. As discussed in section 1.3, one of the upgrades customers can avail of is purchasing dedicated IP addresses, which will be used for their email traffic and theirs only. HubSpot owns a large number of IP addresses in order to facilitate this. 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 4.4.1) of IP addresses, meaning smart IP selection can save valuable characters in a customer's SPF record. As HubSpot is moving towards automating the setting up of customer accounts with dedicated IP addresses (see section 4.2), this IP address selection needed to be automated, while still minimizing the resulting SPF records.

CIDR Notation

CIDR (Classless Inter-Domain Routing) notation is a notation for compactly representing sets of IP addresses. 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

$$\log_2 256 = 8 bits per element in quartet \tag{1}$$

$$8 bits per quartet \times 4 elements in quartet = 32 bit \tag{2}$$

$192.168.1.1$ could be represented as a 32 bit integer by using 192 as the most 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 netowrk. The number following the slash denotes the number of bits to use for the network prefix. Thus $192.168.1.3/31$ specifies that all but the last bit should be used for the network prefix, implying that the address to reach the subnet this host is inside of is $192.168.1.2$. This is because the least significant byte of this IP address is 3 (11_2) and the last bit is to be zeroed, meaning the last byte of the network address is 2 (10_2).

CIDR notation can thus be used to represent a set of IP addresses, provided they are contiguous. The ultimate goal is to represent a set of IP address in as few characters as possible. 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 resulting set of IPs is the set of all IP addresses of the hosts on that subnet. Thus if a customer owns those two IP addresses, their SPF record can simply contain the CIDR notation equivalent of 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 $192.168.1.1$.

High Level Description of the Algorithm

The main objective of the algorithm is sumarised as follows (Note if the CIDR postfix is ommited, $\backslash 32$ is impied):

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 of the CIDR representation of the final set of IP addresses S_f is minimized, where

$$S_f = S_e \cup S_c \tag{3}$$

An example scenario in which the algorithm could be used is given in equation 4

$$\begin{aligned} S_e &= \{1.1.1.1, 1.1.1.2\} \\ S_a &= \{1.1.1.0, 1.1.1.3, 1.1.1.4, 1.1.1.5\} \\ n &= 2 \end{aligned} \tag{4}$$

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 $S_f = \{1.1.1.0/31\}$. 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 most compact representation of these IPs in CIDR notation is $\{1.1.1.1, 1.1.1.2/31\}$.

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:

$$t = |S_f| = |S_c| + n \tag{5}$$

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 6 where n_{ab} represents the number of address bits (the value after the $\backslash$)

$$n_{\text{hb}} = 32 - n_{\text{ab}} \tag{6}$$

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 7.

$$n_{\text{hosts}} = 2^n_{\text{hb}} \tag{7}$$

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 lower than t. For example, if $t = 10$, then 8 would be the largest possible CIDR block as $2^3 = 8$ and $2^4 = 16$.

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. Critically,

the bucket sized used will be an integral power of 2 aligning with the above table. 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. 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 4.1. In this case, bucket $1.1.1.0$ is full and the bucket size is 4, meaning the IPs inside it can be used to create a CIDR block of size 4 $1.1.1.0 \setminus 30$.

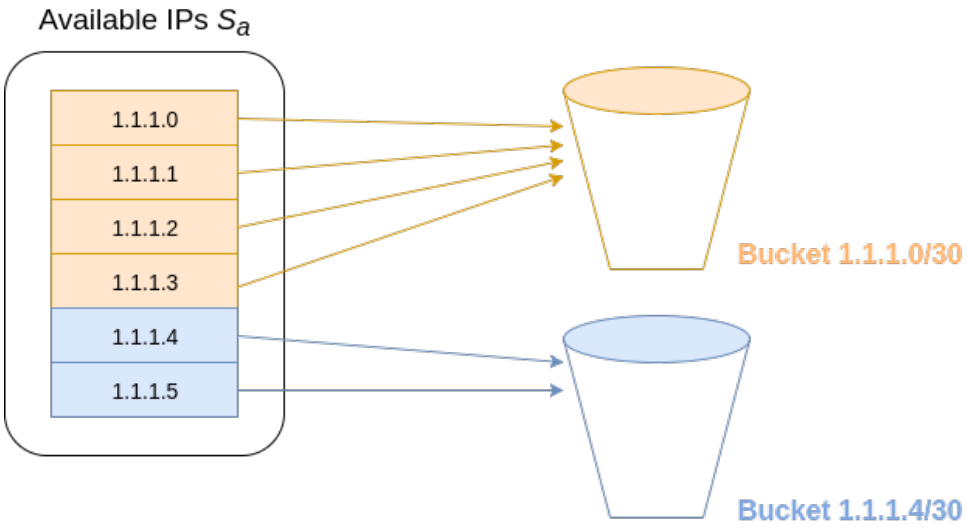

Figure 4.1: Assigning IPs to Buckets with a Bucket Size of 4

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.

At this point, the algorithm begins to take shape. Consider the situation represented in figure 4.1 along with a value of $t = 4$. The inital 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.

Next consider the situation represented in figure 4.1 along with a value of $t = 6$. The initial bucket size will still be 4 (as this is the largest integral power of 2 smaller than

Table 4.1: The Number of IPs in a CIDR Blocks

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
...	...	...

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 recusrive call. By removing all of the (so far) selected IPs from the set of available IPs (forming the S'_a) and setting the new value of t to be the remaning number of IPs required (t'), a recursive call will simply find the best set of IPs from what is left. This is shown in figure 4.2 in which the recusrive call would return $1.1.1.4/31$ as this bucket is full.

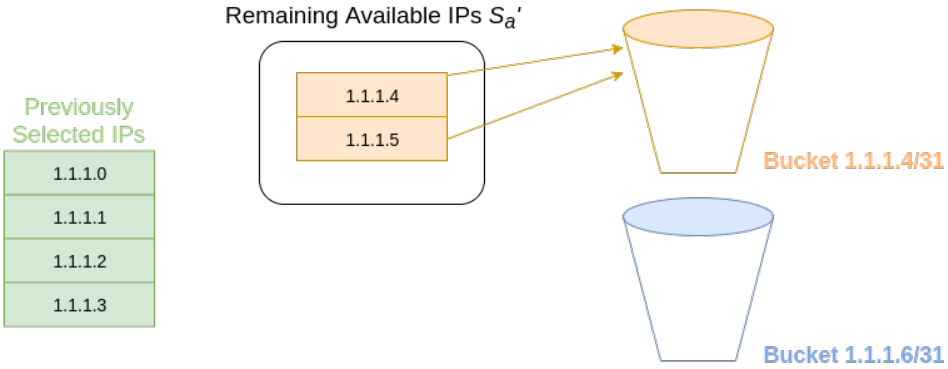

Figure 4.2: Assigning Remaining IPs to Buckets with a Bucket Size of 2

In the case of $t = 7$, the algorithm would proceed as before, with one extra recusrive 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 recusrive 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 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 code path is shown in figure 4.3

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 the total number of IPs requied that is used for this recusrive call be denoted as t'_1. Thus, for the recursive call, $t'_1 = 2$ (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 this first recursive call be denoted as S_{c_1}.

The initial call required 6 IPs to be returned, however this first recursive call will always return 2 IPs that should be used. Thus a second recursive call is required. Let the total number of IPs required that is used for this second recursive call be denoted as t'_2. t'_2 is calculated using the formula given in equation 8. This is simply the remaining number of IPs that need to be chosen after the first recursive call.

$$t'_2 = t - t'_1 \tag{8}$$

Finally, the set of available IPs that is used for the second recursive call (S_{a_2}) is given by equation 9. This is the initial set of all available IPs, with the IPs chosen from the first recursive call ommited.

$$S_{a_2} = S_a - S_{c_1} \tag{9}$$

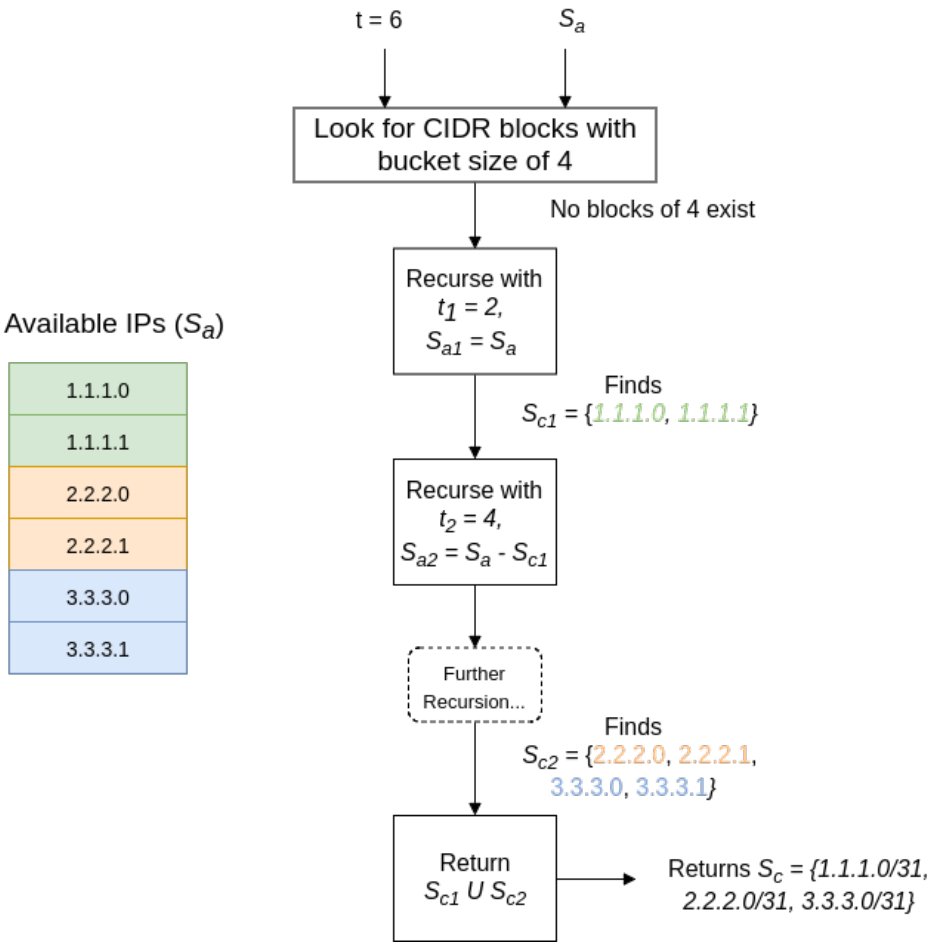

Figure 4.3: Code Path when Algorithm Does Not Find Full Bucket

The steps of the algorithm (for the simplified case of having no existing IPs) are shown in 4.4.2:

- Set bucket size to be largest integral power of 2 that is less than 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 neccessary)
- Otherwise

- Recurse using next biggest integral power of 2 and the initial set of available IPs
- Recurse using the number of remaining IPs required and the set of available IPs, excluding those returned from the previous recursive call
- Return the union of the results of the two recursive calls.

Implementing the Algorithm

Unit Testing the Algorithm

Bibliography

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

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

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

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

[5] 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.

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

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

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

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

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

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

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

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

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

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

Listings

Appendix

You may use appendices to include relevant background information, such as calibration certificates, derivations of key equations or presentation of a particular data reduction method. You should not use the appendices to dump large amounts of additional results or data which are not properly discussed. If these results are really relevant, then they should appear in the main body of the report.

Appendix numbering

Appendices are numbered sequentially, A1, A2, A3... The sections, figures and tables within appendices are numbered in the same way as in the main text. For example, the first figure in Appendix A1 would be Figure A1.1. Equations continue the numbering from the main text.