UiT THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Swiftmend

Data Synchronization in Open mHealth Applications with Restricted Connectivity

—

Christoffer H. Hansen

INF-3981 Master thesis in Computer Science, June 2018

“There’s nothing wrong with having a tree as a friend.”
–Bob Ross

Abstract

Open mHealth applications often include mobile devices and cloud services with replicated data between components. These replicas need periodical synchronization to remain consistent. However, there are no guarantee of connectivity to networks which do not bill users on the quantity of data usage. This might influence users to evade data synchronization. This thesis propose Swiftmend, a system with synchronization that minimize the quantity of I/O used on the network.

Swiftmend includes two reconciliation algorithms; Rejuvenation and Regrowth. The latter utilizes the efficiency of the Merkle tree data structure to reduce the I/O. Merkle trees can sum up the consistency of replicas into compact fingerprints. While the first reconciliation algorithm, Rejuvenation simply inspects the entire replica to identify consistency. Regrowth is shown to produce less quantity of I/O than Rejuvenation when synchronizing replicas. This is due to the compact fingerprints.

Acknowledgements

First and foremost I would like to thank my advisor Håvard D. Johansen, as well as secondary advisors Håvard Espeland, and Lars Brenna. Thank you for your valuable guidance, motivation, and imparting your deep knowledge of computer science.

Further I would like to thank a god in computer science, Dag Johansen, for his encouragement and introducing me to the Corpore Sano research group. Thanks to all researchers at Corpore Sano for providing an excellent working environment.

Addtionally, I would like to thank Pål Halvorsen and my former colleagues at ForzaSys for a great working experience during my internship. I am grateful for being involved in PMSys during the past years and for being able to develop on this system during my capstone and thesis.

Thanks to all my fellow students for all the joy[1], laughs, input and advice. All of you are wonderful and bright computer scientists. All your future colleagues are lucky to be working with you.

Your inclusion and positive environment during my stay at both working premises are deeply appreciated. This has motivated my daily attendance as I look forward to each day working alongside with you.

Thanks to my friends and family for all the support and giving me the opportunity to cultivate my likings for computers from a early age. Even though you threatened to block the Internet sometimes.

Special thanks to Marianne for encouraging and motivating me throughout the study. Thank you for all the laughs and making me food everyday. No one in my class thinks I can cook.

To all, your involvement is deeply appreciated. Again, thank you.

1. https://www.reddit.com/r/PicturesOfJonEating/

Contents

List of Figures

List of Tables

List of Definitions

List of Glossaries

API	application programming interface
CBA	Credential-Based Authentication
DSU	Data Server Unit
GDPR	General Data Protection Regulation
HTTP	Hypertext Transfer Protocol
IOT	Internet of Things
mHealth	Mobile Health
OMH	Open mHealth
RO	Resource Owner
RS	Resource Server
SPA	Single Page Application
SRL	Simula Research Laboratory
TSU	Team Server Unit
UiT	The Arctic University of Norway

1 Introduction

Architectures with mobile applications and cloud services often include replicated data between loosely coupled client and server entities. Such architectures includes clients having network connectivity restrictions in the form of being billed by the quantity of data used. Cellular networks in Norway and cloud services such as Amazon S3 both charge upon this. However, it is desirable to synchronize data even in such conditions, as replicas need to be regularly synchronized to preserve consistency. Data synchronization that generates high amounts of I/O on the network will drain such expenses. This factor might influence users to evade data synchronization. We have examined the systems [1, 2, 3, 4, 5], and conclude that these do not address quantity of I/O used on the network. Swiftmend is proposed to minimize the quantity of I/O used on the network and reassure users to perform data synchronization under restricted network conditions.

The prototype of Swiftmend is implemented for the athlete quantification system, PMSys [5]. Automated systems for collecting athlete performance statistics are becoming a necessary factor for sport clubs to remain competitive. These athlete quantification systems use phenotypic indicators to enable rapid convergence towards improvement of performance and injury treatments. The phenotypic indicators generated are owned by the athletes. Other actors, such as coaches or medical staff, are data interested parties that purposely analyze the indicators. The mobile devices show potential to improve disease prevention and management by extending health interventions in traditional care [6]. Open mHealth (OMH) is a standardization effort for data exchange. The OMH initiative proposes interfaces and a shared component architecture to increase interoperability in Mobile Health (mHealth) applications and services. A key component in the OMH architecture is the Data Server Unit (DSU). It is an open API specification for unified information sharing across data streams.

Athletes need a secure method to access their health data and delegate data to trusted principals [2]. PMSys [5] is a performance monitoring system for athletes, and enables controlled sharing of data through compartmentalized Linux containers to preserve the integrity of processing intent, administrative domains and roles. The system component referred to as the Team Server Unit (TSU), manage controlled sharing of user data using unforgeable tokens [7] and Credential-Based Authentication (CBA) [8]. PMSys was based on the Ohmage SDK [4] from Cornell Tech, and the Open mHealth specification [6]. The system provides monitoring of athlete's internal training load (rating of perceived exertion), wellness (physical and mental health) and injuries, by using smartphones. Data is collected with a subjective questionnaire submitted by the athletes in a mobile app. Coaches can visually inspect the data and trends through a trainer web portal.

The system consists of three main parts; a mobile application, a web-based trainer module, and several backend services. The components in PMSys has a three-tiered structure with storage as the third tier [9]. PMSys consists of loosely coupled Data Storage Units (DSUs) as one of the backend services. Data is transactionally replicated between the mobile application and the DSU, leaving full copies of the database in each component. The database replicas need periodic updates, as data change and leaves the states inconsistent. Hence, a consistency model is needed to guarantee consistency. The DSU therefore implements a weak consistency model, as writes can go to one of several replicas.

1.1 Data Consistency

Distributed database-management systems struggle with guaranteeing ACID properties in database transaction due to network unreliability [10]. Brewer's CAP theorem describes the impossibility of having consistency, availability and partitions simultaneously [11]. A system can guarantee two at most, and therefore requires a selection between the three guarantees. However, the guarantee of partition tolerance is mandatory in distributed system, and can never be sacrificed [12]. Because networks within distributed systems cannot guarantee no message drop, no crashes, networks are unreliable. The choice stands between a variation of consistency or availability with partition tolerance. This leads to the weak consistency model BASE [13], which forfeit consistency and isolation from the traditional ACID properties to increase the availability and performance.

Components in PMSys as the mobile application is affected by geographical separation. Perkins et al. [14] comments on the recent efforts that examine the viability of trading consistency for reduced latency in geo-distributed services. The limited and intermittent connectivity in mobile applications force a weaker consistency model, unlike the mentioned efforts trading consistency for latency. Due to intermittent connectivity, data should be available during absent network connectivity, commonly known as offline mode [15]. Both the web application and mobile application in PMSys contain persistent storage, and therefore complements disconnected operations. A weak consistency model, eventual consistency, enables such facilities opting for availability over consistency, and gives the user an always-on experience [16]. Weak consistency means not guaranteeing that any replica will always have the most recent updates. Updates are often communicated through propagation in a epidemic behavior with gossiping [17, 18]. The machinery serving the replica avoids blocking processes waiting on failed nodes or network to communicate updates of the replica. However, choosing availability over consistency implies client recovery of longer partitions as replicas diverge. Nodes are therefore required to exchange information with each other using data reconciliation to ensure state convergence. Anti-entropy protocols are often used to achieve convergence [19]. Anti-entropy is a type of gossiping, and part of an anti-entropy protocol is to compute the differences of datasets. A node use either merge or reconciliation mechanism to concur a new state by operating the consumed and currently possessed state. Merkle trees[1] is a frequently used data structure to implement anti-entropy protocols.

1.2 Problem Definition

This thesis propose a client-server synchronization architecture for distributed systems consisting of loosely coupled mobile devices and backend cloud services. Synchronization mechanisms that squander the network with high data usage might influence users to avoid synchronization, as cellular networks in Norway and cloud services bill users per quantity of data used.

Our thesis is:

> *Data synchronization in a mobile application can be efficiently supported by a Merkle tree data structure to reduce the I/O on networks*

1. also referred to as hash tree

paid per quantity of data used.

We deduce the following requirements that should be met in our proof-of-work prototype:

Requirement 1 synchronization cycles are required to perform periodically with ten minutes intervals to enable synchronization of athletes data between scheduled activities.

Requirement 2 data synchronization is required to reconcile replicas within a second to maintain transparency towards user experience.

Requirement 3 the system should completely support CRUD operations for data updates.

The proof-of-work prototype is implemented for PMSys. The system maintain multiple user replicas using Merkle trees. The synchronization mechanisms compute the difference on a dataset that is consisting of data objects with phenotypic data provided by the users.

1.3 Scope and Limitations

The thesis describes the design and implementation of data synchronization for PMSys this includes the system components PM Reporter and the TSU, excluding the web-portal for coaches.

The thesis builds on a solution presented in PMSys 3.0, which provides secure channels using unforgeable tokens [7] and CBA [8] for authentication and authorization to prevent data tampering. These aspects are therefore out of scope for this thesis.

While the thesis do investigate a reduction of I/O with experiments, we do not investigate memory or computational efficiency in regards to the synchronization mechanisms proposed.

1.4 Method

The ACM Task Force [20] describes the three major paradigms in the discipline of computing theory. The three paradigms are divided into:

Theory stem from mathematics, and involves the study of mathematical objects and their hypothesized relationships. The hypothesis are interpreted to determine proof, either proven or falsified.

Abstraction stem from the experimental scientific method, and involves investigation of phenomenons from a hypothesis. It includes constructing models for prediction and design experiments for analyzes.

Design stem from engineering, describing constructions of a system to solve a problem. This includes requirements, and specifications of the system. Design and implementation of the system. Lastly, testing of the system's behavior

This thesis use the design methodology to construct a system to enable data synchronization. This includes the design of constructing the system, implementation of the design and an evaluation of the system behavior.

1.5 Context

The Corpore Sano Centre[2] is a research group focusing on high-impact life science research and innovation intersected between the fields of computer science, sport science, and medicine. Specially, focused on novelty in the convergence space of mobility, social network, cloud computing, big medical data, and the Internet of Things (IOT). The conducted studies includes international collaboration with other academic and commercial partners.

These studies arise from the early work of mobile agents [21, 22] and network architectures [23, 24]. Mobile agents are used as a middleware architecture for distributed applications, which moves the computational environment of a mobile user. These experiences have inspired further work as the cloud database service, Jovaku, which demonstrates the viability of global caching by using existing DNS system [25].

Our markedly work in security and fault-tolerance has lead to overlay network protocols like Fireflies, which provide novel trade-off between Byzantine fault tolerance and scalability [26]. Codecaps propose cryptographically protected capabilities containing executable code that improves upon discretionary access control as they are often predefined, and capabilities are unable to be confined. Codecaps supports flexible discretionary access control in cloud-like computing infrastructures. [27]. This work is related to meta-code, which proposes a

2. http://www.corporesano.no/

mechanism that express and enforce security policies when having shared data [28].

With the sports analytics being a growing field of interest, our collaboration with Simula Research Laboratory (SRL) resulted in Bagadus, a real-time sports analysis system [29]. Bagadus has a integrated sensor systems and video processing enabling live monitoring of soccer matches. Muithu expands upon this and provide coaches annotation of live matches, and a social network for players and coaches to track training and nutrition [30].

1.6 Outline

Chapter 2 presents the architectures PMSys is based on, and the Merkle tree data structure used in Swiftmend.

Chapter 3 describes the general design and implementation of Swiftmend and introduce the two reconciliation algorithms; Rejuvenation and Regrowth.

Chapter 4 describes the client-side integration of Swiftmend in PMSys based on the design and implementation presented in Chapter 3.

Chapter 5 describes the server-side integration of Swiftmend in PMSys based on the design and implementation presented in Chapter 3.

Chapter 6 investigate the thesis statement and requirements from Section 1.2, by evaluating the two proposed reconciliation algorithms.

Chapter 7 concludes the thesis and outlines future work.

2 Background

This chapter describes standards and frameworks related to PMSys. We will describe the open architecture of Open mHealth, and the OAuth 2.0 framework that PMSys is based on. We evaluate the legacy PMSys synchronization protocol. Lastly, we introduce the Merkle tree data structure used in Swiftmend to improve the legacy synchronization protocol.

2.1 The Open mHealth Architecture

The mobile devices show potential to excel disease prevention and management by extending health interventions in traditional care [6]. OMH is a standardization effort for data exchange between *siloed* architectures. The motivation for the OMH initiative is the need for development and treatment of chronic diseases outside the traditional clinical settings, and to enable the patient to collect and share data constantly to obtain an agile conclusion to optimize the treatment of a patient.

The OMH initiative proposes interfaces and a shared component architecture to increase interoperability in mHealth applications and services. This counters incompatibility issues related to applications with distinct data format, management and analysis. This approach is referred to as a *siloed* or *stove-pipe*. *Siloed* architectures obstruct data-sharing with other applications, and is therefore inefficient of innovation and limits the potential of mHealth.

An architecture needs to support shared data standards to fully realize the mHealth potential. These architectures are referred to as an open architecture. The benefits of an architecture of this type is the well-defined API, which enables interconnection of systems. They are also called innovation infrastructure due to the interconnected vision. Though, they suffer limitations due to built-in security.

A key component in the OMH architecture is the DSU. It is an open API specification for unified information sharing across data streams. Directed to the architectures having *siloed* data stores, enabling them to share information.

The Open mHealth specification utilizes the OAuth 2.0 protocol to dictate sharing of resources. PMSys includes OAuth 2.0 terminologies and will therefore be described in the following section.

2.1.1 OAuth 2.0

The OAuth 2.0 authorization framework defines a protocol allowing third-party applications limited access to a Hypertext Transfer Protocol (HTTP) service. This is enabled by an user approval interaction between the user and HTTP service, or to obtain access in a third-party application on its own behalf [31].

OAuth addresses several limitations in the traditional client-server authentication model regarding third-party applications accessing restricted resources. This requires that the user shares its credentials with the third party, which cause issues.

OAuth introduces an authorization layer and separates the role of the client from the user. A client needs to request access to a restricted resource on behalf of the user. And this is issued with a different set of credentials than the user's credentials.

Rather than using an user's credentials, the client use an access token. This is a string with designating attributes associated with access, which are scope and lifetime. The scope attribute represents permissions, usually defined in an access control demeanor as read, write, and delete. A client can request different scopes from an authorization server. The scopes requires authorization from the user in the form of a consent, if the user approves, the access token will represent the authorized scopes. The client can then use the access token to access protected resources.

Roles

OAuth defines four roles, which is the terminology we will use with a few exceptions.

The Definition 1. OAuth Roles

Resource Owner: A entity capable of granting access to a protected resource. Usually an end-user, referred to as a person.

Resource Server: A server hosting protected resources, defined with an API for interaction. Able to accept and respond to endpoint requests by using an access token.

Client: An application requesting access to a protected resource on behalf of a Resource Owner (RO) with its authorization.

Authorization Server: A server authenticating and obtaining authorization from the RO and issues access tokens upon successful authentication.

Authorization Grant

An Authorization Grant is the credential representing the ROs authorization. The client uses this credential to obtain an access token. OAuth 2.0 defines four flows, also called grant types, to obtain an access token. Each flow is suited for cases depending on the client type.

The Definition 2. OAuth Grant Types

Authorization Code: Grant flow for web applications executing on a server- and mobile applications using the Proof Key for Code Exchange (PCKE).

Implicit: Grant flow for JavaScript applications, also called Single Page Application (SPA), executing in an user-agent (ROs browser).

Resource Owner Password Credentials: Grant flow for trusted applications only.

Client Credentials: Grant flow for machine-to-machine communication.

OMH define three fundamental components for data sharing. **(1)** Data stores needs the possibility to define the data they wish to share. This enables third-party clients to obtain an uniformed definition of data. **(2)** Third-party

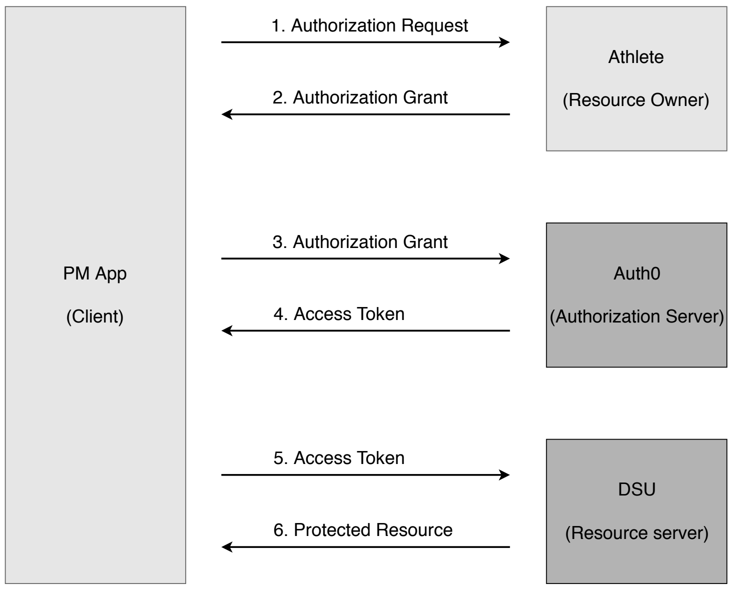


Figure 2.1: OAuth 2.0 architecture in PMSys

applications needs authenticated users to authorize access. **(3)** Servers requires a simple and well-defined API.

2.2 PMSys 3.0

PMSys is a performance monitoring system for athletes that is developed in collaboration with students and researchers at SRL, The Arctic University of Norway (UiT), and ForzaSys AS. The system provides monitoring of an athlete's internal training load (rating of perceived exertion), wellness (physical and mental health) and injuries, by using smartphones. Data is collected with a subjective questionnaire submitted by the athletes in a mobile app. The user interface and both surveys are illustrated in Figure 2.3, 2.4, 2.5, 2.6. Coaches can visually inspect the data and trends through a trainer web portal. Both mobile- and web applications shares the terminology *PM App*, while the mobile application only is referred to as *PM Reporter*. The system consists of three main parts illustrated in Figure 2.2: a mobile application,a web-based trainer module, and several backend services.

The initial version of PMSys was based on the Ohmage SDK from Cornell Tech and the Open mHealth specification. The system was later improved as part of several student projects, leading to the deployment of its version 1.0, which has been in production for four years. Version 2.0 made significant improvement to

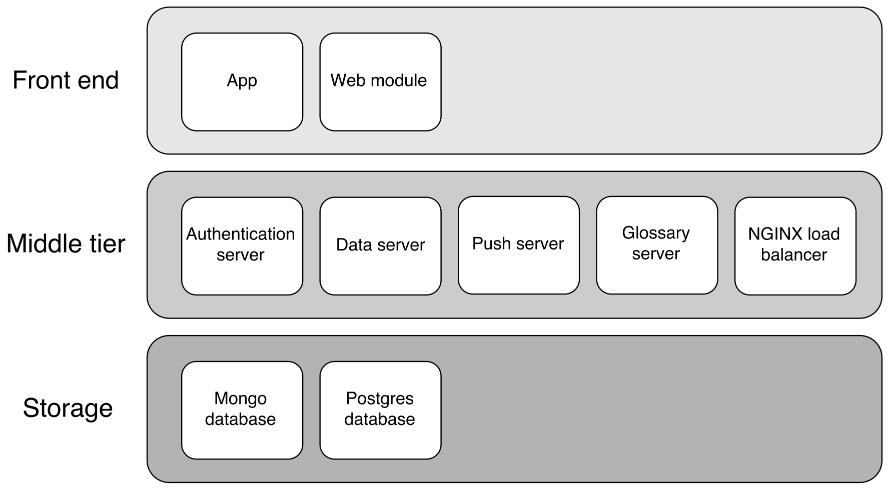
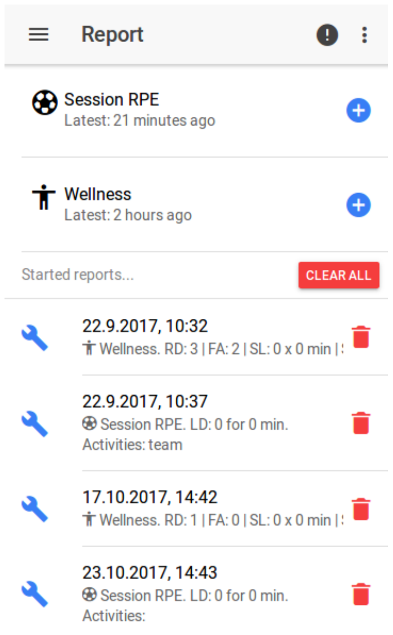


Figure 2.2: PMSys technology components.

the system by incorporating the updated Ionic[1] and Angular[2] frameworks. Also, version 2.0 was constructed with modernized security mechanisms (illustrated in Figure 2.1). Version 2.0 was never put in production, but development was moved towards version 3.0 for better General Data Protection Regulation (GDPR) compliance.

The Open mHealth specification used for PMSys 2.0 dictates use of the OAuth 2.0 protocol for authorizing requests addressed to the Resource Server (RS) or the DSU. Although this clearly defines several authentication flows, it does not say how credentials should be stored. PMSys 2.0 is opted to store credentials in a shared Postgres database on behalf of a RO. Each time the authentication server creates an access token it is stored in the Postgres database, before the token is sent to the requesting client after authentication has been completed. The client can access protected resources by attaching tokens in the HTTP header of subsequent requests to the DSU using the bearer scheme. When the data server receives the token, it queries the shared Postgres database to verify the token. However, due to recommendations in the upcoming GDPR from the European Commission, we want data to separate the shared database to prevent

1. https://ionicframework.com/
2. https://angular.io/



22.9.2017, 10:32
Wellness. RD: 3 | FA: 2 | SL: 0 x 0 min |

22.9.2017, 10:37
Session RPE. LD: 0 for 0 min.
Activities: team

17.10.2017, 14:42
Wellness. RD: 1 | FA: 0 | SL: 0 x 0 min |

23.10.2017, 14:43
Session RPE. LD: 0 for 0 min.
Activities:

Figure 2.3: PM Reporter: Main page

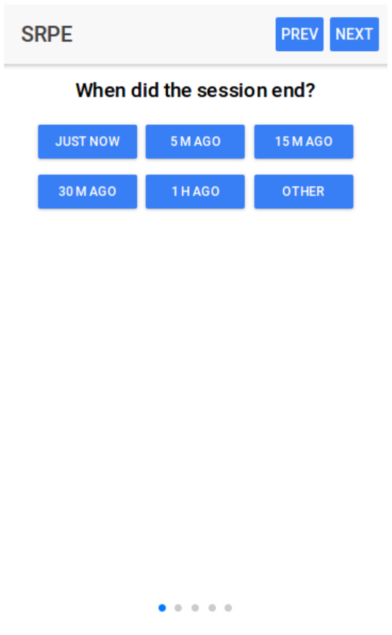


Figure 2.4: PM Reporter: SRPE survey

cross-contamination. Also, shared database is a bottle-neck, having slow query time and being a single point of failure, due to it being centralized.

Version 3.0 improved upon this using unforgeable tokens and CBA to enable decentralized authentication. Also, version 3.0 introduced the new system component, a Go-based TSU, to initially manage controlled sharing of user data using attenuated tokens. The TSU was originally built to communicate with the DSU. However, later development consolidated the two components by reimplementing the DSU into the TSU.

2.2.1 Legacy Synchronization Protocol

PMSys is built on the foundation that users own their data. Phenotypic indicators collected in the mobile application, PM Reporter, are stored locally on the device to substantiate user control. PMSys enables sharing of data through Linux containers in order to preserve the integrity of processing intent, administrative domains and roles. These containers run loosely coupled Data Storage Units (DSUs). The DSU contains replicated phenotypic data originated from PM Reporter.

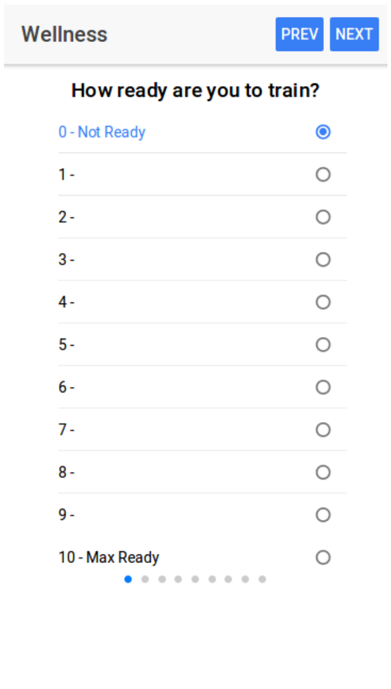


Figure 2.5: PM Reporter: Wellness survey

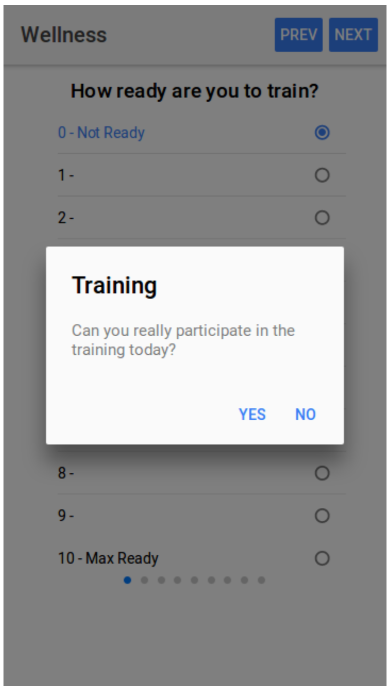

Wellness
PREV
NEXT
How ready are you to train?
0 - Not Ready
1 -
2 -
Training
Can you really participate in the training today?
YES
NO
8 -
9 -
10 - Max Ready

Figure 2.6: PM Reporter: Participation

Each report from an athlete generates a single datapoint, and is persisted in the local storage upon submission. The datapoint is pushed to an out queue with outgoing datapoints addressed to the TSU with access to the DSU before storing it locally. These datapoints are to be stored in the DSU. After a push on the out queue, PM Reporter executes a client and tries to transmit the submitted datapoint. A successful transmission removes the pertaining datapoint from the out queue, and persists the state of the out queue by flushing it to the disk.

Data stored in the DSU is shared to other users that are authorized by the data owner. This allows mobile- and web applications to perform data processing without the need of fetching data directly from each other. The data is then pulled and pushed from the DSU, being an available server application compared to the mobile- and web applications that might be offline. The dataset is therefore replicated and distributed, increasing the reliability and redundancy. However, distribution introduce issues regarding consistency.

Synchronization is required to preserve consistency between replicas, as the datasets can change. The previous synchronization mechanism uses a pull based method. PM Reporter pulls the entire dataset from the DSU and identify undiscovered datapoints present in the DSU to spawn them locally. PM Reporter processes each datapoint by attempting an insert to the local dataset, ignoring datapoints that are already present. There is an overhead related to pulling the entire dataset when datasets are consistent. All the data pulled are wastefully ignored as the data already exists, and results in unnecessary usage of bandwidth and computation. This is problematic to PM Reporter, as it is a mobile device with limited resources that should be used efficiently [32]. The DSU rely on pushed data from PM Reporter in order to preserve consistency between them. The queue handling is an important dependency for dataset correctness. The queue is vulnerable to corruption as it is persisted as a stack [33], and emits items over an unreliable network [10]. The synchronization mechanism lacks a forgiving repair that can detect unexpected inconsistencies. A synchronization mechanism providing idempotent difference of replicas will mend such occurrences.

The current PMSys architecture support two CRUD operations of replica updates: read and write. Athletes being the data owner qualify for all supported CRUD operations, while coaches are permitted read rights when authorized by data owner. Current architecture has no support of CRUD operations, update and delete. Datapoints are only pushed to the DSU after inserted into the out queue. Modifying a datapoint locally would never reach other applications or the DSU, as the modified datapoint would only exist in their respective applications. Because there are no mechanism that push the updated datapoint to the DSU. Other applications pulling the shared data from the DSU would rather receive the version of the datapoint in original state. A local deletion would also be ignored, and the next pull from the DSU would contain the deleted datapoint. The lack of complete CRUD support degrades users data control.

2.3 Merkle Tree

Merkle tree was originally proposed as an alternative signature scheme [34]. The Merkle Signature Scheme is an alternative to todays Digital Signature Scheme (DSA) and RSA signature [35]. Alternative digital signature schemes are motivated to counter the predicted insecurities in todays signature schemes. Digital Signature Scheme (DSA) and RSA signature rely on the difficulties of solving the discrete logarithm problem and the factorization problem. There are currently no existing algorithm that solves these problems efficiently. However, there are theoretically proven algorithms that solves these problems on a quantum computer. It is believed that these computers can be built in the future. Hence, an alternative digital signature scheme is needed.

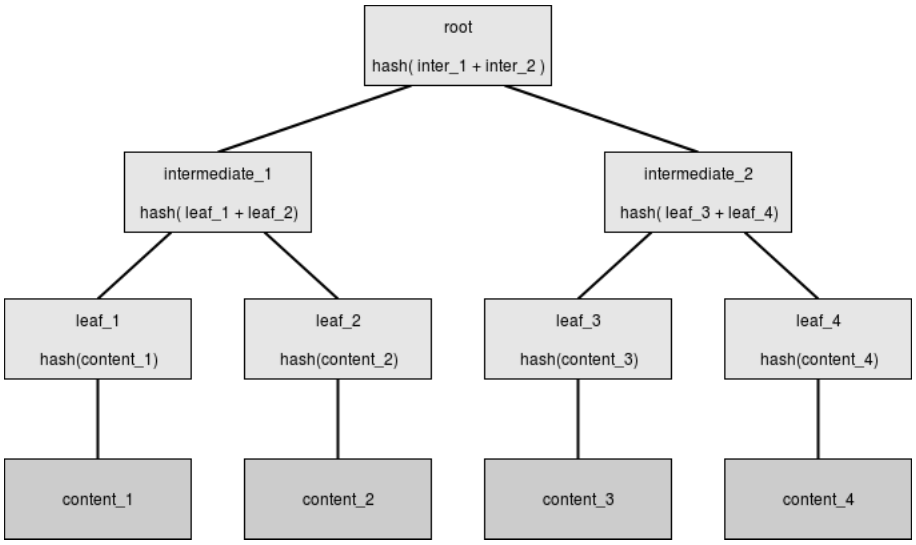


Figure 2.7: Hash tree

The Merkle tree is also known as a hash tree, and the data structure is represented as a binary tree [36]. The data structure is commonly used as either a signature scheme or anti-entropy protocol. The clear advantage of Merkle tree as an anti-entropy protocol is the efficiency [37]. The data structure has the ability to summarize a large data set into a compact fingerprint. Each tree node contain a checksum generated by a cryptographic hash function. The security is therefore dependent on the guarantees served by the cryptographic hash function used [35]. In the terms of usage in an anti-entropy protocol, it is problematic with collisions. The reconciliation process is dependent on the uniqueness in the hash values. Two different content hashed that results in the same checksum can potentially give invalid comparison between two unrelated leaf nodes. As the tree is traversed, a collision caused by duplicated hash that is nearest to the traversal path will always be detected first. The farthest checksum is therefore never discovered unless the traversal order is changed. The checksum serves an efficient purpose for inconsistency checking.

The hash tree is constructed with intermediate nodes containing the concatenated hash of its child nodes hash (shown in Figure 2.7). The leafs contain some content that is used as a hash key to create the leafs hash. The leaf hash is the origin to the intermediate node's hash, due to concatenation. As other hashes depend on the leaf hashes, a leaf insert in the tree requires a rebuild of the tree to recalculate affected branches.

3 Swiftmend: Design and Implementation

This chapter introduces Swiftmend an extension of PMSys 3.0, and propose two reconciliation algorithm for data synchronization; **(1)** Rejuvenation and **(2)** Regrowth. We describe the system components in Section 3.1, and explain the communication between components when performing data synchronization. Section 3.2 advocates the extension of the current data structure with versioning and delete certificates to enable complete CRUD support in Swiftmend. We further introduce the reconciliation algorithms; Rejuvenation in Section 3.3 and Regrowth in Section 3.5. Section 3.4 discusses alternative algorithms and advocates the use of Merkle trees in Regrowth.

3.1 System Components

Swiftmend has a client-server model, and consists of two system components: **(1)** PM Reporter and **(2)** the TSU. Figure 3.1 illustrates the components as two applications: PM Reporter being the mobile application, and the TSU being the server application. The data exchange between the components are formalized with REST API [38]. The server expose accessible endpoints enabling clients to fetch resources from the DSU. PM Reporter includes a REST client for inter-process communication with the server application. The system components

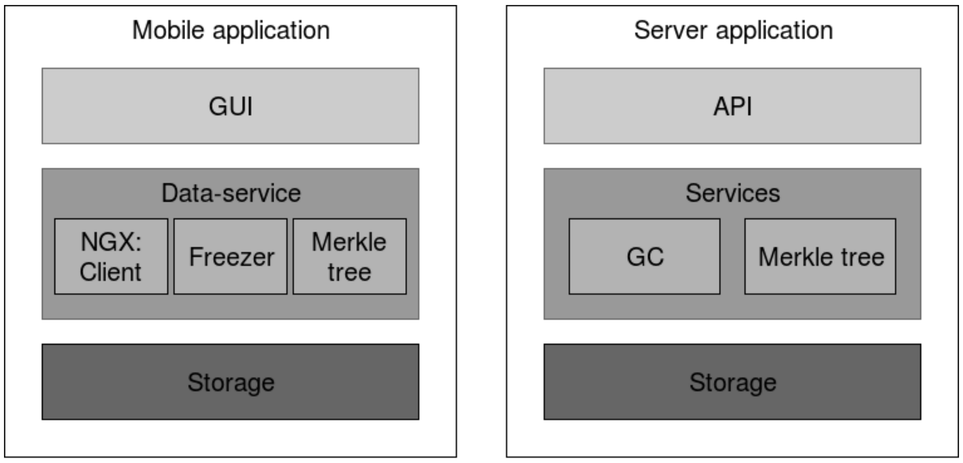


Figure 3.1: Swiftmend: System components

exchange messages on secure channels using unforgeable token of authority [7] implemented in PMSys 3.0. The API provides granular access control through scopes carrying permissions. However, this thesis do not include this previous implementation, as it is out of the scope.

PM Reporter, being a personal device used by the athlete, functions as a data manager and synchronization orchestrator. The control of data resides in the device due to the athlete being a data owner [39].

Swiftmend is built with a multi-master replication to enable data flows between PM Reporter and the TSU with a two-way synchronization mechanism. This approach requires orchestrating to maintain data flow and consistency, which resonates with the data manager/synchronization orchestrator role assigned to PM Reporter. Data-service is the service that functions as the active data manager and synchronization orchestrator. The service therefore maintains the consistency of replicas by invoking reconciliation on data change.

Database replication with the master/slave model suits the roles, appointing PM Reporter as master and the TSU as slave. A master/slave replication scheme requires a single point master federating data to slaves. However, the master is a single point of failure, and slaves is dependent on its uptime. This do not suit PMSys and interferes with the unreliable network bound to the mobile device being master. Inserting new datapoints would have to go through the master, and therefore cause problems when the master is unavailable.

The Rejuvenation reconciliation algorithm compute the difference on the entire dataset consisting of data objects with phenotypic data provided by the users. Regrowth instead use the Merkle tree data structure to detect inconsistencies.

Each athlete has an associated Merkle tree stored on each application, which possesses a replica of the athlete's dataset. In this thesis, it includes the mobile- and server application; PM Reporter and the TSU. Other users such as the athlete's coach, can posses an athlete's dataset upon consented sharing in the web application. However, this is out of the scope for this thesis, as Swiftmend only implements support for the algorithms in the mobile- and server application, leaving out the web application. However, in the case of sharing data with a coach, Swiftmend intends on sharing trees from the DSU as data is also shared from this server application to enable the web application to construct the same trees as the other applications. The trees functions as a reference point to verify tree integrity between PM Reporter and the DSU. Taking the data usage on the network into account, clients do not download or transmit entire trees from each other to avoid network packages of growing size. The clients issue tree branches in order to bundle compact network packages of small size. Since the trees are stored in each client and server means that they have to build the trees individually. Tree maintenance is issued from the clients, contrary to the server an therefore contributes to a stateless server preserving horizontal scaling.

A fundamental problem related to Merkle tree is the construction time addressed in The Merkle tree traversal problem [35]. The cryptographic function used in the tree implies that it is infeasible to invert for malicious users, while being cheap enough to compute for legitimate users. Despite being cheap and providing secure properties as collision resistance, it still issues expensive computation. Athletes in PMSys continually generate datapoints for quantifiaction that ultimately leads to a large tree. Generating bigger trees increase the expense and is therefore impractical to compose regularly. As mentioned above, the applications in PMSys are three-tiered, which includes a storage layer. Tree data structures are persisted in the storage layer separately in each system component. Storing trees avoid time spent generating hash values when there are no updates in the tree that require a rebuild. The solution counters the computational expenses, but still face capacity issues when data grows.

A study that examines the reliability of fifteen popular mobile applications and synchronization services has been conducted [40]. They systematically introduced failures like network disruption, local app crash and device power loss. The study encountered data loss, corruption and inconsistent behavior and generally poor data management. The network disruption test shows loss of data when synchronization fails and is not immediately handled after reconnection. The crash test shows corruptions and inconsistencies for application with objects, while table-only recovers correctly. Swiftmend partially complies with this as PM Reporter has a relational database SQLite as storage medium and the DSU store documents in mongoDB collections. The collections are

	PMSys 3.0	Swiftmend
C	✓	✓
R	✓	✓
U	×	✓
D	×	✓

Table 3.1: CRUD support in PMSys 3.0 and Swiftmend architecture

analogous to a relational database table.[1] However, to which degree this holds is uncertain.

3.2 Data Structure

Inconsistency checks of datasets identifies missing datapoints in either dataset to consolidate their sets and maintaining the most recent version of datapoints. To enable such features requires escalating the CRUD support of PMSys (illustrated in Table 3.1). Swiftmend proposes an extended data structure that enable this.

Additional information describing versioning and delete orders is needed to implement reconciliation between distributed replicas. This information is placed in an extensional set to the OMH standard, called additional properties. The OMH header is modified to contain this set as replica updates will propagate this information to other components. The complete data structure with the extensional set additional properties is illustrated in Listing 3.1.

3.2.1 Versioning

Created data objects initially starts with a sequence identifier stating the version of the object. The sequence identifier is a logical clock [41] that is used to manage versions by capturing causality between object versions. The system avoids virtual synchrony of logical clock updates since modification of data objects are restricted to the data owner. Updates, which include CRUD operations update and delete, are only propagated from the mobile device, leaving the device's own logical clock as the singingly dependent clock. The initial value is 0 and implies no modification.

1. https://docs.mongodb.com/manual/core/databases-and-collections/

3.2.2 Delete Certificate

Revocation is a vital user functionality to empower user control [39]. The data object take in a delete certificate upon deletion [42]. The clause is appended to the additional properties in the header set. The initial value is 0 and implies no deletion, while 1 indicates a delete order.

Both PM Reporter and the TSU have an implemented garbage collector described in Chapter 4 and 5. The garbage collector services manage delete orders from the data owner to enable data deletion in the system.

Listing 3.1: JSON-object

```
{
  "header":{
    "id":  "985ec37c-573f-ad88-be4f-ba69ec51a2f4",
    "creation_date_time":  "2018-03-28T12:30:55.524",
    "schema_id":  {...},
    "acquisition_provenance":  {...},
    "user_id":  "test_id",
    "additional_properties": {
      "sequence_id":  5
      "delete":  0
    }
  },
  "body"  {...}
}
```

3.3 Rejuvenation: Simple Reconciliation

The simple reconciliation is based on the PMSys legacy synchronization protocol described in Subsection 2.2.1. The approaches similarly pull entire database copies upon synchronization. The process of reconciliation differs in the two approaches after obtaining the dataset. Rejuvenation aims at executing an idempotent difference of the dataset that compares all pulled database copies against each other, including the local database copy. Rejuvenation uses the idempotent difference to detect unexpected inconsistencies and repairs both datasets. The legacy version includes no idempotent mechanism and is reliant on the out queue correctness.

The reconciliation process starts by pulling the entire database copy from the DSU into memory of PM Reporter. The datasets are represented with a key-value map, where the key is assigned with the header identifier and value containing the data object. The sets are sorted on size prior to reconciliation in order to preserve array boundaries when repairing. Missing datapoints are discovered by comparing the second dataset with the first. Each key in the second dataset is checked for existence in the first set. A miss in the first set causes an insertion of the missing datapoint from the second. When insertions are directed towards the remotely pulled set, the client will invoke a POST request to the DSU that contains the freshly inserted datapoint. Upon hits, the datapoints from both sets are compared on the sequence identifier to agree on the most recent datapoint version. Likewise with insertions, when the remote set is found holding the oldest datapoint, the client invokes a PUT request to update the current datapoint. The reconciliation process is iterative and continues repeating the same process until all datasets are compared bidirectionally.

Rejuvenation improves upon the legacy synchronization protocol by having an idempotent difference. However, pulling the entire database results in large network packages as the dataset grows. Secondly, inconsistency checks are computational costly as the entirety of all datasets require inspection. Alternative reconciliation algorithms are therefore examined to improve on these limitations by reducing the I/O.

3.4 Alternative Reconciliation Algorithms

A study evaluates four reconciliation algorithms concentrating on accuracy and bandwidth [37]. It is assumed that available network bandwidth is the bottleneck in distributed replica repair. Hence, finding differences in replicas should be minimized at low transfer costs. The experiments has two scenarios inspecting the reconciliation cost of either a failure item removal (failure type regen) or outdated items (failure type update). The worst performance was encountered in the naïve approach that sends a list of all dataset keys and their version to other nodes. The naïve approach has similar features to Rejuvenation, as both issue all dataset keys and versions. However, Rejuvenation also includes the value of the map, containing the datapoint that naïve avoids by only sending the key. Adapting this feature of naïve into Rejuvenation by only sending the header with the sequence identifier would reduce the amount of data sent.

The alternative algorithms described in the paper achieves significantly lower reconciliation costs. For updates, both SHash and Bloomfilter [43] are constant to data load, being an advantage on high load. Merkle tree has an unbeatable low reconciliation cost on small data load. For regen, Bloomfilter is the most

	Trivial	SHash	Bloom	Merkle
Efficiency/load	Intermediate	Intermediate	Intermediate	High
Avg. recon. cost	Worst	Mediocre	Mediocre	Best
Variance	Consistent	Consistent	Variable	Variable

Table 3.2: Reconciliation algorithms

efficient at approximately intermediate load. However, it experiences a low hit on missing items by only finding half of them. The Merkle tree shows similar trend to the update scenario performing well on small data load. The findings related to Merkle tree correlates with the fundamental tradeoff in the data structure. The developer has to balance between tree size (branching factor and tree depth) and accuracy of obsolete datapoints detection. A datapoint/leaf ratio of 1 is most optimal, as leafs would contain a single datapoint, and its hash value giving an immediate hit in the leaf avoiding search in the leaf. This ratio is impractical to obtain when having a large dataset, as it implies a large tree. At last, the algorithm complexity related to the reconciliation cost is observed as $\mathcal{O}(n \cdot \log n)$ for Bloomfilter, SHash and Merkle tree assuming a balanced tree $\mathcal{O}(\frac{n}{b} \cdot \log_v(\frac{n}{b}))$. Their optimized Merkle tree reconciliation cost decrease sub-linearly at higher loads, as smaller trees are less efficient than larger trees due to interval splits.

Additionally, Scuttlebutt Reconciliation is an efficient mechanism to handle high update loads on limited network bandwidth and CPU cycles [44]. These characteristics suit Swiftmend. Scuttlebutt Reconciliation limits the transmitted data of an anti-entropy gossip by the requirement that the data requires a higher version number than any used before. Unfortunately, Scuttlebutt Reconciliation is not evaluated in the research conducted by Kruber et al [37]. The efficiency of the tree data structure served by the compact fingerprint and the algorithm complexity served by trees show satisfying results in the evaluation of Kruber et al. and is therefore preferred over Scuttlebutt.

3.5 Regrowth: Merkle Tree Reconciliation

Regrowth and the Merkle tree implementation proposed by Kruber et al [37] are both using the Merkle tree data structure. However, Regrowth differs by not having Nye's trie [45] in the leaf nodes, and Regrowth issues each branch singularly in a network request, while Kruber et al. issues an entire level into a request.

3.5.1 Leaf Data Structure

Regrowth is implemented with a datapoint/leaf ratio of 1 which is the most optimal [37]. However, the ratio is described as being impractical as the tree size grows. Achieving the most optimal efficiency and accuracy is highly dependent on the trade-off between tree size and false positives in objects needing repair. Neglecting techniques to balance the trade-off results in eventual high communication costs, which is undesired in Swiftmend. Storing a key-range of multiple key-value objects instead of a single object would reduce the tree size. However, the large amount of key-value objects in the key-range needs to be exchanged upon repair. Having a large tree results in exchanging a large amount of metadata to identify inconsistent objects that needs repair.

The data object in the leaf is used as the hash message to generate the leaf node hash. The data object in the leaf is a compact JSON-object representing a subset of an athlete's datapoint. The JSON-object is compact and can be mapped to the athlete's datapoint by using the header identifier as reference. It is possible to cache the entire athlete datapoint in the leaf. However, large objects require more hash computation, while compact objects minimize the expense and size of the leaf bucket. Compact objects take up extra space when persisted on disk, but avoids time spent extracting necessary information from the athlete object. It is expected that the trade-off between hash computation and storage is not worthwhile as the dataset grows. The need for techniques to reduce the tree size will emerge as the dataset grows.

Listing 3.2: Leaf JSON-object

```
{
  "header":{
    "id":  "985ec37c-573f-ad88-be4f-ba69ec51a2f4",
    "additional_properties": {
      "sequence_id":  5
    }
  }
}
```

The leaf JSON-object is illustrated in Listing 3.2, contains a static datapoint identifier extracted from the datapoint header for reference. The sequence identifier within additional properties is the defining attribute that transitions the hash value to indicate inconsistency.

3.5.2 Tree Construction

PM Reporter and the DSU tree are built independently with persisted datapoints as leaf content. The trees shares an invariant regarding the leaf content order. Creating a tree on either client or server requires a fixed sorting approach of the dataset. A unsorted dataset can create inconsistent hashes for trees with identical data. The concatenation of leaf node hashes is sensitive to the sequence of string parameters. Distinct sequences equals dissimilar outcomes. E.g, a concatenated with b, or b concatenated with a.

Since Merkle trees are constructed with tree nodes in the power of two [35], it is required to handle cases of odd tree nodes. Such scenarios with three data objects results in three leaf nodes, and thus identified as an odd number of leaf nodes. Likewise for interior nodes, with the scenario of having six leaf nodes. This will result in three interior nodes being the parent to the leaf nodes and thus is required to increase its extent.

One technique to increase the extent is duplicating the last tree node in a level when having an odd number of nodes. A problematic behavior related to this technique is identifying missing datapoints present in the remote tree. Regrowth verify tree node existence by checking the local tree nodes with the remote tree nodes. In the scenario of remote tree having a new datapoint present at the leaf node slot where the local tree has a duplicated leaf. The tree traversal is preordered and traverse from left to right. The local duplicated tree node being the right node is checked for existence in the remote tree and will return a hit as the left tree node is checked first. The traversal will not further pursue the right tree node. This leaves the new datapoint undiscovered, and the local and remote tree continue being inconsistent unable to identify the tree node needing repair. This case is also true for duplicated interior nodes that can result in an entire branch of new datapoints being undiscovered. A solution to this problem is comparing tree nodes with an attached index number enabling the tree traversal to not confuse the left node with a right positioned duplicate. Another solution is cross-checking the trees mirroring the local to remote check with a remote to local check creating a bidirectional verification. This solution would require the mobile application and server application to act as client and server to exchange information both ways with a defined message structure. This would require the server to handle states and breaks the statelessness of the server and would eventually implicate horizontal scaling with more users. Creating a new server application to handle such states and Merkle trees would offload the DSU and resolve the conflict preserving the statelessness in the DSU.

Rather than implementing such solutions, Regrowth use null pointers to extend odd levels. Using null pointers require careful implementation to manage the null pointers in a correct manner without causing fatal errors. The additional deliberations introduced are considered worthwhile, as the null pointers consume less space compared to duplicated nodes and avoids false positives experienced in the duplication technique.

3.5.3 Tree Verification

The tree is preorder traversed, recursively, in PM Reporter and the DSU. The traversal algorithm is illustrated in Listing 3.3. The initial hash verification of a reconciliation phase targets the tree root. The algorithm continues from the root and traverse down the left and right branches recursively in that sequence.

PM Reporter sends the root hash to the DSU for comparison. The root level is the only level that is executed sequentially, as the other levels check left and right tree node asynchronous. In the case of the tree node being an interior, two recursive call is executed concurrently with the left and right tree node as argument. A tree node hit in the remote tree verifies consistency of all underlying tree nodes, if there are any, in the local tree. PM Reporter registers hits when receiving the standard response for successful HTTP request 200 OK, from the DSU. Oppositely, hash value miss results with status code, 404 Not Found. A hit results in no further investigation for the branch. However, a recorded miss will traverse the branch further until the leafs are reached.

Listing 3.3: Traversal algorithm

```
node = root

traverse(node)
  if hit:
    break
  if miss:
    if node == leaf:
      break
    traverse(node.left)
    traverse(node.right)
```

The reconciliation repairs two types of inconsistencies: missing datapoints and outdates datapoints. The reconciliation process identify the two types in a singular phase. However, the repairs are split into two sections and illustrations describing each repair identification and handling separately. Figure 3.2 presents the process of finding missing datapoints, and Figure 3.3 illustrates outdated datapoints.

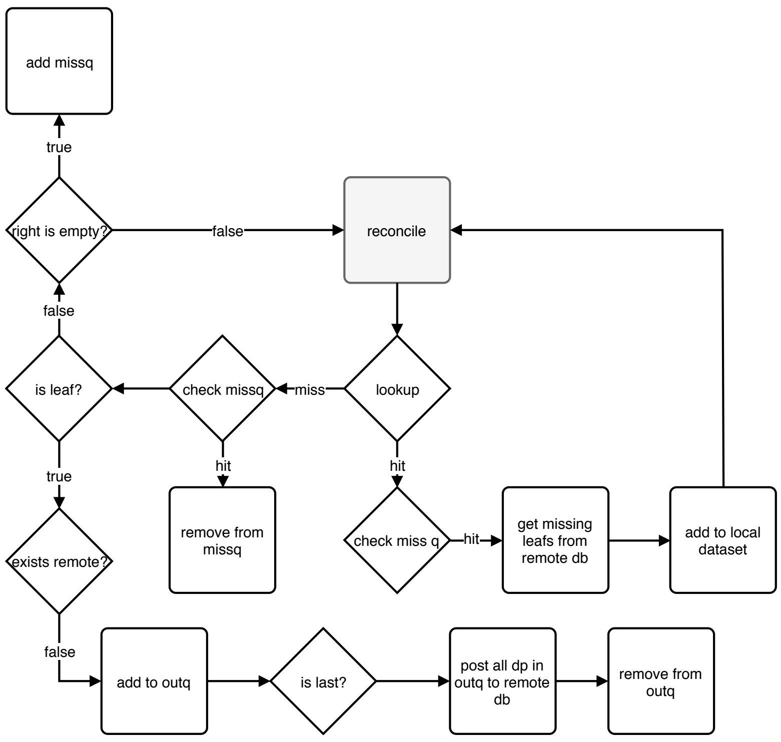


Figure 3.2: Repair missing datapoints

Repair: Missing

The reconciliation process use the two sets MISSQ and OUTQ for identifying missing tree nodes. A missing tree node can eventually entail missing leaf nodes, thus imply missing datapoints. A missing node is either filtered to the MISSQ or OUTQ depending on being identified as absent in the local or remote dataset. The collections differ in content: MISSQ contains node hashes and OUTQ contains datapoints.

OUTQ represents the missing datapoints identified as not present in the DSU. The reconciliation process identify these when the leaf is reached. Leaf nodes contain data objects representing datapoints through an identifier reference. The datapoint is fetched from memory using the identifier, and then used in a GET request addressed the DSU. A status code of 404 with message Not Found indicates absent and inserts the missing datapoint to the OUTQ. The process continue inserting missing datapoints until the last leaf is checked. All collected datapoints are sent with a POST to the DSU, and removes successfully delivered datapoints from the OUTQ set.

MISSQ represents missing datapoints not present in PM Reporter. Identification

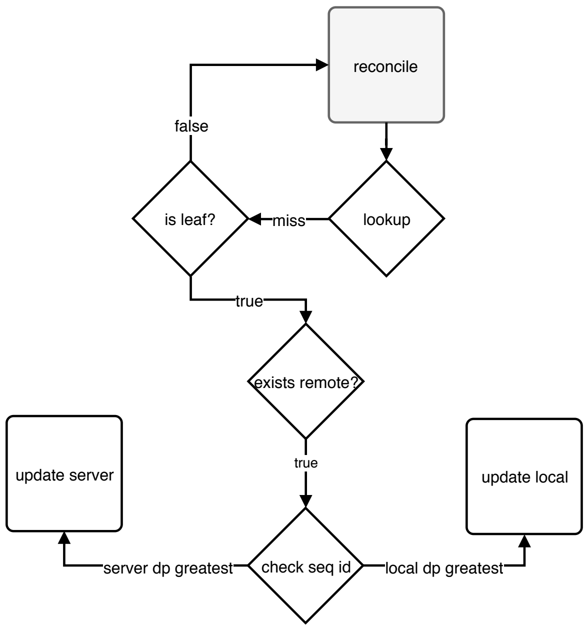


Figure 3.3: Repair outdated datapoints

of these datapoints are obtained by validating emptiness of a local null pointer node. When interior nodes are found inconsistent between trees, it indicates that left and/or right subnodes will predictably lead towards more inconsistencies in deeper branches. Missing datapoints are identified by utilizing this property. The reconciliation process peeks on the next scheduled subnodes and identifies if the right subnode is empty, and thus is a null pointer. Nodes being null pointers are not further pursued, as they contain nothing. However, left subnode will be scheduled for consistency check, and is therefore inserted with the hash value to the MISSQ. A miss in the left subnode upon the next cycle indicates inconsistencies further down the left branch. The left subnode hash value is therefore removed from the MISSQ as further investigation is needed. However, when a left subnode is defined consistent after a check, it indicates that the unchecked local null pointer contains undiscovered branches in the remote tree. PM Reporter will fetch the most recent local datapoint and requests all later datapoints from the DSU.

Repair: Outdated

The reconciliation process eventually reaches the leaf nodes. These nodes contains data objects referring to phenotypic datapoints. The local and remote datapoints are compared with the sequence identifier present in each datapoint. There are two scenarios dependent on the comparison, either the local dataset contains the most recent datapoint or the remote storage. A lower sequence identifier found in the local dataset compared to a greater identifier in the remote dataset means the local datapoint has to request the most recent version of the datapoint from the remote dataset and replace it with the outdated version. Contrary, PM Reporter has to update the remote datapoint with the fresh datapoint present in the local dataset.

4 Client-side Integration

This chapter describes the client-side integration in PM Reporter to support Swiftmend. PM Reporter is a hybrid application written in TypeScript using Ionic and Angular framework. PM Reporter consists of several services and libraries. We will explain the modified PM Reporter services, libraries and their implemented designs for supporting Swiftmend. Section 4.1 describes the service responsible for data manging, including synchronization orchestrating and data persistence. Lastly, Section 4.2 explains MerkleTS a TypeScript library providing Merkle tree data structure used by the data-service to support Merkle tree.

4.1 Data-service

Angular components managing presentation of data are dissociated with services managing, fetching, or saving data. Services are included as providers[1] in the application's dependency injection system and creates a shared instance injected it into all requesting components. The data-service is implemented as a provider and is responsible for persisting local state, orchestrating synchronization of client and server state and provide data for components.

JavaScript features immutability in primitive types. Angular use immutable ob-

1. https://angular.io/guide/providers

jects in their application state to avoid performance overhead used on tracking changes in mutable objects. Pure usage of immutable objects are inefficient when changing a single property of a data object, as a single change would require an entire application state update. To benefit from both immutability and mutability, components has local state with restricted mutability. Updating the state is restricted to the cases of input change or event emits. Components has lifecycle hooks[2] that notify changes upon subscription if needed. Having local state avoids the need of incorporating the desired mutability into application state that breaks component's encapsulation.

4.1.1 Storage

The data-service persists four types of data; the Merkle tree data structure, athletes phenotypic data, synchronization state, and time capsuled objects to be deleted. The persistence of the Merkle tree data structure, synchronization state and time capsuled objects are elaborated on in the following sections (Section 4.2, Subsection 4.1.3, and Subsection 4.1.4). The data-service is responsible for loading the persisted data from storage upon service instantiation and persist changes in each data category upon change.

Data-service uses the Storage module[3] to store key/value pairs and JSON objects. It has the flexibility of utilizing a variety of storage engines. An established prioritization picks the best suited storage engine available depending on the platform. The module favors SQLite for native applications due to its wide use, extensive testing and stability. Less favored databases are localstorage and IndexedDB. These encounter data expunge from the OS in low disk-space situations.

4.1.2 Resource (REST) Client

Data-service use the ngx-resource[4] library as a rest client interface. The library support customizable resource CRUD for accessing desired API. The resource is accessible for the data-service through the application's dependency injection system. The resource provides resource actions that the data-service uses to communicate with the server API presented in the DSU.

The resource actions are executed with asynchronous callbacks. The asynchronous operation's eventual completion or failure is represented in a Promise

2. https://angular.io/guide/lifecycle-hooks
3. https://ionicframework.com/docs/storage/
4. https://github.com/troyanskiy/ngx-resource-core

object,[5] handling a single event. The Promise object is wrapped as an Observable[6] using the fromPromise method exposed in the RxJS Observable object. Observable is preferred since it provides extended features of a Promise. An Observable allows multiple events and represents a push based collection, supporting an array of asynchronous events. An observer subscribes to the Observable and operates upon the emission and notification from the Observable. The event handling pattern provided by the Observable relates to the model of asynchronous programming and design, reactor pattern [46]. The Observable handles the concurrent input by demultiplexing the requests and then dispatch them synchronously to appropriate request handlers. The error or complete notification is captured by the Subscribe operator through channels.

4.1.3 Synchronization Orchestrator

The mobile application manage the reconciliation process as an orchestrator and therefore persists data modification state to avoid unnecessary reconciliation and tree constructions. The state is initially set to unchanged. Any data operation transitions the state to changed and immediately writes the state to disk for persistence. Figure 4.1 shows that the orchestrator starts reading the data modification state to figure out if the dataset has changed from previous reconciliation phase. In the case of data change, the tree is rebuilt with current datapoints fetched from local storage. After successfully constructing the local tree, the change state is set back to unchanged and persisted to disk. The phase is invoked periodically in the mobile device rather than continuously, in order to conserve battery drainage caused by computational activity [32].

Ionic Native provides a TypeScript wrapper for cordova plugins allowing usage of native device capabilities. The data-service uses two plugins: Battery Status[7] and Network[8] to restrict reconciliation processes to convenient moments. This resonates with the proposal of appropriate techniques to reduce power consumption [47]. Additionally, a research question was raised concerning the risk to program and/or data integrity during loss of battery power in the events of a transaction or system update [48]. The limited amount of energy might be lost rapidly during halted transactions or processes. The data-service watch both network and battery parameters for repercussion. The data-service contain two states, they indicate low battery status, and network connectivity. The events of both states are captured in observable subscriptions. The observables push the state upon the event, enabling the state to transition. The synchronization

5. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
6. http://reactivex.io/documentation/observable.html
7. https://ionicframework.com/docs/native/battery-status/
8. https://ionicframework.com/docs/native/network/

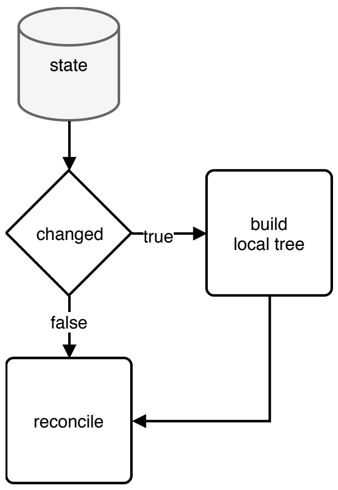


Figure 4.1: Synchronization Orchestrator: local tree construction before consistency check

process is always initiated when the event of network connection is triggered. Additionally, periodic intervals invoke the synchronization process with a ten minutes interval as required in Section 1.2. The intervals are configurable to user behavior and activity. Data objects are all products of submitted reports that describe a training session or morning routine. The chores are part of a schedule, and is therefore a predictable user behavior depending on the chore follow-through. The process will immediately cancel if the conditions of low battery or network disconnection are present. The process comply to a time span in order to avoid frequent execution. The difference between last synchronization and current time indicate that the device has exceeded or remained within the time span. Last synchronization time stamp is persisted in local storage after each synchronization cycle. Surpassing the time span implies the need of resynchronization to maintain consistency.

Inappropriate management of reconciliation from the orchestrator will result in fatal server-side errors. Invoking tree rebuilding during reconciliation on behalf of a user while simultaneously request the server to load the tree from disk results in problematic behavior. The error was experienced in the reconciliation phase when the traversal arrives in the inconsistent leaf nodes issued for repair. Repairing any leaf node originally resulted in frequent requests for tree rebuild. Other tree nodes still searching for inconsistencies would then issue an interfering tree load. The synchronization method barrier [49] solves this by introducing a wait group of processes that stop further proceedings until all processes has reached the barrier. Each repair process are members of the wait group. A counter is incremented when starting a leaf repair to signalize that a member of the wait group has started. Completion of leaf repairs is signalized by decrementing the counter. Each repair process inspects the counter upon completion to see if all repair processes has reached the barrier, and if the inspecting process is the last repair process. Fulfilling those requirements enable proceeding of reconciliation by rebuilding the repaired tree.

Barriers are usually implemented with locks since using the synchronization method with multi-threaded processes accessing the same global variable require locks to avoid race conditions and incorrect updates of the global variable. JavaScript has a concurrency model[9] based on event loops using message queues at runtime for processing. Messages are processed entirely before starting another message process. This differs from programming languages as C that can run code in multiple threads and experience thread context switch, meaning that the thread can be halted at any moment to run another thread. The feature of processing messages completely avoids multiple-threads fighting over the same resource. The barrier implementation is therefore lockless, as there are no need for them due to JavaScript primitives.

4.1.4 Garbage Collector: Freezer

Delete operations are required to completely attain user control. Data objects take in a revocation certificate upon deletion. The clause is appended to the additional properties in the header set. The sequence identifier is incremented to indicate data modification in the data-service. The next reconciliation process propagates the order to the DSU and is later identified by the server-side garbage collector. The combination of periodic reconciliation propagating data and garbage collector conducting the deletes, enables distributed deletes with relaxed consistency.

The Freezer is the alias for garbage collector in PM Reporter. The Freezer has a map of key-value pairs containing information related to deletion of a data object. The key is assigned to the header identifier, and functions as a reference to the data objects contained in the dataset. The value includes the day of the week the pertaining data object receives the delete certificate. The day is a number between 0-6, where 0 represents Sunday, and 6 Saturday. The Freezer functions as a time capsule to delay the delete process until the garbage collector is executed on server-side. The approach is naive and further discussion on that subject is in subsubsection 7.2.4.

The Freezer periodically traverse the map and selects data objects with dissimilar day to current day. Dissimilar days indicate the time capsuled object

9. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

has exceeded the time span bound to the capsule, which is 24 hours. Client and server synchronization during the time span of marking the data objects with delete and initiating the cleaning process signify the data object with certificate is synchronized to server and deleted by the garbage collector. It is therefore safe to remove the data objects from the freezer and the dataset. The object is first removed from the memory stored dataset, then the entire new dataset are persisted to disk in an immutable manner.

4.2 MerkleTS: A TypeScript Based Library

The Merkle tree library was created to feature the data-service with data structure representing a Merkle tree. The library is written in TypeScript, which is a superset of JavaScript giving the option of static typing. TypeScript supports features such as classes from the object-oriented paradigm. Class-based programming includes the ability to extend classes using the object-oriented pattern of inheritance.

The library consists of two primary classes; MerkleTree and MerkleNode. Tree nodes are accessed by traversing from the root property of the MerkleTree class. This property is a type of MerkleNode and has further references to deeper tree nodes.

Avoiding recalculation of hash values requires storage of currently existing hash values. The MerkleTree class has a property that ensembles these hash values into a flat list. This flat list represents a serialized format that is acceptable for storage. The serialization process in the library is intervened in between hash generation. Contrary, to the DSU that processes the serialization in an independent process. The combination used avoids unnecessary tree traversal.

The data-service access the flat list for storage purposes through accessors in MerkleTree. The flat list property requires accessors due to it being a private member of the object. Private modifiers entail that the member cannot be accessed outside of the class. However, the data-service needs member access to use the library. Getters and setters are supported accessors enabling fine-grained control of members in the object. The accessors intercepts the access through customizable code, which enables the possibility of adding restrictions or other semantics during a member access. The control is an exclusive feature giving an advantage compared to public modifiers that are accessible outside the class without accessors. The MerkleTree class has no set accessors, and is therefore automatically inferred to be readonly.

4.2.1 Cryptographic Hash Function

The cryptographic function used in MerkleTS is SHA-256 from CryptoJS, as it is the most supported. The hash function has the smallest footprint of the SHA-2 family while still being collision resistance. The small footprint resonates with the limited resources presented in a mobile device. The SHA-2 function from CryptoJS requires a string type for the message being the hash function input. The JSON-object is therefore converted to a string by using the Javascript JSON method stringify before computing the hash value.

The MerkleNode contains property representing the hash value generated from concatenating child nodes or computing the data object if leaf. The hash values are represented and operated in the library as hexadecimal strings. However, operating with word arrays are optimal, avoiding time and computation spent on conversion. Additionally, word arrays would take up less space when persisted on device. Word array is an exclusive datatype defined and used in CryptoJS. It represents an array of datatype numbers.

Concatenating two hash values by simply adding the two hex representations of the hash values results in incorrect hash value when hashing the sum. The datatype for the hex representation is a string, which has no fixed size. To achieve correct hash concatenation, the arithmetic operation of addition has to use word arrays as addends to preserve the fixed size. Alternatively, convert the concatenated hex string to a word array transposing into the fixed size.

Selecting efficient hash function is relative to each computing entity. SHA256 is faster on 32-bit hardware, since internal state of chunks are 32-bit, while SHA512 is faster on 64-bit due to chunks being 64-bit. PMSys consists of web-, mobile-, and server applications. Each application can hold a replica and is required to use identical hash function to construct alike trees. The mobile device has generally greater limitations than the other computing entities due to size. The computing power of the least greatest device needs consideration when creating mirrored trees. The ARM architecture included the 64-bit processors for smartphones and tablets in 2012[10] and personal computers were introduced to 64-bit CPUs in 2003 with x86-64 processors. The trend of 64-bit CPUs increase the viability of changing hash function from SHA256 to SHA512. However, in practice SHA256 and SHA512 has the same secure features since there are currently no computers recorded to produce a collision. Additionally, SHA256 provides a smaller hash, leading to less space occupied for storage and transmission, resulting in less to compute. The hash value can be trimmed after generation to avoid these extraneous costs and comply with

10. https://www.arm.com/about/newsroom/arm-launches-cortex-a50-series-the-worlds-most-energy-efficient-64-bit-processors.php

a desired size. The theoretically proof of collision in SHA shows that SHA512 is more collision resistance than SHA256 [50].

Additionally, research has shown new hash functions as presented in BLAKE2 that experience improved performance to SHA-2 hash functions [51]. BLAKE2 hash functions, BLAKE-256 and BLAKE-512, are evaluated as being faster than SHA-256 and SHA-512. Such functions could potential improve efficiency of reconciliation if the function is supported.

5 Server-side Integration

This chapter describes the server-side integration in the TSU to support Swiftmend. The TSU is a server application written in Go built with the HTTP web framework Gin[1]. Go has feasible network support through the language and libraries. E.g, the encoding/json library easing the serializing and deserializing of JSON objects. The DSU is extended with new endpoints to support Merkle tree, as described in Section 5.1. The Merkle tree structure is supported with a Go library[2] presenting essential tree functionalities to construct the hash tree. The library is modified to support nil pointers instead of duplicated tree nodes. Section 5.2 describe the process of persisting trees in the server application. Lastly, Section 5.3 explains the garbage collector implemented for relaxed deletes.

5.1 Application Programming Interface

The API extensions provide routes for the synchronization orchestrator to maintain the server persisted tree. The server application received two extending endpoints to support new functionality:

/merkle build or rebuild tree.

1. https://gin-gonic.github.io/gin/
2. https://github.com/cbergoon/merkletree

/merkle:id tree node lookup.

The TSU is consolidated with the DSU, enabling the TSU to directly access the DSU collection containing user data.

The route handler that process tree builds starts loading the content, and later supplies the content to the leaf nodes. All documents belonging to the data owner are fetched from the DSU collection, similarly to previously described. The tree is constructed by using the provided content. The data structure is later serialized to an acceptable storing format and flushed to disk in an independent mongoDB collection.

The last route handler traverse the trees to verify a tree node existence. The handler starts loading the tree content from the DSU and the serialized tree from the tree collection. New tree is created in the case of no persisted tree found, and will further serialize and flush the new tree. Deserialization is executed when a persisted tree exists and reconstructs the persisted tree. This is done by combining the fetched content and serialized tree. A hash tree structure is returned when successfully completed, and is ready for traversal. The route receives corresponding hash value to the tree node through a parameter. The hash value is received from the client as a hexadecimal representation. The first step is to decode the hexadecimal string to byte arrays to interoperate with the data structure used. The tree structure is traversed recursively and compares the two byte arrays representing the hash value from both client and server tree.

5.2 Tree Storage

Trees are expensive to construct and impractical to store in memory since each athlete has their own tree structure that grows with the amount of data stored. Persisting the data structures on disk offloads the memory and computation power generating the hash values for each tree node. The server restores trees instead of rebuilding them in each session. A tree is flushed to disk after each completion of a tree build or rebuild. A build only occurs at the initial state of a tree, while the rebuild process occurs upon leaf add or removal to update the tree to current state.

The data structures are serialized to the storable BSON/JSON-format[3] expected by the mongoDB. The serialization is executed in an independent process. The tree is traversed extracting the hash value and parent hash value contained in

3. https://www.mongodb.com/json-and-bson

each tree node. JSON documents are represented in binary-encoded format BSON when stored in mongoDB. BSON is efficient for encoding and decoding JSON documents. The computed hash values are outputted and managed as raw bytes in the server application to complement this. Hexadecimal represented hash values are presentable and human readable. However, storing raw bytes requires less space and averts the overhead of hexadecimal encoding. The SHA-256 value contains 32 significant bytes, impotent of compression. It requires two hexadecimal digits to store a byte. A hexadecimal representation of the 256 bit long hash (sha256) is 64 digits long, since each digit codes for 4 bits. The string representation would need $256/4 = 64$ characters, each character being 4 bit.

The deserialization process extracts the data structure from the saved BSON/JSON-object and uses that to reconstruct the tree. Tree nodes are reallocated and restored to current state. Restoring node state includes assigning correct hash value, and link to appropriate neighbors. Saving hash values enable bypass of hash computation during tree reconstruction.

5.3 Garbage Collector

The server-side garbage collector is implemented as a time scheduled task, cron job. The task scheduler, cron, is supported with a go implemented package[4] with cron spec parser and job runner. The server creates a cron job upon launch scheduled to midnight to avoid user activity. The job is invoked in independent goroutines asynchronously, which avoids code stalling and preserves server activity.

The job is provided a database connection upon execution. The job removes all documents found in the user data collection matching the selector document, by using the object key representing the delete certificate as selector.

4. github.com/robfig/cron

6 Experiments

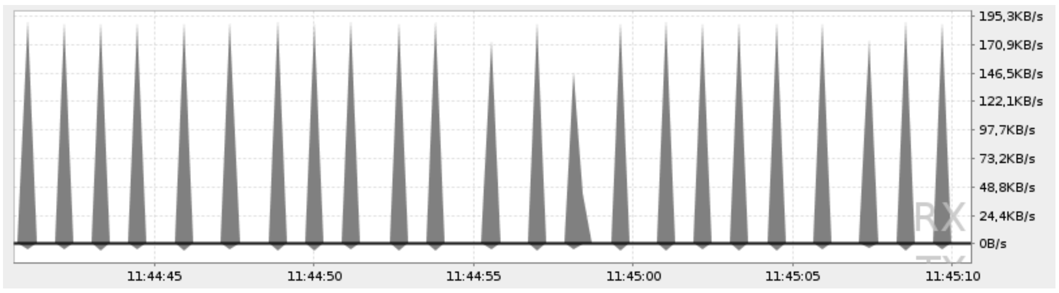
This chapter investigates the thesis in Section 1.2: if a Merkle tree data structure can efficiently reduce the I/O over the network. Section 6.2 evaluates the two reconciliation algorithms; Rejuvenation and Regrowth. Observations show an increase of bytes sent and received when reconciling consistent replicas using Rejuvenation between different loads. Contrary, Regrowth maintains the same amount of bytes sent and received on different loads, due to compact fingerprints indicating replicas consistency.

6.1 Setup

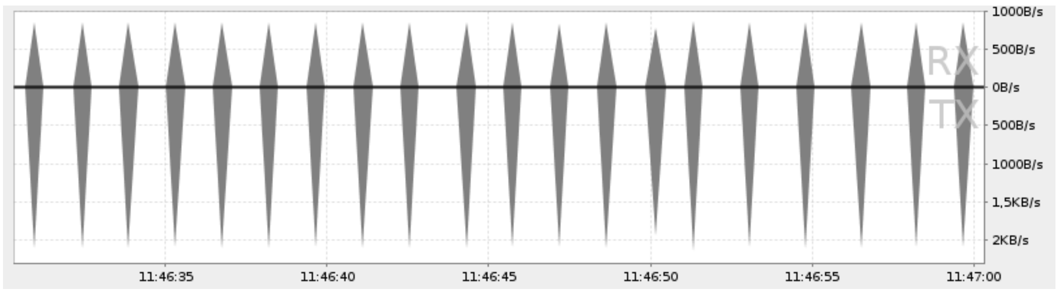
The experiment is conducted with the hybrid application, PM Reporter, deployed on tablet device Lenovo TB3-710F running Lollipop 5.0.1. The device is equipped with four cores running at 1.3 GHz, memory storage of 8 GB and RAM of 1 GB. The tablet is connected to a wireless network with connection speed 65 Mbps and 2.4 GHz band. Measurements are recorded using Android Monitor in Android Studio 2.3.3, and the standalone tool Android Device monitor.

The TSU is running in an Intel server blade S1200SP with Ubuntu 16.04. The machinery is equipped with a Intel Xeon E3-1270 v6 processors running at 3.8 GHz and 64 GB of DDR4 RAM running at 2133 MHz.

Conducting accurate experiments require sequential code execution to preserve



(a)

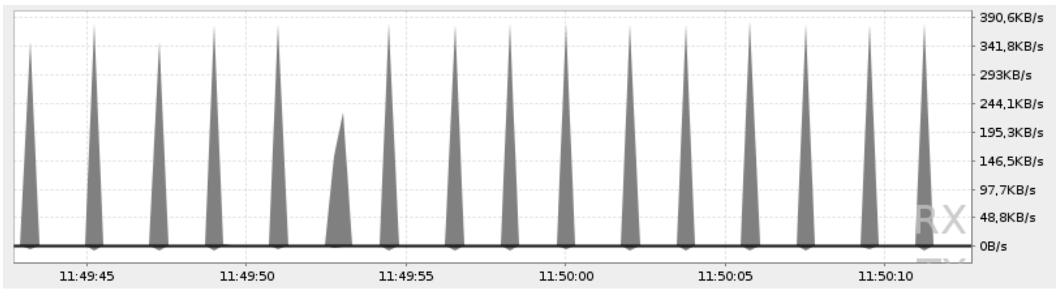


(b)

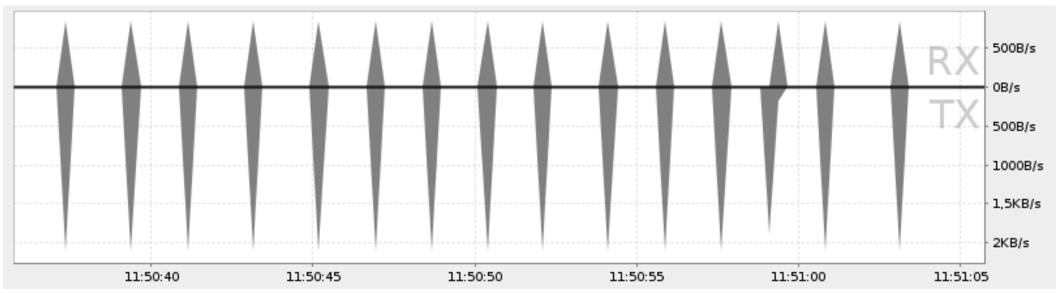
Figure 6.1: (a) Rejuvenation: Received packets (RX) and transmitted packets (TX) on workload=100. (b) Regrowth: Received packets (RX) and transmitted packets (TX) on workload=100.

intentional orders. E.g, measuring process efficiency on a fixed workload demands the proper setup before executing the reconciliation process. JavaScript features asynchronous code execution that has no guarantee that the setup of datasets completes before starting off the process. The problem is experienced in the non-blocking operations using Observables. The experiments are therefore forced to simulate sequential code execution by modifying an asynchronous function to behave synchronous. This is achieved by having an await expression included in the asynchronous function. The code execution is halted and waits for the Promise's resolution.

The number of replicas examined in this chapter are two, with one present in PM Reporter and one in the DSU. The datasets are created in each experimental iteration before initiating the experiment, and flushed after obtaining the observation.



(a)



(b)

Figure 6.2: (a) Rejuvenation: Received packets (RX) and transmitted packets (TX) on workload=200. (b) Regrowth: Received packets (RX) and transmitted packets (TX) on workload=200.

6.2 I/O Traffic

This experiment records the network traffic of Rejuvenation and Regrowth when performing data reconciliation on consistent replicas. The experiment measure transmitted (TX) and received network packets (RX) over time. The experiment is executed upon different workloads, in regards to dataset size. Figure 6.1a and 6.1b reconciliation is done on a consistent dataset of 100 elements, while Figure 6.2a and 6.2b runs with a dataset of 200 elements.

Rejuvenation spikes at approximately 190 KB/s in Figure 6.1a regarding packets received (RX). Rejuvenation receives the entire dataset replicated upon each reconciliation phase. However, the footprint of transmitted packages (TX) are merely visible due to it being single requests asking for the entire dataset. The requests mostly consist of metadata.

Regrowth has a small footprint in received packets consisting of approximately 800 B/s. Instead of receiving the entire dataset replicated at the server, Regrowth receives packets with metadata and the attached hash root for verification purposes. The hash root is represented as string with 64 characters, being 64 B long. However, the transmitted packages in Regrowth are larger than the packages in Rejuvenation due to it consisting of the hash root being transmitted to the server for verification.

The small footprint of Regrowth remains the same when increasing the workload of the dataset to 100 elements, shown in Figure 6.2b. The singular variety captured when increasing the workload was the receiving packets in Rejuvenation, showing an increase to approximately 380 KB/s received in Figure 6.2a. As the dataset grows, requesting the entire dataset increases the data usage.

6.3 Reconciliation Time

The experiment measures the reconciliation time when increasing the dataset size. The datasets are reconciled with a single datapoint being outdated in PM Reporter's dataset. PM Reporter invokes a single datapoint update on each workload measured. Each measurement of a dataset size is calculated from thirty reconciliation executions due to statistical significance.

Figure 6.3 shows that Regrowth do not meet Requirement 2 in Section 1.2, which describes the requirement of synchronizing data within a second. Regrowth exceeds the requirement on 116 data elements, and continue to increase as the dataset grows. Regrowth issue each local traversed branch to the DSU for verification. Each branch is issued with a request, and thus results in an increase of requests issued to the DSU as the tree and dataset grows. Each request's roundtrip time adds to the total latency, and thus increase the reconciliation time. Additionally, as the tree grows in height the more requests are required to search the tree, and thus results in roundtrip delays increasing the difference in the higher limit and lower limit of the confidence interval.

Rejuvenation complies with the one second synchronization requirement as shown in Figure 6.3. Rejuvenation computes the repair locally with both replicas in memory. The replica in the DSU is issued with one requests, and thus do not have roundtrip latency. However, keeping replicas for reconciliation in memory will cause high memory usage as data grows.

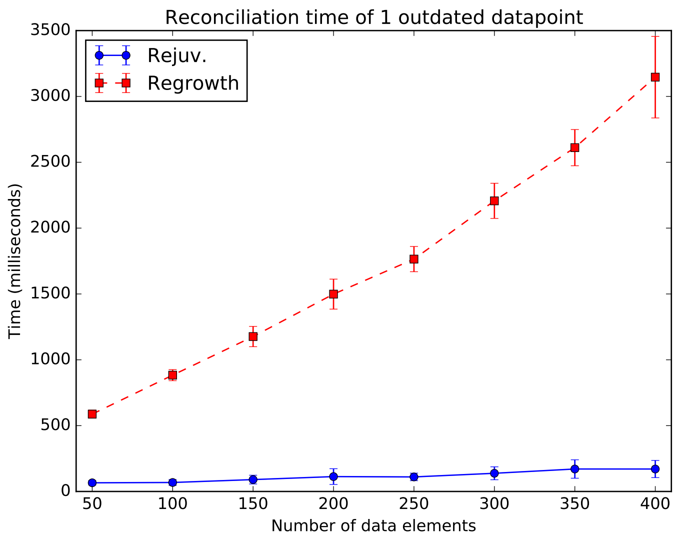


Figure 6.3: Reconciliation time of Rejuvenation and Regrowth.

7 Concluding Remarks

This chapter summarizes and concludes Swiftmend and the evaluation of the proposed reconciliation algorithms. Lastly, we describe potential areas for exploration in the future, as optimizations in both client and server for increased efficiency.

7.1 Summary

This thesis presents Swiftmend, a data synchronization system for OMH applications with restricted connectivity. Swiftmend is based on PMSys 3.0 architecture, and extends the legacy synchronization protocol with two-way synchronization and enable complete support of CRUD operations. We evaluated Swiftmend on the statement made in Section 1.2 and conclude that Swiftmend partially accomplish the statement and requirements. As Regrowth breach Requirement 2 given in Section 1.2.

Swiftmend includes two data reconciliation algorithms; Rejuvenation and Regrowth. Both features repair of missing and outdated data with idempotent difference of replicas, enabling two-way synchronization. Rejuvenation is based on the legacy synchronization protocol. Regrowth differs as it use Merkle trees to reconcile replicas. The evaluation of Rejuvenation show a large data usage when reconciling consistent replicas as the network package grows in size. Rejuvenation execute the idempotent difference of datasets in PM Reporter, and therefore needs to receive all desired replicas. Receiving entire datasets leads to the high data usage. Contrary, Regrowth distinctly show low data usage using compact fingerprints of small size to verify consistency, and thus accomplish the thesis statement.

To achieve Requirement 1 deduced from the thesis statement, Swiftmend implemented intervals of ten minutes in the Synchronization Orchestrator to invoke periodical reconciliation.

The evaluation of Rejuvenation and Regrowth show data synchronization performed in Rejuvenation was within a second, and therefore achieves Requirement 2. Regrowth experience request roundtrip that cause latency after dataset size of 116, and thus exceeds the one second requirement of synchronization.

Swiftmend extends the legacy data structure in PMSys to support complete CRUD operations. Replica updates are propogated with reconciliation, and thus enable the CRUD support. Lastly, Swiftmend implements a garbage collector in PM Reporter and the TSU to enable deletions. These features summed up achieve the final, Requirement 3.

7.2 Future Work

The data sizes experimented on in this thesis are small, Subsection 7.2.1 discusses the applicability of Swiftmend towards big data.

While Swiftmend reduced data usage with Regrowth, there relies potential in the Merkle tree for improving the efficiency of the current design and implementation of Regrowth. Such improvements are presented in Subsection 7.2.2 and Subsection 7.2.3.

Subsection 7.2.4 discusses alternative solutions to garbage collection in the TSU.

7.2.1 Big Data

A phenotypic datapoint of type SRPE has a size of 446 B, while the type wellness has 494 B. The current size of data do not stress the system performance or drain expensive amount of paid data. Magnus et al. [1] presents the breakdown of storage transactions per month, and data of type SRPE and wellness produce less than 1 KB per month. However, when adding additional data of interest the total transaction cost per month is 2.6 GB. This includes video notations, nutritions and physical parameters. Additional data of interest is sensor data measured by attached devices on an athletes body [3]. The ensemble of this data has the characteristics of big data [52] as the volume includes all these sources, the variety differs as data has different formats, and the velocity is related to both the video notations and sensor data as they are in real-time.

Swiftmend want to optimize for these characteristics by providing efficient reconciliation with Regrowth using Merkle trees. However, the choice of database has to be reconsidered in such scenario as crash test show corruptions and inconsistencies for application with objects as video data [40]. As more unstructured data is required for synchronization more considerations are necessary regarding the database medium responsible for consistency during crashes.

7.2.2 Merkle Tree

The Merkle Tree Traversal Problem

A fundamental problem related to Merkle tree is the construction time [35]. There is vast research conducted in regards to this [53].

Szydlo [54] improved the traversal by improving the algorithm to reduce memory requirements by reducing active tree hash instances during tree construction.

Fractal Merkle tree traversal splits the tree into smaller subtrees and saves computation by construction these subtrees instead of single nodes [55].

Balance Trade-off

Regrowth has a datapoint/leaf ratio of 1. The ratio is described as being impractical as the tree size grows [37]. Achieving the most optimal efficiency and accuracy is highly dependent on the trade-off between tree size and false positives in objects needing repair. A poor balance between the trade-off results in high communication costs. Storing a key-range of multiple key-value objects instead of a single object would reduce the tree size. However, the large amount of key-value objects in the key-range needs to be exchanged upon repair. Having a large tree results in exchanging lots of metadata to identify inconsistent objects needing repair. Kruber et al. implements Nye's trie [45] in their leaf nodes instead of key-range of objects. Nye's trie is a modified burst trie.

7.2.3 PM Reporter

Synchronization Orchestrator

PM Reporter transfer compact network packets upon reconciliation. Each branch verification sends a single request, which results in decreased performance due to roundtrips. It is desired to reduce the latency introduced by roundstrips, while still maintaining compact packets to preserve the small packet size.

One solution is Jon's reconciliation algorithm that reduce the amount of requests by transferring entire trees in a single request. PM Reporter is forced to solve the tree difference locally. Though, this approach is non-considerate towards packet size as the trees grow and therefore the network packet containing the tree will grow. Kruber et al., solves improved upon this by bundling all hash values within an inconsistent tree level to each request. The network packets achieve compactness compared to Jon's reconciliation, and avoids unnecessary small packets used in Swiftmend.

MerkleTS

Similarly to Section 5.2, by managing hash values in raw bytes instead of hexadecimal strings will require less space and avert the overhead of hexadecimal encoding.

The support grows for other hash algorithms as BLAKE, it would be worth experimenting on replacing it with SHA to discover potential optimizations.

Additionally, implementing dynamic hash function support steered by detection of device information related to hardware being either 64-bit or 32-bit. This would enable the hardware to run the most optimal algorithm. Though, this would require consensus between other users when sharing replicas, as they need to determine which algorithm to use.

7.2.4 TSU

Each request regarding tree functionality in the DSU spends unnecessary time deserializing, serializing, and constructing. Each request has to load the tree from disk upon handling. This is costly compared to retrieving a tree from memory. Sessions will increase the efficiency by keeping the tree in memory for each user session.

Another caching technique for optimization is introducing hot trees. Hot trees are cached trees being the most frequently accessed. When sharing trees between coach and athlete, both users will access the same tree. Such scenarios will create an increase in frequency on the pertaining tree.

Garbage Collector

The garbage collector implementation is naive as it fails to address loss of causal information [41]. An example is the assumption that client replicas has synchronized before initiating the delete process. Delayed messages and synchronization with outdated clients will cause deleted objects to reemerge.

A simple solution is implementing a logging service in the server that records synchronization of client replicas. The logger can identify synchronized clients and invoke garbage collection upon consensus to safely delete replicas. The objects are forced to carry the tombstone to preserve causality. The additional metadata results in a linear growth in space consumption per tombstone [56]. This will lead to a vast amount of waste over time, which is rather space inefficient.

Another solution is having hybrid consistency. All updates except deletes in Swiftmend are propogated with weak consistency, while the deletes are periodically enforced with strict consistency. A two phase or three phase commit upon consensus would enforce deletion with strict consistency [57]. Locks featured in these approaches imply code stalling. To behave transparently for users, the garbage collector requires a schedule that averts user activity. Such as running at midnight or other inactive periods.

Bibliography

[1] M. Stenhaug, H. Johansen, and D. Johansen, "Transforming healthcare through life-long personal digital footprints," in *the IEEE Conference on Connected Health: Applications, Systems and Engineering Technologies: The 1st International Workshop on Cloud Connected Health*, no. CHASE '16, IEEE, June 2016.

[2] H. Johansen, C. Gurrin, and D. Johansen, "Towards consent-based lifelogging in sport analytic," in *MMM 2015, Part II*, no. 8936, pp. 335–344, Springer International Publishing, Jan. 2015.

[3] H. D. Johansen, W. Zhang, J. Hurley, and D. Johansen, "Management of body-sensor data in sports analytic with operative consent," in *the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, IEEE, Apr. 2014.

[4] H. Tangmunarunkit, C.-K. Hsieh, B. Longstaff, S. Nolen, J. Jenkins, C. Ketcham, J. Selsky, F. Alquaddoomi, D. George, J. Kang, *et al.*, "Ohmage: A general and extensible end-to-end participatory sensing platform," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 3, p. 38, 2015.

[5] T. T. Hoang, "pmsys: Implementation of a digital player monitoring system," Master's thesis, 2015.

[6] D. Estrin and I. Sim, "Open mhealth architecture: an engine for health care innovation," *Science*, vol. 330, no. 6005, pp. 759–760, 2010.

[7] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[8] F. Schneider, "Untitled textbook on cybersecurity. chapter 9: Credentials-based authorization," 2013.

[9] J. Petersen, "Benefits of using the n-tiered approach for web applications," *URL: http://www. adobe. com/devnet/coldfusion/articles/ntier. html*, 2001.

[10] P. Bailis and K. Kingsbury, "The network is reliable," *Queue*, vol. 12, no. 7, p. 20, 2014.

[11] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, 2000.

[12] C. Hale, "You can't sacrifice partition tolerance," *codahale. com*, 2010.

[13] D. Pritchett, "Base: An acid alternative," *Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[14] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, "Simba: Tunable end-to-end data consistency for mobile apps," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 7, ACM, 2015.

[15] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992.

[16] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Queue*, vol. 11, no. 3, p. 20, 2013.

[17] N. T. Bailey *et al.*, *The mathematical theory of infectious diseases and its applications*. Charles Griffin & Company Ltd, 5a Crendon Street, High Wycombe, Bucks HP13 6LE., 1975.

[18] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[19] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, (New York, NY, USA), pp. 1–12, ACM, 1987.

[20] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Computer*, vol. 22, no. 2, pp. 63–70, 1989.

[21] D. Johansen, R. Van Renesse, and F. B. Schneider, "Operating system sup-

port for mobile agents," in *Hot Topics in Operating Systems, 1995.(HotOS-V), Proceedings., Fifth Workshop on*, pp. 42–45, IEEE, 1995.

[22] D. Johansen, H. Johansen, and R. van Renesse, "Environment mobility: moving the desktop around," in *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pp. 150–154, ACM, 2004.

[23] D. Johansen, R. Van Renesse, and F. B. Schneider, "Waif: Web of asynchronous information filters," in *Future directions in distributed computing*, pp. 81–86, Springer, 2003.

[24] H. D. Johansen and D. Johansen, "Improving object search using hints, gossip, and supernodes," in *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pp. 336–340, IEEE, 2002.

[25] R. Pettersen, S. V. Valvag, A. Kvalnes, and D. Johansen, "Jovaku: Globally distributed caching for cloud database services using dns," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*, pp. 127–135, IEEE, 2014.

[26] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, "Fireflies: A secure and scalable membership and gossip service," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 2, p. 5, 2015.

[27] R. van Renesse, H. Johansen, N. Naigaonkar, and D. Johansen, "Secure abstraction with code capabilities," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pp. 542–546, IEEE, 2013.

[28] H. D. Johansen, E. Birrell, R. Van Renesse, F. B. Schneider, M. Stenhaug, and D. Johansen, "Enforcing privacy policies with meta-code," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, p. 16, ACM, 2015.

[29] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, Ø. Landsverk, *et al.*, "Bagadus: An integrated real-time system for soccer analytics," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 10, no. 1s, p. 14, 2014.

[30] D. Johansen, M. Stenhaug, R. B. Hansen, A. Christensen, and P.-M. Høgmo, "Muithu: Smaller footprint, potentially larger imprint," in *Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pp. 205–214, IEEE, 2012.

[31] D. Hardt, “The oauth 2.0 authorization framework,” 2012.

[32] G. H. Forman and J. Zahorjan, “The challenges of mobile computing,” *Computer*, vol. 27, no. 4, pp. 38–47, 1994.

[33] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, “An analysis of data corruption in the storage stack,” *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 8, 2008.

[34] R. C. Merkle, “A digital signature based on a conventional encryption function,” in *Conference on the Theory and Application of Cryptographic Techniques*, pp. 369–378, Springer, 1987.

[35] G. Becker, “Merkle signature schemes, merkle trees and their cryptanalysis,” *Ruhr-University Bochum, Tech. Rep*, 2008.

[36] J. Carpenter and E. Hewitt, *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. " O’Reilly Media, Inc.", 2016.

[37] N. Kruber, M. Lange, and F. Schintke, “Approximate hash-based set reconciliation for distributed replica repair,” in *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*, pp. 166–175, IEEE, 2015.

[38] S. Dustdar and W. Schreiner, “A survey on web services composition,” *International journal of web and grid services*, vol. 1, no. 1, pp. 1–30, 2005.

[39] G. D. P. Regulation, “Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46,” *Official Journal of the European Union (OJ)*, vol. 59, pp. 1–88, 2016.

[40] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, “Reliable, consistent, and efficient data sync for mobile apps.,” in *FAST*, vol. 15, pp. 359–372, 2015.

[41] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[42] M. Naor and K. Nissim, “Certificate revocation and certificate update,” *IEEE Journal on selected areas in communications*, vol. 18, no. 4, pp. 561–570, 2000.

[43] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,”

Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[44] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient reconciliation and flow control for anti-entropy protocols," in *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, (New York, NY, USA), pp. 6:1–6:7, ACM, 2008.

[45] J. Trutna, D. A. Patterson, and A. Fox, *Nye's Trie and Floret Estimators: Techniques for Detecting and Repairing Divergence in the SCADS Distributed Storage Toolkit*. 2010.

[46] D. C. Schmidt, "Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching," 1995.

[47] P. Padmanabhan, L. Gruenwald, A. Vallur, and M. Atiquzzaman, "A survey of data replication techniques for mobile ad hoc network databases," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 17, no. 5, pp. 1143–1164, 2008.

[48] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 397–400, ACM, 2010.

[49] T. S. Axelrod, "Effects of synchronization barriers on multiprocessor performance," *Parallel Computing*, vol. 3, no. 2, pp. 129–140, 1986.

[50] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang, "Preimages for step-reduced sha-2," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 578–597, Springer, 2009.

[51] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "Blake2: simpler, smaller, fast as md5," in *International Conference on Applied Cryptography and Network Security*, pp. 119–135, Springer, 2013.

[52] M. Hilbert, "Big data for development: A review of promises and challenges," *Development Policy Review*, vol. 34, no. 1, pp. 135–174, 2016.

[53] J. Buchmann, E. Dahmen, and M. Schneider, "Merkle tree traversal revisited," in *International Workshop on Post-Quantum Cryptography*, pp. 63–78, Springer, 2008.

[54] M. Szydlo, "Merkle tree traversal in log space and time," in *International Conference on the Theory and Applications of Cryptographic Techniques*,

pp. 541–554, Springer, 2004.

[55] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo, "Fractal merkle tree representation and traversal," in *Cryptographers' Track at the RSA Conference*, pp. 314–326, Springer, 2003.

[56] R. J. T. Gonçalves, P. S. Almeida, C. Baquero, and V. Fonte, "Dotteddb: Anti-entropy without merkle trees, deletes without tombstones," in *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on*, pp. 194–203, IEEE, 2017.

[57] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems," 1987.