Key-value storage system synchronization in peer-to-peer environments

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Sinh An Pham

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Data synchronization is the problem of bringing multiple versions of the same data on different remote devices to the most up to date version. This thesis looks into the particular problem of key-value storage systems synchronization between mobile devices in a peer-to-peer environment. In this research, we describe, implement and evaluate a new key-value storage system synchronization algorithm using a 2-phase approach, combining approximate synchronization in the first phase and exact synchronization in the second phase. The 2-phase architecture helps the algorithm achieve considerable boost in performance in all three major criteria of a data synchronization algorithm, namely synchronization time, processing time and communication cost, while still being suitable to operate in a peer-to-peer environment. The performance increase makes it feasible to employ database synchronization technique in a wider range of mobile applications, especially those operating on a slow peer-to-peer network.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Ralph Deters, for his guidance, support and advice during my study.

Furthermore, I am thankful to my committee members, Dr. Julita Vassileva, Dr. Derek Eager and Dr. Li Chen for their encouraging words, thoughtful criticism, valuable comments and suggestions.

I would also like to acknowledge my friends for their support and encouragement.

Finally, I wish to express my love and gratitude to my beloved family; for their understanding and endless love throughout the duration of my study.

Contents

List of Tables

List of Figures

List of Abbreviations

ART	Approximate Reconciliation Tree
API	Application Programming Interface
CPI	Characteristic Polynomial Interpolation
IBF	Invertible Bloom Filter
PC	Personal Computer
RDBMS	Relational Database Management System
SQL	Structured Query Language

Chapter 1

Introduction

As the number of computing devices continue to increase, nowadays it is normal for one user to have multiple personal computing devices such as smartphone, laptop, tablet, PC, etc. One stark example are smartphones, according to [39], currently there are more than 1 billion smartphones over the globe, and this number is expected to continue growing in the foreseeable future. This phenomenon comes along with the demand of being able to seamlessly switch between devices, as it would be better if a user could continue working on different machines without manually moving the program and the data being worked on. Figure 1.1 demonstrates the phenomenon: a user has multiple computing devices such as smartphone, laptop, traditional desktop PC, and tablet. That user may want to run some identical applications across these devices and keep the data being worked on the same across all devices. As a result, the matter of data synchronization arises.

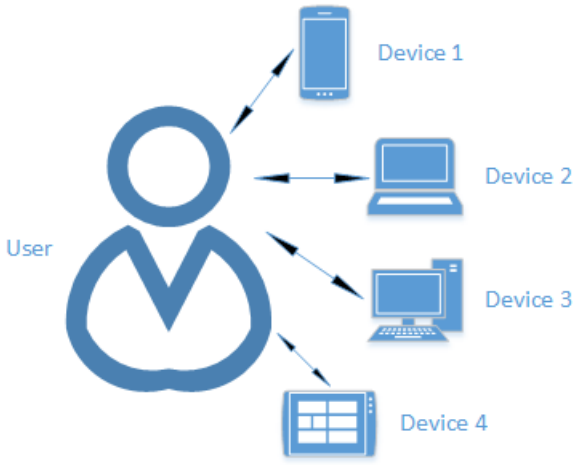

Figure 1.1: Multiple Devices Per User

User data can exist in multiple forms, but they mostly can be classified as structured data and unstructured data. Structured data includes database, JSON files, XML files, etc. Unstructured data includes photos, videos, music files and the like. Regardless, keeping both structured and unstructured data in synchronization across devices is highly beneficial. In order to accomplish that goal, there must be a mechanism to synchronize data between multiple devices. One prominent example of a synchronization algorithm for unstructured data is rsync [38], which is being used widely under Linux. Data synchronization has other important applications,

such as online storage service [17], version control, backup systems, etc.

Different data have different use cases and characteristics, which in turn makes the synchronization problem to have multiple solutions depending on the nature of the data [32, 5, 21, 35]. Take database for example, they are highly volatile, extremely important to the host application, and their integrity is vital. These characteristics make databases one of the most meaningful types of data to keep in synchronization, as they allow applications to maintain the same state across devices. The database synchronization problem has been researched extensively and has numerous solutions, catering to a wide array of scenarios [6, 16, 32, 11, 34].

Database synchronization can be done with the use of a central server keeping the authentic version of the database, however, it is desirable to have a synchronization mechanism that works between devices via local connection, i.e. a peer-to-peer environment. The problem becomes even more complex due to the nature of mobile devices: limited computing power, limited bandwidth and intermittent connectivity among others. In addition, there are numerous types of databases such as traditional Relational Database Management System (RDBMS), graph database, key-value storage systems, etc. Each kind of database has its own requirements for synchronization mechanism.

Key value storage system is a relatively new design for databases, emerged from the need to have a simple storage model which can scale better than traditional relational database approach [15, 10, 33]. Figure 1.2 illustrates a key-value storage system with each value being associated with one key, and vice versa. The most common operation of a key-value storage system is querying value based on the input key. Other supported operations include add, remove, range query, etc.

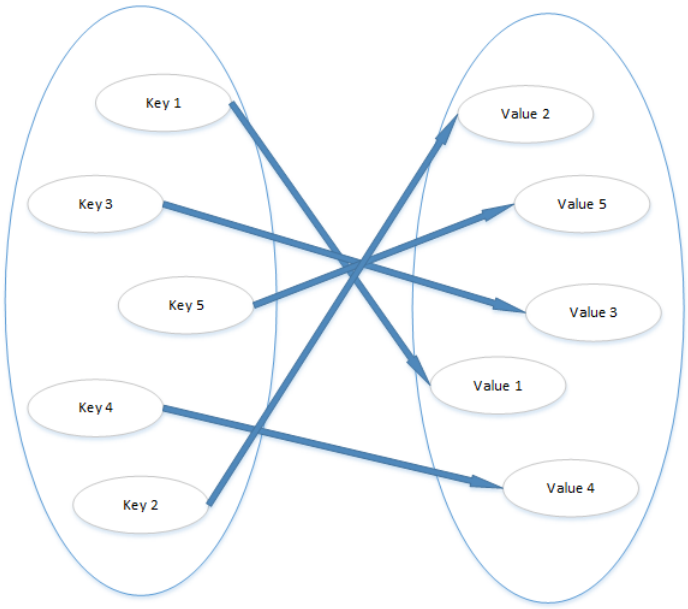

Figure 1.2: Key-value Storage System

Despite being new, key-value storage systems have become quite popular and are being used in numerous applications [13, 22]. There are a number of characteristics which makes key-value storage systems an

appealing choice, especially for web services:

- Flexible data model: unlike traditional relational databases which take data and separate it into many interrelated tables, key-value storage systems provide a very simple data model with only keys and values. The results are duplicated information in contrast to RDBMS, however storage is no longer cost prohibitive. The new data model affords developers more flexibility, ease of efficiently distributing the stored data and better read and write performance.

- Scalability and performance advantage: the simpler data model also makes it possible for key-value storage systems to scale out to multiple commodity servers. Performance is maintained by distributing load across servers, and thus is easier to adapt to the variability of users.

NoSQL (Not Only SQL) refers to storage systems which do not use tabular relations for their data model. Key-value storage system is the simplest form of NoSQL database, but precisely due to its simplicity, it can be embedded easily in less powerful computing devices like smartphones or tablets. One use case of key-value storage systems on mobile devices is caching small, but frequently used, portion of the whole database, which is usually too large to store fully in a hand-held computing device.

With the growing usage of key-value storage systems in mobile devices, the demand to synchronize them is also increasing. Despite its importance, no sufficient solution has been proposed to fill this need. In this research, we take a look at the problem of key-value storage systems synchronization in a peer-to-peer environment. We designed, implemented and measured a new algorithm for this problem which has better transmission requirements than previous methods.

The remainder of this proposal is structured as follow: Chapter 2 explains the research problem in details. Chapter 3 presents existing solutions to the research problem and discusses their pros and cons. The proposed algorithm is described in Chapter 4 along with its implementation details. Chapter 5 specifies the experiments and shows the results. Finally, Chapter 6 summaries our research and outlines some future research ideas.

Chapter 2

Problem Definition

2.1 Definition

The main goals of this research are to design, implement and evaluate a new algorithm for key-value database synchronization between computing devices running on peer-to-peer networks. Figure 2.1 depicts an overview of the objective of the algorithm: synchronize the database in multiple devices, which in this particular case are a traditional PC, a laptop, a smartphone and a tablet. Initially, each device held the same version of one database. However, over time one version of the database in one of these devices changes, and as a result, we need to bring older versions located on the other devices up to date.

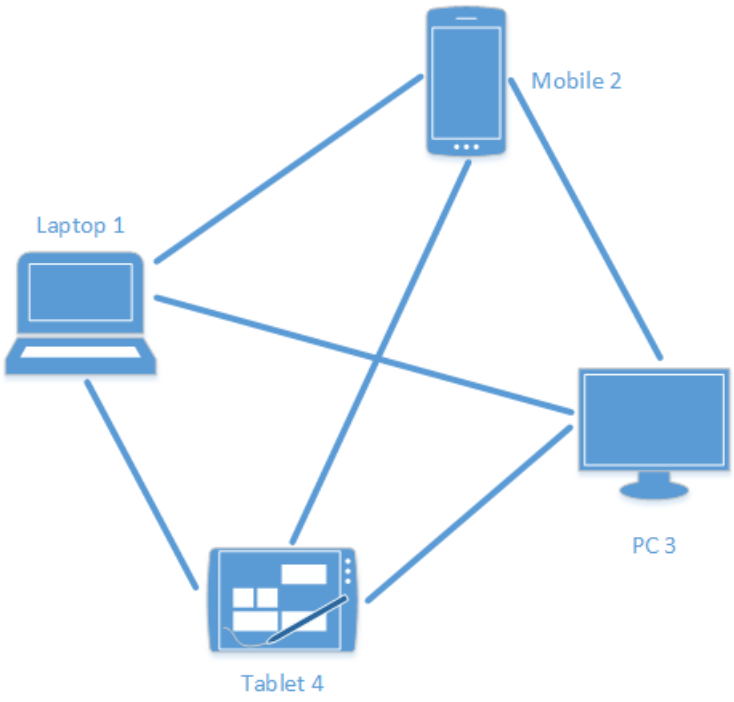

Figure 2.1: Synchronization in Peer-to-peer Environment

The following are the definitions of important keywords mentioned above:

- Algorithm: a step-by-step procedure to achieve a specific intention, which in this thesis is the synchronization of data.

- Peer-to-Peer: the system does not depend upon any single centralized server, in contrast, it could operate solely between the user's devices.
- Key-value database: in this thesis, we focus on only one type of database: the key-value storage system. This database type only supports a few simple operations: get, put and delete. The database can contain the business data of the application, user preferences, user data, etc.
- Computing devices: a general purpose device that can be programmed to carry out a set of operations. Examples include personal computer (PC), laptop, tablet, smartphone, etc.
- Synchronization: the purpose of the algorithm is to bring different versions of a database on multiple devices to the most updated version.

The problem of data synchronization in general is how to update an old version of a document on one machine to the current version located in another machine. In the context of this research, the document is a key-value database. A change in a key-value database could occur as the result of the following operations:

- Adding a new key-value data item
- Changing the value of a key-value data item
- Deleting a data item

Note that we are classifying changes as a change in the database itself, not a change in how the data is stored in the database. The difference is subtle but very important: one database can choose how to store its data in different storing device differently. It can store data by using B+ Tree in traditional magnetic hard drive and another data structure for high speed Solid State Drive (SSD). The difference at the file level would make synchronization algorithms operating at this level confused and would be detected as a huge change, thus imposes a big communication cost. In contrast, since we are looking into algorithms operating at the data item level, we would avoid this problem altogether.

2.2 Challenges

In order to develop such a synchronization algorithm, a number of key challenges need to be addressed:

- Redundancy detection: in majority of cases, between an outdated version and the current version of a database, only relatively small parts actually changed. The problem is how to detect those changed parts effectively, both in term of computing power and communication overhead. This redundancy detection is the core part of the whole synchronization algorithm.
- Protocol design: to operate reliably over the limited bandwidth and intermittent nature of peer-to-peer connection, the protocol must be designed to handle intermittent connectivity and have high fault tolerant property.

- API design: to make the algorithm appealing to other developers, the API need to be easy to use, report clear error messages, easy to extend and customize.

- Cross-platform support: the algorithm aims to run across multiple mobile platforms including iOS, Android and Windows Phone.

2.3 Formalization

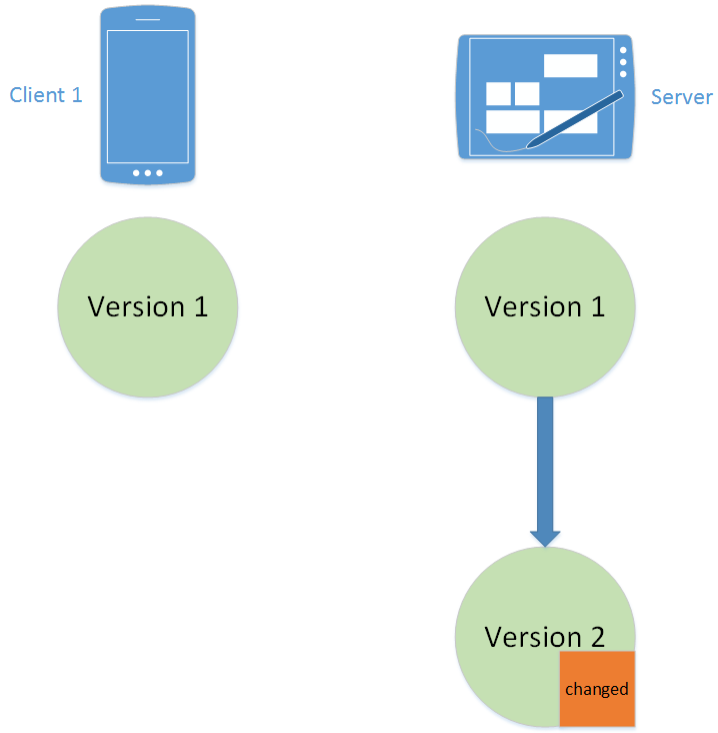

Figure 2.2: Changing Database

The remote data synchronization problem can be described as follow: given two versions of the same data D_{old}, $D_{new} \in \Sigma^*$ over some alphabet Σ (usually character/byte) locate in two machines C (the client) and S (the server) respectively. C and S are connected over a communication link, D_{old} is the outdated version, and D_{new} is the current version of the data. The problem is how to update D_{old} to D_{new}? We also assume that C only has knowledge of D_{old} and S only has knowledge of D_{new}.

The goal is to design a communication protocol between C and S that results in C holding a current copy of D (D_{new}) while minimizing the communication cost.

We call d is the changed portion of the data being synchronized. Usually, the changed part is only a small portion of the whole database.

Figure 2.2 demonstrates this circumstance. Initially both the smartphone and the tablet held an identical copy of the same database, which we called "Version 1" in the picture. Over time, the tablet made some

changes to the database, but only a portion of the database has changed, which is marked with the orange color. The problem is how we detect the changed portion and transfer it to the smartphone in order to bring its copy of the database to the current version.

We can observe that the minimum bytes of data which is needed to bring the old version in the smartphone to the current version in the tablet is d, and thus, the best communication complexity is $O(d)$.

Chapter 3

Literature Review

There are numerous existing approaches to data synchronization in general and database synchronization in particular. Some algorithms are applicable to the data synchronization problem in general, while others are only for key-value database synchronization. The following sections will provide an overview of the most prominent solutions.

3.1 Server Centric Approach

One obvious method for database synchronization is to simply transfer the whole database from the server to the client to replace the old version in the client. This is the simplest form of data synchronization, and thus easiest to implement and understand. In addition, there is minimal processing being done beside reading the database, so the processing cost is negligible. However, we can quickly notice that the communication cost of this method is O(n) - where n is the number of items in the current version of the database - and thus, it is prohibitively expensive for synchronizing practical databases. In a mobile peer-to-peer environment, the weakness of communication cost becomes even more severe since the connection is usually less capable than a wired connection.

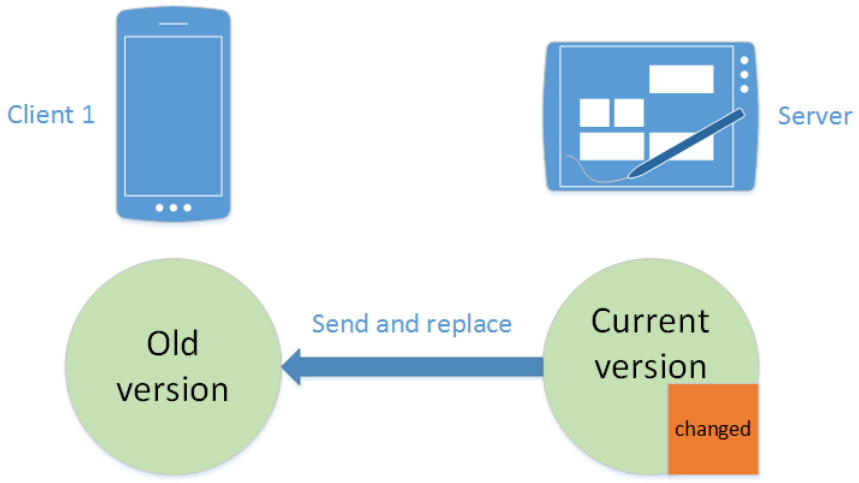

Figure 3.1: Naive Approach

An optimization which can be applied to key-value storage systems is that instead of sending the whole

database, we can just send identifiers, usually in form of hash values, representing each key-value item. In the client, it would have to convert its own key-value items into identifiers using the same algorithm, and then compare them with the received list of identifiers from the server in order to detect the differences. Afterwards, the client now can request the differences from the server. The communication cost of this optimization is still O(n), however it is better than the original naive approach since identifiers are smaller than the whole item.

3.2 Log-based Approach

The main idea is to keep a log of applied operations since the last synchronization on the server for each participating client. In order to respond to a synchronization request, the server would have to determine what new operations have been applied since the last synchronization for the client issuing the request and then transmit them to this client. The transmission cost of this approach is O(d), which is the best theoretical possible communication complexity. In addition, the processing cost for doing synchronization is also negligible, similar to the previous approach when there is minimal processing being done. In this case, the processing cost only includes the looking up operation for the requesting client and reading the log stream.

The advantage of this approach is its simplicity in term of implementation, yet being effective in term of communication cost. Nonetheless, there are two significant disadvantages: storage cost and reduced write performance. To maintain the log, the system would have to use more space alongside with the main database, and for each write operation, the system would have to update the log. Note that the writing problem can be partially mitigated by using batch writing, when each update to the log is kept in memory first and batch written to slower disk system later.

Due to its disadvantages, this approach is only suitable for client/server environment in which the server have sufficient storage capability and write performance. In a peer-to-peer environment with mostly low-powered computing devices such as mobile phones, this approach is unsuitable.

3.3 Timestamps

Timestamp is also one of the traditional methods for data synchronization. To apply this method to key-value storage system synchronization, we mark each key-value data item with a timestamp. Whenever an update occurs, the server increases the timestamp corresponding to the changed data item. When a client requests synchronization, it first sends all keys and their current timestamps to the server, the server would respond by transferring all data items with timestamps being later than the received ones.

Similar to the previous approach of logging, the timestamps approach also requires more storage space and reduces writing performance of the database. Considering storage requirement, this approach stores timestamps along with the authentic data, thus requires more storage space. In addition, each write operation

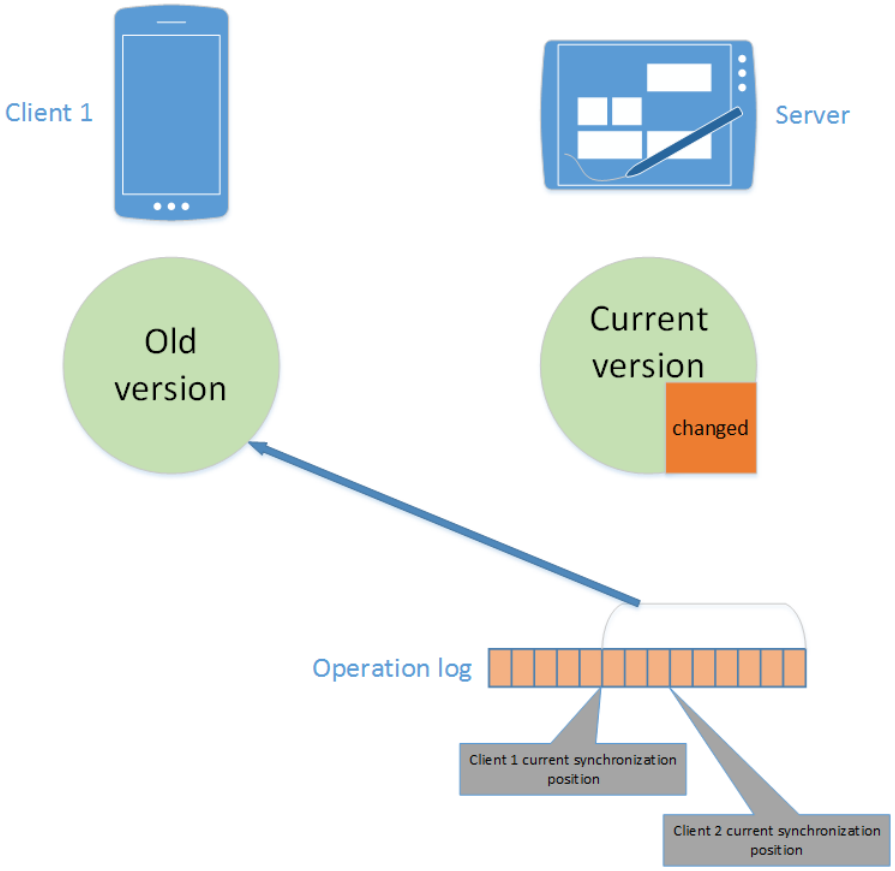

Figure 3.2: Log-based Approach

Figure 3.3: Assigning Timestamps

also needs to update the attached timestamp. The processing time of this approach is O(n), however, it is relatively fast in compared to other O(n) approaches since it only needs to compare timestamps, usually in the form of integers. The communication cost is O(n) since we have to sends all the keys and its timestamps to the server.

There are two major ways to implement timestamp: using the system clock and using a logical clock. The following sections will describe each method in detail.

3.3.1 Using System Clock

As the name implied, the server uses its system clock to mark the timestamps. This method is the most natural way of applying timestamps to data synchronization.

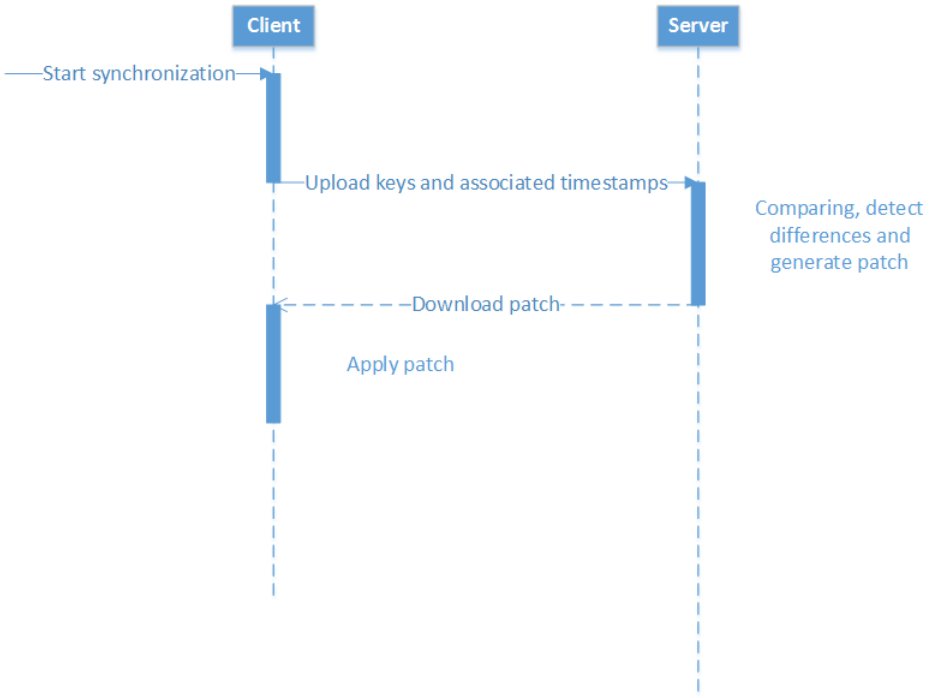

Figure 3.4: Synchronization Using Timestamps

This presents numerous problems due to how system clock works: since it depends on the system clock, now even when the clock is wrong the system administrator might not want to change it as it may cause the synchronization to fail. Another potential issue comes from daylight savings time when the clock is adjusted to compensate for different daylight in different seasons. This can be remedied by not using local system clock but GMT time. However, due to its many pitfalls, using system clock is not recommended to implement timestamp based synchronization.

3.3.2 Using Logical Clock

A logical clock is a method of detecting the order of events in a distributed system. For the purpose of synchronizing key-value database, a simple implementation of logical clock is to assign with each key-value data item a counter number. The synchronization algorithm uses the counter number for detecting which data item is newer.

The algorithm follows a few rules listed below:

1. Each data item increments its counter on each event that changes the data. Each counter starts with number 0.

2. When a client needs to do synchronization with a server, it sends all of its keys and corresponding counters to the server. Note that only the keys and the counters are sent, the values of the data items are not sent.

3. On receiving a message, the server detects modified data items by comparing its own counter values for each key. All the changed and new data items are sent back to the client.

This simple implementation of logical clock works without depending on the system clock and thus, avoids many of its problems.

3.4 Characteristic Polynomial Interpolation - CPISync

The problem of key-value storage system synchronization can be reduced to the set reconciliation problem in a straight forward manner: every key-value item is mapped to an identifier via a mapping mechanism, and then we solve the set reconciliation problem on the identifiers. This reduction step is similar to the optimization of the naive approach described earlier. The set reconciliation problem is defined as follow in [27]:

> Given a pair of hosts A and B, each with a set of length b bitstrings, how can both host determine the union of the two sets with a minimal amount of communication—both with respect to the number of exchanges between the two hosts and with respect to the number of bits of information exchanged.

The set reconciliation problem has multiple approaches [27, 18, 31]. One mathematics based solution for the exact set reconciliation problem is based on characteristic polynomial interpolation (CPI) [27].

3.4.1 Characteristic Polynomial

Given a set S = $\{x_1, x_2, ..., x_n\}$, S can also be represented by its characteristic polynomial, which is a univariate polynomial defined as follow:

$CP_S(x) = (x - x_1)(x - x_2)...(x - x_n)$

An important property of characteristic polynomials is that they allow us to cancel out all the common items in two sets S_A and S_B by using division between the CP_{S_A} and CP_{S_B}:

$\frac{CP_{S_A}(x)}{CP_{S_B}(x)} = \frac{CP_{S_A \cap S_B}(x) * CP_{S_A - S_B}(x)}{CP_{S_A \cap S_B}(x) * CP_{S_B - S_A}(x)} = \frac{CP_{S_A - S_B}(x)}{CP_{S_B - S_A}(x)}$

Since all the common items cancel out, the degrees of $CP_{S_A - S_B}(x)$ and $CP_{S_B - S_A}(x)$ are relatively small. As a result, we can reconstruct $\frac{CP_{S_A - S_B}(x)}{CP_{S_B - S_A}(x)}$ based on a relatively small number of evaluation points.

The set reconciliation algorithm with a known bound $\overline{d}$ on the size of the symmetric difference using characteristic polynomial (the CPISync algorithm) works as follow [37]:

1. Initially, the server holds S_A and the client holds S_B

2. In both the client and the server, map elements of the corresponding set to elements of some finite field F_q where q is a prime number greater than or equal to the maximum item value of S_A and S_B

3. In the client: calculate $CP_{S_B}(x)$ at $\overline{d}$ evaluation points, send the results to the server

4. In the server: calculate $CP_{S_A}(x)$ at $\overline{d}$ evaluation points, use the values of $CP_{S_B}(x)$ received from the client and perform division to get the values of $\frac{CP_{S_A}(x)}{CP_{S_B}(x)}$ at these evaluation points
5. Interpolate the rational function $\frac{CP_{S_A}(x)}{CP_{S_B}(x)}$
6. Factor $CP_{S_A-S_B}(x)$ and $CP_{S_B-S_A}(x)$ to get back $S_A - S_B$ and $S_B - S_A$
7. Send the results to the client

Note that all calculations on the characteristic polynomial are done over a finite field F_q.

In practice, we usually do not know $\overline{d}$ in advance. As a result, without knowing the bound $\overline{d}$ on the size of the symmetric difference, we would have to do another step in order to estimate $\overline{d}$. There are a number of algorithms for this step, they will be described in section 3.7, "Set symmetric difference size estimation".

3.4.2 Example

Consider the sets:

$S_A = \{1, 2, 3, 4\}$

and

$S_B = \{1, 2, 5, 6\}$

stored at nodes A and B respectively. We represent all items of S_a and S_b as elements of a finite field F_{11}.

The characteristic polynomials of S_a and S_b are:

$CP_{S_A}(x) = (x-1)(x-2)(x-3)(x-4)$

$CP_{S_B}(x) = (x-1)(x-2)(x-5)(x-6)$

Since the number of elements that differs between S_a and S_b is 4, we need to evaluate $CP_{S_A}(x)$ and $CP_{S_B}(x)$ at at least 4 different evaluation points in order to determine the union of S_A and S_B. Assuming each host agrees on the evaluation points $E = \{-1, -2, -3, -4\}$, we have the results in Table 3.1.

Table 3.1: Characteristic polynomial

x	-1	-2	-3	-4
$CP_{S_A}(x)$	10	8	4	8
$CP_{S_B}(x)$	10	1	10	5
$CP_{S_A}(x)/CP_{S_B}(x)$	1	8	7	6

Node B evaluates its characteristic polynomial, sends the values at all evaluation points {10, 1, 10, 5} to node A. Node A evaluates its own characteristic polynomial to get {10, 8, 4, 8}, and then uses the value of $CP_{S_B}(x)$ just received to calculate $CP_{S_A}(x)/CP_{S_B}(x)$. It then interpolates the rational polynomial from these values {1, 8, 7, 6} and gets:

$\frac{CP_{S_A}(x)}{CP_{S_B}(x)} = \frac{CP_{S_A-S_B}(x)}{CP_{S_B-S_A}(x)} = \frac{x^2+4x+1}{x^2+8}$

To get back $S_A - S_B$, we need to factor the numerator $x^2 + 4x + 1$, the result is {3, 4}

Similarly, to get back $S_B - S_A$, we need to factor the denominator $x^2 + 8$, the result is {5, 6}

3.4.3 Performance

The CPISync algorithm achieves O(d) in communication cost, with d being the size of the symmetric difference between the two sets. However, in terms of computing cost:

1. The steps of calculating $CP_{S_A}(x)$, $CP_{S_B}(x)$ are O(nd), and $CP_{S_A}(x)/CP_{S_B}(x)$ is O(d)
2. The rational function interpolation is O($\overline{d}^3$)

Thus this algorithm is only suitable when the size of the symmetric difference between the two sets are small.

3.5 Invertible Bloom Filter

Another algorithm for set reconciliation is the Invertible Bloom Filter (IBF) as described in [18]. This approach uses a novel data structure, the Invertible Bloom Filter, which support the set subtraction operation to detect the difference between two remote sets.

3.5.1 Bloom Filters

Definition

A Bloom Filter is a probabilistic data structure used to test whether an element is a member of a set [7]. It achieves small space requirement by allowing false positive in its query operations. Due to low space requirement and constant look up time, it is widely used in situations where storage is limited and some false positives are acceptable or controllable [8].

A Bloom Filter has two components:

- An m-bit array
- K hash functions each yielding a value between 0 and m - 1.

Bloom Filters support two operations: add and query.

1. To add an element
   (a) Hash the element using the k hash functions to get k indices
   (b) Set the bits in the bit array at these indices
2. To query if an element is in the set
   (a) Hash the element using the k hash functions to get k indices. The following two scenarios can occur:

i. If all bits at these k indices are 1, then the element may be in the set

ii. If one or more bit at k hash values are 0, the element does not belong to the set

Example

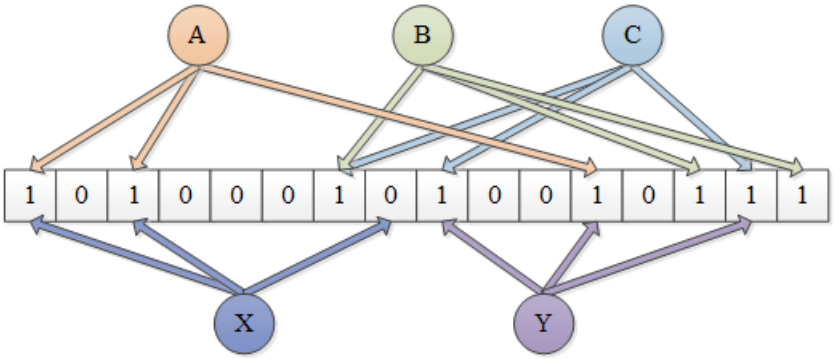

Figure 3.5: Bloom Filter

Figure 3.5 shows an example of a Bloom Filter with 16-bit array and 3 hash functions. In our example, there are 5 data items: A, B, C, X and Y. The three hash functions produce the following values for these items:

- Item A: 0, 2 and 11
- Item B: 6, 13 and 15
- Item C: 6, 8 and 14
- Item X: 0, 2 and 7
- Item Y: 8, 11 and 14

Items A, B and C have been added, thus the corresponding bits of the Bloom Filter are set. When query item X, the Bloom Filter returns "No" since one of its bit is not set. However, querying Y is a case of false positive since all its hash values have been set, but in fact it does not belong to the original set.

False Positive

After inserting n keys into an array of m bits, the probability that a particular bit is still 0 is:

$(1 - \frac{1}{m})^{kn}$

Hence the probability of false positive in querying this Bloom Filter is [1]:

$p_{false} = (1 - (1 - \frac{1}{m})^{kn})^k$

Note that if we knew the number of items that would be added to a Bloom Filter, we can control the false positive rate by changing m (the length of the bit array) and k (the number of the hash functions).

3.5.2 Using Bloom Filter in Data Synchronization

While Bloom Filter does not directly solve the data synchronization problem, it serves as the foundation of multiple data synchronization algorithms as it provides a mechanism to query a set effectively.

The most notable data synchronization algorithms using Bloom Filter are:

- Approximate synchronization, this will be discussed in detail in later section
- Using Bloom Filter in conjunction with other algorithms, in particular with Hash Tries [30]

The second algorithm works as follows:

- Use Bloom Filter to do approximate synchronization first
- Use Hash Tries to do exact synchronization in the second phase, which contains only the false positives left over from the first phase.

Figure 3.6 shows an example of a hash tries containing 6 members, S = "0000", "0010", "0110", "0111", "1001", "1000". Each of the rectangles represents a hash value, while each ellipse depicts a member of the set. All members sharing the same prefix are grouped together, and the data structure uses a hash function to calculate the hash value representing all these members. At the root of the trie, a root hash value is calculated taking into account all members of the set.

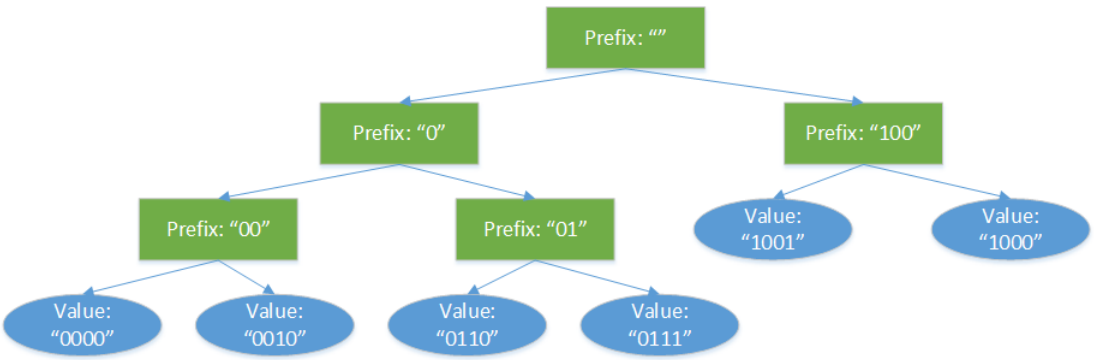

Figure 3.6: Hash Trie

The algorithm accomplishes synchronization by exchanging and comparing the hash values in each level of the trie. First the root hash value is sent and compared, if it is the same, the algorithms concludes that the whole database is identical and no more steps are required. If the root hash values are not the same, we go on to compares the hash values at the second level, below the root level. The algorithm continues these steps until it detects all different data members of the two sets.

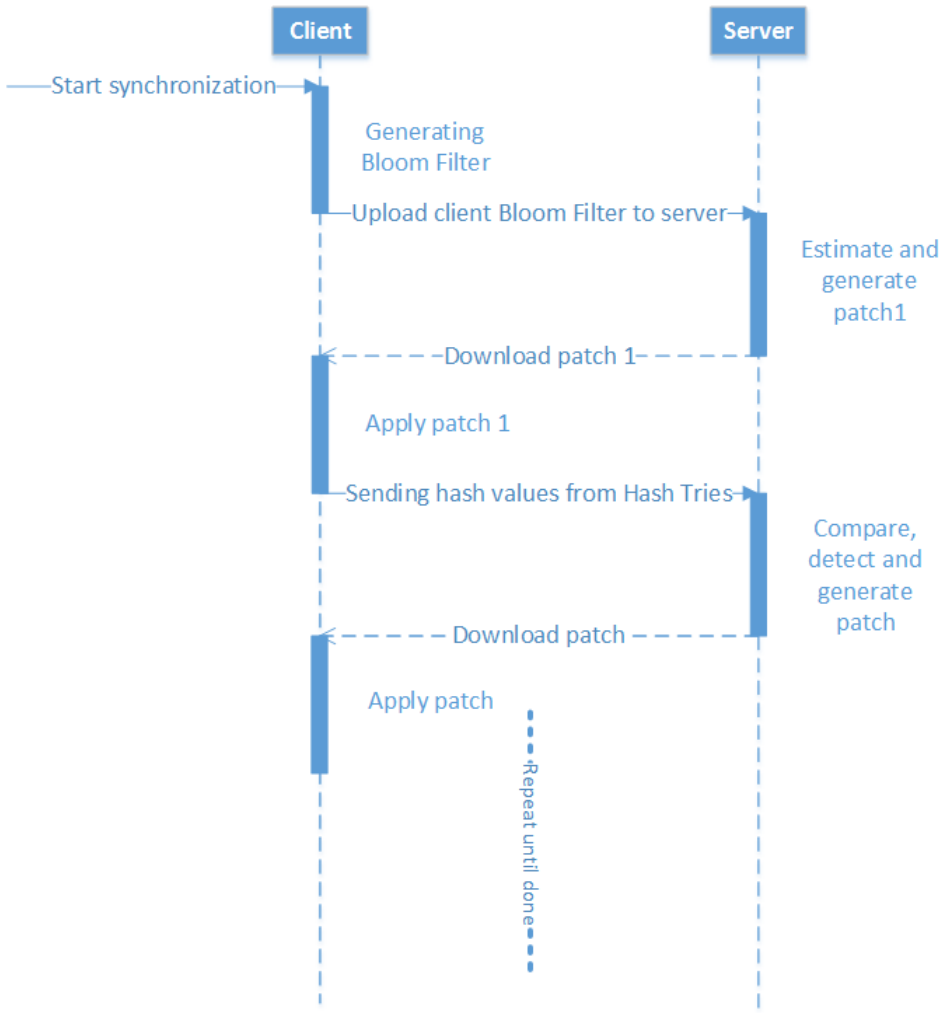

Figure 3.7: Bloom Filter Hash Tries Sequence

Figure 3.7 shows the sequence of this synchronization algorithm. Note that this algorithm uses multiple rounds of communication for the second phase.

3.5.3 Invertible Bloom Filter

Definition

The original Bloom Filter does not allow retrieving elements after they have been added to the Bloom Filter. An Invertible Bloom Filter is a variation of Bloom Filter designed specifically for this feature: under the right settings, it can be inverted to yield the elements inserted[18]. Instead of a bit array in traditional Bloom Filter, an IBF has an array of cells and each individual cell has 3 fields inside:

- idSum: XOR value of all item ids which have been hashed into this cell
- hashSum: XOR value of all hash values, calculated by an independent H_c hash function

- count: number of items in this cell

To add an item with id idx into an IBF, we use k hash functions and one special hash function H_c. K hash functions are used to generate the indices of the cells, and then we XOR idx into B[i].idSum, XOR $H_c(idx)$ into B[i].hashSum, and increase B[i].count by 1.

Example

Figure 3.8 demonstrates an IBF. Items A, B and C have been added to the IBF using 3 hash functions to determine their indices. We can clearly see in Figure 3.8 that the corresponding idSum, hashSum and count fields have different values than the initial value of 0 due to this fact.

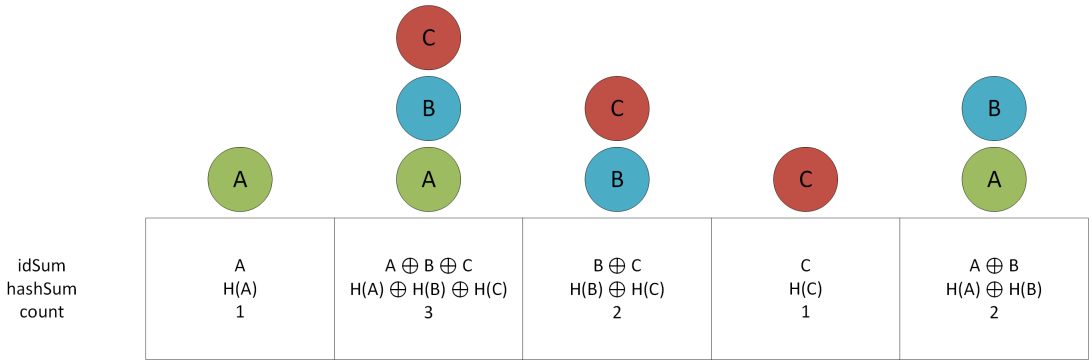

Figure 3.8: Invertible Bloom Filter

By utilizing these 3 fields instead of being just a bit array, IBF has the ability to do subtraction on sets. This depends on the fact that common items would cancel out when doing subtraction, due to the characteristic of the XOR operation used in both idSum and hashSum fields:

$A \oplus B \oplus A = B$

We can see this subtraction feature of IBF by considering an example. Considering the server set, $S_{server} = \{B, C, D, E, F, G\}$. The IBF generated by this set is shown in Figure 3.9.

The client set, $S_{client} = \{A, D, E, F, G\}$. The IBF of this set is illustrated in Figure 3.10.

The subtraction operation is done by applying the steps shown in Algorithm 1.

input : IBF_1 and IBF_2

output: IBF_3, the result of $IBF_1 - IBF_2$

for *i in 0..n-1* **do**

$IBF_3[i].idSum = IBF_1[i].idSum \oplus IBF_2[i].idSum$

$IBF_3[i].hashSum = IBF_1[i].hashSum \oplus IBF_2[i].hashSum$

$IBF_3[i].count = IBF_1[i].count - IBF_2[i].count$

end

Algorithm 1: IBF subtraction

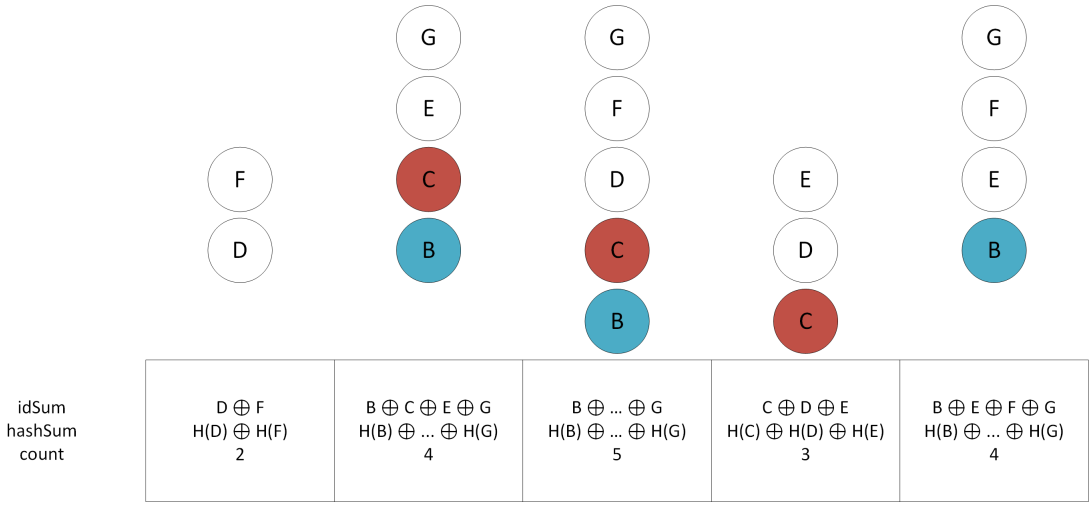

Figure 3.9: Invertible Bloom Filter - Subtraction, Server IBF

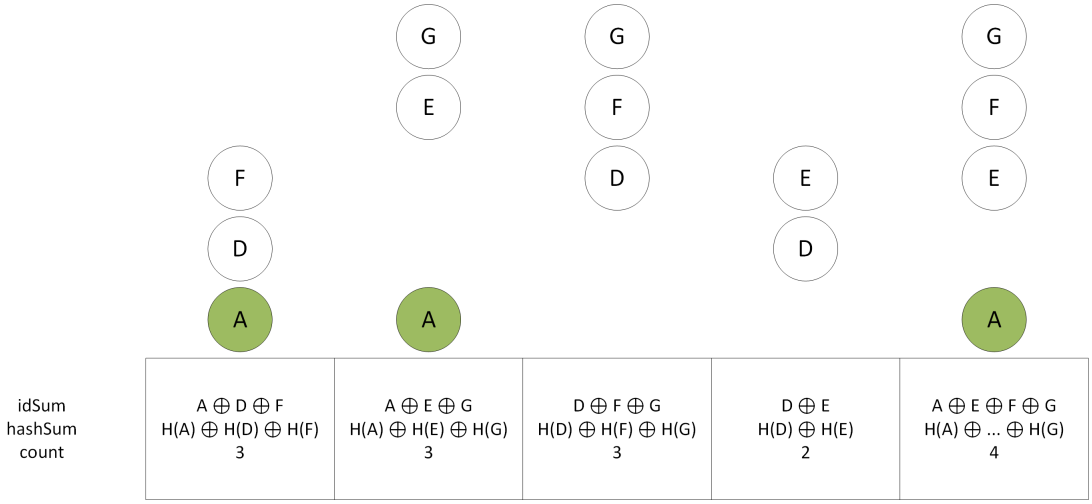

Figure 3.10: Invertible Bloom Filter - Subtraction, Client IBF

Take $S_{server} - S_{client}$ for example, for the first cell, we would have the result cell:

1. idSum = $\{D \oplus F\} \oplus \{A \oplus D \oplus F\} = A$
2. hashSum = $\{H(D) \oplus H(F)\} \oplus \{H(A) \oplus H(D) \oplus H(F)\} = H(A)$
3. count = 2 - 3 = -1

Notice this would cancel out the common items in the idSum and hashSum fields of the result IBF. We do this to the remaining cells of the IBF. The result of doing these steps for the IBFs of the server set S_{server} and the client set S_{client} is shown in Figure 3.11

To recover the elements of the set which has been added to the IBF, firstly we need to identify all the "pure" cells in the IBF. In an IBF, a pure cell is defined as follow:

1. The count field must be 1 or -1

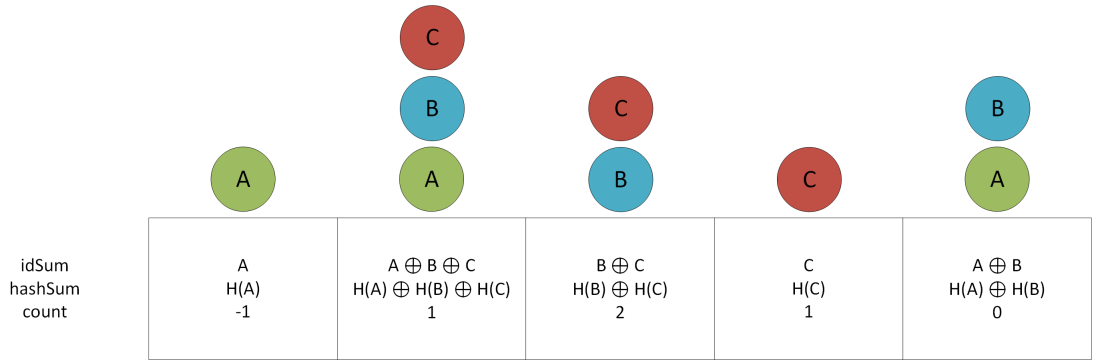

Figure 3.11: Invertible Bloom Filter - Subtraction, Result

2. The hashSum field must equal to $H_c(idSum)$

A pure cell will have its id sum field equal to the id of the only inserted element it contains. The reason IBFs pure cells must have the count field equal to 1 or -1 is to allow decoding IBFs which are results of subtraction operation. For example, the result IBF of $IBF_{S_A} - IBF_{S_B}$ will have 1 as the counts of cells belonging to S_A, and vice versa.

The following algorithm is used to recover elements encoded in a IBF[18]:

1. Identify pure cells, put them in a pure cell list.

2. Recover an element from one of the pure cells, and remove that element from the IBF and the pure cell list. This step could possibly make other cells in the IBF pure, which are then added to the pure cell list.

3. Repeat until done, which means there are no indices remaining in the list of pure cells. At this point, either all items in the IBF have been recovered, or some remain encoded in the filter.

Example - IBF decoding

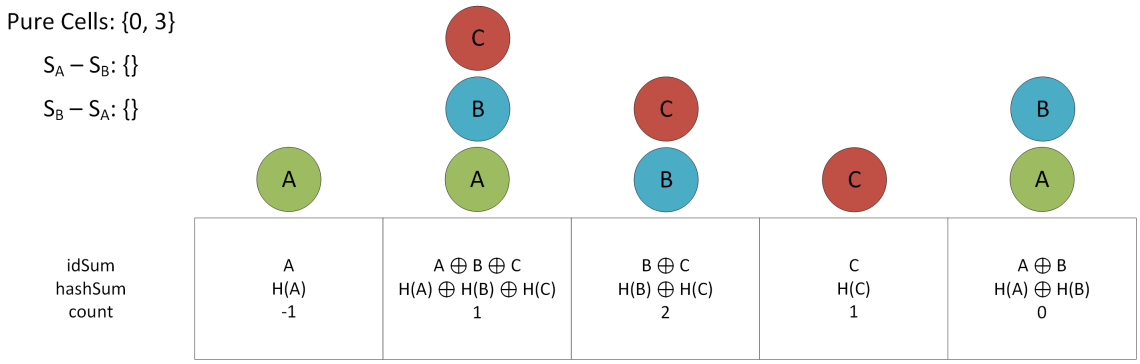

Figure 3.12: Invertible Bloom Filter - Recovery Step 1

Figure 3.12 shows the first recovery step of an IBF, we use the result IBF in the last step as an example. Initially we detect 0, 3 as the pure cells, and thus they are added to the pure cell list. We then examine the first pure cell: cell index 0. It has the count field value of -1, thus it belongs to the $S_B - S_A$ list. The id "A" is fetched from the idSum field, and "A" is added to $S_B - S_A$.

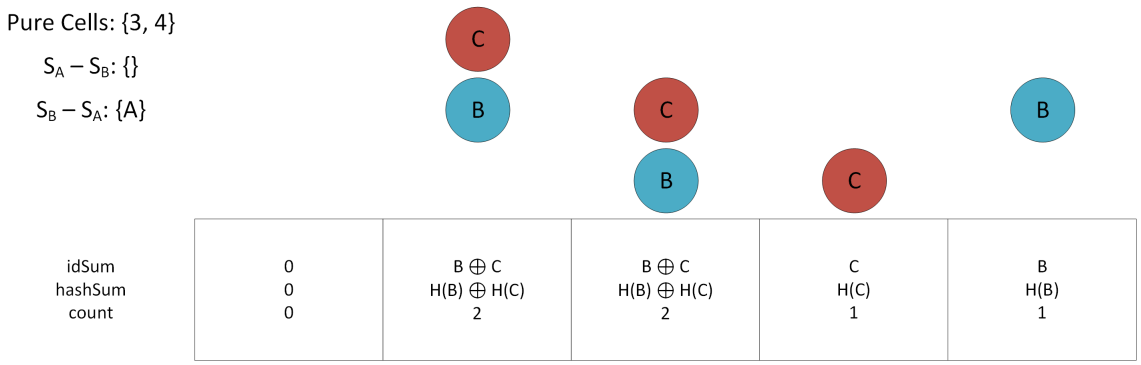

Figure 3.13: Invertible Bloom Filter - Recovery Step 2

We then remove item A from the IBF and remove 0 from the pure cell list. By doing so, cell 4 also becomes a pure cell, thus is added to the pure cell list. Figure 3.13 shows the state of the IBF after these steps.

The process continues until the pure cell list is empty. At this point, if all cells in the IBF are 0 then we have fully recovered all items from the IBF. If the IBF still contains cells which are not 0, we conclude that the recovery process has failed.

Parameters

The recovery process of an IBF depends on the pure cells we could identify, therefore one major problem of implementing an IBF is how many cells should each element be hashed into? We refer to this number as *hash_count*. If *hash_count* is too small, for example 1, then we can easily see that there will be a high probability of finding pure cells initially. However, once all the initial pure cells have been recovered, we will have no other pure cell to consider, since the *hash_count* is 1, removing one pure cell will not lead to any other pure cell. On the other hand, if *hash_count* is too big, it is unlikely that there will be pure cell at the beginning of the recovery process. In [18], the authors showed that *hash_count* values of 3 or 4 work well in practice.

In addition, similar to the original Bloom Filter, the length of the IBF is directly proportional to the number of items it needs to decode[18].

There is a chance the operation could not complete, however we can control this probability to be low enough in practice and then have recovery mechanism in case it happens.

3.5.4 Invertible Bloom Filter in Set Reconciliation

Since IBF allows us to do set subtraction and decode items that have been added to the IBF, we can apply IBF to the set reconciliation problem as follows[18]:

1. The client calculates and sends its IBF to the server
2. The server calculates its own IBF, does the subtraction with the IBF received from the client, and then recovers the resulted IBF to achieve the set of $S_A - S_B$
3. The server sends all these detected items to the client
4. The client uses the different items to update its own database

Note that in order for the above algorithm to work, more specifically for the recovery process of IBF to be successful, we must first know the size of symmetric difference between two sets. This parameter is used to allocate the suitable size for the IBF.

3.6 Approximate Set Reconciliation

Another approach in data synchronization is approximate synchronization: instead of synchronizing all the data items, the algorithm only tries to synchronize as many items as possible. The motivation is the fact that numerous applications can tolerate a small error and thus, can work with partially synchronized data. In addition, approximate synchronization can also be used as a prelude to a more expensive exact synchronization algorithm for applications requiring exact synchronization.

3.6.1 Bloom Filters for Approximate Synchronization

One simple algorithm for approximate synchronization is to use a Bloom Filter. As observed in [9], this algorithm is surprisingly effective, especially when the number of differences is a large fraction of the set size. Figure 3.14 illustrates the algorithm.

The algorithm works as follows:

1. The client creates a Bloom Filter containing all its data items.
2. The client sends the Bloom Filter created in the last step to the server.
3. The server queries the received Bloom Filter to detect elements which do not exist in the client. Because of the nature of Bloom Filter, there will be some false positive (items thought to exist in the client while in fact they do not)
4. The server sends those detected data items to the client.
5. The client updates its database.

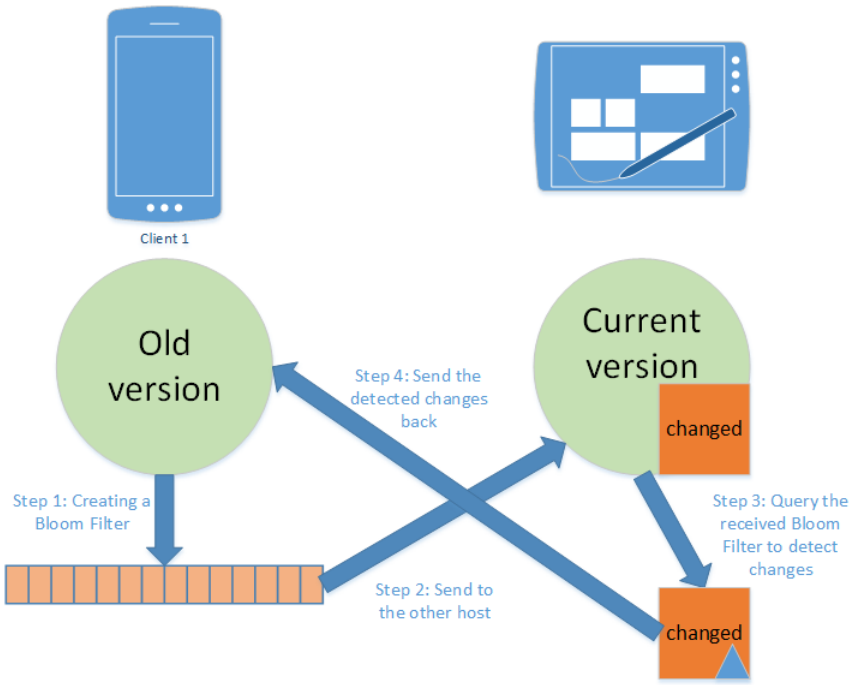

Figure 3.14: Approximate Synchronization Using Bloom Filter

3.6.2 Approximate Reconciliation Tree

Using Bloom Filter leads to a simple algorithm with good result, however, in [9], the authors proposed a data structure, called Approximate Reconciliation Tree (ART), based on Bloom Filter, Patricia Tries and Merkle Tree for better results. When using Bloom Filters for approximate synchronization, the false positive of a Bloom filter is f, then using a corresponding ART will reduce the false positive to:

$p_x = (1 - f)^d$

In the above equation, d is the depth of the ART in which the current element is being examined.

3.7 Set Symmetric Difference Size Estimation

Given two sets S_A and S_B, the symmetric difference of them is defined as:

$S_A \Delta S_B = (S_A - S_B) \cup (S_B - S_A)$

Both previously discussed approaches to the exact set reconciliation problem, namely CPISync (synchronization algorithm based on CPI) and IBFSync (synchronization algorithm based on IBF), need to know before hand the size of symmetric difference of the two sets. Unfortunately, in many cases and especially in our context of key-value storage systems synchronization, this information is not available. Thus, before being able to apply one of these two algorithms, we need to estimate the size of $S_A \Delta S_B$.

There are many existing approaches to estimate the symmetric difference of two remote sets, notably one-by-one, exponential guessing, strata estimator and using Bloom Filter.

3.7.1 One-by-one Method

This method uses multiple rounds of communication. The client uses an initial value for the symmetric difference and then increases it by a pre-defined number d after each round. In each round, the synchronization algorithm detects if it has successfully run, if not, the estimated symmetric difference increases by d.

Since this method requires multiple communication rounds, it is not suitable in high latency environments.

3.7.2 Exponential Guessing

This method also uses multiple rounds of communication, starting with the smallest possible symmetric difference and then increasing it each round according to some pre-defined rule. Most commonly is to multiply it by 2 after each round, however depending on the implementation, different factor can be used. In each round, it uses the failure-detection mechanism of the corresponding algorithm (CPI or IBF) to determine if it has successfully completed the algorithm or not. If not, then it would re-run the algorithm with an increased estimated symmetric difference.

This solution requires sending CP or IBF of the set O(log d) times, and since it consumes multiple rounds of communication, it suffers from the same drawback as the one-by-one method: unsuitable for environments with high latency.

3.7.3 Strata Estimator

The Strata Estimator[18] uses IBF to estimate the symmetric difference between two sets. The algorithm works as follow:

1. Denote U as the universe of the set values, $|U|$ is the size of U
2. Apply a hash function H_z to each elements to assure the mapped values in S_A and S_B are uniformly distributed throughout U.
3. Stratify U into L = $log_2(|U|)$ partitions, $P_0, P_1...P_L$, such that partition P_i covers $1/2^{(i+1)}$ of U.
4. Create L IBFs, each IBF contains items belonging to the corresponding partition

The algorithms for encoding and decoding strata estimator are given below.

for $s \in S$ **do**
    i = number of trailing zeros in $H_z(i)$;
    Insert s into the i_th IBF;
end

Algorithm 2: Strata estimator encoding using hash function H_z

count = 0;
for $i = log_2(|U|)$ *down to -1* **do**
  if $i < 0$ *or* $IBF_1[i] - IBF_2[i]$ *does not decode* **then**
    return $2^{i+1} * count$
  else
    count += number of elements in $IBF_1[i] - IBF_2[i]$
  end
end

Algorithm 3: Strata estimator decoding

3.7.4 Using Bloom Filter

In [36], the authors proposed the following equations, called the *quasi-intersection* method, to estimate the size of the symmetric difference between S_A and S_B using Bloom Filters:

$|S_A \cap S_B| \approx |S_B| - \frac{|S_B|-n_0}{1-(1-e^{-k|S_A|/m})^k}$

Thus:

$d_0 \approx |S_A| - |S_B| + 2\frac{|S_B|-n_0}{1-(1-e^{-k|S_A|/m})^k}$

In which:

- n_0 is the number of elements in S_B which are considered to possibly exist in S_A by querying $BF(S_A)$
- K is the number of hash functions used in $BF(S_A)$
- m is the number of bits in $BF(S_A)$
- d_0 is the estimated symmetric difference between S_A and S_B

Note that $(1 - e^{-k|S_A|/m})^k$ is the approximation of the false positive rate of $BF(S_A)$. However, in [12], it has been shown that this formula has a large error margin for small Bloom filters. Therefore, we used the original equation of calculating false positive:

$p_{false} = (1 - (1 - \frac{1}{m})^{kn})^k$

in our implementation. Therefore, the equations become:

$|S_A \cap S_B| \approx |S_B| - \frac{|S_B|-n_0}{1-p_{false}}$

$d_0 \approx |S_A| - |S_B| + 2\frac{|S_B|-n_0}{1-p_{false}}$

The authors showed that this method is more efficient than one-by-one and exponential guessing method in [36].

3.8 Network Protocols

In this section, we take a look into applicable networking protocols for transferring data in a peer-to-peer network environment.

Note that although the system works as a whole in a peer-to-peer environment, each synchronization happens under a scenario which can be viewed as a client-server architecture: a server holding the current version of the database and the client trying to update its outdated database to that current version.

The following sections will details some of the most popular protocols for transferring data.

3.8.1 Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) [19] is an application protocol primarily used for serving webpages. However, it has multiple characteristics that make it appealing for our problem:

1. It's standard and ubiquitous, with support on practically every platform (Windows, Linux, MacOS, Android, iOS, Windows Phone...)
2. It can be used to transfer binary data, not just text.
3. Standard responses for different error scenarios.

Using HTTP would make it possible for our solution to implement a RESTful [20] service for database synchronization on each device, and that would make it very easy to interoperate with other systems. However, it also involves multiple drawbacks:

1. Bigger overhead than necessary. Each HTTP request must include a verbose header.
2. Using it for transferring binary data is possible, but since it's originally designed for text, transferring binary data is harder than other protocols designed for binary data.

The overhead of this protocol is significant, below is an example of a standard response from the server:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 3833
Accept-Ranges: bytes
Server: HFS 2.3
Set-Cookie: HFS_SID=0.811487414175645; path=/;
Cache-Control: no-cache, no-store, must-revalidate, max-age=-1
```

Verbose headers make using HTTP in our solution unnecessarily inefficient. Therefore, we decided against using HTTP.

3.8.2 File Transfer Protocol

File Transfer Protocol(FTP) [29] is a popular protocol for transferring files over a TCP-based network, such as a local LAN or the Internet. It supports authentication, normally with a username and password, however it can also operate anonymously. The protocol has extensive data representation formats:

1. ASCII mode
2. Image or binary mode
3. EBCDIC mode
4. Local/proprietary mode

There are also multiple modes for data transferring:

1. Stream mode
2. Block mode
3. Compressed mode

Similar to HTTP, FTP is well supported by multiple platforms. In addition, it is more efficient since the header overhead is smaller. We chose FTP as the data transfer protocol for our current implementation. In detail, we use binary mode for data representation and stream mode for data transferring since they are the most suitable settings for our data.

FTP is notably appropriate for our problem since each transferring data phase of our algorithm can be modeled as a file transfer problem.

3.8.3 Raw TCP

This approach uses the raw TCP connection [28] to send the data, without invoking any higher level protocol. The most significant advantage is this method has the smallest overhead, since we can implement the protocol specifically for our use cases, not for the bigger general use cases HTTP or FTP was designed for. Because of this, we can eliminate unnecessary features from the protocol and make it more efficient.

The disadvantage of this method is since it is not a standard protocol, we need to implement it by ourselves. We decided against this method due to the time limit, however, this method has the potential to become the most efficient data transfer protocol possible.

3.9 Summary

This chapter describes numerous approaches in three key areas of our problem: the synchronization algorithms, the symmetric size difference estimation algorithms, and possible protocols for transferring data over the network.

This chapter describes some notable approaches to the key-value storage systems synchronization problem. Those algorithms are:

1. Naive or server centric algorithm
2. Log-based
3. Using timestamps
4. Using Characteristic Polynomial
5. Using Invertible Bloom Filters
6. Using Bloom Filter in conjunction with Hash Tries
7. Approximate synchronization using Bloom Filter
8. Approximate synchronization using Approximate Reconciliation Tree

Each algorithm has its own unique properties regarding processing time, communication cost, synchronization time. However, these solutions can be classified into the following major categories:

1. According to the number of communication round
    - One round or limited round: naive approach, log based, CPI or IBF with estimated symmetric difference, approximate reconciliation tree
    - Unbounded round: CPI or IBF with exponential guessing
2. According to the accuracy of the algorithm
    - Exact synchronization: naive, log-based, CPI, IBF
    - Approximate synchronization: using bloom filter, approximate reconciliation tree
3. According to the usage of prior context, or meta data:
    - Using prior context: log based, timestamps
    - Not using prior context: naive, CPI, IBF, both approximate synchronization methods (using Bloom Filter and Approximation Reconciliation Tree)

In addition, this chapter also describes some common methods to estimate the symmetric difference between two remote sets, which is essential for the algorithms of CPI and IBF to work. These methods are:

- One-by-one: increase the estimated value by a pre-defined number d after each time.
- Exponential guessing: using multiple rounds of communication, increase the guessed number of difference each time until we get the acceptable values.

- Strata Estimator: using multiple invertible Bloom Filters to estimate.
- Using Bloom Filter: using just one Bloom Filter to estimate.

Finally, we discussed different ways of transferring data over the network, including HTTP, FTP and raw TCP. We see them under their applicability to our problem, and FTP was chosen as it provides some of the most useful benefits while the drawbacks are negligible.

Although the above mentioned approaches provide a number of solutions to our problem of key-value storage system synchronization, each method has its own drawbacks and can be improved upon. Examples include significant storage cost for methods using prior context such as log based or timestamps, or long processing time for CPI and IBF in case of big difference between the old version and the current database version. In the next chapter, we describe a new solution which addresses a lot of the flaws found in these existing methods, while maintaining good performance characteristics in communication cost and processing time.

Chapter 4

Arichitecture Design

4.1 Discussion of Existing Approaches

There are multiple approaches to the data synchronization problem, however, they differ in many key areas and thus, can be categorized into 3 main categories. Each category has its own pros and cons, the major ones are shown in Table 4.1 below.

Table 4.1: Pros and cons of synchronization algorithm categories

Category	Sub-category	Pros	Cons
Communication rounds	Limited	Lower latency	Potential bigger communication cost
	Unbounded	Potential lower communication cost	Higher latency
Accuracy	Approximate	Faster	Limited applicability
	Exact	Wider range of applicable scenarios	Slower
Usage of prior context	Yes	Optimal communication cost	Performance penalty for write operations
	No	No extra data needed	Bigger communication cost

In order to have the widest range of applicable scenarios, we propose an algorithm conforming to these characteristics:

- Number of communication rounds: using a limited number of communication rounds to behave better in high latency peer-to-peer network environment.
- Accuracy: exact synchronization for better applicability.
- Usage of prior context data: not using any prior context data to maintain high performance for the database.

Notable algorithms which have these 3 characteristics are the server centric (naive) approach and IBFSync (to a lesser extend, CPISync too with a limited round estimator). Both CPISync and IBFSync have very good performance in term of communication cost, achieving the theoretical limits. The differences are:

- CPISync is more demanding in term of computing power, mostly due to the factoring polynomial over finite field algorithm.
- IBFSync have bigger communication cost, due to the overhead of the IBF data structure

4.2 Proposed Solution

In this section, we describe our new algorithm for synchronizing key-value database. We call our algorithm ASync. Figure 4.1 shows the position of our algorithm in a complete software stack.

Starting with the problem, the problem of key-value database synchronization can be divided into 2 steps: reducing to the set reconciliation problem, and synchronization of the two remote sets. The first step is rather trivial, and will be described in detail later in Section 4.3.

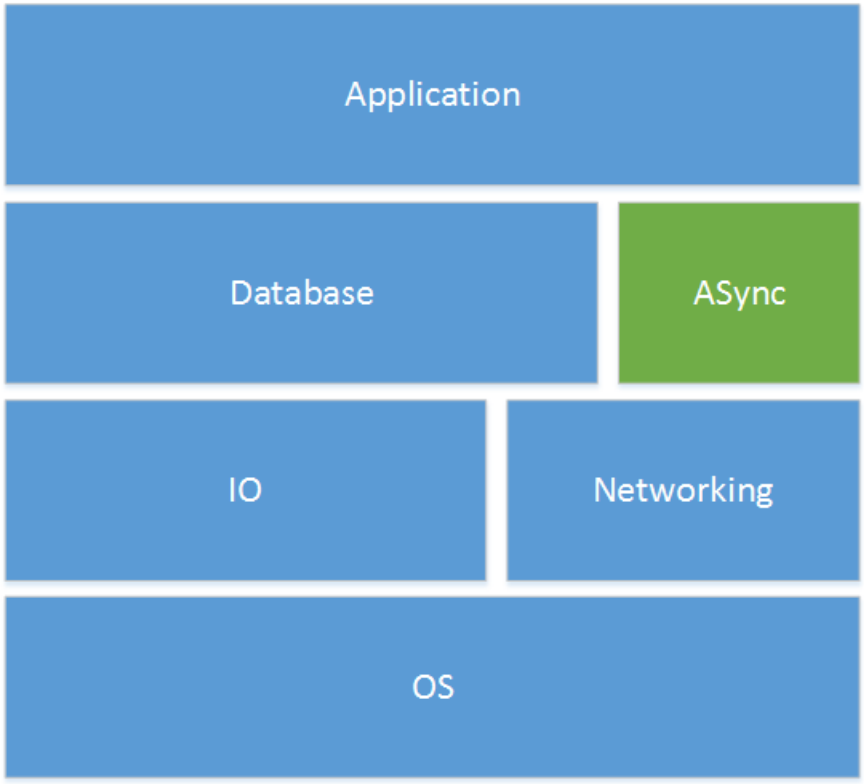

Figure 4.1: ASync Component

We will now focus on the second step: solving the set reconciliation problem, which is the main goal of ASync. In this research, we propose a 2-phase solution for the synchronization problem, the first phase is to do the approximate synchronization using Bloom Filter, and the second phase is to do exact synchronization

using Invertible Bloom Filter. The key idea of the algorithm is in the first phase, we could use one Bloom Filter to achieve multiple goals:

- Estimate the symmetric difference of the two sets in preparation for the next phase. This information is crucial for the next step to operate.
- Approximate synchronization, which in itself is greatly beneficial: reducing the number of different items of the two sets would make the next phase more efficient, since the size of the IBF depends upon the size of the symmetric difference between the two sets.

The algorithm contains 2 main phases and a preparatory phase:

1. Phase 0: reduce the problem to the problem of set reconciliation
2. Phase 1: approximate synchronization and estimation of the size of symmetric difference between two remote sets
3. Phase 2: exact set reconciliation based on IBF

Figure 4.2 demonstrates the sequence of our algorithm.

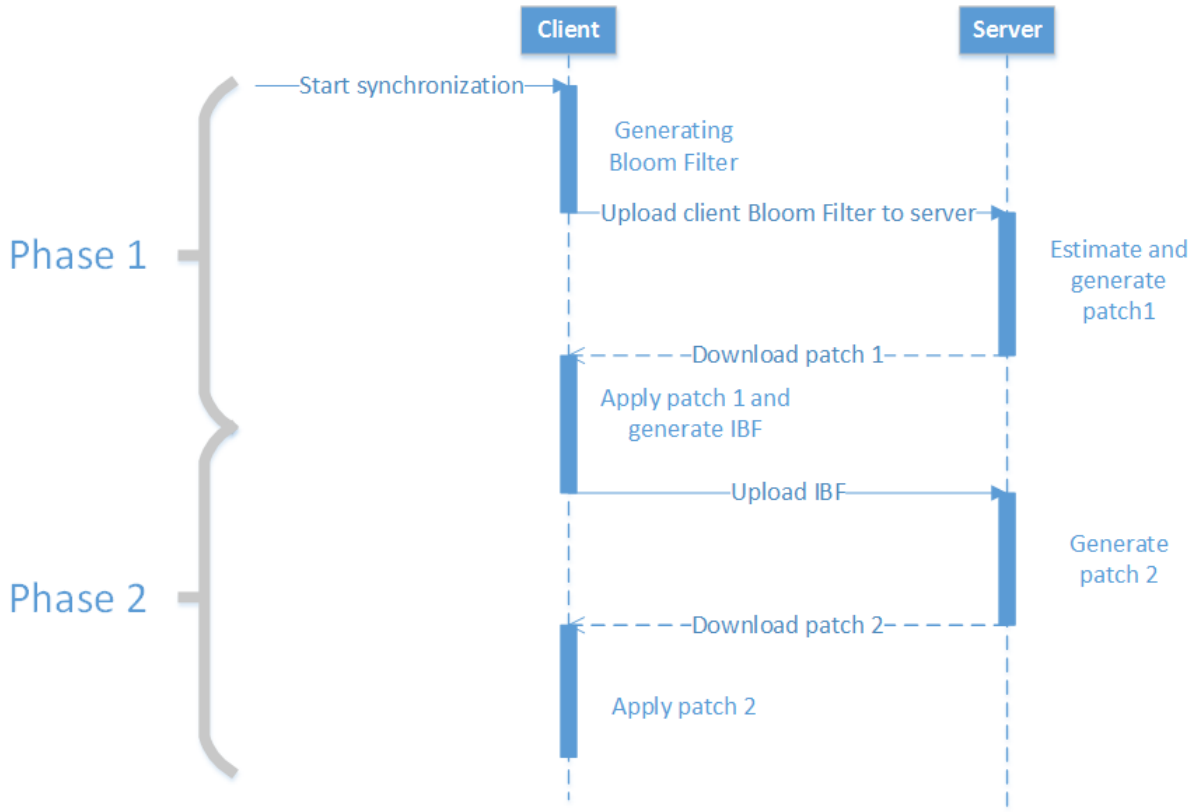

Figure 4.2: ASync Sequence

The following sections will discuss each phase in detail.

4.3 Phase 0: Reducing to the Set Reconciliation Problem

4.3.1 Mapping Data Items to Identifiers

This phase is the preparatory phase, it does not carry any real data synchronization, however, it lays out the foundation for the next steps as it converts our problem to the simpler set reconciliation problem.

The idea is to convert all key-value data items to identifiers, thus essentially reducing a database of key-value data items to a set of identifiers. In order to do so, a mapping function is needed. This function must take an arbitrary key-value data item as the input and produce an output suitable to used as an identifier. In our implementation, we use 64-bit integer numbers as our identifiers, so the mapping function should output a 64-bit integer number for each key-value data item.

In our implementation, we use a hash function to execute this mapping mechanism.

input : key-value data items in the database

output: identifiers

for *i in 0..n-1* **do**
| $iden[i] = H(keys[i], values[i])$

end

Algorithm 4: Mapping key-value data items to identifiers

The criteria for choosing a hash function for our solution are:

- Speed: as the hash function must be run for every single data item in the database, to keep the speed of the whole algorithm fast, the hash function must be fast.
- Good random distribution: to avoid having the same hash value from two different data items.

We considered two famous hash functions: MD5 and SHA1 for our algorithm, but since we do not need the cryptography features from them, they are slower than necessary. Instead, we choose Murmurhash [3] for our solution, since it provides very good speed and have good random distribution.

4.3.2 Size of Identifiers

We now consider what size is sufficient for identifiers. Since we use a hash function to convert our data items to identifiers, we want to minimize the risk of hash collision as much as possible. We have the probability of hash collision for k N-bit hash values as [2]:

$1 - e^{\frac{-k(k-1)}{2N}}$

With a 32 bit value, 77163 values would have a 50% probability of having hash collision, which is unacceptable for our problem since we want to work with databases containing millions of values. Table 4.2 [2] shows the number of elements corresponding to the probability of a hash collision for each size of hash values.

As a result, we chose 64 bit length for our identifiers.

Table 4.2: Hash collision probability[2]

Odd of a hash collision	32 bit	64 bits	160 bits
1 in 2	77163	5.06 billion	$1.42 * 10^{24}$
1 in 10	30084	1.97 billion	$5.55 * 10^{23}$
1 in 100	9292	609 million	$1.71 * 10^{23}$
1 in 1000	2932	192 million	$5.41 * 10^{22}$
1 in 10000	927	60.7 million	$1.71 * 10^{22}$
1 in 100000	294	19.2 million	$5.41 * 10^{21}$
1 in a million	93	6.07 million	$1.71 * 10^{21}$
1 in 10 million	30	1.92 million	$5.41 * 10^{20}$
1 in 100 million	10	607401	$1.71 * 10^{20}$
1 in a billion		192077	$5.41 * 10^{19}$

4.4 Phase 1: Approximate Synchronization and Symmetric Difference Size Estimation

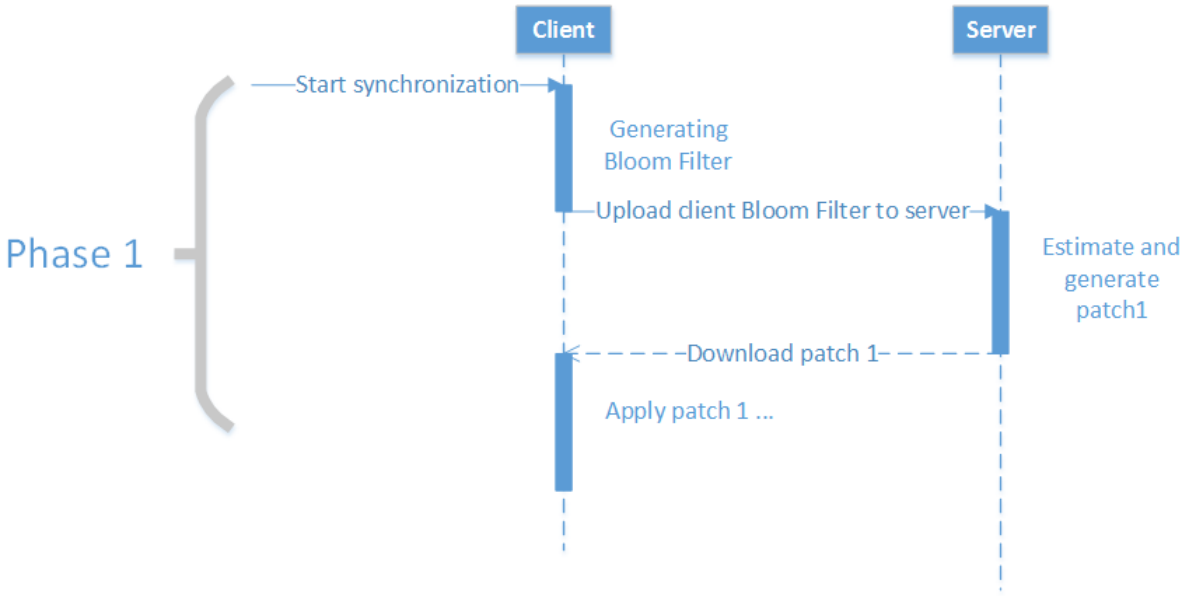

Figure 4.3: ASync - Phase 1

In this phase, we use Bloom Filter to achieve two goals at the same time:

- Approximate synchronization: we detect the majority of the differences between the two sets by using

Bloom Filter, and synchronize the data using this knowledge. This also helps to reduce the data needed for the next phase, exact synchronization.

- Estimate the size of the symmetric difference between two remote sets: this information is required for the next phase to successfully run.

In order to do so, first we create a Bloom Filter from the set of identifiers in the client.

input : set of identifiers
output: Bloom filter containing these identifiers
for *i in 0..n-1* **do**
| *AddToBloomFilter*($S[i]$)
end

Algorithm 5: Creating Bloom filter from identifiers

In our implementation, when create a Bloom Filter from the set of identifiers in the client, we use the following parameters:

- Size in bits: $m = |S_{client}| * 12$

- Number of hash functions: $k = 5$

These parameters give us a Bloom Filter with $p_{false} \approx 0.5\%$ [1], which is acceptable for our purpose.

In the server, the Bloom Filter received from the client is used to detect difference in the two sets with allowable false positives:

input : Bloom filter of client identifiers
output: Set of identifiers not present in client
for *i in 0..n-1* **do**
| **if** *BloomFilterContains(S[i])* **then**
| | continue
| **else**
| | AddToMissingIdentifiers(S[i])
| **end**
end

Algorithm 6: Detect missing identifiers from the client

And finally, the last objective of this phase is to estimate the size of the symmetric difference between two sets. We use the equation described in [36], which was outlined in section 3.7.4 of this thesis.

input : $|S_A|$, $|S_B|$, n_0

output: d_0 - estimated symmetric difference

$d_0 = |S_A| - |S_B| + 2\frac{|S_B|-n_0}{1-p_{false}}$

Algorithm 7: Estimate symmetric difference

4.5 Phase 2: Exact Synchronization

After getting the estimated number for the size of set symmetric difference, we can continue to phase 2 of the algorithm: exact synchronization.

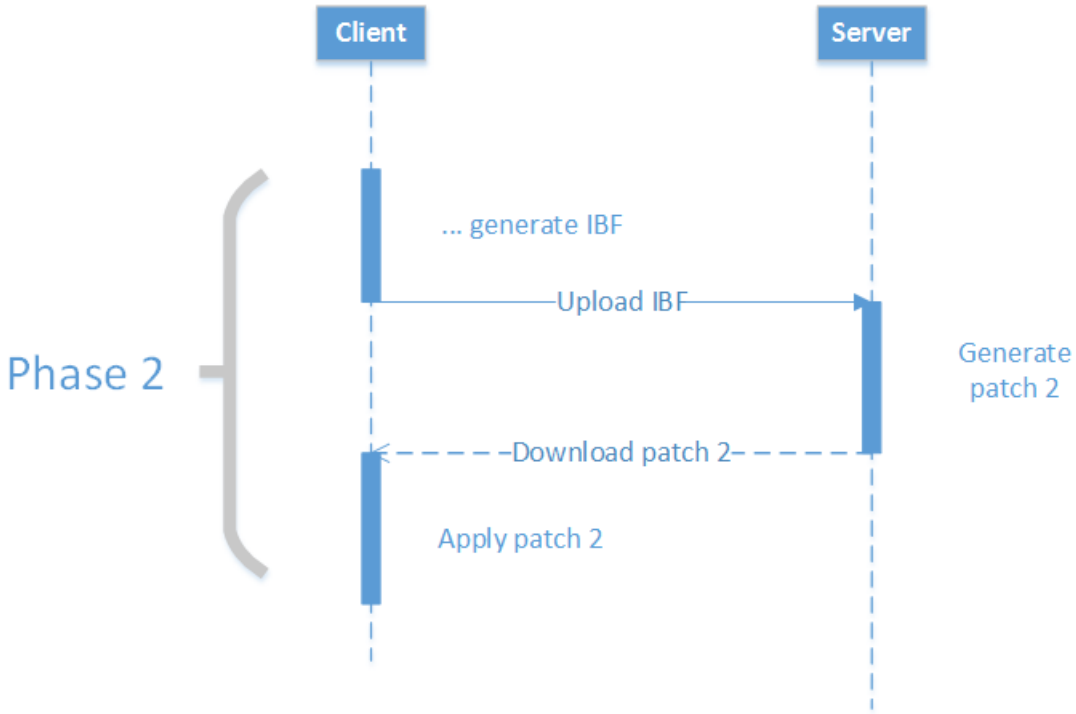

Figure 4.4: ASync - Phase 2

We use Invertible Bloom Filter (IBF) for this phase, as it provides fast computation time and optimal theoretical communication cost.

We create the IBF in the client with the following parameters:

- Length sufficient to hold d_0 elements (the IBF should have approximately $1.5d_0$ cells[18])
- Number of hash functions: $k = 3$
- H_c function for calculating HashSum: MurmurHash3 with Seed = 123456789, producing 32-bit hash values.

Algorithm 8 outlines steps to create an IBF in the client:

input : set of identifiers

output: Invertible Bloom filter containing these identifiers

for *i in 0..n-1* **do**
| $AddToIBF(S[i])$

end

Algorithm 8: Creating an Invertible Bloom filter from identifiers

The created IBF is then serialized and sent to the server. The server, after receiving the IBF from the client, creates its own IBF and the performs the subtraction operation on the two IBFs in order to detect the differences between the two sets of identifiers.

After decoding the result IBF, we now have the set of identifiers that are on the server, but not on the client. We now do a reverse lookup to determine which key-value data item correspond to each identifier, and send these data items to the client.

4.6 Other Implementation Details

4.6.1 Delete Operation

Due to the nature of the set reconciliation problem, handling delete operations is tricky. Naive implementations of set reconciliation algorithms will mistakenly identify deletion as addition of the other set and thus, do not correctly synchronize the two sets.

In our system, we sidestep this problem by using a marking mechanism to signify deleted data items. In the database, instead of truly deleting the data item, we write a special data value, "DELETED". Essentially, we convert delete operation to modify operations.

4.6.2 Handling Failures

There are scenarios in which the proposed algorithm could not synchronize the two databases after the above mentioned 2 steps. They are:

- Case 1: wrong estimation of the symmetric difference in phase 1, leading to insufficient information to complete phase 2.
- Case 2: In phase 2, there is a chance the IBF could not successfully yield all of its added items.
- Case 3: Hash collision makes the algorithm synchronize the wrong data.

For each error scenario, we have sufficient detection mechanisms and counter-measures. For detection:

- Case 1: detect on the server as IBF fails to decode.

- Case 2: similar to case 1, IBF fails to decode.
- Case 3: using a hash value for the whole database to compare the final result.

After detecting an error has occurred, we have 2 choices for recovery: (1) re-run the phase 2 with the added information (increase the length of IBF) or (2) fall back to transferring the whole database from the server to the client.

The current version of ASync only implements recovery mechanism 2, which means we have to transfer the whole database from the server to the client when ASync fails to synchronize in phase 2. However, we do not face any error during our experiments in Chapter 5.

4.7 Comparison

4.7.1 With IBFSync

Our proposed algorithm is similar to IBFSync, both algorithm uses IBF as the building blocks for synchronization. The major difference lies in the first step: while we combined approximate synchronization and estimation of the size of set symmetric difference in one phase, the original IBFSync algorithm only utilize the first phase to estimate using Strata estimator.

Figure 4.5 shows the sequence of the original IBFSync algorithm.

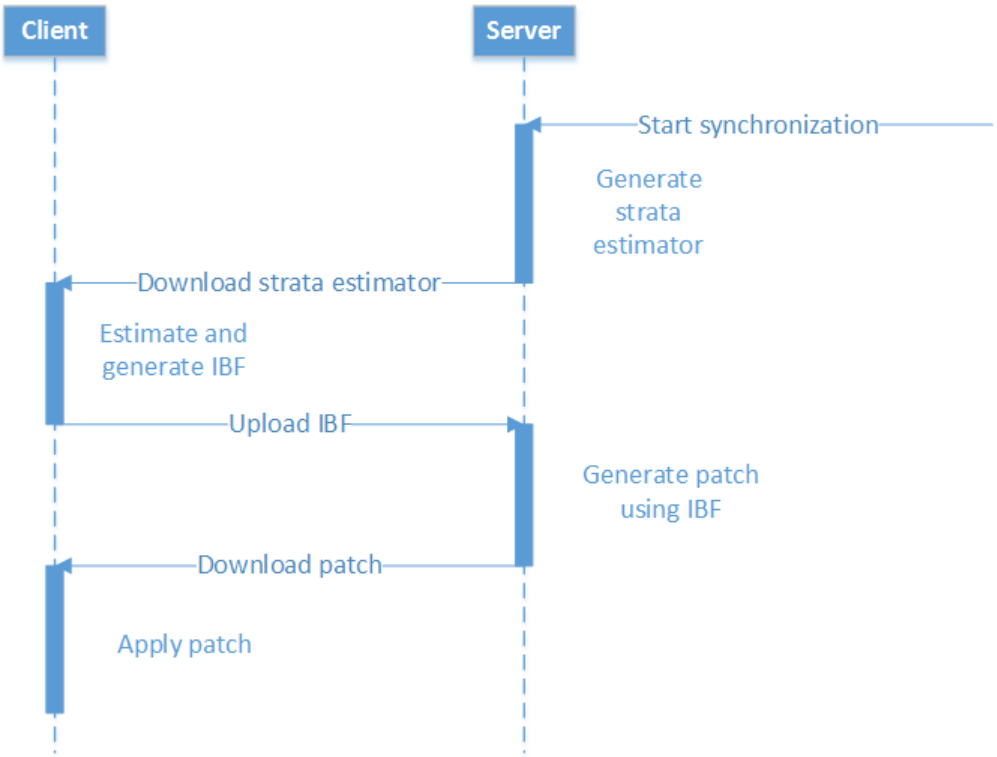

Figure 4.5: IBF Sequence

Since we are doing more in the first step, the data required for our algorithm's first step is much bigger than IBFSync. However, as will be shown in the experiments, that would help us reduce the needed data in the second phase and for the whole synchronization process.

4.7.2 With Bloom Filter and Hash Tries

The main difference between ASync and BFHT (Bloom Filter and Hash Tries) is in phase 2: when ASync uses IBF to detect the difference, BFHT uses hash trie. And as a result, BFHT uses multiple rounds of communication in order to accomplish synchronization. Figure 4.6 shows phase 2 of the BFHT algorithm.

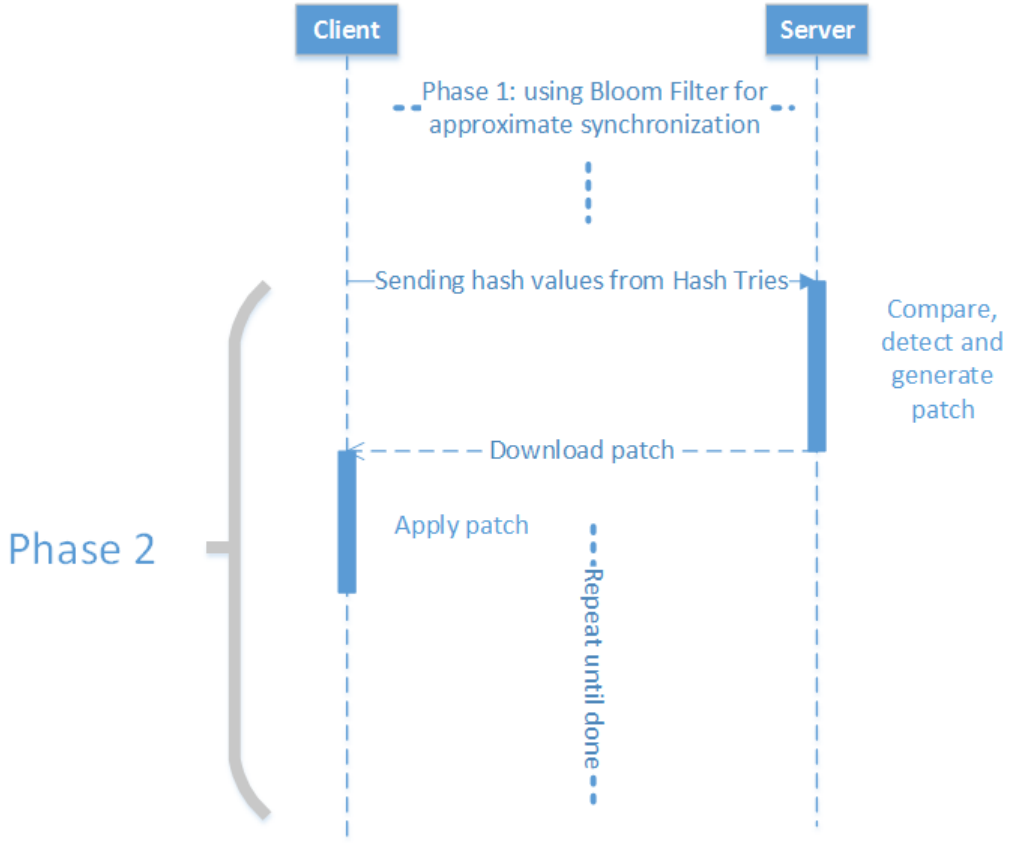

Figure 4.6: Bloom Filter and Hash Tries Synchronization - Phase 2

The first phase is pretty similar on both algorithm: employing Bloom Filter for approximate synchronization. However, ASync not only uses Bloom Filter for approximate synchronization, but we also get an estimation of the size of symmetric difference by using Bloom Filter.

4.8 Applicable Scenarios

The proposed algorithm can work on any key-value storage system. However, due to its characteristics, applications which conform to the following criteria are the most suitable ones:

1. Need to store a local key-value database which acts as a cache of a bigger database.
2. The database is frequently changed.
3. There is a method to detect which version of the database is the newer one.
4. Need to do synchronization with a relatively low speed network. This is due to the fact that our algorithm basically trades processing time for communication cost.

Considering all these criteria, an example of an application for our algorithm is the Wikipedia mobile application. Many mobile Wikipedia applications currently consist of a thin UI layer to access the Wikipedia website, thus they require Internet connection in order to work. One way to make the application work without Internet connection is to store the whole database locally. The database of a mobile Wikipedia application can be a key-value storage system with the keys being the article title and the values being the content. However, the storage requirement may be quite large, as of January 29, 2014, the compressed form of all page in Wikipedia English consume 9.7GB [4].

One obvious observation can be made is the most accessed article are more likely to be requested by the user and only amount for a relatively small storage space. Hence, it is acceptable for the mobile application to only store these most viewed articles, for example the top 5% accessed articles. As a result, the application can consume less storage space, provide content to users in the majority of cases, and does not require Internet connection to work.

The need for synchronization arises when we need to update our locally stored content to the last version, since the content of Wikipedia articles are frequently changed. A version number can be used to keep track of which version is the newer one, and thus detecting which device has the newer version. With our proposed algorithm, the mobile application can synchronize its content with the server as well as with any peer in its local network.

Similar to Wikipedia, many dictionary application could find our synchronization algorithm useful. Other possible applications including scoring boards for games, content archiving application, etc.

4.9 Summary

In this chapter, we categorized data synchronization algorithms based on three major areas:

1. The number of communication round
2. The accuracy
3. The usage of prior context data

We noticed that for our problem of key-value storage system synchronization in peer-to-peer network environment, the most suitable characteristics of a synchronization algorithm are: (1) limited numbers of communication rounds, (2) exact synchronization and (3) do not use prior context data.

We discussed in detail our proposed synchronization algorithm, ASync. The algorithm has 2-phase architecture in addition to a preliminary phase, with the role of each phase as follow:

1. Phase 0: converting to the set reconciliation problem
2. Phase 1: approximate synchronization and estimate the size of symmetric difference
3. Phase 2: exact set reconciliation based on Invertible Bloom Filter

In addition, we described some specific implementation details and the reasons behind them. In addition, we compared our proposed algorithm with IBFSync and BFST, pointed out the differences and explained why these differences should help us achieve better performance in synchronization. Finally, this chapter gives some example of potential applications that could employ our new algorithm.

Chapter 5

Experiments and Results

In this chapter, the proposed algorithm is evaluated with regard to the challenges stated in Chapter 2. The experiments aim to study the performance of the proposed algorithm under different scenarios. The experiment analysis and evaluation are to help demonstrate algorithm's feasibility in various situations also to identify the best performing scenarios.

5.1 Metrics

To understand the proposed algorithm and how it compares with other algorithms serving the same purpose, first we need to establish a number of metrics to quality data synchronization algorithms.

1. Message size: the communication cost of the algorithm, measured in how many bytes that have to be sent over the network to complete the synchronization process.

2. Processing time: the time it takes for the client and the server to do the calculations required for the algorithm. Measured in seconds.

3. Synchronization time: the total time it takes for the outdated database in the client to be fully synchronized with the current database in the server. Measured in seconds.

Out of the above mentioned three metrics, the most important metric is the synchronization time. It is roughly equal to the sum of the processing time and the time it takes to transfer data over the network. The message size, also known as the communication cost, shows how the algorithm behaves in different network environments. In a fast network, this metric is less important and vice versa, the slower the network is, the more important this metric becomes. Finally, the processing time metric is to show how the algorithm runs on different hardware.

The three above mentioned metrics are not only limited to key-value database synchronization, but they are generally applicable to evaluate any data synchronization algorithm.

5.2 Experiment Setup

5.2.1 Datasets

Characteristics

We begin with the original databases, located on the client. We classify our databases into 3 categories:

1. Small database: 100,000 key-value items, the total size of the database is around 10MB.

2. Medium database: 500,000 key-value items, the total size of the database is about 50MB.

3. Big database: 1,000,000 key-value items, the total size of the database is about 100MB.

We also have 3 categories of changes:

1. Small change: 4% of the total key-value data items were changed

2. Medium change: 20% of the total key-value data items were changed

3. Big change: 50% of the total key-value data items were changed

The changes in our candidate databases were half modifying changes and half adding changes. That means when we say 1 database has 20% of its data changed, 10% of the items have been modified and 10% new items have been added. Figure 5.1 demonstrates the changes in our candidate databases.

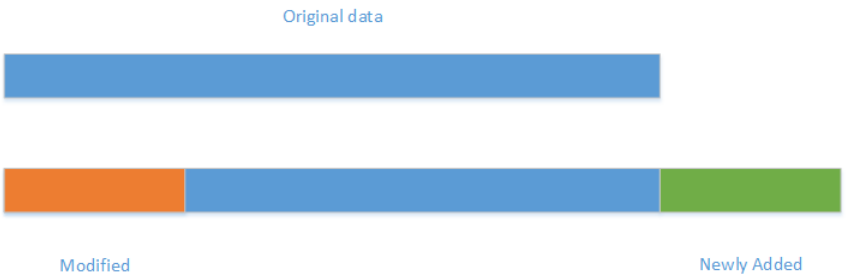

Figure 5.1: Testing Databases

This is to better emulate database in a real working environment, when changes to the database mainly come from adding new items and modifying existing items. Note that we treat deleting the same as modifying, see Section 4.6.1 for details.

The sizes of the client databases and the server databases in each experimental scenario are shown in Table 5.1.

Table 5.1: Dataset sizes

Number of data items	Change percentage	Client size (in bytes)	Server size (in bytes)
100,000	4%	10,488,890	10,700,890
	20%	10,488,890	11,548,890
	50%	10,488,890	13,138,890
500,000	4%	52,888,890	53,948,890
	20%	52,888,890	58,188,890
	50%	52,888,890	66,138,890
1,000,000	4%	105,888,890	108,028,890
	20%	105,888,890	116,588,890
	50%	105,888,890	132,638,890

Generating Datasets

We create our datasets by employing a simple key-value database with key as string and value as string. The value is the string representation of the hash value of the current key using a hash function.

The generating algorithm is shown in Algorithm 9.

input : *num* - number of items in the database

output: Client database

for *i in 0..num-1* **do**

  $key = i.ToString()$

  $val = BitConverter.ToString(hFunc.ComputeHash(BitConverter.GetBytes(i)))$

  database.Add(key, val)

end

Algorithm 9: Database generating algorithm, client

5.2.2 Hardware and Software Configuration

Hardware Configuration

For each experiment, the hardware we use are the following:

1. The server containing the current version of the database will be a tablet-laptop hybrid, specifically it's a Microsoft Surface Pro device.

2. The client trying to update its database will be a mobile phone, in our experiment we will use a Nexus 7 (2013 version) device.

Figure 5.2 shows the hardware and the synchronization scenario used in all test cases.

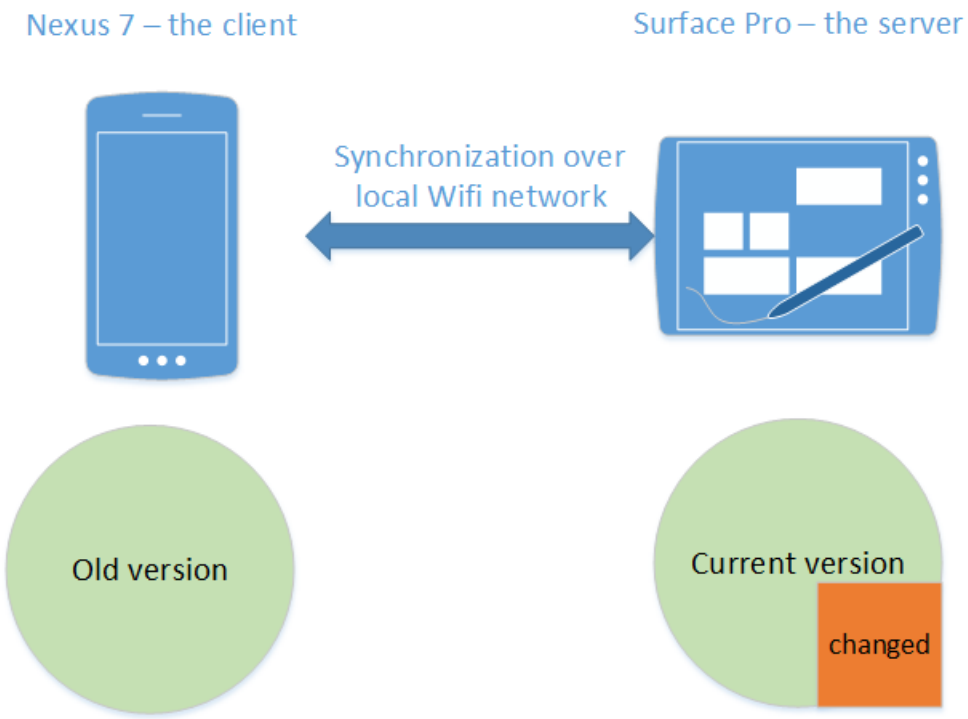

Figure 5.2: System Configuration

The hardware specifications of the Surface Pro are given below in Table 5.2

Table 5.2: Microsoft Surface Pro specifications

CPU	Dual-core 1.7 GHz (Turbo Boost to 2.6 GHz) Intel Core i5-3317U
Memory	4 GB dual-channel DDR3-1600 (25.6 GB/sec)
Storage	128 GB (83 GB available)
Graphics	Intel HD Graphics 4000
OS	Microsoft Windows 8.1 Pro

The hardware specifications of the Nexus 7 are given below in Table 5.3

Table 5.3: Nexus 7 (2013 version) specifications

Manufacturer	Asus
System on chip	Qualcomm Snapdragon S4 Pro APQ80641AA
CPU	1.51 GHz quad-core Krait 300
GPU	400 MHz quad-core Adreno 320
Memory	2 GB DDR3L RAM
Storage	16 GB
OS	Android 4.4.2

For our experiments, we will use the local Wifi network, but the algorithm works similarly over Bluetooth or NFC connection. Figure 5.2 depicts the system configuration used for our experiments.

Software Configuration

The algorithm is implemented using C#. For the server, we use the C# compiler bundled with Visual Studio 2012. For the client, we use Xamarin.Android to compile the C# code for the Android platform.

For all the experiments, the code was compiled under "Release" mode for both the server (Windows platform) and the client (Android platform).

5.3 Comparison and Measuring

In order to evaluate our proposed algorithm, we compared our algorithm with the following existing algorithms:

1. Server-centric approach of copying the whole database from the server to the client algorithm
2. IBFSync with strata estimator

All these 3 algorithms belongs to the same categories in all 3 major aspects, which are:

1. Communication rounds: limited
2. Accuracy: exact
3. Usage of prior context: no

Because of this, the three algorithms can be used interchangeably. It means if an application already uses one of these three algorithms, it can consider switching to another.

For each kind of database and each type of change, we run our algorithm and record the 3 metrics mentioned earlier. The units for each metrics are: bytes for communication cost, seconds for both processing time and synchronization time.

5.4 Results

5.4.1 Communication Cost

Figure 5.3 shows the communication cost for each algorithm in our experiment. We can easily see from the chart that both ASync and IBFSync outperform the naive server-centric approach by a wide margin in all test scenarios. The reason is also obvious: the naive server centric approach has communication cost proportional to the size of the server's database (O(n) complexity), while the remaining two algorithms both have communication cost proportional to the size of the difference between the server's database and the client's database (O(d) complexity).

Note that IBFSync failed to complete synchronization for one dataset (the test case with 1 million data items and 50% change). In our experiment, we only record data when IBFSync can complete the synchronization process in 2 communication round (as it is designed to achieve the majority of the cases), so we do not consider this case for comparison. In practice, this error case would be handled by using more communication round and/or fall back to server-centric method for the next round.

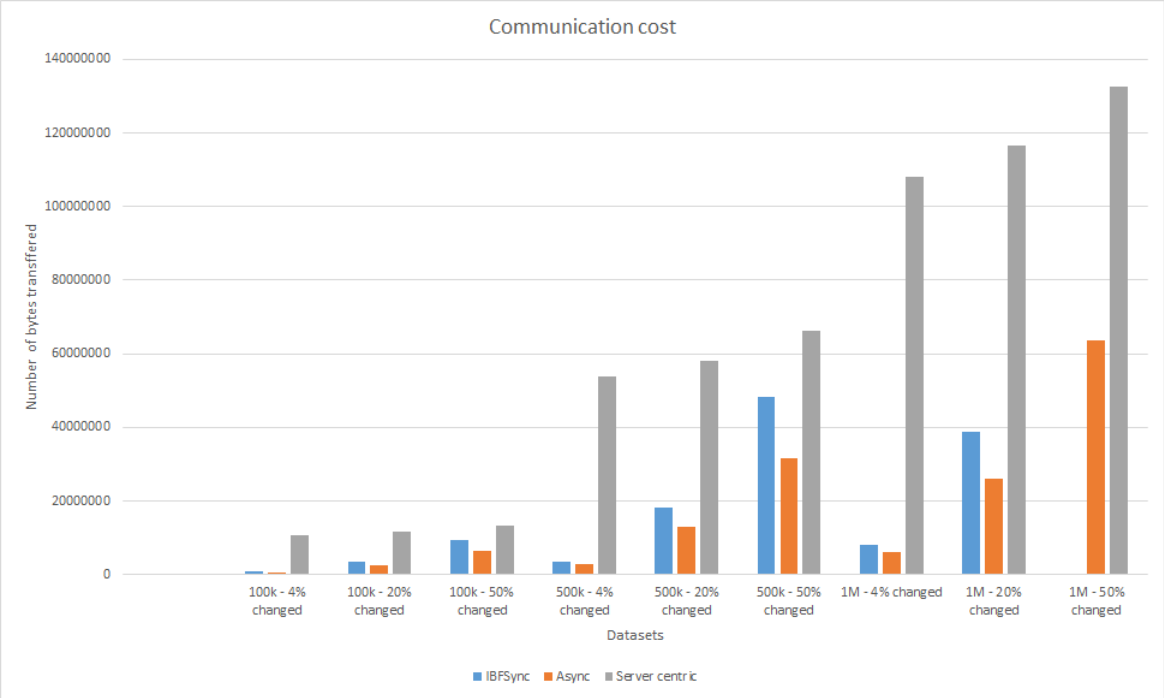

Figure 5.3: Communication Cost

The difference becomes bigger with bigger database size and smaller number of changed items. In our experimental scenarios, the biggest difference comes from the test case with large database size (1,000,000 items) and small change percentage (4%). In addition, in all cases ASync algorithm is better than IBFSync in all tested scenarios.

Figure 5.4 shows in percentage how faster our proposed algorithm is than IBF. We can see that IBF needs more resource in term of network traffic than ASync from around 20% to around 50%. The best gain observed is from the datasets with 500k data items and 20% changed.

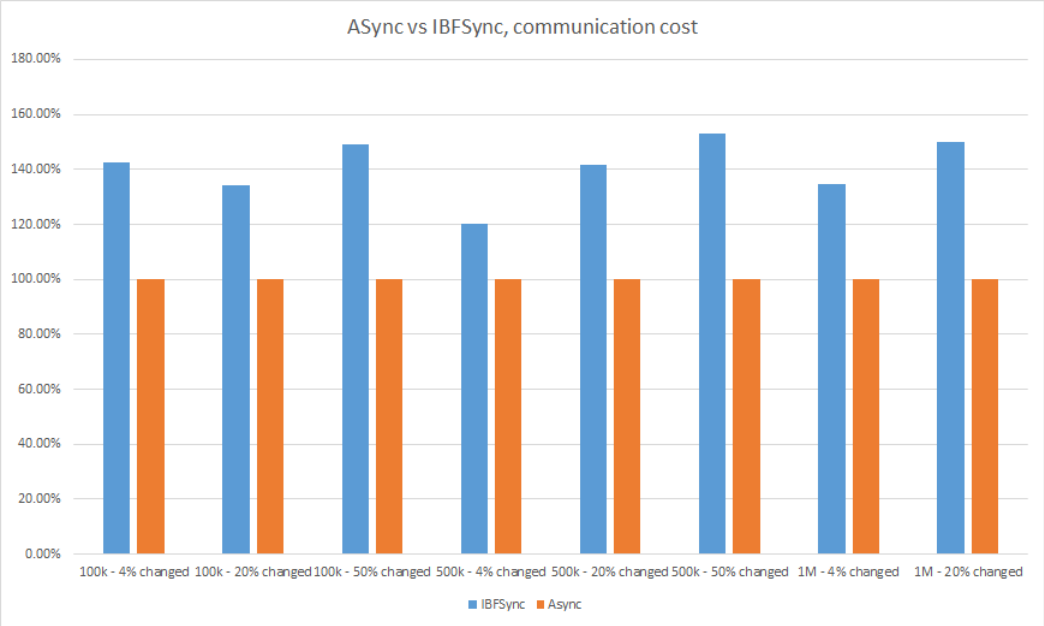

Figure 5.4: Comparison Between ASync and IBFSync, Communication Cost

The saving in communication cost comes from using Bloom Filter as an estimator and doing approximate synchronization during phase 1 of ASync. Since the algorithm does approximate synchronization during phase 1, it reduces the number of changed items after that, thus lowers the size of the Invertible Bloom Filter needed in phase 2.

5.4.2 Processing Time

Processing time is the time it takes for the synchronization algorithm to do its calculation. For the server centric approach, this time is essentially 0. For the other two algorithms, however, the processing time represents a significant portion of the whole synchronization time. As a result, in our chart we only show the processing time for the 2 algorithms, IBFSync and ASync.

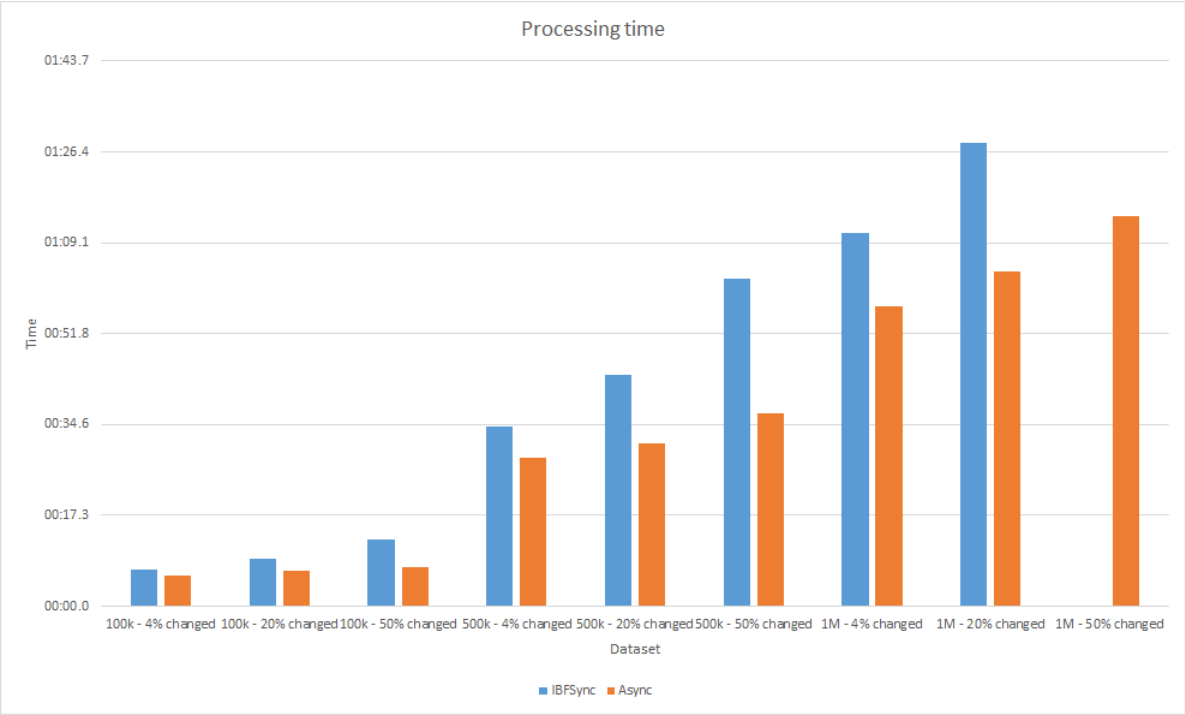

Figure 5.5: Processing Time

Figure 5.6 shows the difference in processing time between ASync and IBFSync. We can see that in all cases ASync needs less time than IBFSync, the difference ranging from 20% to almost 70%.

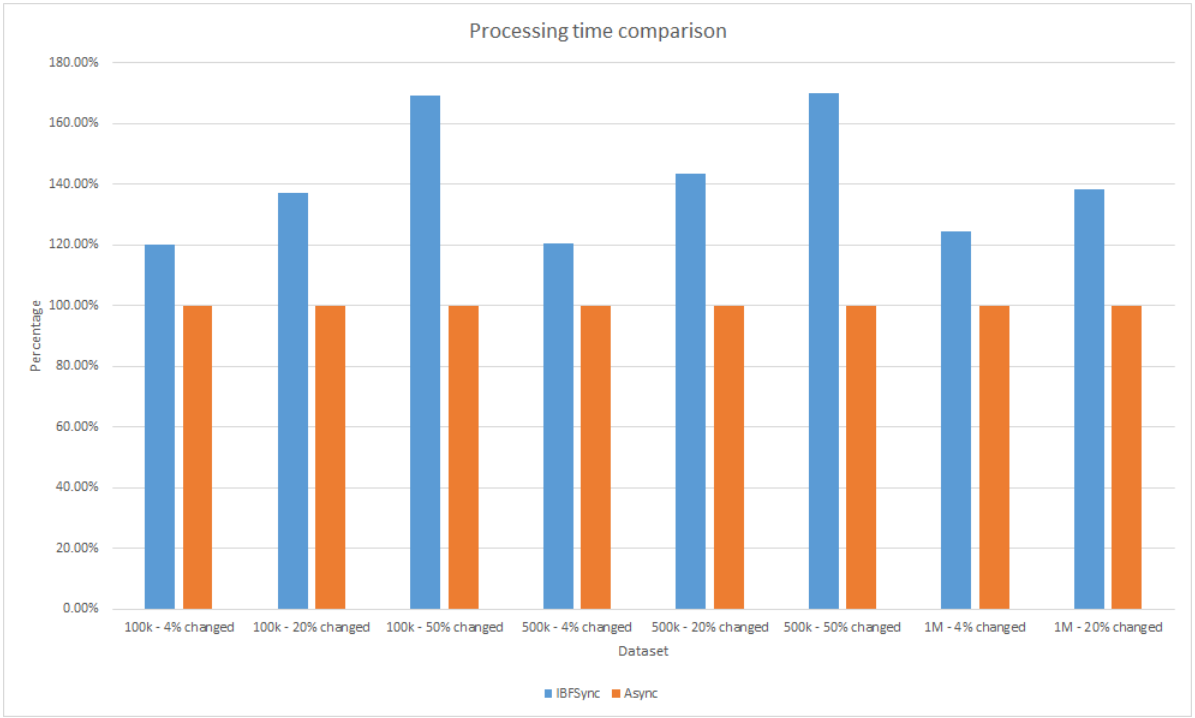

Figure 5.6: Comparison Between ASync and IBFSync, Processing Time

Similar to the reason ASync has better communication cost than IBFSync, the processing time is also lower due to the 2-phase architecture, when the most complex part (creating and querying the IBF) is cut down thank to the approximate synchronization in the first phase.

5.4.3 Synchronization Time

In this section, we take a look at the synchronization time and the synchronization speed of ASync and IBFSync.

Synchronization Time

Synchronization time is the time it takes for the whole synchronization process to complete. It is the total time of the processing time and the data transfer time. This is the most important metric of a data synchronization algorithm.

Figure 5.7 shows the synchronization time for ASync and IBFSync. We do not show the synchronization time for the sever-centric method since it directly depends on the network speed. We can see in the chart that ASync performs better than IBFSync in all test cases. This result is predictable from the above two metrics, since ASync is better than IBFSync in both communication cost and processing time.

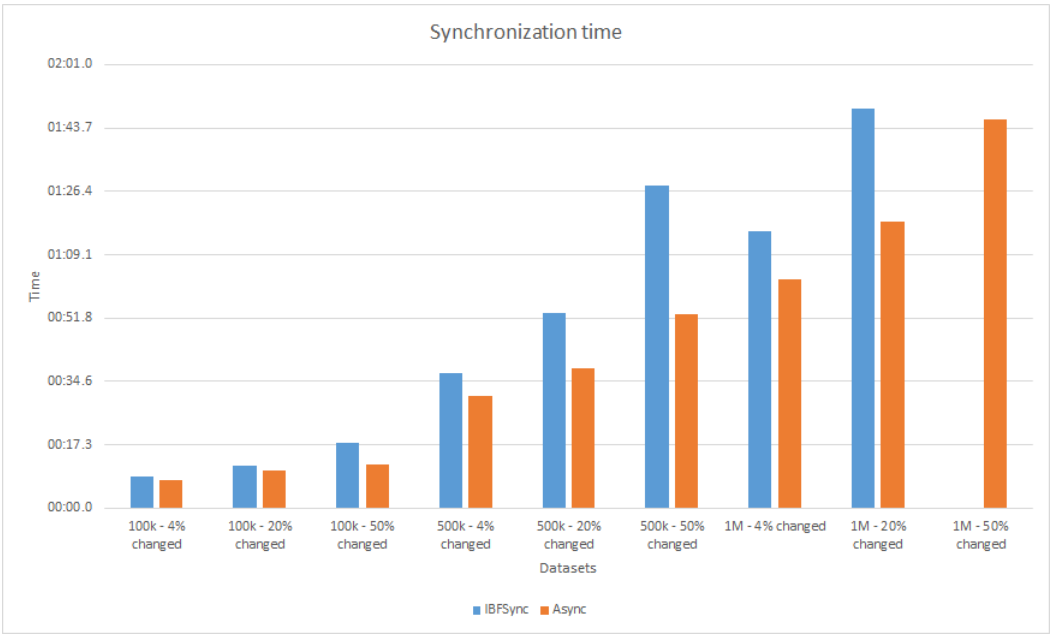

Figure 5.7: Synchronization Time

In order to see the difference between ASync and IBFSync, Figure 5.8 shows the total synchronization time of IBFSync relative to ASync. We can see that IBFSync is slower in all scenarios, ranging from around 10% to 65%.

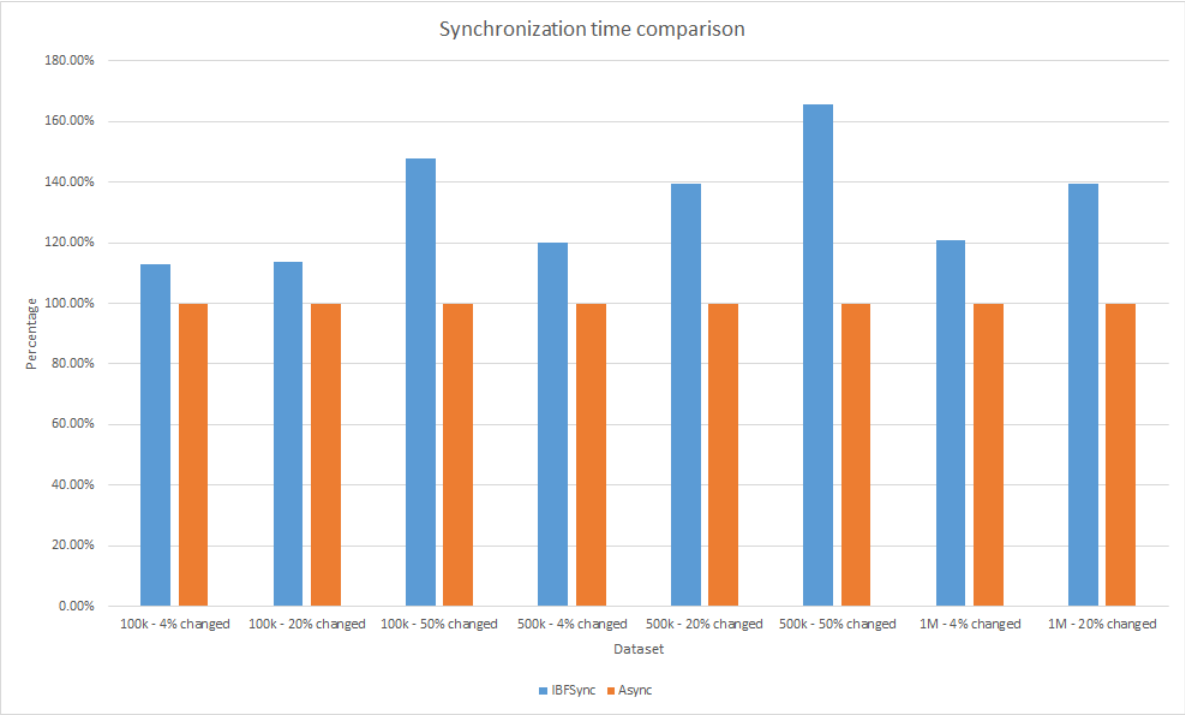

Figure 5.8: Comparison Between ASync and IBFSync, Synchronization Time

Synchronization Speed

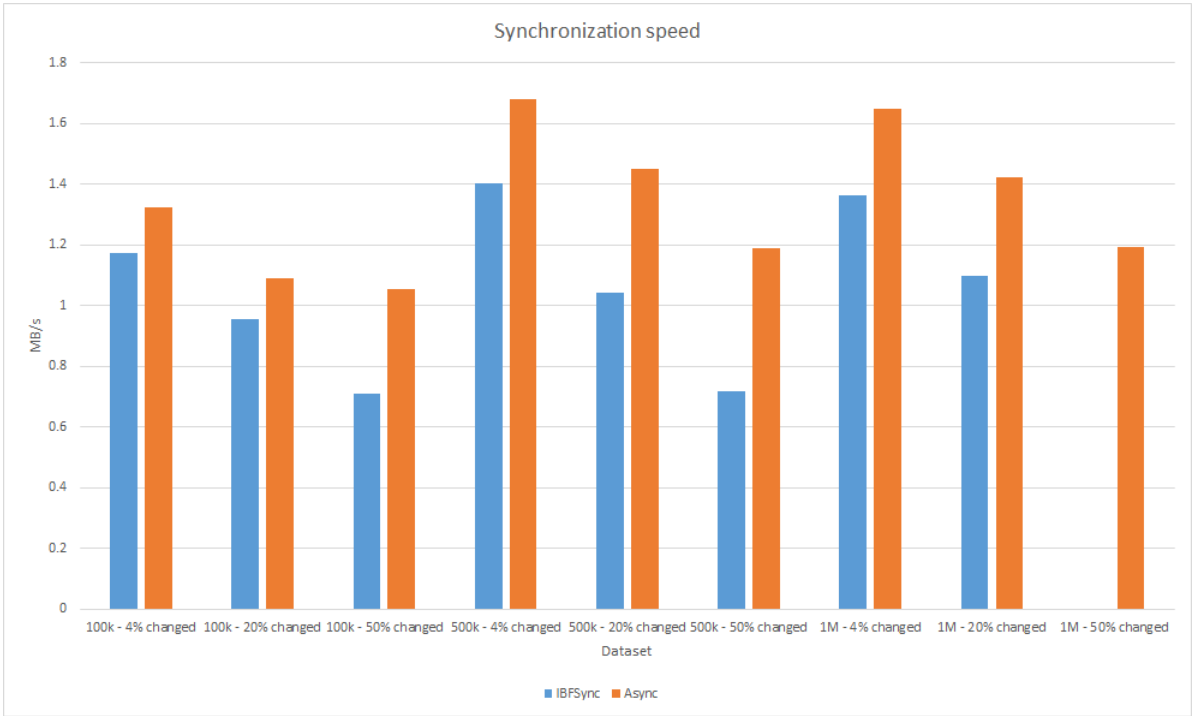

Figure 5.9: Synchronization Speed

In order to compare with the server-centric method, we measured both ASync and IBFSync algorithms in term of MB/s. The synchronization speed is calculated by dividing the size of the server database to the synchronization time. This metric let us directly compare ASync and IBFSync with the server-centric method, as the synchronization time of the server-centric method directly depends on the speed of the network. Figure 5.9 shows the synchronization speed of ASync and IBFSync.

We can see in this experiment result that ASync achieves synchronization speed ranging from around 1MB/s to 1.7MB/s in our testing environment. This means if the bandwidth of our network is more than 1.7MB/s, it is better to just use the server-centric method. Otherwise, it is better to employ our proposed algorithm, ASync. However, the synchronization speed of ASync would increase with better hardware (yielding better processing time) and different candidate databases (the smaller the changed percentage, the faster it would get).

In a network environment with fast, stable connection such as LAN (Local Wi-Fi network can have theoretical bandwidth of 75 MB/s in case of Wi-Fi n), ASync does not perform very well compared to the naive server-centric method. However, in many other network environments with slower speed such as peer-to-peer Bluetooth, NFC, or pay to use network like 3G, 4G, the strength of ASync on reducing communication cost can be hugely beneficial.

5.5 Summary

In this section, we evaluate our proposed algorithm, ASync, and see how it compares with existing algorithms in the same class. Three metrics for assessing synchronization algorithms are:

1. Communication cost
2. Processing time
3. Synchronization time

We designed the experiments and datasets to measure these metrics. We showed the experiment results of our proposed algorithm, ASync, as well as 2 other algorithms in the same class: IBFSync and server-centric approach.

The results show our proposed algorithm performing better in all 3 metrics compared to IBFSync, and achieve reasonable synchronization speed.

Chapter 6

Summary and Future Works

6.1 Summary

In this research, we set out to look into the problem of key-value database synchronization between mobile devices on peer-to-peer networks. We identified the challenges, analyzed the constraints and established the requirements of the problem. Taking these things into account, we proposed a new algorithm in order to achieve better performance characteristics in all three major metrics: communication cost, processing time and synchronization time compared to previous approaches. It does so while retaining the characteristics which make the algorithm suitable for a peer-to-peer network environment consisting of mostly mobile devices:

1. Limited number of communication rounds
2. Exact synchronization
3. Do not require any prior context data

The main ingredient that let our algorithm to achieve these goals is its 2-phase architecture:

1. The first phase doing approximate synchronization and calculating needed information for the second phase
2. The second phase doing exact synchronization

We did experiments regarding those three above mentioned metrics (communication cost, processing time and communication time) to validate our proposed algorithm. The result is that the proposed algorithm performs better than IBFSync, ranging from 10% to 65% in synchronization time, the most important metric. It also achieves the synchronization speed of 1.0MB/s to 1.7MB/s. Therefore, it can be applied to mobile applications operating on slow network, for example, Bluetooth or NFC connection.

In comparison to previous approaches, the proposed algorithm is novel in the following areas:

1. Application domain: while other works have been done on the set reconciliation problem, we applied this research on the specific problem of key-value storage system synchronization.
2. 2-phase architect with the first phase doing approximate synchronization and calculating needed information for the second phase, which does exact synchronization.

6.2 Contributions

This research has the following major contributions:

1. Study of existing approaches to key-value database synchronization, identify the characteristics needed for our specific problem of key-value database synchronization between mobile devices in a peer-to-peer network.

2. Designed and implemented a new algorithm for key-value database synchronization, which performs better than existing approaches in the same class.

3. Evaluated and measured the performance characteristics of the proposed algorithm, as well as compare it with existing algorithms.

6.3 Future Work

6.3.1 Other Components

Although the proposed algorithm is the core of a synchronization solution, it does not fulfill all the necessary tasks of a whole synchronization system. The following sections describe the missing components which could be implemented in the future to provide a complete synchronization solution for key-value databases.

Mechanisms for Discovering Peers

Our proposed solution does not include any mechanism for discovering peers. This sub-problem could be solved by numerous approaches. One simple method works as follows:

- Every device uses UDP to broadcast the synchronization service
- Every device also listen for the broadcast signal to identify devices supporting the synchronization service.

Other advanced techniques include exchanging peer list, caching peer list locally [24].

Conflict Resolution

Currently we leave the task of conflict resolution to the application layer. This design has the advantage of having an application-specific strategy for conflict resolution, however it also puts more work on the application developers. The situation could be improved upon by presenting a number of default conflict resolution strategies in the synchronization component, while still permitting application developers to use their own mechanism for conflict resolution.

6.3.2 Data Transfer Protocol

The current version of our solution uses FTP, however, using FTP requires the device to accept TCP connection. The FTP protocol is acceptable when every device is under the same LAN, however, over the Internet, we face the problem of NAT traversal.

For the future version, we could employ other protocols that work better over NAT to overcome this limit. This change in the data transferring protocol would enable the algorithm to work over the Internet, not just local LAN.

6.3.3 Algorithm

There are numerous areas in the proposed algorithm could be explored further for better performance. The following sections presents some potential areas for improvement.

Parallel Processing

The current implementation of our algorithm does each step sequentially. One possibility to speed up the algorithm is to do multiple steps in parallel, both locally in one device and across the client and the server.

One prime candidate for employing parallel processing across client/server is while the client uploads its Bloom Filter to the server, it can prepare for the next step by starting to calculate its identifiers right away. Currently, it only does so after applying the patch 1 returned by the server.

For local parallel processing, using multiple threads and lock-free data structures may provide some performance boost, especially considering the fact that nowadays even mobile processors have multiple cores.

Incremental Bloom Filter

Instead of creating Bloom Filters for each time we need to do synchronization, we keep a Bloom Filter as we modify the database. We can also employ other variations of Bloom Filter such as Incremental Bloom Filter [23], a better performance version of Bloom Filter using only two hash functions [25].

Better Estimator Based on Bloom Filter

Although the currently used method for estimating the symmetric difference using Bloom Filter is working, we expect to find even more accurate one.

Data Compression

Due to the nature of the IBF and Bloom Filter, they contain mostly integers as the data. Therefore, we can potentially save more network bandwidth by exploring data compression, especially using integer compression algorithms. However, the speed of the data compression algorithms must be fast enough to have a noticeable gain in our synchronization time.

A few notable fast integer compression algorithms includes Variable Byte Encoding [26] and Group Variable Integer [14].

References

[1] Bloom filters - the math. `http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html`. [Online; accessed July-2014].

[2] Hash collision probabilities. `http://preshing.com/20110504/hash-collision-probabilities/`. [Online; accessed July-2014].

[3] Murmurhash homepage. `https://code.google.com/p/smhasher/`. [Online; accessed July-2014].

[4] Wikipedia database dump. `http://dumps.wikimedia.org/enwiki/latest/`, January 2014.

[5] Sachin Agarwal, David Starobinski, and Ari Trachtenberg. On the scalability of data synchronization protocols for pdas and mobile devices. *Network, IEEE*, 16(4):22–28, 2002.

[6] Madhu Ahluwalia, Ruchika Gupta, Aryya Gangopadhyay, Yelena Yesha, and Michael McAllister. Target-based database synchronization. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1643–1647. ACM, 2010.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of The ACM*, 13:422–426, 1970.

[8] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[9] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Fast approximate reconciliation of set differences. In *BU Computer Science TR*, pages 2002–19, 2002.

[10] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[11] Mi-Young Choi, Eun-Ae Cho, Dae-Ha Park, Chang-Joo Moon, and Doo-Kwon Baik. A database synchronization algorithm for mobile devices. *Consumer Electronics, IEEE Transactions on*, 56(2):392–398, 2010.

[12] Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944 – 949, 2010.

[13] Couchbase. Why nosql. Technical report, Couchbase, 2013.

[14] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 1–1. ACM, 2009.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[16] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[17] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[18] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. What's the difference?: efficient set reconciliation without prior context. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 218–229. ACM, 2011.

[19] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[20] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[21] J Nathan Foster, Michael B Greenwald, Christian Kirkegaard, Benjamin C Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4):669–689, 2007.

[22] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[23] Fang Hao, Murali Kodialam, and TV Lakshman. Incremental bloom filters. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.

[24] Mandar Kelaskar, Vincent Matossian, Preeti Mehra, Dennis Paul, and Manish Parashar. A study of discovery mechanisms for peer-to-peer applications. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 444–444. IEEE, 2002.

[25] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[26] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[27] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49:2213–2218, 2003.

[28] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[29] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (INTERNET STANDARD), October 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151.

[30] Sebastian Schildt, Johannes Morgenroth, and Lars Wolf. Efficient false positive free set synchronization using an extended bloom filter approach. *Computer Communications*, 36(1011):1245 – 1254, 2013.

[31] Magnus Skjegstad and Torleiv Maseng. Low complexity set reconciliation using bloom filters. In *Proceedings of the 7th ACM ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*, pages 33–41. ACM, 2011.

[32] David Starobinski, Ari Trachtenberg, and Sachin Agarwal. Efficient pda synchronization. *Mobile Computing, IEEE Transactions on*, 2(1):40–51, 2003.

[33] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.

[34] Chen Tang, Anton Donner, Javier Mulero Chaves, and Muhammad Muhammad. Performance of database synchronization algorithms via satellite. In *Advanced satellite multimedia systems conference (asma) and the 11th signal processing for space communications workshop (spsc), 2010 5th*, pages 455–461. IEEE, 2010.

[35] Sasu Tarkoma, Jaakko Kangasharju, Tancred Lindholm, and Kimmo Raatikainen. Fuego: Experiences with mobile data communication and synchronization. In *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, pages 1–5. IEEE, 2006.

[36] Xiaomei Tian, Dafang Zhang, Kun Xie, Can Hu, Mengfan Wang, and Jinguo Deng. Exact set reconciliation based on bloom filters. 2011.

[37] Ari Trachtenberg, David Starobinski, and Sachin Agarwal. Fast pda synchronization using characteristic polynomial interpolation. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1510–1519. IEEE, 2002.

[38] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, The Australian National University, February 1999.

[39] Jun Yang. Smartphones in use surpass 1 billion, will double by 2015. `http://www.bloomberg.com/news/2012-10-17/smartphones-in-use-surpass-1-billion-will-double-by-2015.html`. [Online; accessed July-2014].