

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

1. Einleitung	17
2. Fundamentals	19
2.1. Cloud Foundry	20
3. Suggested Algorithms	25
3.1. Problem	25
3.2. V2 Service	26
3.3. SYCTH'S APPROACH	30
3.4. MARC'S APPROACH	34
3.5. COMBINED APPROACH	39
4. Kapitel zwei	45
5. Überschrift auf Ebene 0 (chapter)	47
5.1. Überschrift auf Ebene 1 (section)	47
5.2. Listen	48
6. Zusammenfassung und Ausblick	51
A. LaTeX-Tipps	53
A.1. File-Encoding und Unterstützung von Umlauten	53
A.2. Zitate	53
A.3. Mathematische Formeln	54
A.4. Quellcode	54
A.5. Abbildungen	55
A.6. Tabellen	55
A.7. Pseudocode	55
A.8. Abkürzungen	58
A.9. Verweise	58
A.10. Definitionen	59
A.11. Fußnoten	59
A.12. Verschiedenes	59
A.13. Weitere Illustrationen	59
A.14. Plots with pgfplots	62
A.15. Figures with tikz	62

A.16. Schlusswort	63
Literaturverzeichnis	65

Abbildungsverzeichnis

2.1.	Types of Service Models [Kat13]	20
2.2.	Architecture of Cloud Foundry	21
2.3.	Components of Cloud Foundry [Clo17b]	22
3.1.	V2 Service	28
3.2.	Sycth's Algorithm	32
3.3.	Marc's Algorithm	36
3.4.	Directory Structure [Ash]	36
3.5.	Combined Algorithm	41
A.1.	Beispiel-Choreographie	55
A.2.	Beispiel-Choreographie	56
A.3.	Beispiel um 3 Abbildung nebeneinander zu stellen nur jedes einzeln referenzieren zu können. Abbildung A.3b ist die mittlere Abbildung.	56
A.4.	Beispiel-Choreographie I	60
A.5.	Beispiel-Choreographie II	61
A.6.	$\sin(x)$ mit pgfplots.	62
A.7.	Eine tikz-Graphik.	62

Tabellenverzeichnis

A.1. Beispieltabelle	56
A.2. Beispieltabelle für 4 Bedingungen (W-Z) mit jeweils 4 Parameters mit (M und SD). Hinweist: immer die selbe anzahl an Nachkommastellen angeben.	57

Verzeichnis der Listings

A.1. Istlisting in einer Listings-Umgebung, damit das Listing durch Balken abgetrennt ist	54
---	----

Verzeichnis der Algorithmen

A.1. Sample algorithm	57
A.2. Description	58

Abkürzungsverzeichnis

ER error rate. 58

FR Fehlerrate. 58

RDBMS Relational Database Management System. 58

1. Einleitung

In diesem Kapitel steht die Einleitung zu dieser Arbeit. Sie soll nur als Beispiel dienen und hat nichts mit dem Buch [WCL+05] zu tun. Nun viel Erfolg bei der Arbeit!

Bei \LaTeX werden Absätze durch freie Zeilen angegeben. Da die Arbeit über ein Versionskontrollsystem versioniert wird, ist es sinnvoll, pro *Satz* eine neue Zeile im .tex-Dokument anzufangen. So kann einfacher ein Vergleich von Versionsständen vorgenommen werden.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 4 – Kapitel zwei: Hier werden werden die Grundlagen dieser Arbeit beschrieben.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. 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 2.1:

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.

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.

2. Fundamentals

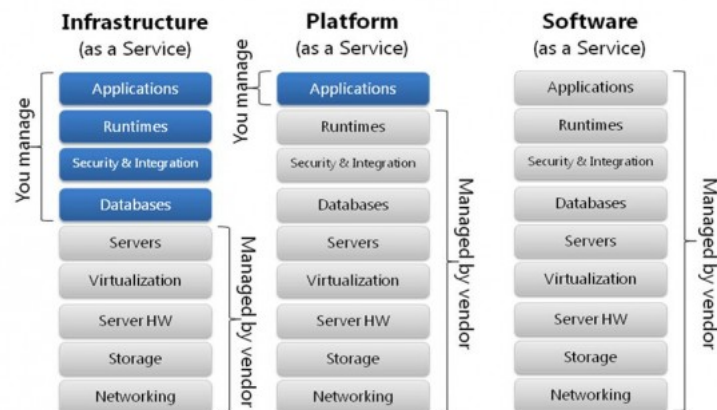


Abbildung 2.1.: Types of Service Models [Kat13]

2.1. 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.

2.1.1. Architecture

This section deals with the architectural functioning of cloud foundry. In diagram 2.2, 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 2.2 the red blocks represents the virtual machines started by Bosh.

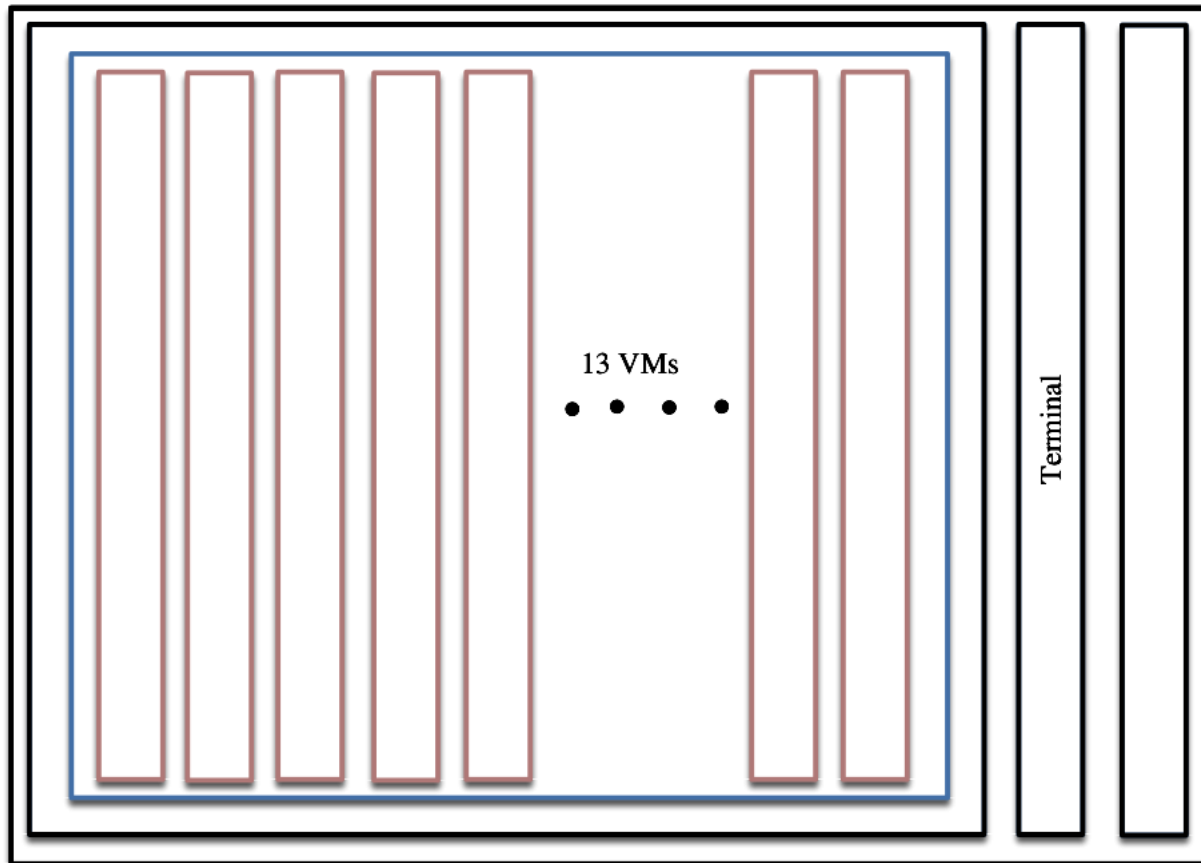


Abbildung 2.2.: Architecture of Cloud Foundry

2.1.2. 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 2.3 shows the major components of Cloud Foundry:

2. Fundamentals

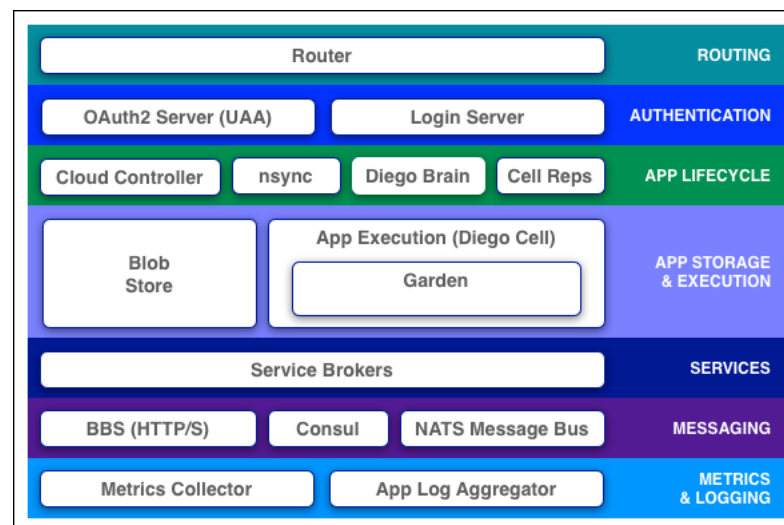


Abbildung 2.3.: Components of Cloud Foundry [Clo17b]

This section [Clo17b] describes the working of each component of Cloud Foundry depicted in diagram 2.3. The 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 constitutes 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 performs application life cycle tasks. Nsync, Cell reps help 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 depend 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 communicate via HTTP and HTTPS protocols, sharing messages and using consul server and bulletin board system (BBS). Loggregator streams application logs to the client. Metric collector collects the statistics that is used by operator to monitor cloud foundry deployment.

2.1.3. 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** called **Bits Service** [Mara] . It encapsulates all the "bits operations" into 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.

2.1.4. 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 and they 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.
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 or an external like S3. For the purpose of implementation and evaluation we have used S3 as our blob store.

2.1.5. Bits Service

3. Suggested Algorithms

This chapter deals with the explanation of suggested algorithms. In the following sections we describe the problem and various suggested solutions. Before looking at the solutions let's try to have a detailed look into the problem that we are trying to solve.

3.1. Problem

This section describes the problem that we are trying to solve through this thesis. Suppose we have a blob on two different machines A and B. Then we change the blob of machine A and we want that blob on machine B should get those changes. This is the problem of 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 blob on one machine identical to a blob 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 assumptions made by few algorithms was regarding the database at bits service side, they assumed that the blobs were stored in a local cache on the bits service.

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 with bits service. We are trying to extend this topic of resource matching a bit by solving the issue of blob matching for application blobs. A computer application consists of many blobs depending on the type of application like if: 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 blobs and folders created by different programming language.

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

3. Suggested Algorithms

Since each type of programming language has its own type of folder structure and can have a range of different number of blob, we have to consider this factor while solving the issue. Another factor that needs to be considered while solving the issue of blob synchronization is that application blobs and folders are not stored on a local cache on the bits service but they are stored on a Blobstore. We are looking for a solution to a problem of blob 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 blobs 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.

3.1.1. 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. **State at client:** We do not store the state of bits service at client. By state we mean metadata about the blobs that are already stored on the bits service.
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.

3.2. V2 Service

In this section we discuss the existing V2 algorithm^{3.1} 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 3.1, 3.2.

Algorithm 3.1 Client

```

hash[]  $\leftarrow$  SHA of each blob
for each blob  $\in$  Application do
  hash[]  $\leftarrow$  Digest::SHA1.hexdigest blob
end for

knownhash[]  $\leftarrow$  PUT /match hash[]
for each sha  $\in$  Knownhash[] 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 3.2 bits service

```

/match do
  knownhash[]  $\leftarrow$ 
  SHA known by bits service
  bits service receives List of SHAs
  for each sha  $\in$  List 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  $\in$  Folder do
    if blob.size > 64kB then
      blobhash  $\leftarrow$  hashofblob
      PUT blobhash, blobcontent to Blobstore
    end if
  end for
end /bits

```

3.2.1. ALGORITHM:

As we can see in the figure 3.1 in the first 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. Client

3. Suggested Algorithms

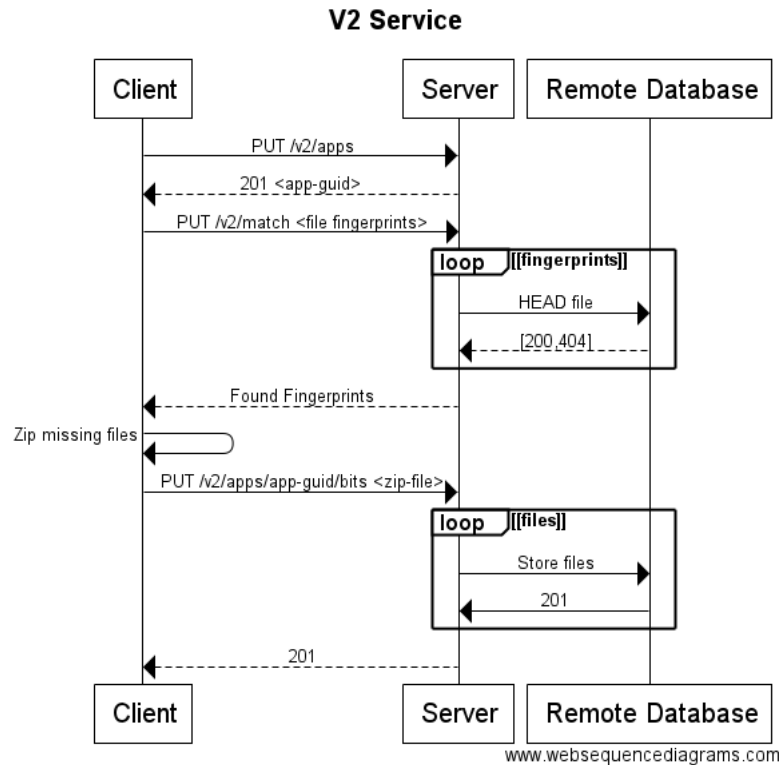


Abbildung 3.1.: V2 Service

then calculates SHAs of all the blobs in the application and sends this list to bits service. bits service upon receiving the list of SHAs makes HEAD request to the Blobstore. If the hash exist on the Blobstore it returns a status code of 200 else it returns a code of 404. For all the hashes, for which bits service received a 200, it keeps them in a list and sends that list to the client. Client upon receiving list of known hashes by bits service, creates a zip blob of the blobs that were unknown by bits service and makes a PUT request with the zip blob. bits service unzips the zip blob and calculate SHAs of all the blobs and calculates SHA of each blob. For each blob, if the size of blob is more than 64KB it sends its SHA as its name, its path and content to remote bits service else the blobs are not saved on bits service. bits service after string all the blobs respond with a 201 status code.

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 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 makes a POST request for each of them and saves them on Blobstore. 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.

3.2.2. 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 they save 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. Since this approach does not store

3. Suggested Algorithms

blobs greater than 64KB, client needs to send all the blobs that are less than or equal to 64KB every time. 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. 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.

3.2.3. 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 Sycth suggested an approach which we would be discussing in upcoming sections.

3.3. SYCTH'S APPROACH

To overcome the shortcomings of above approach, Sycth suggested an algorithm where he introduced a local data store 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, update the data store access time and increment the hit count. The initial data store 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 score 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 overcome the major shortcomings of the existing approach. Since all the hashes 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 now save all the application blobs on the database, irrespective of size. 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 3.3 and bits service 3.4.

Algorithm 3.3 Client

```

hash[]  $\leftarrow$  SHA of each blob
for each blob  $\in$  Application do
    hash[]  $\leftarrow$  Digest::SHA1.hexdigest blob
end for

unknownhash[]  $\leftarrow$  PUT /match hash[]
for each sha  $\in$  Unknownhash[] do
    folder  $\leftarrow$  blob corresponding to sha
end for
zipblob  $\leftarrow$  zip(folder)

PUT /bits zipblob

```

Algorithm 3.4 bits service

```

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

/bits do
    folder  $\leftarrow$  unzip(zipblob)
    for each blob  $\in$  Folder do
        blobhash  $\leftarrow$  hashofblob
        PUT blobhash, blobcontent to remote
        database
    end for
end /bits

```

3.3.1. ALGORITHM:

Figure 3.2 explains Sycth's approach where 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. Client then calculates SHAs of all the blobs in the application and sends this list to bits service. bits service queries its local cache, 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, else 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. Client receives the list of unknown hashes and creates a folder with all the blobs that corresponds to those hashes

3. Suggested Algorithms

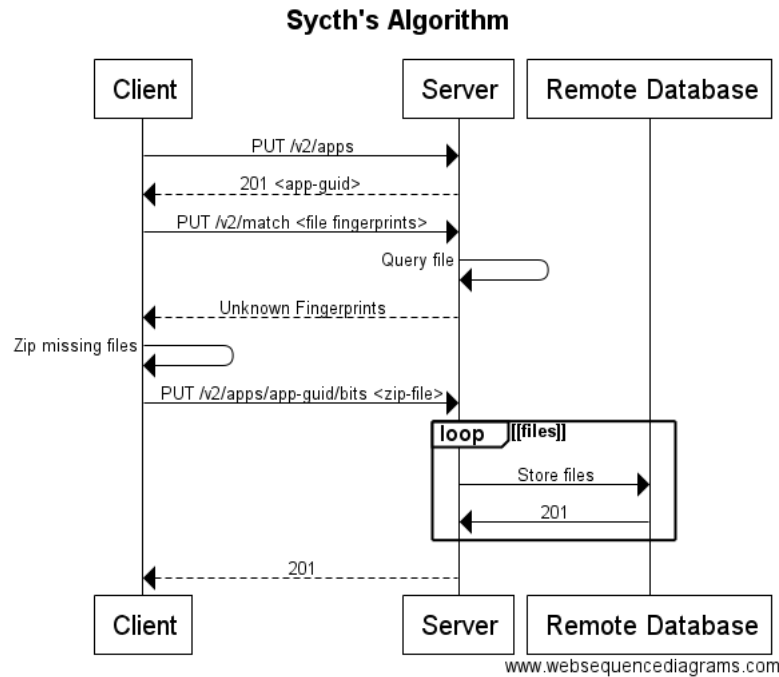


Abbildung 3.2.: Sycth's Algorithm

after adding all the unknown blobs to a folder, client creates a zip blob of the folder and makes a PUT request to bits service with the zip blob. bits service unzips the zip blob and for each blob it creates a hash and saves that hash into its local cache along with other metadata like hit count, creation date and size of the blob. 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: $O(n) + O(n) + O(n) = O(n)$

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 takes $O(n)$ 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 takes $O(n)$ 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.

3.3.2. 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 to the bits service. It saves all the HEAD requests and does not require to send blobs less than 64KB every time. This approach has same time complexity as that of V2 Service3.1 but has a different and greater space complexity than V2 Service3.1. 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.

3.3.3. 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.

3.4. 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 ?? literature review, 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 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 3.1.1. 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 3.1 3.2. Following is the algorithm describing client 3.5 and bits service 3.6.

Algorithm 3.5 Client

```

hash[]  $\leftarrow$  SHA of each blob and directory
for each directory  $\in$  Application do
    hash[]  $\leftarrow$  createTree()
end for

unknownhash[]  $\leftarrow$  PUT /match hash[]
for each sha  $\in$  Unknownhash[] do
    folder  $\leftarrow$  blob corresponding to sha
end for
zipblob  $\leftarrow$  zip(folder)

PUT /bits zipblob

```

Algorithm 3.6 bits service

```

/match do
    unknownhash[]  $\leftarrow$  SHA not known by
    bits service
    bits service receives List of SHAs
    for each sha  $\in$  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[]  $\leftarrow$  sha
            end if
        else
            bits service makes HEAD request to remote
            database
            if request.status = 404 then
                unknownhash[]  $\leftarrow$  sha
            end if
        end if
    end for
    return unknownhash[]
end /match

/bits do
    folder  $\leftarrow$  unzip(zipblob)
    for each blob  $\in$  Folder do
        blobhash  $\leftarrow$  hashofblob
        PUT blobhash, blobcontent to remote
        database
    end for
end /bits

```

3.4.1. ALGORITHM:

The diagram 3.3 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 3.4 of an application directory. Here the "Root" represents

3. Suggested Algorithms

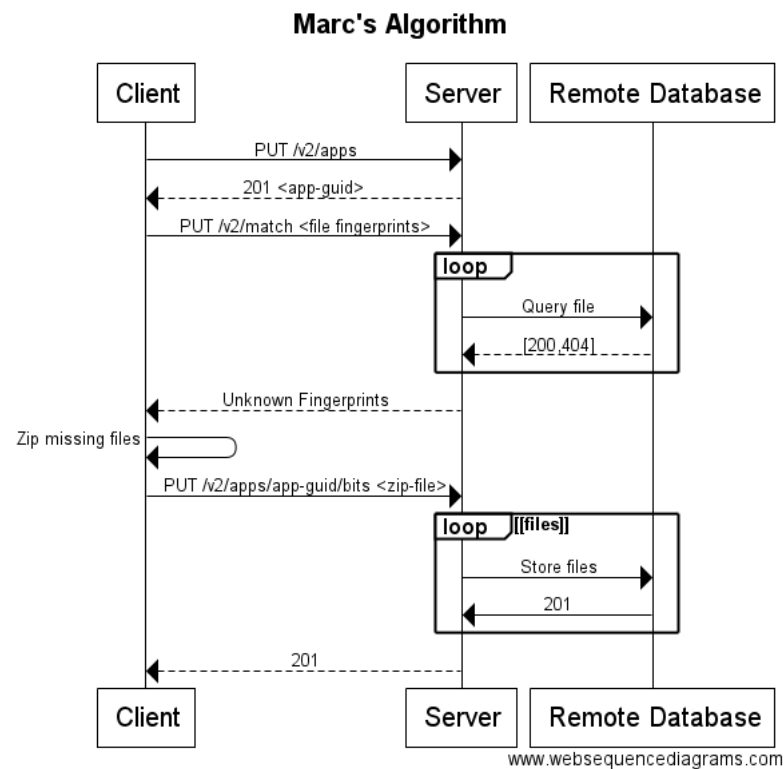


Abbildung 3.3.: Marc's Algorithm

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

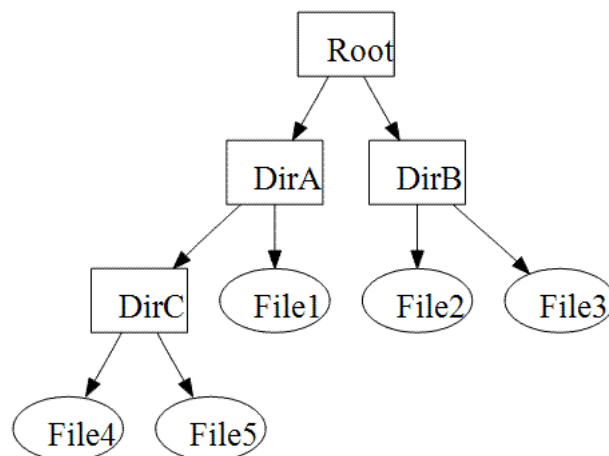


Abbildung 3.4.: Directory Structure [Ash]

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"]}
}
```

Client sends this JSON along with other details like GUID to the bits service. bits service receives this JSON and for each key, if it is a directory, bits service asks Blobstore for all the blobs under this directory using a HEAD request. After receiving all the blobs from the Blobstore it will calculate SHA of each blob and a joint SHA of the directory. After computing the joint SHA, bits service will match it against the SHA received from client. If it matches it will look into the sibling directories of that directory if there are any, else it will then store this SHA and path for future use and look into the children of that SHA.

For eg., After receiving the above JSON, bits service will take the first key i.e. "i", it will look into its path and ask the client for all the blobs under Root Folder. After receiving all the blobs it will repeat the same procedure as client, for all the blobs with same path, it will calculate SHAs and joint SHAs and store them into a local flat blob. After storing the SHAs, it will compare the SHAs from the client. If "i" is present inside the list of SHA on flat blob, then bits service does not match further and inform client that the blobs sent by client are already on the bits service. Else it looks into the children of "i" i.e. "g" and "h" and if either or both of them doesn't match it will continue the process until it finds the blobs that are changed. After this step bits service send

3. Suggested Algorithms

all the unknown SHAs to client and client creates a zip blob of all the unknown blobs. After the zip is ready it is sent to the bits service where it unzips and assemble the entire package.

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 $O(n^2)$ 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: $O(1)+O(n^2)+O(n) = O(n^2)$

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:

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 Blobstore:** When the bits service receives list of SHAs from client, it queries Blobstore for all blobs. And for each blob it calculates path and hash. Then it repeats the process of calculation of joint hash. This operation take $O(n)+O(n^2)$ time, where n is number of SHAs sent by client.
3. **Calculate unkown SHAs :** After having a list of directory and blob SHAs, bits service makes comparison with SHA sent by client. This takes $O(\log n)$ time, where n is number of SHAs.
4. **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: $O(n) + O(n^2) + O(n) = O(n^2)$, 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.

3.4.2. DISCUSSION:

This approach reduced the number of comparison reduced to $O(\log n)$ as compared to other approaches discussed so far. But even after this reduction in complexity of comparison the overall complexity increased. The reason of this increase is the HEAD request that bits service has to make in order to get all the blobs from Blobstore and after that it calculates hash and path of each blob. This algorithm was able to solve all of the issues of V2 service 3.1 but the estimated time was way more than it 3.1.

3.4.3. IMPROVEMENT:

The algorithm made a very important improvement for the phase of comparison of blob SHAs but it worsened total time complexity. But we can improve the overall resource matching algorithm by combining Sycth's 3.2 and Marc's 3.3 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.

3.5. COMBINED APPROACH

In this approach we are trying to have a combination of the approaches suggested by Sycth, 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 Sycth'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 $O(\log n)$.was Following is the client 3.7 and bits service 3.8 algorithms of combined approach that takes benefits of both the approaches.

3. Suggested Algorithms

Algorithm 3.7 Client

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

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

PUT /bits zipblob
```

Algorithm 3.8 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
```

3.5.1. ALGORITHM:

Figure 3.5 is the sequence diagram for the combined approach. This figure looks similar to Sycth's sequence diagram 3.2 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 3.4.1 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.

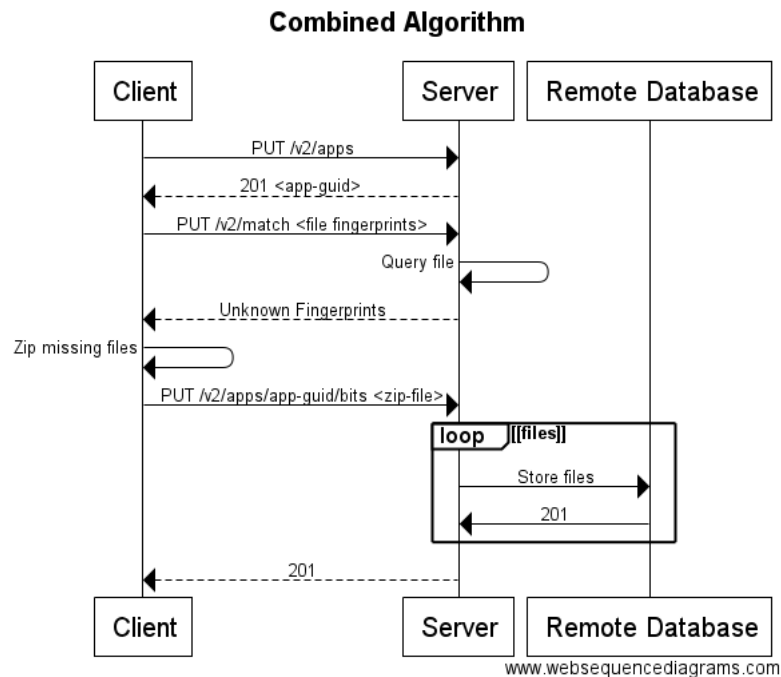


Abbildung 3.5.: Combined Algorithm

- b) If it is not the first time the application is uploaded to bits service, it will query its local cache for the first 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.
5. bits service unzips the blob received from client and for each blob it makes a POST request to Blobstore. Blobstore replies with a status code of 201 for each blob.
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.

3.5.2. SPACE AND TIME COMPLEXITY:

Time complexity at client 3.7 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.

3. Suggested Algorithms

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: $O(1)+O(n^2)+O(n) = O(n^2)$

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 3.8:

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 takes $O(n)$ 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 takes $O(\log n)$ 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.

3.5.3. DISCUSSION:

In this algorithm we saw a lot of improvements in time complexity. The time complexity reduced from $O(n)$ 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.

3.5.4. IMPROVEMENT:

Since this algorithm combined the approach of both Sycth and Marc, it had the benefits of both. The time complexity of Marc's approach was only possible because of the local cache approach of Sycth. 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. Following we suggest an improved approach which would reduce the space complexity at the bits service to $O(1)$. That is we do not need to have an additional database on bits service.

4. Kapitel zwei

Hier wird der Hauptteil stehen. Falls mehrere Kapitel gewünscht, entweder mehrmals `\chapter` benutzen oder pro Kapitel eine eigene Datei anlegen und `ausarbeitung.tex` anpassen.

LaTeX-Hinweise stehen in Anhang A.

5. Überschrift auf Ebene 0 (chapter)

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. $\sin^2(\alpha) + \cos^2(\beta) = 1$. Der Text gibt lediglich den Grauwert der Schrift an $E = mc^2$. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. $\sqrt[n]{a} \cdot \sqrt[n]{b} = \sqrt[n]{ab}$. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. $\frac{\sqrt[n]{a}}{\sqrt[n]{b}} = \sqrt[n]{\frac{a}{b}}$. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. $a\sqrt[n]{b} = \sqrt[n]{a^n b}$. Er muss keinen Sinn ergeben, sollte aber lesbar sein. $d\Omega = \sin\vartheta d\vartheta d\varphi$. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

5.1. Überschrift auf Ebene 1 (section)

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. $\sin^2(\alpha) + \cos^2(\beta) = 1$. Der Text gibt lediglich den Grauwert der Schrift an $E = mc^2$. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. $\sqrt[n]{a} \cdot \sqrt[n]{b} = \sqrt[n]{ab}$. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. $\frac{\sqrt[n]{a}}{\sqrt[n]{b}} = \sqrt[n]{\frac{a}{b}}$. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. $a\sqrt[n]{b} = \sqrt[n]{a^n b}$. Er muss keinen Sinn ergeben, sollte aber lesbar sein. $d\Omega = \sin\vartheta d\vartheta d\varphi$. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

5.1.1. Überschrift auf Ebene 2 (subsection)

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. $\sin^2(\alpha) + \cos^2(\beta) = 1$. Der Text gibt lediglich den Grauwert der Schrift an $E = mc^2$. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. $\sqrt[n]{a} \cdot \sqrt[n]{b} = \sqrt[n]{ab}$. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander

5. Überschrift auf Ebene 0 (chapter)

stehen und prüfe, wie breit oder schmal sie läuft. $\frac{\sqrt[n]{a}}{\sqrt[n]{b}} = \sqrt[n]{\frac{a}{b}}$. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. $a\sqrt[n]{b} = \sqrt[n]{a^n b}$. Er muss keinen Sinn ergeben, sollte aber lesbar sein. $d\Omega = \sin\vartheta d\vartheta d\varphi$. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Überschrift auf Ebene 3 (subsubsection)

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. $\sin^2(\alpha) + \cos^2(\beta) = 1$. Der Text gibt lediglich den Grauwert der Schrift an $E = mc^2$. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. $\sqrt[n]{a} \cdot \sqrt[n]{b} = \sqrt[n]{ab}$. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. $\frac{\sqrt[n]{a}}{\sqrt[n]{b}} = \sqrt[n]{\frac{a}{b}}$. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. $a\sqrt[n]{b} = \sqrt[n]{a^n b}$. Er muss keinen Sinn ergeben, sollte aber lesbar sein. $d\Omega = \sin\vartheta d\vartheta d\varphi$. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Überschrift auf Ebene 4 (paragraph) Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. $\sin^2(\alpha) + \cos^2(\beta) = 1$. Der Text gibt lediglich den Grauwert der Schrift an $E = mc^2$. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. $\sqrt[n]{a} \cdot \sqrt[n]{b} = \sqrt[n]{ab}$. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. $\frac{\sqrt[n]{a}}{\sqrt[n]{b}} = \sqrt[n]{\frac{a}{b}}$. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. $a\sqrt[n]{b} = \sqrt[n]{a^n b}$. Er muss keinen Sinn ergeben, sollte aber lesbar sein. $d\Omega = \sin\vartheta d\vartheta d\varphi$. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

5.2. Listen

5.2.1. Beispiel einer Liste (itemize)

- Erster Listenpunkt, Stufe 1
- Zweiter Listenpunkt, Stufe 1
- Dritter Listenpunkt, Stufe 1

- Vierter Listenpunkt, Stufe 1
- Fünfter Listenpunkt, Stufe 1

Beispiel einer Liste (4*itemize)

- Erster Listenpunkt, Stufe 1
 - Erster Listenpunkt, Stufe 2
 - * Erster Listenpunkt, Stufe 3
 - Erster Listenpunkt, Stufe 4
 - Zweiter Listenpunkt, Stufe 4
 - * Zweiter Listenpunkt, Stufe 3
 - Zweiter Listenpunkt, Stufe 2
- Zweiter Listenpunkt, Stufe 1

5.2.2. Beispiel einer Liste (enumerate)

1. Erster Listenpunkt, Stufe 1
2. Zweiter Listenpunkt, Stufe 1
3. Dritter Listenpunkt, Stufe 1
4. Vierter Listenpunkt, Stufe 1
5. Fünfter Listenpunkt, Stufe 1

Beispiel einer Liste (4*enumerate)

1. Erster Listenpunkt, Stufe 1
 - a) Erster Listenpunkt, Stufe 2
 - i. Erster Listenpunkt, Stufe 3
 - A. Erster Listenpunkt, Stufe 4
 - B. Zweiter Listenpunkt, Stufe 4
 - ii. Zweiter Listenpunkt, Stufe 3
 - b) Zweiter Listenpunkt, Stufe 2

2. Zweiter Listenpunkt, Stufe 1

5.2.3. Beispiel einer Liste (description)

Erster Listenpunkt, Stufe 1

Zweiter Listenpunkt, Stufe 1

Dritter Listenpunkt, Stufe 1

Vierter Listenpunkt, Stufe 1

Fünfter Listenpunkt, Stufe 1

Beispiel einer Liste (4*description)

Erster Listenpunkt, Stufe 1

Erster Listenpunkt, Stufe 2

Erster Listenpunkt, Stufe 3

Erster Listenpunkt, Stufe 4

Zweiter Listenpunkt, Stufe 4

Zweiter Listenpunkt, Stufe 3

Zweiter Listenpunkt, Stufe 2

Zweiter Listenpunkt, Stufe 1

6. Zusammenfassung und Ausblick

Hier bitte einen kurzen Durchgang durch die Arbeit.

Ausblick

...und anschließend einen Ausblick

A. LaTeX-Tipps

Probleme kann man niemals mit
derselben Denkweise lösen,
durch die sie entstanden sind.

(Albert Einstein)

Pro Satz eine neue Zeile. Das ist wichtig, um sauber versionieren zu können. In LaTeX werden Absätze durch eine Leerzeile getrennt.

Folglich werden neue Abstände insbesondere *nicht* durch Doppelbackslashes erzeugt. Der letzte Satz kam in einem neuen Absatz.

A.1. File-Encoding und Unterstützung von Umlauten

Die Vorlage wurde 2010 auf UTF-8 umgestellt. Alle neueren Editoren sollten damit keine Schwierigkeiten haben.

A.2. Zitate

Referenzen werden mittels `\cite[key]` gesetzt. Beispiel: [WCL+05] oder mit Autorenangabe: Weerawarana et al. [WCL+05].

Der folgende Satz demonstriert 1. die Großschreibung von Autorennamen am Satzanfang, 2. die richtige Zitation unter Verwendung von Autorennamen und der Referenz, 3. dass die Autorennamen ein Hyperlink auf das Literaturverzeichnis sind sowie 4. dass in dem Literaturverzeichnis der Namenspräfix „van der“ von „Wil M. P. van der Aalst“ steht. Reijers, Vanderfeesten und van der Aalst [RVA16] präsentieren eine Studie über die Effektivität von Workflow-Management-Systemen.

Der folgende Satz demonstriert, dass man mittels `label` in einem Bibliographie-Eintrag den Textteil des generierten Labels überschreiben kann, aber das Jahr und die Eindeutigkeit noch von biber generiert wird. Die Apache ODE Engine [ASF16] ist eine Workflow-Maschine, die BPEL-Prozesse zuverlässig ausführt.

Listing A.1 `lstlisting` in einer `Listings`-Umgebung, damit das Listing durch Balken abgetrennt ist

```
<listing name="second sample">
  <content>not interesting</content>
</listing>
```

Wörter am besten mittels `\enquote{...}` „einschließen“, dann werden die richtigen Anführungszeichen verwendet.

Beim Erstellen der Bibtex-Datei wird empfohlen darauf zu achten, dass die DOI aufgeführt wird.

A.3. Mathematische Formeln

Mathematische Formeln kann man *so* setzen. `symbols-a4.pdf` (zu finden auf <http://www.ctan.org/tex-archive/info/symbols/comprehensive/symbols-a4.pdf>) enthält eine Liste der unter LaTeX direkt verfügbaren Symbole. Z. B. \mathbb{N} für die Menge der natürlichen Zahlen. Für eine vollständige Dokumentation für mathematischen Formelsatz sollte die Dokumentation zu `amsmath`, <ftp://ftp.ams.org/pub/tex/doc/amsmath/> gelesen werden.

Folgende Gleichung erhält keine Nummer, da `\equation*` verwendet wurde.

$$x = y$$

Die Gleichung A.1 erhält eine Nummer:

$$x = y \tag{A.1}$$

Eine ausführliche Anleitung zum Mathematikmodus von LaTeX findet sich in <http://www.ctan.org/tex-archive/help/Catalogue/entries/voss-mathmode.html>.

A.4. Quellcode

Listing A.1 zeigt, wie man Programmlistings einbindet. Mittels `\lstinputlisting` kann man den Inhalt direkt aus Dateien lesen.

Quellcode im `<listing />` ist auch möglich.

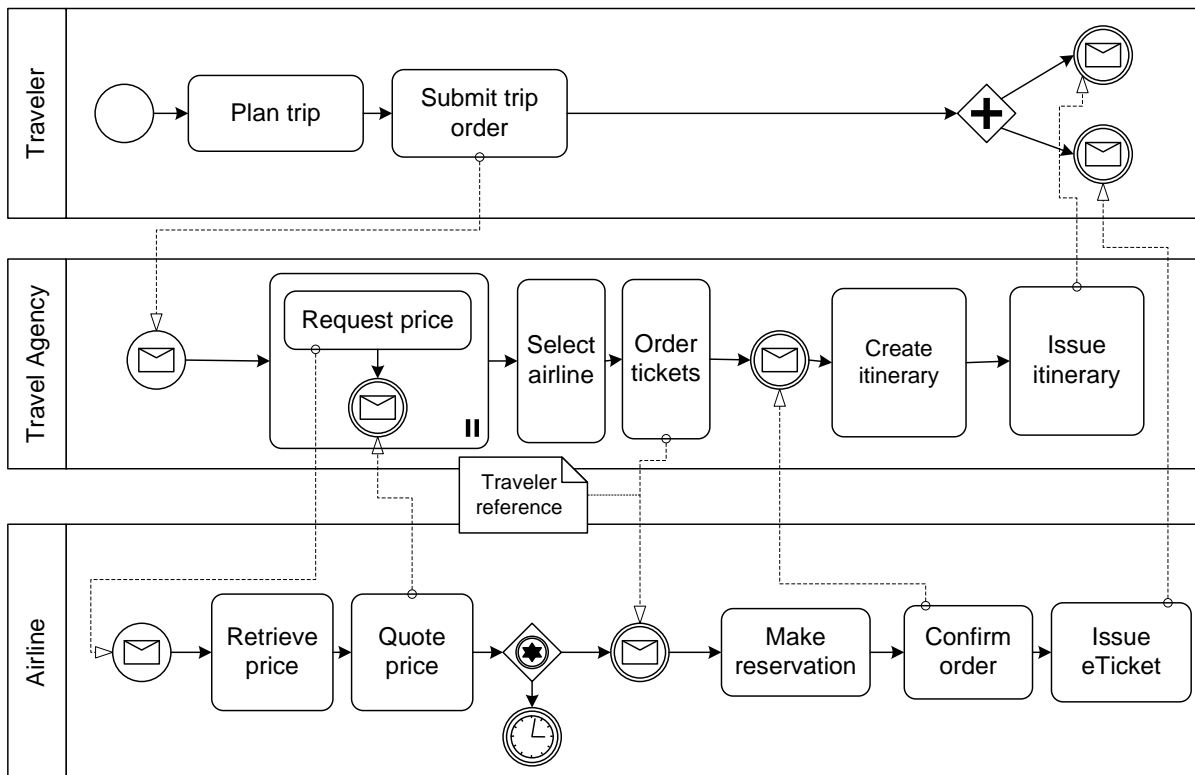


Abbildung A.1.: Beispiel-Choreographie

A.5. Abbildungen

Die Abbildung A.1 und A.2 sind für das Verständnis dieses Dokuments wichtig. Im Anhang zeigt Abbildung A.4 auf Seite 60 erneut die komplette Choreographie.

Es ist möglich, SVGs direkt beim Kompilieren in PDF umzuwandeln. Dies ist im Quellcode zu latex-tips.tex beschrieben, allerdings auskommentiert.

A.6. Tabellen

Tabelle A.1 zeigt Ergebnisse und die Tabelle A.1 zeigt wie numerische Daten in einer Tabelle representiert werden können.

A.7. Pseudocode

Algorithmus A.1 zeigt einen Beispiyalgorithmus.

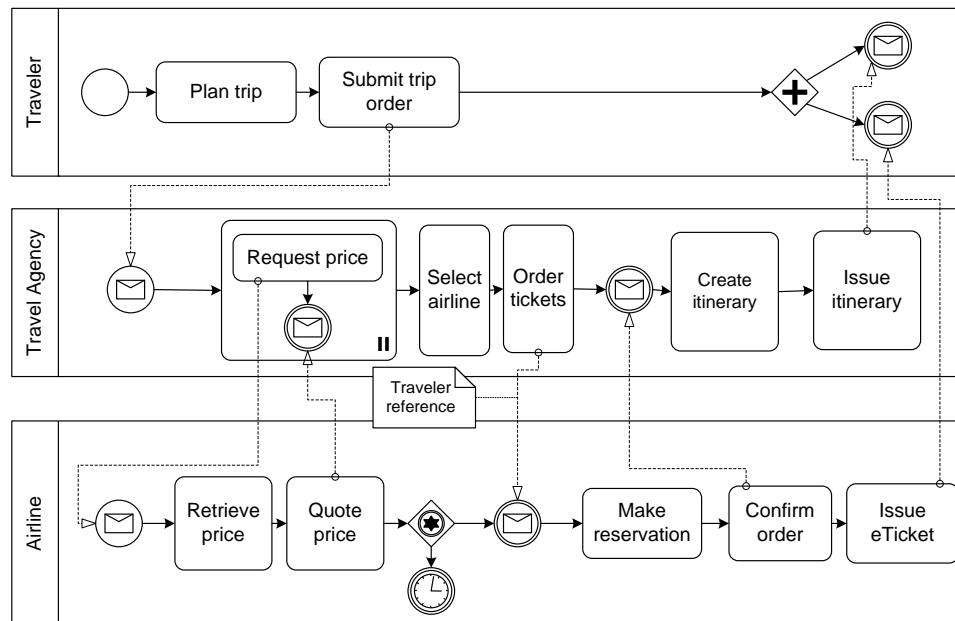


Abbildung A.2.: Die Beispiel-Choreographie. Nun etwas kleiner, damit `\textwidth` demonstriert wird. Und auch die Verwendung von alternativen Bildunterschriften für das Verzeichnis der Abbildungen. Letzteres ist allerdings nur Bedingt zu empfehlen, denn wer liest schon so viel Text unter einem Bild? Oder ist es einfach nur Stilsache?

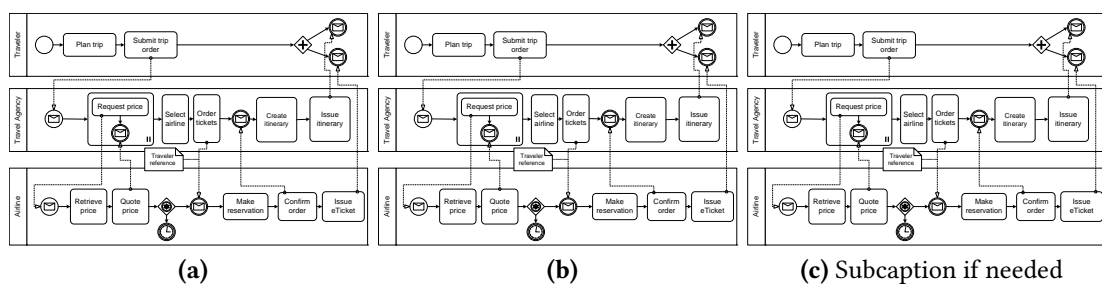


Abbildung A.3.: Beispiel um 3 Abbildung nebeneinander zu stellen nur jedes einzeln referenzieren zu können. Abbildung A.3b ist die mittlere Abbildung.

zusammengefasst		Titel
Tabelle	wie	in
tabsatz.pdf	empfohlen	gesetzt
Beispiel	ein schönes Beispiel für die Verwendung von „multirow“	

Tabelle A.1.: Beispieltabelle – siehe <http://www.ctan.org/tex-archive/info/german/tabsatz/>

Bedingungen	Parameter 1		Parameter 2		Parameter 3		Parameter 4	
	M	SD	M	SD	M	SD	M	SD
W	1,1	5,55	6,66	,01				
X	22,22	0,0	77,5	,1				
Y	333,3	,1	11,11	,05				
Z	4444,44	77,77	14,06	,3				

Tabelle A.2.: Beispieltabelle für 4 Bedingungen (W-Z) mit jeweils 4 Parameters mit (M und SD). Hinweis: immer die selbe anzahl an Nachkommastellen angeben.

Algorithmus A.1 Sample algorithm

```

procedure SAMPLE( $a, v_e$ )
  parentHandled  $\leftarrow (a = \text{process}) \vee \text{visited}(a'), (a', c, a) \in \text{HR}$ 
  //  $(a', c'a) \in \text{HR}$  denotes that  $a'$  is the parent of  $a$ 
  if parentHandled  $\wedge (\mathcal{L}_{in}(a) = \emptyset \vee \forall l \in \mathcal{L}_{in}(a) : \text{visited}(l))$  then
    visited( $a$ )  $\leftarrow$  true
    writeso( $a, v_e$ )  $\leftarrow$   $\begin{cases} \text{joinLinks}(a, v_e) & |\mathcal{L}_{in}(a)| > 0 \\ \text{writes}_o(p, v_e) & \exists p : (p, c, a) \in \text{HR} \\ (\emptyset, \emptyset, \emptyset, false) & \text{otherwise} \end{cases}$ 
    if  $a \in \mathcal{A}_{basic}$  then
      HANDLEBASICACTIVITY( $a, v_e$ )
    else if  $a \in \mathcal{A}_{flow}$  then
      HANDLEFLOW( $a, v_e$ )
    else if  $a = \text{process}$  then // Directly handle the contained activity
      HANDLEACTIVITY( $a', v_e$ ),  $(a, \perp, a') \in \text{HR}$ 
      writes•( $a$ )  $\leftarrow$  writes•( $a'$ )
    end if
    for all  $l \in \mathcal{L}_{out}(a)$  do
      HANDLELINK( $l, v_e$ )
    end for
  end if
end procedure

```

Und wer einen Algorithmus schreiben möchte, der über mehrere Seiten geht, der kann das nur mit folgendem **üblen** Hack tun:

Algorithmus A.2 Description

code goes here
test2

A.8. Abkürzungen

Beim ersten Durchlauf betrug die Fehlerrate (FR) 5. Beim zweiten Durchlauf war die FR 3. Die Pluralform sieht man hier: error rates (ERs). Um zu demonstrieren, wie das Abkürzungsverzeichnis bei längeren Beschreibungstexten aussieht, muss hier noch Relational Database Management Systems (RDBMS) erwähnt werden.

Mit `\gls{...}` können Abkürzungen eingebaut werden, beim ersten Aufrufen wird die lange Form eingesetzt. Beim wiederholten Verwenden von `\gls{...}` wird automatisch die kurz Form angezeigt. Außerdem wird die Abkürzung automatisch in die Abkürzungsliste eingefügt. Mit `\glspl{...}` wird die Pluralform verwendet. Möchte man, dass bei der ersten Verwendung direkt die Kurzform erscheint, so kann man mit `\glsunset{...}` eine Abkürzung als bereits verwendet markieren. Das Gegenteil erreicht man mit `\glsreset{...}`.

Definiert werden Abkürzungen in der Datei *content*
ausarbeitung.tex mithilfe von `\newacronym{...}{...}{...}`.

Mehr Infos unter: <http://tug.ctan.org/macros/latex/contrib/glossaries/glossariesbegin.pdf>

A.9. Verweise

Für weit entfernte Abschnitte ist „`varioref`“ zu empfehlen: „Siehe Anhang A.3 auf Seite 54“. Das Kommando `\vref` funktioniert ähnlich wie `\cref` mit dem Unterschied, dass zusätzlich ein Verweis auf die Seite hinzugefügt wird. `vref`: „Anhang A.1 auf Seite 53“, `cref`: „Anhang A.1“, `ref`: „A.1“.

Falls „`varioref`“ Schwierigkeiten macht, dann kann man stattdessen „`cref`“ verwenden. Dies erzeugt auch das Wort „Abschnitt“ automatisch: Anhang A.3. Das geht auch für Abbildungen usw. Im Englischen bitte `\Cref{...}` (mit großem „C“ am Anfang) verwenden.

A.10. Definitionen

Definition A.10.1 (Title)

Definition Text

Definition A.10.1 zeigt ...

A.11. Fußnoten

Fußnoten können mit dem Befehl `\footnote{...}` gesetzt werden¹. Mehrfache Verwendung von Fußnoten ist möglich indem man zu erst ein Label in der Fußnote setzt `\footnote{\label{...}...}` und anschließend mittels `\cref{...}` die Fußnote erneut verwendet¹.

A.12. Verschiedenes

Ziffern (123 654 789) werden schön gesetzt. Entweder in einer Linie oder als Minuskel-Ziffern. Letzteres erreicht man durch den Parameter `osf` bei dem Paket `libertine` bzw. `mathpazo` in `fonts.tex`.

KAPITÄLCHEN werden schön gesperrt...

- I. Man kann auch die Nummerierung dank `paralist` kompakt halten
- II. und auf eine andere Nummerierung umstellen

A.13. Weitere Illustrationen

Abbildungen A.4 und A.5 zeigen zwei Choreographien, die den Sachverhalt weiter erläutern sollen. Die zweite Abbildung ist um 90 Grad gedreht, um das Paket `pdfscape` zu demonstrieren.

¹Diese Fußnote ist ein Beispiel.

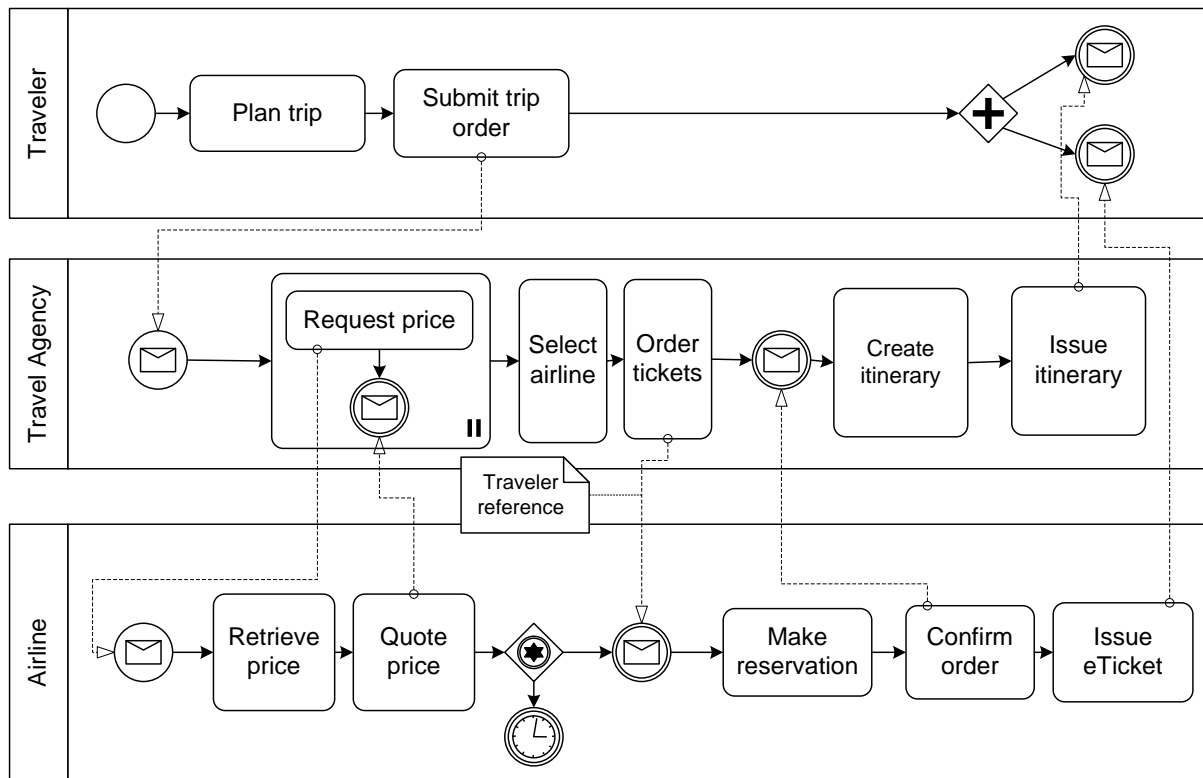


Abbildung A.4.: Beispiel-Choreographie I

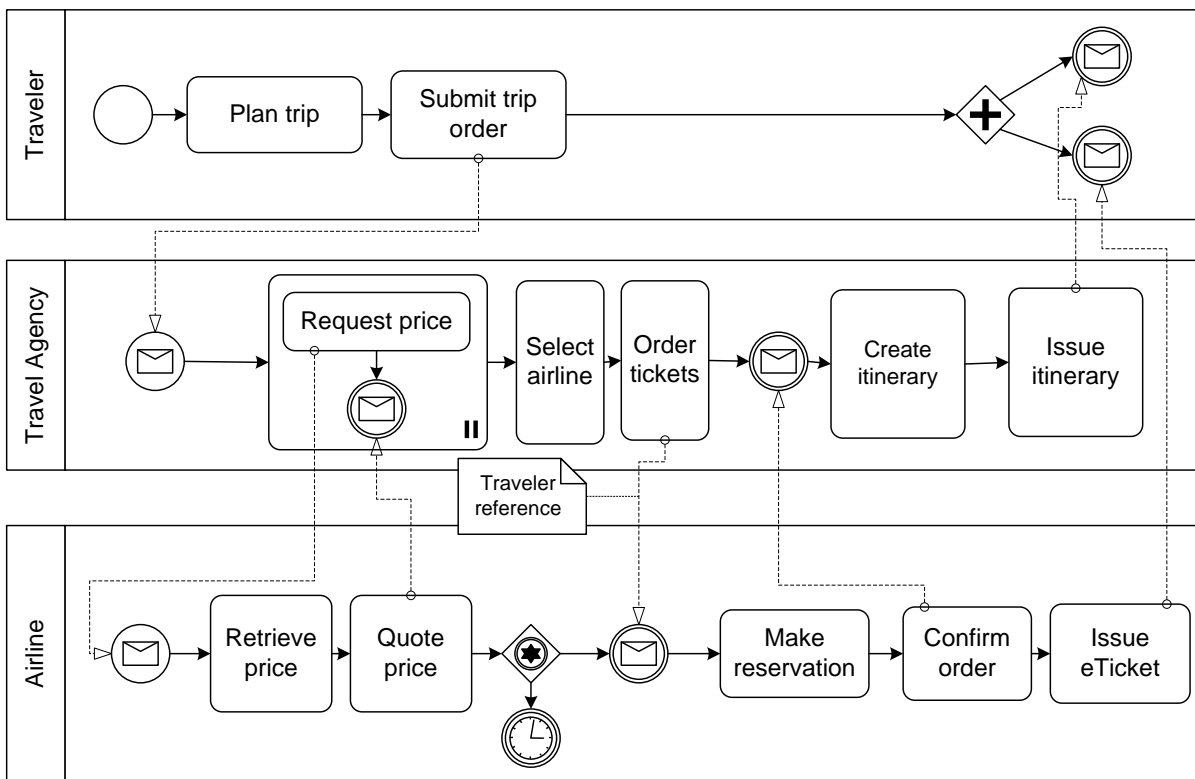


Abbildung A.5.: Beispiel-Choreographie II

A.14. Plots with pgfplots

Pgfplot ist ein Paket um Graphen zu plotten ohne den Umweg über gnuplot oder matplotlib zu gehen.

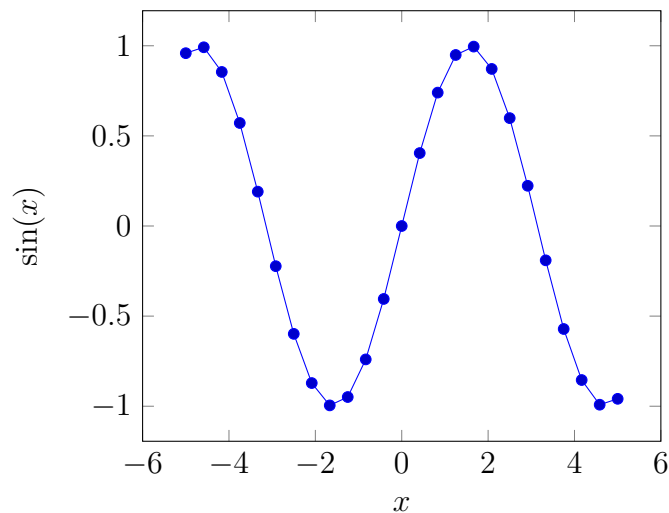


Abbildung A.6.: $\sin(x)$ mit pgfplots.

A.15. Figures with tikz

TikZ ist ein Paket um Zeichnungen mittels Programmierung zu erstellen. Dieses Paket eignet sich um Gitter zu erstellen oder andere regelmäßige Strukturen zu erstellen.

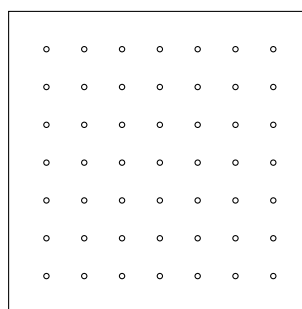


Abbildung A.7.: Eine tikz-Graphik.

A.16. Schlusswort

Verbesserungsvorschläge für diese Vorlage sind immer willkommen. Bitte bei GitHub ein Ticket eintragen (<https://github.com/latextemplates/uni-stuttgart-computer-science-template/issues>).

Literaturverzeichnis

- [Ang14] Angel Tomala-Reyes. *What is IBM Bluemix?* 2014. URL: <https://www.ibm.com/developerworks/cloud/library/cl-bluemixfoundry/> (zitiert auf S. 19, 20).
- [ASF16] The Apache Software Foundation. *Apache ODE™ – The Orchestration Director Engine*. 2016. URL: <http://ode.apache.org> (zitiert auf S. 53).
- [Ash] Ashutosh Phoujdar. URL: <https://www.codeproject.com/Articles/26628/Tree-structure-generator> (zitiert auf S. 36).
- [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. 20).
- [Clo17b] Cloud Foundry Foundation. *Cloud Foundry Documentation*. 2017. URL: <https://docs.cloudfoundry.org/concepts/architecture/> (zitiert auf S. 21, 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).
- [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. 20).
- [RVA16] H. Reijers, I. Vanderfeesten, W. van der Aalst. „The effectiveness of workflow management systems: A longitudinal study“. In: *International Journal of Information Management* 36.1 (Feb. 2016), S. 126–141. DOI: [10.1016/j.ijinfomgt.2015.08.003](https://doi.org/10.1016/j.ijinfomgt.2015.08.003) (zitiert auf S. 53).
- [WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. ISBN: 0131488740. DOI: [10.1.1/jpb001](https://doi.org/10.1.1/jpb001) (zitiert auf S. 17, 53).

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