Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Nr. MCS-0017

Optimized Resource Matching for uploading bits in a PaaS Cloud

Niharika Mittal

Course of Study:	Computer Science
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Dr. rer. nat. Johannes Wettinger, Dipl.Ing. Steffen Uhlig
Commenced:	1. November 2016
Completed:	3. May 2017
CR-Classification:	C.2.4

Abstract

Cloud Computing is gaining momentum every day as most companies are adopting cloud for their business. Cloud computing provides different kinds of service models: Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service. Different organizations provide different solutions for each kind of cloud service model. Since the cloud services are used by most of the businesses, it needs to be very fast, accurate, easy to use, secure and fault tolerant. In this thesis, we improve the performance of a PaaS cloud called Bluemix, developed by IBM. Along with the performance, we improve the overall experience of the developers using a PaaS cloud for uploading their applications.

Bluemix is used by developers to create, store, manage and run their applications. Developers create their application and push it to Bluemix to run the latest version of the application. We research on the algorithm behind the push mechanism of applications. This algorithm is know as the "file synchronization" algorithm. File Synchronization in PaaS cloud has three main components: Client, Server and Blobstore. These three components are connected to each other via network. Therefore, network latency and bandwidth are major factors deciding the performance of an approach for file synchronization.

The goal of this thesis is to improve the overall user experience and performance of file synchronization in a PaaS cloud. To this end, we survey different solutions available for the file synchronization. One of the prominent examples is Rsync. We formally describe and evaluate the three suggested approaches for file synchronization in PaaS cloud in this thesis. We test the performance of the three approaches with different types of cache. We compare these approaches with the currently implemented approach.

We measure the performance of an approach by measuring the total time of file synchronization, total bandwidth usage, total amount of storage used and the speed of synchronization. After evaluating all the approaches including the existing one, on the former metrics, we have successfully reduced the total time of synchronization, total storage and total bandwidth usage by a significant amount.

1 Acknowledgement

At the outset, I would like to acknowledge with deep gratitude, the guidance and consistent encouragement of my Master thesis supervisor at IBM, Steffen Uhlig. His insightful discussion, valuable comments and handmade diagrams had been immense help to me in my understanding of sophisticated concepts and in the growth of my knowledge. I thank him for his continued support and patience throughout my thesis work. He was always available for me whenever I needed his assistance. I could have not imagined having a better advisor and mentor.

I would also like to thank my master thesis supervisor at University of Stuttgart, Johannes Wettinger, for taking the thesis under his supervision and providing me with his valuable feedback on my work. Without him I would not have been able to work on this research project. I thank him for his continuous support during my master study at University of Stuttgart. I am very grateful to have a supervisor like him for my entire Master study.

I would also like to thank the connoisseur who helped me understand the technologies better, Marc Schunk. His detailed diagrams, discussions and comments had been of great help to me in my understanding of the technologies used. I would like to thank him and Matthew Sykes for the ideas they provided that helped me build some optimal solutions. I thank them for the great ideas.

I would like to thank the experts who gave valuable feedback on every iteration of my work: Peter Goetz, Simon Moser and Bernhard Schmid. Their valuable feedback helped me to improve my work and presentation skills.

I would like to express my sincere gratitude to Simon Moser for providing me the opportunity to work on this thesis with his team. Furthermore, I would like to thank Professor Frank Leymann for providing me the opportunity to work on this thesis at his institute.

Finally, I thank my parents and friends for providing me with unfailing support and continuous motivation throughout my years of study and through the process of researching and writing this thesis.

Contents

1 Acknowledgement 5

2 Introduction 15

3 Fundamentals 17
- 3.1 Basic of File Synchronization 17
- 3.2 File Synchronization 20
- 3.3 Hash Tree 22
- 3.4 Factors affecting the performance of a File Synchronization Algorithm . . 23
- 3.5 Cloud Computing 24
- 3.6 Cloud Foundry 25
- 3.7 Bits Service 29
- 3.8 Summary 31

4 Related Work 33
- 4.1 Assumption 33
- 4.2 Rsync 34
- 4.3 Zsync 36
- 4.4 BURP 38
- 4.5 Map Based 39
- 4.6 Bloom Filter 41
- 4.7 TAPER 42
- 4.8 Stack Sync 46
- 4.9 Set Reconciliation 48
- 4.10 Git Approach 49

5 Suggested Approaches 51
- 5.1 The Problem 51
- 5.2 Current Approach 53
- 5.3 Local Database Approach 60
- 5.4 Tree Based Approach 66
- 5.5 Combined Approach 73
- 5.6 Comparison and Summary 79

6 Implementation and Evaluation 81
6.1 Performance 81
6.2 Implementation 85
6.3 Test Setup 85
6.4 First Push 88
6.5 Subsequent Push 91
6.6 Number of HTTP Requests 95
6.7 Pushing without any optimization 98

7 Conclusion and Future Work 101
7.1 Conclusion 101
7.2 Performance 102
7.3 Concurrency Issues 103
7.4 Incomplete Push 104
7.5 Disaster Recovery 104
7.6 Further Opportunities 105

Bibliography 107

List of Figures

3.1 Example of a Java Application Package 18
3.2 One way file synchronization 20
3.3 Two way file synchronization 21
3.4 Hash Tree [wik] 22
3.5 Types of Service Models [Kat13] 25
3.6 Architecture of Cloud Foundry 27
3.7 Components of Cloud Foundry 28
3.8 State of Art 30
3.9 Architecture with Bits Service 31

4.1 Rsync 35
4.2 Map Construction 40
4.3 Bloom Filter 41
4.4 Phases of TAPER 43
4.5 Hierarchical tree structure 44
4.6 Content Defined Composition 44
4.7 Level Of Granularity 45
4.8 Architecture of Stack Sync 47
4.9 Interaction of different components of Stack Sync 47
4.10 Git Workflow 50

5.1 Current Approach 55
5.2 Current Approach Pie Chart 59
5.3 Local Database Approach 62
5.4 Local Database Approach Pie Chart 65
5.5 Tree Based Approach 68
5.6 Directory Structure [Ash] 70
5.7 Tree Based Approach Pie Chart 73
5.8 Combined Approach 75
5.9 Combined Approach Pie Chart 78

6.1 Component Diagram 86
6.2 Class Diagram of Services on the Server 87
6.3 Test Setup 88

6.4 First Push 91
6.5 Subsequent Push 94
6.6 Assemble Phase 98
6.7 Graph for HTTP Requests 99

7.1 Components 105

List of Tables

4.1 Evaluation Result of Rsync and Zsync 38

5.1 Differences between approaches 80

6.1 Type of Applications 84
6.2 Basic information about application files 89
6.3 First Push of application files 90
6.4 Second Push of application files 93
6.5 Match Phase at Bits-Service 95
6.6 Total Time to assemble package 96
6.7 Total Number of HTTP Request 97
6.8 Comparison of Tree Based Approach with No-Sync 100

List of Algorithms and Approaches

4.1 Rsync Client 34
4.2 Rsync Server 34

5.1 Current Approach CF Client 54
5.2 Current Approach Bits-Service 54
5.3 Local Database Approach CF Client 61
5.4 Local Database Approach Bits-Service 61
5.5 Tree Based Approach CF Client 67
5.6 Tree Based Approach Bits-Service 67
5.7 Combined Approach CF Client 74
5.8 Combined Approach Bits-Service 74

2 Introduction

A Platform as a service(PaaS) is a cloud service that allows the developers to create their applications without worrying about the complexity of the underlying infrastructure. It takes care of all the infrastructural operations involved in running, maintaining and developing applications. We are working on a PaaS cloud called Bluemix [Ang14], developed by IBM [IBMb] which is based on Cloud Foundry [Fou]. Bluemix uses Softlayer[Sof] as its underlying infrastructure. Bluemix supports many programming languages, services and integrated DevOps tooling which helps the developers to create, maintain, run and store their applications in the cloud.

Bluemix is based on client-server interaction model, where a client develops applications and deploy them on the cloud. Whenever a client is creating a new application or it is changing its application, it uploads the application to the Bluemix to run the updated version of application. When an application is uploaded to the cloud, a file synchronization algorithm starts in the background that detects all the changed files.

In this thesis, we are evaluating different solutions that would improve the performance of file synchronization in cloud. To this end, we aim to reduce the total time of synchronization and to use the resources like bandwidth and storage optimally.

File synchronization is a process that includes two components, the source and the target. The same data is maintained on the source and the target. Suppose there is a file A on the source and the target. The source updates its file from A to A' but the target still has A. File synchronization algorithm ensures that all the changes on A are transferred to target, so that there exists an exact copy of the source on the target.

In our case, source is the client and target is the server. Client has an application directory where it stores and updates all the application files and directories. Server has a local database that saves the basic metadata like name, GUID, size of application, number of files in application etc about the applications coming from client. We will use additional metadata in some of our approaches. For storing all the application files and application package we have used Softlayer as the underlying infrastructure.

In our scenario, file synchronization is not like a normal file synchronization where either the source and target are on the same machine or they both are connected to each

other by network. Along with source and target we have a blobstore(for file storage) that stores the application data and this blobstore is also connected to server via a network.

To perform file synchronization in a PaaS cloud (where the client, the server and the blobstore are connected to each other via a network) we consider many factors like limited bandwidth between client and server, huge number of extremely small application files, limited computation power at server etc. We achieve following enhancements in this thesis : reduce the number of network requests over the network(as each request has a cost over the network), efficient usage of the Softlayer storage and reduced total time of file synchronization.

We have implemented three different approaches for overcoming the challenges and for optimization of factors such as bandwidth and storage. We have evaluated all the three approaches against the current file synchronization algorithm. All of them are performing better than the existing approach and they are using all the resources like storage at blobstore and computation at server more efficiently. One of the suggested algorithms is performing better than the rest. It has reduced the storage at the blobstore and reduced the total time to compare and calculate the delta between application files at client and at server. It has also reduced the number of requests over the network and total time to assemble the package. We will discuss about these approaches in Chapter 5.

Structure Of Thesis

The structure of the remaining thesis is organized as follows: In Chapter 3, we present the basic fundamentals related to the topic of the thesis. We discuss the basics of file synchronization, cloud computing and PaaS. We present detailed explanation of the concepts of Cloud Foundry and Bluemix.

In Chapter 4 describes the factor affecting file synchronization, we present research work performed in the area related to the topic of this thesis and describe the existing solutions like Rsync.

In Chapter 5 we present all four approaches (including the existing one) that we implemented. Along with the details of approaches we present sequence diagrams for each approach.

In Chapter 6 we present the implementation details, test setup and result of comparison of all the four approaches and description of our test environment.

Chapter 7 completes the thesis with a conclusion and potential for future work.

3 Fundamentals

In this Chapter, we discuss basics that will help to understand the content of this thesis. We will start with the basics of file synchronization, terms and algorithms used in it. We will explain in detail, the meaning of specific terms used in this thesis. In this chapter, we will also describe the terms that can be used interchangeably in this thesis. After describing of file synchronization, we will discuss the basics of cloud computing and its service models. We will explain the reason behind the research of file synchronization and its application to the PaaS cloud. Further, we explain the components involved in the PaaS cloud and the service which handles the bits.

We will start with the discussion of the basic terms used in the process of file synchronization explained in Section 3.2. There are some common terms that are mostly used while discussing file synchronization. Some of them are listed below. This section also explains the common algorithms used during file synchronization.

3.1 Basic of File Synchronization

3.1.1 Client

A client is a computer hardware that can access server locally or over the network depending on location of the server. In our case, client is any system that is running Cloud Foundry and pushing applications to the Cloud Foundry. We use the term CF client in our approaches instead of client.

3.1.2 Server

Server is a computer system that provides services to the clients. In our case, it stores the metadata of application files and stages the applications. We are building approaches for the Bits-Service[Ste16b] that is hosted on a remote server.

3.1.3 File or Blob

A file or a blob is single entity that has to be synchronized. It can contain some text or some application data. In our case we are using application files eg. configuration files, files containing source codes in different programming languages (Java, Python etc), database dumps etc. We will use the term file and blobs interchangeably.

3.1.4 Application Package

Application package is the main directory containing all the files or blobs linked to an application. Figure 3.1 represents an example of Java application package. The top most node of the figure represents the application directory. The next level represents the folders inside the application directory and the leaf nodes represents the application files eg. class and jar files. It contains all the code files, jars, configuration files, gem or

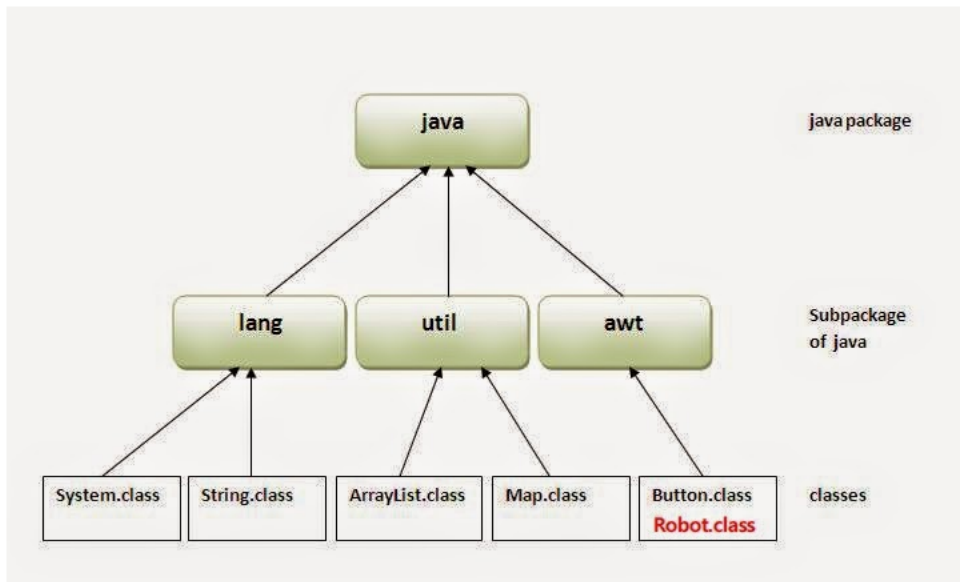


Figure 3.1: Example of a Java Application Package
[Kar15]

package.json etc. During file synchronization we ensure that the application package is consistent between client and server.

3.1.5 Block or Byte

For calculation of the changes between the files at client and server, file can be split into blocks of fixed or variable size or files can be read byte by byte. If file is divided into blocks, the hash is calculated for each block. If the file is read as bytes then hash is generated after reading the entire data of the file. We would be reading the entire file in all our approaches.

3.1.6 Delta or Difference

Delta or difference is the difference between the application files stored at the client and the server. Client changes the application files and sends them to the server. Server calculates the difference between the files that were already stored on it and the files sent by the client.

3.1.7 Hash Function

The source (that can be a client or server, but in our case it's the client) and the target (in our case it's the server) calculates fingerprints of all the files or chunk of files depending on the granularity of algorithm. Hash calculation can be done by using different algorithms like secure hash algorithm [SHA], message digest algorithm [Riv] etc. If chunk wise hash is calculated, a rolling checksum can also be calculated as explained in Section 3.1.8. The calculated fingerprints can be stored in a list or a tree like fashion as explained in Section 3.3. Most important property of hash function is that it generates unique hash for each input.

3.1.8 Rabin Karp Fingerprinting

As explained in [Mil] it calculates the hash of a file in a rolling fashion. It creates hash of fixed size block. It moves the block by one byte and calculates the hash again for a fixed size block. In this manner, it calculates the hash of entire file. This approach is called rolling checksum. It is used by many algorithms like Rsync, TAPER explained in Chapter 4

3.2 File Synchronization

File Synchronization is the process which involves two components called client and server. There can be any number of clients or servers involved in the process. The goal of file synchronization is to update the data in all the locations(client and server). Suppose there is a file A at client and a similar file A* is at the server. File synchronization is a process where both the client and server gets the change and in the end, have the same file. File synchronization can be of two types. One way and Two way.

1. **One Way:** As explained in Figure 3.2,in one way file synchronization, either the client or the server gets synchronized with the changes sent by the other one. Either always the client updates the server or server always updates the client. In our case, we always update the server with all the changes done at the client

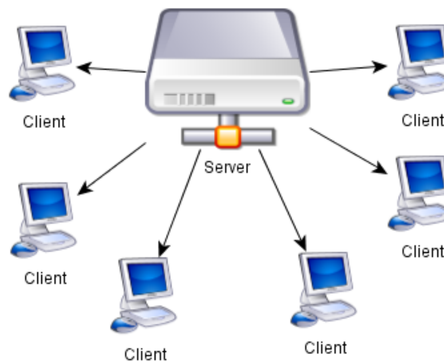


Figure 3.2: One way file synchronization
Maanka Maanka [Maa15]

2. **Two Way:** As explained in figure 3.3, in Two way file synchronization the files are synchronized both ways. Client updates the server and simultaneously server updates the client.

In this thesis we only consider one way file synchronization i.e. from client to server. We are using application packages as our data to be synchronized. We need a solution that could update the PaaS cloud every time client makes a change to the application files. The solution must be very fast and use all the resources like network and storage efficiently. There are three steps that we will always use for file synchronization. The first step is to find the fingerprint, second step is to match the fingerprints and the third step is to assemble the application package with changed files.

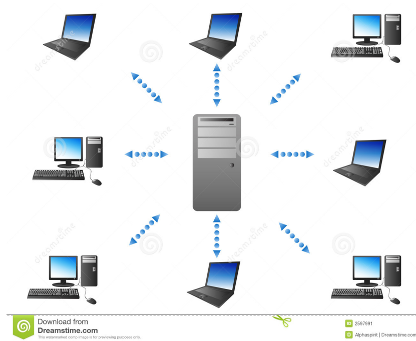


Figure 3.3: Two way file synchronization
[SUR]

3.2.1 Steps

Calculate Hash

The source (that can be a client or server, but in our case it's the client) and the target (in our case it's the server) calculates hash of all the files or chunk of files depending on the granularity of algorithm. Hash calculation can be done by using different algorithms like secure hash algorithm[SHA], message digest algorithm[Riv] etc. If chunk wise hash is calculated, a rolling checksum can also be calculated as explained in Section 3.1.8. Calculated hashes can be stored in a list or a tree like fashion as explained in Section3.3.

Match

The target calculates the difference after receiving hash from the source. The comparison could be a simple linear search for each hash received if it is sent as a list or it could be a binary tree search if hash tree based approach is used. The time taken would be much less in the later case. In former case the complexity would be O(n) whereas it would be O(log n) in later. The source or target can have a local database to store hash and metadata of application files.

Assemble Application Package

After calculation of the changed files at source or target, the entire application package is reconstructed to save the changes to the server. The application package could be saved on the server or on a remote blob store attached to server. In our case, we save it on a remote blob store attached to server. The files can be assembled depending on

way they are saved. If they are saved in a zip file then the zip file has to be downloaded and appended with the changes. If the files are individually stored then they have to be assembled by downloading each file individually. We will see both the implementations in Chapter 5

3.3 Hash Tree

Hash tree is a variant of Merkle Tree [Ral79] which was discovered by Ralph Merkle in year 1979. Merkle tree is a tree structure that assumes only two child nodes of a node. But a hash tree can have more than 2 children nodes to a node. Both, Merkle tree and hash tree have leaf nodes as the hash of data and the parent of each node is the combined value of the hashes of their child nodes. Figure 3.4 represents a hash tree. The yellow blocks in the figure represents the data. White nodes just above the yellow nodes are hash of the data and leaf nodes of the hash tree. The nodes with label "Hash 0" and "Hash 1" are combined hash of "Hash0-0", "Hash0-1" and "Hash1-0", "Hash1-1" respectively. Top hash is the hash of combine hashes of "Hash 0" and "Hash 1".

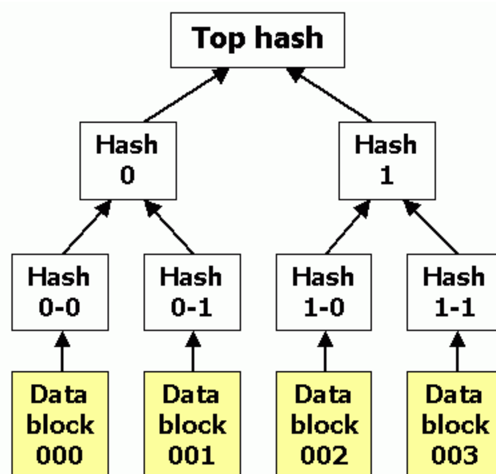


Figure 3.4: Hash Tree [wik]
[wik]

3.3.1 Advantages Of Hash Tree

1. Since each node is a hash of it's child node. Any fast and cheap hashing algorithm can be used to hash the data. Chances of hash collision reduces significantly in

this case. Since hash is now a function of random characters rather than the fixed set of characters.

2. Time complexity of the search operation is O(log n).
3. It is an easy way to store and retrieve metadata.

3.4 Factors affecting the performance of a File Synchronization Algorithm

1. **Computation of difference on Client side or server side**: This factor determines whether the difference or delta (as described in section 3.1.6 of Chapter 3) should be calculated at client side or at the server side. It is an important factor to be considered as the computation power present at server is always limited, while the computation power at client is unlimited.
2. **Byte or block** : This factor determines whether the difference between two files must be calculated block wise or byte wise. As explained in Section 3.1.5, the application file can be divided into blocks of fixed or variable size or it can be read byte wise. As described in [G P14a] file is chunked into variable chunk sizes, it becomes easier to calculate the difference without recalculating the chunks that did not change. Changing a single character can lead to change of all the chunks in fixed size chunking. Both of these chunking methods have different effect on performance of algorithm. We will discuss this in detail in Chapter 4.
3. **Differencing algorithm**: If an algorithm is used to find difference between two files on same machine, it is done using differencing algorithm. The algorithm that is run to find the difference between two files uses different string matching techniques. Those String matching techniques differ from each other in time complexities, speed and usage of other resources. This factor is one of the factors deciding the overall performance. But we do not consider this factor as our files are separated by network.
4. **Compression** : Different compression algorithms are used to compress the changes between two files. These compression algorithms affects the performance of the entire file synchronization directly. There are different types of algorithms discussed in last section of this Chapter.
5. **Integrity of messages over the network**: Since server and the client are separated by network, a lot of messages are sent over the network. Integrity of those messages has to be ensured.

6. **Use of Hash functions**: There are many hashing algorithms available to create a hash of the data. Prominent examples are SHA, MD4 . These all algorithms differ from each other by the strength of hash, speed of creating hash, cost of creating hash etc.

3.5 Cloud Computing

Cloud computing, enables the users to use computation resources as a service over the Internet. It is an Internet-based practice where remote servers are used for storage, computation and other on-demand services. Some main advantages of cloud computing are pay per use and flexibility. The user pays for only the amount of time it has used the service and not for the entire duration when the server was running. Usage can be scaled up and down as per the requirement and this can save investments in maintaining local infrastructure. Cloud Computing is divided into three service models-IaaS, SaaS, PaaS described in figure 3.5:

3.5.1 Models

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 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.

In this thesis, we are looking for a solution to reduce the total time to upload files to a PaaS cloud. There are many PaaS solutions available. Some prominent examples are listed below:

1. IBM Bluemix: Developed by IBM and based on Cloud Foundry. [IBMa]

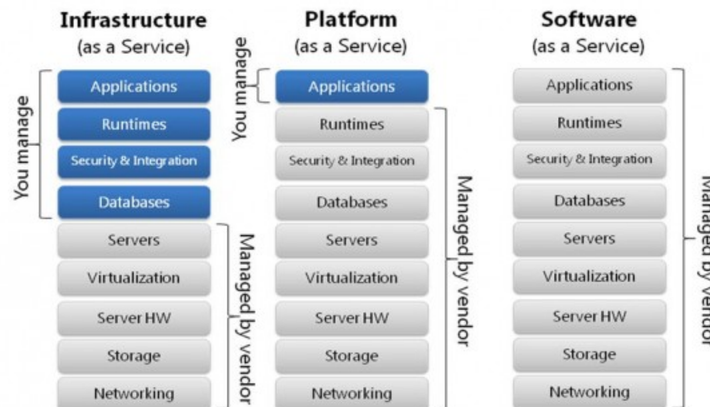


Figure 3.5: Types of Service Models [Kat13]

2. Google App Engine: Developed by Google. [Goo]
3. Amazon AWS: Developed by Amazon. [Ama]
4. Windows Azure Cloud: Developed by Microsoft. [Azu]
5. Heroku: Developed by Heroku. Heroku [Her]

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

3.6 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 as described in[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 different custom frameworks.
3. Application Services: Cloud Foundry offers support for MySQL, MongoDB, PostgreSQL, Redis, RabbitMQ and custom services.

3.6.1 Architecture

This section deals with the architectural functioning of Cloud Foundry. In the diagram 3.6, we can see the architecture of Cloud Foundry on a physical machine. The outer box represents a physical machine with a host operating system and processes like terminal, virtual box etc. The big 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 Release. For this, the 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, as described in[Clo17a], we first upload the stem cells. Stem cells are versioned images of minimum operating system skeleton, wrapped with IaaS specific packaging. We create a Cloud Foundry release 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. The status of virtual machines should be "running". In Figure 3.6 , thirteen red blocks represents the virtual machines started by Bosh.

3.6.2 Components

As described in [Clo17b], Cloud Foundry has self-service application execution engine, an automation engine for application deployment and life cycle 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. Diagram 3.7, shows the major components of Cloud Foundry:

The Figure 3.7 depicts the working of each component of Cloud Foundry as described in [Clo17b]. The router routes all the traffic that is coming in to the 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

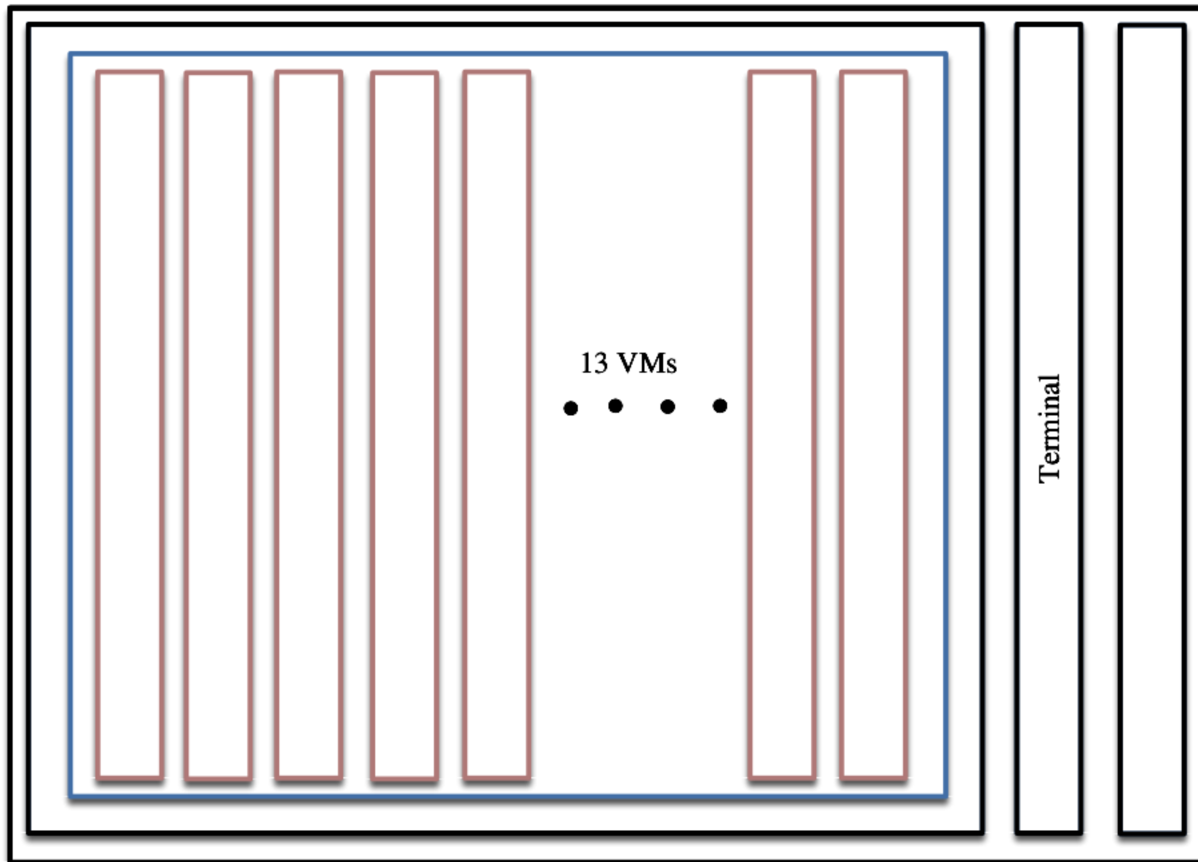


Figure 3.6: Architecture of Cloud Foundry

the applications and updates it's routing table accordingly. The next component is Authentication server OAuth2. It works together with login server to provide Identity Management. After authentication comes, the next part that manages application life cycle. This part constitute Bits Service(Cloud controller), nsync, Diego brain and cell reps. When an application is pushed into the Cloud Foundry, it is basically targeted to Cloud Controller which directs the Diego brain to stage and run the application. Diego executes the droplet and perform application life cycle tasks. Nsync, Cell reps helps in making the application available every time.

When a client scales the application nsync and cell reps work together along a chain to keep applications running. The next component deals with application storage and execution. An application is stored in a blob store. A blob store is a repository that stores large binary files which include application code packages, build packs and droplets. It can be configured as an internal file server or an external like S3. We will discuss blob store in detail in further sections. The Diego cell manages the start, stop and status of the applications. Applications in Cloud Foundry basically depends upon services.

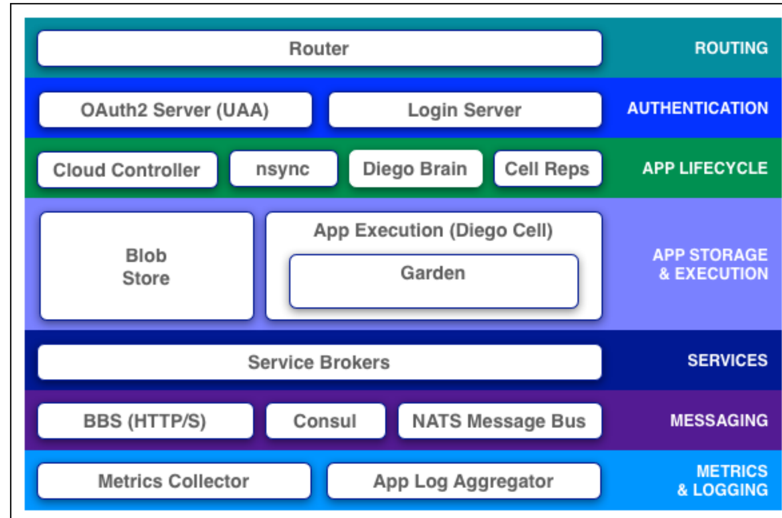


Figure 3.7: Components of Cloud Foundry
[Clo17b]

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

3.6.3 Load Balancing with Cloud Foundry

As described in [Clo17c], Cloud Foundry performs load balancing at following three levels:

1. **BOSH**: As described in [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**: As described in [Clo17d], provides REST end points to the requests coming from client. It maintains a database for the users and their applications. It stores user space, region, organization etc. It runs the applications and other processes on the cloud's VMs, balancing demand and

managing application life cycle [Clo17c]. An extraction to cloud controller is being developed by IBM and Pivotal called **Bits Service** [**bit'sService**]. It encapsulates all the "bit's operations" into separable scalable services. We will discuss about Bits service in detail in later section.

3. **The router**: It routes incoming traffic from the world to the VMs that are running the apps that the traffic demands, usually working with a customer-provided load balancer.

3.6.4 Blob Store

As described in[Clo17d]. the Bits Service manages a blob store for the following reasons:

1. **Application Cache**: It is also called as app cache. Files are stored in application cache with a unique SHA as filenames. Hash is a unique string of bytes that represents a files content. So in application cache files are addressed by their content. Files that do not change do not need to be re-uploaded and can be reused. For example, a Java Servlet file is always used by a java application. This file is uploaded once for all the clients.
2. **Application Package**: It consists of a zipped file that represents an application. Diego reads the application from this cache along with the droplet, buildpack etc. Blob store saves last three application package of an application and the most recent version is used by diego. If there is an error in the most recent version it is deleted and a version prior to it is used by diego.
3. **Droplet** : It is the result of taking application package staging is using buildpack and preparing it to run.

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

3.7 Bits Service

Bits service [Mara] is a micro service developed by IBM and Pivotal to handle only bits operation of the applications uploaded to Cloud Foundry. Bits can be application packages, droplets or buildpacks. The service encapsulates all the bits related functions. It will be used for carrying out bits related operations like: upload of application bits

to cloud (upload will happen when there is a new application or there is a change in application running on cloud), downloads of application bits from cloud to Cloud Foundry client (download will happen when the changes need to be merged with older version of application), for resource matching and for handling application packages, droplets, buildpacks. Bits can be stored locally on the disk or remotely on S3 or Openstack swift or Softlayer. It is used while resource matching described in 3.7.1.

According to the state of art (in 3.8), Cloud Controller along with some other components, performs all the functions of saving, running and maintaining the applications. Cloud controller is a huge code base and if something goes wrong with it, it becomes really difficult to handle the error. Bits service is required to free the cloud controller from bits handling operation, so that it can be scaled easily and independently. Bits-Service extracts the functionality of bits handling from Cloud Controller. Since this service only deals with application bits, it is easy to maintain the service and to extend it further. Bits Service lead to simpler operations. It is a cleaner API that only deals with bits. Since it extracts all the bits operations, cloud controller will become more responsive. File upload, download and push will become faster.

Figure 3.8: State of Art

In the current design of Bits Service(in 3.9), it has a local SQL database, where it stores all the metadata related to all the applications (eg. Application Name, GUID, Application size, number of files etc). It is connected to Softlayer (i.e a cloud Infrastructure as a Service (IaaS) provided by IBM) to store the application files, application packages. It is directly connected to Cloud Foundry clients with CLI. Clients can directly push their applications to Bits service in order to save them and run them on Cloud Foundry.

3.7.1 Application Deployment on Cloud Foundry

Cloud Foundry comes with a command line interface (cf CLI). Cloud Foundry CLI can be used to change the language of terminal output, to log into Cloud Foundry, to push applications to Cloud Foundry etc.

To deploy an application first time or to sync the changes to Cloud Foundry, "cf push" is used. "cf push" is responsible for the resource matching in Cloud Foundry. "cf push"

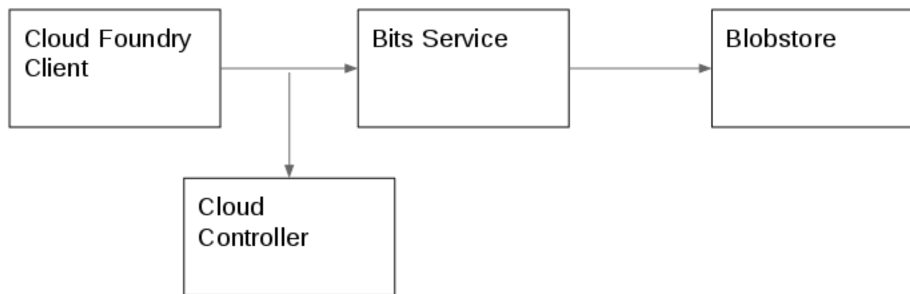


Figure 3.9: Architecture with Bits Service

takes the name of the application and domain. Application name and domain are sent to the Bits Service. Bits service maintains a local SQL database where it stores application name, it's domain and other metadata related to application. If the application is pushed for the first time, a GUID is generated by the Bits service for that application. GUID is a unique ID given by Bits service to each application, that is used later for storing the applications on blobstore. The SHA of files are then uploaded by Cloud Foundry to the Bits service. Bits service looks into application cache 1 for all the SHA sent by Cloud Foundry. It then returns back a list of SHAs that were found and Cloud Foundry uploads the bits that were not already present on application cache to Blobstore 3.6.4. In the end, the server sends a 201 status code to the CF client.

3.8 Summary

In this Chapter, we discussed the basic term and technologies used during file synchronization in a PaaS cloud. We would be using the technical terms, described here in the description of (suggested) approaches in Chapter 5. For example, we will use the term Bits-Service/server hosting Bits-Service in place of server. As described in Section 3.7 we have developed this service just for taking care of the bits in the cloud. Another example is the remote file server which we call as a blobstore. In our scenario, Softlayer is the blob store and it has five different caches for storing application package(zip of entire application) Application cache, build cache, droplet cache and package cache. We will use two of them: application cache for storing files greater than 64KB and Package cache for storing the entire application package. There are many other terms that are used interchangeably with the terms specific to this project, all such terms can be read here in this Chapter.

In the following Chapter, we will present the work already done in the field of file synchronization. We will discuss the existing approaches for file synchronization. In the subsequent Chapters, we will describe the solutions we implemented for the scenario discussed in Section 3.7 followed by the presentation of evaluation results and further opportunities.

4 Related Work

This chapter discusses the existing work that has been done in the field of file synchronization. File synchronization is a process where the two components i.e. the client and the server exchange information in order to maintain same information at both ends. In the following section we present the previous work that has been done in the field of file synchronization. Before we discuss the existing work, we lay down some factors (in next section) based on which we studied the existing work. The factors listed below directly affect the performance of any file synchronization and this can be verified shortly in following sections. We are looking for a solution for file synchronization that has all the three components (client, server and blobstore) communicating with each other via network and this problem can be solved in many ways depending on the factors discussed below. In the next chapter, we discuss our proposed approaches based on the approaches present in this chapter.

4.1 Assumption

Only the research that was done or continued after year 2000 is studied for this thesis. Only exception to this is Rsync Algorithm as it is considered the standard algorithm for file synchronization. There are over 90,000 researches available in this field and it is not possible to read all the existing work in the small duration of master thesis. The reason we chose to study the research after year 2000 is that we are looking for a synchronization approach in cloud and the term "Cloud Computing" started gaining popularity only after year 2000. Another reason is that all the significant references of the research we did, are after the year 2000.

4.2 Rsync

4.2.1 Algorithm Description:

Algorithm 4.1 Client

```
for each blob ∈ Application do
    Splitblob into non overlapping
    blocks of size S
    Array ← blocks
end for
for each block ∈ Array do
    Weak_Array ← calculate 128 bit
    rolling checksum
    Strong_Array ← calculate 32 bit
    MD4 checksum
end for

PUT/matchWeak_Array, Strong_Array
```

Algorithm 4.2 Server

```
/match do
for each block ∈ Application do
    find weakChecksum(block)
     in WeakArray
    find strongChecksum(block)
    in StrongArray
    if blocknotfound then
        UnknowBlock[] ← block
    end if
end for
Sends instructions for construction
 of files at client
Each is either a reference to known
 block or literal data
end /match
```

Rsync stands for Remote Synchronization, the client and the server are connected to each other by network or are connected locally. It is a block-based approach. Its main purpose is to produce mirror of content of applications. Also it runs in the back end of various backup systems. An assumption made by Rsync algorithm is that the **communication link between the client and server is very slow**. It falls under the category of algorithms that require only single round trip for computation of the match and computes it by calculating checksum of block of application files. Since, it require only a single round trip, it minimizes the effect of network latency. It computes a weak rolling checksum followed by a strong MD4 checksum. Following is the algorithmic description of the Rsync algorithm as described in paper [And98] .

The algorithm 4.1 and 4.2 represent the client and server of Rsync algorithm respectively. Client divides its application files into blocks of fixed size S. For each block, the client calculates a weak rolling checksum(explained in section 3.1.8) and a strong

checksum. The client sends these checksums to the server. The server generates a checksum of the files using 32 bit rolling checksum scheme which happens in following three levels:

1. It uses 16 bits of the 32 bit hash and creates a hash table for 216 entries. The list of client checksum is sorted according to 16 bit hash. Each entry, points to first element of list of that hash value and is null if there is no value. At each offset in the server file, its 32 bit rolling checksum and 16 bit checksum is calculated. If the value of that hash table entry is not empty, second level hash is invoked.
2. At this level, server scans the sorted checksum list starting with entry pointed by hash table entry, it looks for an entry whose 32 bit rolling checksum matches current value. Scan is terminated when its 16 bit hash differs. If this level finds a match then third level is invoked.
3. At this level server calculates strong checksum for current offset and compares it to the list of strong checksum from client. If it matches, match is confirmed.

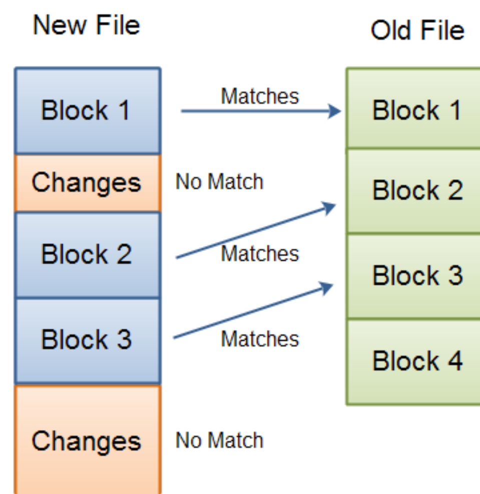


Figure 4.1: Rsync
[Jac14]

The calculation of the difference between old and new files can be understood by figure 4.1. In first step the checksums received from client are sorted. A hash table is maintained on the server, such that each entry of the hash table points to the list of checksums sent by client. A 16 bit hash and a 32 bit rolling hash is calculated for each files on the server. Each entry is checked against hash table. If the entry is not null, second level comparison is invoked where the values are matched, if the value gets matched third level gets invoked where strong checksum is calculated and if that matches, match is found.

4.2.2 Discussion

Rsync is a block-based approach where it calculates blocks of fixed size at the client and creates a rolling checksum. It first matches the blocks of fixed size and if it is matched it creates a rolling checksum. If a single byte is changed in starting position between the two files, rsync reads the entire file and generates the entire file as a changed file. Changing a single byte in a file is the worst case for rsync. If all the changes are clustered around one location it can be the best case for rsync.

We would not be using a block-based approach for finding the change as we consider sufficient amount of bandwidth and latency whereas rsync considers low bandwidth and latency. Block-based approach would be a bottleneck for the performance of the entire algorithm. Since, rsync considers low bandwidth, it utilizes that time in performing i/o operations like creating fixed size blocks and generating their hash.

4.3 Zsync

4.3.1 Algorithm Basics and Description

Zsync, as described in the paper [Col05b], is an algorithm that is similar to rsync. It is used for file synchronization of data that is distributed, for example a file is present on a server and can be downloaded by many clients. It does not use any special server functionality, a simple web server can be used to host the files. It has several advantages over rsync:

1. It uses the same algorithm as rsync but makes most of its computation on the client instead of server. Rsync computes the change between two files on the server, zsync computes the change in same way but it makes all the computation on client.
2. To perform web transfers it uses HTTP/1.1 compliant web servers, so that it can make the transfers through firewalls and it is more secure than Rsync.
3. Rsync can only handle compressed files that are compressed by patched version of gzip while zsync can handle the gzipped files in a special way.

As discussed in [Col05b] overcomes some of the drawbacks of rsync. Rsync does not store the checksums, it has to recompute all the checksums every time. Client can not compute checksum at all the offsets as it would be 4 times the total data. Server has to compute all the checksums as it can not pre-compute deltas. Hence CPU requirement at server is very high. Server has to maintain a hash table or a similar structure to save all the incoming hashes. Hence, a huge amount of storage is also required at server

side. Since the computation and storage power at the server is always very limited and it varies on different client, zsync overcomes this drawback by simply computing the deltas on the client and requiring a simple Web server with limited computation and storage power.

Zsync is client side rsync. The meta file can be calculated in advance at the server side storing hashes of fixed blocks. It does not need any special server as there is no per client calculation happening on the server. Zsync creates a control file that contains weak and strong hashes along with other metadata like size and permission of file blocks. It follows the same procedure to calculate delta as followed by [And98] that is by using strong and weak checksums. It makes a change by continuing the match procedure by weak checksum. If two consecutive block matches, the comparison is continued with weak checksum as chances of hash collision becomes less in this situation. Only after getting an unmatched block, a strong checksum of the huge block is calculated and matched. This reduces the calculations and also reduces time as MD4 checksum, that is strong checksum, takes more time to compute checksum than SHA1. Zsync algorithm works effectively with compressed files. Libzsync is the library that implements zsync and details about it can be read from [Col05a].

4.3.2 Discussion

As stated in the paper by [Col05b], zsync is client side rsync which saves all the computation on the server. As seen from the table 4.1 zsync was evaluated with sarge-i386-netinst.iso, with the user updating from the 2004-10-29 snapshot to the 2004-10-30 snapshot. The table evaluated the amount of data transfered with more number of similar files as there is a difference of only one day between two snapshots. Rsync performed better with very small and very large block size, while zsync performed well with the small and medium block sizes. Further tests were performed with different types of data files and different number of changes. These details can be read from [Col05b].

The performance of zsync was almost similar to rsync, the difference is that it does not require any special server. But zsync can not be considered for uploading bits to our scenario for the same reason as Rsync, as it makes many computations while calculating delta and while assembling the package.

	512 Bytes	1024 Bytes	2048 Bytes	4096 Bytes	8192 Bytes	16384 Bytes
Zsync-0.0.6	13278966	11347004	10784543	10409473	10357172	10562326
Rsync		9479309	9867537	9946883	10109455	
Zsync-0.2.0	10420370	10367061	10093596	10111121	10250799	10684

Table 4.1: Evaluation Result of Rsync and Zsync
[Col05b]

4.4 BURP

4.4.1 Differences Between Rsync, Zsync and BURP

Burp [G P14a] is a solution used for backing up data from a computer to multiple and heterogeneous systems. It uses file synchronization algorithm similar to Rsync and zsync. It uses librsync [Marc] which is the library implementation of rsync. Burp uses a client server architecture and all the new files are stored on a central server. It works in following four different phases :

1. In the first phase, the client scans all the files that it need to synchronize. In this scan it does not open any file, it just records the metadata of the files and sends it to the server. Since no file is opened, this phase is completed in very little time. Server saves all this information in a manifest.

2. Server goes through the manifest created by it and also through the manifest that it had from last synchronization.

   - If a file is there only on new manifest and not on old manifest, server asks for the entire file from the client.

   - If a file is there on both manifests and the modification date is same on both, server skips that file for synchronization.

   - If a file is there on both manifests and the modification date is different on both, server asks the client to synchronize by librsync. Server sends a list of all hashes of such files.

3. In the third phase, the client only sends back the changed blocks to server and breaks the connection with server.

4. In the fourth phase, the server assembles the package using the final manifest. It needs a lot of time to assemble the package as it performs multiple steps mentioned in [G P14b].

We have seen that Rsync, Zsync and Burp algorithms work in similar ways. Burp (as mentioned in [G P14a]) uses sparse indexing and variable length chunking when it uses librsync to synchronize. The main algorithm in all three of them remains the same. Zsync and Burp calculates the changes in files by calculating them on client unlike rsync that does it on server.

4.4.2 Discussion

All the algorithms discussed above are the variations of rsync. The basic idea of rsync remained the same in all of them, but zsync and BURP works on the shortcomings of rsync. Both are able to over come some shortcomings of rsync but they still can not be considered as a solution candidate of our problem as they all require a lot of computation overhead. This time would add on to the total time taken to complete the process of file synchronization. Since, we are considering a reasonable bandwidth between client and server, we are looking for a solution where minimum amount of CPU could be used. For this purpose we discuss the next approach which makes use of minimum amount of CPU and does not create any bottle neck for network latency.

4.5 Map Based

4.5.1 Different Phase Description

This algorithm, as described in [Tor04], is based on recursive splitting. Recursive splitting is a practice of splitting a file recursively starting from entire file to a fixed sized blocks. It works in two phases **Map Construction** and **Delta Compression**. It considers that the application files present at the client are old files (represented by f_{old}) and at the server are new (represented by f_{new}). It requires multiple rounds of communication over the network. It assumes a slow network hence to overcome low latency it makes some computations at both ends. It uses hash values to find common substrings in f_{old} and f_{new}. Following are its two phases:

1. **Map Construction:** This is the phase where algorithm calculate the difference between two files. Both the client and server uses multiple round trips to calculate difference. The client generates a map of the files present at the server f_{new} and

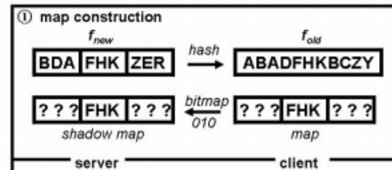


Figure 4.2: Map Construction

the server creates a shadow map to keep track of map. Figure 4.2 represents the map phase. The client and the server splits file into blocks of fixed size S. For each block, the client calculates a hash and sends it to the server. If any of the blocks sent by the client, matches at the server, it returns a 1 else a 0 (marked by a question mark in the figure 4.2)and saves a shadow of its response. After receiving the response, for all the block that client has received a 0, it repeats the process by splitting those blocks again. This process goes on until a fixed block size is reached or until literals are reached. Client collects unmatched blocks and sends it to the delta compressor to compress.

2. **Delta Compressor:** This phase transmits unknown parts of f_{new}(calculated by map construction phase) to the server. Client creates a f_{ref} with parts of f_{new} that are known by server and a f_{target} file of rest. Client compresses the f_{target} with respect to f_{ref}. Server after receiving the files recreates the map and assemble the file to obtain f_{new}.

4.5.2 Discussion

It utilizes maximum bandwidth by making use of multiple rounds. It can be further improved by sending verification hashes from client to server. But verification rounds would add to the number of messages exchanged between client and server. In a PaaS cloud message exchanged is a costly operation. This algorithm requires a lot of messages to be exchanged in order to find different data. Also this algorithm very efficiently computes small blocks of data in a file. But in a PaaS cloud, CPU computation would become a overhead as we do not have the restriction of low bandwidth.

4.6 Bloom Filter

4.6.1 Working

It is a method to find an element from set of elements. If an element is present in a set and it should return true else it should return false. Bloom filter was implemented by Burton H. Bloom in his paper [Bur]. Bloom filters are lighter version of hash tables. Time complexity to find an element in the bloom filter is same as that of a hash function that is O(1).

As described in the paper [Bur], Bloom filter is an n bit array with all the bits set to 0 value. A bloom filter uses m different hash functions, each of m different hash function hashes to a position at one of the bits of n bit array randomly. Figure 4.3 explains the concept of bloom filter. For example, there are m different hash functions like h_0,h_1, h_2,h_3...h_m and each of them would evaluate to on position between 0.....n-1, where m is smaller than n. If the element is book, h_0(book) would be a value between 0.....n-1. Similarly for all the m hash functions each element would be evaluated. The issue with

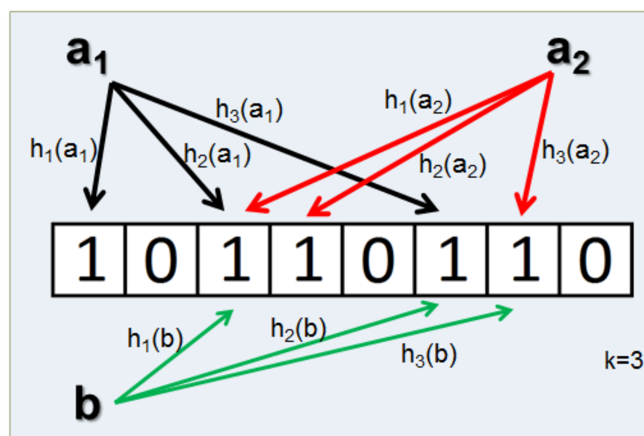


Figure 4.3: Bloom Filter
[Wer17]

bloom filters is that it can give false positives.A false positive means that the element is not present in the set but bloom filter returns true. This arises because bloom filter uses multiple hash function for each element. If bloom filter finds all the m positions set, it returns that the element is already present, else it sets those positions as true. As shown in the picture 4.3, there are 8 bit positions in the bloom filter. It uses 3 hash functions. Each hash function is setting one bit for each element. So for element a1 position 0, 2 and 5 are set by hash functions h_1, h_2, h_3 respectively. For element a2 position 2, 3 and 6 are set. This time third and sixth positions are found as free hence it moves to the next

element. But for element b, positions 2, 3 and 5 are checked by h_1, h_2, h_3 and all of them were already set by some different function and element. For element b, bloom filter will return true that the element is already present in group when it is actually not.

4.6.2 Discussion

Bloom filters are widely used and have many applications in resource matching. For example it is implemented by Quora [quo09] for finding out the feed are read by the users. It has many other applications. The shortcoming of false negative can be reduced to minimum by using many methods like inverted bloom filters, counting bloom filters, multilevel bloom filters etc as discussed in paper [Sal05].

Due to the issue of false negative we can not consider bloom filters for synchronization of files in PaaS cloud. The error can be reduced to minimum in bloom filters but this error would cause unreliable synchronization of application files. We are looking for a solution that has almost a similar time and space complexity but it should reliably synchronize the files between the client and server. Hence we look at the next approach that uses bloom filters as a part for synchronization but not as a whole.

4.7 TAPER

4.7.1 Different Phases

Taper [Nav05] is a hierarchical redundancy elimination protocol for replica synchronization. It does not require any knowledge of the state at client or server. Unlike rsync it does not depend on the path or name of file. It works in multiple phases and also assumes low bandwidth. It works in four stages and uses bloom filters in one of them. It provides four properties:

1. It provides a scalable solution as it assures a low reusable computation at source.
2. It reduces the matching time as it uses hash tables and bloom tables for matching at the server
3. It finds out maximum similarity between the client and server.
4. It also uses bandwidth efficiently as it minimizes the total amount of meta data.

In Figure 4.4, we describe the four phases that this algorithm is divided into.

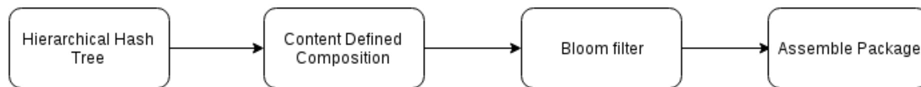


Figure 4.4: Phases of TAPER

1. **First Phase:** This phase eliminates all the common files and directories using content based hierarchical hash tree. This tree is formed using file contents and hash functions. Client encodes the directory structure and contents of directory as a list of hash values for every node in the tree. Figure 4.5 is a directory tree structure. This tree structure can be understood by the Merkle tree explained in Chapter 3, Section 3.3. Nodes are root directories, sub directories, leaf directories and files. The hashes are precomputed and stored in a global or persistent database at client. A persistent database enhances computational efficiency. Server computes hierarchical hash tree of its directory tree and stores each in a hash table. Each element of hierarchical hash tree sent by client starting root node of the directory tree and if necessary progressing downwards to file nodes is used to index into target hash table. At the end all the common files and directories are pruned.

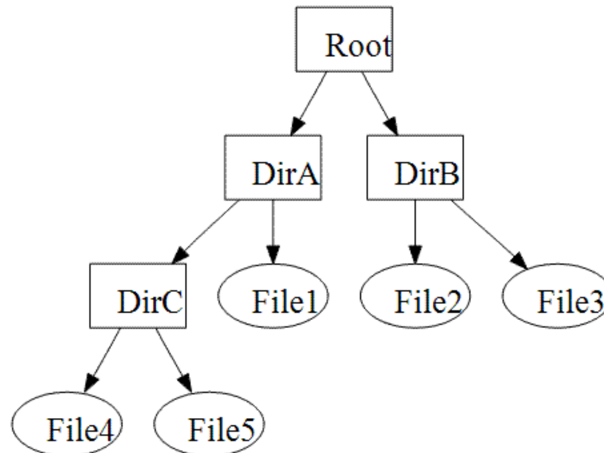
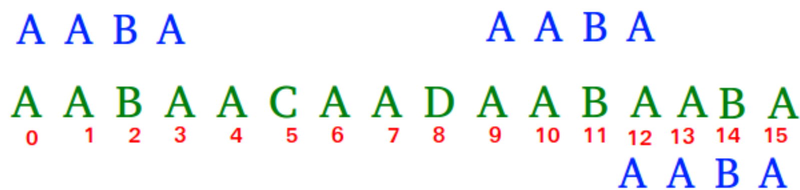


Figure 4.5: Hierarchical tree structure

2. **Second Phase:** Once all the same files and directories are eliminated, unmatched files are left. During this phase, client sends SHA1 hash values of unique content defined chunks of all the remaining files to server. Content defined chunk hashes can be indexed for fast matching . Content defined chunking can be computed by Rabin Karp fingerprinting with some marker value. Figure 4.6 represents content defined chunking. Chunks can be matched using simple hash lookup. Content defined chunking perform better for large blocks. Client removes all the similar blocks from the data.

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A



Pattern Found at 1, 9 and 12

Figure 4.6: Content Defined Composition
[gee]

3. **Third Phase:** After completing the second level of matching only those chunks are left that did not match any chunk at server. At the third level two contiguous chunks of same file are merged together and are called holes. To reduce the size of holes, finer grained block matching is used. The sliding block match can be applied to pair of files to match pair of holes. Bloom filter(as explained in 4.6) is used for finding the similarity. A strong checksum is used to calculate hash of fixed size blocks of relatively smaller size. Unmatched blocks are sent to client.
4. **Fourth Phase:** Client sends unmatched blocks to server using delta encoding and server sends back the checksum to validate data.

With each phase it reduces the data granularity, it starts with directories, files and then reduces to larger chunks, blocks and finally reduces to literals. The level of granularity can be seen in figure 4.7. Granularity decreases with phases and complexity increases.

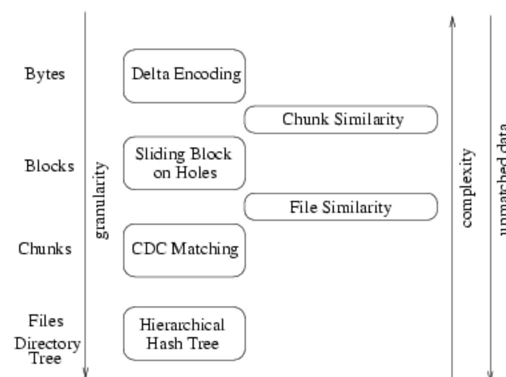


Figure 4.7: Level Of Granularity

4.7.2 Discussion

This algorithm reduced the amount of data at every phase. In first phase it prunes out similar files and directories. In second phase it prunes out similar chunks of data in a file that was not found similar in phase I. In third phase chunks are reduced to blocks and finally unfamiliar blocks are sent to the client. It used very clever techniques at every phase to prune out common data. But when considering a PaaS platform there are a few things that would affect the performance of file matching algorithm. Following are a few points which decide if it is a good candidate for a PaaS platform or not.

1. **Multi Request:** Since there are four phases and after each of the four phases, either the client or the server has to open a connection. This could be a costly operation when scaling in a PaaS cloud is considered.
2. **Hierarchical Hash tree:** This phase calculates the hash of all the files and folders of the application. It prunes out same files and folders in one run. Hence it is one of the most useful part of the algorithm. Since, we consider a reasonable amount of bandwidth, this phase is very fast in calculating the changed files.
3. **Storage of Hash on client and server:** This algorithm saves the hashes in a global cache which is accessible by both. For a PaaS cloud we do not want to save the state of client in any form.
4. **Bloom Filters:** As discussed in section 4.6, bloom filters can give false positives. In a PaaS cloud we can not afford to have false positives.
5. **Computation:** This algorithm depends on a lot of computation eg., computation of tree, computation of SHA1, computation of Bloom filter, computation of content defined composition etc. Since we have a reasonable amount of bandwidth, this much computation would become a bottleneck for the overall performance.

Since there are many factors that would make this entire algorithm unsuitable for a PaaS cloud, we cannot use the entire algorithm. But we can use the first phase of this algorithm to find different files at client and server. In chapter 5 we will evaluate this against other candidates for resource matching in PaaS cloud.

4.8 Stack Sync

4.8.1 Architecture and Working

Another possible solution for file synchronization is saving the state of the target on the source. A similar approach is practiced by Stack sync in the paper [Ped13]. This paper presents an architecture and approach for file synchronization. Figure 4.8 shows the architecture diagram of stack sync. Figure 4.8 shows a client that combines with Operating system file explorer capabilities. It monitors the change in its application files on client with the help of operating system. It calls the application package as workspace and monitors the workspace for any change in files. Component that monitors the workspace is called as watcher in stacksync. If there is any change a component is notified called indexer. Interaction between the components can be seen from figure 4.9. Indexer also maintains a local database on the client that stores all the files in chunks. These chunks are hashed with SHA 256 and are identified with the SHA value. This

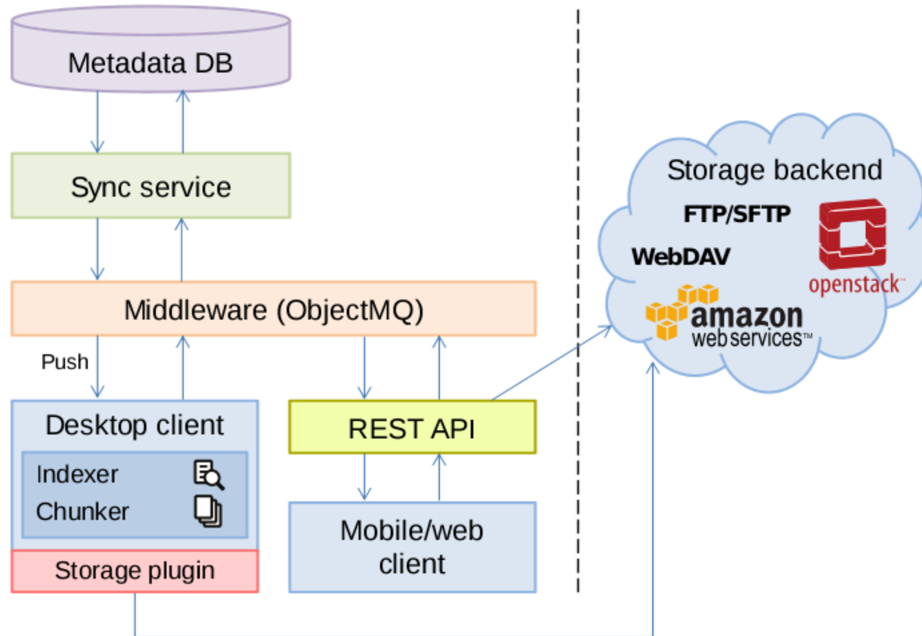


Figure 4.8: Architecture of Stack Sync
[Ped13] [Ped13]

stores the state of the application so that only the changed parts could be sent to storage back end so that bandwidth and storage can be saved. Indexer de-duplicates the data and save files as fixed length or variable length chunks. There are components like synchronization service that will check for existing chunks in case of cross- user. It also uses a metadata back end that stores metadata into a MySQL or NoSQL database.

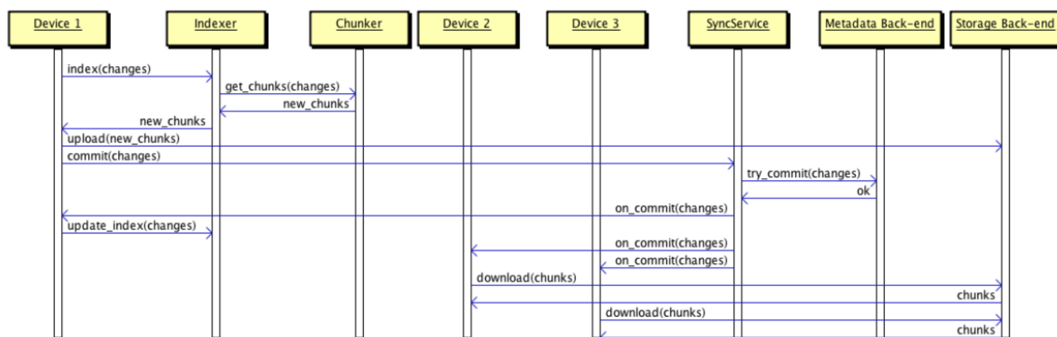


Figure 4.9: Interaction of different components of Stack Sync
[Ped13]

If a new file is added chunks are added to database and if existing file is changed only respective chunks are indexed. Indexer compares chunks in local database to the newly created chunks, if they exist, only newly created chunks are updated to the server. Once a unique block is stored into storage backend indexer will communicate with metadata backend via synchronization service.

4.8.2 Discussion

This approach is a very fast way to calculate changes. It also sends just the changes without almost any duplication hence it saves a lot of network connections and bandwidth. The reason here is that it stores the state at client which we do not consider as one of the safest option. Saving the state at client assumes that the exact state is at the server, but we need to consider the situations like server outage or corrupt disk at server. Considering these situations we do not consider this approach of saving state.

4.9 Set Reconciliation

4.9.1 Description

Set reconciliation problem as discussed in paper [Hao08] is an algorithm that synchronizes the files from client and server together by finding out the common and uncommon parts from both at same time. Both the client and server creates a set of all the files or file chunks. Both of them decides which part is present in the union of the two sets and which are present in the intersection of two. The goal of this paper is to reduce the number of messages exchanged between the client and server while calculating the number of changes between the sets. This can be calculated by using the symmetric difference between the file and this can be done using characteristic polynomial which can be studied from [Hao08].

The paper [Hao08] assumes that it needs only one message to be exchanged between the client and server. This algorithm computes the performance in terms of number of bits sent over network. The client uses a 2 way min technique(described in the paper) to calculate overlapping blocks and calculates hash of all of them. Client uses set reconciliation to calculate the symmetric difference and the in just one message it is transferred to server. The server sends back to the client, the parts that are not already known in a special way. It sends the parts in a zip file that is gzipped and sends the metadata to reconstruct the new file by encoding it with special golomb codes explained in [Ian99].

4.9.2 Discussion

This approach has many advantages over the other approaches discussed. It uses just a single message to calculate the changes. Hence it saves a lot of bandwidth. It computes 2-way-min as described in [Hao08] which is a content-dependent partition technique. It is better than calculating fixed size blocks (calculated in rsync, zsync etc) because it will not affect the entire file if there is a very small change in the start of the file. The purpose is to find the optimal value for the set difference which should not be very high(else the server would send every thing) and it should not be very low, otherwise all the changes would never reach client.

It uses an algorithm that can be read from [Hao08]. This approach is able to find minimum set difference but with a error percentage of 0.1 percent. This value is very less but error percentage of any range can not be accommodated for synchronization of application files to the PaaS cloud. It could be a huge loss even if 1 in 10 applications are not synchronized properly on the cloud.

4.10 Git Approach

4.10.1 Description

Git as explained in [CS14] is a source code management service that is hosted on the Internet. It saves and allows to view all the versions and change history of the files stored on it. It is a distributed version control system which keep files synchronized in a distributed environment even when the server is down. It is the most commonly used source code management system. It calculates the difference between two files using diff algorithm described in FABIEN SANGLARD [FAB]. The approach used by git to synchronize application files is described in [Sco09].

Git is also based on client server architecture. Where the client makes changes to the files and sends it to server. As explained in [Bac] when a directory is made as git directory, the files and directories inside it are monitored and all the changes in that directory are tracked. The repository that is present on each client is called a local repository and there can be many local repositories connected to a project that resides on a remote repository. A remote repository are remote servers that take care of all the projects.

Figure 4.10 represents the entire work flow. Workspace is the working tree, index represents index file, local and remote repositories represents the client and server repositories. Along with local repository, client maintains an index file where changes are stored and working tree where the whole project resides. When there is a change

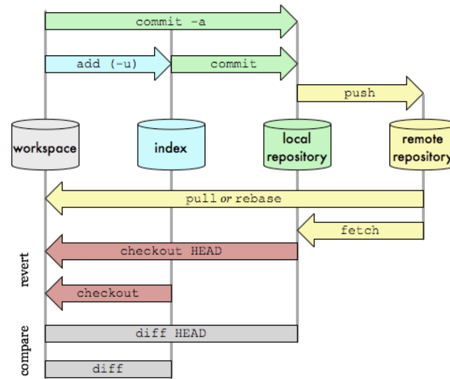


Figure 4.10: Git Workflow
[Von]

in the project i.e., when there is a change in working tree, those changes are added to index and from there they are sent to local repository. And when a push command is run the changes are merged with remote repository.

4.10.2 Discussion

The approach used by git is fast and efficient. Git maintains all the versions in the local repository. Whenever the client makes any change to the files or folders in workspace, it is recorded in the index file and all those changes are committed to local repository of the client on running commit command. These changes are saved to the remote server on running push command. We have not considered the factor of monitoring the changes and saving them locally. Git uses diff (as explained in [FAB]) algorithm for calculating changes between the current workspace and any of the previous commits. We have not considered the approach of maintaining a file that records all the changes of that particular commit. Git stores the state of its last commit in the local repository. We have not considered saving state at client as a safe option.

5 Suggested Approaches

In this chapter, we will discuss four (suggested) solutions to file synchronization in a PaaS cloud. As discussed in chapter 3, file synchronization is a process of updating a file on the client/server when the same has been updated on the server/client. In all our approaches, we divide file synchronization into three different phases: fingerprint calculation, match phase and assemble package phase. Using our solutions, we are reducing the total time of file synchronization, improving user experience, reducing the number of requests over the network, optimizing the usage of resources like bandwidth and storage. In this chapter, we design three approaches that optimizes all these factors. We discuss all the three approaches with their implementation, design details and their improvement over the existing approach.

We start with the discussion of our target problem domain. We discuss different types of applications (to be pushed into Cloud Foundry) with different characteristics (generalizing all other applications with similar characteristics). In the further section, we discuss the assumptions that we make while designing the approaches, followed by the (existing and suggested) approaches and their design details. In the last section, we summarize all the approaches with a high level description of the differences among them. In the next chapter, we discuss the application characteristics and evaluation result of all the approaches.

5.1 The Problem

In this section, we describe the problem domain of "Resource Matching in PaaS cloud", that we are to solving through this thesis. We know the problem of file synchronization from chapter 2. There are many solutions available to this problem as discussed in chapter 4. There are some assumptions made by existing approaches as discussed in chapter 4. Some solutions assumed that the available bandwidth is low, some assumed it is high. Some other assumptions are regarding the database at the server side, they assumed that the files are stored in a local database on the server.

As we have discussed in chapter 2, we are solving the issue of uploading the application bits to the Bits-Service (that is hosted in a remote server). In the following two situations, we apply file synchronization in our approaches: First, when a CF Client creates an application (during first push), it uploads the application files to the cloud and second, when the client changes its application files (during subsequent push), it uploads the updated application to the cloud. We are always uploading only application files to the approaches that we are designing here. An application consists of different types of files, depending on the type of application: banking applications, insurance applications, engineering design applications, health care applications, marketing applications, gaming applications etc. It also depends on the type of programming language used to write these application. To understand the problem we are solving, it is important to understand the structure of files and folders created by different programming language. Some prominent examples are:

Java: A typical Java application has thousands of JAR files and has a complex directory structure. For example: /src folder has all the class files, /test folder contains files for unit tests and other different tests, /docs folder is for manually generated and edited files. /lib folder is for third party libraries, /bin (or /classes) folder is for compiled classes or for storing the output of compiled classes and /dist is for auto generated by build system.

Node.js: A typical node application has a complex directory structure as well with thousands of files. It has a /src folder that contains different folders like /graphics, /style, /routes etc. /Graphics folder contains all the graphics for the project, /style contains all the css files and so on. Other than /src folder there are other different directories like /lib that contains third party libraries, /scripts, /controllers, /config etc.

We can see that, each type of programming language has its own type of folder structure and can have a range of different number of files. We have to consider these factor while solving the issue of file synchronization.

Another factor that needs to be considered while solving the issue of file synchronization is that the application files and folders are not stored on a local blobstore on the server but they are stored in a remote blobstore. Since the Cloud Foundry is based on IaaS cloud, we have to consider network as an important factor as it will cost bandwidth and connection cost.

Different types of folder structure, different number of files and a blobstore are the additional factors that needs to be considered while designing an approach for uploading application bits from CF Client to Bits-Service. In following section we will discuss the assumptions we make while designing the approaches. In the following section we will look at the existing approach for uploading bits to the Bits-Service followed by the suggested approaches.

5.1.1 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, we have a reasonably high bandwidth. By reasonably high, we mean it is not high enough that we can push the entire application every time we perform 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. We ensure that CPU computations does not become a bottleneck while calculating changes in the application.
2. **CF Client/Server:** We are running the CF Client and server from same machine for implementation and evaluation(although we will use a bandwidth limiter to simulate the real world scenarios).
3. **Storage:** We assume to have unlimited storage at the Blobstore. We have chosen Softlayer for the implementation, storage and evaluation of the approaches.
4. **One way synchronization:** We need to synchronize only from CF Client to the Bits-Service and not vice-versa.

5.2 Current Approach

In this section we discuss the Current approach5.1 described in [Gly16] and its approach towards file synchronization. To upload an application, the CF Client computes the fingerprints of all the blobs in the application. The CF Client sends a “match” request to the Bits-Service which includes a list of fingerprints of all the blobs. The Bits-Service responds by returning all the blobs that are already available in the blobstore. The CF Client then uploads the missing blobs from the application as a zip to the Bits-Service. The Bits-Service assembles the application using the zip file and downloads known files from the blobstore. The Bits-Service stores blobs greater than 64 KB in size, to the blobstore, a majority of small blobs, e.g. class files are not stored in the blobstore.

However, the CF Client is unaware of this and it includes all blobs (fingerprint) in its resource matching query to the Bits-Service. Following are the steps for Current Approach, CF Client and Bits-Service are represented by 5.1, 5.2 respectively.

Algorithm 5.1 CF Client

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

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

PUT /bits *zipblob*

Algorithm 5.2 Bits-Service

/match do
$knownfingerprint[] \leftarrow$ SHA known *by Bits-Service*
Bits-Service receives *List of fingerprints*
for each $sha \in \mathcal{L}ist$ **do**
if $size > 64kB$ **then**
HEAD sha to blobstore
if $request.status =$ *200* **then**
$knownfingerprint[] \leftarrow sha$
end if
end if
end for
return *knownfingerprint[]*
end /match

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

5.2.1 The Approach

In the figure 5.1, there is a CF Client communicating with the Bits-Service and a blobstore that has the application cache and stores applications in package cache. This approach works in following steps

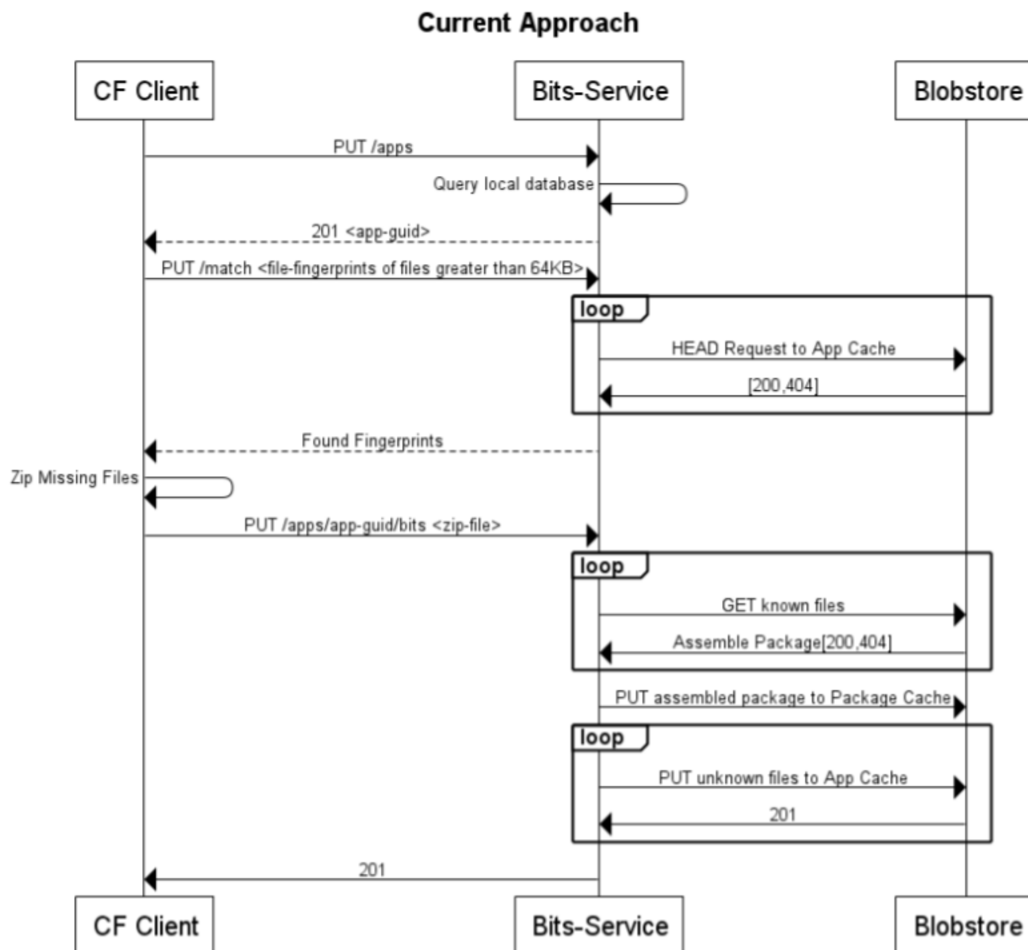


Figure 5.1: Current Approach
[Ste16a]

1. In the first PUT request CF Client sends application name to the Bits-Service and the Bits-Service creates a new GUID for the CF Client and sends the GUID with status code 201 to the CF client. If the local database in Bits-Service already has the GUID, server sends the GUID with status code 200.
2. In second step, the CF Client calculates SHA of all the blobs present in application directory and sends these as a JSON list with other details like organization, GUID,size to the Bits-Service.
   a) If the application is uploaded for the first time to the Bits-Service. It will return the same list to CF Client as the unknown blobs.
   b) If it is not the first time the application is uploaded to the Bits-Service, it will make HEAD request to the application cache for all the files that are

greater than 64KB in size and it will send back a list of all the fingerprints that matched with the fingerprints sent by CF Client.

3. CF Client receives a list of fingerprints that are known by Bits-Service and creates a zip of all the blobs that are not found and that are less than 64KB and sends the zip to Bits-Service.

4. Bits-Service unzips the zip file received from CF Client. For each file with size more than 64KB it calculates SHA and saves it with the name of the file as its SHA in the application cache.

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

   b) If it is not the first time the application is pushed, Bits-Service downloads each application file that is related to the application and creates the entire package. It uploads the files(greater than 64KB) that are sent by CF Client in the zip to application cache. After assembling the package it uploads the entire package back to package cache.

5. Finally, when all the blobs are stored on the Blobstore, Bits-Service sends a status code of 201 to CF Client and this completes the entire process.

5.2.2 Implementation

A generic implementation of all the approaches can be found in Chapter 6. For implementing this approach we used Sinatra for server side programming and Ruby REST Client for Client side programming. We installed different gems eg. JSON for the data exchange between CF Client and server, File utils for using files on the local machine, digest for using SHA algorithm, REST-Client for using REST calls, pathname for creating paths, ruby zip for creating zip of application files, aws-sdk for connecting to remote blobstore and sequel for using SQLite database. In addition we used a local database on server side which stores some information about the applications. For this purpose we used a SQLite database that is a file based database for storing the basic information. We used the Softlayer as our remote database for storing application files and application packages.

5.2.3 Space and Time Complexity

Time complexity at CF Client can be calculated by the operations performed by CF Client. Following are the operations performed by CF 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 CF Client goes through each blob and calculates fingerprint 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 fingerprints from Bits-Service, CF Client goes through the list of fingerprints that it already has and calculates the fingerprints 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, CF Client puts all unknown blob into a zip blob. This operation takes O(n) time.

So, the total time taken by CF Client is: **O(1)+O(n)+O(n)+O(n) = O(n)**
Total Space complexity on CF Client is: **O(n)** as it saves the application blobs only once.

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 CF 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 CF 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 fingerprints from CF Client, it makes a HEAD request for each fingerprint to the Blobstore. This operation take O(n) time, where n is number of fingerprints sent by CF Client.
3. **Saves changed/new blobs on Blobstore:** When Bits-Service receives changed/new blobs from CF Client, it downloads the zip file from blobstore and assembles the package. This operation takes O(n) time, where n is number of changed/new blobs.

So, the total time taken by Bits-Service is: **O(n)+O(n)+O(n) = O(n)**, where n is either number of blobs on CF 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.

5.2.4 Discussion

Since this approach saves only the blobs with size greater than 64KB in Blob store, first upload to the Blob store is faster as compared to the second upload. For the blobs with size less than 64KB, the Bits-Service does not have to make POST request to Blob store to save the blob. This saves some storage space, bandwidth from Bits-Service to Blob store. This approach is used because it is not possible to store all the blobs on the Blob store as it would cause a large number of PUT requests on first upload and large number of HEAD and GET requests in subsequent upload (as described in[IBM13])

As described in [IBM13], this approach does not store blobs of size greater than 64KB, CF Client needs to send all the blobs that are less than or equal to 64KB in every push. Some prominent examples are: zip kin web Jar contains over 16000 blobs, including over 15,600 .class blobs, linux kernel contains over 57000 and more than 54,000 are less than 64Kb. The JAR of zipkin is 32 MB zipped and contains 72 MB of data, Linux Kernel is 350 MB zipped and contains 600 MB of data.

Figure 5.2 represents a pie chart with the estimate of time taken by different phases.

1. Time T1 represents the SHA calculation phase. This phase is the total of the time taken by client to read all the files and to calculate their SHA.
2. Time T2 represents the match phase . This phase is the total of the time taken by client to send the SHA to the server and receive back the changed SHAs.
3. Time T3 represents the total time taken by client to zip the changed files and upload them to the server.
4. Time T4 represents the total time taken by server to assemble the package. In this phase the server receives the changes from client and downloads all the required files from the application cache.
5. Time T5 represents the total time to upload the package from server to the blob store.

Figure 5.2 show that it took 8-10 percent of time to upload bits to Bits-Service and 45-50 percent of time to assemble the package by GET and PUT requests and 20-25 percent to perform the resource match with HEAD requests.

In this approach, Bits-Service sends a list of fingerprints that are already present on it, CF Client after receiving this list calculates the fingerprints 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 is, it does not deal with deletion of blobs. If a blob is deleted by CF Client, it does not get deleted at the application cache which leads to unnecessary load on Blob store.

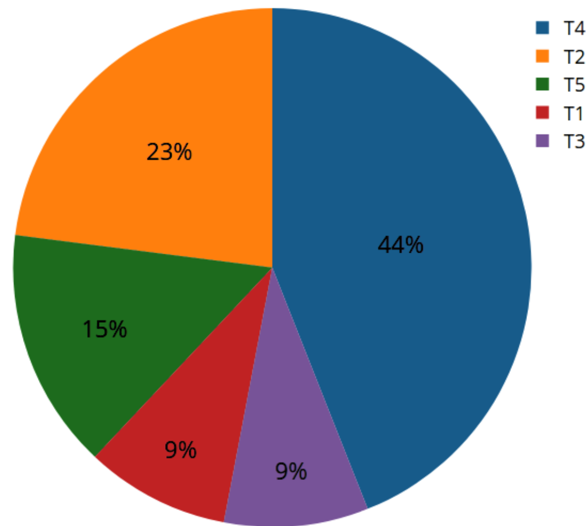


Figure 5.2: Current Approach Pie Chart

5.2.5 Significance of the Current Approach

This is the approach that is currently implemented for file synchronization in the PaaS cloud. It reduces the total number of requests over the network by only requesting for the files that are less than 64kb. In general, 90 percent files in each applications (that we evaluated) are less than 64kb. Also, it optimizes the storage by reducing the total number of files stored. It stores the files which are less than 64kb in the remote blobstore.

5.2.6 Improvements Required for The Current Approach

This approach does not store blobs of size less than 64KB and therefore, 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 a few of these blobs have changed and we lose a lot of bandwidth from CF Client to Bits-Service and we lose a lot of time while uploading these bits. It makes a lot of PUT and GET requests while assembling the package. This incurs a lot of connection cost and time. To overcome some of these issues we discuss the Local DB based approach.

5.3 Local Database Approach

To overcome the shortcomings of the above approach, Matthew Sykes suggested an approach where he introduces a local cache for the Bits-Service that can hold the fingerprints of the blob, the size of the blob, the last accessed time and the hit count. When a resource is successfully matched, it gets updated on the local cache with fingerprint of file, access time and the hit count is incremented. The initial cache population can be done by iterating over all of the objects in the Bits-Service. The last accessed time should be the creation time from the Blobstore metadata and the hit count should be set to 0. A delayed job is setup to purge any resources in the cache that has a low hit counts or have not been accessed within some period of time. Both of those values are 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 which 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 fingerprints 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. In the previous approach it is 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. Only those blobs are uploaded that are actually changed. Another small improvement is over the calculation of difference that is made by both CF Client and Bits-Service. In this approach we send a list of unknown fingerprints to the CF Client so that CF Client does not have to calculate it again. Following is the approach describing the steps at the CF Client 5.3 and Bits-Service 5.4.

Algorithm 5.3 CF Client

```
fingerprint[] ← SHA of each blob
for each blob ∈ Application do
    fingerprint[] ← Digest::SHA blob
end for

unknowfingerprint[] ←
PUT /match  fingerprint[]
for each sha ∈ Unknownfingerprint[]
do
    folder ← blob corresponding to sha
end for
zipblob ← zip(folder)

PUT /bits  zipblob
```

Algorithm 5.4 Bits-Service

```
/match do
unknownfingerprint[] ←
SHA not known by
Bits-Service receives List of fingerprints
for each sha ∈ List do
    SELECT * FROM table
    WHERE fingerprint= sha
    if fingerprint.exist = false then
        unknownfingerprint[] ← sha
    end if
end for
return unknownfingerprint[]
end /match

/bits do
folder ← unzip(zipblob)
for each blob ∈ Folder do
    if blob.size > 64kB then
        blobfingerprint ←
fingerprintofblob
        PUT blobfingerprint, blobcontent
        to Blobstore
    end if
end for
end /bits
```

5.3.1 The Approach

In Figure 5.3, the CF Client is communicating with the Bits-Service and a blobstore that stores application package. This time a local cache is maintained at the Bits-Service. This approach works in following steps

1. In the first PUT request, the CF Client sends application name to the Bits-Service and the Bits-Service creates a new GUID for the CF Client and sends the GUID with status 201. If the local database in Bits-Service already has the GUID it sends the GUID with status 200.

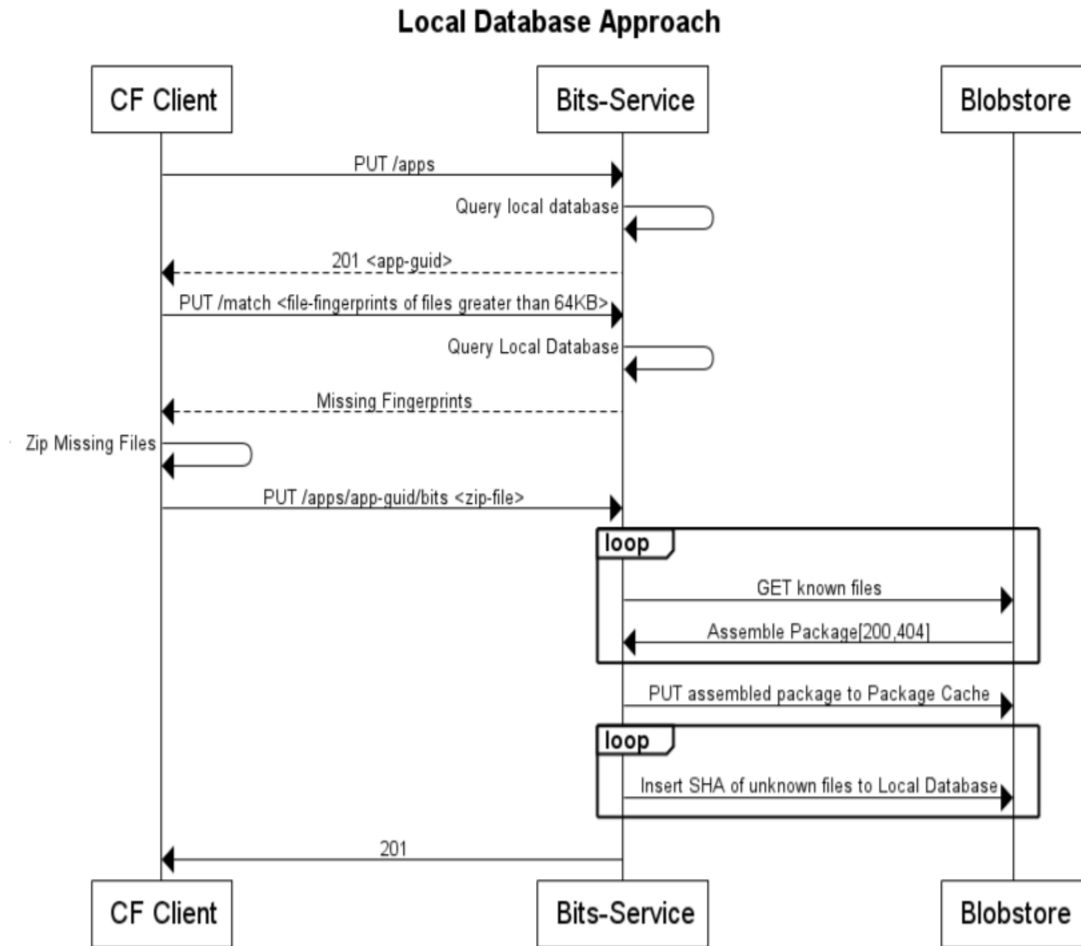


Figure 5.3: Local Database Approach

2. In second step the CF Client calculates SHA of all the blobs present in application directory and sends this as a JSON with other details like hit count, access time, size, organization, GUID to the Bits-Service.

    a) If it is the first upload of the application bits, it saves list of fingerprint in its local cache and return the list to CF Client as a list of unknown fingerprint,

    b) If it is not the first time the application is uploaded to the Bits-Service, it queries its local cache for each fingerprint in the list and for all the fingerprint that are not present in the local cache, the Bits-Service sends them as a list of unknown fingerprints. For all the fingerprints that are available in the local cache, it increments the hit count by 1 and changes the access time of that SHA.

3. CF Client receives the list of unknown fingerprints and creates a folder with all the blobs that corresponds to those fingerprints after adding all the unknown blobs to a folder, CF Client creates a zip blob of the folder and sends it to Bits-Service.

4. Bits-Service unzips the zip file received from CF Client. For each file with size more than 64KB it calculates SHA and saves it as the name of the file in application cache. Then it saves each of this file in the application cache.

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

   b) If it is not the first time the application is pushed, Bits-Service downloads each application file that is related to the application and creates the entire package. It saves the files(greater than 64KB) that are sent by CF Client in the zip to application cache. After assembling the package it uploads the entire package back to package cache.

5. Then it changes the name of blob to its fingerprint and makes a PUT request to Blobstore(Application Cache) to store the blobs. Blobstore replies with a 201status code for each blob and after all the blobs are saved on Blobstore, Bits-Service sends a 201 status code to the CF Client.

5.3.2 Implementation Details of Local Database Approach

For implementing this approach, we used Sinatra for server side programming and Ruby rest CF Client for CF Client side programming. We installed different gems eg. json for the data exchange between CF Client and server, File utils for using files on the local machine, digest for using SHA, rest-CF Client for using rest calls, pathname for creating paths, ruby zip for creating zip of application files, aws-sdk for connecting to remote blobstore and sequel for using SQLite database. In addition we used a local database on server side which stores some information about the applications. For this purpose we used a SQLite database that is a file based database for storing the metadata information. We used Softlayer as remote blobstore for storing application packages.

5.3.3 Space and Time Complexity

Time complexity at CF Client can be calculated by the operations performed by CF Client. Following are the operations performed by CF 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 CF 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, CF Client puts all unknown blob into a zip blob. This operation takes O(n) time.

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

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 CF 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 CF Client. This operation take O(n) time where n is the number of applications on Bits-Service.

2. **Query local cache:** When the Bits-Service receives list of fingerprints from CF Client, it queries for each SHA against the local cache. This operation take O(n) time, where n is number of fingerprints sent by CF Client.

3. **Saves changed/new blobs on remote and local cache:** When Bits-Service receives changed/new blobs from CF Client, it makes a PUT request for each of them and saves them on the 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 CF 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.

5.3.4 Discussion

This approach saves fingerprints of all the applications on a local cache. It save all the HEAD requests by querying local database, instead. This approach has same time complexity as that of Current Service5.1 but has a different and greater space complexity than Current Service5.1. 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.

Figure 5.4 represents a pie chart with different phases involved in file synchronization.

1. Time T1 represents the SHA calculation phase. This phase is the total of the time taken by client to read all the files and to calculate their SHA.
2. Time T2 represents the match phase . This phase is the total of the time taken by client to send the SHA to the server and receive back the changed SHAs.
3. Time T3 represents the total time taken by client to zip the changed files and upload them to the server.
4. Time T4 represents the total time taken by server to assemble the package. In this phase the server receives the changes from client and downloads all the required files from the application cache.
5. Time T5 represents the total time to upload the package from server to the blob store.

In figure 5.4, we can see that the total time of match phase has reduced to 2-5 percent. The reason of this reduce is the elimination of HEAD requests and addition of a local database containing metadata. The total time of assemble phase still remains the same as we are assembling the package by a lot of PUT and GET requests. The time taken by other phases still remains the same.

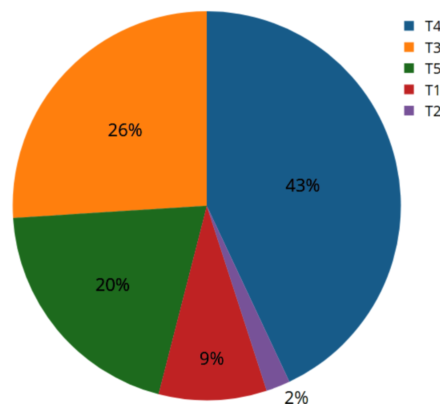


Figure 5.4: Local Database Approach Pie Chart

5.3.5 Significance of Local Database Approach

This approach is an improvement over the existing approach. It saves all the HEAD request as it stores all the metadata information in local database. It will also store all

the files in application cache. Local database contains all the metadata eg. hit count, date and time, size of the file. This information is used to save the right candidates into blobstore and later to prune the file metadata and application cache (that is old) with less number of hit counts. Total time to store and retrieve metadata on the local database is much less than the total time to make HEAD requests

5.3.6 Improvement required for Local Database Approach

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 assembling of package (it takes almost 50 percent of time). To further improve the speed of this approach and to improve the user experience while uploading bits, we suggest an approach that would reduce the time and space complexity of the whole process by a reasonable amount.

5.4 Tree Based Approach

Tree Based approach is the result of discussion that my team had after the first presentation of my thesis. Assemble phase of this approach is suggested by Marc Schunk. As discussed in the chapter 4, the TAPER approach described in section4.7, 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 fingerprints. In its second phase, it matches these SHA's with the fingerprints on the server using a Bloom Filter4.6. In its third phase, it calculates chunks of data that are changed in each blob using a content-based similarity detection technique described in 4.7.

In Tree Based approach, it used the first phase of the four phases of TAPER approach. We are not matching the fingerprints using a bloom's filter as there are chances of false positives as described in [Bur]. Since this approach is used to upload bits to a PaaS cloud, false positives are not acceptable. A false positive occurs when a blob is not present on the Blobstore but the bloom filter approach could return that it is already present on the blobstore. Therefore, bloom's filter can only be used to find out which directories are similar. The third phase is chunking blob phase that is also not considered due to the bandwidth assumption as stated in 5.1.1. This would add to the time and computation overhead on the Bits-Service. In this approach fingerprints are created in a Hierarchical

tree structure. Chances of fingerprint collision are more in SHA. But, in this approach fingerprints of all the blobs and directories in same directory are merged to create a single directory fingerprint as described in Section 3.3.

While matching fingerprints, first the directory fingerprints are matched and if it is matched, blobs inside are not matched. Chances of fingerprint collisions are minimized as directory fingerprints are combination of all the blob fingerprints. This approach reduces the time complexity and space complexity drastically as compared to the above stated approaches 5.1, 5.3. Following are the steps of Tree based approach describing CF Client 5.5 and Bits-Service 5.6.

Algorithm 5.5 CF Client

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

unknowfingerprint[] ← PUT /match fingerprint[]
for each sha ∈ Unknownfingerprint[] do
    folder ← blob corresponding to sha
end for
zipblob ← zip(folder)

PUT /bits zipblob
```

Algorithm 5.6 Bits-Service

```
/match do
unknownfingerprint[] ← SHA not known
download GUID.json from blob store
Bits-Service receives json tree structure of fingerprints from CF Client
for each shaFromCFClient do
    for each shaFromGuid.json do
        Compare recursively
        if fingerprint.exist = false then
            unknownfingerprint[] ← sha
        end if
    end for
end for
return unknownfingerprint[]
end /match

/bits do
folder ← unzip(zipblob)
for each blob ∈ Folder do
    blobfingerprint ← fingerprintofblob
    PUT blobfingerprint, blobcontent toremotedatabase
end for
end /bits
```

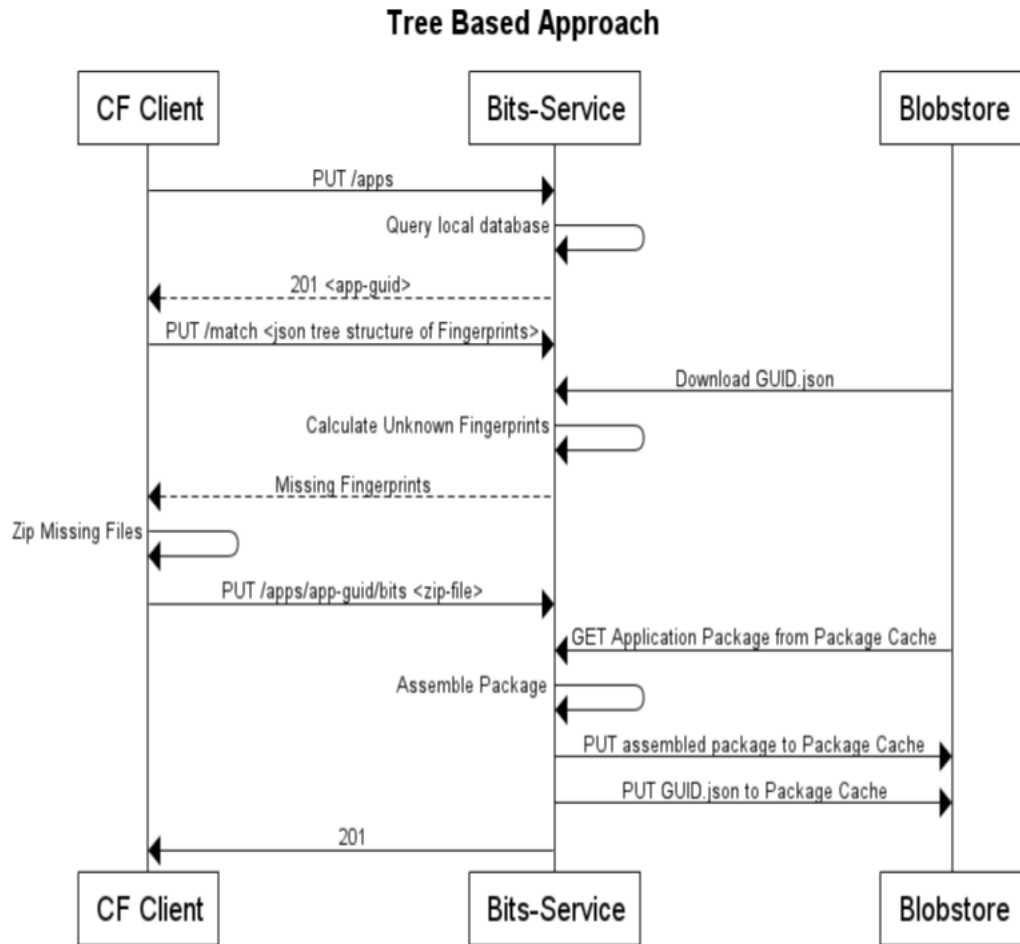


Figure 5.5: Tree Based Approach

5.4.1 The Approach

In figure 5.5, the setup remains the same as of the previous approaches. In this approach, we do not save the application metadata in the local database but we store application name and other details in local database. This approach works in following steps:

1. In the first PUT request in figure 5.5 CF Client sends application name to the Bits-Service and the Bits-Service creates a new GUID for the CF Client and sends the GUID with status 201. If the local database in Bits-Service already has the GUID it sends the GUID with status 200.

2. In second step the CF Client calculates SHA of all the blobs present in application directory and sends this as a JSON with other details like organization, GUID to the Bits-Service.

a) If it is the first upload of the application bits, Bits-Service saves list of the fingerprint in a JSON file and saves that file with GUID as its name to the blobstore and return the list to CF Client as a list of unknown fingerprint,

b) If it is not the first time the application is uploaded to the Bits-Service, it downloads the JSON file from the blob store and compares the two JSONs. Bits-Service then returns a list of all the unmatched blobs to CF Client.

3. CF Client receives the list of unknown fingerprints and creates a folder with all the blobs that corresponds to those fingerprints after adding all the unknown blobs to a folder, CF Client creates a zip file of the folder and sends it to Bits-Service. Along with the zip file it sends a file with JSON tree structure of all the blobs.
4. Bits-Service receives the zipped file and JSON file from the CF Client.

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

   b) If it is not the first time the application is pushed, Bits-Service downloads the zip file(named as GUID received from CF Client) from blobstore. Unzips this file and unzips the file received from CF Client. For each file present in the the zip file from CF Client it saves the file in the package received from blobstore. After assembling the entire package, Bits-Service zips it again and sends it to blob store.
5. Finally, when all the blobs are stored on the blobstore, Bits-Service sends a status code of 201 to CF Client and this completes the entire process.

5.4.2 Implementation Detail of Tree based Approach

For implementing this approach we used the same implementation setup as used in previous approach. We used Softlayer as our remote database for storing application files and application packages. For creating the tree structure we used a library called "find". This library finds out all the folders in top down fashion. We used this structure to find out the last folder and then we start creating a tree from inside to outside.

The diagram 5.5 might look similar to the diagram of Current Approach 5.1, but this differs in the way CF Client make POST request with list of fingerprints to Bits-Service. Before getting into details of this approach let us have a look on the directory tree structure created by **createTree()** function. Following is a sample directory structure 5.6 of an application directory. Here the "Root" represents application directory, "DirA" and "DirB" represents directories directly under application directory. "DirC" and blob1 are under DirA and so on.

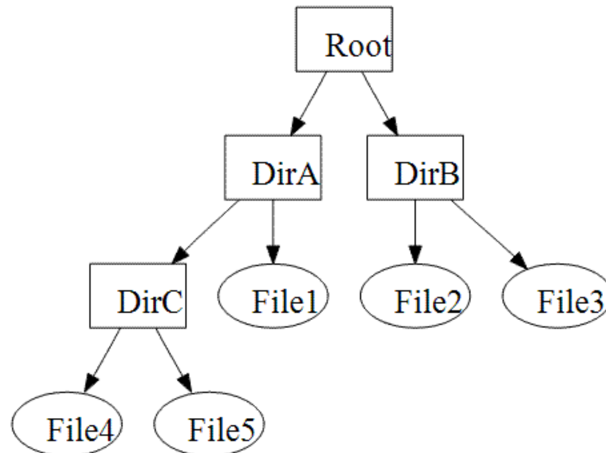


Figure 5.6: Directory Structure [Ash]

CF Client goes through each blob in the application directory and calculates their SHA and path. For all blobs that have same paths, their fingerprints 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 fingerprintes in a tree structure.

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

The fingerprints calculated for blobs and directories are a,b,c,d,e,f,g,h and i. Following is the tree structure of fingerprint in JSON format calculated by CF 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"]}
}
```

5.4.3 Space and Time Complexity

Time complexity at CF Client can be calculated by the operations performed by CF Client. Following are the operations performed by CF 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 CF Client goes through each blob and calculates SHA and path of each blob. It then goes through the path of all blobs and creates a joint SHA for blobs with same path.It takes O(nk) where n is the number of blobs and k is the number of directories.
3. **Calculates zip blob:** After receiving list of unknown blobs, CF Client puts all unknown blob into a zip blob. This operation takes O(n) time.

So, the total time taken by CF Client is: **O(1)+O(nk)+O(n) = O(nk)**
Total Space complexity on CF 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 CF 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 CF Client. This operation take O(n) time where n is the number of applications on Bits-Service.
2. **Calculate unkown fingerprints :** After receiving a list of directory and blob fingerprints from blobstore, Bits-Service makes comparison with SHA sent by CF Client. This takes O(log n) time, where n is number of fingerprints.
3. **Saves changed/new blobs on remote and local cache:** When Bits-Service receives changed/new blobs from CF Client, it makes a POST request for each of them and saves them on remote and local cache. This operation takes O(k) time, where n is number of changed/new blobs.

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

5.4.4 Discussion

This approach reduces the number of comparisons to O(log n) as compared to other approaches discussed so far. This approach does not need an extra component (like a local database). But it takes a lot of time in downloading application package file from package cache during assemble phase. This approach is able to solve all of the issues of Current service 5.1.

In Figure 5.7 we can see the distribution of time in different phases.

1. Time T1 represents the SHA calculation phase. This phase is the total of the time taken by client to read all the files and to calculate a tree structure of their SHA.
2. Time T2 represents the match phase . This phase is the total of the time taken by client to send the SHA to the server and receive back the changed SHAs.
3. Time T3 represents the total time taken by client to zip the changed files and upload them to the server.
4. Time T4 represents the total time taken by server to assemble the package. In this phase the server receives the changes from client and downloads the package from the blobstore.
5. Time T5 represents the total time to upload the package from server to the blob store.

The pie chart 5.7, shows the total time taken by different phases. As we can see that total time taken to compare the fingerprints is lesser than other approaches. The total time in assemble phase still remains high because it downloads the entire application from the blobstore and then assembles the package. But as compared to the previous approach, this time would remain constant irrespective of number of changes. In the previous approaches if the number of changes increases the total time of assemble would increase as well.

5.4.5 Significance of Tree Based Approach

This approach is a major improvement on Current and local DB approach. It does not need a local or remote database for storing files and metadata. Hence, there is a component less to be maintained. It creates a JSON tree structure and stores that on a file. It saves the file in one PUT request to the remote blobstore. This file can be downloaded again using a PUT request hence, all the HEAD requests to remote blobstore and all the read request from local database are saved. It does not save any individual file in the application cache but only the application package. It reuses that package for

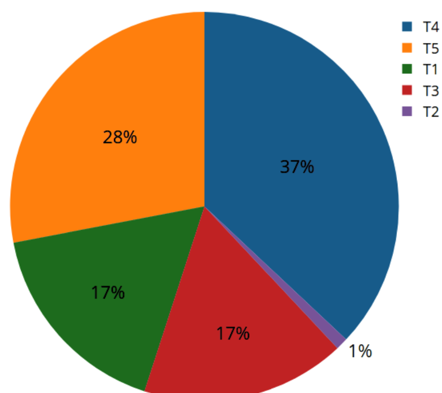


Figure 5.7: Tree Based Approach Pie Chart

assembling every time. Hence, it saves the storage on blobstore. This is faster than other approaches as it does not save any metadata or files on local or remote blobstore.

5.5 Combined Approach

In this approach we combine the two approaches: the Local Database approach, which suggests to have a local cache (with fingerprints of all the blobs on the Blobstore) and the Tree Based Approach, which suggests to have a hierarchical tree structure of application directory. Benefit of Local Database 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 is O(n) as the local cache have to be queried for all the blobs in application. Benefit of Tree Based approach is that the time complexity of comparison of fingerprints is O(log n). Following is the CF Client 5.7 and Bits-Service 5.8 approaches of combined approach that takes benefits of both the approaches.

Algorithm 5.7 CF Client

$fingerprint[] \leftarrow$ SHA of each blob and directory
for each $directory \in \mathcal{A}pplication$ **do**
  $fingerprint[] \leftarrow createTree()$
end for

$unknowfingerprint[] \leftarrow$
PUT /match *fingerprint[]*
for each $sha \in \mathcal{U}nknownfingerprint[]$ **do**
  $folder \leftarrow$ blob corresponding to sha
end for
$zipblob \leftarrow$ *zip(folder)*

PUT /bits *zipblob*

Algorithm 5.8 Bits-Service

/match do
$unknownfingerprint[] \leftarrow$ SHA not k$nownbyBits - Service$
Bits-Service receives *List of fingerprints*
for each $sha \in \mathcal{L}ist$ **do**
  queries local cache
  SELECT * FROM table
  WHERE fingerprint= ***sha***
  if $fingerprint.exist = false$ **then**
    $unknownfingerprint[] \leftarrow sha$
  end if
end for
return *unknownfingerprint[]*
end /match

/bits do
$folder \leftarrow$ *unzip(zipblob)*
for each $blob \in \mathcal{F}older$ **do**
  $blobfingerprint \leftarrow$ *fingerprint* $textitofblob$
  PUT *blobfingerprint, blobcontent* t$oremotedatabase$
end for
end /bits

5.5.1 The Approach

Figure 5.8 is the sequence diagram for the combined approach. This figure looks similar to Local database sequence diagram 5.3 because the Bits-Service has a local cache that stores all the fingerprints. This approach works in following steps:

1. In the first PUT request CF Client sends application name to the Bits-Service and the Bits-Service creates a new GUID for the CF 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 CF Client calculates SHA of all the blobs present in application directory in hierarchical tree structure as explained in 5.4.2 and sends this as

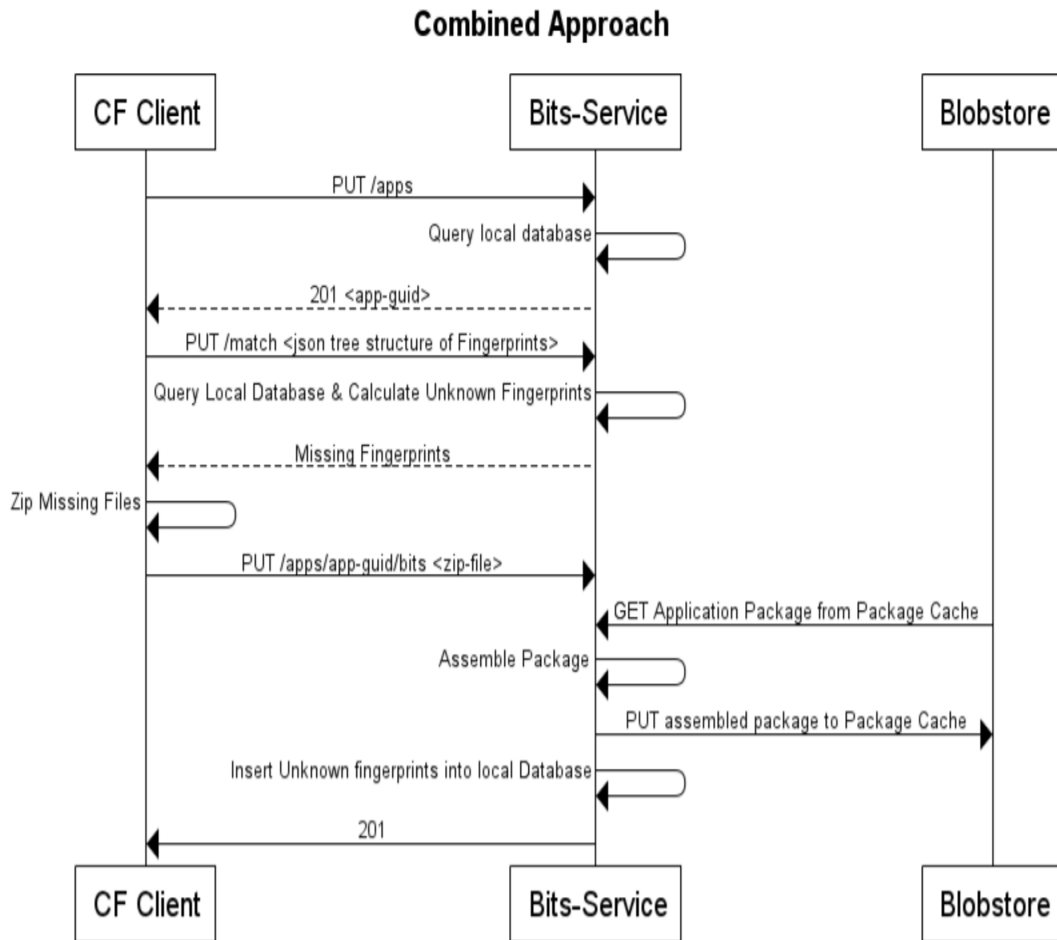


Figure 5.8: Combined Approach

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 fingerprints sent by CF Client to its local cache and return the same list to CF Client as the unknown blobs.

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

3. For all the blob's fingerprints that didn't match the fingerprints in Bits-Service database are put into a list and that list is sent as a list of Unknown blob fingerprints to CF Client.

4. CF Client creates a zip of all the blobs and sends it to Bits-Service along with the JSON structure.

5. Bits-Service receives the zipped file and JSON from the CF Client.

   a) if it is the first time the application is pushed, the each key of JSON is saved in the local cache and zipped file is saved as a application package to the blobstore.

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

6. Finally, when all the blobs are stored on the blobstore, Bits-Service sends a status code of 201 to CF Client and this completes the entire process.

5.5.2 Implementation

For implementing this approach we used Sinatra for server side programming and Ruby rest CF Client for CF Client side programming. We installed different gems eg. json for the data exchange between CF Client and server, File utils for using files on the local machine, digest for using SHA, rest-CF Client for using rest calls, pathname for creating paths, ruby zip for creating zip of application files, aws-sdk for connecting to remote blobstore and sequel for using SQLite database. In addition we used a local database on server side which stores some information about the applications. For this purpose we used a sqlite database that is a file based database for storing the basic information. We used softlayer as our remote database for storing application files and application packages. For creating the tree structure we used a library called "find". This library finds out all the folders in top down fashion. We used this structure to find out the last folder and then we start creating a tree from inside to outside.

5.5.3 Space and Time Complexity

Time complexity at CF Client 5.7 can be calculated by the operations performed by CF Client. Following are the operations performed by CF 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 CF 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, CF Client puts all unknown blob into a zip blob. This operation takes O(n) time.

So, the total time taken by CF Client is: **$O(1)+O(n^2)+O(n) = O(n^2)$**
Total Space complexity on CF 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 5.8:

1. **Finds the GUID corresponding to application name:** When Bits-Service receives application name from the CF 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 CF Client. This operation take O(n) time where n is the number of applications on Bits-Service.

2. **Query local cache:** When the Bits-Service receives list of fingerprints from CF Client, it queries local cache for all blobs fingerprints.This operation take O(log n) time, where n is number of fingerprints sent by CF Client.

3. **Saves changed/new blobs on remote and local cache:** When Bits-Service receives changed/new blobs from CF 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 CF 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.

5.5.4 Discussion

In this approach we saw a lot of improvements in time complexity. The time complexity reduced to O(log n) where n is the number of application blobs. This approach does not impose any rule of not storing blobs of size less than 64KB and hence, only the changed/new blobs are sent from CF Client to Bits-Service. Since this approach calculates directory fingerprint and compares directory fingerprint before comparing blob fingerprint, chances of fingerprint collision is reduced. But by keeping a local cache at

the Bits-Service it increases the space complexity at Bits-Service side to O(nk) where n is the number of blobs and directory in application directory and k is the number of applications on the Bits-Service. This would increase with increase in applications on the Bits-Service.

The Figure 5.9 represents the pie chart for the distribution of time among different phases of file synchronization using the combined approach.

1. Time T1 represents the SHA calculation phase. This phase is the total of the time taken by client to read all the files and to calculate a tree structure of their SHA.
2. Time T2 represents the match phase . This phase is the total of the time taken by client to send the SHA to the server and receive back the changed SHAs.
3. Time T3 represents the total time taken by client to zip the changed files and upload them to the server.
4. Time T4 represents the total time taken by server to assemble the package. In this phase the server receives the changes from client and downloads the package from the blobstore.
5. Time T5 represents the total time to upload the package from server to the blob store.

In figure 5.9 we can see that total time of match phase is very less. The total time of all the phases is same as the tree based approach.

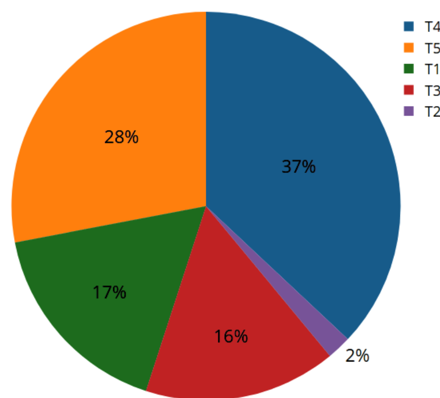


Figure 5.9: Combined Approach Pie Chart

5.5.5 Significance of Combined Approach

This approach is a combination of local database approach and tree based approach. It calculates fingerprints and stores it in the form of tree structure. But instead of storing it in a file it stores it in a local database. Along with fingerprints it stores other metadata information like hit count, size, data and time of creation. It also requires an extra handle to prune the database using the metadata. This file can be downloaded in a PUT request (all the HEAD requests to remote blobstore). It does not save any file on blobstore but only the application package. It reuses the package previously stored for assembling every time. Hence, it saves the storage on blobstore.

5.5.6 Improvement required for Combined Approach

Since this approach combined the approach of both Local DB and Tree Based, it has the benefits of both. There is a trade off between the space complexity of combined approach and a little bandwidth of downloading JSON file from Tree Based approach which needs to be considered while considering an approach for production purposes.

5.6 Comparison and Summary

As discussed, we consider three phases of file synchronization approach: Calculation of SHA, match and assemble phase. Table 5.1 represents the key difference between all the four approaches based on the three phases of file synchronization. Current Approach and Local database approach uses a list data structure for the first phase and Tree based and combined approach uses Tree based approach. Tree based approach improves the performance of approach as they have O(log n) complexity while others have a complexity of O(n). Tree based approach stores its metadata in a file and stores it in the remote database along with the application package. Other approaches need an extra component to be maintained to save the metadata, for eg. Local database approach and combined approach needs a local database to store the metadata, while Current needs a remote blobstore to store the files. These extra components would grow with the number of applications in the cloud. To maintain these components an extra handler is needed that would delete some data after a fixed amount of time. But Tree based approach stores the metadata in a file and then it keeps that file along with application package into the blobstore. There is no need of any extra component in tree based approach. Hence it has a simpler design that needs very less maintenance.

	Current API (5.1)	Local DB Approach (5.3)	Tree Based approach (5.5)	Combined approach (5.8)
fingerprint Calculation Phase	List	List	Tree	Tree
Match Phase	HEAD request	First Push: all writes, Consecutive Push: all reads	Compare Guid.json from CF Client and server	First Push: all writes, Consecutive Push: some reads
Assemble Package Phase	Download files from application cache	Download files from application cache	Download zip from package cache	Download zip from package cache
Complexity of fingerprint Calculation Phase	O(n)	O(n)	O(nk)	O(nk)
Complexity of Match Phase	O(n)	O(n)	O(log n)	O(log n)
Complexity of Assembling package Phase	O(n)	O(n)	O(n)	O(n)

Table 5.1: Differences between approaches

Tree based and Combined approaches are faster than others as they need very less time to match the fingerprints. We can see this from the match phase complexity in 5.1. Tree based and combined approach has a complexity of O(log n) which is much lesser than other approaches. Time taken by the former approaches for tree construction is almost similar to the time taken by others to construct a list structure. The complexity of tree construction is O(nk) where n is the number of files and k is the number of directories. Since number of directories is very less than number of files we consider it to be 1 i.e. k ««« n, O(nk)   O(n). In the next chapter we will see the evaluation of these approaches against different types of applications. We will be discussing different application characteristics and their effects on the performance of these different approaches.

6 Implementation and Evaluation

We discussed the four approaches in chapter 5. Each of them are different and optimize the performance by reducing the total time of file synchronization and the total resource usage.In this chapter, we start with the discussion of factors that are used to measure the performance of the file synchronization approaches. Then in following section, we discuss some major factors that affects the performance of file synchronization approaches. In the following section, we discuss some of the test applications that we have used for testing all the four approaches. In next few sections, we will discuss the effect of all the four approaches on first and subsequent push, followed by the analysis of the total time of first and subsequent push by discussing the time taken by each approach in three phases of file synchronization: fingerprint calculation phase, match phase and assemble package phase.

6.1 Performance

As we have discussed in chapter 3, performance of an approach is measured by the results of following parameters:

1. **Total time of synchronization:** It is the total time taken by the complete process of file synchronization. It starts with the time when the CF Client requests a "**cf push**" and ends when the changed application becomes available for the next CF Client in the Blobstore. It includes the time taken by following phases of resource matching depending upon the approach.
   a) Time taken by CF Client/Bits-Service to calculate fingerprints
   b) Time taken by CF Client/Bits-Service to send calculated fingerprints to Bits-Service/CF Client
   c) Time taken by CF Client/Bits-Service to calculate changes
   d) Time taken by CF Client to compress the changed files
   e) Time taken by CF Client to upload zip of changed files to Bits-Service

f) Time taken by Bits-Service to download the previous version from blobstore

g) Time taken by Bits-Service to assemble the application package

h) Time taken by Bits-Service to upload the changed package to blobstore

2. **Bandwidth usage:** It is the amount of bandwidth used during the process of file synchronization. We are assuming a reasonable amount of available bandwidth. Therefore, we make sure that CPU computation of any approach does not become a bottleneck for the performance. Also, to use optimal amount of bandwidth, we are not uploading any extra bit that is already known by Bits-Service.

3. **Usage of computation resources at server(Bits-Service):** CF Clients have varied amounts of computation power but a fixed amount of computation resources are available at the server. Hence, the approaches must use minimum amount of CPU at the Bits-Service.

4. **Total number of messages exchanged between CF Client, Bits-Service and remote backend:** This is one of the major factor deciding the performance of an approach. Since all the three components: CF Client, Bits-Service and remote backend are connected to each other via network. All the approaches must make sure that number of HTTP requests between each component are minimal.

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

6.1.1 Factors Affecting Performance:

This section describe the factors that affect the performance of all the approaches. Following are the factors:

1. **Application Folder Characteristics:** This is a major factor that affects the performance of any approach. We will see in next few sections the effect of this factor directly on the performance. This factor is a combination of following sub factors:

   a) **Number of files in an application:** It affects the time taken by approach to calculate fingerprints. A prominent example of this comes from evaluation of the test applications we chose in 6.1, if the number of files is more than 60,000 it takes around 20 second to calculate all the fingerprints else if it is around 1000, it takes less than 1 second to calculate fingerprints.

b) **Size of application:** It affects the total time of first push, as the whole application has to be pushed from the CF Client to the server. It also affects the time taken by the server to download/upload the package from/to package Blobstore.

c) **Number of Folders:** While calculating JSON tree for Tree Based and Combined approach, the metadata of directories must be present in the memory. It is directly proportional to the memory used by CF Client while calculating fingerprints.

d) **Number of nested folders:** It affects the fingerprint calculation phase and match phase of Tree based and Combined approach. If the application folder is heavily nested it increases the number of calculations which in turn increase the computation and time of match phase at the Bits-Service and vice versa.

e) **Size of changed file:** It affects the time taken by the CF Client to send changed files and also affects the time to assemble the package. It also affects the time taken for uploading changes from CF client to Bits-Service.

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

These sample applications are just example applications that are considered for evaluation of all the four approaches. These are test numbers and are generalized for all applications with similar characteristics.

2. **Network Speed:** It is the upload and download speed of the network. It depends on the upload speed of CF Client, upload and download speeds of the server hosting Bits-Service. It affects the overall time of file synchronization as all the three components(CF Client, server and blobstore) communicate via network.

3. **Cache:** It is the local database that stores the metadata of the application package and the blobstore that saves actual application package. It affects the phase of calculating the difference, if it is a remote cache(a remote Blobstore), a lot of time and connection requests will be required retrieving/writing metadata from/to remote cache. If it is a local cache, retrieving/writing metadata becomes faster.

4. **Hash calculation:** The calculation of a list of fingerprints or a tree of fingerprints takes more or less the same amount of time. The approach used to calculate the hash affects the total time.

Above stated factors affect the performance of resource matching approach for uploading bits to PaaS cloud. As we have seen in the above factors that the type of input is a major deciding factor for measuring the performance of an approach. We have taken a few example applications with a combination of factors to evaluate the approaches.

	Size of Application(MB)	Number of Files	Nested	Example
Large Number of small and medium sized files	683	57,000	Medium	Linux Kernel
Small Number of medium and large files	143.8	3,704	Low	Emacs
Large Number of small files	74.4	22,530	High	Zipkin-web
Small Number of large files	50	153	Less	Tex
Large number of small files	24	6,632	Medium	Java-Unwritable-Dir
Large Number of very small files	8.6	1082	Medium-High	Portal Mail Master Nodejs

Table 6.1: Type of Applications

Table 6.1 describes the types and combinations of application files. The first application that we select is a linux kernel. The reason we select it is that it has a large number of files and around 80 percent of files are less than 64kB in size. The second application that we select is Emacs. The reason we select it is that it has a great combination of small, medium and large number of files. The third application we select is Zipkin-web. The reason we select this application is that it has a huge number of files and it has a deep nested level of directories. The fourth application that we select is a Latex project. It has large number of files greater than 64 KB size and it does not have a nested directory structure. The fifth application that we select is a Java application. The reason is that it has a huge number of very small files and it has medium level of nesting. The last application that we select is a Node.js application. It has medium-high nesting of directories and small-medium sized files. These applications covers all the characteristics we want to measure. We will see the results of these applications in next few sections.

In the next table that is 6.2, we describe the basic behavior of applications described in 6.1. Basic behavior includes the time taken by application to zip, upload and download etc.

6.2 Implementation

In Figure 6.1, we present the components involved in the implementation of the client and the server. Client uses local file system for the applications. It uses the digest component for calculating the SHA of each file. To find all the paths in an application, it uses a component called find. For making REST calls it uses the Ruby REST Client. For Zipping the application package it uses Ruby Zip. Similarly, the server uses almost the same components. It uses the Blobstore to store the application files and Local Data base for storing the file metadata.

As shown in Figure 6.2, the server offers three service endpoints to the client: apps, match and bits. The diagram represents the two boxes attached to each of the three service endpoints. The left box signifies the input to each service. This input is provided by the client. After receiving the input, the server processes it and outputs the results shown by the right boxes(of each service).

We can see the direct implementation of each of the three service endpoints in Chapter 5. Figure 6.2 is a generic representation of the implementation of all the approaches. As we can see in Figure 5.1, the first request from the client to the server is the request to /apps endpoint. It takes the name and size of application as input from client and sends it to server. The server queries the local database with the application name and returns back GUID to the client. Similarly, the second request from client to server takes as input the size, GUID, List of SHAs, hit count as input from client. Server compares it with the metadata on the components attached to it(local database and blobstore) and returns unmatched SHAs. In the third request, the client sends a zip of the unmatched files to the server and server updates itself and the attached components and returns with a status code of 201.

6.3 Test Setup

In this section we discuss the setup that is used to evaluate all four approaches. Figure 6.3 represents the test setup. Cloud foundry CF Client is the local machine that is connected to the local university network. The download bandwidth in university network is 30Mbps. Since originally CF Client and the server hosting Bits-Services are situated at a distance from each other, we simulated a bandwidth of 10Mbps between them by using a bandwidth limiter. For this purpose we used Toxiproxy [Sim] that is used to simulate poor network condition. Server and Blobstore are always situated closed to each other, hence we used the download bandwidth of 30 Mbps. Bits-Service is also connected to a local database that stores some metadata about application files.

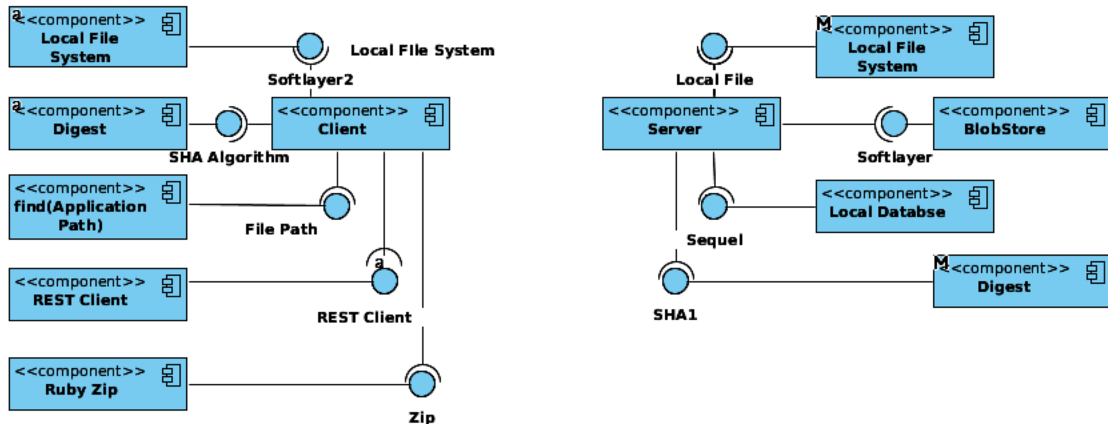


Figure 6.1: Component Diagram

For storing the application files and packages we have used a S3-compatible back end in Softlayer, connected via an internal 1 Gbps network.

6.3.1 Application Characteristics

As we have seen in table 6.1, there are different types of applications, different kinds of folder structure, different sizes, different number of files etc. We have taken into consideration each factor described in 6.1.1 (application folder characteristics). In the table, the first column describes the type of application, second column describes the size of application in MB, third column describes the number of files in the application, fourth column describes the level of nesting of folders inside application folder and the last column is the example application that we used to push into Cloud Foundry. In following points we describe the meaning of the terms that we used in the table 6.1 .

1. Small sized files: Files of size 64KB or less.
2. Medium sized files: Files of size 64KB - 500KB.
3. Large sized files: Files of size more than 500KB.
4. Low nesting: Files are present either directly inside the application folder(level 0) or level 1 or level 2.
5. Medium nesting: Files are lying at level 3-level 5
6. High nesting: Files are lying at level 5-level 7.

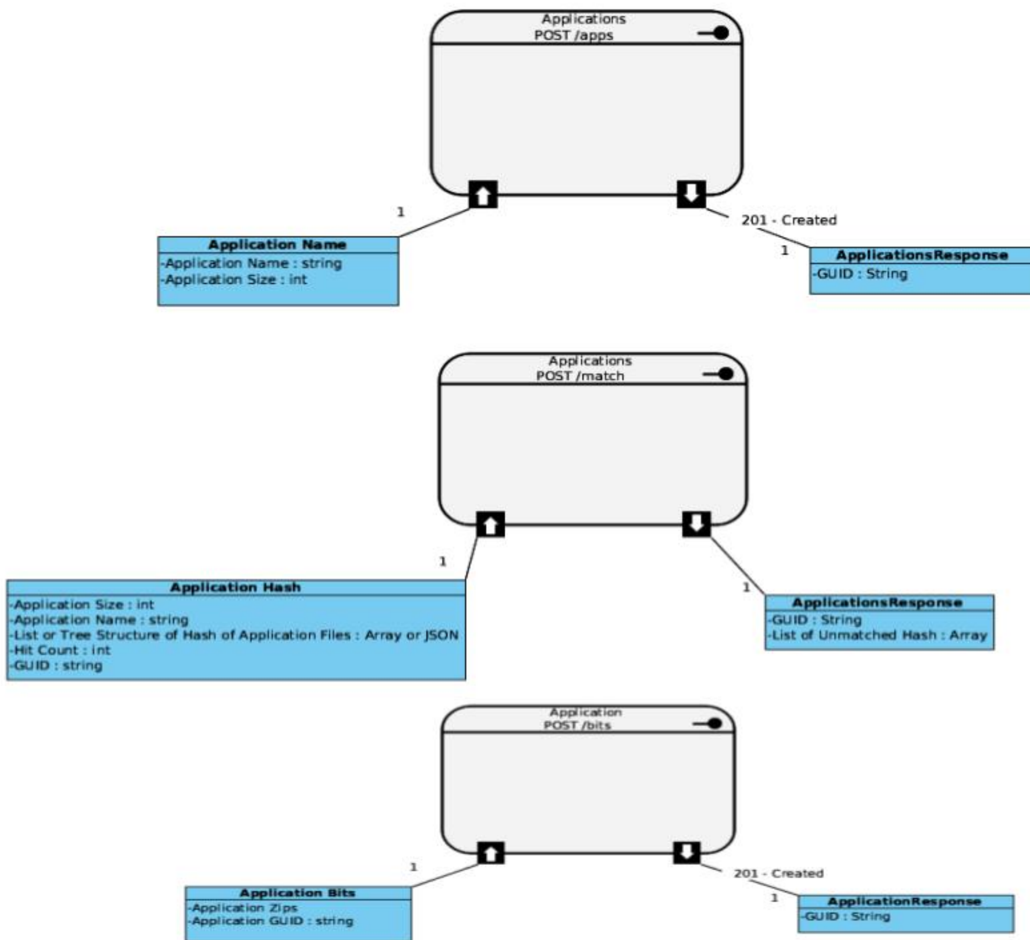


Figure 6.2: Class Diagram of Services on the Server

In table 6.2 we describe the basic information about applications. First column describes the type of application, second column describes the size of application in MB, third column describes the time taken by Bits-Service to download application package from remote Blobstore, fourth column describes the time taken by Bits-Service to upload application package to Blobstore and the last column describes the time taken by CF Client to zip the application. This table describes the behavior of network speed on different kinds of applications.

Before we move on further with the evaluation results of different approaches discussed in chapter 5, following are parameters of our development and test environments.

- Memory: 15.3 GiB

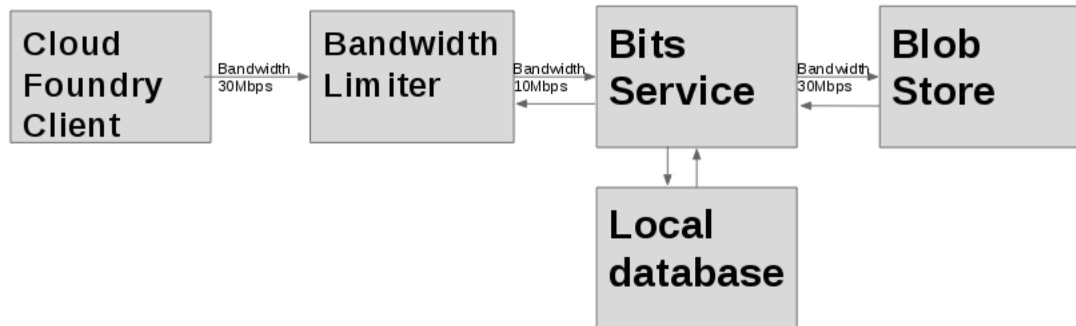


Figure 6.3: Test Setup

- Processor: Intel® Core™ i7-3740QM CPU @ 2.70GHz × 8
- OS-Type: 64 Bit, Red Hat Enterprise Linux
- Upload Speed between CF Client and Bits-Service: 10 Mbps
- Upload Speed between Bits-Service and Blobstore: 10 Mbps
- Download Speed between Bits-Service and Blobstore: 30 Mbps
- Blobstore: Softlayer

6.4 First Push

6.4.1 Description

First push is the first time an application is pushed into Cloud Foundry. Time taken for first push differs for different approaches. First push is a combination of operations: CF Client calculates fingerprints, CF Client creates a zip of all files, Server hosting Bits-Service uploads the zip to the Blobstore, the server uploads fingerprints and content of each file greater that 64KB to Blobstore or metadata of all the files and folder in local cache or stores the metadata in a file on blobstore(depending on the approach).

- In the current approach, all the files that are greater than 64KB in size are pushed to application cache with their name as their fingerprint. Then the application folder is assembled, zipped and pushed to Blobstore.

	Size(MB) (Unzipped)	Size(Zipped) (MB)	Download Time (Blobstore -> Bits-Service) (Seconds)	Upload Time(Bits-Service -> BlobStore) (Seconds)	Time Taken to zip(Seconds)
Large number of small and medium files 6.1	683	179	100	28	161
Small Number of medium and large files 6.1	143.8	41	16	5	9
Large Number of small files 6.1	74.4	34	20	2	39
Small number of large files 6.1	50	30	14	3	2
Large Number of small files 6.1	24	11	5	1	8
Small Number of Large files 6.1	11	4	2	1	1
Large Number of very small files 6.1	9	5	1	2	1

Table 6.2: Basic information about application files

- In the local DB approach, instead of saving entire files in remote application cache, file fingerprints are saved in a local cache with their metadata and then application package is uploaded to Blobstore.
- In Tree based approach, guid.json and application folder are both uploaded to Blobstore respectively.
- In combined approach, application folder is uploaded to Blobstore and the fingerprints of its files and folders are saved into the local cache.

In table 6.3, we display the results of pushing different types of application described in 6.1 first time from CF Client to Bits-Service using different approaches.

	Current Approach (Seconds)	Local Database Approach (Seconds)	Tree based Approach (Seconds)	Combined Approach (Seconds)
Large number of small and medium files	>1000	>1000	120	>1000
Small Number of medium and large files	900	650	50	120
Large Number of small files	230	212	89	700
Small number of large files	214	146	20	23
Large Number of small files	109	56	25	45
Large Number of very small files	44	30	7	20

Table 6.3: First Push of application files

6.4.2 Observation:

Graph 6.4 represents the time taken for the first push by each application when pushed with different approaches described in 5. The blue line represents the results of Current Approach, Yellow line represents results of Local database approach, Green line represents results of Tree based approach and Red Line represents the result of Combined approach.

We can observe from the graph that minimum amount of time is taken by Tree based approach. The reason for this is, in this approach we do not save metadata of each file individually, instead we save the entire metadata in a file and push that file along with application zip to Blobstore. This is very similar to just pushing the entire application without any file synchronization. There is a sudden shoot in the graph of combined approach. The reason for this is the time taken by combined approach to update the metadata of all the application files in local database. In Current approach and Local Database approach, we perform many PUT requests on the first push to assemble the application package. In Local Db and Combined approach, we perform a lot of write operations to the local cache. We consider a file based system for the local cache hence it takes longer to store metadata on first push. In the next section, we will discuss the results of time taken for subsequent push.

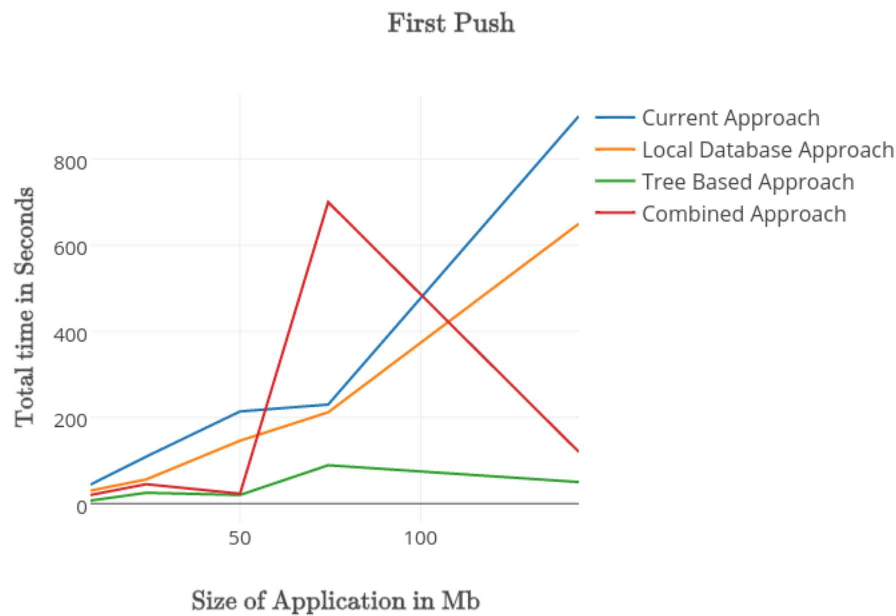


Figure 6.4: First Push

6.5 Subsequent Push

6.5.1 Description

Subsequent push are made to the application when the metadata is already there in the cache (local or remote) and application in Package Cache. The actual file matching starts from the subsequent push. Since in a PaaS cloud, server and Blobstore are placed close to each other but the CF Client is at a distance from them, we require an approach that minimizes the amount of data transfer between the CF Client and the server. In all the four approaches described in chapter 5 we minimize this data transfer between the CF Client and the server. Following are the summarized points describing how suggested approaches differ in second push.

- As described in first push, in Current Approach the CF Client saves fingerprints of all the files into the application cache. For the second push the server(after receiving a list of fingerprints from CF Client) makes HEAD requests to the Blobstore for each fingerprint. It sends back the fingerprints that were found at the Blobstore. CF Client zips the files unknown by Bits-Service and sends it to Bits-Service. Bits-Service downloads all the files that did not change and are greater than 64 KB from App Cache and assembles them with the changes sent by CF Client. After assembling, Bits-Service uploads the package again.

- As we have seen in Local Database approach, we save all the fingerprints with their meta data into a local cache. Therefore, on the second push, the Bits-Service(after receiving a list of fingerprints from CF Client) queries its local cache for the fingerprints and sends back the fingerprints that are unknown to it, to the CF Client. CF Client zips the files unknown by Bits-Service and sends them to Bits-Service. Bits-Service downloads all the files that did not change and are greater than 64 KB from App Cache and assembles them with the changes sent by CF Client. After assembling Bits-Service uploads the package to the Blobstore.
- Tree based approach saves the file with tree structure of fingerprints at the Blobstore. During second push, server downloads this file from Blobstore and compares it with the fingerprints received from CF Client. It then returns the fingerprints that are not present in the file downloaded from the Blobstore. The CF Client zips the files unknown by Bits-Service and sends it to the Bits-Service. The Bits-Service downloads the previous application package and assembles it with the changes sent by CF Client. After assembling Bits-Service uploads the package again.
- Combined approach saves the directory tree structure of fingerprint's in a local cache. It matches the fingerprint tree structure received from CF Client with its local cache and sends the unknown fingerprints to CF Client. CF Client zips the files unknown by Bits-Service and sends it to Bits-Service. Bits-Service downloads the previous application package and assembles it with the changes sent by CF Client. After assembling Bits-Service uploads the package again to blobstore.

Another difference was in calculation of fingerprints of files in the form of list and tree structure. But there is not much difference in performance of tree or list calculation. Table 6.4 displays the total times take to push 15 changed files from CF Client to Blobstore.

6.5.2 Observation

Graph 6.5 represents the time taken by applications during second and subsequent push. As we can see form figure 6.5 the total time taken for subsequent push by tree based approach is the least. The time taken by combined and tree based approach are similar. They overlap each other due to the scale of the graph. But total time for second push is least for the tree based approach. This is because of following reasons:

1. It optimally utilizes the slow bandwidth between CF Client and Bits-Service. Tree based approach reduces the total amount of data transfer between CF Client and Bits-Service. Since the bandwidth between the CF Client and the Server hosting

	Current Approach (Seconds)	Local Database Approach (Seconds)	Tree Based Approach (Seconds)	Combined Approach (Seconds)
Large number of small and medium files	1000	1000	115	118
Small Number of medium and large files	637	635	106	55
Large Number of small files	230	170	78	90
Small number of large files	214	36	20	23
Large Number of small files	109	57	25	25
Large Number of very small files	44	27	7	7

Table 6.4: Second Push of application files

Bits Service is low and total data transferred is less (as compared to The Current and Local Database Approach), total time of this upload is reduced as well.

2. It optimally utilizes the fast bandwidth between Bits-Service and Blobstore by downloading the entire package from blobstore and uploading the package. It saves all the time used by other approaches to update the metadata in cache.

6.5.3 Match Time

Time taken during subsequent push is the total time taken by changed application at CF Client to reach Blobstore and become available for the next CF Client. During subsequent push, the approach of calculation of changes in the application(at CF Client) differs in all the four different approach.

This section describes the time taken by all the approaches (discussed in 5) to calculate the changes in application. Differences between the approaches has been described in section 6.5.1. Table 6.5 displays the total time taken by server to calculate changes in different approaches.

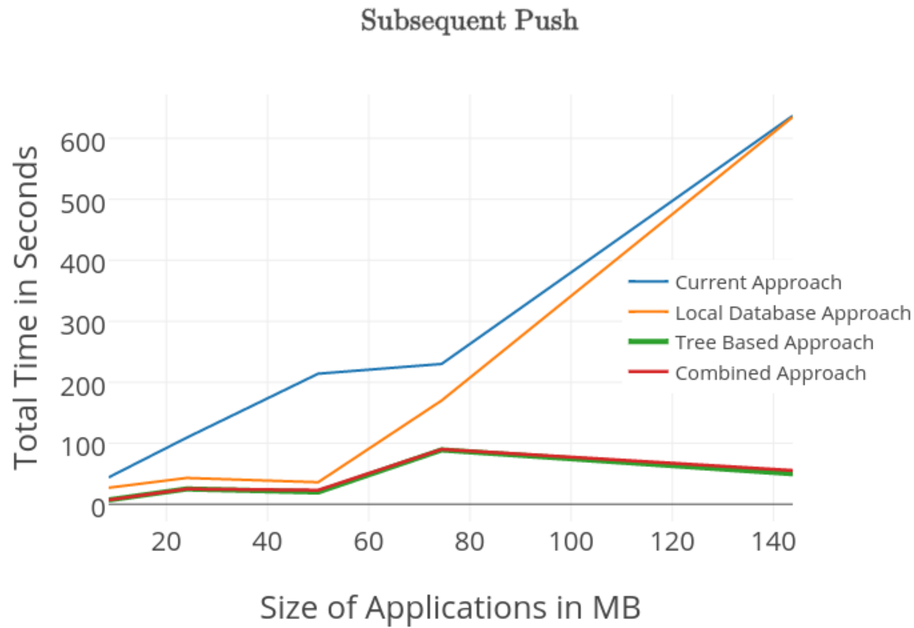


Figure 6.5: Subsequent Push

6.5.4 Assemble Package Phase

A major portion of time (of file synchronization) is invested in assembling the package during second push. There are two different ways, we considered for assembling the package. Table 6.6 displays the time taken by server to assemble and upload.

1. In Current and Local Database approach, the CF Client zips the files unknown by Bits-Service and sends it to Bits-Service. Bits-Service downloads all the files that did not change and are greater than 64 KB and assembles them with the changes sent by CF Client. After assembling Bits-Service uploads the package again.
2. In Tree Based and Combined approach, the CF Client zips the files unknown by Bits-Service and sends it to Bits-Service. Bits-Service downloads the previous application package and assembles it with the changes sent by CF Client. After assembling Bits-Service uploads the package again to blobstore.

The time displayed in table 6.6 is the total time takes to assemble the package and upload it to Blobstore. As we can see in figure 6.6, there is a huge difference between the assemble phase of two approaches. We have calculated the assembling of package when 15 files were changed in each application during subsequent push. For the approach used by the current and local database approach this time can increase or decrease further if number of changed files(greater than 64 KB) are more than 15. The time taken

	Current Approach (Seconds)	Local Database Approach (Seconds)	Tree Based Approach (Seconds)	Combined Approach (Seconds)
Large number of small and medium files 6.1	>1000	2	<1	2
Small Number of medium and large files 6.1	191	2	<1	2
Large Number of small files 6.1	11	1	<1	<1
Small number of large files 6.1	43	<1	<1	<1
Large Number of small files 6.1	4.4	<1	<1	<1
Large Number of very small files 6.1	8	<1	<1	<1

Table 6.5: Match Phase at Bits-Service

in assemble phase of the Tree based and Combined approach would remain the same until the total size of application remains the same.

6.6 Number of HTTP Requests

As we discussed in section 6.1, number of HTTP requests exchanged over the network is also an important factor that decides the performance of the approach. In table 6.7, we present the total number of HTTP request performed by each application. We can observe that the number of HTTP requests remain constant in all the applications irrespective of size and number of files in application. The number can increase or decrease in the current and local database approaches depending on the number of files that are greater than 64Kb (in size). In graph 6.7, we represent the number of HTTP request for each application in each approach. The yellow bars represents HEAD requests, the reason these bars are thin is that HEAD requests are light (with only the metadata). The green and blue bars represents PUT and GET requests. The reason these are thick is that they contain "payload" along with header.

	Current and Local Database Approach (seconds)	Tree Based and Combined Approach (seconds)
Large number of small and medium files 6.1	>1000	110
Small Number of medium and large files 6.1	645	100
Large Number of small files 6.1	140	70
Small number of large files 6.1	144	16
Large Number of small files 6.1	34	22
Large Number of very small files 6.1	24	5

Table 6.6: Total Time to assemble package

First bars of each application in the graph represents the Current Approach. It has maximum number of requests as this approach uses HEAD requests to calculate changes. Since the Current and Local database Approach uses same method to assemble package, the number of PUT and GET requests in first two bars (A and B) of each application are same. The only difference between these two is the way they calculate differences. The last two bars of each application represents GET and PUT requests of Tree based and Combined approach. Since both the approach use similar methods to assemble, the number of requests are also similar. They are reduced as compared to the other two because of following reasons:

1. Both of them saves all the HEAD requests. Tree based Approach performs only one GET request to download the JSON file that contains all the metadata. Combined approach saves this GET request as well. It reads the local database for the metadata of files.
2. Both of them downloads the entire application package instead of downloading each file individually.

	Current Approach	Local Database Approach	Tree Based Approach	Combined Approach (seconds)
Large number of small and medium files 6.1	3160	1580	6	4
Small Number of medium and large files 6.1	878	439	6	4
Large Number of small files 6.1	50	25	6	4
Small number of large files 6.1	164	82	6	4
Large Number of small files 6.1	16	8	6	4
Large Number of very small files 6.1	30	15	6	4

Table 6.7: Total Number of HTTP Request

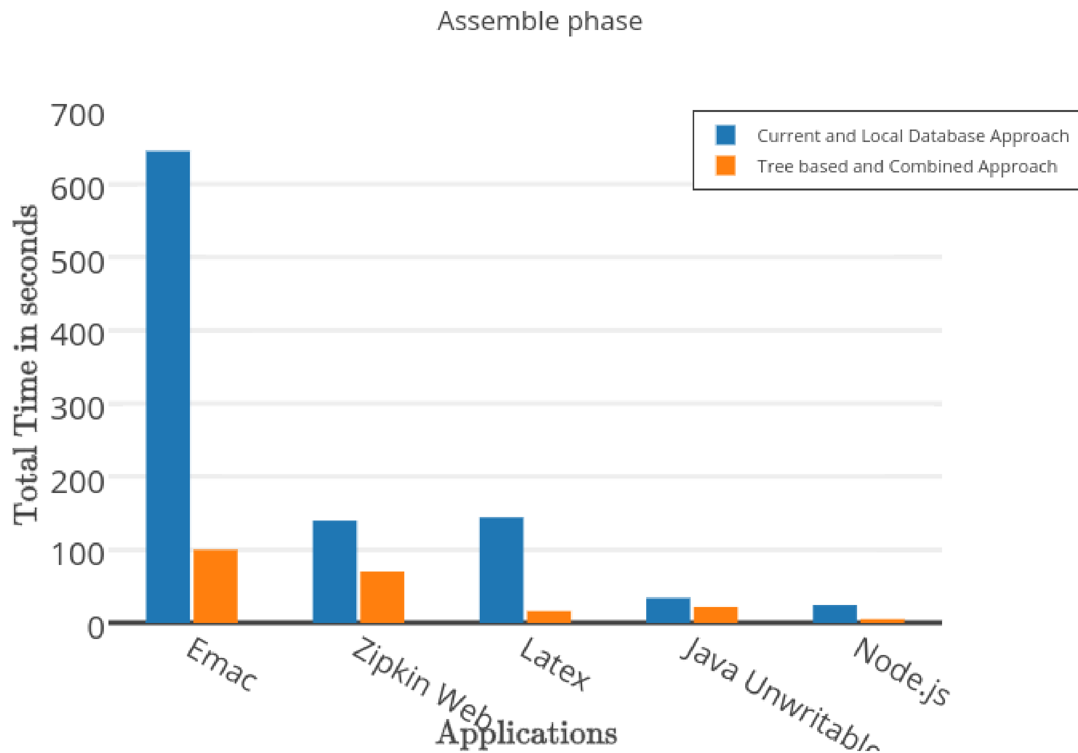


Figure 6.6: Assemble Phase

3. Tree based Approach performs an extra PUT request to upload the JSON file of metadata along with the application package. Combine Approach saves this PUT by updating the local database.

6.7 Pushing without any optimization

In last few sections, we read about the comparison of all the four approaches based on different factors. From the comparison, we can clearly see that Tree Based Approach is performing better than others. We performed the comparison of subsequent push of Tree Based Approach with No-Sync i.e. we directly uploaded files from CF Client to Blobstore without any optimization. The bandwidth between the CF client and Blobstore is low.

In table 6.8, we compare the total time taken to upload the applications with tree based approach and with no-sync. The total time for no-sync includes the time to zip the application and to upload it with a speed of 5Mbps. The results are similar to the results of Tree Based Approach because in Tree Based Approach the Upload of package is faster as it is done by the server hosting Bits-Service. The advantage of no-sync is that there

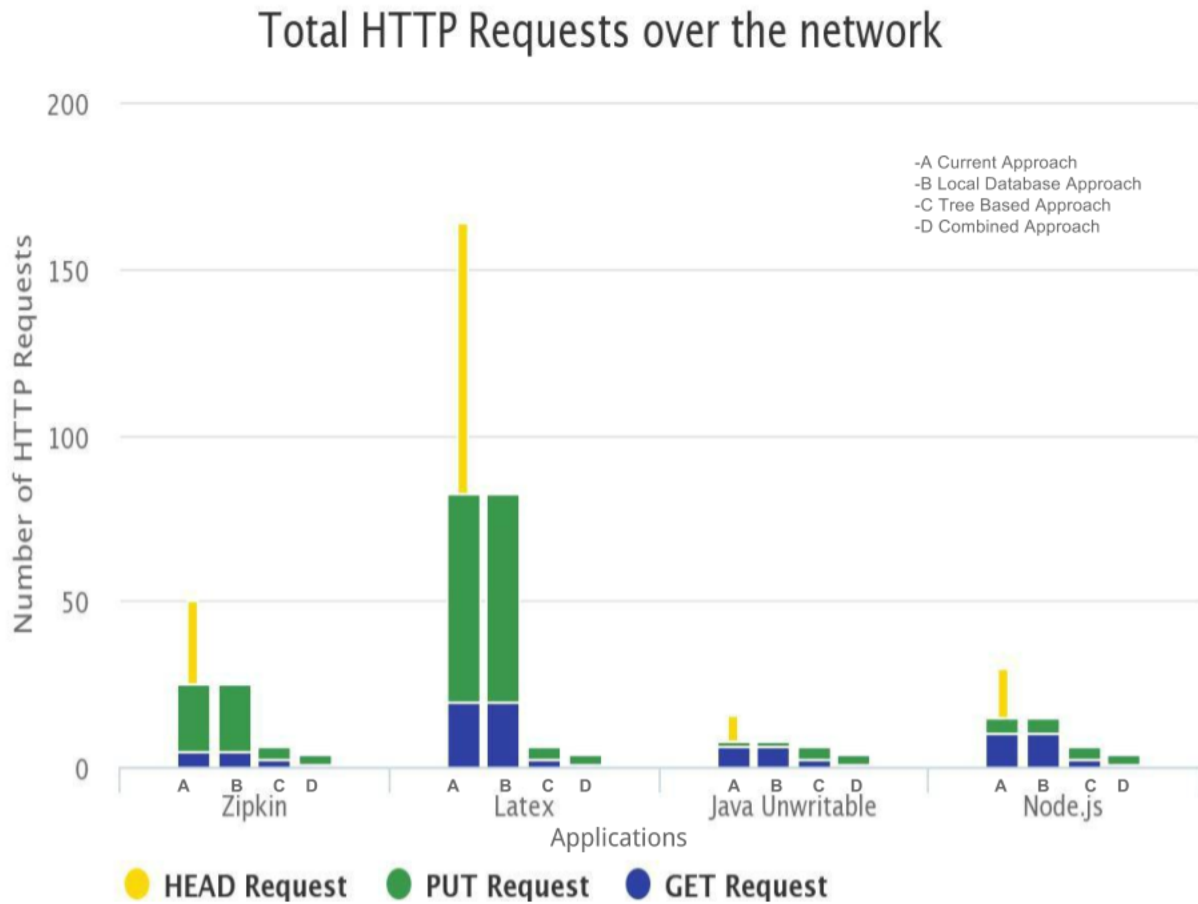


Figure 6.7: Graph for HTTP Requests

is no need to maintain an extra service to handle bits operation. The performance of no-sync highly depends on the bandwidth between CF Client and Blobstore. It is directly proportional to the bandwidth. If bandwidth improves the performance would improve else it will become worse.

	No Sync (seconds)	Tree Based Approach (seconds)
Large number of small and medium files 6.1	250	115
Small Number of medium and large files 6.1	90	106
Large Number of small files 6.1	70	78
Small number of large files 6.1	29	25
Large Number of small files 6.1	36	20
Large Number of very small files 6.1	10	7

Table 6.8: Comparison of Tree Based Approach with No-Sync

7 Conclusion and Future Work

In this chapter, we discuss the conclusion we draw from the implementation and evaluation of different file synchronization approaches. In chapter 5 and 6 we discussed four different algorithms and their results respectively. In this chapter, we will conclude the thesis with the discussion of factors like concurrency issues, incomplete Push and disaster recovery for each approach. As we saw in chapter 6, tree based approach performed better amongst all others. We measured the performance on the basis of total time taken and usage of network and storage. But in this chapter we will measure these approaches in some different situations like concurrency issue, incomplete push and disaster recovery.

In the further sections, we will discuss about the future work opportunities. These enhancements could enhance the performance of the approaches further and improve the overall result.

7.1 Conclusion

In the Chapter 6, we saw the results of evaluation of different factors like total time, number of requests etc of all the approaches for different types of application. We also saw the results of different phases of file synchronization. In almost all the phases tree based approach was performing better than the others. Following are the reasons tree based algorithm performs better than others:

1. In the first phase, it calculates fingerprint in a tree structure and stores this structure in JSON. Total time taken to calculate a tree structure is almost equal to the time taken to calculate a list structure. The complexity to calculate list structure is O(n) while to calculate tree structure is O(nk) where n is number of files and k is number of directories. In case of applications k « n which implies complexity is O(n) and hence the numerical result is proved by this analysis.
2. In the second phase i.e. match phase, Tree Based Approach is the fastest. Current approach makes HEAD request for all the files that are greater than 64KB. It takes around 1 minute to make 100 requests. Although number of files with size

greater than 64Kb in almost every application are very less, but it still takes more time than tree based approach. In local DB approach all the fingerprints of size greater than 64 Kb in the list are queried against the fingerprints saved in local database. Total time to query fingerprints is a little more than the time taken by tree based approach to calculate changes. Time taken by combined approach is almost similar to the time taken by tree based approach. Tree based approach is faster because of the tree structure, it downloads the file that has tree structure from remote blobstore and compares the received fingerprints with the fingerprints from CF Client in a tree based fashion where only few comparisons are made to find changed files.

3. In the third phase i.e file assembling phase also tree based approach is better as it downloads the package every time it needs to assemble. Unlike Current Approach and Local Database, it assemble the downloaded zip from the Blobstore, with the zip received from CF Client. As we know that the bandwidth between CF Client and server is minimum, tree based approach makes proper utilization of bandwidth between CF Client and server by only sending the changed files unlike Current and Local Database Approach that sends all files that are less than 64kb and the files that are changed. Current approach and local Db approach request each file from the blobstore to assemble the package received from CF Client. Tree based approach also reduces these number of requests.

We conclude from the points discussed above that tree based approach performs better than others in all the three phases of file synchronization. Total time taken by it is also very less, compared to others. It does not need any extra component to be maintained to store the metadata. It uses bandwidth between the CF Client and the server and between the server and the blobstore most efficiently. Also it uses the storage between them in a very efficient manner, compared to other approaches.

7.2 Performance

As seen from the evaluation results of chapter 6 tree based approach performed the best with all types of applications. The reason was also discussed in previous chapter that it does not require to store any thing in local or remote blobstore on first push and does not require to read from them in subsequent push. There are other reasons, this approach performs better than others. Following are the reasons:

1. **It does not require any local or remote cache to store file metadata**: It does not require any extra component to store metadata or application files. Therefore, there is no need to maintain any extra component unlike other approaches.

2. **It saves all the HEAD requests to calculate changes**: When compared to current approach, while comparing fingerprints, it saves all the HEAD requests and is very fast than the current approach.
3. **It saves all the PUT and GET requests to assemble the package**: When compared to current approach, while assembling the package it reuses the package zip file stored in package cache in blob store. The Current Approach stores individual files (of size greater than 64Kb) in the blob store. While assembling, the current approach assembles the package by making multiple PUT requests while tree based approach makes just one request to download the package.
4. **It makes use of the bandwidth and storage properly**: Tree based approach minimizes the number of request over network and also does not require any extra storage other than storing a very small file (that contains JSON structure) to package cache.
5. **It is a lot faster than all the other approaches**: Since it uses a tree based approach, it is faster in match phase as it makes only a few comparisons. It is faster in assembling the package, as the server downloads the entire package from blob store utilizing the bandwidth (that is a lot more than the bandwidth between client and server).

7.3 Concurrency Issues

These are the issues that arise when multiple client try to synchronize at same time with different changes. We discussed four approaches in chapter 5, in this section we will discuss the case of concurrency in all the four approaches.

1. In current approach, if two different clients start to synchronize their application with the server, both of them would store their files that are greater than 64KB on the blobstore. But only the one that would be the last to write the application package into the package cache would win.
2. In local DB approach, if two different clients start to synchronize their application with the server, both of them would store their file metadata into local database on the server. But only the one that was last to write the application package into the package cache would win.
3. In tree based approach, if two different clients start to synchronize their application with the server, only the one that was last to write the application package into the package cache would win. Also it would be able to write the file with tree JSON (guid.json) structure into blobstore.

4. In combined approach, if two different clients start to synchronize their application with the server, both of them would store their application metadata into local database on the server. But only the one that was last to write the application package into the package cache would win.

One issue could arise in current, local database and combined approach. If two clients are synchronizing simultaneously then both of them will save their fingerprints into local cache or remote cache. But the last one to write will win. The fingerprints of the previous push would already be present in local or Remote cache. The client who lost while synchronizing with the one who won, would no longer be able to synchronize its changes.

7.4 Incomplete Push

In this section we discuss the effect of incomplete push on different approaches. Incomplete push could happen due to various reasons like the client shuts down suddenly, client hangs or the server shuts down suddenly. Since all the four approaches save their metadata into local or remote database only after the application package is stored into blobstore, all the four approaches would recover from incomplete push only in following two cases.

1. If an incomplete push happens before calculation of fingerprints, nothing has changed and next push would be the same as previous one.
2. If an incomplete push happens after match phase, nothing has changed and next push would be the same as previous one.

If an incomplete push happens in between assemble phase, one issue could arise in current, local database and combined approach. If some fingerprints are saved, the fingerprints saved during that push would be there in local or Remote cache already. There could be a chance that the next push would not store all the changes. But in tree based approach, since a file based approach is used and this file is only stored after package is stored, if the file is not stored completely, the next push would be same as first push.

7.5 Disaster Recovery

This is the situation that arises when one or more component in the architecture goes down. There are various component attached for file synchronization in PaaS cloud.

Figure 7.1 shows the number of components involved in file synchronization. We will discuss the implications that could arise if one or more of them goes down.

1. **Remote Blobstore**: There are two components of blobstore used by current: App cache where it stores application files and package cache where it stores application packages.
   - If app cache goes down, Current approach would not work at all. Other three approaches would work fine.
   - If package cache goes down, the whole process of file synchronization would stop.
2. **Local Cache**: In all the approaches we use local cache to store some metadata about the application. Meta data includes mapping of application name to GUIDs. If this cache goes down server would treat each push to itself as first push and for all the four approaches, every push would behave as the first push.
3. **Bits Service**: If the server or the bits service goes down, file synchronization would stop.

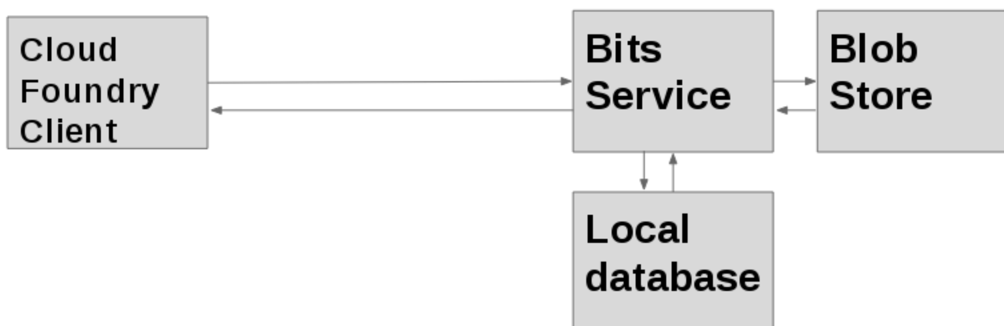


Figure 7.1: Components

7.6 Further Opportunities

There were a few things that were not implemented due the time constraint. Some of them are discussed below:

1. **Difference calculation on client side**: In all the four approaches the difference between application files is calculated at the server. Since servers have a fixed

amount of computation power but clients come with different computation powers, the difference calculation at client would improve the usage of server resources.

2. **Calculation of hierarchical tree can be improved**: Calculation of the tree structure could be improved further by using a different approach.

3. **Different Zipping algorithm**: We use ruby zip as our zipping algorithm. Other zipping algorithms could perform better than ruby zip.

4. **File based local cache**: Local database used for metadata storage is a file based databased: SQLite. Other database could improve the performance.

5. As we discussed in chapter 6, No-sync performed better in some situations and Tree based Approach performed better in others. Since a reasonable amount of bandwidth is assumed, a classification system can be formed for some applications (where the size of application is very less) which do not need to be synchronized and can be uploaded directly to the blobstore.

6. We considered the sample applications with certain features (as explained in 6.5.1). There could be a different set of Application Folder characteristics and applications. Further research can be done to find out more factors (that affect the performance) and applications (having those factors). They can be evaluated with all the four approaches or a combination of them or a new approach.

Bibliography

[Ama] Amazon. *Amazon Web Service*. URL: https://aws.amazon.com/ (cit. on p. 25).

[And98] P. Andrew Tridgell. *The rsync algorithm*. 1998. URL: https://rsync.samba.org/tech_report/node2.html (cit. on pp. 34, 37).

[Ang14] Angel Tomala-Reyes. *What is IBM Bluemix?* 2014. URL: https://www.ibm.com/developerworks/cloud/library/cl-bluemixfoundry/ (cit. on pp. 15, 25).

[Ash] Ashutosh Phoujdar. URL: https://www.codeproject.com/Articles/26628/Tree-structure-generator (cit. on p. 70).

[Azu] Azure. *Azure*. URL: https://azure.microsoft.com/en-us/?b=17.14 (cit. on p. 25).

[Bac] Backlog tool. *Git beginners guide for dummies*. URL: https://backlogtool.com/git-guide/en/intro/intro1_1.html (cit. on p. 49).

[Bos] Bosh. *What is Bosh?* URL: http://bosh.io/docs/about.html (cit. on p. 28).

[Bur] Burton H. Bloom. *Space/time trade-offs in hash coding with allowable errors*. URL: https://dl.acm.org/citation.cfm?doid=362686.362692 (cit. on pp. 41, 66).

[Clo17a] Cloud Foundry Documentation. *Deploying Cloud Foundry*. 2017. URL: http://docs.cloudfoundry.org/deploying/common/deploy.html#deploy (cit. on p. 26).

[Clo17b] Cloud Foundry Foundation. *Cloud Foundry Documentation*. 2017. URL: https://docs.cloudfoundry.org/concepts/architecture/ (cit. on pp. 26, 28).

[Clo17c] Cloud Foundry Foundation. *Cloud Foundry Documentation*. 2017. URL: https://docs.cloudfoundry.org/concepts/overview.html (cit. on pp. 28, 29).

[Clo17d] Cloud Foundry Foundation. *Cloud Foundry Documentation*. 2017. URL: https://docs.cloudfoundry.org/concepts/architecture/cloud-controller.html (cit. on pp. 28, 29).

[Col05a] Colin Phipps. "Libzsync." In: (2005). URL: http://zsync.moria.org.uk/paper200503/ch04.html (cit. on p. 37).

[Col05b] Colin Phipps. *zsync — Optimised rsync over HTTP*. 2005. URL: http://zsync.moria.org.uk/paper200503/ (cit. on pp. 36–38).

[CS14] S. Chacon, B. Straub. *Pro Git Book*. Apress, 2014. ISBN: 1484200772 (cit. on p. 49).

[FAB] FABIEN SANGLARD. *GIT SOURCE CODE REVIEW: DIFF ALGORITHMS*. URL: http://fabiensanglard.net/git_code_review/diff.php (cit. on pp. 49, 50).

[Fou] C. Foundry. *Cloud-Foundry*. URL: https://www.cloudfoundry.org/ (cit. on p. 15).

[G P14a] G. P. E. Keeling. *Improved Open Source Backup:Incorporating inline deduplication and sparse indexing solutions*. 2014. URL: http://burp.grke.org/images/burp2-report.pdf (cit. on pp. 23, 38, 39).

[G P14b] G. P. E. Keeling. *Improved Open Source Backup:Incorporating inline deduplication and sparse indexing solutions*. 2014. URL: http://burp.grke.org/burp2/15appa.html (cit. on p. 39).

[gee] geeksforgeeks. *Searching for Patterns | Set 3 (Rabin-Karp Algorithm)*. URL: http://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/ (cit. on p. 44).

[Gly16] Glyn Normington and Steve Powell. *Faster cf push for applications with lots of little bits*. 2016. URL: https://docs.google.com/document/d/12SJRxUtv4cxqGecDqL7CwmUPboqTAC5ICVdLsFLypO8/edit (cit. on p. 53).

[Goo] Google. *Google app Engine*. URL: https://cloud.google.com/docs/ (cit. on p. 25).

[Hao08] Hao Yan and Utku Imrak and Torsten Suel. *Algorithms for Low Latency Remote File Synchronization,* 2008. URL: https://pdfs.semanticscholar.org/ba16/6cd8c396eabd94210f63644e0dcf7ad674b5.pdf (cit. on pp. 48, 49).

[Her] Heroku. *Heroku*. URL: https://www.heroku.com/ (cit. on p. 25).

[Ian99] Ian H. Witten and Alistair Moffat and Timothy C. Bell. *Managing Gigabytes:Compressing and Indexing documents and images*. 1999. URL: https://sigmodrecord.org/publications/sigmodRecord/0406/RB2.Nagaraj.pdf (cit. on p. 48).

[IBMa] IBM. *Bluemix*. URL: https://www.ibm.com/cloud-computing/bluemix/ (cit. on p. 24).

[IBMb] IBM. *IBM*. URL: https://www.ibm.com/us-en/ (cit. on p. 15).

[IBM13] IBM. *The Differences between IaaS, SaaS and PaaS*. 2013 (cit. on p. 58).

[Jac14] Jacob Jencov. *Rsync*. 2014. URL: http://tutorials.jenkov.com/rsync/index.html (cit. on p. 35).

[Kar15] Karthik Byggari. *Robot class in Java - Type automatically and Capture screen with Robot class*. 2015. URL: https://logicallyproven.blogspot.de/2015/02/robot-class-in-java-type-automatically.html (cit. on p. 18).

[Kat13] Katie Frampton. *The Differences between IaaS, SaaS and PaaS*. 2013. URL: https://www.smartfile.com/blog/the-differences-between-iaas-saas-and-paas/ (cit. on p. 25).

[Maa15] Maanka Maanka. *Networking client and server*. 2015. URL: https://www.slideshare.net/yasminabdikaadirasseir/networking-client-and-server (cit. on p. 20).

[Mara] Marc Schunk and Steffen Uhlig and Peter Götz and Simon Moser. *Bits Service*. URL: https://github.com/cloudfoundry-incubator/bits-service (cit. on p. 29).

[Marb] Mariash. *A guide to using Bosh*. URL: http://mariash.github.io/learn-bosh/ (cit. on p. 26).

[Marc] Martin Pool. *Librsync*. URL: http://librsync.sourcefrog.net/ (cit. on p. 38).

[Mil] Miller Rabin and Dick Karp. *The Rabin Karp Algorithm*. URL: http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15451-f14/www/lectures/lec6/karp-rabin-09-15-14.pdf (cit. on p. 19).

[Nav05] Navendu Jain and Mike Dahlin and Renu Tewari. "TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization." In: (2005). URL: http://dl.acm.org/citation.cfm?id=1251049 (cit. on p. 42).

[Ped13] A. S. Pedro Garc´ıa Lopez and Marc S ´anchez Artigas and Cristian Cotes Guillermo Guerrero. *StackSync: Architecturing the Personal Cloud to Be in Sync*. 2013. URL: http://stacksync.org/wp-content/uploads/2013/11/stacksync_full_paper.pdf (cit. on pp. 46, 47).

[quo09] quora. *Quora*. 2009. URL: https://www.quora.com/ (cit. on p. 42).

[Ral79] Ralph Merkel. *Method of providing digital signatures*. 1979. URL: https://www.google.com/patents/US4309569 (cit. on p. 22).

[Riv] R. Rivest. *Message Digest Algorithm*. URL: https://www.ietf.org/rfc/rfc1321.txt (cit. on pp. 19, 21).

[Sal05] A. Salman. "BLOOM'S FILTERS : THEIR TYPES AND ANALYSIS." In: 2 (Feb. 2005), pp. 268–278. URL: http://journal.dogus.edu.tr/index.php/duj/article/viewFile/134/pdf_salman (cit. on p. 42).

[Sco09] Scott Chacon and Ben Straub. *Pro Git*. 2009. URL: https://git-scm.com/book/en/v2 (cit. on p. 49).

[SHA] SHA. *SHA*. URL: https://en.wikipedia.org/wiki/Secure_Hash_Algorithms (cit. on pp. 19, 21).

[Sim] Simon Eskildsen. *Toxiproxy*. URL: https://github.com/Shopify/toxiproxy-ruby (cit. on p. 85).

[Sof] Softlayer. *Softlayer*. URL: http://www.softlayer.com/ (cit. on p. 15).

[Ste16a] Steffen Uhlig and Peter Goetz. *cf-push –no-start*. 2016. URL: https://github.com/cloudfoundry-incubator/bits-service/blob/master/docs/create-app.png (cit. on p. 55).

[Ste16b] Steffen Uhlig and Peter Götz. *Bits Service*. 2016. URL: https://youtu.be/91P-Sg16bIQ (cit. on p. 17).

[SUR] SURVAR. *SURVAR*. URL: http://de.ovo.bg/shop/74/desc/survar-onlaj-radio-pod-naem (cit. on p. 21).

[Tor04] Torsten Suel and Patrick Noel and Dimitre Trendafilov. "Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks." In: (2004). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.510&rep=rep1&type=pdf (cit. on p. 39).

[Von] VonC. *Git Workflow*. URL: http://stackoverflow.com/questions/3689838/difference-between-head-working-tree-index-in-git (cit. on p. 50).

[Wer17] Werner Vogels. *Back-to-Basics Weekend Reading - Bloom Filters*. 2017. URL: http://www.allthingsdistributed.com/2017/02/bloom-filters.html (cit. on p. 41).

[wik] wikipedia. *Hash Tree*. URL: https://en.wikipedia.org/wiki/Hash_tree (cit. on p. 22).

All links were last followed on May 1, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature