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

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.

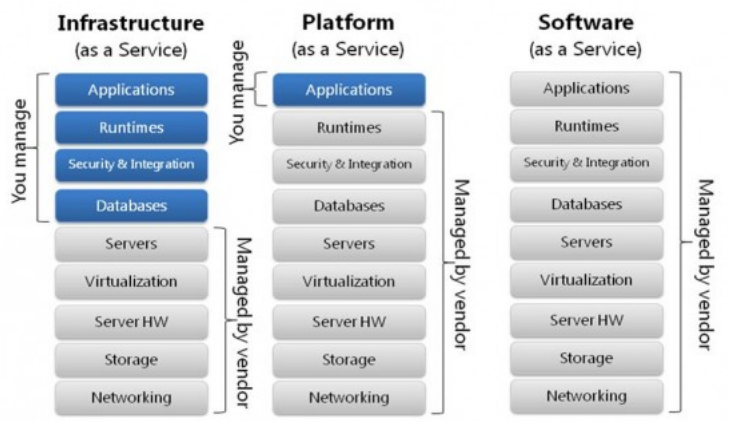


Abbildung 2.1.: Types of Service Models [Kat13]

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.

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 lets try to have a detailed look into the problem that we are trying to save.

3.1. Problem

This section describes the problem that we are trying to solve through this thesis. Suppose we have a file on tow different machines A and B. Then we change the file of machine A and we want that file 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 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 assumptions made by few algorithms was 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 server that is a PaaS cloud. When a client creates an application it pushes the application files to the cloud and every time it makes changes to its application files, it pushes the changed application to the cloud again so that it can synchronize the files with server. We are trying to extend this topic of resource matching a bit by solving the issue of file matching for application files. A computer application consists of many files 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 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 file, 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 remote database. 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 files present on database.

Different types of folder structure, different number of files and a remote database are the additional problems that needs to be dealt with while designing an algorithm that would upload application bits from client to server. 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 server 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 server. For eg., in case of a java application there are a millions of small files 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 server at client. By state we mean metadata about the files that are already stored on the server.
3. **Storage:** We assume to have unlimited storage at the remote database. 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 server.

3.2. V2 Service

In this section we discuss the existing V2 algorithm3.1 and how it helps to solve the problem. To upload an application the client computes the SHA1s of all the files in the application. The client then sends a “match” request to the server which includes the SHA1s of all the files. The

server responds by saying which of those files are already available in the remote database. The client then uploads the remaining files from the application as a zip file. The server then reconstitutes the application using the zip file and the relevant files from the remote database. The server database is typically configured to store files greater than 64 KB in size, so the vast majority of small files, e.g., .class files, are not stored in the remote database. However, the client is not currently aware of this and includes all files (by SHA1) in its resource matching query to the server. Since these files are not matched, the CLI then stores them in the application zip file in the request to upload the application to the server. Following is the algorithm for V2 Service client and server 3.1, 3.2.

Algorithm 3.1 Client

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

$knowhash[] \leftarrow$ **PUT /match** *hash[]*
for each $sha \in Knownhash[]$ **do**
  if $hash[]$ does not include *sha* **then**
    $folder \leftarrow$ file corresponding to *sha*
  end if
end for
$zipfile \leftarrow$ *zip(folder)*

PUT /bits *zipfile*

Algorithm 3.2 Server

/match do
$knownhash[] \leftarrow$ SHA known by server
Server receives *List of SHAs*
for each $sha \in List$ **do**
  Server 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(zipfile)*
for each $file \in Folder$ **do**
  if $file.size >$ *64kB* **then**
    $filehash \leftarrow hashoffile$
    PUT *filehash, filecontent* to remote database
  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 server and the server creates a new GUID for the client and sends the GUID with status 201. If the server already knows the application it sends the GUID with status 200. Client then calculates SHAs of all the files in the application and sends this list to server. Server upon receiving the

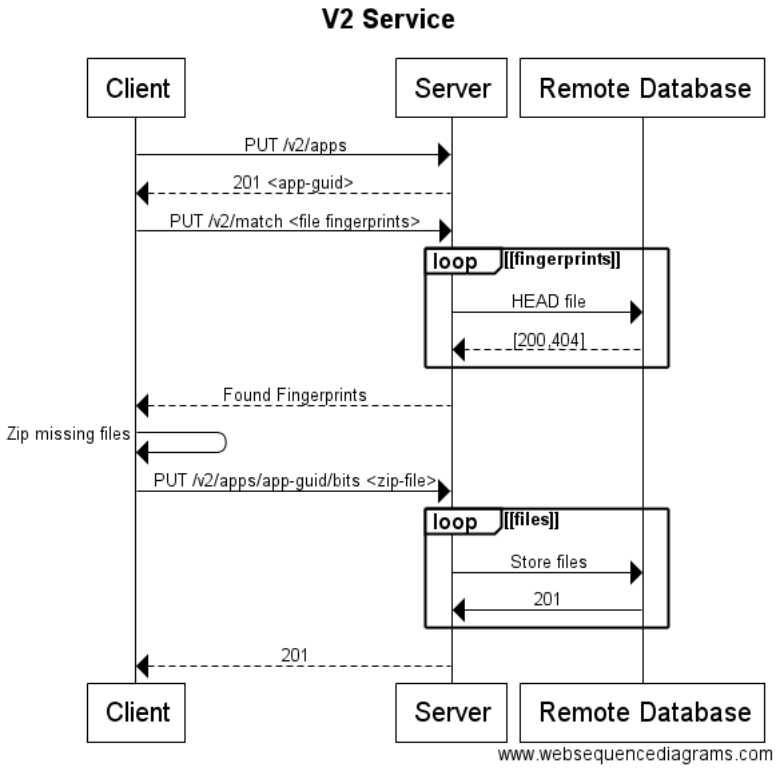


Abbildung 3.1.: V2 Service

list of SHAs makes HEAD request to the remote database. If the hash exist on the remote database it returns a status code of 200 else it returns a code of 404. For all the hashes, for which server received a 200,it keeps them in a list and sends that list to the client. Client upon receiving list of known hashes by server, creates a zip file of the files that were unknown by server and makes a PUT request with the zip file. Server unzips the zip file and calculate SHAs of all the files and calculates SHA of each file.For each file,if the size of file is more than 64KB it sends its SHA as its name, its path and content to remote server else the files are not saved on server.Server after string all the files 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 server.
2. **Calculates SHA of application files:** In this operation the client goes through each file and calculates SHA of each file. It takes O(n) where n is the number of files.

3. **Calculates unknown SHA at server:** After receiving known list of SHAs from server, client goes through the list of SHAs that it already has and calculates the SHAs that are not present at server. Thus this operation takes O(n) where n is number of application files.
4. **Calculates zip file:** After calculating unknown file, client puts all unknown file into a zip file. 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 server we need to look at the time taken to perform each operation. Following are the operations performed on server side:

1. **Finds the GUID corresponding to application name:** When server 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 server,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 server.
2. **Makes HEAD request to remote database:** When the server receives list of SHAs from client, it makes a HEAD request for each SHA against the remote database. This operation take O(n) time, where n is number of SHAs sent by client.
3. **Saves changed/new files on remote database:** When server receives changed/new files from client, it makes a POST request for each of them and saves them on remote database. This operation takes O(n) time, where n is number of changed/new files.

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

3.2.2. DISCUSSION:

Since this algorithm saves only the files with size greater than 64KB in remote database, first upload to the remote database is faster as compared to the other algorithms discussed in following section(as they save all files). For the files with size less than 64KB the server does not have to make POST request to remote database to save the file. This saves some storage space, bandwidth from server to Remote database. This approach was taken because it was not possible to store all the files on remote database 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 remote database as it would have caused large number of writes on every upload. Since this approach does not store files greater than 64KB, client needs to send all the files that are less than or equal to 64KB every time. For eg., zipkin web Jar contains over 16000 files,including over 15,600 .class files. 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 server and only 10-15 percent to do the resource match. In this approach server sends a list SHAs that are already present on it, client after receiving this list calculates the SHAs that are unknown to server. 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 files. If a file is deleted by client, it does not get deleted at the server which leads to unnecessary load on remote database.

3.2.3. IMPROVEMENTS:

This algorithm does not store files of size less than 64KB and hence every time all the files with size greater than or equal to 64KB have to be uploaded to server. Most of the time only few of these files have changed and we lose a lot of bandwidth from client to server 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 server that can hold the hash of the file, the size of the file, 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 server. The last accessed time should be the create time from the remote database 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 remote database and have the metadata necessary to determine what file 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 hashes are stored with metadata on a local database, all the HEAD request made to remote database (to get metadata about the files stored on it)can be saved. Another improvement, we can now save all the application files on the database, irrespective of size. In the previous approach it was not possible to store all the files because it increased the number of reads from the remote database. In this approach, since we need to read the metadata of the file, we can do this by querying the local database. Since this approach uploads everything on the first push, we have saved a lot of bandwidth that was used by previous approach to send files less than 64KB on every upload.Now only those files are uploaded that are actually changed. Another small improvement was over the calculation of difference that was made by both client and server, 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 server 3.4.

Algorithm 3.3 Client

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

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

PUT /bits zipfile
```

Algorithm 3.4 Server

```
/match do
unknownhash[] ← SHA not known by
server
Server receives List of SHAs
for each sha ∈ List do
    queries local database
    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(zipfile)
for each file ∈ Folder do
    filehash ← hashoffile
    PUT filehash, filecontent 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 server and the server creates a new GUID for the client and sends the GUID with status 201. If the server already knows the application it sends the GUID with status 200. Client then calculates SHAs of all the files in the application and sends this list to server. Server queries its local database, if it is the first upload of the application bits it saves list of hash in its local database and return the list to client as a list of unknown hash, else it queries its local database for each hash in the list and for all the hash that are not present in the local database the server sends them as a list of unknown hashes. Client receives the list of unknown hashes and creates a folder with all the files that corresponds to those hashes after adding all the unknown files to a folder, client creates a zip file of the folder and makes a PUT request to server with the zip file. Server unzips the zip file and for each file it creates a hash and saves

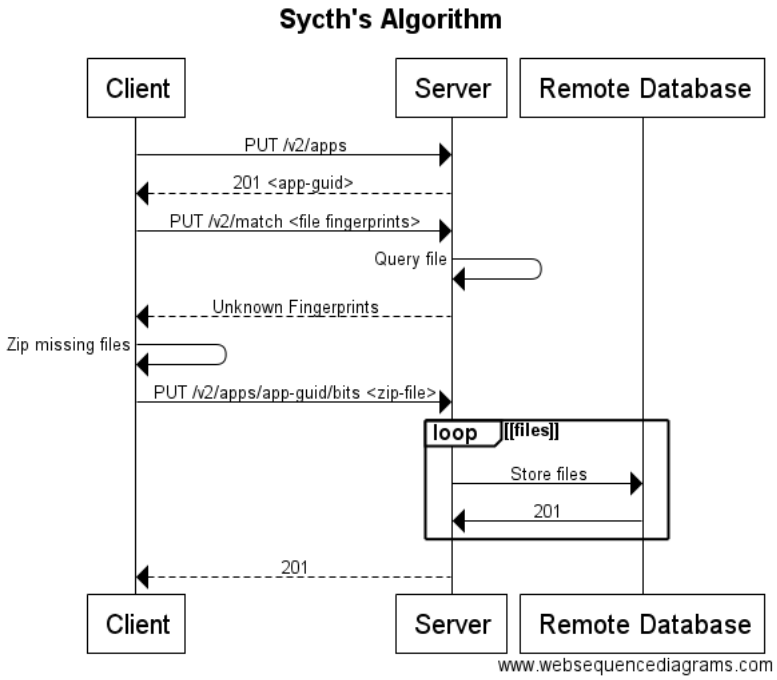


Abbildung 3.2.: Sycth's Algorithm

that hash into its local database along with other metadata like hit count, creation date and size of the file. Then it changes the name of file to its hash and makes a PUT request to remote database to store the files.Remote database replies with a 201 for each file and after all the files are saved on remote database server 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 server.
2. **Calculates SHA of application files:** In this operation the client goes through each file and calculates SHA of each file. It takes O(n) where n is the number of files.
3. **Calculates zip file:** After receiving list of unknown files, client puts all unknown file into a zip file. 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 server we need to look at the time taken to perform each operation. Following are the operations performed on server side:

1. **Finds the GUID corresponding to application name:** When server 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 server,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 server.
2. **Query local database:** When the server receives list of SHAs from client, it queries for each SHA against the local database. This operation take O(n) time, where n is number of SHAs sent by client.
3. **Saves changed/new files on remote and local database:** When server receives changed/new files from client, it makes a POST request for each of them and saves them on remote and local database. This operation takes O(n) time, where n is number of changed/new files.

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

3.3.2. DISCUSSION:

This algorithm saves SHAs of all the applications on a local database. This is a huge improvement over the previous approach as this saves a lot of time uploading bits to the server. It save all the HEAD requests and does not require to send files 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 database on the server,it leads to a greater space complexity as the number of applications increase on server. This space complexity would increase with the number of applications on the server.

3.3.3. Improvement:

Since this is a solution for uploading bits to a PaaS cloud, scaling the local database could be an issue in future as the number of applications increase on server. 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 files then it joins the SHA for all the files 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 server using a Bloom Filter**??**. In its third phase it calculates chunks of data that is changed in each file 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 file is not present on the server 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 file 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 server.

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 files 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, files inside are not matched. Chances of hash collisions are minimized as directory SHAs are combination of all the file 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 server 3.6.

Algorithm 3.5 Client

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

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

PUT /bits zipfile
```

Algorithm 3.6 Server

```
/match do
unknownhash[] ← SHA not known by
server
Server receives List of SHAs
for each sha ∈ List do
    if sha = directory then
        Server makes HEAD request to remote
        database directory and gets hash of all files
        in directory and then calculate a joint hash
        if request.status = 404 then
            unknownhash[] ← sha
        end if
    else
        Server 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(zipfile)
for each file ∈ Folder do
    filehash ← hashoffile
    PUT filehash, filecontent 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 server. 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 appli-

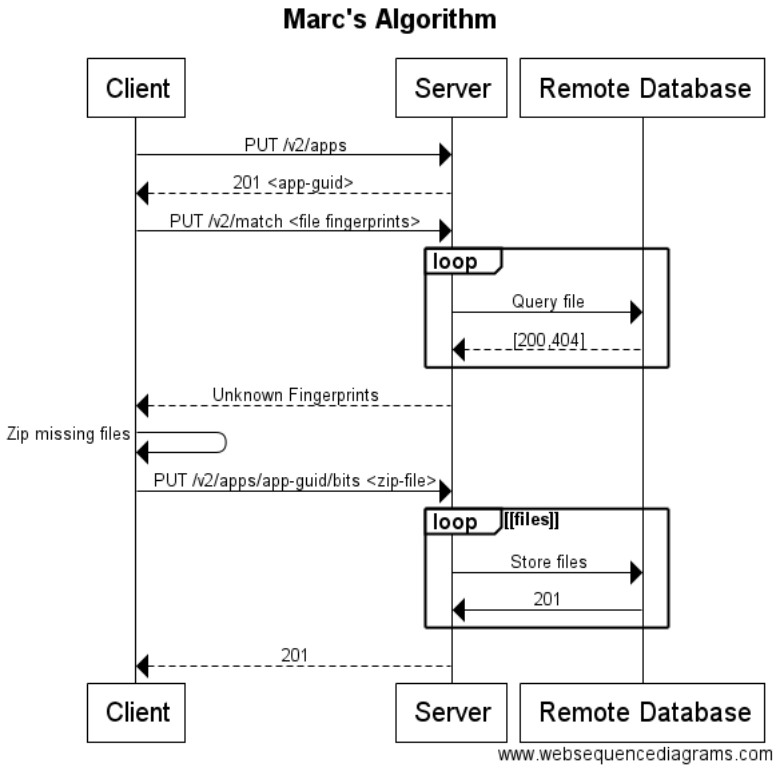


Abbildung 3.3.: Marc's Algorithm

cation directory, "DirAänd "DirB"represents directories directly under application directory. "DirCänd File1 are under DirA and so on.

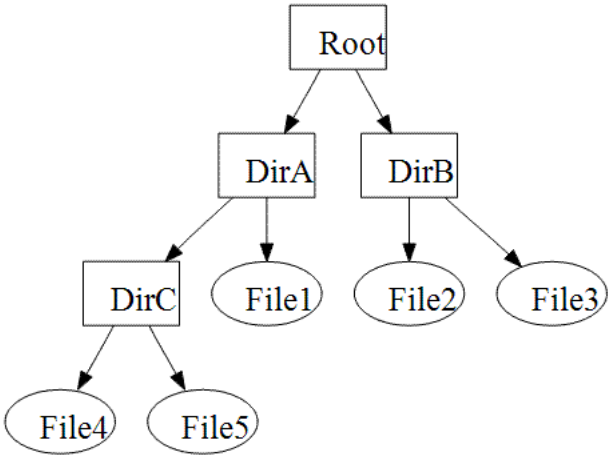


Abbildung 3.4.: Directory Structure [Ash]

Client goes through each file in the application directory and calculates their SHA and path. For all files that are have same paths, their SHAs are joined together and a directory SHA with path is calculated. The process continues until all the files 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 File1,**a**= Digest::SHA1.hexdigest file1,
SHA of File2,**b**= Digest::SHA1.hexdigest file2,
SHA of File3,**c**= Digest::SHA1.hexdigest file3,
SHA of File4,**d**= Digest::SHA1.hexdigest file4,
SHA of File5,**e**= Digest::SHA1.hexdigest file5,
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 files 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 server.

```
{
  "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 server. Server receives this JSON and for each key, if it is a directory, server asks remote database for all the files under this directory using a HEAD request.After receiving all the files from the remote database it will calculate SHA of each file and a joint SHA of the directory. After computing the joint SHA, server 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, server will take the first key i.e. ï", it will look into its path and ask the client for all the files under Root Folder. After receiving all the files it will repeat the same procedure as client,for all the files with same path, it will calculate SHAs and joint SHAs and store them into a local flat file. After storing the SHAs, it will compare the SHAs from the client. If ïis present inside the list of SHA on flat file,then server does not match further and inform client that the files sent by client are already on the server. Else it looks into the children of ïi.e "gänd "händ if either or both of them doesn't match it will continue the process until it finds the files that are changed. After this step server send all the unknown

SHAs to client and client creates a zip file of all the unknown files. After the zip is ready it is sent to the server 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 server.
2. **Calculates SHA of application files:** In this operation the client goes through each file and calculates SHA and path of each file. It then goes through the path of all files and creates a joint SHA for files with same path.It takes $O(n^2)$ where n is the number of files.
3. **Calculates zip file:** After receiving list of unknown files, client puts all unknown file into a zip file. 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 server we need to look at the time taken to perform each operation. Following are the operations performed on server side:

1. **Finds the GUID corresponding to application name:** When server 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 server,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 server.
2. **Query remote database:** When the server receives list of SHAs from client, it queries remote database for all files.And for each file it 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 file SHAs, server makes comparison with SHA sent by client. This takes O(log n) time,where n is number of SHAs.
4. **Saves changed/new files on remote and local database:** When server receives changed/new files from client, it makes a POST request for each of them and saves them on remote and local database. This operation takes O(n) time, where n is number of changed/new files.

So, the total time taken by server is: $\mathbf{O(n)+O(n^2)+O(n) = O(n^2)}$, where n is number of files on client. Total space complexity of server is: **O(1)**,Since server 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 server has to make in order to get all the files from remote database and after that it calculates hash and path of each file. 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 file 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 database (with SHAs of all the files on the remote database) 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 database at server, we save all the HEAD request made by server to remote database. 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 database have to be queried for all the files 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 server 3.8 algorithms of combined approach that takes benefits of both the approaches.

Algorithm 3.7 Client

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

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

PUT /bits zipfile
```

Algorithm 3.8 Server

```
/match do
unknownhash[] ← SHA not known by
server
Server receives List of SHAs
for each sha ∈ List do
    queries local database
    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(zipfile)
for each file ∈ Folder do
    filehash ← hashoffile
    PUT filehash, filecontent 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 Syth's sequence diagram 3.2 because the server has a local database that stores all the SHAs. This algorithm works in following steps:

1. In the first PUT request client sends application name to the server and the server creates a new GUID for the client and sends the GUID with status 201. If the server already knows the application it sends the GUID with status 200.
2. In second step the client calculates SHA of all the files 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 server
    a) If it is the first time the application is uploaded to the server. It will save all the SHAs sent by client to its local database and return the same list to client as the unknown files.

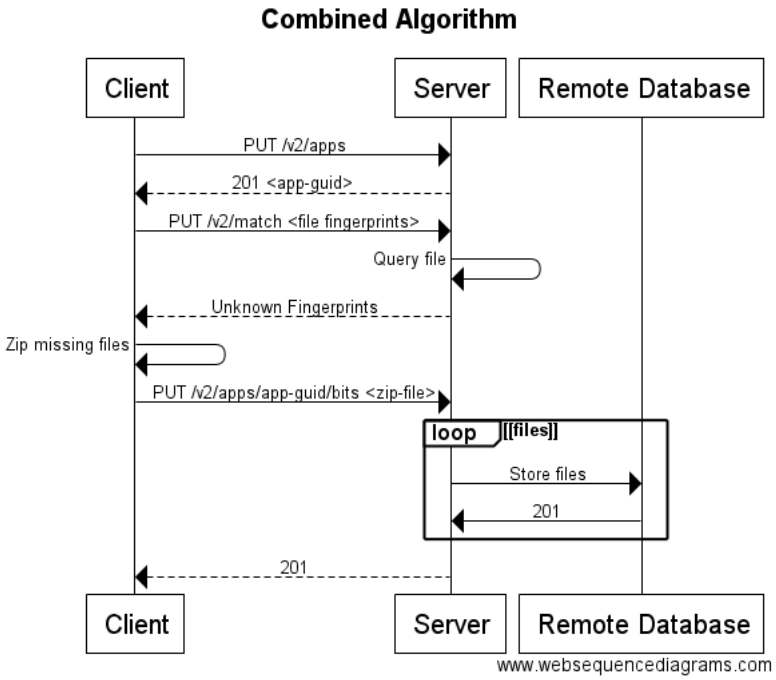


Abbildung 3.5.: Combined Algorithm

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

3. For all the file's SHAs that didn't match the SHAs in server database are put into a list and that list is sent as a list of Unknown file SHAs to client.
4. Client creates a zip of all the files and sends it to server.
5. Server unzips the file received from client and for each file it makes a POST request to remote database. Remote database replies with a status code of 201 for each file.
6. Finally, when all the files are stored on the remote database server 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 server.

2. **Calculates SHA of application files:** In this operation the client goes through each file and calculates SHA and path of each file. It then goes through the path of all files and creates a joint SHA for files with same path.It takes $O(n^2)$ where n is the number of files.
3. **Calculates zip file:** After receiving list of unknown files, client puts all unknown file into a zip file. 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 server we need to look at the time taken to perform each operation. Following are the operations performed on server side 3.8:

1. **Finds the GUID corresponding to application name:** When server 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 server,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 server.
2. **Query local database:** When the server receives list of SHAs from client, it queries local database for all files SHAs.This operation take O(log n) time, where n is number of SHAs sent by client.
3. **Saves changed/new files on remote and local database:** When server receives changed/new files from client, it makes a POST request for each of them and saves them on remote and local database. This operation takes O(n) time, where n is number of changed/new files.

So, the total time taken by server is: **O(n)+O(log n)+O(n) = O(n)**, where O(log n) is number of files on client. Total space complexity of server is: **O(nk)**,n is number of application files and k is number of applications on server Since server 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 files. This algorithm does not impose any rule of not storing files of size less than 64KB and hence, while sending the files only changed/new files are sent from client to server. Since this algorithm calculates directory hash and compares directory hash before comparing file 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 database approach of Sycth. But by keeping a local database at the server it increases the space complexity at server side to O(nk) where n is the number of files and directory in application directory and k is the number of applications on the server. This would increase with increase in applications on the server. Following we suggest an improved approach which would reduce the space complexity at the server to O(1). That is we do not need to have an additional database on server.

3.6. IMPROVED ALGORITHM

In the previous algorithm 3.5 we combined the two approaches of Sycth 3.2 and Marc 3.3 and saw that the results improved drastically. Th time complexity of resource matching reduced from O(n) to O(log n). But Space complexity at server side was still O(nk) where where n is the number of files and directory in application directory and k is the number of applications on the server. To reduce this space complexity we have to find a solution to the problem that would avoid having a database on the server. For this factor, we can make use of one of the assumptions that we made in the beginning 3.1.1 of this chapter i.e., the assumption of having unlimited storage at remote database. Given this assumption we can store the list of file and directory SHAs in a file with other metadata about the application on remote database. And whenever there is a match request from the client we can make a request for this file to the server and calculate the difference. Following is the improved algorithm for client and server.

Algorithm 3.9 Client

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

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

PUT /bits zipfile
```

Algorithm 3.10 Server

```
/match do
unknownhash[] ← SHA not known by
server
Server receives List of SHAs
Server requests remote database
file of metadata with hash[]
for each sha ∈ List do
    if hash[] does not include sha then
        unknownhash[] ← sha
    end if
end for
return unknownhash[]
end /match

/bits do
folder ← unzip(zipfile)
for each file ∈ Folder do
    filehash ← hashoffile
    PUT filehash, filecontent to remote
    database
end for
end /bits
```

3.6.1. ALGORITHM:

Figure 3.6 is the sequence diagram for the improve approach. This approach creates a metadata file which saves all the metadata required by resource matching approach. This algorithm works in following steps:

1. In the first PUT request client sends application name to the server and the server creates a new GUID for the client and sends the GUID with status 201. If the server already knows the application it sends the GUID with status 200.
2. In second step the client calculates SHA of all the files 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 server
    a) If it is the first time the application is uploaded to the server. It will return the same list to client as the unknown files.

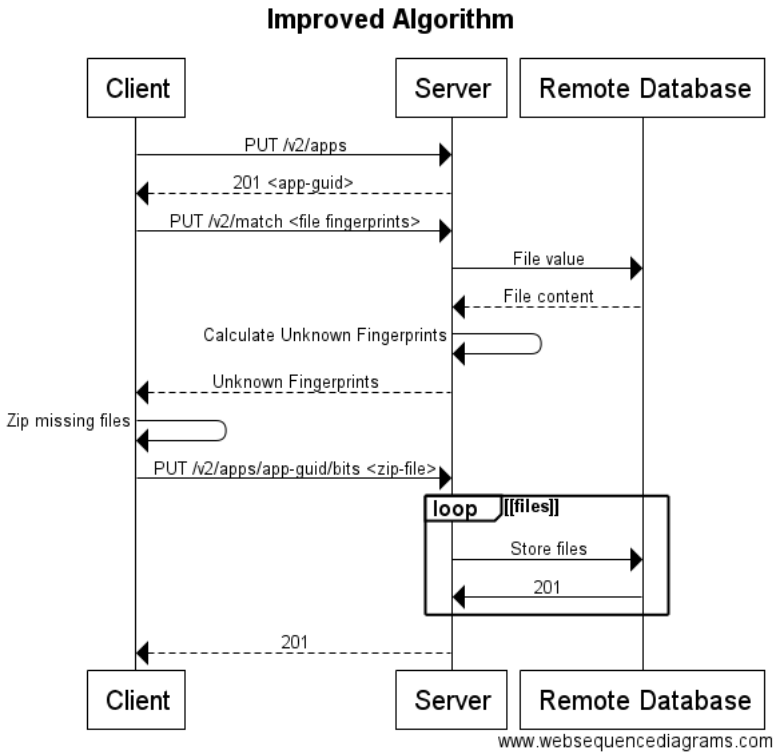


Abbildung 3.6.: Improved Algorithm

b) If it is not the first time the application is uploaded to server, it will query its remote database for the metadata file content. After receiving the metadata it compares the SHAs received from the client to the one received from server using the same tree based approach as discussed above 3.5.

3. For all the file's SHAs that didn't match the SHAs in server database are put into a list and that list is sent as a list of Unknown file SHAs to client.
4. Client creates a zip of all the files along with a metafile where it stores metadata for all application files and sends it to server.
5. Server unzips the file received from client and for each file it makes a POST request to remote database. Remote database replies with a status code of 201 for each file.
6. Finally, when all the files are stored on the remote database server sends a status code of 201 to client and this completes the entire process.

3.6.2. SPACE AND TIME COMPLEXITY:

Time complexity at client 3.9 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 server.
2. **Calculates SHA of application files:** In this operation the client goes through each file and calculates SHA and path of each file. It then goes through the path of all files and creates a joint SHA for files with same path.It takes $O(n^2)$ where n is the number of files.
3. **Calculates zip file:** After receiving list of unknown files, client puts all unknown file into a zip file. 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 server we need to look at the time taken to perform each operation. Following are the operations performed on server side 3.10:

1. **Finds the GUID corresponding to application name:** When server 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 server,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 server.
2. **Requests remote database:** When the server receives list of SHAs from client, it queries remote database for the content of metafile. This operation takes O(1)
3. **Calculates diff:** When the server receives contents of metafile from remote server, for all files SHAs it compares the value with the client SHA's.This operation take O(log n) time, where n is number of SHAs sent by client.
4. **Saves changed/new files on remote and local database:** When server receives changed/new files from client, it makes a POST request for each of them and saves them on remote and local database. This operation takes O(n) time, where n is number of changed/new files.

So, the total time taken by server is: **O(n)+O(1)+O(log n)+O(n) = O(n)**, where O(log n) is number of files on client. Total space complexity of server is: **O(1)**.

3.6.3. DISCUSSION:

In this algorithm we saw a lot of improvements in time and space complexity. The time complexity reduced from O(n) to O(log n) where n is the number of application files and space on the server reduced to O(1) as we do not maintain any local database. This algorithm does not impose any rule of not storing files of size less than 64KB and hence, while sending the files only changed/new files are sent from client to server. Since this algorithm calculates directory hash and compares directory hash before comparing file hash, chances of hash collision is

reduced. Since this algorithm saves metadata of the application files, this metadata can be used for deleting old or unwanted files from remote database.

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. $\mathrm{d}\Omega = \sin\vartheta \mathrm{d}\vartheta \mathrm{d}\varphi$. 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. $\mathrm{d}\Omega = \sin\vartheta \mathrm{d}\vartheta \mathrm{d}\varphi$. 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

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. $\mathrm{d}\Omega = \sin\vartheta \mathrm{d}\vartheta \mathrm{d}\varphi$. 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. $\mathrm{d}\Omega = \sin\vartheta \mathrm{d}\vartheta \mathrm{d}\varphi$. 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. $\mathrm{d}\Omega = \sin\vartheta \mathrm{d}\vartheta \mathrm{d}\varphi$. 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 Abstäze 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 Bibliopgrahie-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.

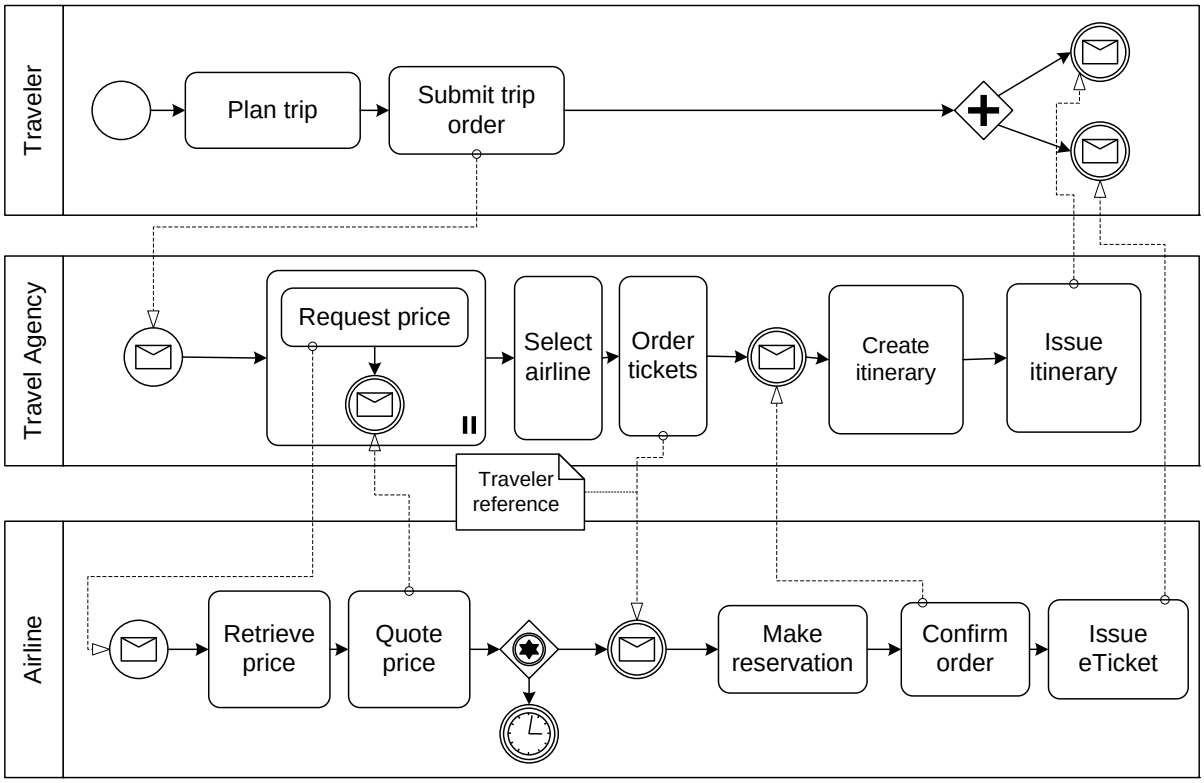


Abbildung A.1.: Beispiel-Choreographie

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-tipps.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 Beispielalgorithmus.

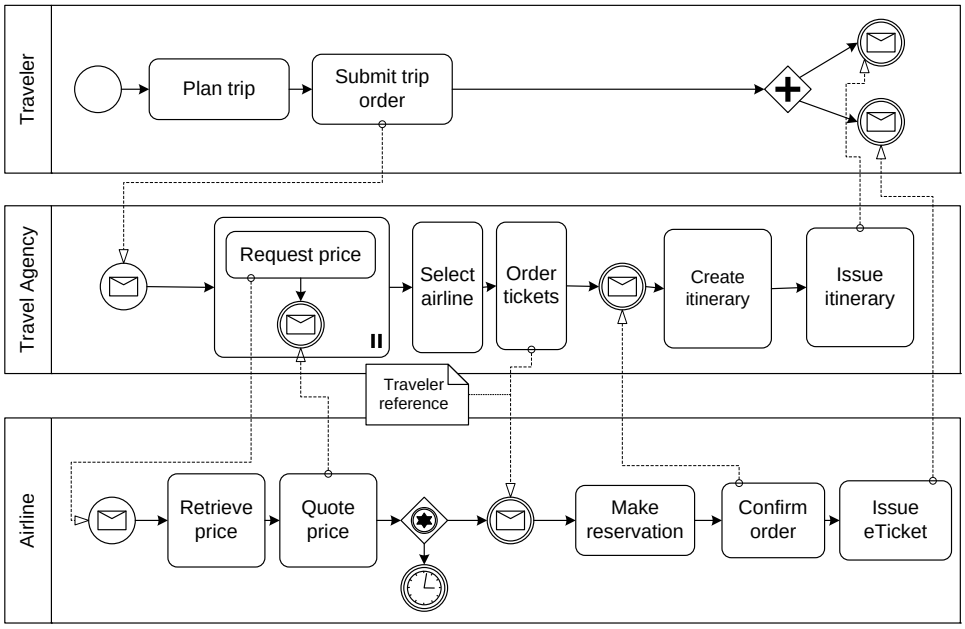
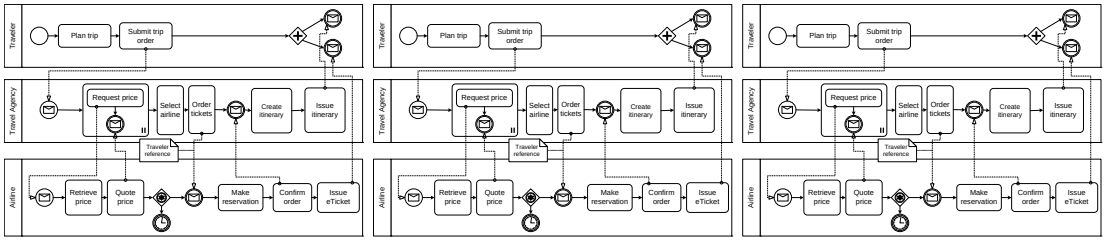


Abbildung A.2.: Die Beispiel-Choreographie. Nun etwas kleiner, damit `\textwidth` demonstriert wird. Und auch die Verwendung von alternativen Bildunterschriften für das Verzeichnis der Abbildungen. Letzteres ist allerdings nur Bedingt zu empfehlen, denn wer liest schon so viel Text unter einem Bild?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



(a) **(b)** **(c)** Subcaption if needed

Abbildung A.3.: Beispiel um 3 Abbildung nebeneinader 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/

	Parameter 1		Parameter 2		Parameter 3		Parameter 4	
Bedingungen	**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). Hinweiß: immer die selbe anzahl an Nachkommastellen angeben.

Algorithmus A.1 Sample algorithm

procedure SAMPLE(a,v_e)
  $\mathsf{parentHandled} \leftarrow (a = \mathsf{process}) \vee \mathsf{visited}(a'), (a', c, a) \in \mathsf{HR}$
    // $(a', c'a) \in \mathsf{HR}$ denotes that a' is the parent of a
  if $\mathsf{parentHandled} \wedge (\mathcal{L}_{in}(a) = \emptyset \vee \forall l \in \mathcal{L}_{in}(a) : \mathsf{visited}(l))$ **then**
    $\mathsf{visited}(a) \leftarrow \mathsf{true}$
    $$\mathsf{writes}_\circ(a, v_e) \leftarrow \begin{cases} \mathsf{joinLinks}(a, v_e) & |\mathcal{L}_{in}(a)| > 0 \\ \mathsf{writes}_\circ(p, v_e) & \exists p : (p, c, a) \in \mathsf{HR} \\ (\emptyset, \emptyset, \emptyset, false) & \text{otherwise} \end{cases}$$
    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 $a = \mathsf{process}$ **then** // Directly handle the contained activity
      HANDLEACTIVITY(a',v_e), $(a, \perp, a') \in \mathsf{HR}$
      $\mathsf{writes}_\bullet(a) \leftarrow \mathsf{writes}_\bullet(a')$
    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 werden[1]. 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 verwendet[1].

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 `pdflscape` zu demonstrieren.

[1] Diese Fußnote ist ein Beispiel.

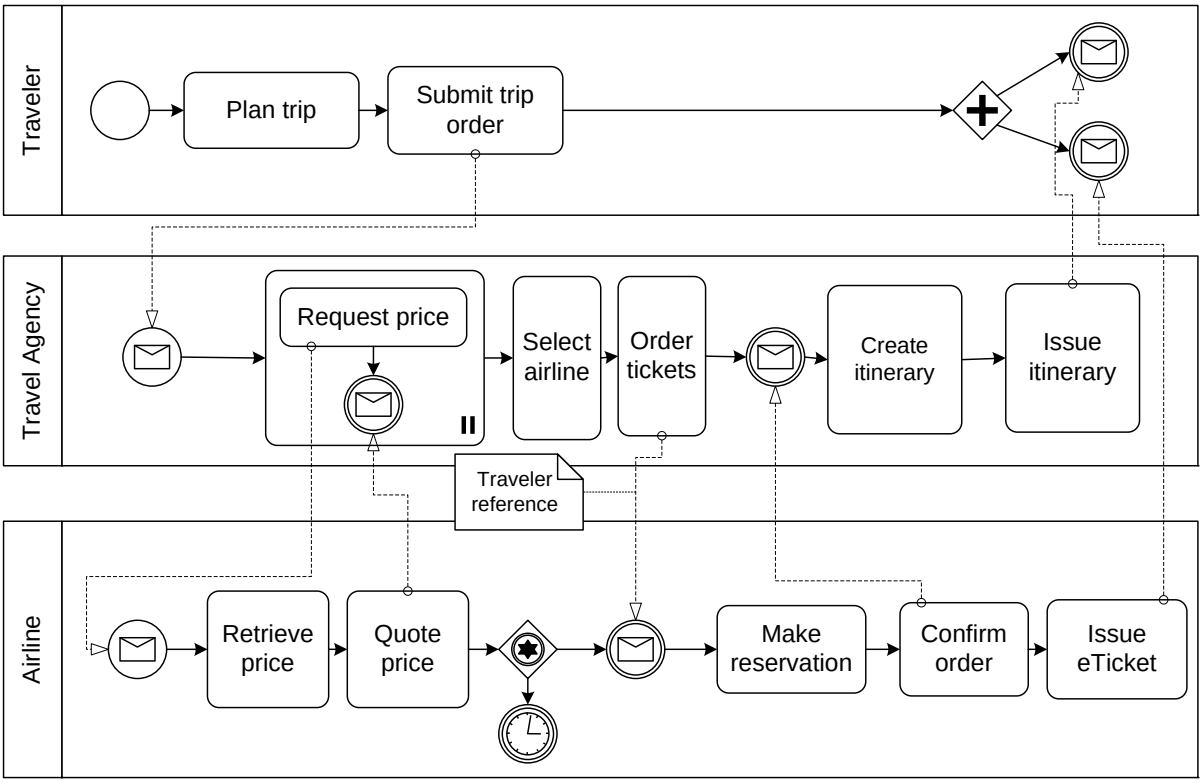


Abbildung A.4.: Beispiel-Choreographie I

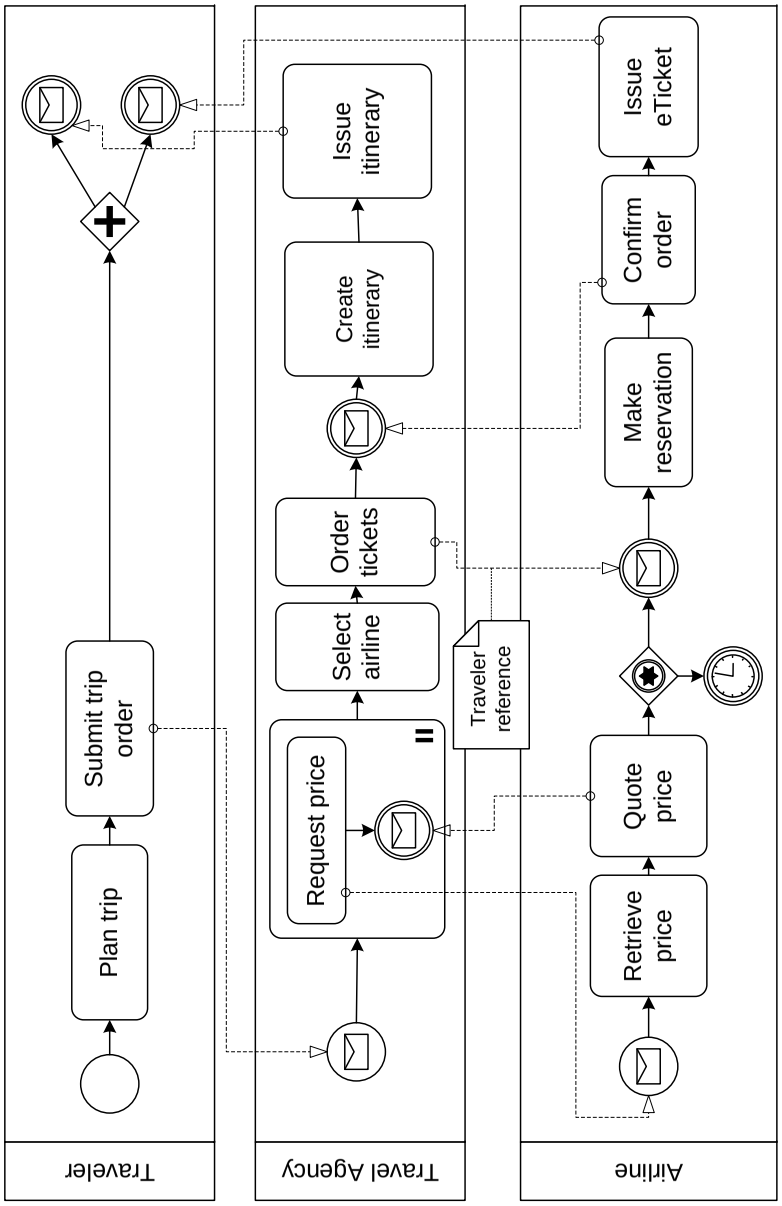


Abbildung A.5.: Beispiel-Choreographie II

A.14. Plots with pgfplots

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

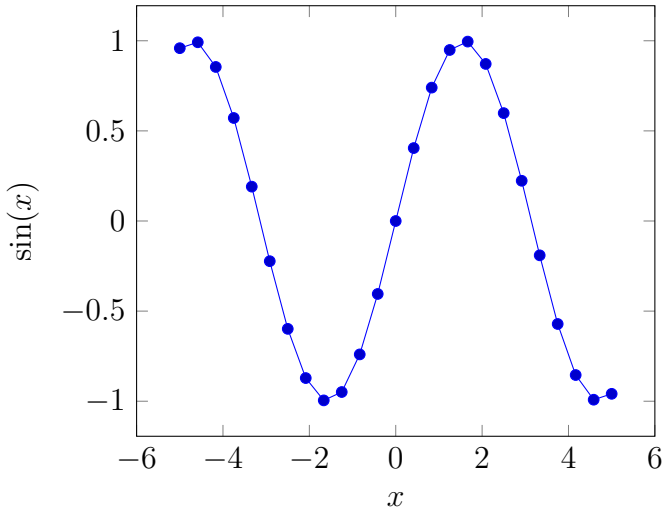


Abbildung A.6.: $\sin(x)$ mit pgfplots.

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.

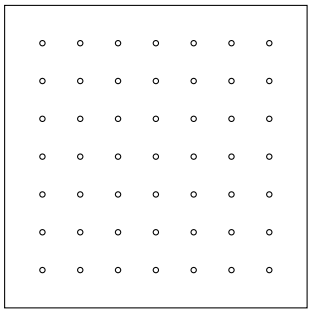


Abbildung A.7.: Eine tikz-Graphik.

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. 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).

[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 (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 (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