Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 12345

Förderungswürdigkeit der Förderung von Öl

Lars K.

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Uwe Fessor
Betreuer/in:	Dipl.-Inf. Roman Tiker, Dipl.-Inf. Laura Stern, Otto Normalverbraucher, M.Sc.
Beginn am:	5. Juli 2013
Beendet am:	5. Januar 2014
CR-Nummer:	I.7.2

Kurzfassung

…... Short summary of the thesis ...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Introduction

2 Related Work

3 Fundamentals

??

This chapter discusses concepts that help to understand the content of this thesis. We will start with the basics of cloud computing, its service models and then how one of the service models is related to IBM Bluemix and Cloud Foundry.

Cloud computing, also some-times called as „the cloud“ enables the user to consume on-demand compute resources as a utility over Internet. It is an Internet-based practice where remote servers are used for storage, computation and other on-demand services. Some of the main advantages of cloud computing are pay per use and flexibility. The organization pays for only the amount of time it has used the service and not for the entire duration when the server was running. Organizations can scale up and down as per their requirement and can save a lot of their investment in maintaining local infrastructure. Cloud Computing is divided into three service models-IaaS, SaaS,PaaS 3.1:

1. Infrastructure as a service(IaaS): The providers of this kind of service provides computational resources and storage resources through virtual environments. For eg., AWS, developed by Amazon, provides virtual server instances and application programming interface(API's) for computation and storage.
2. Platform as a service(PaaS): The provider of this kind of service provides a platform for development on their own infrastructure. For eg., Bluemix, developed by IBM is a PaaS service which supports many different languages like Java, Nodejs, Go, swift, Python etc. and services. It has built in DevOps to build, run, deploy and manage applications on cloud. We will discuss in detail about this service later in this chapter.
3. Software as a service(SaaS): In this model software applications are centrally hosted. It removes the need for organizations to install maintain and run applications locally. For eg., Google applications.

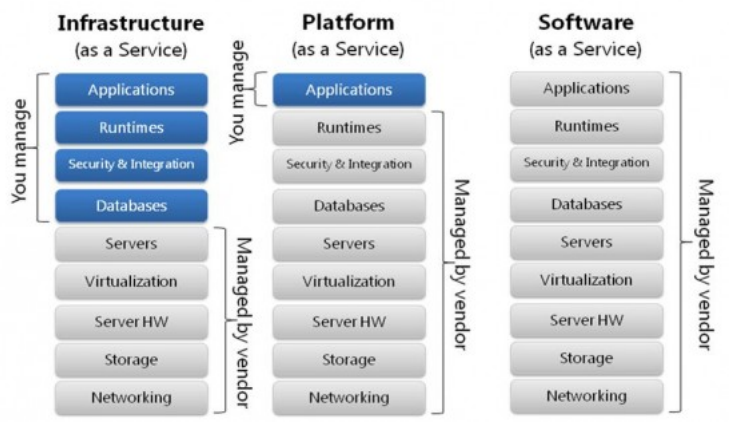

Abbildung 3.1: Types of Service Models [Kat13]

Bluemix [Ang14] Bluemix is an implementation of IBM's Open Cloud Architecture, based on cloud foundry. It lets user to create, deploy, run and manage cloud applications.Since it is based on cloud foundry,it supports more services and frameworks in addition to the services and frameworks supported by cloud foundry. It provides an additional dashboard where the user can create, view and manage applications or services and view resource usage of the application.

3.1 Cloud Foundry

Cloud Foundry is a platform as a service that can be either deployed on private infrastructure or on public IaaS like AWS, Softlayer etc. It is an Open source project and is not vendor specific. It makes deployment and scaling of applications easier with the existing tools without making any change to the code.It provides the following options [Ang14]:

1. Cloud: Developers and organizations can choose to run Cloud Foundry in Public, Private, VMWare and OpenStack-based clouds.
2. Developer Framework: Cloud Foundry supports Java code, Spring, Ruby, Node.js, and custom frameworks.
3. Application Services: Cloud Foundry offers support for MySQL, MongoDB, PostgreSQL, Redis, RabbitMQ, and custom services.

3.1.1 Architecture

This section deals with the architectural functioning of cloud foundry. In diagram 3.2, we can see the architecture of cloud foundry on a physical machine.The outer box represents an physical

machine with a host operating system and processes like terminal,virtual box etc. The Blue box represents a Bosh Lite Director. A Bosh lite [Marb] is a vagrant virtual machine that includes a Bosh server(Director). Bosh is a tool used to deploy cloud foundry.Bosh needs the application to be packaged in a specific form called CF Release. For this, release needs to have all the source files, configuration files, manifest file etc. For deploying cloud foundry a machine must have 8GB of RAM and 100GB of disk space. To deploy Cloud foundry [Clo17a] first we upload stem cells. Stem cells are versioned image of minimum operating system skeleton,wrapped with IaaS specific packaging.We create a Cloud Foundry release after the stem cells and upload the generated release to Bosh director.After the upload a deploy command is run and cloud foundry is deployed. If the cloud foundry is deployed successfully "bosh vms"command provides an overview of all the virtual machines and the status of virtual machines should be "running".In Figure 3.2 the red blocks represents the virtual machines started by Bosh.

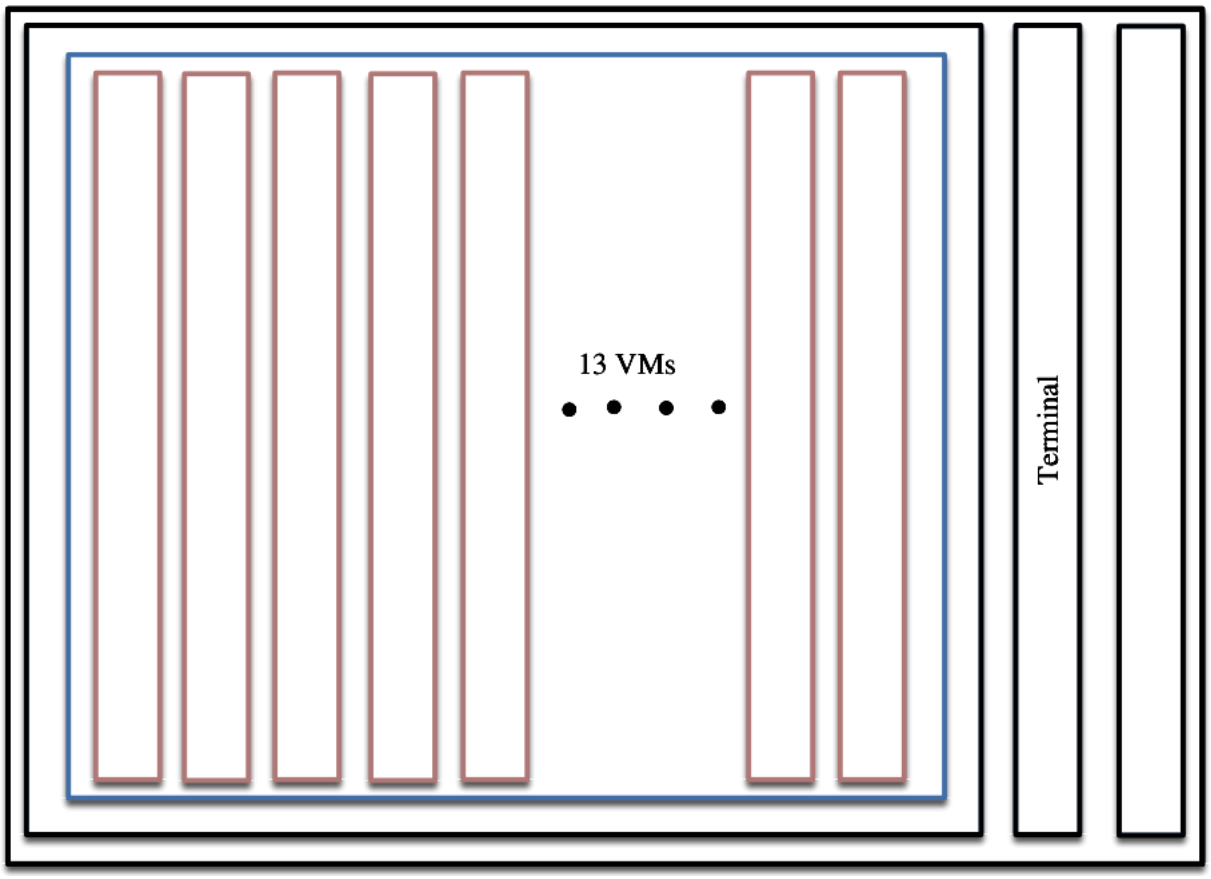

Abbildung 3.2: Architecture of Cloud Foundry

3.1.2 Components

[Clo17b] Cloud Foundry has self-service application execution engine, an automation engine for application deployment and lifecycle management, and a scriptable command line interface (CLI), as well as integration with development tools to ease deployment processes. It has an open architecture that includes a buildpack mechanism for adding frameworks, an application services interface, and a cloud provider interface. Following diagram 3.3 shows the major components of Cloud Foundry:

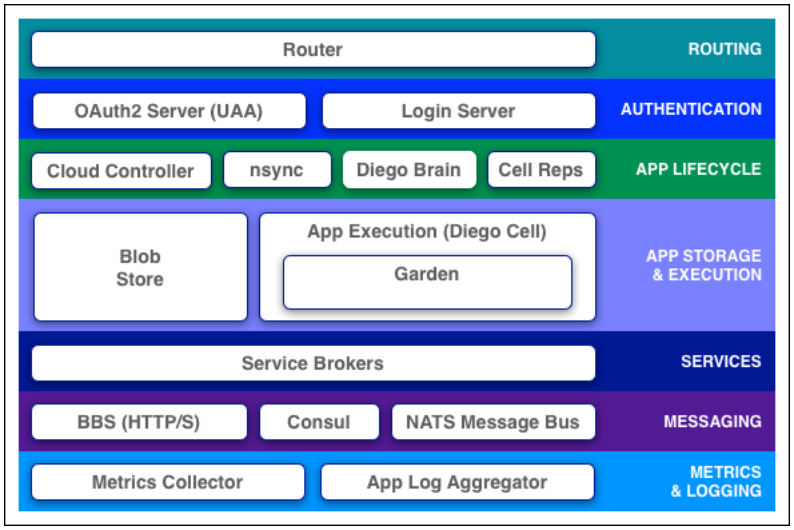

Abbildung 3.3: Components of Cloud Foundry [Clo17b]

This section [Clo17b] describes the working of each component of Cloud Foundry depicted in diagram 3.3. Th router routes all traffic that is coming in to appropriate components that is either the Bits service(Cloud Controller in this case) or to the application that is running on Diego. Router queries Diego periodically regarding the applications and updates its routing table accordingly. The next component that is Authentication server OAuth2 works together with login server to provide Identity Management. After authentication is the next part that manages application life cycle. This part constitute Bits Service(Cloud controller), nsync, Diego brain and cell reps. When an application is pushed into Cloud Foundry, it is basically targeted to bits service which directs the diego brain to stage and run the application.Diego executes the droplet and perform application life cycle tasks. Nsync, Cell reps helps in making the application available every time. When a client scales the application nsync and cell reps work together along a chain to keep applications running. The next component deals with application storage and execution. An application is stored in a blob store. A blob store is a repository that stores large binary files which include application code packages, build packs and droplets. It can be configured as an internal file server or an external like S3. We will discuss blob store in detail in further sections. The diego cell manages the start, stop and status of the

applications. Applications in cloud foundry basically depends upon services. Services are third party applications like database etc. When a service bind is requested in Cloud Foundry its broker is responsible for providing an instance. Components of cloud foundry communicates via HTTP and HTTPS protocols, sharing messages and using consul server and bulletin board system(BBS). Loggregater streams application logs to the client. Metric collector collects the statistics that is used by operator to monitor cloud foundry deployment.

3.1.3 Load Balancing with cloud foundry

Cloud foundry [Clo17c] performs load balancing at following three levels:

1. **BOSH** [Bos] is a project that can provision and deploy small to large scale applications. It was initially developed to deploy cloud foundry but it can be used for any software deployment. It performs monitoring, software updates and failure recovery with zero to minimal downtime . [Clo17c] It creates and deploys virtual machines (VMs) on top of a physical computing infrastructure, and deploys and runs Cloud Foundry on top of this cloud. To configure the deployment, BOSH follows a manifest document.
2. **The Cloud foundry Cloud Controller** [Clo17d] provides REST end points to the requests coming from client. It maintains a database for the users and their applications. It stores user space, region, organization etc. It runs the applications and other processes on the cloud's VMs, balancing demand and managing application life cycle [Clo17c]. An extraction to cloud controller is being developed by IBM and Pivotal called **Bits Service** [Mara] . It encapsulates all the "bits operationsïnto separable scalable services. We will discuss about Bits service in detail in later section.
3. **The router** routes incoming traffic from the world to the VMs that are running the apps that the traffic demands, usually working with a customer-provided load balancer.

3.1.4 Blob Store

[Clo17d] The Bits Service manages a blob store for the following reasons:

1. **Application Cache** Files are stored in application cache with a unique SHA as filenames. We call file a blob object. Hash is a unique string of bytes that represents a files content. So in application cache files are addressed by there content .Files that do not change do not need to be re-uploaded and can be reused. For example a java servlet file is always used by a java application. This file is uploaded once for all the clients.
2. **Application Package** comprises of a zipped file that represents an application. The application is read from this package along with droplet, buildpack etc by Diego and is run. Blob store saves last three application package of an application and the most

recent version is used by diego. If there is an error in the most recent version it is deleted and a version prior to it is used by diego.

3. **Droplet** is the result of taking application package staging is using buildpack and preparing it to run.

It can be configured as an internal file server(but this is not a scalable solution) or an external like S3, openstack swift etc. For the purpose of implementation and evaluation we have used S3 as our blob store.

3.2 Bits Service

Bits service [Ste16] is a micro service developed by IBM and Pivotal to handle bits. Bits are application packages, droplets or buildpacks.It encapsulates all the bits related functions in one service. It will be used for uploads and downloads of application bits, it will be used for resource matching and handles packages, droplets, buildpack. Bits can be stored locally on the disk or remotely on S3 or Openstack swift. It is used while resource matching described in 3.2.2. Since it extracts all the bits operations cloud controller will become more responsive. File upload download and push will become faster.

3.2.1 Need For Bits-service

Bits service is required to free the cloud controller from bits handling so that it can be scaled easily and independently. It is easy to maintain the service and extend it further. Cloud controller is a huge code base and if something goes wrong with it becomes really difficult. It leads to simpler operations. It is a cleaner API that is only related to bits.

3.2.2 Application Deployment on Cloud Foundry

Cloud Foundry comes with a command line interface cf CLI. cf CLI can be used to change the language of terminal output, to log into cloud foundry, to push applications to cloud foundry etc. To deploy an application to first time or to sync the changes to cloud foundry "cf pushïs used. "cf pushïs responsible for the resource matching in cloud foundry. cf push takes the name of the application and domain. Application name and domain are then sent to Bits Service. Bits service maintains a local database where it stores application name, its domain and other metadata related to application. If the application is pushed for the first time, a GUID is created by bits service for that application. GUID is a unique ID given by bits service to each application that is used later for storing the application on blobstore. The SHA of files are then uploaded by cloud foundry to the bits service. Bits service looks into application cache 1 for all the SHA

sent by cloud foundry. It then returns back a list of SHAs that were found and cloud foundry uploads the bits that were not already present on application cache to Blobstore 3.1.4.

4 Suggested Algorithms

This chapter deals with the algorithms, their complexity analysis, advantages, disadvantages and improvements of the approaches that we are implementing. In the following sections we describe the problem and various suggested solutions.Before discussing at the solutions lets try to have a detailed look into the problem that we are trying to solve.

4.1 Problem

This section describes the problem of "Resource Matching", that we are trying to solve through this thesis. Suppose we have a file on two different machines A and B.We change the file of machine A and we want that file on machine B should get those changes. This problem is know as Resource matching. There are many solutions available to this problem as discussed in previous chapter **??**. Resource matching is an algorithm that tries to solve the issue of updating a file on one machine identical to a file on another machine. There are many assumptions made by different algorithms discussed in previous chapter. Some assumed that bandwidth available was low, some assumed it was high. Some other assumptions made by few other algorithms were regarding the database at server side, they assumed that the files were stored in a local database on the server.

As we have discussed in chapter ?? we are solving the issue of uploading an application bits to the bits service that is a PaaS cloud. When a client creates an application it pushes the application blobs to the cloud and every time it makes changes to its application blobs, it pushes the changed application to the cloud again so that it can synchronize the blobs in the bits service. We are trying to extend this topic of resource matching a bit by solving the issue of resource matching for application files. A computer application consists of many blobs depending on the type of application: banking, insurance, engineering design, health care, marketing gaming etc and type of programming language used to write the application. To understand the problem that we are trying to solve, it is important to understand the structure of files and folders created by different programming language.

1. **Java:**
2. **Python:**

Since each type of programming language has its own type of folder structure and can have a range of different number of files, we have to consider this factor while solving the issue. Another factor that needs to be considered while solving the issue of file synchronization is that application files and folders are not stored on a local database on the server but they are stored on a Blobstore(that is remote). We are looking for a solution to a problem of file synchronization for a PaaS cloud which is based on IaaS cloud. Since the cloud foundry is based on IaaS cloud, which is a remote setup, we have to keep this factor also under consideration that it will cost us some bandwidth and connection cost even if we need metadata of the blobs present on database.

Different types of folder structure, different number of files and a blobstore are the additional problems that needs to be dealt with while designing an algorithm that would upload application bits from client to bits service. In following sub section we will discuss the assumptions we made,while we are trying to solve through this project.In the following section we will look at the existing algorithm for uploading bits to the bits service followed by other different suggested approaches.

4.1.1 ASSUMPTIONS

Here we state some assumptions that are kept under consideration while solving the problem of resource matching. Following are the assumptions:

1. **Bandwidth:** We assume to have reasonably high bandwidth. By reasonably high, we mean it is not high enough, that we can push the entire application every time we do a "cf push"to the bits service. For eg., in case of a java application there are a millions of small blobs and the application could be of the range of GBs. So, we try that CPU computations do not become a bottleneck while calculating changes in the application.
2. **Client/Server:** We are running the client and server from same machine for implementation and evaluation.
3. **Storage:** We assume to have unlimited storage at the Blobstore. I have chosen AWS S3 for implementation, storage and evaluation of my algorithms.
4. **One way synchronization:** We need to synchronize only from client to bits service.

4.2 V2 API

In this section we discuss the existing V2 algorithm4.1 and how it helps to solve the problem. To upload an application the client computes the SHA1s of all the blobs in the application. The client then sends a “match” request to the bits service which includes the SHA1s of all

the blobs. The bits service responds by saying which of those blobs are already available in the Blobstore. The client then uploads the remaining blobs from the application as a zip blob. The bits service then reconstitutes the application using the zip blob and the relevant blobs from the Blobstore. The bits service database is typically configured to store blobs greater than 64 KB in size, so the vast majority of small blobs, e.g., .class blobs, are not stored in the Blobstore. However, the client is not currently aware of this and includes all blobs (by SHA1) in its resource matching query to the bits service. Since these blobs are not matched, the CLI then stores them in the application zip blob in the request to upload the application to the bits service. Following is the algorithm for V2 Service client and bits service 4.1, 4.2.

Algorithm 4.1 Client

$hash[] \leftarrow$ SHA of each blob
for each $blob \in \mathcal{A}pplication$ **do**
    $hash[] \leftarrow$ Digest::SHA1.hexdigest *blob*
end for

$knowhash[] \leftarrow$ **PUT /match** *hash[]*
for each $sha \in \mathcal{K}nownhash[]$ **do**
    if $hash[]$ does not include *sha* **then**
        folder $\leftarrow$ blob corresponding to *sha*
    end if
end for
zipblob $\leftarrow$ *zip(folder)*

PUT /bits *zipblob*

Algorithm 4.2 bits service

/match do
$knownhash[] \leftarrow$ SHA known by bits service
bits service receives *List of SHAs*
for each $sha \in \mathcal{L}ist$ **do**
    bits service makes ***HEAD*** request to remote database
    if $request.status = 200$ **then**
        $knownhash[] \leftarrow sha$
    end if
end for
return *knownhash[]*
end /match

/bits do
$folder \leftarrow$ *unzip(zipblob)*
for each $blob \in \mathcal{F}older$ **do**
    if $blob.size >$ *64kB* **then**
        $blobhash \leftarrow hashofblob$
        PUT *blobhash, blobcontent* to Blobstore
    end if
end for
end /bits

4.2.1 ALGORITHM:

As we can see in the figure 4.1,there is a client communicating with the bits service and a blobstore that has the application cache and stores application package. This algorithm works in following steps

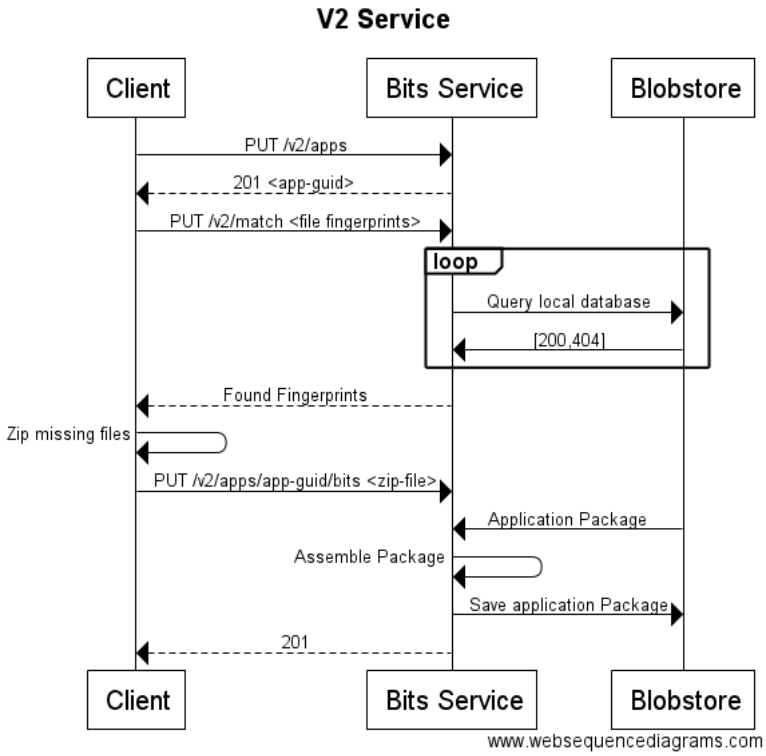

Abbildung 4.1: V2 Service

1. In the first PUT request client sends application name to the bits service and the bits service creates a new GUID for the client and sends the GUID with status 201. If the local database in bits service already has the GUID it sends the GUID with status 200.
2. In second step the client calculates SHA of all the blobs present in application directory and sends this as a JSON with other details like organisation, GUID to the bits service.
   a) If it is the first time the application is uploaded to the bits service. It will return the same list to client as the unknown blobs.
   b) If it is not the first time the application is uploaded to the bits service, it will make HEAD request to the application cache and it will send all the SHAs that matched to the client.
3. Client receives a list of SHAs that were known by bits service and creates a zip of all the blobs that were not found and sends the zip to bits service.
4. Bits Service unzips the zip file received from client. For each file with size more than 64KB it calculates SHA and saves it as the name of the file. Then it saves each of this file in the application cache.

a) if it is the first time the application is pushed, the fileis zipped and is saved as an application package with name as GUID.

b) If it is not the first time the application is pushed, bits service downloads the zip file(named as GUID received from client) from blobstore. Unzips this file and unzips the file received from client. For each file present in the the zip file from client it saves the file in the package received from blobstore. After assembling the entire package, bits service zips it again and sends it to blob store.

5. Finally, when all the blobs are stored on the Blobstore bits service sends a status code of 201 to client and this completes the entire process.

SPACE AND TIME COMPLEXITY:

Time complexity at client can be calculated by the operations performed by client. Following are the operations performed by client:

1. **Sends Application name:** This operation take O(1) as it just reads the name of application and sends it to bits service.
2. **Calculates SHA of application blobs:** In this operation the client goes through each blob and calculates SHA1 of each blob. It takes O(n) where n is the number of blobs.
3. **Calculates unknown SHA at bits service:** After receiving known list of SHAs from bits service, client goes through the list of SHAs that it already has and calculates the SHAs that are not present at bits service. Thus this operation takes O(n) where n is number of application blobs.
4. **Calculates zip blob:** After calculating unknown blob, client puts all unknown blob into a zip blob. This operation takes O(n) time.

So, the total time taken by Client is: **O(1)+O(n)+O(n)+O(n) = O(n)**
Total Space complexity on client is: **O(1)**

To estimate the total time taken by bits service we need to look at the time taken to perform each operation. Following are the operations performed on bits service side:

1. **Finds the GUID corresponding to application name:** When bits service receives application name from the client it goes through the list of applications that are already saved and sends a list of GUIDs. If application name is not present on the bits service,it creates a new GUID for application and sends it to client. This operation take O(n) time where n is the number of applications on bits service.
2. **Makes HEAD request to Blobstore:** When the bits service receives list of SHAs from client, it makes a HEAD request for each SHA against the Blobstore. This operation take O(n) time, where n is number of SHAs sent by client.

3. **Saves changed/new blobs on Blobstore:** When bits service receives changed/new blobs from client, it downloads the zip file from blobstore and assembles the package. This operation takes O(n) time, where n is number of changed/new blobs.

So, the total time taken by bits service is: **O(n)+O(n)+O(n) = O(n)**, where n is either number of blobs on client or number of applications on bits service(whichever is greater) Total space taken by bits service is: **O(n)**, where n is number of applications saved on bits service.

4.2.2 DISCUSSION:

Since this algorithm saves only the blobs with size greater than 64KB in Blobstore, first upload to the Blobstore is faster as compared to the other algorithms discussed in following section(as it saves all blobs). For the blobs with size less than 64KB the bits service does not have to make POST request to Blobstore to save the blob. This saves some storage space, bandwidth from bits service to Blobstore. This approach was taken because it was not possible to store all the blobs on Blobstore as it would have caused large number of write on first upload and large number of reads in second upload, also it was not possible to store nothing on Blobstore as it would have caused large number of writes on every upload.[IBM13] Since this approach does not store blobs of size greater than 64KB, client needs to send all the blobs that are less than or equal to 64KB in every push. For eg., zipkin web Jar contains over 16000 blobs,including over 15,600 .class blobs. The JAR is 32 MB zipped and contains 72 MB of data. Uploading this JAR showed that it took 60-70 percent of time to upload bits to bits service and only 10-15 percent to do the resource match. [IBM13] In this approach bits service sends a list SHAs that are already present on it, client after receiving this list calculates the SHAs that are unknown to bits service. This step calculates the difference twice and adds to the total time of resource matching. Another issue with this approach was, it does not deal with deletion of blobs. If a blob is deleted by client, it does not get deleted at the bits service which leads to unnecessary load on Blobstore.

4.2.3 IMPROVEMENTS:

This algorithm does not store blobs of size less than 64KB and hence every time all the blobs with size greater than or equal to 64KB have to be uploaded to bits service. Most of the time only few of these blobs have changed and we lose a lot of bandwidth from client to bits service and time to upload these bits. To overcome this issue Syke suggested an approach which we would be discussing in upcoming sections.

4.3 SYKE'S APPROACH

To overcome the shortcomings of above approach, Syke suggested an algorithm where he introduces a local cache for the bits service that can hold the hash of the blob, the size of the blob, the last accessed time, and the hit count. When a resource is successfully matched,it gets updated on the local cache with hash of file, access time and the hit count is incremented. The initial cache population can be done by iterating over all of the objects in the bits service. The last accessed time should be the create time from the Blobstore metadata and the hit count should be set to 0. A delayed job should be setup to purge any resources in the cache that have low hit counts or have not been accessed within some period of time. Both of those values should be configurable. An alternative would be to ßcore"the resources based on size, hit count, and last accessed time and purge anything with a low score.With that structure, it is possible to have an insight into the Blobstore and have the metadata necessary to determine what blob are good remote candidates and which are not. This approach can be used to over come the major shortcomings of the existing approach. Since all the SHAs are stored with metadata on a local cache, all the HEAD request made to Blobstore (to get metadata about the blobs stored on it)can be saved. Another improvement, we can remove the application cache, as we are maintaining a local cache for bits sevice. In the previous approach it was not possible to store all the blobs because it increased the number of reads from the Blobstore. In this approach, since we need to read the metadata of the blob, we can do this by querying the local cache. Since this approach uploads everything on the first push, we have saved a lot of bandwidth that was used by previous approach to send blobs less than 64KB on every upload.Now only those blobs are uploaded that are actually changed. Another small improvement was over the calculation of difference that was made by both client and bits service, in this approach we send a list of unknown SHAs to the client so that client does not have to calculate it again. Following is the algorithm describing this the client 4.3 and bits service 4.4.

Algorithm 4.3 Client

```
hash[] ← SHA of each blob
for each blob ∈ Application do
    hash[] ← Digest::SHA1.hexdigest blob
end for

unknowhash[] ← PUT /match hash[]
for each sha ∈ Unknownhash[] do
    folder ← blob corresponding to sha
end for
zipblob ← zip(folder)

PUT /bits zipblob
```

Algorithm 4.4 bits service

```
/match do
unknownhash[] ← SHA not known by
bitsservice
bits service receives List of SHAs
for each sha ∈ List do
    queries local cache
    SELECT * FROM table WHERE hash= sha
    if hash.exist = false then
        unknownhash[] ← sha
    end if
end for
return unknownhash[]
end /match

/bits do
folder ← unzip(zipblob)
for each blob ∈ Folder do
    blobhash ← hashofblob
    PUT blobhash, blobcontent to remote
    database
end for
end /bits
```

4.3.1 ALGORITHM:

Figure 4.2, again there is a client communicating with the bits service and a blobstore that stores application package. This time a local cache is maintained at the bits service. This algorithm works in following steps

1. In the first PUT request client sends application name to the bits service and the bits service creates a new GUID for the client and sends the GUID with status 201. If the local database in bits service already has the GUID it sends the GUID with status 200.
2. In second step the client calculates SHA of all the blobs present in application directory and sends this as a JSON with other details like hit count, access time, size, organisation, GUID to the bits service.
    a) If it is the first upload of the application bits it saves list of hash in its local cache and return the list to client as a list of unknown hash,

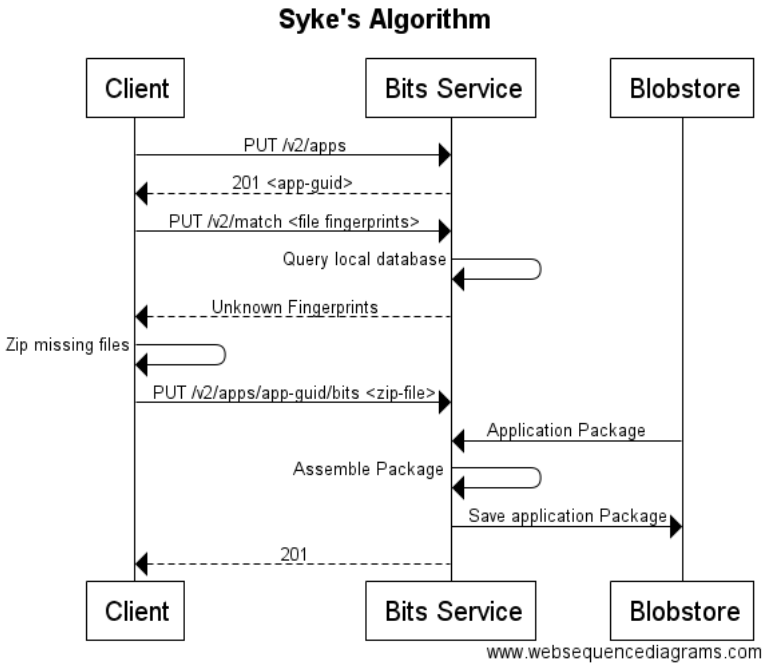

Abbildung 4.2: Syke's Algorithm

b) If it is not the first time the application is uploaded to the bits service,it queries its local cache for each hash in the list and for all the hash that are not present in the local cache the bits service sends them as a list of unknown hashes. For all the SHAs that were available in the local cache, it increments the hit count by 1 and changes the access time of that SHA.

3. Client receives the list of unknown hashes and creates a folder with all the blobs that corresponds to those hashes after adding all the unknown blobs to a folder, client creates a zip blob of the folder and sends it to bits service.

4. Bits Service unzips the zip file received from client. For each file, it calculates SHA and saves it as the name of the file in local cache. Along with SHA it stores the local data like size, hit count etc. Then it saves each of this file in the application cache.

   a) if it is the first time the application is pushed, the file is zipped and is saved as an application package with name as GUID.

   b) If it is not the first time the application is pushed, bits service downloads the zip file(named as GUID received from client) from blobstore. Unzips this file and unzips the file received from client. For each file present in the the zip file from client it saves the file in the package received from blobstore. After assembling the entire package, bits service zips it again and sends it to blob store.

5. Finally, when all the blobs are stored on the blobstore, bits service sends a status code of 201 to client and this completes the entire process.

Then it changes the name of blob to its hash and makes a PUT request to Blobstore to store the blobs.Blobstore replies with a 201 for each blob and after all the blobs are saved on Blobstore bits service sends a 201 to the client.

SPACE AND TIME COMPLEXITY:

Time complexity at client can be calculated by the operations performed by client. Following are the operations performed by client:

1. **Sends Application name:** This operation take O(1) as it just reads the name of application and sends it to bits service.
2. **Calculates SHA of application blobs:** In this operation the client goes through each blob and calculates SHA of each blob. It takes O(n) where n is the number of blobs.
3. **Calculates zip blob:** After receiving list of unknown blobs, client puts all unknown blob into a zip blob. This operation takes O(n) time.

So, the total time taken by Client is: **O(1)+O(n)+O(n) = O(n)**
Total Space complexity on client is: **O(1)**

To estimate the total time taken by bits service we need to look at the time taken to perform each operation. Following are the operations performed on bits service side:

1. **Finds the GUID corresponding to application name:** When bits service receives application name from the client it goes through the list of applications that are already saved and sends a list of GUIDs. If application name is not present on the bits service,it creates a new GUID for application and sends it to client. This operation take O(n) time where n is the number of applications on bits service.
2. **Query local cache:** When the bits service receives list of SHAs from client, it queries for each SHA against the local cache. This operation take O(n) time, where n is number of SHAs sent by client.
3. **Saves changed/new blobs on remote and local cache:** When bits service receives changed/new blobs from client, it makes a POST request for each of them and saves them on remote and local cache. This operation takes O(n) time, where n is number of changed/new blobs.

So, the total time taken by bits service is: **O(n)+O(n)+O(n) = O(n)**, where n is either number of blobs on client or number of applications on bits service(whichever is greater) Total space taken by bits service is: **O(nk)**, where n is number of applications saved on bits service and k is the number of application blobs.

4.3.2 DISCUSSION:

This algorithm saves SHAs of all the applications on a local cache. This is a huge improvement over the previous approach as this saves a lot of time uploading bits that are less than 64KB to the bits service. It save all the HEAD requests and does not require to send blobs (to application package) less than 64KB every time. This approach has same time complexity as that of V2 Service4.1 but has a different and greater space complexity than V2 Service4.1. Since this approach maintains a local cache on the bits service,it leads to a greater space complexity as the number of applications increase on bits service. This space complexity would increase with the number of applications on the bits service.

4.3.3 Improvement:

Since this is a solution for uploading bits to a PaaS cloud, scaling the local cache could be an issue in future as the number of applications increase on bits service. Another issue is the time complexity of total resource matching that is O(n). To further improve the speed of this algorithm and to improve the user experience while uploading bits we suggest an algorithm that would reduce the time and space complexity of the whole process by a reasonable amount.

4.4 MARC'S APPROACH

Marc's approach was the result of discussion, my team had after the literature review of my thesis. As discussed in the chapter **??**, the TAPER approach **??** is a data replication protocol that works in different phases. In the first phase it creates a hierarchical tree structure of the application directory. In this directory structure it calculates SHA of all the blobs then it joins the SHA for all the blobs and directories in a single directory and creates a directory SHA. Similarly it creates a hierarchical directory tree with SHAs. In its second phase, it matches these SHA's with the SHAs on the bits service using a Bloom Filter**??**. In its third phase it calculates chunks of data that is changed in each blob using a content-based similarity detection technique.

In Marc's approach, he used the first phase of the four phases of TAPER approach. We are not matching the calculated SHAs using a bloom's filter as there are chances of false positives as described in TAPER approach and since this algorithm is used to upload bits to a PaaS cloud, false positives are not acceptable. A false positive means a blob is not present on the bits service but bloom filter algorithm could say that it is already present. Hence bloom's filter can only be used to find out which directories are similar. The third phase, that is chunking blob phase was also not considered due to the bandwidth assumption as stated in 4.1.1. This would add to the time and computation overhead on the bits service.

In this approach SHAs are created in a Hierarchical tree structure. Few advantages of this approach are:

1. Calculating SHA is a cheaper operation than calculating checksum using MD4/MD5. MD4/MD5 creates a strong checksum but it is a slow operation.
2. Chances of hash collision are more in SHA. But since in this approach SHAs of all the blobs and directories in same directory are merged to create a single directory SHA. While matching SHAs, first the directory SHAs are matched and if it is matched, blobs inside are not matched. Chances of hash collisions are minimized as directory SHAs are combination of all the blob SHAs.

This algorithm reduces the time complexity and space complexity drastically as compared to the above stated algorithms 4.1 4.2. Following is the algorithm describing client 4.5 and bits service 4.6.

Algorithm 4.5 Client

```
hash[] ← SHA of each blob and directory
for each directory ∈ Application do
    hash[] ← createTree()
end for

unknowhash[] ← PUT /match hash[]
for each sha ∈ Unknownhash[] do
    folder ← blob corresponding to sha
end for
zipblob ← zip(folder)

PUT /bits zipblob
```

Algorithm 4.6 bits service

```
/match do
unknownhash[] ← SHA not known by
bitsservice
bits service receives List of SHAs
for each sha ∈ List do
    if sha = directory then
        bits service makes HEAD request to remote
        database directory and gets hash of all blobs
        in directory and then calculate a joint hash
        if request.status = 404 then
            unknownhash[] ← sha
        end if
    else
        bits service makes HEAD request to remote
        database
        if request.status = 404 then
            unknownhash[] ← sha
        end if
    end if
end for
return unknownhash[]
end /match

/bits do
folder ← unzip(zipblob)
for each blob ∈ Folder do
    blobhash ← hashofblob
    PUT blobhash, blobcontent to remote
    database
end for
end /bits
```

4.4.1 ALGORITHM:

The diagram 4.3 might look similar to the diagram of V2 Service **??**, but this differs in the way client make POST request with list of SHAs to bits service. Before getting into details of this algorithm let us have a look on the directory tree structure created by **createTree()** function. Following is a sample directory structure 4.4 of an application directory. Here the "Root"represents

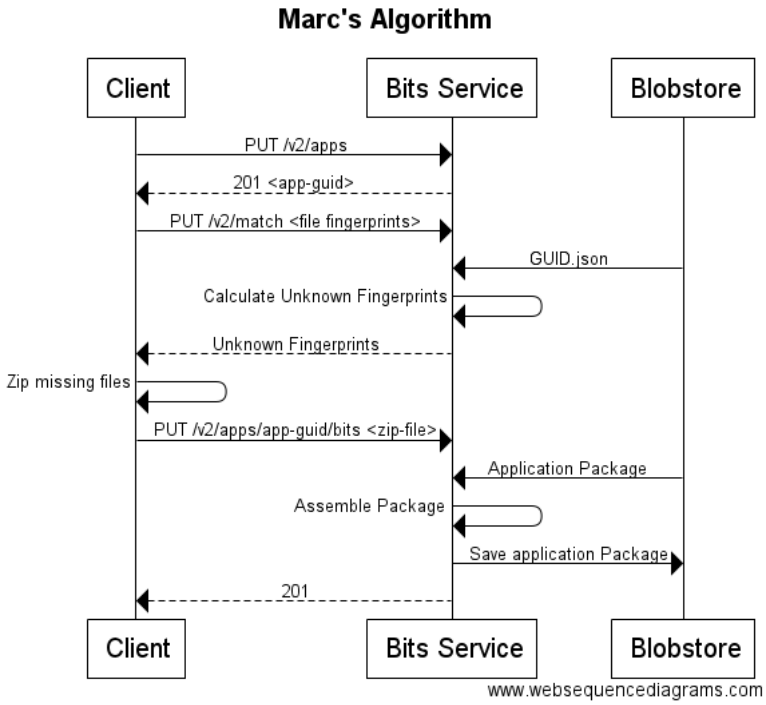

Abbildung 4.3: Marc's Algorithm

application directory, "DirAänd "DirB"represents directories directly under application directory. "DirCänd blob1 are under DirA and so on.

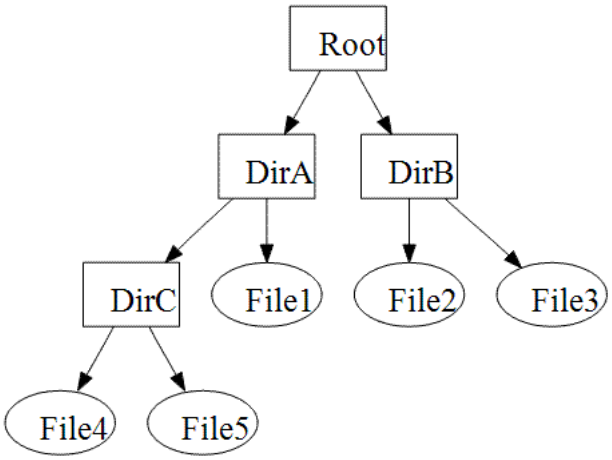

Abbildung 4.4: Directory Structure [Ash]

Client goes through each blob in the application directory and calculates their SHA and path. For all blobs that have same paths, their SHAs are joined together and a directory SHA

with path is calculated. The process continues until all the blobs and folders are covered. **createTree()** function performs calculation of hashes in a tree structure.

SHA of each entry can be calculated as follows:
SHA of blob1,**a**= Digest::SHA1.hexdigest blob1,
SHA of blob2,**b**= Digest::SHA1.hexdigest blob2,
SHA of blob3,**c**= Digest::SHA1.hexdigest blob3,
SHA of blob4,**d**= Digest::SHA1.hexdigest blob4,
SHA of blob5,**e**= Digest::SHA1.hexdigest blob5,
SHA of DirC,**f**= Digest::SHA1.hexdigest [d,e],
SHA of DirA,**g**= Digest::SHA1.hexdigest [f,a],
SHA of DirB,**h**= Digest::SHA1.hexdigest [b,c],
SHA of DirB,**i**= Digest::SHA1.hexdigest [g,h]

The SHAs calculated for blobs and directories are a,b,c,d,e,f,g,h and i. Following is the tree structure of hash in JSON format calculated by client and sent to bits service.

```
{
  "i":{"path":"Root","children": ["g","h"]},
  "g":{"path":"Root/DirA","children": ["f","a"]},
  "h":{"path":"Root/DirB","children": ["b","c"]},
  "f":{"path":"Root/DirA/DirC","children": ["d","e"]}
}
```

1. In the first PUT request in figure 4.3 client sends application name to the bits service and the bits service creates a new GUID for the client and sends the GUID with status 201. If the local database in bits service already has the GUID it sends the GUID with status 200.
2. In second step the client calculates SHA of all the blobs present in application directory and sends this as a JSON with other details like organisation, GUID to the bits service.
    a) If it is the first upload of the application bits, bits service saves list the of hash in a JSON file and saves that file with GUID as its name to the blobstore and return the list to client as a list of unknown hash,
    b) If it is not the first time the application is uploaded to the bits service,it downloads the JSON file from the blob store and comapres the two JSONs. Bits service then returns a list of all the unmatched blobs to client.
3. Client receives the list of unknown hashes and creates a folder with all the blobs that corresponds to those hashes after adding all the unknown blobs to a folder, client creates a zip file of the folder and sends it to bits service. Along with the zip file it sends a file with JSON tree structure of all the blobs.
4. Bits Service receives the zipped file and json file from the client.

a) if it is the first time the application is pushed, the file is saved as an application package with name as GUID along with the JSON file.

b) If it is not the first time the application is pushed, bits service downloads the zip file(named as GUID received from client) from blobstore. Unzips this file and unzips the file received from client. For each file present in the the zip file from client it saves the file in the package received from blobstore. After assembling the entire package, bits service zips it again and sends it to blob store.

5. Finally, when all the blobs are stored on the blobstore, bits service sends a status code of 201 to client and this completes the entire process.

SPACE AND TIME COMPLEXITY:

Time complexity at client can be calculated by the operations performed by client. Following are the operations performed by client:

1. **Sends Application name:** This operation take O(1) as it just reads the name of application and sends it to bits service.
2. **Calculates SHA of application blobs:** In this operation the client goes through each blob and calculates SHA and path of each blob. It then goes through the path of all blobs and creates a joint SHA for blobs with same path.It takes O(nk) where n is the number of blobs and k is the number of directories.
3. **Calculates zip blob:** After receiving list of unknown blobs, client puts all unknown blob into a zip blob. This operation takes O(n) time.

So, the total time taken by Client is: **O(1)+O(nk)+O(n) = O(nk)**
Total Space complexity on client is: **O(1)**

To estimate the total time taken by bits service we need to look at the time taken to perform each operation. Following are the operations performed on bits service side:

1. **Finds the GUID corresponding to application name:** When bits service receives application name from the client it goes through the list of applications that are already saved and sends a list of GUIDs. If application name is not present on the bits service,it creates a new GUID for application and sends it to client. This operation take O(n) time where n is the number of applications on bits service.
2. **Calculate unkown SHAs :** After receiving a list of directory and blob SHAs from blobstore, bits service makes comparison with SHA sent by client. This takes O(log n) time,where n is number of SHAs.

3. **Saves changed/new blobs on remote and local cache:** When bits service receives changed/new blobs from client, it makes a POST request for each of them and saves them on remote and local cache. This operation takes O(k) time, where n is number of changed/new blobs.

So, the total time taken by bits service is: **O(n)+O(log n)+O(k) = O(log n)**, where n is number of blobs on client. Total space complexity of bits service is: **O(1)**,Since bits service does not store anything locally.

4.4.2 DISCUSSION:

This approach reduced the number of comparison to O(log n) as compared to other approaches discussed so far. But it takes some time in downloading JSON file. This algorithm was able to solve all of the issues of V2 service 4.1.

4.4.3 IMPROVEMENT:

The algorithm made a very important improvement for the phase of comparison of blob SHAs but it takes some time and uses some bandwidth while downloading the JSON file from blobstore. But we can improve the overall resource matching algorithm by combining Syke's 4.2 and Marc's 4.3 approach, so that we can have best of both the approaches. Following section describes a combined approach which has best of both the approaches and can be considered as one of the good candidates for resource matching algorithm in PaaS cloud.

4.5 COMBINED APPROACH

In this approach we are trying to have a combination of the approaches suggested by Syke, where he suggests to have a local cache (with SHAs of all the blobs on the Blobstore) and Marc's, where he suggests to have a hierarchical tree structure of application directory to be uploaded. Benefit of Syke's approach is that we have a local cache at bits service, we save all the HEAD request made by bits service to Blobstore. This saves a lot of time, bandwidth and improves the speed of the whole process. But the time complexity of this approach was O(n) as the local cache have to be queried for all the blobs in application. Benefit of Marc's approach is that the time complexity of comparison of SHAs is O(log n). Following is the client 4.7 and bits service 4.8 algorithms of combined approach that takes benefits of both the approaches.

Algorithm 4.7 Client

```
hash[] ← SHA of each blob and directory
for each directory ∈ Application do
    hash[] ← createTree()
end for

unknowhash[] ← PUT /match hash[]
for each sha ∈ Unknownhash[] do
    folder ← blob corresponding to sha
end for
zipblob ← zip(folder)

PUT /bits zipblob
```

Algorithm 4.8 bits service

```
/match do
unknownhash[] ← SHA not known by
bitsservice
bits service receives List of SHAs
for each sha ∈ List do
    queries local cache
    SELECT * FROM table WHERE hash= sha
    if hash.exist = false then
        unknownhash[] ← sha
    end if
end for
return unknownhash[]
end /match

/bits do
folder ← unzip(zipblob)
for each blob ∈ Folder do
    blobhash ← hashofblob
    PUT blobhash, blobcontent to remote
    database
end for
end /bits
```

4.5.1 ALGORITHM:

Figure 4.5 is the sequence diagram for the combined approach. This figure looks similar to Syke's sequence diagram 4.2 because the bits service has a local cache that stores all the SHAs. This algorithm works in following steps:

1. In the first PUT request client sends application name to the bits service and the bits service creates a new GUID for the client and sends the GUID with status 201. If the bits service already knows the application it sends the GUID with status 200.
2. In second step the client calculates SHA of all the blobs present in application directory in hierarchical tree structure as explained in 4.4.1 and sends this as a JSON with other details(like GUID, creation date, hit count, size etc) to the bits service
    a) If it is the first time the application is uploaded to the bits service. It will save all the SHAs sent by client to its local cache and return the same list to client as the unknown blobs.

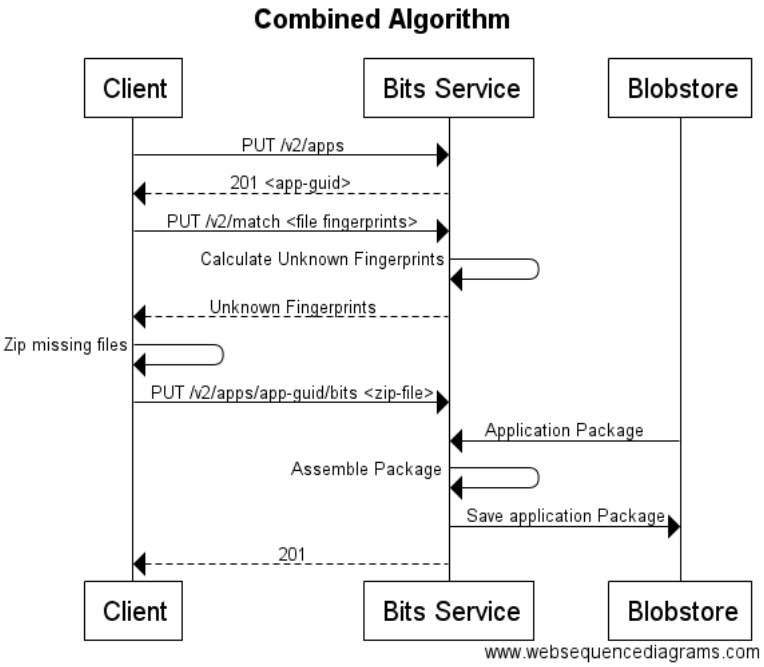

Abbildung 4.5: Combined Algorithm

b) If it is not the first time the application is uploaded to bits service, it will query its local cache for the start key that it reads in JSON, if it does not match, it saves that SHA into the local cache and it will query for its children and repeat the process until it reaches the leaf nodes of the tree i.e., blobs.

3. For all the blob's SHAs that didn't match the SHAs in bits service database are put into a list and that list is sent as a list of Unknown blob SHAs to client.
4. Client creates a zip of all the blobs and sends it to bits service along with the JSON structure.
5. Bits Service receives the zipped file and json from the client.
   a) if it is the first time the application is pushed, the each key of JSON is saved in the local cache and zipped file is saved as a application package to the blobstore.
   b) If it is not the first time the application is pushed, bits service downloads the zip file(named as GUID received from client) from blobstore. Unzips this file and unzips the file received from client. For each file present in the the zip file from client it saves the file in the package received from blobstore. After assembling the entire package, bits service zips it again and sends it to blob store.
6. Finally, when all the blobs are stored on the blobstore, bits service sends a status code of 201 to client and this completes the entire process.

4.5.2 SPACE AND TIME COMPLEXITY:

Time complexity at client 4.7 can be calculated by the operations performed by client. Following are the operations performed by client:

1. **Sends Application name:** This operation take O(1) as it just reads the name of application and sends it to bits service.
2. **Calculates SHA of application blobs:** In this operation the client goes through each blob and calculates SHA and path of each blob. It then goes through the path of all blobs and creates a joint SHA for blobs with same path.It takes $O(n^2)$ where n is the number of blobs.
3. **Calculates zip blob:** After receiving list of unknown blobs, client puts all unknown blob into a zip blob. This operation takes O(n) time.

So, the total time taken by Client is: $\mathbf{O(1)+O(n^2)+O(n) = O(n^2)}$
Total Space complexity on client is: **O(1)**

To estimate the total time taken by bits service we need to look at the time taken to perform each operation. Following are the operations performed on bits service side 4.8:

1. **Finds the GUID corresponding to application name:** When bits service receives application name from the client it goes through the list of applications that are already saved and sends a list of GUIDs. If application name is not present on the bits service,it creates a new GUID for application and sends it to client. This operation take O(n) time where n is the number of applications on bits service.
2. **Query local cache:** When the bits service receives list of SHAs from client, it queries local cache for all blobs SHAs.This operation take O(log n) time, where n is number of SHAs sent by client.
3. **Saves changed/new blobs on remote and local cache:** When bits service receives changed/new blobs from client, it makes a POST request for each of them and saves them on remote and local cache. This operation takes O(n) time, where n is number of changed/new blobs.

So, the total time taken by bits service is: **O(n)+O(log n)+O(n) = O(n)**, where O(log n) is number of blobs on client. Total space complexity of bits service is: **O(nk)**,n is number of application blobs and k is number of applications on bits service Since bits service maintains a database locally.

4.5.3 DISCUSSION:

In this algorithm we saw a lot of improvements in time complexity. The time complexity reduced to O(log n) where n is the number of application blobs. This algorithm does not impose any rule of not storing blobs of size less than 64KB and hence, while sending the blobs only changed/new blobs are sent from client to bits service. Since this algorithm calculates directory hash and compares directory hash before comparing blob hash, chances of hash collision is reduced. But by keeping a local cache at the bits service it increases the space complexity at bits service side to O(nk) where n is the number of blobs and directory in application directory and k is the number of applications on the bits service. This would increase with increase in applications on the bits service.

4.5.4 IMPROVEMENT:

Since this algorithm combined the approach of both Syke and Marc, it had the benefits of both. There is a tradeoff between the space complexity of combined approach and a little bandwidth of downloading JSON file from Marc's apprach which needs to be considered while considering an algorithm for production purposes.

In the next chapter we present the results obtained after evaluation of all these algorithms.

5 Evaluation

This chapter is dedicated to evaluation of all the four algorithms that were listed in chapter ??.

5.1 Performance

As we have discussed in **??** performance of an algorithm is measured by the results of following parameters:

1. **Total time of synchronization:** is the time taken by the complete process of file synchronization. It starts with the time when client requests a **"cf push"** and ends when the changed application becomes available for the next client in blobstore. It includes the time taken by following phases of resource matching depending upon the algorithm.
    a) Time taken by client/bits-service to calculate SHAs
    b) Time taken by client/bits-service to send calculated SHAs to bits-service/client
    c) Time taken by client/bits-service to calculate changes
    d) Time taken by client to compress the changed files
    e) Time taken by bits-service to download the previous version from blobstore
    f) Time taken by bits-service to assemble the application package
    g) Time taken by bits-service to upload the changed package to blobstore
2. **Bandwidth usage:** is the amount of bandwidth used during the process of file synchronization. We are assuming that we have a reasonable amount of bandwidth available. Hence we make sure that CPU computation of an algorithm does not become a bottleneck for the performance of algorithm. Also we make sure that we are not uploading extra bits that are already known by bits service, so that optimum amount of bandwidth is used.
3. **Usage of computation resources at server(bits-service):** Clients come with varied amounts of computation power but there is a fixed amount of computation resources available at the server. Hence the algorithms must use minimum amount of CPU at server side.

4. **Total number of messages exchanged between client , bits-service and remote backend:** This is one of the major factor deciding the performance of algorithm. Since all the three components client, bits service and remote backend are connected to each other via a networks, algorithms must make sure that number of messages exchanged between each component is minimal. By message exchange we mean HTTP requests here.

Results of all the above parameters decide if an algorithm is a good candidate for uploading bits to a PaaS cloud or not. But the performance of an algorithm also depends on the type of input that is given to it, the network speed, type of blob store and SHA calculation. Following subsections describe these factors in detail.

5.1.1 Factors Affecting Performance:

This section describes the factor that affect the performance of any algorithm. Following are the factors that affect the performance.

1. **Application Folder Characteristics:** This is a major factor that affect the performance of an algorithm. We will see in next few subsections the affect this factor makes on the performance. This factor is a combination of following sub factors:

   a) **Number of files in an application:** affects the time taken by algorithm to calculate SHAs. If number of files is more than 60,000 it takes around 20 second to calculate all the SHAs else if it is around 1000, it takes less than 1 second to calculate SHAs.

   b) **Size of application:** affects the total time of first push as whole application has to be pushed from client to server and it also affects the time taken by server to download/upload the package from/to package blob store.

   c) **Number of Folders:** while calculating SHA tree for Marc's and Combined algorithm, all the directory names needs to be called in the memory. It is directly proportional to the memory used at client side while calculating SHA's.

   d) **Number of nested folders:** affects the SHA calculation phase and difference calculation phase of Marc's and Combined algorithm. If the application folder is heavily nested it increases the number of calculations which in turn increase the computation at server and vice versa.

   e) **Size of changed file:** affects the time taken by client to send changed files and also affects the time to assemble the package. If a file size is 2-3 Mb it takes around 3-5 seconds to assemble and if the file size is few Kbs then it takes 1-2 seconds to assemble.

	Size of Application(MB)	Number of Files	Nested	Example
Large Number of small and medium sized files	683	57000	Medium	Linux Kernel
Small Number of medium and large files	143.8	3704	Low	Emacs
Large Number of small files	74.4	22,530	High	Zipkin-web
Small Number of large files	50	153	Less	Tex
Large number of small files	24	6632	Medium	Java-Unwritable-Dir
Small Number of large files	10.8	109	Medium	Dot-net
Large Number of very small files	8.6	1082	Medium-High	Portal Mail Master Nodejs

Tabelle 5.1: Type of Applications

f) **Number of changed files:** affects the time taken by client to send changed files and also affects the time to assemble the package.

2. **Network Speed:** is the upload and download speed for an application. It depends on the upload speed of client, upload and download speeds of server. It affects the overall time of resource match as all the three components(client, server and blobstore) communicate via network.

3. **Cache:** is the blobstore that saves actual application package and store metadata about the package. It affects the phase of calculating the difference, if it is a remote cache a lot of time and connection requests will go into getting/writing meta data from/to remote cache. If it is a local cache getting/writing metadata becomes faster.

4. **Hash calculation:** Calculating a list of Hashes or a tree of Hashes takes more or less the same amount of time. The algorithm used to calculate the Hash affects the total time. If MD4/MD5 is used it takes a bit more longer to calculate the hash of each file but if SHA1/SHA256 is used, SHA calculation becomes faster.

Above stated factors affect the performance of resource matching algorithm for uploading bits to PaaS cloud. As we have seen in the above factors that the type of input is a major deciding factor for measuring the performance of an algorithm. We have taken a few example applications with a combination of sub factors to evaluate the algorithms.Table 5.1 describes the types and combinations of application files. The next table that is 5.2 describes the basic

behavior of applications described in 5.1. Basic behavior includes the time taken by application to zip, to upload and download etc.

5.1.2 Application Characteristics

As we have seen in table 5.1 there are different types of application different kinds of folder structure, different size, different number of files etc. We have tried to take into consideration each factor described in 5.1.1 application folder characteristics.In this table the first column describes the type of application, second column describes the size of application in MB, third column describes the number of files in the application, fourth column describes the level of nesting of folders inside application folder and the last column is the example application that I used to push into Cloud Foundry. Following are some facts that were not described in the table. 5.1 .

1. By small sized files we mean files of size 64KB or less.
2. By medium sized files we mean files of size 64KB - 500KB.
3. By large sized file we mean files of size more than 500KB.
4. By low nesting we mean files are lying either directly inside the application folder(level 0) or level 1 or level 2.
5. By medium nesting we mean files are lying at level 3- level 5
6. By high nesting we mean files are lying at level 5 - level 7.

In table 5.2 we describe the basic information about applications. First column describes the type of application, second column describes the size of application in MB, third column describes the time taken by bits service to download application package from remote blob store, fourth column describes the time taken by bits service to upload application package to blob store and the last column describes the time taken by client to zip the application on first push. This table describes the behavior of network speed on different kinds of applications.

Before we move on further with the evaluation results of different algorithms discussed in chapter **??** , following are parameters of our development and test environments.

- Memory: 15.3 GiB
- Processor: Intel® Core™ i7-3740QM CPU @ 2.70GHz × 8
- OS-Type: 64 Bit, Red Hat Enterprise Linux
- Upload Speed: ??
- Download Speed: ??

	Size(MB) (Unzipped)	Size(Zipped) (MB)	Download Time(Blob Store -> Bits Service) (Seconds)	Upload Time(Bits Service -> BlobStore) (Seconds)	Time Taken to zip(Seconds)
Large number of small and medium files ??	683	179	100	27.9	160.85
Small Number of medium and large files ??	143.8	40.7	15.5	4.53	9.09
Large Number of small files ??	74.4	33.5	20.01	2.13	38.94
Small number of large files ??	50	29.9	13.27	3.35	1.74
Large Number of small files ??	24	11.1	4.9	0.83	8.2
Small Number of Large files ??	10.8	4.4	1.89	0.615	0.62
Large Number of very small files ??	8.6	4.5	1.46	1.78	0.779

Tabelle 5.2: Basic information about application files

- Blob Store: Amazon Simple Storage Service (S3)

5.2 First Push

5.2.1 Description

First push is the first time an application is pushed into Cloud Foundry. First push differs for different algorithms. First push is a combination of operations like: Client calculates list of SHAs, Client creates a zip of all files, Server uploads the zip to blob store, Server uploads SHA and content of each file greater that 64KB to blob store or metadata of all the files and folder in local cache or stores the metadata in a file on blobstore(depending on the algorithm).

- In the current v2 algorithm, all the files that are greater than 64KB in size are pushed to application cache with their name as their SHA. Then the application folder is zipped and pushed to blob store.

	V2 API	Syke's Algorithm	Marc's Algorithm	Combined Algorithm
Large number of small and medium files ??	45min	2.5 hr	510	2.5 hr
Small Number of medium and large files ??	5 min	700 sec	35.5 sec	700 sec
Large Number of small files ??	18 min	1.5 hr	88.9 sec	1.5 hr
Small number of large files ??	15 sec	70 sec	43.68sec	70 sec
Large Number of small files ??	-	0.5 hr	69.3 sec	0.5 hr
Small Number of Large files ??	12 sec	32 sec	5.12 sec	32 sec
Large Number of very small files ??	1 min	150 sec	5 sec	150 sec

Tabelle 5.3: First Push of application files

- In the Syke's algorithm instead of saving entire files in remote application cache, files are saved in a local cache with there meta data and then application package is uploaded to blob store.
- In Marc's algorithm guid.json and application folder are both pushed to blob store respectively.
- In combined algorithm application folder is pushed to blob store and the SHA of its files and folders are saved into the local cache.

In table 5.3 we display the results of pushing different types of application described in 5.1 first time from client to Bits Service using different approaches.

5.2.2 Observation:

We observe from the above results that minimum amount of time is taken by Marc's algorithm. The reason for this is in this algorithm we do not save metadata of each file at a time, instead we save the entire metadata at a time in a file and push that file along with application zip to blob store. V2 performs a lot of PUT request on the first push as the files are not already there in remote application cache. Sykes' and Combined algorithm perform a lot of write operations on the local cache. We considered a file based system for the local cache hence

it took longer to store metadata on first push, but another DBMS would definitely reduce these numbers. In the next section we will look into the results of second and subsequent pushes.

5.3 Second Push

5.3.1 Description

Second push is the second and subsequent pushes that are made to the application after first push. Client compares the application files with those that are stored on the server. The real file matching starts from the second push. Since in a PaaS cloud, server and blob store are placed close to each other but client is at a distance from them, we require an algorithm that minimizes the amount of data transfer between client and server. In all the four algorithms described in chapter **??** we try to minimize this data transfer between client and server. Following are the summarized points describing how suggested algorithms differ in second push.

- As described in first push in V2 API the client saves SHAs of all the files into the application cache. For the second push the server(after receiving a list of SHAs from client) makes HEAD requests to the blob store for each SHA. It sends back the SHAs that were found at the blob store.
- As we have seen in Syke's algorithm, we save all the SHAs with their meta data into a local cache. Hence on the second push the server(after receiving a list of SHAs from client) queries its local cache for the SHAs and sends back the SHAs that were unknown to it to the client.
- Marc's Algorithm saves the file with tree structure of SHAs at the blob store. During second push, server downloads this file from blob store and matches it against the SHAs received from client. It then returns the SHAs that were not present in the list.
- Combined Algorithm saves the directory tree structure of SHA's in a local cache. It matches the SHA tree structure received from client against its local cache and sends the unknown SHAs to client.

Another difference was in calculation of SHAs of files in the form of list and tree structure. But there is not much difference in performance of tree or list calculation. After calculation of SHAs and the difference of data at client and server the next phases are almost the same i.e., Client creates a zip of all the unknown files and sends that zip to Bits Service. Bits Service downloads the already existing package from the blob store and assembles that package. After assembling that package is uploaded back to blob store. Table 5.4 displays the total times take to push 15 changed files from client to blob store.

	V2 API	Syke's Algorithm	Marc's Algorithm	Combined Algorithm
Large number of small and medium files ??	31min	550sec	427 sec	-
Small Number of medium and large files ??	2.5 min	60 sec	38 sec	33.4 sec
Large Number of small files ??	15 min	170 sec	66.3 sec	63.3
Small number of large files ??	8 sec	36 sec	36.5sec	36sec
Large Number of small files ??	-	43sec	13.8 sec	10sec
Small Number of Large files ??	8.3 sec	7sec	3.9 sec	5 sec
Large Number of very small files ??	1 min	6 sec	6.5 sec	6 sec

Tabelle 5.4: Second Push of application files

5.3.2 Match Time

Second push is the total time taken by changed application at client to reach blob store and be available for the next client. Major part of second push that is calculation of change in the application differs with all the four different algorithm. This section describes the time taken by different suggested approaches to calculate the difference. Difference of how different algorithms compute the change has been described in section 5.3.1. Table 5.5 displays the total time taken by server to calculate changes in different algorithms.

5.3.3 Assemble Package

Another portion of time goes in assembling the package during second push. While assembling the package the server downloads the zip file from blob store and receives the zip file with changed application files from client. It then transfers all the files received from client zip into server zip and uploads this package again to blob store. Table 5.6 displays the time taken by server to assemble and upload.

The time displayed in table 5.6 is total time take to assemble the package and upload it to blob store. This time is almost the same for each kind of application. It is directly proportional to the size of the file to be assembled and the number of files to be assembled. If the size of file is

	V2 API	Syke's Algorithm	Marc's Algorithm	Combined Algorithm
Large number of small and medium files ??	30min	285sec	0.08 sec	-
Small Number of medium and large files ??	2 min	18.5 sec	0.005 sec	0.2 sec
Large Number of small files ??	15 min	110 sec	0.04 sec	0.3
Small number of large files ??	6 sec	0.5 sec	0.024sec	0.01sec
Large Number of small files ??	-	33.5sec	0.006sec	0.1sec
Small Number of Large files ??	6.3 sec	0.5sec	0.004 sec	0.3sec
Large Number of very small files ??	50sec	1.5 sec	0.003 sec	0.4 sec

Tabelle 5.5: Calculation of changed files at Server

	Same for all (assemble + upload)(seconds)
Large number of small and medium files ??	141+100
Small Number of medium and large files ??	20.79+ 15.5
Large Number of small files ??	30 + 20
Small number of large files ??	22.5+ 12.27
Large Number of small files ??	7.5 + 4.9
Small Number of Large files ??	3.64 + 2
Large Number of very small files ??	3.64+ 1.89

Tabelle 5.6: Time to assemble package

2-3 MB it takes 4-5 seconds to assemble the file, else if it is few Kbs it takes around 1-3 seconds to assemble.

5.3.4 Observation

We observe that time taken for the second push was least for Marc's algorithm.

5.4 Discussion

6 Conclusion and Further Work

Literaturverzeichnis

[Ang14] Angel Tomala-Reyes. *What is IBM Bluemix?* 2014. URL: https://www.ibm.com/developerworks/cloud/library/cl-bluemixfoundry/ (zitiert auf S. 20).

[Ash] Ashutosh Phoujdar. URL: https://www.codeproject.com/Articles/26628/Tree-structure-generator (zitiert auf S. 40).

[Bos] Bosh. *What is Bosh?* URL: http://bosh.io/docs/about.html (zitiert auf S. 23).

[Clo17a] Cloud Foundry Documentation. *Deploying Cloud Foundry*. 2017. URL: http://docs.cloudfoundry.org/deploying/common/deploy.html#deploy (zitiert auf S. 21).

[Clo17b] Cloud Foundry Foundation. *Cloud Foundry Documentation*. 2017. URL: https://docs.cloudfoundry.org/concepts/architecture/ (zitiert auf S. 22).

[Clo17c] Cloud Foundry Foundation. *Cloud Foundry Documentation*. 2017. URL: https://docs.cloudfoundry.org/concepts/overview.html (zitiert auf S. 23).

[Clo17d] Cloud Foundry Foundation. *Cloud Foundry Documentation*. 2017. URL: https://docs.cloudfoundry.org/concepts/architecture/cloud-controller.html (zitiert auf S. 23).

[IBM13] IBM. *The Differences between IaaS, SaaS and PaaS*. 2013 (zitiert auf S. 32).

[Kat13] Katie Frampton. *The Differences between IaaS, SaaS and PaaS*. 2013. URL: https://www.smartfile.com/blog/the-differences-between-iaas-saas-and-paas/ (zitiert auf S. 20).

[Mara] P. S. Marc Schunk Steffen Uhlig. *Bits Service*. URL: https://github.com/cloudfoundry-incubator/bits-service (zitiert auf S. 23).

[Marb] Mariash. *A guide to using Bosh*. URL: http://mariash.github.io/learn-bosh/ (zitiert auf S. 21).

[Ste16] Steffen Uhlig,Peter Götz. *Bits Service*. 2016. URL: https://youtu.be/91P-Sg16bIQ (zitiert auf S. 24).

Alle URLs wurden zuletzt am 17. 03. 2008 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift