Approximate Distributed Set Reconciliation with Defined Accuracy

Dissertation

zur Erlangung des akademischen Grades

Dr. rer. nat.
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Dipl.-Inf. Nico Kruber

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter/in:

1. Prof. Dr. Alexander Reinefeld
2. Prof. Dr. Nicole Schweikardt
3. Prof. Dr. Artur Andrzejak

Tag der Verteidigung: 1. November 2019

Ich widme diese Arbeit
meiner Familie und meinen Freunden.

Abstract

The objective comparison of approximate versioned set reconciliation algorithms is challenging. Each algorithm's behaviour can be tuned for a given use case, e.g. low bandwidth or computational overhead, using different sets of parameters. Changes of these parameters, however, often also influence the algorithm's accuracy in recognising differences between participating sets and thus hinder objective comparisons based on the same level of accuracy.

We develop a method to fairly compare approximate set reconciliation algorithms by enforcing a fixed accuracy and deriving accuracy-influencing parameters accordingly. We show this method's universal applicability by adopting two trivial hash-based algorithms as well as set reconciliation with Bloom filters and Merkle trees. Compared to previous research on Merkle trees, we propose to use dynamic hash sizes to align the transfer overhead with the desired accuracy and create a new Merkle tree reconciliation algorithm with an adjustable accuracy target. An extensive evaluation of each algorithm under this accuracy model verifies its feasibility and ranks these four algorithms.

Our results allow to easily choose an efficient algorithm for practical set reconciliation tasks based on the required level of accuracy. Our way to find configuration parameters for different, yet equally accurate, algorithms can also be adopted to other set reconciliation algorithms and allows to rate their respective performance in an objective manner. The resultant new approximate Merkle tree reconciliation broadens the applicability of Merkle trees and sheds some new light on its effectiveness.

Distributed Systems, Set Reconciliation, Approximate Algorithms, Accuracy Models, Merkle Tree, Bloom Filter, Synchronisation, Replication

Zusammenfassung

Mit aktuell vorhandenen Mitteln ist es schwierig, objektiv approximative Algorithmen zum Mengenabgleich gegenüberzustellen und zu vergleichen. Jeder Algorithmus kann durch unterschiedliche Wahl seiner jeweiligen Parameter an ein gegebenes Szenario angepasst werden und so zum Beispiel Bandbreiten- oder CPU-optimiert werden. Änderungen an den Parametern gehen jedoch meistens auch mit Änderungen an der Genauigkeit bei der Erkennung von Differenzen in den teilnehmenden Mengen einher und behindern somit objektive Vergleiche, die auf derselben Genauigkeit basieren.

In dieser Arbeit wird eine Methodik entwickelt, die einen fairen Vergleich von approximativen Algorithmen zum Mengenabgleich erlaubt. Dabei wird eine feste Zielgenauigkeit definiert und im Weiteren alle die Genauigkeit beeinflussenden Parameter entsprechend gesetzt. Diese Methode ist universell genug, um für eine breite Masse an Algorithmen eingesetzt zu werden. In der Arbeit wird sie auf zwei triviale hashbasierte Algorithmen, einem basierend auf Bloom Filtern und einem basierend auf Merkle Trees angewandt, um dies zu untermauern. Im Vergleich zu vorherigen Arbeiten zu Merkle Trees wird vorgeschlagen, die Größe der Hashsummen dynamisch im Baum zu wählen und so den Bandbreitenbedarf an die gewünschte Zielgenauigkeit anzupassen. Dabei entsteht eine neue Variante des Mengenabgleichs mit Merkle Trees, bei der sich erstmalig die Genauigkeit konfigurieren lässt. Eine umfassende Evaluation eines jeden der vier unter dem Genauigkeitsmodell angepassten Algorithmen bestätigt die Anwendbarkeit der entwickelten Methodik und nimmt eine Neubewertung dieser Algorithmen vor.

Die vorliegenden Ergebnisse erlauben die Auswahl eines effizienten Algorithmus für unterschiedliche praktische Szenarien basierend auf einer gewünschten Zielgenauigkeit. Die präsentierte Methodik zur Bestimmung passender Parameter, um für unterschiedliche Algorithmen die gleiche Genauigkeit zu erreichen, kann auch auf weitere Algorithmen zum Mengenabgleich angewandt werden und erlaubt eine objektive, allgemeingültige Einordnung ihrer Leistung unter verschiedenen Metriken. Der in der Arbeit entstandene neue approximative Mengenabgleich mit Merkle Trees erweitert die Anwendbarkeit von Merkle Trees und wirft ein neues Licht auf dessen Effektivität.

Verteilte Systeme, Mengenabgleich, Approximative Algorithmen, Genauigkeitsmodelle, Merkle Tree, Bloom Filter, Synchronisation, Replikation

Contents

Chapter 1

Motivation & Outline

Set reconciliation is a common tool for distributed systems with replicated data that may diverge over time. This includes diverse examples such as resource discovery and failure detection [33, 37, 85, 69], content delivery and mobile data [11, 80], cryptocurrency block updates [18, 84, 65, 66], and data replicated inside cloud providers' architectures [22, 12]. Amazon [22] and Azure [12], for example, use loosely synchronised replicas and further copies may exist at the users' (partly disconnected) devices. Eventually, these need to be synchronised, e.g. by a periodic anti-entropy protocol, where differences are identified by a set reconciliation algorithm. Commonly, approximate algorithms based on Bloom filters [3, 26, 36] or Merkle trees [52, 10] are used to reduce the computational complexity or the transfer costs compared to an exact algorithm.

Despite the variety of approximate set reconciliation methods proposed in literature, these algorithms have not been compared well in practice. This is partly due to different parameters having diverse effects on both the algorithms' costs and the resulting accuracy, making approximate algorithms difficult to compare in general. Existing comparisons commonly fix the transfer costs (per item), i.e. data structure sizes, and compare the accuracy of the algorithms [11, 56]. Some also design their algorithm to a fixed accuracy [36] or manually find appropriate parameters to compare with equally accurate algorithms [26]. To the best of our knowledge, none offers a generic way of comparing the costs of equally accurate approximate set reconciliation algorithms, although starting from a defined accuracy is better suited for distributed systems with reliability service-level agreements.

Furthermore, approximate algorithms such as Merkle tree set reconciliation [10] may not offer much to tune their accuracy. In this case, only the hash sizes may be changed but due to the adaptiveness of the algorithm, the effects on the accuracy highly depend on the application scenario. For distributed systems with fixed reliability, however, we require algorithms that support defining a fixed accuracy and adapt and tune accuracy-influencing parameters to the current application scenario. A generic accuracy model that allows this also enables a fair comparison of equally accurate algorithms, as desired.

1.1 Goal and Approach

In an extension and refinement of the ideas and techniques of [45], we present a generic accuracy model for approximate set reconciliation algorithms that allows them to set and follow a given accuracy target. We apply this model to four well-known approximate set reconciliation algorithms, i.e. two simple hash-based approaches, one based on Bloom filters, and one on Merkle trees, by deriving accuracy-influencing parameters from a common upper failure bound.

While the data structures of the simple hash-based and the Bloom filter algorithms already contain some accuracy-influencing parameters and—for the latter—previous work was already done to reduce them to just one parameter [27, 56], these need to be applied to a generic set reconciliation embedding under our accuracy model. By applying this model to Merkle tree set reconciliation, however, we also create a new approximate set reconciliation algorithm with adjustable accuracy that adapts to the data it represents just like the Merkle reconciliation itself. For this, we propose to use dynamic hash sizes for each node of the tree to align the transfer overhead with the desired accuracy.

For each of the four algorithms, we show how to apply our accuracy model and derive accuracy-influencing parameters, analyse the resulting algorithm theoretically, and evaluate it in various application scenarios based on an embedding into the replica repair service of the Scalaris [74, 73] key-value store. This shows the feasibility of the accuracy model and compares the four algorithms in a fair manner as part of a versioned set reconciliation. Both the analysis and the evaluation are focussed on the accuracy and the volume of data to be transferred with an additional discussion on the latency where necessary, i.e. for the multi-round Merkle tree reconciliation. Techniques to reduce the computational or maintenance overhead are not in the focus of this work but can be applied orthogonally.

Our results show that our accuracy model can be easily applied to a broad set of set reconciliation algorithms. The presented way of finding parameters to create equally accurate algorithms may be adopted to even more approximate set reconciliation algorithms and makes their accuracy adjustable. This allows an assessment of the performance of these algorithms under the same level of accuracy and thus allows users to choose an efficient algorithm for practical set reconciliation tasks with reliability constraints. Additionally, resultant new approximate set reconciliation algorithms with adjustable accuracy, such as our new approximate Merkle tree reconciliation, broaden the applicability of the original algorithm and shed some new light on their effectiveness.

1.2 Contributions

- We present a method to fairly compare approximate set reconciliation algorithms (Section 2.5) by enforcing a fixed accuracy in terms of the expected failure rate.

- We apply this method to four different approximate set reconciliation algorithms and deduce each algorithm's accuracy-influencing parameters accordingly (Chapters 5 to 7).

- We develop a new *approximate* variant of a Merkle tree based set reconciliation algorithm [14] with adjustable accuracy by applying our accuracy model. This algorithm reduces transfer costs by dynamically adopting Merkle hash lengths (Chapter 8). Additionally, costs of bucketing items in leaf nodes are mitigated which makes it beneficial in more scenarios.

- We evaluate the four different set reconciliation algorithms in terms of their bandwidth usage, accuracy, and scalability (Sections 5.6, 6.6, 7.7 and 8.7). For reference, the naïve but exact algorithm for set reconciliation as well as an approach using rsync are presented and evaluated the same way (Chapter 4).

- We provide a comparative evaluation of the four approximate set reconciliations and rsync that identifies which algorithms are most suitable for which situations (Chapter 9). Our optimised Merkle reconciliation is not only more efficient for very low differences (as expected), compared to the approximate *trivial* reconciliation, but also has lower transfer costs for differences up to 50 %. It is thus also suitable for situations where Merkle trees have been deemed inefficient in the past [14].

1.3 Outline

Chapter 2 gives a brief introduction into the set reconciliation problem and introduces the accuracy model we employ. Chapter 3 describes our evaluation method of embedding the set reconciliation algorithms into a generic replica repair service. It also presents the experiments we use and the metrics we collect. Chapter 4 puts our approximate set reconciliation methods in perspective by describing and evaluating the naïve and exact set reconciliation as well as rsync-based approaches using different parameters.

Chapters 5 to 8 present four approximate set reconciliation methods: *trivial*, SHash, Bloom filter, and Merkle tree. For each of these, we present the protocol and data structures of the basic algorithm, the deduction of accuracy-influencing parameters according to the accuracy metric of Section 2.5, the resulting accuracy, related work (where applicable), and an empirical evaluation. A comparative evaluation of these algorithms is provided in Chapter 9 which identifies the most suitable algorithm for each of the different scenarios. Finally, Chapter 10 concludes with a short summary of this work and its results.

Chapter 2

Introduction

2.1 Definitions & Theory

The reconciliation of data stored at different nodes is a common problem in distributed systems and is formalised as follows:

Definition 1 (Set Reconciliation)**.** *Considering a pair of nodes* $\mathtt{A}$ *and* $\mathtt{B}$*, each with a set of items* $S_{\mathtt{A}}$ *and* $S_{\mathtt{B}}$ *(and* $n_{\mathtt{A}} \coloneqq |S_{\mathtt{A}}|, n_{\mathtt{B}} \coloneqq |S_{\mathtt{B}}|$*); find* $\Delta_{\mathtt{A}} \coloneqq S_{\mathtt{B}} \setminus S_{\mathtt{A}}$ *at* $\mathtt{A}$ *and* $\Delta_{\mathtt{B}} \coloneqq S_{\mathtt{A}} \setminus S_{\mathtt{B}}$ *at* $\mathtt{B}$ *with minimal cost.*

Cost may be defined as any combination of (a) communication cost, i.e. number of transferred bits, (b) computational cost, or (c) time complexity, i.e. the number of (causally dependent) message rounds. Minsky and Trachtenberg [53] choose to minimise communication cost which we target as well.

A similar definition [55, 60] lets both nodes determine the set union $S_{\mathtt{A}} \cup S_{\mathtt{B}}$ but is equivalent to the definition above due to the following equalities:

$$S_{\mathtt{A}} \cup S_{\mathtt{B}} = S_{\mathtt{A}} \cup (S_{\mathtt{B}} \setminus S_{\mathtt{A}}) \qquad S_{\mathtt{A}} \cup S_{\mathtt{B}} = S_{\mathtt{B}} \cup (S_{\mathtt{A}} \setminus S_{\mathtt{B}})$$
$$S_{\mathtt{B}} \setminus S_{\mathtt{A}} = (S_{\mathtt{A}} \cup S_{\mathtt{B}}) \setminus S_{\mathtt{A}} \qquad S_{\mathtt{A}} \setminus S_{\mathtt{B}} = (S_{\mathtt{A}} \cup S_{\mathtt{B}}) \setminus S_{\mathtt{B}}$$

A more restricted variation of the above is the *one-way* set reconciliation [41] which only determines the set difference at one node.

> ℹ Set Reconciliation falls into the general problem of *data synchronisation* which establishes consistency among multiple copies of some (set of) data.

We extend the classic set reconciliation from above with a more relevant and broader definition of a *versioned set reconciliation* which assumes that items are mutable and finds the newest versions of all of them. Thus, in the versioned set reconciliation, each item i is associated with an immutable, globally unique identifier key i_k and an item-specific version i_v—along with further data such as the item's value. Item keys may be naturally given by an item's name, an attribute, or a hash of the item's contents, or can be given

artificially, e.g. a storage address, a UUID, or a random number. Versions can be timestamps or any other sort of logical version number with a total order under $\leq_v$. For practical purposes and some of the algorithms, we also assume that keys are totally ordered under an arbitrary order $\leq_k$, e.g. by using the natural order of their integer representation. Further on, we use the common $\leq$ as the context will make clear whether keys or versions are compared.

Definition 2 (Versioned Set Reconciliation)**.** *Consider a pair of nodes* A *and* B*, each with a set of items* S_A *and* S_B *where each item is associated with a key* i_k *and a version* i_v*. Find items that are outdated or missing on* A*, i.e.* Δ_A*, and* B*, i.e.* Δ_B*, at* A *and* B*, respectively, with minimal cost.*

Definition 3 (Missing Items)**.** *Versioned items missing on node* A *(similarly* B*) are given by* $Mis_A \coloneqq S_B \setminus_v S_A \coloneqq \{i \in S_B : \nexists j \in S_A : i_k = j_k\}$.

Definition 4 (Outdated/Newer Items)**.** *Versioned items outdated on node* A *(similarly* B*) are given by* $Old_A \coloneqq \{i \in S_A : \exists j \in S_B : i_k = j_k \wedge j_v > i_v\}$. *Conversely, newer-versioned items on node* B *are given by* $New_B \coloneqq \{i \in S_B : \exists j \in S_A : i_k = j_k \wedge i_v > j_v\}$.

We further define the set differences $\Delta_\mathtt{A} \coloneqq Mis_\mathtt{A} \cup New_\mathtt{B}$, $\Delta_\mathtt{B} \coloneqq Mis_\mathtt{B} \cup New_\mathtt{A}$, $\Delta \coloneqq \Delta_\mathtt{A} \cup \Delta_\mathtt{B}$, $S_{\mathtt{A}_\Delta} \coloneqq Mis_\mathtt{B} \cup New_\mathtt{A} \cup Old_\mathtt{A}$, and $S_{\mathtt{B}_\Delta} \coloneqq Mis_\mathtt{A} \cup New_\mathtt{B} \cup Old_\mathtt{B}$. Additionally, we define the total number of items (irrespective of their version) as $n \coloneqq |S_\mathtt{A} \cup Mis_\mathtt{A}| = |S_\mathtt{B} \cup Mis_\mathtt{B}|$ and the relative difference among them as $\delta \coloneqq {}^{|\Delta|}/_{n}$ (commonly expressed in %). We will also use the expression of a key $k \in X$ equivalently to an item with key k being in X, i.e. $k \in X \Leftrightarrow \exists i \in X : i_k = k$.

Definition 5 (Approximate Versioned Set Reconciliation)**.** *A versioned set reconciliation which finds* Δ'_A *and* Δ'_B *instead of* Δ_A *and* Δ_B *and allows these sets to be different from* Δ_A *or* Δ_B*, respectively, with a failure probability* $P(\Delta'_A \neq \Delta_A \vee \Delta'_B \neq \Delta_B)$ *is called an approximate versioned set reconciliation.*

Similarly, $Old'_\mathtt{A}$, $Old'_\mathtt{B}$, $New'_\mathtt{A}$, $New'_\mathtt{B}$, and Δ' are defined as the recognised difference sets of their counterparts. Algorithms for the approximate versioned set reconciliation may thus show any or both of the following error types:

(a) *unrecognised items of* Δ: items which are different on both nodes or non-existing on one of them but were not identified as such ($\Delta \setminus \Delta'$), and

(b) *redundantly transferred items*: items which were sent to a node although the newest version of the item is already there ($\Delta'_\mathtt{A} \setminus \Delta_\mathtt{A}$ and $\Delta'_\mathtt{B} \setminus \Delta_\mathtt{B}$).

We will use these errors to define an algorithm-independent accuracy model for a fair comparison of approximate versioned set reconciliation algorithms in Section 2.5.

☞ In this work, we consider different algorithms and protocols for their use in the *approximate versioned set reconciliation* problem aiming at minimal communication cost. We find this problem most useful in practical set reconciliation tasks (ref. Section 2.2).

Note that in contrast to our definition of the *approximate* set reconciliation which is commonly used and based on [60, 36, 26, 10], diverging definitions exist, depending on the requirements. Chen et al. [17], for example, define a *robust set reconciliation* where elements are considered equal if their difference is small, e.g. the distance of points in the d-dimensional Euclidean space. This can be used to tolerate different (lossy) compression schemes of images or to tolerate rounding errors in floating-point computations.

2.1.1 Asymptotic Lower Bound

Information-theoretic bounds on the communication costs for the (exact and non-versioned) set reconciliation problem with items of size $b \geq \log_2(|S_\mathtt{A} \cup S_\mathtt{B}|)$ bits have been identified by Minsky et al. [55] who revised their previously erroneous version from [53]. Based on the fact that $\mathtt{A}$ needs to discern $|\Delta_\mathtt{A}|$ b-bit items from the $2^b - |S_\mathtt{A}| = 2^b - |S_\mathtt{A} \cap S_\mathtt{B}| - |\Delta_\mathtt{B}|$ items it may be missing and similarly for $\mathtt{B}$, the lower bound $\hat{C}_\infty$ in bits with no bound on the number of communication rounds and assuming constant $|\Delta| = |\Delta_\mathtt{A}| + |\Delta_\mathtt{B}|$ is given as:

$$\hat{C}_\infty \geq \log_2 \left(\binom{2^b - |S_\mathtt{A}|}{|\Delta_\mathtt{A}|} \cdot \underbrace{\binom{2^b - |S_\mathtt{B}|}{|\Delta_\mathtt{B}|}}_{= \binom{2^b - |S_\mathtt{A} \cap S_\mathtt{B}| - |\Delta_\mathtt{A}|}{|\Delta_\mathtt{B}|} \geq \binom{2^b - |S_\mathtt{A} \cap S_\mathtt{B}| - |\Delta_\mathtt{A}| - |\Delta_\mathtt{B}|}{|\Delta_\mathtt{B}|} = \binom{2^b - |S_\mathtt{A}| - |\Delta_\mathtt{A}|}{|\Delta_\mathtt{B}|}} \right) \tag{2.1}$$

$$\geq \log_2 \binom{2^b - |S_\mathtt{A}|}{|\Delta|} \qquad \text{from } \binom{n}{j} \cdot \binom{n-j}{k} \geq \binom{n}{j+k}$$

$$\geq \log_2 \binom{2^{b-1}}{|\Delta|} \qquad \text{when } 2^b \geq 2 \cdot |S_\mathtt{A}| \text{ or } 2^b \geq 2 \cdot |S_\mathtt{B}|$$
(analogously from eq. (2.1) down with $|S_\mathtt{B}|$)

$$\geq \log_2 \left(\frac{2^{b-1}}{|\Delta|} \right)^{|\Delta|} = |\Delta| \cdot (b - 1 - \log_2 |\Delta|)$$

$$\gtrapprox |\Delta| \cdot b - |\Delta| \cdot \log_2 |\Delta| \tag{2.2}$$

Note that the latter bounds are not necessarily close to the first one. Nevertheless, since $b \geq \log_2 n$ constant, a lower bound complexity class of the exact—versioned or non-versioned—set reconciliation problem is thus given as

$$\mathcal{O}(|\Delta| \cdot \log n) \tag{2.3}$$

A lower bound for the *approximate* set reconciliation problem may be derived from the minimum number of bits required to approximately represent a set as discussed below.

Lower Bound for Approximate Set Representation

For a representation of a set with n elements and a false positive probability of a *single* item check that is bound by ϵ, Broder and Mitzenmacher [8] prove the following lower bound (shown in Figure 2.1):

$$n \cdot \log_2 (1/\epsilon) \text{ bits} \tag{2.4}$$

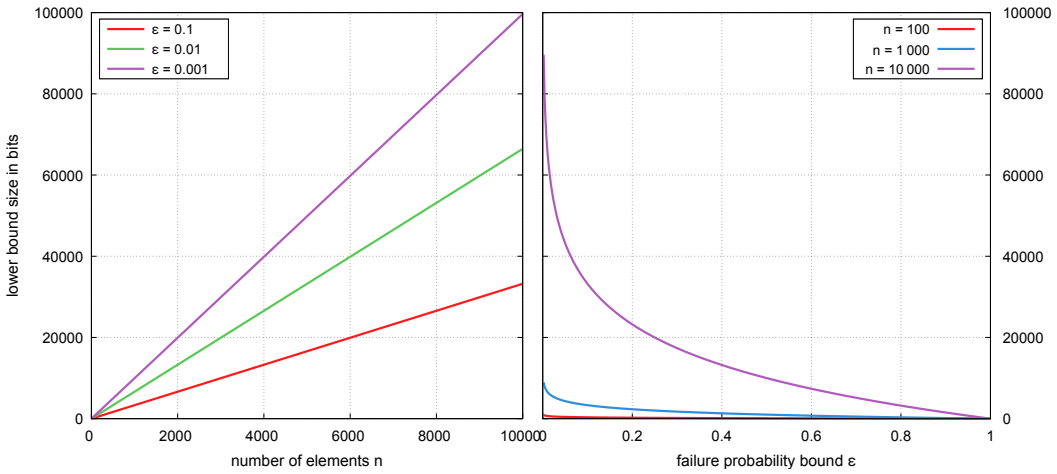


Figure 2.1: Theoretical lower bound for approximate set representation (eq. (2.4)) for variable n and selected ϵ (left) and vice versa (right).

Such an optimal approximate representation of a *static* set of n elements with a false positive probability of $1/2^j$ can, for example, be achieved by using a perfect hash table that stores n hashes of j bits each.

Although the argumentation of [8] only covers false positives, it can be easily extended to include false negatives, too. In this case, fewer items are accepted by a set representation but the resulting lower bound of eq. (2.4) still holds as long as the probability of *any* failure is bound by ϵ.

Lower Bound for Approximate (Versioned) Set Reconciliation

When approximate set representation algorithms are used for approximate (versioned) set reconciliation protocols, their size should ideally not depend on the original sets $S_\mathtt{A}$ and $S_\mathtt{B}$ but on the differences $\Delta_\mathtt{A}$ and $\Delta_\mathtt{B}$ instead.

A lower bound for the approximate set reconciliation problem using set representation algorithms is clearly established if `B` sends $\Delta_\mathtt{B}$ to `A` and `A` sends $\Delta_\mathtt{A}$ to `B`. Afterwards, `A` and `B` each determine the items to send by querying items from $S_\mathtt{A} \setminus_v \Delta_\mathtt{A}$ in $\Delta_\mathtt{B}$ and $S_\mathtt{B} \setminus_v \Delta_\mathtt{B}$ in $\Delta_\mathtt{A}$, respectively. Since only item keys need to be sent in $\Delta_\mathtt{A}$ and $\Delta_\mathtt{B}$, the following argumentation applies to both the versioned and the non-versioned approximate set reconciliation.

If both set representations have a failure probability ϵ' of a single query, the overall failure probability $P(\Delta'_\mathtt{A} \neq \Delta_\mathtt{A} \vee \Delta'_\mathtt{B} \neq \Delta_\mathtt{B})$ of the approximate set

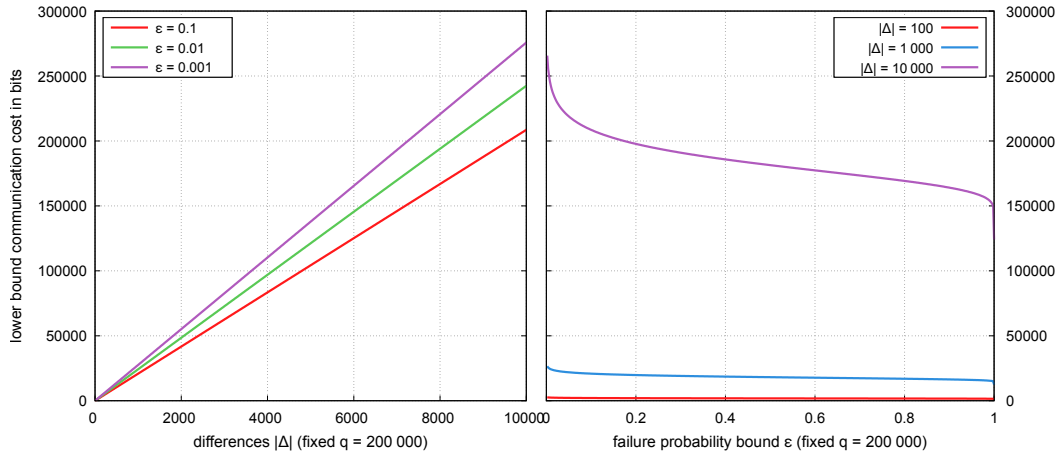


Figure 2.2: Theoretical lower bound for approximate set reconciliation (eq. (2.5)) for $200\,000 = q =: |S_\mathtt{A} \setminus_v \Delta_\mathtt{A}| + |S_\mathtt{B} \setminus_v \Delta_\mathtt{B}|$.

reconciliation can be calculated using the success probability for $|S_\mathtt{A} \setminus_v \Delta_\mathtt{A}|$ and $|S_\mathtt{B} \setminus_v \Delta_\mathtt{B}|$ query operations, respectively, assuming independence:

$$\begin{aligned} P(\Delta'_\mathtt{A} \neq \Delta_\mathtt{A} \vee \Delta'_\mathtt{B} \neq \Delta_\mathtt{B}) &= 1 - \overbrace{(1-\epsilon')^{|S_\mathtt{A} \setminus_v \Delta_\mathtt{A}|}}^{\text{all queries at A successful}} \cdot \overbrace{(1-\epsilon')^{|S_\mathtt{B} \setminus_v \Delta_\mathtt{B}|}}^{\text{all queries at B successful}} \\ &= 1 - (1-\epsilon')^{|S_\mathtt{A} \setminus_v \Delta_\mathtt{A}| + |S_\mathtt{B} \setminus_v \Delta_\mathtt{B}|} =: 1 - (1-\epsilon')^q \end{aligned}$$

Therefore, in order to fulfil $P(\Delta'_\mathtt{A} \neq \Delta_\mathtt{A} \vee \Delta'_\mathtt{B} \neq \Delta_\mathtt{B}) \leq \epsilon$, we have to use $\epsilon' \leq 1 - \sqrt[q]{1-\epsilon}$. We establish the following lower bound in bits from eq. (2.4):

$$|\Delta_\mathtt{A}| \cdot \log_2 \frac{1}{\epsilon'} + |\Delta_\mathtt{B}| \cdot \log_2 \frac{1}{\epsilon'} = |\Delta| \cdot \log_2 \left(\frac{1}{1 - \sqrt[q]{1-\epsilon}} \right) \tag{2.5}$$

This function is shown in Figure 2.2 for different $|\Delta|$ and fixed ϵ (left), and different ϵ with fixed $|\Delta|$ (right). Furthermore, with the help of the following two Lemmas, we will show that this is in $\mathcal{O}\left(|\Delta| \cdot \log\left({}^{n}/_{\epsilon}\right)\right)$ (Theorem 2.1.3).

Lemma 2.1.1. $\ln(1-\epsilon) \leq k \cdot \ln\left(1 - \frac{\epsilon}{k}\right)$ *for* $\epsilon \in \mathbb{R}$, $0 < \epsilon < 1, 1 \leq k \in \mathbb{N}$

Proof. For $k = 1$, this inequality follows immediately. For all other values of k, we use the following inequality from [79] for $-1 < x \leq 0$:

$$\frac{x}{2} \cdot \frac{2+x}{1+x} \leq \ln(1+x) \leq \frac{2x}{2+x} \tag{2.6}$$

By inserting $x_1 := -\epsilon$ and $x_2 := -{}^{\epsilon}/_{k}$, we show the following equation and thus prove this lemma.

$$\overbrace{\ln(1-\epsilon) \leq \frac{-2\epsilon}{2-\epsilon}}^{\text{eq. (2.6) for } x=-\epsilon} \overset{?}{\leq} \overbrace{k \cdot \left(\frac{-\frac{\epsilon}{k}}{2} \cdot \frac{2-\frac{\epsilon}{k}}{1-\frac{\epsilon}{k}} \right) \leq k \cdot \ln\left(1-\frac{\epsilon}{k}\right)}^{\text{eq. (2.6) for } x=-\epsilon/k}$$

$$\begin{aligned}
\Leftrightarrow \quad & -\frac{2\epsilon}{2-\epsilon} \leq -\frac{\epsilon}{2} \cdot \frac{2-\frac{\epsilon}{k}}{1-\frac{\epsilon}{k}} \\
\Leftrightarrow \quad & -4 \cdot \left(1-\frac{\epsilon}{k}\right) \leq -\left(2-\frac{\epsilon}{k}\right) \cdot (2-\epsilon) \\
\Leftrightarrow \quad & -4 + \frac{4\epsilon}{k} \leq -4 + 2\epsilon + \frac{2\epsilon}{k} - \frac{\epsilon^2}{k} \\
\Leftrightarrow \quad & 0 \leq 2 - \frac{2}{k} - \frac{\epsilon}{k} \\
\Leftarrow \quad & 0 \leq 2k - 3 \leq 2k - 2 - \epsilon \\
\Leftarrow \quad & 3/2 \leq k
\end{aligned}$$

□

Lemma 2.1.2. $\log_2\left(\frac{1}{1-\sqrt[k]{1-\epsilon}}\right) \leq \log_2 \frac{k}{\epsilon}$ *for* $\epsilon \in \mathbb{R}$, $0 < \epsilon < 1, 1 \leq k \in \mathbb{N}$

Proof. The following transformations will prove this Lemma:

$$\begin{aligned}
& \log_2\left(\frac{1}{1-\sqrt[k]{1-\epsilon}}\right) \overset{?}{\leq} \log_2 \frac{k}{\epsilon} \\
\Leftrightarrow \quad & \frac{1}{1-\sqrt[k]{1-\epsilon}} \leq \frac{k}{\epsilon} \\
\Leftrightarrow \quad & \sqrt[k]{1-\epsilon} \leq 1 - \frac{\epsilon}{k} \\
\Leftrightarrow \quad & 1-\epsilon \leq \left(1-\frac{\epsilon}{k}\right)^k \\
\Leftrightarrow \quad & \ln(1-\epsilon) \leq k \cdot \ln\left(1-\frac{\epsilon}{k}\right) \qquad \text{(see Lemma 2.1.1)}
\end{aligned}$$

□

Theorem 2.1.3. *The lower bound of the communication costs of the approximate (versioned) set reconciliation problem with* $n \coloneqq |S_A \cup Mis_A|$ *from above is in* $\mathcal{O}\left(|\Delta| \cdot \log\left({}^{n}/_{\epsilon}\right)\right)_{n\to\infty,\ \epsilon\to 0}$ *bits.*

Proof. This follows immediately from eq. (2.5), Lemma 2.1.2, and with $q \coloneqq |S_A \setminus_v \Delta_A| + |S_B \setminus_v \Delta_B| \leq |S_A| + |S_B| \leq 2n$:

$$|\Delta| \cdot \log_2\left(\frac{1}{1-\sqrt[q]{1-\epsilon}}\right) \leq |\Delta| \cdot \log_2 \frac{q}{\epsilon} \in \underset{(\text{for } n\to\infty,\ \epsilon\to 0)}{\mathcal{O}\left(|\Delta| \cdot \log \frac{n}{\epsilon}\right)}$$

□

2.1.2 Error Correction Codes vs. Set Reconciliation

As the following examples show, under certain conditions, set reconciliation algorithms can be used for error correcting codes and vice versa—these two fields are thus closely related. Intuitively, in the set reconciliation problem, if node `A` is given error-correction data it may re-create node `B`'s set using this data and its own (differing) set. Similarly, data sent during the reconciliation of two copies of a file (represented as an ordered set of blocks) can be seen as an error-correcting code for the file.

Despite their similarities, however, Karpovsky et al. [41] point out a subtle but important difference between the definition of error correction and (set) reconciliation. In the error correction problem, it is assumed that one host has an uncorrupted version and the other host has a corrupted version of the data and both nodes know which one has which. In the reconciliation problem, any node may have missing data and the transfer direction of item updates is not only unknown but may also be different per item, not per node.

Using Set Reconciliation for Error Correction Codes

Mitzenmacher and Varghese [59] use invertible Bloom lookup tables (`IBLT`) introduced by Goodrich and Mitzenmacher [34] to create a fast error correction code for large data under the assumption that errors are mutation errors and the sequence order remains intact. With their approach, they effectively provide a general reduction from error correcting codes to set reconciliation. When node `B` sends node `A` a message, additional set reconciliation information is sent. Node `A` reconciles the received message with the `IBLT`, extracts all erroneous entries, and reconstructs the original message. Although this approach is not space-optimal and the expected overhead is roughly a factor of 6 over the optimal amount—which is still small compared to the items' data sent later—, its advantage is the simplicity and encoding/decoding speed. This also makes it useful for smaller messages when computational efficiency is of importance.

Using Error Correction Codes for Set Reconciliation

Minsky et al. [55] describe the general idea behind the use of different error correcting codes in literature in order to solve the set reconciliation problem. Basically, a set $S \subset \mathcal{U}$ (with a common universal set $\mathcal{U}$) can be represented by a length $|\mathcal{U}|$ bitstring that has a 1 at location i if and only if the i-th element of $\mathcal{U}$ (under some arbitrary ordering) is present in S. They reference to literature where this was exploited with different error correction codes [64, 1, 41] and point out that, unfortunately, the computational complexity for traditional error correcting codes depends exponentially on the representation sizes of an element of S. They also show that their set reconciliation protocol based on characteristic polynomials can essentially be seen as a variation of a Reed-Solomon code which transmits the redundancy information.

2.1.3 Straggler Identification vs. Set Reconciliation

Eppstein and Goodrich [25] develop invertible Bloom filters to solve the straggler identification problem, i.e. given a universe $U = \{x_1, x_2, \ldots, x_n\}$ of unique positive identifiers representable with $\mathcal{O}(\log n)$ bits and an upper bound $d < n$; design an indexing structure for a set $S \subseteq U$ using $o(n)$ bits which efficiently supports (a) adding new elements, (b) removing existing ones, and (c) listing all elements of S if $|S| \leq d$. Only a few set reconciliation methods support listing items and are thus suitable. On the other hand, a solution to the straggler identification problem that tolerates false deletions, i.e. removing items from the index which are not in S, can be used for set reconciliation. It can be applied just like the invertible Bloom filters presented here: one node inserts its items into the index and sends it to the other node which removes its items and lists the remaining ones.

Further algorithms from the straggler identification domain that may be used for set reconciliation include those of Ganguly and Majumder [30] and Cormode and Muthukrishnan [19].

2.1.4 Open Problems

Mitzenmacher and Varghese [60] consider the general field of object reconciliation and its relation to set difference, coding, and rumour spreading. For the 2-node set reconciliation, they present upper bounds for computation, communication and time complexity based on existing algorithms, i.e. characteristic polynomials [55] and invertible Bloom filters [26]. They describe interesting extensions where these algorithms can be used or generalised, e.g. for sequence reconciliation and coding schemes, and present open problems in these fields.

While 2-node graphs seem well studied, extensions for reconciling multiple nodes are not. It is, for example, an open problem whether there is an object reconciliation method for general graphs that is more efficient in computation, communication, and/or time than a push/pull approach with 2-party set reconciliations. Further open problems include other types of objects to reconcile, e.g. sets of sets, "approximate" reconciliation where replica nodes must not contain every object but only a single representative of an object class, or dynamic reconciliation where objects of the network graph change over time.

2.2 Set Reconciliation Applications

The set reconciliation problem typically arises in systems where poor or unpredictable network, storage, or node availability is tackled by temporarily sacrificing consistency. In these systems, efficient algorithms are particularly useful to reduce the maintenance traffic's bandwidth and use it for the system's main purpose. Set reconciliation can, however, also be useful in several other fields. This section gives a brief overview of some of the interesting applications.

2.2.1 Set Representation

Since the algorithms presented below and in related work are optimised and adapted variants of common set representation algorithms—an integral part of any set reconciliation algorithm—, the applicability of the accuracy model introduced in Section 2.5 and the algorithms themselves grows beyond set reconciliation alone. They can generally be applied to any application that requires an efficient representation of a set of items.

Typical uses outside the context of set reconciliation have been provided by Broder and Mitzenmacher [8] and Tarkoma et al. [78] who present several applications of Bloom filters as a form of set representation. These applications are, however, not necessarily specific to Bloom filters. They range from utilisations as dictionaries improving memory efficiency or lookup speed over network traffic optimisations in databases, web caches, or peer-to-peer lookups up to resource and packet routing as well as system monitoring and flow detection.

2.2.2 Network Applications

Gossip Protocols

The problem of reconciling data stored at different nodes arises naturally in the context of gossip protocols [53] where nodes periodically exchange data items with each other. While gossiping aggregates may be useful to obtain estimates of global properties with high confidence and low overhead, e.g. for load balancing [44], their content is usually small and can be sent directly. However, when larger sets of data are sent, e.g. when distributing USENET postings [39], bibliographic databases [33], or a a list of hosts participating in a system [33, 37, 85], set reconciliation may be useful to reduce the message sizes [53]. Similar results are needed when bandwidth is scarce, e.g. when disseminating data, configuration parameters, or new programming instructions in sensor networks [49].

Content Delivery

Byers et al. [11] apply set reconciliation using Bloom filters to determine those parts of a distributed, erasure-encoded (large) file that are missing on one peer and offered by another. This way, peers can collaborate during downloads by receiving packets from the source as well as other peers to increase the overall download rate.

Set Intersection

Similarly to finding the set union via the set difference, set reconciliation can also be used to find the set intersection of two nodes. Reynolds and Vahdat [70], for example, use this approach with Bloom filters to perform keyword

searches in their P2P system. They need the set intersection to answer queries for documents with two keywords if different peers are responsible for them.

De-duplication

By determining the set intersection, we can also use set reconciliation techniques for de-duplication tasks such as the one described by Li et al. [48]. They approximate the similarity of two sets, i.e. the size of their intersection, with Bloom filter techniques from [8] in order to estimate the usefulness of running a full de-duplication protocol between the two sets. Bloom filters are also often used in order to detect duplicates in local scenarios [89, 71] but the same techniques work in a distributed system, too, and are not limited to Bloom filters either.

2.2.3 Cryptocurrencies

Cryptocurrencies based on blockchain technology such as Bitcoin [62] and Ethereum [87] need to publish newly created blocks with successfully added transactions in their network. Driven by (a) the immanent need for fast block updates, (b) an increasing number of transactions per block due to techniques like Segregated Witness [50] or increasing block sizes, and (c) given the fact that each transaction may be present in multiple miners' memory pools, optimisations using set reconciliation techniques were discussed. The focus was on reducing the size of the block updates and thus the transmission time.

Similarly to our *SHash* algorithm (ref. Chapter 6), Compact Blocks [18] use shortened transaction hashes (48 bits instead of 256) in their block updates. The receiving node may then be able to reconstruct the block from its memory pool or may need a second phase asking for the remaining transactions. Overall, this reconciliation requires up to 3 round trips.

In addition to using only 64-bit transaction hashes, Bitcoin Xtreme Thinblocks [84] use Bloom filters of the transactions in the memory pool of the node lacking one block behind. The receiving node then responds with the block's header, all hashes of transactions in the block and any transaction in the block but not in the Bloom filter. The block can then be reconstructed with high probability or, alternatively, a second phase will request all missing transactions by their hashes. This typically shrinks block update messages to anything between $1/40$ and $1/100$ of a regular block update message. Compared to Compact Blocks, however, message sizes may increase, but Xtreme Thinblocks, on the other hand, only require a maximum of two round trips.

Compared to the previous two approaches, Ozisik et al. [65, 66] devise *Graphene* which is able to reduce block update message sizes even further by keeping the maximum of two round trips. Similarly, they use Graphene to optimise status report messages which they propose to quickly identify malicious miners and eclipse attacks and to quantify the risk of a double-spend attack within minutes instead of hours. The sender of a new block creates

both an invertible Bloom lookup table (`IBLT`) and an (ordinary) Bloom filter (`BF`) from the transaction IDs in the block. On the receiver, `BF` is used to filter out block-unrelated transactions from its memory pool and create an own `IBLT` from the remaining ones. Then, the set reconciliation algorithm by Eppstein et al. [26] (also ref. Section 2.3.6) is used to identify any transaction IDs missing on the receiver which it requests in a subsequent phase. Compared to Xtreme Thinblocks, the Bloom filters only contain IDs of transactions in the block and can thus be smaller, also the block's (shortened) transaction IDs are not sent. Compared to Compact Blocks, Graphene encodes block updates with 75 % fewer bytes. However, without an implicit assumption on the ordering of transactions inside a block, in the worst case, an additional cost of $n \cdot \log_2(n)$ bits is required to send the order along.

2.2.4 Miscellaneous Applications

Car-to-Car Communication

Yu and Bai [88] apply set reconciliation to message exchanges in vehicular P2P systems consisting of multiple cars connected via dedicated short range communication. P2P applications envisioned by the automotive industry include location-based service, music and video sharing as well as diagnostic and prognostic information collaboration. Coarse-granularity sketches first estimate each partner's contribution and allow a prioritisation of communication partners and tasks. Fine-granularity summaries with set reconciliation algorithms such as Bloom filters allow membership tests to avoid redundant transfers. This way, the throughput may be improved by around 25 % which is particularly important in such a challenging scenario, i.e. short link durations of 20–40 s as well as frequent wireless link breakages.

Security in Body Sensors

Ali and Khan [2] use set reconciliation with characteristic polynomials [55] to implement a common key agreement protocol among wireless body area sensors and a personal server for encryption. They use electrocardiogram (EKG) feature sets and let the personal server—a powerful sensor node—broadcast some parts of this set's characteristic polynomial (as a set of evaluation points and values at these points). The receivers then determine the common set of these features and use it to agree on a common key.

2.3 Typical Set Reconciliation Algorithms

The most common data structures used for approximate set reconciliation in research and applications are (a) Bloom filters [3] and their variants which were, for instance, deployed in Tribler [68] and OceanStore [46], and (b) Merkle trees [52] which were implemented by Dynamo [22], Cassandra [63], and DHash [14],

for example. We therefore dedicate separate chapters to these two algorithms (Chapters 7 and 8), present a detailed analysis of them, and show how to apply the accuracy model of Section 2.5. Individual related work regarding these two data structures is presented there (Sections 7.6 and 8.6, respectively) and includes some of the variants and alternatives proposed in literature. However, two Bloom filter variants which have been especially tailored for and have been analysed as part of set reconciliation algorithms, i.e. counting Bloom filters and invertible Bloom filters, are shown in this section further below.

Despite the popularity of Bloom filters and Merkle trees, *versioned* set reconciliation has not been explicitly described in literature. To the best of our knowledge, only Lin and Levis [49] explicitly consider versioned data in their set reconciliation algorithm for sensor networks. Their data structure, however, assumes a fully covered—and thus small—key space where the only thing that is used during the reconciliation is the version numbers with implicit assumptions on the according keys. It evades the problem of finding matching items and is not as generic as our definition of a versioned set reconciliation where the key space may be arbitrarily large. Alternatively, if we think of a (compound) item as a set on its own consisting of a key, a version, and some payload, we could try algorithms for the reconciliation of sets of sets such as the one proposed by Mitzenmacher and Morgan [57] using IBLTs of IBLTs[a]. However, as with other standard set reconciliation algorithms without explicit support for versions, it would try to update new items with old content since it merely identifies the existence of a difference, not the direction of the transfer needed to bring both nodes up to date.

Orthogonally to the two-party set reconciliations that we focus on in this work, proposals were made on adapting common set reconciliation algorithms to multi-party reconciliations where more than two parties want to identify their set difference collectively. Mitzenmacher and Pagh [58] extend IBLTs to r-ary reconciliations by changing the sum fields, e.g. `keySum` and `keyHashSum`, to work on a finite field $\mathbb{F}_r$ so that only if the same entry is added r times it will cancel out. Similarly, due to the different combinations of nodes having or not having an item, the lookup and list operations are adapted for recovering items which were added t times, for $2 \leq t < r$, i.e. by checking for matching values in the `sum` and `hashSum` fields based on the `count`[b]. Similarly, Boral and Mitzenmacher [4] extend set reconciliation with characteristic polynomials (ref. Section 2.3.3) to r parties. Both extensions lead to reduced communications costs in terms of message size and number of messages compared to $r - 1$ two-party set reconciliations.

While set reconciliation can be solved in many different ways, there are a few algorithms considered optimal under the information-theoretic communication

[a] For an extensive description of invertible Bloom lookup tables (IBLTs), see Section 2.3.6.

[b] For simplicity of the presentation, here, we assume computations over a field $\mathbb{F}_r$ while, actually, Mitzenmacher and Pagh [58] perform computations over $\mathbb{F}_p$ with $p \geq r$ and p prime and compensate for the missing $p - r$ parties by adding one of the IBLTs $p - r + 1$ times

cost complexity of eq. (2.5). We will describe these and other interesting set reconciliation algorithms in the following sections. Note, however, that the complexity class may hide (large) constant factors which may not make these algorithms optimal under a specific use case (see [40], for example). Also, practical considerations like required CPU power or implementation complexity were hindering the adoption of some of these in favour of less optimal but simpler solutions. We will go into detail in the individual sections.

2.3.1 Elementary Symmetric Polynomials

For the special case where $S_\mathtt{A} \subset S_\mathtt{B}$, Minsky and Trachtenberg [53] describe a near-optimal set reconciliation algorithm based on elementary symmetric polynomials which was later picked up by Eppstein and Goodrich [25] for the straggler identification problem. The i-th elementary symmetric polynomial $\sigma_i(x_1, x_2, \ldots, x_n)$ is the sum of all possible products of i terms of the input, i.e.:

$$\begin{aligned}
\sigma_0(x_1, x_2, \ldots, x_n) &\coloneqq 1 \\
\sigma_1(x_1, x_2, \ldots, x_n) &= x_1 + x_2 + \ldots + x_n \\
\sigma_2(x_1, x_2, \ldots, x_n) &= x_1x_2 + \ldots + x_1x_n + x_2x_3 + \ldots + x_2x_n + \ldots + x_{n-1}x_n \\
\sigma_3(x_1, x_2, \ldots, x_n) &= x_1x_2x_3 + x_1x_2x_4 + \ldots + x_{n-2}x_{n-1}x_n \\
&\ldots
\end{aligned}$$

Minsky and Trachtenberg use the following convolution property to create Algorithm 1 for $1 \leq |\Delta| \leq {}^{|S_\mathtt{A}|}/_2$ with three messages transferring $\mathcal{O}(|\Delta| \cdot b)$ bits in total with b being an element's size in bits:

$$\sigma_k(S_\mathtt{B}) = \sum_{0 \leq i \leq k} \sigma_i(S_\mathtt{A}) \cdot \sigma_{k-i}(\Delta) \tag{2.7}$$

At first, $|\Delta|$ is determined, then for $1 \leq i \leq |\Delta|$, node A reads $\sigma_i(S_\mathtt{B})$ and computes $\sigma_i(S_\mathtt{A})$ and $\sigma_i(\Delta)$ iteratively. Finally, the following equation is solved providing all elements $x \in \Delta$:

$$\prod_{x \in \Delta}(z - x) = \sum_{i=0}^{|\Delta|}(-1)^i \cdot \sigma_i(\Delta) \cdot z^{|\Delta|-i} = 0 \tag{2.8}$$

To minimise communication complexity, all these operations are performed over a finite field $\mathbb{F}_p$ for some prime $p > 2^b$. This leads to at most $|\Delta| \cdot b + (b + \phi)$ bits during communication with $\phi \in \mathbb{R}, 1 \leq \phi \leq |\Delta|$ depending on p which is clearly in $\mathcal{O}(|\Delta| \cdot b)$. There are, however, two bottlenecks with Algorithm 1: the computation of the elementary symmetric polynomials and finding the roots at the end. The former can be amortised during element additions but the latter factorisation still has an expected time of $\mathcal{O}(|\Delta|^{1.83} \cdot \log p)$ [53].

Alternatively, Minsky and Trachtenberg present a divide-and-conquer algo-

Algorithm 1 Subset reconciliation with elementary symmetric polynomials [53]

function ESPolySyncA ▷ executed on node A
  Receive $|S_B|$ ▷ from node B
  $|\Delta| \leftarrow |S_B| - |S_A|$
  Send($|\Delta|$) ▷ to node A
  Receive $\sigma_i(S_B)\ \forall 1 \leq i \leq |\Delta|$ ▷ one message from B
  Calculate $\sigma_i(S_A)\ \forall 1 \leq i \leq |\Delta|$
  Calculate $\sigma_i(\Delta)\ \forall 1 \leq i \leq |\Delta|$ ▷ eq. (2.7)
  Determine all $x \in \Delta$ ▷ by solving eq. (2.8)
  return Δ
end function

rithm only using Algorithm 1 for $|\Delta| = 1$ in the recursion step which provides lower computational complexity but higher communication complexity.

2.3.2 Hash-based Divide-and-Conquer

For the more general case of arbitrary differences between the sets S_A and S_B, Minsky and Trachtenberg [53] adapt their divide-and-conquer algorithm omitting the elementary symmetric polynomials and using a hash function to determine whether sets are equal (Algorithm 2).

Algorithm 2 Divide-and-conquer set reconciliation from [53]

function DivideAndConquerA(S_A,b) ▷ on node A, set S_A, b-bit elements
  if $|S_A| = 0$ **then**
    Receive missing elements from B
  else if $|S_B| = 0$ **then**
    Send S_A to B ▷ missing elements on node B
  else if $hash(S_A) = hash(S_B)$ **then**
    return
  else
    $S_{A,1} \leftarrow \{x \in S_A : x[b] = 0\}$ ▷ b'th bit is 0
    $S_{A,2} \leftarrow \{x \in S_A : x[b] = 1\}$ ▷ b'th bit is 1
    DivideAndConquerA($S_{A,1}, b-1$)
    DivideAndConquerA($S_{A,2}, b-1$)
  end if
end function

Although Minsky and Trachtenberg do not argue about the failure probability of their algorithm, with an appropriately large hash size h, this may be negligible. During this reconciliation, $|\Delta| \cdot b \cdot (4h+1)$ bits are transferred which is clearly in the optimal communication complexity class $\mathcal{O}(|\Delta| \cdot b)$ for exact algorithms but may be too high for practical purposes. The trivial exact algorithm, for example, transfers $|S_B| \cdot b$ bits and thus Algorithm 2 is only more effective if $h < 1/4 \cdot (|S_B|/|\Delta| - 1)$. For $|S_B| = 100\,000$ and $|\Delta| = 2\,000$ (2 % differences),

h must be smaller than or equal to 12 bits for which the failure probability is not negligible. Algorithm 2 is thus only suitable for very low $|\Delta|$ such as $|\Delta| = 200 \Rightarrow h \leq 124$ or $|\Delta| = 20 \Rightarrow h \leq 1249$ in this case.

Further improvements may be gained by splitting the actual sets based on their median rather than the address space. This variation requires at most $2 \cdot (|\Delta| \cdot \log_2 |S_\mathtt{B}| + 1) \cdot (b + h + 2) + |\Delta| \cdot b$ bits to be sent when node `B` is the *active* partner, i.e. the one from which the median is taken. Please refer to [53] for further details.

2.3.3 Characteristic Polynomials with an Upper Bound on $|\Delta|$

Minsky et al. [55] present an almost optimal set reconciliation algorithm based on characteristic polynomials. They represent a set $S = \{x_1, x_2, \ldots, x_n\}$ with the following characteristic polynomial $\chi_S(Z)$:

$$\chi_S(Z) \coloneqq (Z - x_1) \cdot (Z - x_2) \cdot \ldots \cdot (Z - x_n)$$

Here, the set's elements x_i, i.e. b-bit strings, are mapped to some field $\mathbb{F}_q$ for $q \geq 2^b$. However, since this representation contains all the original information, it is not cheaper to transmit χ_S instead of transmitting the set itself. Minsky et al. therefore use the ratio between two sets' characteristic polynomials:

$$\frac{\chi_{S_\mathtt{A}}(Z)}{\chi_{S_\mathtt{B}}(Z)} = \frac{\chi_{S_\mathtt{A} \cap S_\mathtt{B}}(Z) \cdot \chi_{S_\mathtt{A} \setminus S_\mathtt{B}}(Z)}{\chi_{S_\mathtt{A} \cap S_\mathtt{B}}(Z) \cdot \chi_{S_\mathtt{B} \setminus S_\mathtt{A}}(Z)} = \frac{\chi_{S_\mathtt{A} \setminus S_\mathtt{B}}(Z)}{\chi_{S_\mathtt{B} \setminus S_\mathtt{A}}(Z)} \tag{2.9}$$

To allow both nodes `A` and `B` to compute this fraction efficiently, the polynomials are evaluated at a set of evaluation points which are used to interpolate the resulting rational function. The degree of this function—and thus the number of evaluation points required for interpolation—depends on the differences between the nodes, i.e. $S_\mathtt{A} \setminus S_\mathtt{B}$ and $S_\mathtt{B} \setminus S_\mathtt{A}$. Algorithm 3 shows the set reconciliation protocol for a known upper bound $|\Delta|_{up} \geq |\Delta| = |S_\mathtt{A} \setminus S_\mathtt{B}| + |S_\mathtt{B} \setminus S_\mathtt{A}|$.

The existence and uniqueness (up to equivalence) of the interpolated and recovered rational function ${}^{\chi_{S_\mathtt{A} \setminus S_\mathtt{B}}(Z)}\!/_{\chi_{S_\mathtt{B} \setminus S_\mathtt{A}}(Z)}$ under $|\Delta|_{up}$ evaluation points is proven in [55] and therefore both $S_\mathtt{A} \setminus S_\mathtt{B}$ and $S_\mathtt{B} \setminus S_\mathtt{A}$ can be recovered. For practical purposes, however, q is set so that $q \geq 2^b + |\Delta|_{up}$ in order to prevent complications with 0-valued characteristic polynomials of evaluation points from $S_\mathtt{A}$ or $S_\mathtt{B}$. This costs at most one extra bit per item.

Minsky et al. [55] determine the communication complexity of Algorithm 3 as $(|\Delta|_{up} + 1) \cdot (b + 1) - 1$ bits which is in the optimal $\mathcal{O}(|\Delta| \cdot b)$ bits for exact algorithms if $|\Delta|_{up}$ is close to $|\Delta|$ (and known a priori). The computational complexity is composed of $\mathcal{O}(|S_\mathtt{A}| \cdot |\Delta|_{up})$ evaluations of the characteristic polynomial at node `A` and the $\mathcal{O}(|\Delta|_{up}^3)$ cost for each of the interpolation and root finding steps. This—as well as the complex mathematics behind Algorithm 3—may make it unsuitable for some applications.

Algorithm 3 Set reconciliation with characteristic polynomials [55]

function CpiA ▷ on node A with pre-defined set of $|\Delta|_{up}$ evaluation points
  Receive values $\chi_{S_B}(Z)$ of $|\Delta|_{up}$ evaluation points Z ▷ from node B
  Compute values $\chi_{S_A}(Z)/\chi_{S_B}(Z)$ at these points ▷ eq. (2.9)
  Interpolate and recover the coefficients of $\chi_{S_A \setminus S_B}(Z)/\chi_{S_B \setminus S_A}(Z)$
  Factor $\chi_{S_A \setminus S_B}(Z)$ and $\chi_{S_B \setminus S_A}(Z)$
  return $S_A \setminus S_B$ and $S_B \setminus S_A$
end function

2.3.4 Characteristic Polynomials with Unknown $|\Delta|$

In cases where an upper bound $|\Delta|_{up} \geq |\Delta|$ is not known, Minsky et al. [55] extend Algorithm 3 by running it with increasing values of $|\Delta|_{up}$ until a sufficiently large value is found. Determining whether a value of $|\Delta|_{up}$ is sufficient, however, requires a probabilistic check in order to be efficient. Here, the equality of the recovered rational functions is tested by comparing them at (additional) random evaluation points. In this case, the failure probability of two different such functions agreeing on the value of a randomly selected point is bound by $\rho = {}^{|\Delta|}/_{|E|}$ where E is the subset of $\mathbb{F}_q$ from which the evaluation points are chosen.

In order to minimise communication complexity in terms of bits, the values of $\chi_{S_B}(Z)$ can be received individually. A recomputes the interpolated rational function $g(Z)$ every time a new value is not confirmed by the last function. $g(Z)$ is accepted as equal to $\chi_{S_A \setminus S_B}(Z)/\chi_{S_B \setminus S_A}(Z)$ when k evaluation points in a row are confirmed. To achieve a failure probability of ϵ, no more than

$$k = \left\lceil \log_\rho \frac{\epsilon}{|\Delta|} \right\rceil = \left\lceil \frac{\log_2 \frac{\epsilon}{|\Delta|}}{\log_2 \frac{|\Delta|}{|E|}} \right\rceil \tag{2.10}$$

extra evaluation points are needed. With this k and by using a pseudo-random number generator and sending its seed instead of sending all random evaluation points individually, the communication complexity is bound by $(b+2)\cdot(|\Delta|+k)+b$ bits in a total of $|\Delta|+k$ rounds. The computational complexity, however, further increases to $\mathcal{O}(|\Delta|^4)$ due to the repeated interpolation step.

Minsky et al. also describe a variant with a reduced number of $\lceil \log_c(|\Delta| + k)\rceil$ communication rounds by sending c extra evaluation points per round. This variant increases the communication complexity roughly by factor c to $(b+1)\cdot c\cdot(|\Delta|+k)+b+\lceil \log_c(|\Delta|+k)\rceil$. On the other hand, for fixed c, the computational complexity is reduced to $\mathcal{O}((|\Delta|+k)^3)$.

Minsky and Trachtenberg [54] also propose another variant of their algorithm which reduces the computational complexity by applying a divide-and-conquer approach as above. They use a *fixed* recovery bound $|\Delta|_{up}$ and recursively execute their set reconciliation algorithm on p partitions until the reconciliation is successful. Higher computational costs for large $|\Delta|_{up}$ are thus avoided

at the cost of having a higher (worst-case) communication complexity of $\Theta(|\Delta| \cdot b^2 \cdot {}^{p}/_{\log p})$ in at most $1 + p \cdot \lceil b \cdot \log_p 2 \rceil \cdot {}^{|\Delta|}/_{|\Delta|_{up}}$ message rounds. For fixed k, the computational complexity is reduced to $\Theta(|\Delta| \cdot |\Delta|^2_{up} \cdot b^2 \cdot {}^{p}/_{\log p})$.

2.3.5 Counting Bloom Filters

Counting Bloom filters [27] are an extension to (standard) Bloom filters (ref. Chapter 7) by having a field of m small integer counters instead of bits. Additionally to the support for item insertions, this also allows deletions and thus a lightweight maintenance during database changes. Also, counting Bloom filters can be subtracted from one another which enables Guo and Li [36] to present a nearly optimal approximate set reconciliation protocol where m *mostly* depends on the set difference alone (see the comments below).

Data Structure

In a counting Bloom filter `CBF`, initially, all counters are 0 and items are added by incrementing k counters by 1 using k independent and uniformly distributed hash functions $g_{i \in \{0,1,\ldots,k-1\}}$ (ref. Figure 2.3). Similarly, membership queries for an item x check that *all* the counters at $g_i(x)$ are non-zero. If any of them is 0 then $x \notin \texttt{CBF}$, otherwise $x \in \texttt{CBF}$ with a false-positive probability FP—for a *single* membership query. Since this is the same behaviour as the standard Bloom filter, all of its theory and mathematical analysis applies to counting Bloom filters, too (see Section 7.2 for more details).

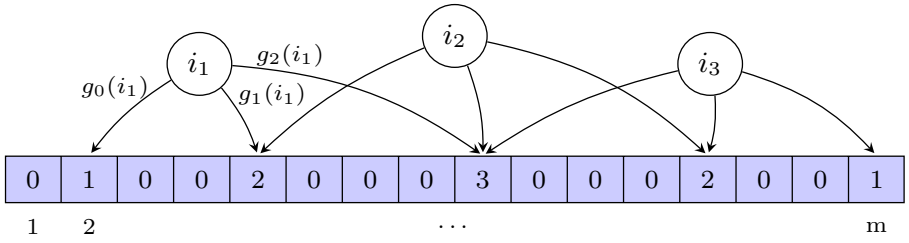


Figure 2.3: *Counting Bloom filter* example for items i_1, i_2, and i_3 with $k = 3$.

Unlike standard Bloom filters, items *already encoded* in `CBF` can be removed by decrementing their k positions by 1. If, however, items are removed which are not in `CBF`, counters may decrement to 0 and thus introduce false negatives. Analogously, for the set reconciliation problem, Guo and Li [36] define a *minus* operation of two equally-sized counting Bloom filters with the same hash functions by using the cell-wise subtraction of the counters.

The result of $\texttt{CBF}(S_\texttt{A}) - \texttt{CBF}(S_\texttt{B}) =: \texttt{CBF}(\Delta)'$ represents an approximation of the counting Bloom filter for Δ, i.e. $\texttt{CBF}(\Delta)$, which both nodes may use to check whether their items are in the difference. Since common items zero each other out in the subtraction, the $\texttt{CBF}(\Delta)'$ may equivalently be written as $\texttt{CBF}(S_\texttt{A} \setminus S_\texttt{B}) - \texttt{CBF}(S_\texttt{B} \setminus S_\texttt{A})$. This $\texttt{CBF}(\Delta)'$, however, may now have false

negatives during membership queries—additionally to false positives. These were created from zero-valued *outlier cells* in the subtraction which originated from different members of the counting Bloom filters rather than common ones, i.e. non-zero cells in both $\texttt{CBF}(S_\texttt{A} \setminus S_\texttt{B})$ and $\texttt{CBF}(S_\texttt{B} \setminus S_\texttt{A})$ which are zero in $\texttt{CBF}(\Delta)'$.

Set Reconciliation Protocol

During the set reconciliation protocol by Guo and Li [36] (Algorithm 4), nodes A and B create $\texttt{CBF}(S_\texttt{A})$ and $\texttt{CBF}(S_\texttt{B})$, respectively, and send it to the other node. A and B then determine $\texttt{CBF}(\Delta)' \coloneqq \texttt{CBF}(S_\texttt{A}) - \texttt{CBF}(S_\texttt{B})$ and check for all their members whether they are in this filter. For this protocol, Guo and Li analyse the expected number of the resulting false negatives as well as false positives separately (outlined below) and set an upper limit of 1 each to determine appropriate parameters for their counting Bloom filters. For simplicity in the formulae, let $d_1 \coloneqq |S_\texttt{A} \setminus S_\texttt{B}|$ and $d_2 \coloneqq |S_\texttt{B} \setminus S_\texttt{A}|$ be the number of unique elements in $S_\texttt{A}$ and $S_\texttt{B}$, respectively, and define $d \coloneqq d_1 + d_2 = |\Delta|$.

Algorithm 4 Counting Bloom filter set reconciliation [36]

function CBFSyncA	▷ on node A (analogously on B)
Estimate $d_1 \coloneqq \lvert S_\texttt{A} \setminus S_\texttt{B}\rvert$, $d_2 \coloneqq \lvert S_\texttt{B} \setminus S_\texttt{A}\rvert$	▷ via a separate protocol
Receive $\lvert S_\texttt{B}\rvert$	▷ from node B
Compute $\texttt{CBF}(S_\texttt{A})$	
Send($\texttt{CBF}(S_\texttt{A})$)	▷ to node B
Receive $\texttt{CBF}(S_\texttt{B})$	▷ from node B
Compute $\texttt{CBF}(\Delta)' \coloneqq \texttt{CBF}(S_\texttt{A}) - \texttt{CBF}(S_\texttt{B})$	
return $\{x \in S_\texttt{A} : x \in \texttt{CBF}(\Delta)'\}$	▷ $\approx S_\texttt{A} \setminus S_\texttt{B}$
end function	

False Negatives

False negatives can only occur from elements in $S_\texttt{A} \setminus S_\texttt{B}$ or $S_\texttt{A} \setminus S_\texttt{B}$ and for any outlier cell in $\texttt{CBF}(\Delta)'$ where both cells in $\texttt{CBF}(S_\texttt{A} \setminus S_\texttt{B})$ and $\texttt{CBF}(S_\texttt{B} \setminus S_\texttt{A})$ are non-zero. The number of non-zero cells in a (counting) Bloom filter of a set with $d_{x\in\{1,2\}}$ items can be derived from eq. (7.1) (page 91) and is given as:

$$nz(d_x) \coloneqq m \cdot \left(1 - \underbrace{(1 - 1/m)^{k \cdot d_x}}_{\substack{\text{probability that any} \\ \text{given bit remains 0}}}\right)$$

Guo and Li [36] further determine the expected number of outlier cells o in $\texttt{CBF}(\Delta)'$ with an upper bound at $d_1 = d_2 = d/2$:

$$o \coloneqq m \cdot \sum_{j=1}^{\min(d_1,d_2)\cdot k} \binom{k \cdot d_1}{j} \cdot \binom{k \cdot d_2}{j} \cdot \frac{(1 - 1/m)^{k \cdot d}}{(m-1)^{2j}}$$

With these, they give the total number of expected false negatives as the following symmetric function which is also maximised at $d_1 = d_2 = {}^{d}/_{2}$.

$$fn \coloneqq d_1 \cdot \left(1 - \left(1 - \underbrace{\frac{o}{nz(d_1)}}_{\text{non-zero cell of } S_\mathtt{A} \setminus S_\mathtt{B} \text{ becoming an outlier cell}}\right)^k\right) + d_2 \cdot \left(1 - \left(1 - \underbrace{\frac{o}{nz(d_2)}}_{\text{non-zero cell of } S_\mathtt{B} \setminus S_\mathtt{A} \text{ becoming an outlier cell}}\right)^k\right) \tag{2.11}$$

False Positives

Similarly, the probability FP of a false positive during a single membership query in $\mathtt{CBF}(\Delta)'$ can be given based on the probability that any given cell is zero [36]:

$$FP \coloneqq \left(1 - \underbrace{\left(1 - {}^{1}/_{m}\right)^{k \cdot d}}_{\text{cell in } \mathtt{CBF}(\Delta) \text{ is } 0} - \underbrace{\frac{o}{m}}_{\text{outlier cell}}\right)^k$$

An upper bound is established if no outlier cells exist and a lower bound exists by maximising the number of outlier cells, i.e. at $d_1 = d_2 = {}^{d}/_{2}$. With $|S_\mathtt{A}| - d_1 = |S_\mathtt{B}| - d_2$ common items, the total number of false positives is thus given as:

$$fp \coloneqq (|S_\mathtt{A}| - d_1) \cdot FP = (|S_\mathtt{B}| - d_2) \cdot FP \tag{2.12}$$

Estimating d_1 and d_2

In order to limit the number of false negatives and/or false positives by setting appropriate parameters m and k, the set reconciliation protocol outlined in Algorithm 4 requires estimates of $d_1 \coloneqq |S_\mathtt{A} \setminus S_\mathtt{B}|$ and $d_2 \coloneqq |S_\mathtt{B} \setminus S_\mathtt{A}|$. Guo and Li [36] use counting Bloom filters for this step, too, and compare the theoretical number of zero-valued cells $m \cdot (1 - {}^{1}/_{m})^{k \cdot d_x} + o$ with the actual number of zero-valued cells. Furthermore, they use the ratio r of the number of cells with positive values to the number of cells with negative values in $\mathtt{CBF}(\Delta)'$ to distinguish $d_2 = {}^{d}/_{(1+r)}$ from $d_1 = d - {}^{d}/_{(1+r)}$. They determine that $6d$ cells are sufficient for an accurate estimation but do not describe a full protocol without a known bound on d. A recursive protocol doubling an estimate of d and observing the convergence of the resulting estimates of d_1 and d_2 seems possible and requires $\mathcal{O}(\log d)$ rounds.

Comments & Summary

Although Guo and Li [36] claim that the communication cost of their protocol is in $\mathcal{O}(d)$ when limiting the number of false negatives *fn* and false positives *fp* to no more than one (eqs. (2.11) and (2.12)), they do not prove their claim and it is unclear whether it is actually in a higher complexity class. Additionally, while *fn* from eq. (2.11) only depends on m, k, d_1, and d_2, *fp* from eq. (2.12) also depends on the number of common objects $|S_\mathtt{A}| - d_1 = |S_\mathtt{B}| - d_2$ and thus

on the total number of items in the sets, i.e. n. This dependency is a result of performing membership queries in $\texttt{CBF}(\Delta)'$ and cannot be avoided with counting Bloom filters. Even if only the number of false negatives is limited by one, false positives would still create redundant item transfers and thus ultimately add to the communication cost. In this case they are added to the costs of the item transfers, not the set reconciliation itself.

The empirical evaluation by Guo and Li shows that a set reconciliation using counting Bloom filters may be more cost-efficient than using standard Bloom filters for low set differences. However, it seems that they set up both Bloom filter variants with a fixed $k = 3$ which may not be the optimal k for achieving the lowest costs for either Bloom filter (m and k both determine the failure probabilities and costs as shown in Figure 7.3, page 91). Finding this optimal k for counting Bloom filters, though, remains an open problem and it might as well be based on n, too. Thus further improvements for counting Bloom filters seem possible but the presented empirical comparison is unfair and needs to be re-evaluated due to the different influences of k in each algorithm.

Overall, a counting Bloom filter allows easier maintenance compared to a standard Bloom filter by offering the ability to remove existing items. Used alone, it also offers the same false positives probability. The use of subtracting one counting Bloom filter from another to get an approximation of the counting Bloom filter on the differences and embed this into a set reconciliation protocol is compelling. It does, however, also introduce false negatives and the communication cost for keeping a fixed accuracy still depends on both d and n, with an unknown complexity class.

2.3.6 Invertible Bloom Filters

Invertible Bloom filters [26, 34, 25] are an extension to counting Bloom filters and add additional fields for the sum of keys and values, e.g. using XOR. They can be subtracted cell-wise from one another just like counting Bloom filters but, in contrast, allow the extraction of the original keys or key-value pairs from the added fields. Invertible Bloom filters were first introduced by Eppstein and Goodrich [25] and later extensively studied as a data structure by Goodrich and Mitzenmacher [34] and as a means for set reconciliation by Eppstein et al. [26]. Below, we will focus on the pristine invertible Bloom filter `IBF`—as used in [26]—and only briefly discuss the extensions and challenges of using an invertible Bloom lookup table `IBLT` [34]. It will be an interesting field for future work to see whether the `IBLT` is a more efficient data structure for solving the approximate *versioned* set reconciliation problem.

Data Structure

Invertible Bloom filters (`IBF`) consist of an array of m cells, each with multiple fields and all initialised with 0: `count` stores the number of elements in this cell, `keySum` stores the `XOR` of all keys in this cell, and `keyHashSum` stores the

XOR of all hashes of these keys to reduce the likelihood of decoding errors. In addition, an invertible Bloom lookup table IBLT adds two more fields to store the XOR of all values in a cell (valueSum) and to store the XOR of all hashes of these values (valueHashSum). Figure 2.4 shows an example for both an IBF and an IBLT.

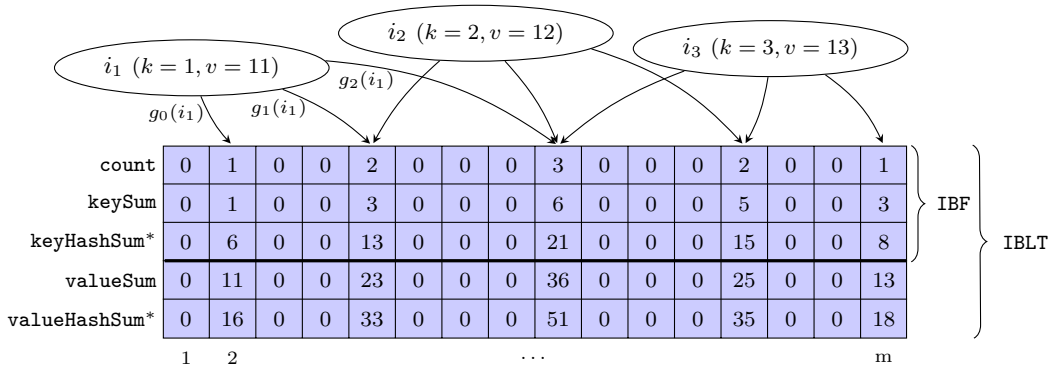


Figure 2.4: *Invertible Bloom filter (IBF) / lookup table (IBLT)* example for items i_1, i_2, and i_3 with $k = 3$ hash functions.
(*optional fields for extraneous key deletions and removal of non-inserted values for a key)

An item i is added by hashing it to k cells using k independent, uniformly distributed hash functions $g_{i \in \{0,1,\ldots,k-1\}}$ just like (counting) Bloom filters. The hash sum fields use two more—separate—hash functions G_k and G_v for the keys and values, respectively. Then, in each of those cells: increase count by one, add key i_k to keySum, add $G_k(i_k)$ to keyHashSum, add value i_v to valueSum, and add $G_v(i_v)$ to valueHashSum. Membership queries would work just like for counting Bloom filters by checking that *all* the counts in cells $g_i(x)$ are non-zero. If any of them is 0 then $x \notin$ IBF, otherwise $x \in$ IBF with a false-positive probability FP—for a *single* membership query (ref. Section 7.2). Analogously to counting Bloom filters, items may also be removed or whole IBFs may be subtracted from one another at the cost of introducing the possibility of false negatives during membership queries.

The true power of the invertible Bloom filter, however, comes from the possibility to *list contained items* which will be described in detail in the context of the set reconciliation protocol below. To briefly sum up: it is an iterative procedure that is based on retrieving keys (and values) from "pure" cells, i.e. cells which represent only a single item. In the presence of multiple insertions and/or false deletions[c], though, we cannot simply assume a pure cell if the count field is 1. Therefore, we also verify pure cells via the keyHashSum field (also valueHashSum for IBLT).

[c]Please note that with the set reconciliation protocol described here, false deletions, i.e. removing an item from a cell which did not contain that item, may occur but, assuming a node knows its own items and does not re-insert existing once, multiple insertions will not.

Set Reconciliation Protocol

Set reconciliation with invertible Bloom filters has been proposed by Eppstein et al. [26] and works in three steps: (1) encode, (2) subtract, and (3) decode. During *encode*, first assume that for a given $S_\mathtt{A}$ and $S_\mathtt{B}$, an oracle returns the size of the set difference $|\Delta|$ as Δ_{exp} (see *Estimating* $|\Delta|$, page 28). Then both nodes `A` and `B` create an `IBF` of size $m = \alpha \cdot \Delta_{exp}$ cells with $\alpha \geq 1$ using the same size and hash functions. On node `A` (analogously on node `B`), after receiving $\mathtt{IBF}_\mathtt{B}$, we create $\mathtt{IBF}_{\mathtt{A}-\mathtt{B}}$ by *subtracting* each cell's fields cell by cell. This eliminates common items from the resulting `IBF` so that the final *decode* phase may determine $S_\mathtt{B} \setminus S_\mathtt{A}$ and $S_\mathtt{A} \setminus S_\mathtt{B}$ on both nodes.

This *Decode* method is a specialized variant of the `ListEntries` method of an `IBF` which identifies both set differences based on the sign of the `count` field as a result of the *subtract* phase. As outlined above, it is based on iteratively decoding *pure* cells, where here a pure cell must satisfy two conditions: (a) `count` must be 1 or -1 (false deletion in a cell with `count` $= 0$), and (b) $G_k(\mathtt{keySum}) = \mathtt{keyHashSum}$. For each pure cell, we extract the key of the original item and add it to the appropriate difference set. Additionally, we remove it from all other cells it belongs to, allowing them to become pure cells, as well. Decoding continues until no further pure cells remain. At this point, if all cells have been cleared, then all items in the differences between $S_\mathtt{A}$ and $S_\mathtt{B}$ have been decoded. Otherwise some items remain encoded and there is not enough information to decode them. The full procedure is given by Algorithm 5.

Algorithm 5 IbfDecode [26] (non-optimised pseudo-code)

```
function IbfDecode(IBF_A, IBF_B, S_A \ S_B, S_B \ S_A)      ▷ adds to set differences
    IBF_{A−B} ← IBF_A − IBF_B                    ▷ create diff-IBF via cell-wise subtraction

    while ∃j ∈ {1, 2, ..., m} : IBF_{A−B}[j].IsPure do          ▷ any pure cell left?
        i_k ← IBF_{A−B}[j].keySum
        c ← IBF_{A−B}[j].count
        if c > 0 then                                          ▷ determine difference set
            (S_A \ S_B).Add(i_k)
        else
            (S_B \ S_A).Add(i_k)
        end if
        IBF_{A−B}.Delete(i_k, c)                          ▷ may create new pure cells
    end while

    if ∀j ∈ {1, 2, ..., m} : IBF_{A−B}[j].IsEmpty then   ▷ remaining IBF_{A−B} empty?
        return SUCCESS
    else
        return FAIL                                            ▷ decoding incomplete
    end if
end function
```

Please note that this algorithm only shows the approximate *non-versioned* set reconciliation. To make this algorithm solve the approximate *versioned* set reconciliation problem, you may add id-version pairs to both IBFs (at the cost of items with different versions being counted twice for Δ_{exp}) and use a second phase to determine which of the decoded items belong to $\Delta_{\mathtt{A}}$ and which to $\Delta_{\mathtt{B}}$, just like our algorithms described in the chapters below. Alternatively, an IBLT may be used with the version as the item's value. If we omit the `valueHashSum` field, during the *subtract* step as well as any other item delete operation during *decode*, we may add special handling for making the `count` field 0 with an i_k that matches the `keyHashSum` field. In that case, the resulting `valueSum` from $\mathtt{IBF}_{\mathtt{A-B}}$ would actually indicate which node has the newer version and we would reset this cell. Delete operations from other keys with differing versions, however, will falsify the information in the value field. We will leave the discussion and evaluation of this algorithm—including effects on failure probabilities or cost comparisons between these two variants and our variants below—up for future work.

Decoding errors

Eppstein et al. [26] find that decoding an IBF with $m = \alpha \cdot \Delta_{exp}$ and $\alpha = k + 1$, at most Δ_{exp} elements in the symmetric difference, and at least $\Omega(k \cdot \log \Delta_{exp})$ bits in each `keyHashSum` field will succeed with probability at least $\mathcal{O}(\Delta_{exp}^{-k})$. They also indicate that, in practice, $\alpha < 2$ suffices and refer to further results from [34] where they draw the connection to finding the 2-core of random hypergraphs.

In particular, for listing the contents of an IBLT without any hash sums, Goodrich and Mitzenmacher [34] give this theorem:

Theorem 2.3.1. *As long as m is chosen so that $m > (c_k + \epsilon) \cdot t$ for some $\epsilon > 0$ and c_k from eq. (2.13), listing the entries of an IBLT fails with probability $\mathcal{O}(t^{-k+2})$ whenever $n \leq t$.*

$$c_k^{-1} = \sup\left\{\gamma : 0 < \gamma < 1; \forall x \in (0,1), 1 - e^{-k \cdot \gamma \cdot x^{k-1}} < x\right\} \tag{2.13}$$

From this, $c_k \leq k$ can be derived easily by looking at $\gamma = 1/k$. More specifically, though, c_k grows much more slowly with k as indicated by Table 2.1 which shows the numerical values of c_k for $3 \leq k \leq 7$.

Table 2.1: Thresholds for finding the 2-core in random hypergraphs, rounded to four decimal places [34].

k	3	4	5	6	7
c_k	1.222	1.295	1.425	1.570	1.721

By adding a `keyHashSum` for allowing extraneous deletions (required for the set reconciliation algorithm), we must make sure (with high probability) not to

mark an impure cell to be pure because that would fail `IbfDecode`. Hence the requirement on having at least $\Omega(k \cdot \log \Delta_{exp})$ bits in each `keyHashSum` field.

Estimating $|\Delta|$ - The Strata Estimator

A crucial part of the set reconciliation algorithm with invertible Bloom filters is to get a proper estimate Δ_{exp} of the difference set size $|\Delta|$, even if $|\Delta|$ is much smaller than n. Eppstein et al. [26] present the *Strata Estimator* based on a size estimation algorithm by Flajolet and Martin [29] using a hierarchy of `IBF`s.

At first, the universe of potential set elements $\mathbb{U}$ is divided into $L = \log_2(|\mathbb{U}|)$ partitions $P_0, P_1, \ldots, P_{L-1}$ so that P_i covers $1/2^{i+1}$ of $\mathbb{U}$, e.g. by assigning each element i to the partition corresponding to the number of trailing zeros in $H_z(i)$ using some hash function H_z. Then, L `IBF`s are created representing the elements of the set ($S_\mathtt{A}$ or $S_\mathtt{B}$) that fall into the appropriate partition. The structure containing these `IBF`s is called the *Strata Estimator* $SE_{S_\mathtt{A}}$ and $SE_{S_\mathtt{B}}$, respectively. Please note that by using a hash function H_z for partitioning the set, we also effectively reduce the skew among the partitions that may be caused by non-uniformly distributed data in the set.

Assuming node `A` sent $SE_{S_\mathtt{A}}$, node `B` will use its own estimator to decode the set difference size using Algorithm 6. Starting at stratum L and progressing down to 0, it will subtract `A`'s `IBF` from its own and try to decode the resulting `IBF`. If this succeeds, it will add the number of decoded elements to the set difference counter. If it doesn't succeed at index idx, we will return the current set difference counter scaled by 2^{idx+1}.

Algorithm 6 StrataEstimatorDecode [26]

```
function SEDecode(SE_{S_A}, SE_{S_B})          ▷ returns set difference size estimate
    count ← 0
    for idx ← L, L − 1, ..., 0 do
        IBF_{B−A,idx} ← SE_{S_B}[idx] − SE_{S_A}[idx]
        if IBF_{B−A,idx} does not decode then
            return count · 2^{idx+1}
        end if
        count ← count + number of elements in IBF_{B−A,idx}
    end for
    return count
end function
```

Eppstein et al. [26] propose to use 80 cells and 4 hash functions in each `IBF` of their Strata Estimator and scaling this by a factor of 1.39 to ensure that the resulting estimate of Δ_{exp} is greater or equal to Δ_{exp} 99.9 % of the time, based on an empirical evaluation. They claim that, with these settings, this is the only estimator that is accurate at very small set differences and able to handle differences of up to 2^{32} items at the same time. However, they also say that, in practice, they would build a composite estimator that eliminates higher strata

and replacing them by the MinWise estimator [6, 7] since that is more accurate for larger differences.

Please note the theoretical analysis of the Strata Estimator [26] leads to the following bounds in the accuracy of the estimation: For some constants $0 < \epsilon, \gamma < 1$, and by using C cells with k hash functions for the `IBF`s where C and k only depend on ϵ and γ, the Strata Estimator estimates the size of the set difference within a factor of $(1 \pm \gamma)$ of $|\Delta|$ with probability of at least $1 - \epsilon$.

Comments & Summary

Invertible Bloom filters (`IBF`) extend the original Bloom filter data structure even more than the Counting Bloom filters above. With these extensions it is also possible to recover the contents stored within (with high probability). Since this capability remains even after subtracting two `IBF`s from one another, `IBF`s become particularly useful for solving the set reconciliation problem in space that is proportional to the size of the sets' difference (number of cells in the `IBF`, but also used to determine the number of bits in the hash sums) multiplied by the logarithm of the key space (entries in `keySum`). Moreover, `IBF`s do not only provide the set difference Δ, but the specific mappings to $S_{\texttt{A}} \setminus S_{\texttt{B}}$ and $S_{\texttt{B}} \setminus S_{\texttt{A}}$. Please also note that the definition of a failed decoding is very restrictive in saying that it failed if any (few) number of items were not decoded. In the approximate set reconciliation problem we may tolerate a few non-decoded items and rely on them being fixed in a subsequent execution of a set reconciliation service (see Chapter 3 below).

Since the set difference size is needed for tuning the `IBF` correctly, Eppstein et al. [26] also propose a set difference estimation algorithm using a hierarchical `IBF` structure, the *Strata Estimator*. It works with size proportional to the logarithm of the key space to get an estimate within a factor of $(1 \pm \gamma)$ of $|\Delta|$. To ensure that this estimate is always greater than or equal to the true value—and therefore allow a successful set reconciliation—, a correction factor of 1.39 is applied, based on empirical results for an accuracy of 99 %.

Similarly, for the set reconciliation itself, we need to correct this again by a factor of at least c_k to successfully decode the difference with probability $\mathcal{O}(|\Delta|^{-k+2})$. Problems may arise with the $\mathcal{O}$ notation which may hide certain things away, e.g. (constant) factors, that we need to be aware of when aiming for some specific accuracy target. We can see this in the evaluation of the space overhead in [26] where the accuracy target is fixed at 99 % and the overhead to achieve it varies greatly from c_k.

2.4 Estimating the Set Difference Size

Although estimation algorithms for the size $|\Delta|$ of the difference of two (remote) sets $S_\mathtt{A}$ and $S_\mathtt{B}$ are not in the focus of this work, several of the presented algorithms as well as all of our approximate set reconciliation algorithms below do require an estimate of $|\Delta|$ or δ to meet their accuracy targets or select optimal data structure sizes. Therefore, in this section, we will give a brief overview of some of the algorithms in that space.

The simplest way of estimating $|\Delta|$ with low communication overhead is to take a random sample of constant size from one set, e.g. $S_\mathtt{A}$, and search for those in the other set ($S_\mathtt{B}$). However, this will not include the difference from elements in $S_\mathtt{B}$ that are not in $S_\mathtt{A}$ and, additionally, this may become very inaccurate for low δ.

Min-wise sketches [7] extend the random sampling technique by calculating the resemblence of two sets $S_\mathtt{A}$ and $S_\mathtt{B}$, $r(S_\mathtt{A}, S_\mathtt{B}) := {}^{|S_\mathtt{A} \cap S_\mathtt{B}|}/_{|S_\mathtt{A} \cup S_\mathtt{B}|}$, using k independent random permutation functions $\pi_1, \pi_2, \ldots, \pi_k : \mathbb{U} \mapsto \mathbb{U}$. With $\min(\pi_i(S)) := \min(\{\pi_i(s) : s \in S\})$ being the minimum value of all mapped values of elements in S under π_i, a Min-wise sketch M_S contains the k values $\min(\pi_1(S)), \min(\pi_2(S)), \ldots, \min(\pi_k(S))$. We now calculate $M_{S_\mathtt{A}}$ as well as $M_{S_\mathtt{B}}$ and, based on the results of [6], we know that $r(M_{S_\mathtt{A}}, M_{S_\mathtt{B}})$ is an unbiased estimate of $r(S_\mathtt{A}, S_\mathtt{B})$. Therefore, if $M_{S_\mathtt{A}}$ and $M_{S_\mathtt{B}}$ have m matching cells, we estimate $r(S_\mathtt{A}, S_\mathtt{B}) \approx r = {}^m/_k$ and derive the set difference from[d]:

$$
\begin{aligned}
|\Delta| = |S_\mathtt{A} \cup S_\mathtt{B}| - |S_\mathtt{A} \cap S_\mathtt{B}| &= \frac{|S_\mathtt{A} \cup S_\mathtt{B}| - |S_\mathtt{A} \cap S_\mathtt{B}|}{|S_\mathtt{A} \cup S_\mathtt{B}| + |S_\mathtt{A} \cap S_\mathtt{B}|} \cdot (|S_\mathtt{A} \cup S_\mathtt{B}| + |S_\mathtt{A} \cap S_\mathtt{B}|) \\
&= \frac{\frac{|S_\mathtt{A} \cup S_\mathtt{B}| - |S_\mathtt{A} \cap S_\mathtt{B}|}{|S_\mathtt{A} \cup S_\mathtt{B}|}}{\frac{|S_\mathtt{A} \cup S_\mathtt{B}| + |S_\mathtt{A} \cap S_\mathtt{B}|}{|S_\mathtt{A} \cup S_\mathtt{B}|}} \cdot (|S_\mathtt{A}| + |S_\mathtt{B}|) = \frac{1 - r(S_\mathtt{A}, S_\mathtt{B})}{1 + r(S_\mathtt{A}, S_\mathtt{B})} \cdot (|S_\mathtt{A}| + |S_\mathtt{B}|) \\
\Rightarrow \quad |\Delta| &\approx \frac{1 - r}{1 + r} (|S_\mathtt{A}| + |S_\mathtt{B}|)
\end{aligned}
$$

However, while Min-wise sketches may work sufficiently well for large δ, they may easily become very inaccurate for smaller δ or too small values of k.

Similarly to Min-wise sketches, Cormode et al. [21] describe a method to estimate set sizes using a *Dynamic Inverse Sampling* technique that is based on a hierarchy of sums and counts and a small collision detection structure. Since it supports both insertions and deletions, one node could add all its elements while the other nodes deletes its elements from the data structure. We can then derive an approximate set difference size from this result. Queries on such a data structure of size $\mathcal{O}({}^1/_{\epsilon^2} \cdot \log {}^1/_\gamma)$ can be answered with additive error less than ϵ with probability at least $1 - \gamma$. Further sketch-based approaches are described by, for example, Cormode and Muthukrishnan [20] and Schweller et al. [75].

[d] $S_\mathtt{A} \cup S_\mathtt{B} = \emptyset$ can be identified from $M_{S_\mathtt{A}}$ and $M_{S_\mathtt{B}}$ and immediately leads to $\Delta = \emptyset$.

Feigenbaum et al. [28] interpret incoming streams of data items as functions and present a method for approximating the function difference. Their data structure contains $^{\log 1/\gamma}/_{\epsilon^2}$ counters of values at most $|\mathbb{U}| \cdot n$, as well as master seeds with the same (asymptotic) space complexity. By using algebraic computations over finite fields, they achieve an approximation Δ_{exp} which is within a factor of $(1 \pm \epsilon)$ of $|\Delta|$ with probability of at least $1 - \gamma$. This data structure can actually also be viewed as a *sketch* but outperforms Min-wise sketches when approximating the symmetric difference for two sets with small δ.

Similar space complexity as well as approximation error properties are achieved by the Strata Estimator by Eppstein et al. [26] which uses invertible Bloom filters instead of algebraic computations over fields. It is described in detail in Section 2.3.6.

Further uses of Bloom filter variants for estimating the size of the symmetric difference between two sets are presented by Guo and Li [36] and Broder and Mitzenmacher [8]. Guo and Li [36] compare the theoretical number of zero-valued cells in a (small) counting Bloom filter of the difference with the actual number of zeros. Additionally, by looking at the ratio of positive to negative cells, they are able to estimate $|S_\mathtt{A} \setminus S_\mathtt{B}|$ and $|S_\mathtt{B} \setminus S_\mathtt{A}|$ (ref. Section 2.3.5 for details). Based on empirical results, Guo and Li [36] claim a (fairly) exact estimation even for small differences.

Broder and Mitzenmacher [8] present a method to use two sets' (ordinary) Bloom filters and their inner product, i.e. bitwise AND, to estimate the size of the intersection of the two sets. Let $Z_\mathtt{A}$ and $Z_\mathtt{B}$ denote the number of zero bits in the Bloom filters of $S_\mathtt{A}$ and $S_\mathtt{B}$, respectively; also let $Z_\mathtt{AB}$ be the number of zero bits in the inner product of $\mathtt{BF}(S_\mathtt{A})$ and $\mathtt{BF}(S_\mathtt{B})$. Then, $|S_\mathtt{A} \cap S_\mathtt{B}|$ can be derived from

$$\frac{1}{m} \cdot \left(1 - \frac{1}{m}\right)^{-k \cdot |S_\mathtt{A} \cap S_\mathtt{B}|} \approx \frac{Z_\mathtt{A} + Z_\mathtt{B} - Z_\mathtt{AB}}{Z_\mathtt{A} \cdot Z_\mathtt{A}}.$$

With $|S_\mathtt{A} \cap S_\mathtt{B}|$, we can further derive $|\Delta|$ from $|\Delta| = |S_\mathtt{A}| + |S_\mathtt{B}| - 2 \cdot |S_\mathtt{A} \cap S_\mathtt{B}|$. Unfortunately, Broder and Mitzenmacher [8] did not further analyse how good this approximation is. Papapetrou et al. [67] extend on this mechanism and provide theoretical bounds for estimating set cardinalities of not just Bloom filter intersections but also Bloom filter unions.

Irrespective of which algorithm has been used to estimate the size of the difference between two sets, throughout the rest of this work, we will use an estimation as our expected set difference and formalise it as follows.

Definition 6 (Expected Set Difference)**.** *Let $\Delta_{exp} \approx |\Delta|$ be the expected set difference for our set reconciliation algorithms, based on the estimated size of the set difference between S_A and S_B. Similarly, let $\delta_{exp} \approx \delta := {}^{|\Delta|}/_{n}$ be the expected* relative *set difference.*

2.5 An Accuracy Model for Fair Comparisons of Approximate Set Reconciliation Algorithms

Approximate set reconciliation algorithms can be compared in many different ways depending on the requirements, e.g. low time complexity/latency [35] or low communication costs [22, 63, 14], or—to some extend—both [84, 66]. We set a fixed upper bound on the algorithms' accuracy in terms of the number of expected failures and evaluate the algorithms' transfer costs since we believe that algorithms with lower transfer overhead may especially be useful in distributed systems with low available bandwidth. Optimisations for the construction or maintenance of the individual data structures of our algorithms are out of this work's focus but can be applied orthogonally (ref. Sections 7.6 and 8.6).

2.5.1 Accuracy

Recall that an approximate versioned set reconciliation identifies $\Delta'_{\mathtt{A}}$ and $\Delta'_{\mathtt{B}}$ instead of $\Delta_{\mathtt{A}}$ and $\Delta_{\mathtt{B}}$ with some failure probability $P(\Delta'_{\mathtt{A}} \neq \Delta_{\mathtt{A}} \vee \Delta'_{\mathtt{B}} \neq \Delta_{\mathtt{B}})$. To allow a fair comparison between different algorithms with different parameters, we refine the accuracy metric of [45] based on the expected number of errors instead. Compared to using the failure probability above, we thus also take the cardinalities of the differences into account and give bigger weight to outcomes with the same probability but more than one error.

Definition 7 (Failure Rate)**.** *Let X be a discrete random variable with values according to the number of errors x_i of an approximate versioned set reconciliation and the appropriate probabilities p_i according to the chosen algorithm. Let the failure rate fr be the expected number of errors* $\mathrm{E}[X]$ *per set reconciliation attempt:*

$$fr \coloneqq \mathrm{E}[X] = \sum_i x_i \cdot p_i \tag{2.14}$$

Linearity of Expected Values

For two—not necessarily independent—random variables X and Y, the following holds true:

$$\mathrm{E}[a \cdot X + b \cdot Y] = a \cdot \mathrm{E}[X] + b \cdot \mathrm{E}[Y]$$

We will make heavy use of this property in the calculations further below.

Note that, in contrast to using a metric relative to the number of items, we decided to define the failure rate per set reconciliation attempt instead. This does not only allow comparisons with exact set reconciliation algorithms but also allows a system configuration independent of the number of items and thus getting arbitrarily close towards exact set reconciliation.

Using a Close Upper Bound FR on the Failure Rate

☞ For each approximate set reconciliation algorithm, we develop models to deduce individual accuracy-influencing parameters from a given upper bound $FR > 0$ on the algorithms' failure rate fr such that

$$fr \leq FR \text{ and } fr \approx FR$$

Therefore, we try to find parameters where fr is closest to FR.

In the search for parameters to meet $fr \leq FR$—we will generally use the term *fulfil FR* below—, we may be bound to certain restrictions the parameters pose such as bit sizes being integral. Additionally, the algorithms' effective accuracy in the current system state may not be known before executing it, e.g. in cases where the accuracy depends on the actual differences between the two nodes. In this case, we make worst-case assumptions in order to *always* fulfil FR. Overall, we may end up with a more accurate algorithm than anticipated and $fr \lessapprox FR$ which usually implies higher communication costs. The individual algorithms' chapters will include a detailed discussion of these differences and will try to mitigate them as much as possible.

Note that most of the algorithms discussed in this work only have a single accuracy-influencing parameter. The search for a parameter set bringing the failure rate closest to FR in algorithms with multiple accuracy-influencing parameters, however, is more complex. Also, there may be multiple parameter instances with the same accuracy in which case other constraints, e.g. lowest communication costs, lowest computational cost, or lowest time complexity, can be used to select among them. Bloom filters, for example, have two accuracy-influencing parameters but one of them has been optimised for minimal communication cost in literature and thus only one remains. Optima for Bloom filter extensions such as counting Bloom filters or invertible Bloom filters which both retain the same two parameters are not known yet.

Related Work

Guo and Li [36] use a similar approach as above and calculate the expected number of errors in both error cases, i.e. false negatives and false positives. They argue to set a fixed upper accuracy limit for each of them at 1 and would thus achieve an overall accuracy of 2. However, they do not state how to (algebraicly) derive accuracy-influencing parameters from these bounds and also assume the number of hash functions k is a fixed parameter. Hence, they would need to find optimal values for the two parameters k and m with a given bound on the accuracy, e.g. via sampling the parameters.

Similarly to Definition 5 (page 6), Eppstein et al. [26] use a failure probability of $P(\Delta'_{\mathtt{A}} \neq \Delta_{\mathtt{A}} \vee \Delta'_{\mathtt{B}} \neq \Delta_{\mathtt{B}})$ to describe their algorithm's accuracy. They state that their invertible Bloom filter with k hash functions and $C = (k+1) \cdot |\Delta|$

cells achieves a failure probability of $\mathcal{O}(|\Delta|^{-k})$. Closer bounds with lower C are given by Goodrich and Mitzenmacher [34]. However, although they also design an estimation algorithm for the value of $|\Delta|$, they do not use it to derive optimal values of k (and thus C) based on an accuracy constraint as we do. Nevertheless, Eppstein et al. show empirical results of the average accuracy for different values of $|\Delta|$ and k based on 1 000 samples.

In contrast to [26], Byers et al. [10] define the accuracy of their one-way approximate set reconciliation algorithm as the probability that a given element in $\Delta_{\mathtt{A}}$ is identified by node `B` after node `A` sent its approximate reconciliation tree (ART) to `B`. In the analysis of ART, however, they are only concerned with keeping the accuracy constant, i.e. solving the set reconciliation *with high probability*, and only analyse the exact accuracy empirically based on 1 000 samples.

2.5.2 Asymptotic Lower Bound under *FR*

Recall the optimal approximate set reconciliation algorithm depicted in Section 2.1.1 using a perfect oracle so that node `A` knows $\Delta_{\mathtt{A}}$ and sends it to `B` and, similarly, node `B` sends $\Delta_{\mathtt{B}}$ to `A`. `A` and `B` then query their items from $S_{\mathtt{A}} \setminus_v \Delta_{\mathtt{A}}$ and $S_{\mathtt{B}} \setminus_v \Delta_{\mathtt{B}}$ in approximate set representations of $\Delta_{\mathtt{B}}$ and $\Delta_{\mathtt{A}}$. A single such query can be represented by a random variable Y returning a single error at probability ϵ'. Note that there is no more than one error per query in this algorithm and therefore $\mathrm{E}[Y] = 1 \cdot \epsilon'$. By further using the linearity of the expected value, the failure rate of the whole set reconciliation with a total of $q = |S_{\mathtt{A}} \setminus_v \Delta_{\mathtt{A}}| + |S_{\mathtt{B}} \setminus_v \Delta_{\mathtt{B}}|$ queries can thus be given as:

$$fr = \mathrm{E}[q \cdot Y] = q \cdot \mathrm{E}[Y] = q \cdot \epsilon' \tag{2.15}$$

From $fr \leq FR$ and for all practical cases where $q > FR$, we further derive $\epsilon' \leq {}^{FR}/_{q}$ and thus establish the following lower bound in bits from eq. (2.4):

$$|\Delta_{\mathtt{A}}| \cdot \log_2 \frac{1}{\epsilon'} + |\Delta_{\mathtt{B}}| \cdot \log_2 \frac{1}{\epsilon'} = |\Delta| \cdot \log_2 \frac{q}{FR} \tag{2.16}$$

This function is shown in Figure 2.5 for variable q with fixed $|\Delta|$ and selected values of FR (left), and variable FR with fixed $|\Delta|$ and selected values of q (right). Please refer to Figure 2.1 (page 8) for the third combination with variable $|\Delta|$ and fixed $\epsilon = {}^{FR}/_{q}$. Please also note that, as shown by Theorem 2.5.1 below, this is in $\mathcal{O}\left(|\Delta| \cdot \log\left({}^{n}/_{FR}\right)\right)$.

Theorem 2.5.1. *The lower bound of the communication costs of the approximate (versioned) set reconciliation problem with $fr \leq FR$, $q > FR$, and $n := |S_{\mathtt{A}} \cup Mis_{\mathtt{A}}|$ from above is in $\mathcal{O}\left(|\Delta| \cdot \log\left({}^{n}/_{FR}\right)\right)_{n\to\infty,\ FR\to 0}$ bits.*

Proof. This follows immediately from eq. (2.16) and $q \leq |S_{\mathtt{A}}| + |S_{\mathtt{B}}| \leq 2n$. □

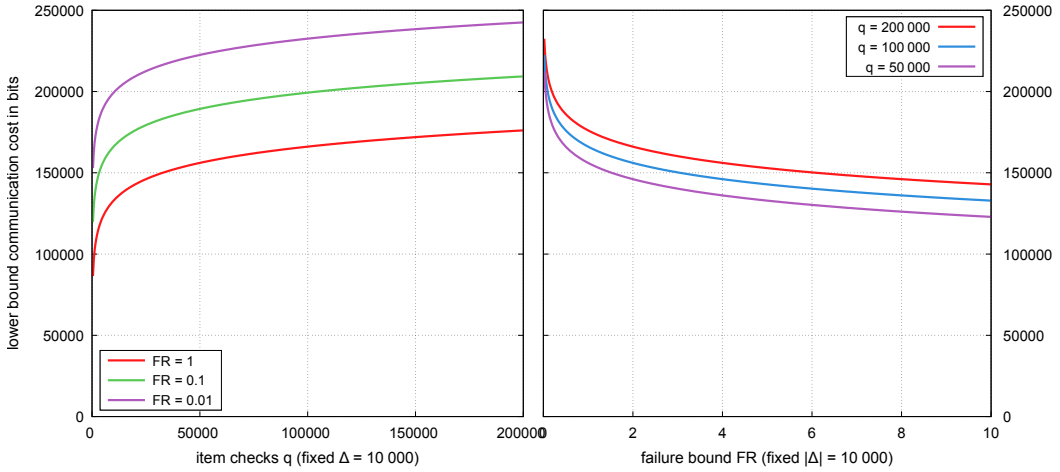


Figure 2.5: Theoretical lower bound for approximate set reconciliation under FR (eq. (2.16)) for $|\Delta| = 10\,000$ and variable q (left) or variable FR (right).

2.5.3 Item Distribution with Hash Functions

Most of the approximate set reconciliation algorithms use hash functions to either verify success, e.g. rsync [81] and hash-based divide-and-conquer by Minsky and Trachtenberg [53], or as part of their data structures, e.g. Bloom filters and their variants [3, 78], Merkle trees [52, 10], and algorithms sending lists of hashes like our trivial and SHash reconciliation in Chapters 5 and 6. A hash function maps items from a set $\mathbb{U}$ to a set of indices I with (usually) $|\mathbb{U}| \gg |I|$. Both use cases require hash functions which distribute items uniformly in I. Similarly, since we want to derive accuracy-influencing parameters from the bound FR, we need to calculate failure probabilities and will thus also assume uniform hash functions in the analysis below.

Commonly, cryptographic hash functions such as MD5 [72] and SHA-1 [24] are used since, in their attempt of making it difficult for an adversary to find a hash collision, they already distribute items uniformly enough for practical inputs [9, 38]. However, even stronger cryptographic hash functions like the extensions of the Secure Hash Standard in [76, 77] do not guarantee a uniform hash distribution.

Universal hash functions [13] provide stronger guarantees for hash collision probabilities, e.g. that the hash values of any two items $x, y \in \mathbb{U}$ collide only with probability $\mathcal{O}(1/|I|)$. This is achieved by using not only a single hash function but instead a class of hash functions and selecting one at random for each input, i.e. a set of items to hash. We could even switch to a different hash function from this class if the chosen one turns out to perform poorly[e].

[e] Although using a different, random hash function on each input is working well with the presented algorithms, this technique may hinder some maintenance optimisations where the hashed data structure is kept over time while elements are added (and deleted). Changing the hash function would then require a full re-hash of all items on both nodes.

By using strongly universal hash functions, i.e. with k-wise independent hash functions [86], we also get the property of uniformly distributed hash values in I. However, while universal hash functions provide guarantees under *any* circumstance, we do find that cryptographic hash functions perform well enough for most practical use cases as in Sections 5.6, 6.6, 7.7 and 8.7 and chapter 9 and do not use this technique.

Chapter 3

Evaluation Method

Among the many applications of set reconciliation algorithms presented in Section 2.2, we chose to evaluate our protocols in a replica repair setting. The embedding of our protocols into a generic replica repair service is described in Section 3.1 followed by a few implementation details in Section 3.2. The parameters for which we analyse this setting, i.e. the evaluation scenarios, are defined in Section 3.3 and we conclude by introducing the metrics we collect and a brief explanation of the plots we use (Section 3.4).

In general, the evaluation is split into several chapters, one for each set reconciliation protocol which is individually described and evaluated (Chapters 4 to 8) and a comparative evaluation in Chapter 9.

3.1 Replica Repair Service

Assume that each node is responsible for a range of items whose keys reside in some interval, e.g. $I_{\texttt{A}}$ for node `A`, and that items are replicated among different nodes each covering an arbitrary interval. These intervals may overlap in which case an item is replicated to all responsible nodes.

In its simplest form, a replica repair service synchronises two nodes' item sets in a common synchronisation interval by identifying the most up-to-date items at either node based on the items' keys and versions. We use the set reconciliation methods presented below and embed these algorithms into a generic replica repair service which is mostly opaque to the reconciliation algorithms. This service is based on a pairwise data synchronisation and shown in Figure 3.1. For each synchronisation request, request-specific services are created on demand to allow multiple parallel instances, e.g. with different replica nodes.

Upon a synchronisation request on node `A`, `A` first sends its own interval $I_{\texttt{A}}$ of keys it covers to another node `B` responsible for (some) replicas of items at `A`. `B` then determines the common synchronisation interval I_{AB} of keys both nodes should cover. This concludes the handshake and starts the reconciliation protocol specific to the selected algorithm using all items in this interval.

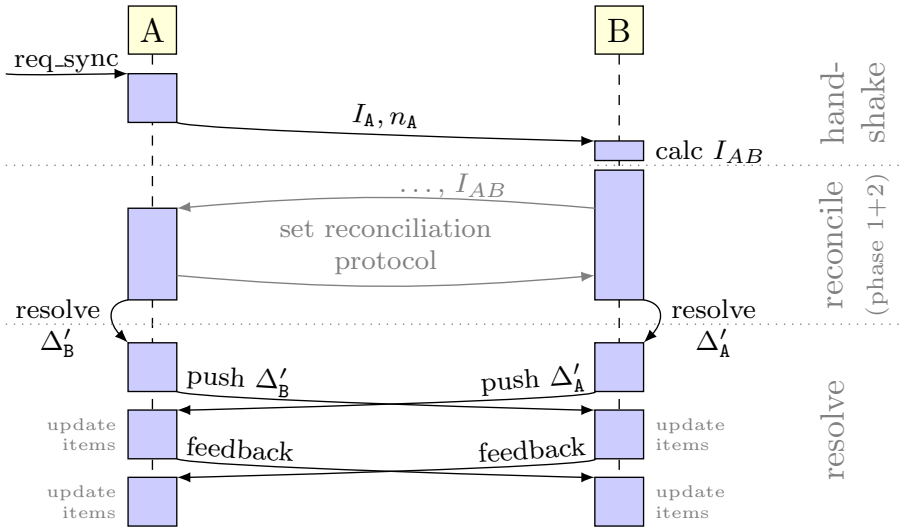


Figure 3.1: Replica repair service with an arbitrary set reconciliation protocol.

Such a protocol typically consists of two phases (ref. Chapters 4 to 8): (1) identifying the differences themselves and (2) identifying which items to send in the next step, i.e. Δ'_B and Δ'_A.

In the final *resolve* step, A and B push the recognised differences Δ'_B and Δ'_A to the other node, respectively. Any items in $\Delta'_A \setminus \Delta_A$ and $\Delta'_B \setminus \Delta_B$ for which a different item exists at A and B is sent back in a feedback message, respectively. Redundant item transfers originate from (a) sending a common item or (b) sending an older item—instead of the newer one—for which the feedback ensures that the item is updated nonetheless, however at the cost of one unnecessary item transfer. Because of the resolved difference, we also add these items to the set of recognised differences Δ' in the evaluation metrics.

Note that we use the push-only approach for resolving items since we are able to create 'pull requests' more efficiently inside our reconciliation algorithms than by simply sending a list of keys to request. Also note that items are only overwritten with newer versions of their replicas during the resolve step so that parallel synchronisation instances always improve the stored set of items.

Instead of simply pushing full item contents to the other node during the two resolve processes, item transfer costs could be optimised by using delta-transfer algorithms such as rsync [81] which efficiently handle cases of minor content changes in files. Optimising file transfers, however, is out of the focus of this work. For rsync as an alternative set reconciliation method, though, please refer to Chapter 4.

3.2 Implementation with Erlang and Scalaris

The set reconciliation algorithms presented in Chapters 5 to 8 are all implemented with Erlang 18.2.3 and evaluated inside the replica repair service of the distributed transactional key-value store Scalaris 0.9+ (git hash d14f8e96d5). Scalaris [74, 73] uses MD5 hashes [72] to create 128 bit item keys from item names and thus $i_k \in [0, 2^{128})$ in our evaluation. Version numbers are arbitrary integers which are increased by 1 for each item change. An item's data is an arbitrary term and further metadata such as locks ensure consistency during changes. Both are not used in the evaluation of the set reconciliation algorithms.

> ℹ Scalaris uses symmetric replication [31] to spread replicas of items in the key space. Node ranges are pairwise disjoint and keys of replicated items are mapped to some other point in the key space thus effectively splitting it into r symmetric segments with r being the replication factor. Therefore, before running set reconciliation algorithms, replicated keys need to be mapped into a common key space—where node ranges are allowed to overlap—in order to match one another.

The naïve set reconciliation protocol as well as the one using rsync (Chapter 4) are comparative examples and are implemented separately in Erlang without the full stack of Scalaris (ref. Appendix A.1, page 161). In case of rsync, our code is a simple wrapper to call the rsync binary with the evaluation setup outlined below.

3.3 Evaluation Scenarios and Experiments

Set reconciliation methods are used in a diverse set of applications for various different systems and environments. We aim at covering these by evaluating the algorithms in a variety of different *scenarios*. Note that despite the concrete setting inside a replica repair service, these scenarios are generic to *any* task that reconciles key-version sets (with subsequent value transfer).

In the versioned set reconciliation problem, items are associated with keys and versions. Item keys which originate from naturally given identifiers such as names are typically prone to a skewed item distribution. Item keys originating from hashes are practically uniformly distributed. In our scenarios, we cover these cases by two different item key distributions (`kdist`) using (a) binomially distributed keys with $B(n, p = 0.2)$ ($data_{bin\,0.2}$) or (b) uniformly distributed keys ($data_{rand}$). Note however, that by applying a hash function to all item keys we can always convert any key distribution to a uniform one, assuming a perfectly random hash function. Some algorithms even require uniformly distributed keys and perform the hashing themselves, others show different effects when applied to differently distributed data. We analyse both cases and raise awareness that a uniform distribution can always be established.

Similarly to the key distribution, the failure distribution (`fdist`) can be modelled in different ways and we present two examples: uniformly distributed failures ($fail_{rand}$) and binomially distributed failures ($fail_{bin_{0.2}}$). Note that by re-hashing the items' keys, the failure distribution also transforms to a uniform distribution.

The failure type (`ftype`) indicates whether items are *outdated* or *missing*. We always evaluate both types separately and note that in mixed scenarios the evaluation metrics below will show a similar mix of the appropriate separately evaluated metrics.

As defined in Section 2.1, the total number of different item keys is $n := |S_{\mathtt{A}} \cup Mis_{\mathtt{A}}| = |S_{\mathtt{B}} \cup Mis_{\mathtt{B}}|$ and the relative number of actual differences is $\delta := {}^{|\Delta|}/_{n}$ (expressed in %). δ is approximated by an (assumed) set difference estimation algorithm that reports the expected difference size δ_{exp}. This δ_{exp} can be set to δ which implies a (fairly) exact estimation which recent approaches claim (ref. Section 2.4). It can also be any other (incorrect) value in order to evaluate the algorithms' sensitivity to wrong estimates.

The whole set of configuration parameters defining a scenario is summarised by Table 3.1.

Table 3.1: Configuration parameters defining a *scenario*.

n	number of item keys
`kdist`	item key distribution ($data_{rand}$ or $data_{bin\,0.2}$)
δ	relative number of differences (0–100 %)
δ_{exp}	expected relative number of differences (0–100 % or δ)
`ftype`	item failure type (*outdated* or *missing*)
`fdist`	failure distribution ($fail_{rand}$ or $fail_{bin_{0.2}}$)

Based on these scenario definitions, we present five types of experiments, each for the *outdated* and *missing* failure types:

(a) scenarios with different values of $\delta_{exp} = \delta$ and uniformly distributed keys ($data_{rand}$, $fail_{rand}$, $n = 100\,000$)

(b) scenarios with different values of δ and uniformly distributed keys but a wrong estimate of $\delta_{exp} \neq \delta$ ($data_{rand}$, $fail_{rand}$, $n = 100\,000$)

(c) scenarios with different data and failure distributions (uniform vs. binomial distribution, $\delta_{exp} = \delta \in (0, 10]\,\%$, $n = 100\,000$)

(d) scenarios with different item numbers n and uniformly distributed keys ($\delta_{exp} = \delta = 3\,\%$, $data_{rand}$, $fail_{rand}$)

(e) scenarios with different accuracy targets FR and uniformly distributed keys ($n = 100\,000$, $\delta_{exp} = \delta = 3\,\%$, $data_{rand}$, $fail_{rand}$)

3.3.1 Simulation Setup

For each scenario with the configuration parameters from Table 3.1, we set up a (pair-wise) reconciliation of two nodes' item sets. We create a database with n items with keys distributed in the address space of a node using `kdist`. The items' (32-bit) version numbers are uniformly distributed in the range $[1, 2^{20})$. Then, for a relative total difference δ, failure items are drawn among the data items based on `fdist`. Based on `ftype`, each failure item is either removed or its version is decreased by a random number in $[1, 512)$ on one of the two nodes uniformly at random. The resulting databases are then distributed to the nodes and the simulation starts with the protocol from Section 3.1 and a selected set reconciliation method.

3.4 Evaluation Metrics

Each of the scenarios defined above is set up 1 000 times differently due to the given random distributions. During the evaluation, we will show the averages and the standard deviation of the following metrics related to the set reconciliation protocol and collected by the simulation:

$|\Delta|$ missed: the (absolute) number of false negatives, i.e. unreconciled items which were wrongly identified as not being in the set difference

Red.: the (absolute) number of redundantly transferred data items (false positives), i.e. pushing an item identified as missing when it is not or pushing an outdated or equal version to the receiving node

Transfer costs: the costs of the two different phases of the set reconciliation algorithm (without handshake or resolve) in KiB based on zlib-compressed messages

Note that from a practical perspective, redundantly transferred items are not "real" errors and only cause additional costs. These costs—in terms of transferred bytes, computational overhead, or time complexity—depend on the actual items and their sizes and the transfer algorithm used in the two resolve steps. Also, the effect of these increased costs varies and may become problematic in bandwidth-limited systems, for example. We thus present false positives as errors but separate them from the false negatives.

3.4.1 Plots

The plots we use in our evaluation condense a lot of information from the actual scenarios used and the metrics collected. Figures 3.2 and 3.3 show examples for experiment types (a) to (e) (Section 3.3). Each of these plots is composed

of sub-plots in a matrix of the three metrics (one per row) and the two failure types (one per column). A common x-axis is shown at the bottom, error bars denote the standard deviation and may be cut off if too high.

In the plots for experiment types (a), (b), (d) and (e) (Figure 3.2), *transfer costs* of phase 1 are shown as darker colour bars with lighter colour bars stacked on top being phase 2 costs. The x-axis either shows different values of δ or n.

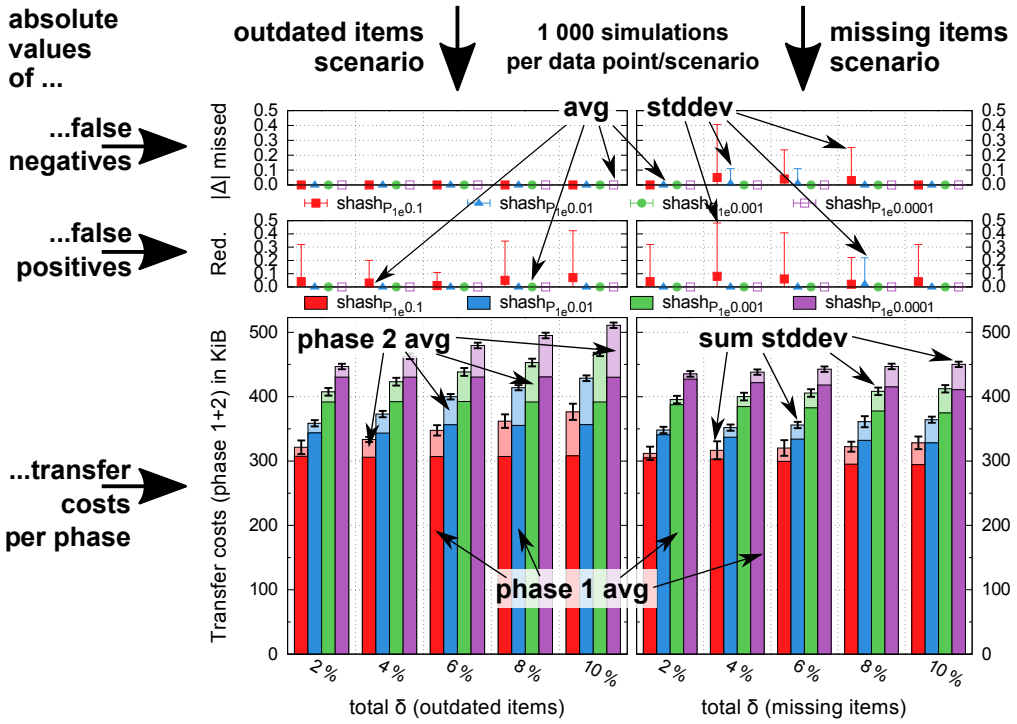


Figure 3.2: Example plot of experiment types (a), (b), (d) and (e) with descriptions added (avg = average, stddev = standard deviation).

For experiment type (c) (different data and failure distributions), the relative metrics of a given algorithm compared to the $data_{rand}, fail_{rand}$ scenario are shown, i.e. Figure 3.3 shows the differences to Figure 3.2. Here, transfer costs show the differences of the total transfer costs, i.e. the differences of the sums of the two phases.

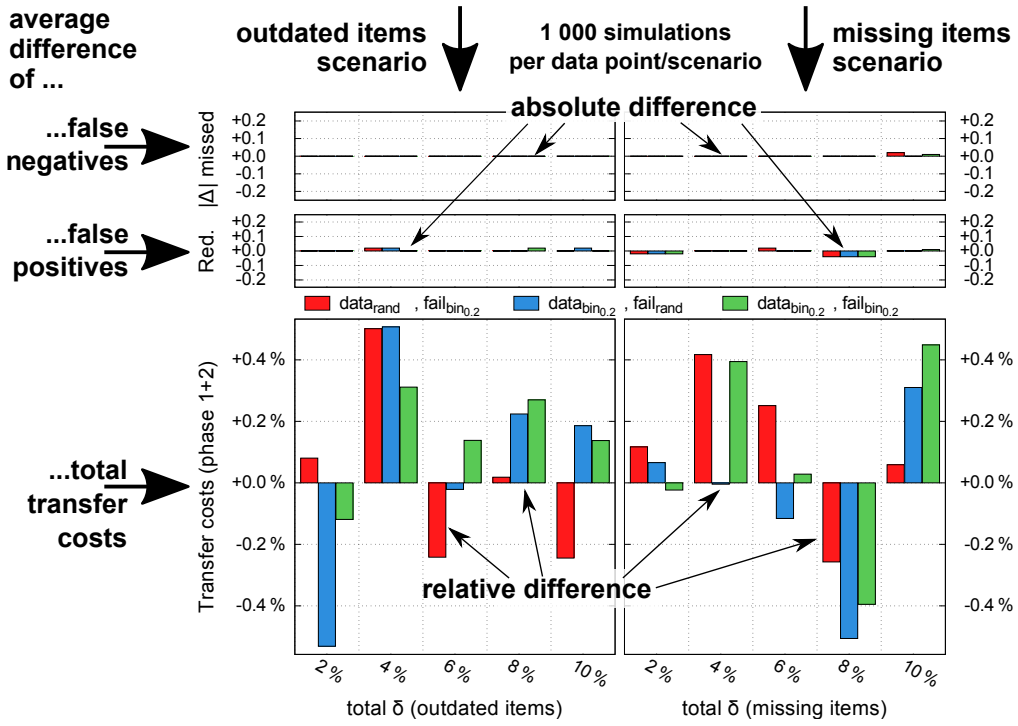


Figure 3.3: Example plot of experiment type (c) showing different data and failure distributions compared to $data_{rand}, fail_{rand}$.

Chapter 4

Naïve and Rsync Reconciliation

To put things into perspective, we take a brief look at how the naïve approach as well as rsync [81]—regardless of being exact and quasi-exact algorithms, respectively—perform in a similar scenario. We simulate the first phase of the reconciliation, i.e. sending the key-version list from node B to node A, excluding the overhead of signalling B which items to push back to A—thus only A is aware of the differences. These (missing) phase 2 costs are similar to the *trivial* reconciliation below and are thus negligible (ref. Section 5.6).

4.1 Protocol

Figure 4.1 sketches the protocol used by the naïve algorithm and rsync. Although not being included in the following discussions and evaluation, phase 2 is added here for completeness.

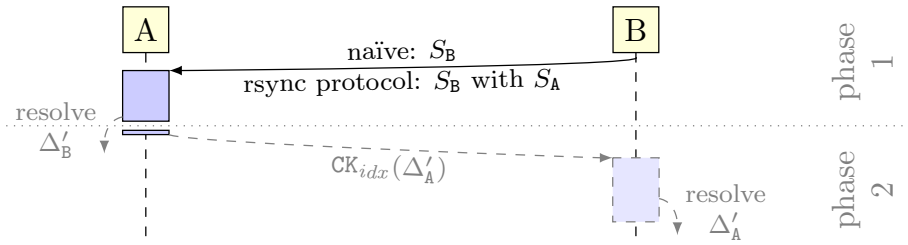


Figure 4.1: Naïve and rsync reconciliation protocol.

The naïve approach sends a zlib-compressed key-version list of S_B in phase 1. Rsync is actually not a data structure sent over in one message but an own protocol running in $\mathcal{O}(1)$ rounds. It has an efficient delta-transfer algorithm used for synchronising two files on different hosts over limited bandwidth connections. To reconcile the two item sets at nodes A and B with rsync, we create two files f_A and f_B with all key-version pairs at node A and B, respectively. Rsync's delta-transfer algorithm is then used to re-construct f_B at node A which can thus determine Δ'_B.

4.2 Algorithm Details

While there is not much to describe for the naïve approach, there are some optimisations in the rsync version used for evaluation not described in detail in [81] or [82]. Unfortunately, the official rsync website[a] does not describe the current protocol version either, so the following description is partly based on the source code of version 3.1.1 (protocol version 31).

If two nodes `A` and `B` have two similar files f_A and f_B, respectively, the rsync delta-transfer algorithm allows an efficient transfer of a file on one node to the other node, e.g. f_B to `A`. Thereby, only pieces of f_B which are different and not found in f_A plus a small amount of checksums are sent. The basic algorithm works as follows [82]:

1. f_A is split into non-overlapping blocks of (fixed) size s bytes (default in rsync 3.1.1: $s = \left\lceil \sqrt{|f_A|} \cdot 1/8 \right\rceil \cdot 8$ with file size $|f_A|$ in bytes)
2. for each block, two checksums are calculated at node `A`: (a) a weak "rolling" checksum (32 bit) and (b) a strong checksum (128 bit)
3. node `A` sends these checksums to node `B`
4. `B` searches f_B for blocks of size s at *any* offset with the same weak and strong checksum as one of the blocks of f_A
5. `B` sends instructions for `A` to re-construct f_B using either references to blocks of f_A or sending literal data from f_B (for non-matching blocks)

Herein, the rolling checksum must have the property that it is very cheap to calculate the checksum of bytes $x_2 \dots x_{i+1}$ given the checksum of bytes $x_1 \dots x_i$ as well as x_1 and x_{i+1}. It is based on Adler-32. The strong checksum was originally defined as MD4 but since protocol version 30, MD5 is used.

4.2.1 Improvements in Later Versions

Protocol version 14 changed the fixed strong checksum size from 128 to 16 bits with huge savings in step 3. It was made dynamic in protocol version 27 where two phases were introduced. In the first phase, the strong checksum's actual size `schksum_bits` varies between 16 and 128 bits based on the file size $|f_A|$ and the block size s. In rsync 3.1.1 (protocol version 31), it is given as:

$$\texttt{schksum\_bits} = \left\lceil (10 + 2 \cdot \log_2(|f_A|) - \log_2(s) + 1 - 32) \cdot 1/8 \right\rceil \cdot 8 \qquad (4.1)$$

Step 5 then also sends a full strong checksum (128 bit) of the whole file f_B. If this checksum does not match after re-constructing f_B at node `A` then the whole process is repeated in a second phase using 128 bits for the strong checksum of step 2. If it still does not match afterwards, the whole file is sent.

[a] `https://rsync.samba.org/` (retrieved 22 April 2016)

4.3 Evaluation

Similarly to the other algorithms below, for each scenario, we create (binary) files for the two nodes by concatenating a 128 bit key and a 32 bit integer version number $\in [1, 2^{20})$ for each of the appropriate node's items and select a node with an outdated/missing item at random. (ref. Listing A.1 on page 161).

Phase 1 costs of the naïve approach are equal to the size of the zlib-compressed version of the key-version file. For rsync, we synchronise the key-version files using `rsync -B<s> -Iz -no-W -stats` with different block sizes s in bytes (parameter `-B`). Parameter `-I` synchronises the two (key-version) files independently of their size and time; `-no-W` forces the use of the delta-transfer algorithm for local files, too. The other parameters enable compression (`-z`)—especially useful for the re-construction instructions sent from `B` to `A`—and some more statistics (`-stats`). Algorithms given as $\texttt{rsync}_s$ present default rsync synchronisations with block size s. Additionally, we provide $\texttt{rsync}_{s,cs128}$ using a patched rsync that always uses strong checksums of 128 bit size for comparison with our algorithms below which do not use similar optimisations.

4.3.1 General Analysis for Different δ

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$fail_{rand}$

The number of bytes transferred during these synchronisations (the sum of the sent and received bytes) is shown in Figures 4.2 and 4.3 together with the naïve algorithm. The naïve algorithm's transfer costs are as expected. In the *outdated* scenarios, this is roughly $n \cdot (128 + 32)$ bits due to the low compression rate of random data. The communication cost of the *missing* scenarios varies with the number of items on each node since higher values of δ lead to fewer items on each node and in turn fewer items to transmit.

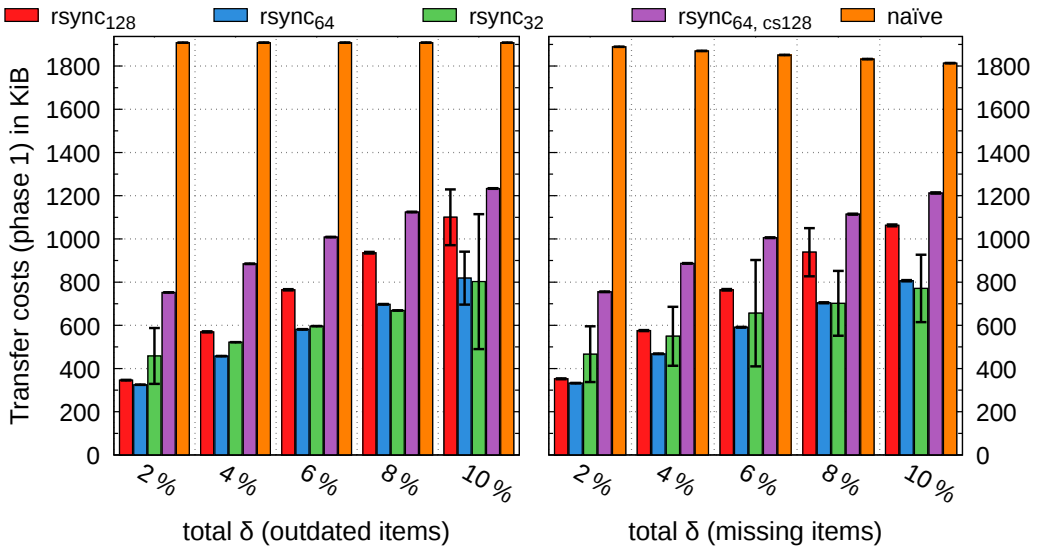


Figure 4.2: *Naïve* and *rsync* reconciliation costs with small δ.

Rsync, however, transfers two checksums per block and only transmits a block's data if it is not found on the other node. Therefore, there is a constant cost of transmitting all checksums and an increasing cost for the block mismatches. Eventually, this is more expensive than the naïve file transfer. Depending on the number of block mismatches, one of these two costs dominates the other and determines whether smaller or larger block sizes are beneficial.

Without further knowledge of the failure distribution, the number of block mismatches cannot be accurately estimated. With a preceding re-hash and re-sorting step of the items' keys, however, a uniform (key and) failure distribution can be established but since rsync actually benefits from skewed, non-uniform distributions (ref. Section 4.3.2), this step may be disadvantageous.

The *default* block size for a file with $n \cdot (128 + 32) = 2\,000\,000$ bytes is 1 408 bytes and although this makes the $\delta = 0\,\%$ case very cheap, it is disadvantageous for any other δ and thus not shown here. A block size of 64 bytes seems to be a good trade-off for the $0\,\% < \delta \leq 10\,\%$ scenario with uniformly distributed keys and failures (Figure 4.2) and is reasonably good for larger δ, too (Figure 4.3). For this block size, we also present the results of an unoptimised rsync with full, i.e. 128 bit, strong block checksums where sending the checksums is considerably more expensive.

Figure 4.2 also shows the benefits of the two phases introduced into rsync protocol version 27 for the average case: With a blocksize of $s = 64$, there are 31 250 blocks and sending 32+128 bits for each of them would yield a constant overhead of 610 KiB. Instead, only 32+16 bits are sent which results in only 183 KiB total checksum size. However, there are cases when rsync has to fall back to the full 128 bit strong block checksum in a second phase which causes the increased values of the standard deviation in the plots.

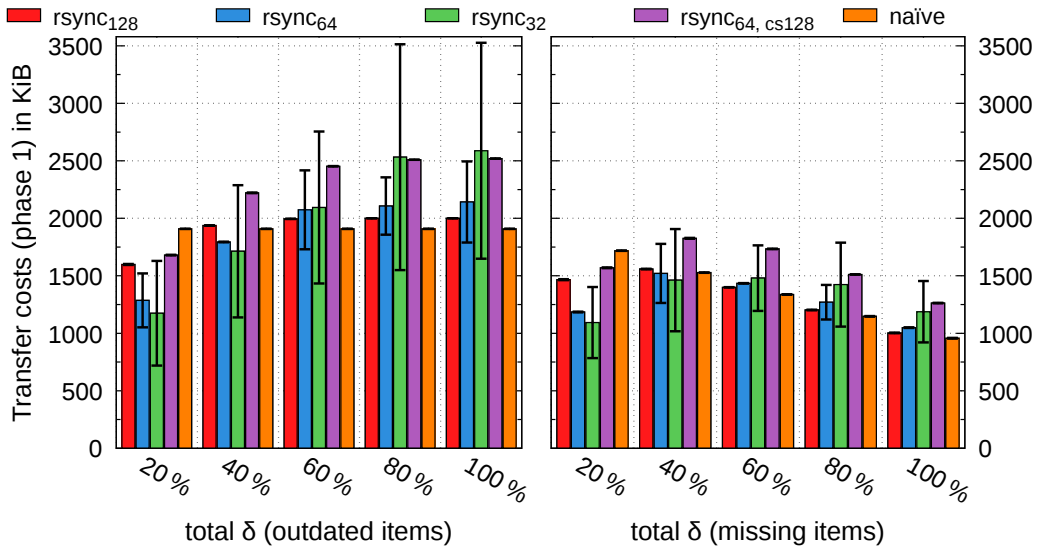


Figure 4.3: *Naïve* and *rsync* reconciliation costs with high δ.

4.3.2 Data and Failure Distribution Sensitivity

Note that the actual transfer costs of rsync depend on the failure distribution. The more failures cluster inside the blocks of size s, the less literal data tokens need to be transmitted. Therefore, the uniform distribution presented here is rsync's worst case scenario. Please also note that (at least) in the *outdated* scenario, rsync cannot gain from the file-wide search for matching blocks due to our quasi-random data (keys and versions). In the *missing* scenario, rsync may find a matching block if there is only one error at the beginning of the block.

4.3.3 Scalability with the System Size n

$n = variable$
$data_{rand}$
$\delta = 3\,\%$
$fail_{rand}$

Figure 4.4 shows how the naïve and rsync reconciliation protocols perform when the data size n is increased. Compared to the uncompressed $n \cdot (128 + 32)$ bits, the naïve algorithm shows a slight increase in compression rate but otherwise progresses as expected. Similarly, rsync progresses linearly for any fixed block size s, a 128 bit hash sum each in the worst case, and a number of roughly $\delta = 3\,\%$ of the blocks being different and thus transmitted in the $data_{rand}$ and $fail_{rand}$ scenario.

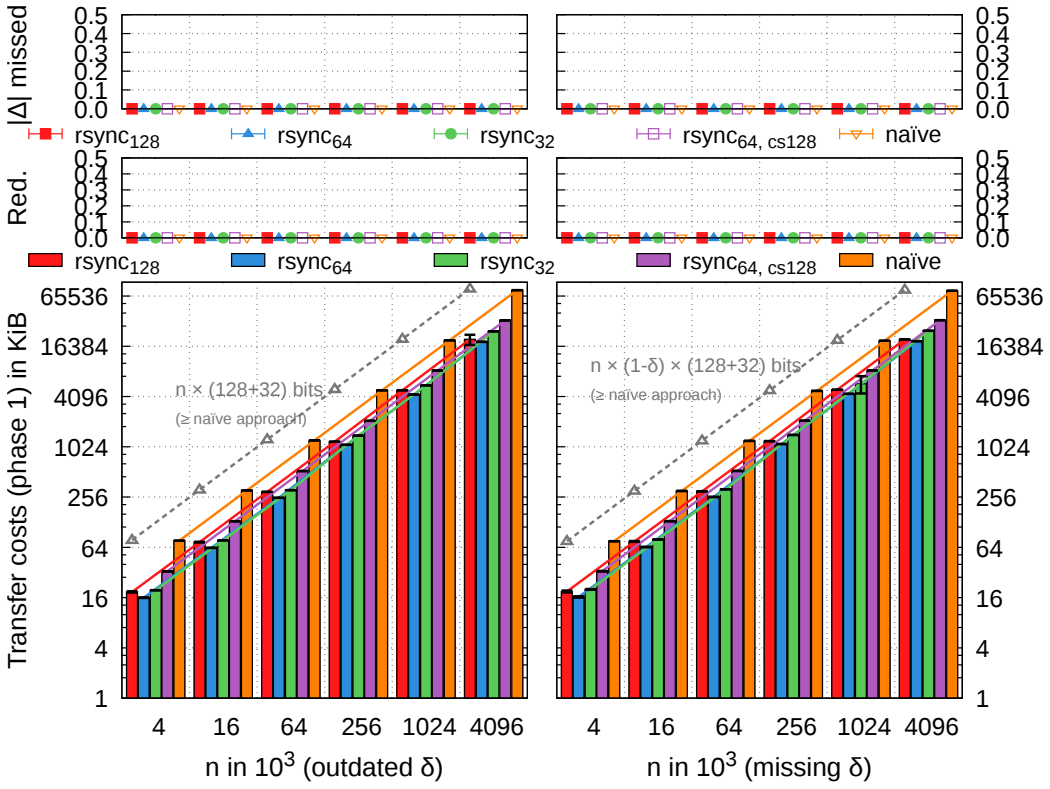


Figure 4.4: *Naïve* and *rsync* reconciliation scalability with data size n.

Chapter 5

Trivial Reconciliation

In this chapter we introduce a trivial *approximate* set reconciliation protocol based on the naïve protocol of sending the full key-version list. Furthermore, we apply our accuracy model and derive appropriate parameters to bind the *trivial* reconciliation's number of expected failures by a fixed value of *FR*.

5.1 Protocol

The *trivial* reconciliation protocol is depicted in Figure 5.1. Node `B` first creates a list of compressed keys and version numbers of its items, sorts them and packs them into a delta-encoded[a] compressed key-value structure `CKV` which is detailed below (ref. Section 5.2). `CKV` is then transmitted to node `A` together with the used parameters. `A` checks whether its data items are present in `CKV` and which items are newer or older. In this first phase, $\Delta'_{\mathtt{B}}$ and $Old'_{\mathtt{A}}$ are known to `A` which pushes $\Delta'_{\mathtt{B}}$ to `B`. $Mis'_{\mathtt{A}}$ is identified by all unmatched keys in `CKV`. In *phase 2*, we thus pack the indices of $Old'_{\mathtt{A}}$ and $Mis'_{\mathtt{A}}$ inside `CKV` into a delta-encoded $\mathtt{CK}_{idx}$ binary and send it to `B`. `B` translates them back into keys and resolves the remaining $\Delta'_{\mathtt{A}}$. The delta-encoding as well as the use of indices instead of the keys themselves potentially reduces the number of bits and allows a better compression. This is due to larger similarities inside `CKV` and $\mathtt{CK}_{idx}$ depending on the data and failure distribution, respectively.

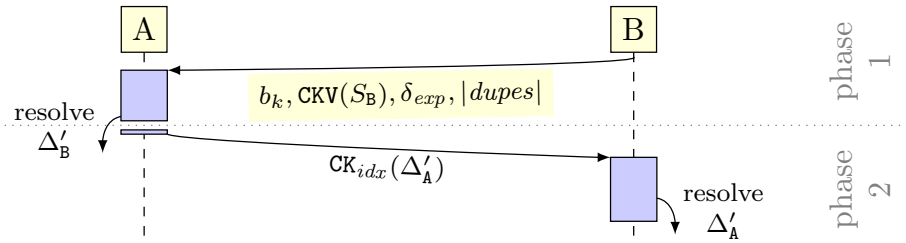


Figure 5.1: *Trivial* reconciliation protocol.

[a] `http://en.wikipedia.org/wiki/Delta_encoding` (retrieved 10 August 2016)

5.2 `CKV` Data Structure Details

In order to create the compressed key-value structure `CKV` (Algorithm 7), item keys are first compressed by hashing them to b_k bits, e.g. by using the b_k least significant bits of an MD5 hash, and thus distributing them uniformly in $[0, 2^{b_k})$. We then sort these entries by their hashed keys, add items with non-unique hashes to $\Delta'_{\mathtt{A}}$ (ref. Section 5.3), and remove them from the list which is then split into a hash list and version list with elements at the same position belonging to each other. The hash list is further compressed using delta-encoding; the version list concatenates all 32-bit version numbers.

Algorithm 7 `CKV` creation

```
function CKV(Entries)
    for all i ∈ Entries do                                  ▷ hash all items' keys
        i_h ← MD5(i_k, b_k)        ▷ use the b_k least significant bits of an MD5 hash
    end for
    SORT(Entries)                                           ▷ sort by item hashes i_h
    Δ'_A ← REMOVEDUPES(Entries)        ▷ remove non-unique i_h, add them to Δ'_A
    hashes ← [], versions ← []                              ▷ create two empty lists
    for all i ∈ Entries do                  ▷ split Entries into two separate lists
        hashes.APPEND(i_h)
        versions.APPEND(i_v)
    end for
    HBin ← DELTAENCODE(hashes, b_k)   ▷ apply the Delta-Encoding from below
    VBin ← CONCATENATE(versions)      ▷ concatenate all 32 bit version numbers
    return {HBin, VBin}      ▷ a tuple of the compressed hashes and the versions
end function
```

5.2.1 Delta-Encoding

By using delta-encoding, we do not transmit the list of sorted hashes, i.e. integers, themselves but encode the differences between two consecutive hashes and send this list instead. This provides two advantages: (a) the number of bits needed to encode the difference may be lower than the original hash size and (b) the differences may be more alike. Both allow a better compression.

Assuming a sorted list of integers $k_j \in [0, 2^{b_k}) \subset \mathbb{N}, j \in \{1, 2, \ldots, x\}$, standard delta-encoding transmits $k_1, (k_2 - k_1), \ldots, (k_x - k_{x-1})$. Since we remove any duplicates (and thus $k_j < k_{j+1}$), we instead transmit $k_1, (k_2 - k_1 - 1), \ldots, (k_k - k_{k-1} - 1)$ for further potential savings. We determine the maximum max_{diff} among these and pack each number into $b_d := \lceil \log_2(max_{diff} + 1) \rceil \leq \lceil \log_2(2^{b_k}) \rceil$ bits of the transmitted binary. A prefix of $\lceil \log_2(b_k) \rceil$ bits adds b_d and allows the receiving node to decode the binary analogously with its knowledge of b_k.

The following listings show two examples using Erlang pseudo-code where `[1,2,...]` denotes a list of integers and `«a:b,c:d,...»` denotes a bitstring encoding integer `a` using `b` bits, `c` using `d` bits etc., both in the given order.

```
DeltaEncode([3,7,11,15,18,20,23,27,36,44,47], 32) % b_k = 32
%Input :                          -- Max: 47 => b_d = 6 => 72 bits
%Delta : 3,4,4,4,3,2,3,4,9,8,3    -- Max:  9 => b_d = 4 => 50 bits
%Delta': 3,3,3,3,2,1,2,3,8,7,2    -- Max:  8 => b_d = 4 => 50 bits
«4:6,3:4,3:4,3:4,3:4,2:4,1:4,2:4,3:4,8:4,7:4,2:4»
```

```
DeltaEncode([2,8,11,12,16,19,20,21,22,30,32], 16) % b_k = 16
%Input :                          -- Max: 32 => b_d = 6 => 71 bits
%Delta : 2,6,3,1,4,3,1,1,1,8,2    -- Max:  8 => b_d = 4 => 49 bits
%Delta': 2,5,2,0,3,2,0,0,0,7,1    -- Max:  7 => b_d = 3 => 38 bits
«3:5,2:3,5:3,2:3,0:3,3:3,2:3,0:3,0:3,0:3,7:3,1:3»
```

5.3 Using CKV for Set Reconciliation

During the reconciliation, each of the $n_\mathtt{A}$ keys of node A is hashed the same way as those in CKV and then compared with each of the $n_\mathtt{B}$ b_k-bit hashes in CKV (see Algorithm 8). If a match is found, the items' versions are compared and the item is added to the appropriate difference set, i.e. $\Delta'_\mathtt{A}$ or $\Delta'_\mathtt{B}$. Items with identical versions are ignored. If no match is found, $i \in S_\mathtt{A}$ must be in $Mis_\mathtt{B}$ and is added to $\Delta'_\mathtt{B}$. At the end, after checking all items of $S_\mathtt{A}$, all unmatched hashes in CKV are added to $\Delta'_\mathtt{A}$ and requested in phase 2 (ref. Section 5.3.2).

Algorithm 8 *Trivial* reconciliation at node A

```
function TrivialSync(S_A, CKV, b_k)      ▷ after receiving b_k and CKV (S_B) from B
    {HBin, VBin} ← CKV                   ▷ a tuple of the compressed hashes and the versions
    hashes ← DeltaDecode(HBin, b_k)      ▷ revert the Delta-Encoding from above
    versions ← Split(VBin)               ▷ retrieve individual 32 bit version numbers
    for all i ∈ S_A do                                          ▷ hash all items' keys
        i_h ← MD5(i_k, b_k)              ▷ use the b_k least significant bits of an MD5 hash
    end for
    Δ'_B ← RemoveDupes(S_A)              ▷ remove non-unique i_h, add them to Δ'_B
    for all i ∈ S_A do                                  ▷ for all remaining items of S_A
        if idx = hashes.Find(i_h) then                  ▷ find index of i_h in hashes
            if versions[idx] > i_v then                                  ▷ newer item on B
                Δ'_A.Append(i_h)             ▷ request the item behind this hash from B
            else if versions[idx] < i_v then                             ▷ newer item on A
                Δ'_B.Append(i)                                           ▷ send this item to B
            end if
            hashes.Remove(idx), versions.Remove(idx)
        else                                 ▷ not present in hashes, i.e. not found in CKV
            Δ'_B.Append(i)                                               ▷ send this item to B
        end if
    end for
    Δ'_A.AppendAll(hashes)               ▷ request all items from unmatched hashes
end function
```

5.3.1 Implications of Hash Collisions

It may happen that different items from $S_\mathtt{B}$ collide, i.e. they have the same hash although their keys differ. Also, items from $S_\mathtt{A}$ may collide with each other and with items from $S_\mathtt{B}$. Overall, $|S_\mathtt{A} \cup Mis_\mathtt{A}| = |S_\mathtt{B} \cup Mis_\mathtt{B}| = n$ keys may collide. For each hash collision among items whose (different) keys are hashed to the same value h, i.e. with keys from $X_h \coloneqq \{i_k \in S_\mathtt{A} \cup S_\mathtt{B} : i_h = h\}$ if there is more than one key, node `A` needs to request $X_{h,\mathtt{B}} \coloneqq \{i \in S_\mathtt{B} : i_k \in X_h\}$ and needs to send $X_{h,\mathtt{A}} \coloneqq \{i \in S_\mathtt{A} : i_k \in X_h\}$ in order to identify all items of Δ. This follows from the worst-case scenario that $X_{h,\mathtt{B}} \subseteq Mis_\mathtt{A}$ and $X_{h,\mathtt{A}} \subseteq Mis_\mathtt{B}$.

During the reconciliation, however, we cannot distinguish all possible cases and thus cause the following errors (summarised by Table 5.1):

- *a single collision of $x \in S_A$ with $y \in S_B$, i.e. $X_{x_h} = \{x_k, y_k\}, y_k \neq x_k$, $|X_{x_h,A}| = |X_{x_h,B}| = 1$:* We cannot distinguish this case from a valid match with $x \in S_\mathtt{B}$ and thus assume that $x \in S_\mathtt{B}$. Therefore, we fail to recognise two items if $x_v = y_v$ and one item if $x_v \neq y_v$. Note that this case only results from $x \in Mis_\mathtt{B}$ and $y \in Mis_\mathtt{A}$.

- *multiple collisions in X_h with at least two items from S_A or from S_B*: Keys in X_h originate from either common items or actual differences. Hence `A` requests $X_{h,\mathtt{B}}$ and sends $X_{h,\mathtt{A}}$. If all of them are common items, though, $2 \cdot |X_h|$ items are redundantly transferred. For any outdated/newer item in $X_{h,\mathtt{B}}$ and $X_{h,\mathtt{A}}$, either the old or the new item is sent and the opposite is requested (each is present in one of the sets!). The two messages from resolving the outdated one (including the feedback message; ref. Section 3.1) are redundant and thus the same bound applies. Missing items do not cause any overhead.

The total number of errors is thus bound by twice the number of keys with non-unique hashes (among the n unique keys). Also, this is the closest bound we can give without assuming further knowledge on the failure distribution.

Table 5.1: *Trivial* reconciliation errors for hash value h with collision set X_h.

Failure Case	Result
$X_h = \{x_k \in Mis_\mathtt{B}, y_k \in Mis_\mathtt{A}\}, x_v = y_v$	two unrecognised items of Δ
$X_h = \{x_k \in Mis_\mathtt{B}, y_k \in Mis_\mathtt{A}\}, x_v \neq y_v$	one unrecognised item of Δ
$\lvert X_{h,\mathtt{A}}\rvert \geq 2$ or $\lvert X_{h,\mathtt{B}}\rvert \geq 2$	none to $2 \cdot \lvert X_h\rvert$ redundant transfers
$\hookrightarrow$ if only outdated/newer, common	$2 \cdot \lvert X_h\rvert$ redundant transfers

Note that Algorithm 8 above realises the behaviour for the *multiple collisions* case by removing all items in $S_\mathtt{B}$ with non-unique hashes from `CKV` and adding them to $\Delta'_\mathtt{A}$. Since then, items of $S_\mathtt{A}$ with the same collision are not found in `CKV`, they are sent to `B` as desired. Similarly, all items in $S_\mathtt{A}$ with non-unique hashes are added to $\Delta'_\mathtt{B}$ and not checked against `CKV`. Therefore, any item in `CKV` with the same collision is unmatched and added to $\Delta'_\mathtt{A}$.

5.3.2 Phase 2 Details – $\texttt{CK}_{idx}$

In phase 2, all items in $\Delta'_{\texttt{A}}$ are requested from B using their (sorted) positions in the CKV structure and applying delta-encoding as presented by Algorithm 9. The resulting $\texttt{CK}_{idx}$ is not only small but also allows further (zlib) compression. Let the maximum difference be max_{diff}, then $\texttt{CK}_{idx}$ only requires $s := \lceil \log_2(max_{diff} + 1) \rceil \leq \lceil \log_2(n'_{\texttt{B}}) \rceil \leq \lceil \log_2(n_{\texttt{B}}) \rceil$ bits per index—with $n'_{\texttt{B}}$ being the number of hashes encoded in CKV—and a prefix of size $t := \lceil \log_2(\lceil \log_2(n'_{\texttt{B}}) \rceil + 1) \rceil$ bits to encode s so that B can decode $\texttt{CK}_{idx}$ again.

Algorithm 9 $\texttt{CK}_{idx}$ creation

```
function CK_idx(hashes, Δ'_A)                    ▷ hashes from CKV, Δ'_A with item hashes
    n'_B ← hashes.SIZE( ), indices ← []          ▷ start with an empty index list
    for idx ← 0, 1, ..., n'_B − 1 do             ▷ traverse hashes in order
        if hashes[idx] ∈ Δ'_A then               ▷ lookup hash at index idx
            indices.APPEND(idx)                  ▷ add the index to the end
        end if
    end for
    return DELTAENCODE(indices, ⌈log2(n'_B)⌉)    ▷ see Section 5.2.1
end function
```

5.4 Parameter Deduction from *FR*

Although the *trivial* set reconciliation protocol consists of two phases, only the first one is approximate and we derive an appropriate b_k so that it fulfils *FR*.

5.4.1 Phase 1

Since version comparisons are exact as long as items are correctly matched with their counterparts, errors are only caused by hash collisions and the expected number of errors is bound by $2 \cdot \text{E}[\text{non-unique hashes}]$ (ref. Section 5.3.1). The expected number of non-unique hashes can be determined using its linearity and the expectation of each of the n item keys' hashes being non-unique[b] assuming uniformly distributed hashes of $n = |S_{\texttt{A}} \cup Mis_{\texttt{A}}|$ unique keys:

$$\text{E}[\text{non-unique hashes}] = n \cdot \Big(\underbrace{1 - \big(1 - \overbrace{1/2^{b_k}}^{\text{probability that two given hashes collide}}\big)^{n-1}}_{\substack{\text{probability of a given hash colliding with at least one of the } n-1 \text{ others} \\ =\text{E[a given hash is not unique]}}} \Big) \tag{5.1}$$

With this, we calculate an appropriate b_k so that for the *trivial* algorithm's failure rate fr_t the accuracy bound $fr_t \leq FR$ holds:

[b] Note that the n random variables that each originate from a given item key's hash being non-unique are not independent but this is not required for the linearity of the expected value (ref. Section 2.5.1).

$$fr_t \leq 2 \cdot \mathrm{E}[\text{non-unique hashes}] = 2 \cdot n \cdot \left(1 - (1 - 1/2^{b_k})^{n-1}\right) \leq FR \quad (5.2)$$

$$\Leftarrow \quad \frac{FR}{2n} \geq 1 - (1 - 1/2^{b_k})^{n-1}$$

$$\Leftrightarrow \quad 1 - 1/2^{b_k} \geq \sqrt[n-1]{1 - \frac{FR}{2n}}$$

$$\Leftrightarrow \quad 2^{b_k} \geq \frac{1}{1 - \sqrt[n-1]{1 - \frac{FR}{2n}}}$$

$$\Leftrightarrow \quad b_k \geq \log_2 \frac{1}{1 - \sqrt[n-1]{1 - \frac{FR}{2n}}} \quad (5.3)$$

An Upper Bound on the Number of Unique Keys

In order to calculate an appropriate b_k from eq. (5.3), we need the number of different keys n and approximate an $\tilde{n} \gtrapprox n$ using a known $\delta_{exp} \gtrapprox \delta$. If δ_{exp} is unknown, we can use the pessimistic assumption of $\delta_{exp} = 100\,\% \ (= 1.0)$. Here, the *outdated* scenario constitutes a best case due to $n_{\mathtt{A}} = n_{\mathtt{B}} =: n_{Old}$. In the *missing* scenario, however, $n_{Mis} \geq n_{Y \in \{\mathtt{A},\mathtt{B}\}}$ can be calculated from $n_{\mathtt{A}}$ and $n_{\mathtt{B}}$ assuming that the differences δ are distributed to the two nodes using some factor $\alpha \in [0, 1] \subset \mathbb{R}$:

$$\begin{aligned} n_{\mathtt{A}} = n_{Mis} \cdot (1 - \delta \cdot \alpha) \quad &\Leftrightarrow \quad n_{Mis} \cdot \delta \cdot \alpha = n_{Mis} - n_{\mathtt{A}} \\ n_{\mathtt{B}} = n_{Mis} \cdot (1 - \delta \cdot (1 - \alpha)) \quad &\Leftrightarrow \quad n_{Mis} \cdot \delta - n_{Mis} \cdot \delta \cdot \alpha = n_{Mis} - n_{\mathtt{B}} \end{aligned}$$

$$\begin{aligned} &\Leftrightarrow \quad n_{Mis} \cdot \delta - (n_{Mis} - n_{\mathtt{A}}) = n_{Mis} - n_{\mathtt{B}} \\ &\Leftrightarrow \quad n_{Mis} \cdot \delta - 2n_{Mis} = -n_{\mathtt{A}} - n_{\mathtt{B}}) \\ &\Leftrightarrow \quad n_{Mis} = \frac{n_{\mathtt{A}} + n_{\mathtt{B}}}{2 - \delta} \end{aligned}$$

With $\delta_{exp} \approx \delta$ and accounting for integral n and imprecise δ_{exp}, we use:

$$\tilde{n} := \max\left(\left\lceil \frac{n_{\mathtt{A}} + n_{\mathtt{B}}}{2 - \delta_{exp}} \right\rceil, \underbrace{|n_{\mathtt{A}} - n_{\mathtt{B}}|}_{\text{minimum differences}}\right) \approx \lceil n_{Mis} \rceil = \lceil \max(n_{Mis}, n_{Old}) \rceil \geq n \quad (5.4)$$

Phase 1 Formulae Wrap-Up

Finally, we calculate an appropriate b_k to fulfil FR from eq. (5.3) using the lowest suitable integral value for b_k with $\tilde{n}$ from eq. (5.4) above:

$$b_k := \left\lceil \log_2 \frac{1}{1 - \sqrt[\tilde{n}-1]{1 - \frac{FR}{2\tilde{n}}}} \right\rceil \quad (5.5)$$

Floating Point Precision Calculating b_k

Since $z \coloneqq {}^{FR}/_{2\tilde{n}}$ is near 0, there is a problem in the floating point representation of $1 - z$ and $\sqrt[\tilde{n}-1]{1-z}$. Instead of implementing eq. (5.5) as given, for $z < 10^{-8}$, we use the fifth-order Taylor series expansion of $1 - \sqrt[x]{1-z}$, $x \coloneqq n - 1$ at $z = 0$ which eliminates the need for the incriminating terms:

$$\begin{aligned} 1 - \sqrt[x]{1-z} \approx z \cdot \frac{1}{x} \;+\; z^2 \cdot \frac{x-1}{2x^2} \;+\; z^3 \cdot \frac{2x^2 - 3x + 1}{6x^3} \\ + z^4 \cdot \frac{6x^3 - 11x^2 + 6x - 1}{24x^4} \\ + z^5 \cdot \frac{24x^4 - 50x^3 + 35x^2 - 10x + 1}{120x^5} \;+\; \mathcal{O}(z^6) \end{aligned} \tag{5.5a}$$

Similarly, an upper bound on the failure rate can be given from eq. (5.2) which is bound by FR with this b_k:

$$fr_t \leq 2 \cdot n \cdot \left(1 - (1 - {}^1/_{2^{b_k}})^{n-1}\right) \lessapprox \underbrace{2 \cdot \tilde{n} \cdot \left(1 - (1 - {}^1/_{2^{b_k}})^{\tilde{n}-1}\right)}_{=: fr'_t} \leq FR \tag{5.6}$$

Floating Point Precision Calculating fr'_t

For high b_k, the calculation of fr'_t suffers similar problems with floating point numbers near 1 as eq. (5.5) above. We use the same Taylor series expansion (eq. (5.5a)) but with $x \coloneqq {}^1/_{(\tilde{n}-1)}$ and $z \coloneqq {}^1/_{2^{b_k}}$ to circumvent them.

5.4.2 Overall Costs

The costs of both $\texttt{CKV}$ and $\texttt{CK}_{idx}$ without delta-encoding or zlib-compression can be easily derived from the calculations above. The total cost C_t of the *trivial* algorithm in bits where these techniques are applied, however, cannot be given in the general case and we thus establish an upper bound without delta-encoding or zlib-compression as:

$$C_t \leq \underbrace{\left|\texttt{CKV}(S_\texttt{B})\right|}_{\leq n_B \cdot (b_k + 32)} + \underbrace{\left|\texttt{CK}_{idx}(\Delta'_\texttt{A})\right|}_{\substack{\leq |\Delta'_\texttt{A}| \cdot s + t \\ \lessapprox |\Delta| \cdot \lceil \log_2 n \rceil + t}} \qquad \in \mathcal{O}\left(n \cdot \log \frac{\tilde{n}}{FR} + |\Delta| \cdot \log n\right) \tag{5.7}$$

$$\text{(for } n, \tilde{n}, |\Delta| \to \infty,\ FR \to 0\text{)}$$

Note that due to $\tilde{n} \leq 2n$ (worst-case in the *outdated* scenario with $\delta_{exp} = 100\,\%$ in eq. (5.4)) and also $\tilde{n} \leq n \cdot (1 + \delta_{exp})$ (easy to prove from eq. (5.4) with $n_{Y \in \{\texttt{A},\texttt{B}\}} \leq n$) $\tilde{n}$ can be replaced by n or $n \cdot (1 + \delta_{exp})$ inside the $\mathcal{O}$ notation.

Proving the Communication Complexity

From Lemma 2.1.2 (page 10) follows that $b_k \leq \log_2 \frac{\tilde{n}-1}{FR/2\tilde{n}}$ and therefore:

$$\begin{aligned}\texttt{CKV}(S_\texttt{B}) &\leq n_\texttt{B} \cdot (b_k + 32) \leq n \cdot (b_k + 32) \leq n \cdot \left(\log_2 \frac{\tilde{n}-1}{\frac{FR}{2\tilde{n}}} + 32\right) \\ &\leq n \cdot \left(\log_2 \frac{3\tilde{n}}{FR} + 32\right) \in \underset{\text{(for } n,\tilde{n}\to\infty,\ FR\to 0)}{\mathcal{O}}\left(n \cdot \log \frac{\tilde{n}}{FR}\right) \end{aligned} \tag{5.8}$$

Finally, eq. (5.7) follows with the bound on $\texttt{CK}_{idx}$ as indicated above:

$$\begin{aligned}\texttt{CK}_{idx} &\leq |\Delta'_\texttt{A}| \cdot s + t = |\Delta'_\texttt{A}| \cdot \lceil \log_2(max_{diff} + 1)\rceil + \lceil \log_2(\lceil \log_2(n'_\texttt{B})\rceil + 1)\rceil \\ &\leq \underbrace{|\Delta'|}_{\approx|\Delta|} \cdot \lceil \log_2 n\rceil + \lceil \log_2(\lceil \log_2 n\rceil + 1)\rceil \in \underset{\text{(for } n,|\Delta|\to\infty)}{\mathcal{O}}\left(|\Delta| \cdot \log n\right) \end{aligned} \tag{5.9}$$

5.5 Effective Worst-Case Accuracy

Since b_k from eq. (5.5) must be integral and fulfil FR, it is rounded up to the nearest integer which—compared to the theoretical (non-integral) optimum—may make the *trivial* algorithm more expensive and more accurate than required. With this b_k, however, we do achieve the highest possible accuracy below FR. The following two sections discuss the resulting differences and evaluate them for three different values of FR, i.e. 10, 0.1, and 0.001, by showing the actual b_k as well as the resulting worst-case failure rate fr'_t (eq. (5.6)) based on the information the algorithm has, i.e. $n_\texttt{A}$, $n_\texttt{B}$, δ_{exp} and FR.

5.5.1 Outdated Items Scenario

Table 5.2 shows the actual values of b_k and fr'_t for different parameters in the *outdated* scenario and verifies that the *trivial* reconciliation algorithm may be more accurate than the configured FR. The differences vary greatly and depend on how far fr'_t would be from FR for the next larger b_k. From the values presented here and the arguments below, we conclude that the *trivial* reconciliation is up to twice as accurate as intended. Please note, however, that the $\tilde{n}$ from eq. (5.4) used for the calculation of both b_k as well as fr'_t assumes the worst-case, i.e. all differences being missing items, and is thus too large for the *outdated* scenario. The actual *observed* failure rate may thus be even lower (ref. Section 5.6).

Figure 5.2 shows the effective failure rates as well as their overall minimum, maximum, and averages for different $n = n_\texttt{A} = n_\texttt{B}$, δ_{exp} and FR in the *outdated* scenario. The plot shows a repetitive pattern with fluctuations of the fr'_t values within an interval of roughly $[FR/2, FR]$. Although $\tilde{n}$ also influences the accuracy

Table 5.2: *Trivial* bit size b_k (eq. (5.5)) and effective worst-case failure rate fr'_t (eq. (5.6)) in the *outdated* scenario with $n = n_A = n_B = 100\,000$.

	$\delta_{exp} = 1\,\%$			$\delta_{exp} = 10\,\%$		
$\tilde{n}$	100 503	100 503	100 503	105 264	105 264	105 264
FR	0.0010000	0.10000	10.000	0.0010000	0.10000	10.000
b_k	45	38	31	45	38	32
fr'_t	0.0005742	0.07349	9.407	0.0006298	0.08062	5.160

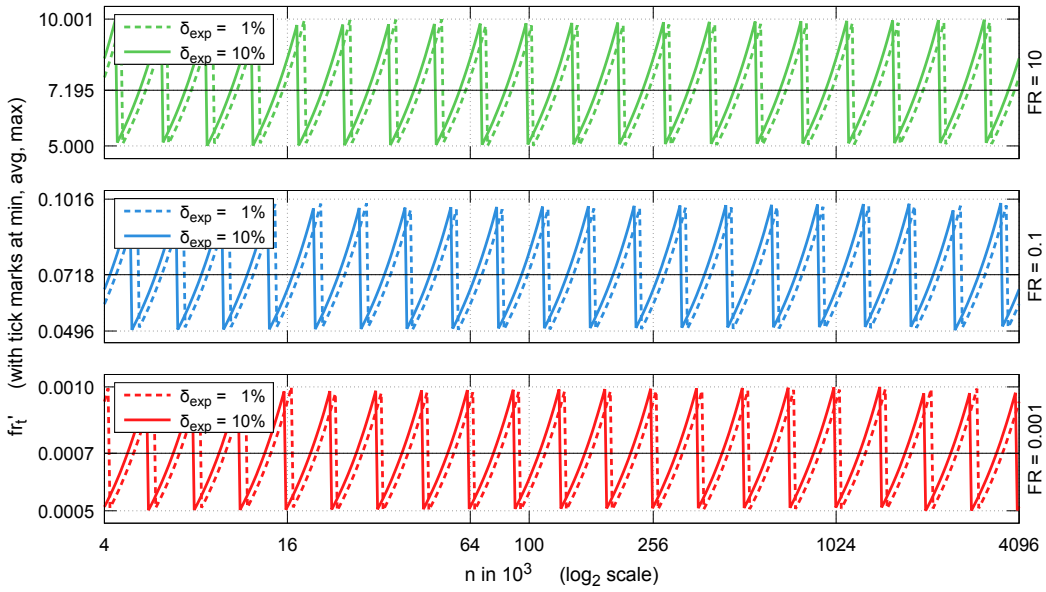


Figure 5.2: *Trivial* fr'_t in the *outdated* scenario for different FR, n, and δ_{exp}.

directly (eq. (5.6)), the main reason for the variation in fr'_t and a peak-to-peak amplitude of roughly $FR/2$ is the ceiling of b_k (eq. (5.5)) where one additional bit roughly halves the failure rate[c]. After a peak at $\tilde{n}_1$, fr'_t increases with $\tilde{n}$ towards FR until roughly $\sqrt{2} \cdot \tilde{n}_1$ where b_k grows again[d]. These peak values are not reached in every such iteration which is most likely due to floating point inaccuracies. Regarding the influence of δ_{exp}, we note that since it only influences $\tilde{n}$, it shifts the values along the x-axis.

Figure 5.2 also shows that the maximum may be slightly above FR and the minimum slightly below $FR/2$ which should both not happen. These are apparently caused by further floating point inaccuracies which are not the result of the Taylor series expansion of eq. (5.5a). We tolerate these deviations but

[c] This follows from using the first Taylor term of eq. (5.5a), i.e. z/x, for eq. (5.6) with $x := 1/(\tilde{n}-1)$ and $z := 1/2^{b_k}$ vs. $z := 1/2^{b_k+1}$.

[d] This follows similarly by using the first Taylor term of eq. (5.5a), i.e. z/x, for eq. (5.5) with $x := \tilde{n} - 1 \approx \tilde{n}$ and $z := FR/2\tilde{n}$ that leads to $z/x \approx FR/(2\tilde{n}^2)$ vs. $x := \sqrt{2} \cdot \tilde{n} - 1 \approx \sqrt{2} \cdot \tilde{n}$ and $z := FR/(2 \cdot \sqrt{2} \cdot \tilde{n})$ that leads to $z/x \approx FR/(4\tilde{n}^2)$ and thus a $\log_2 x/z$ that increases by 1.

note that we may reduce some by always selecting the highest integer *greater* than the calculated value of b_k instead of ceiling it in eq. (5.5).

5.5.2 Missing Items Scenario

Table 5.3 and Figure 5.3 show the results in the *missing* scenario for $n = 100\,000$ similar to the evaluation below, i.e. $\delta = \delta_{exp}$ failure items are distributed equally among the two nodes (vs. a uniform distribution in the evaluation). Therefore, instead of $\tilde{n}$, $n_{\mathtt{A}} = n_{\mathtt{B}}$ varies with δ_{exp} and thus $\tilde{n} = n$ which results in δ_{exp} not influencing b_k or fr'_t. Other than that, this scenario does not show any different results than the *outdated* scenario in Section 5.5.1 above.

Table 5.3: *Trivial* bit size b_k (eq. (5.5)) and effective worst-case failure rate fr'_t (eq. (5.6)) in the *missing* scenario with $n = 100\,000$ and $n_{\mathtt{A}} = n_{\mathtt{B}}$ accordingly.

	$\delta_{exp} = 1\,\%$			$\delta_{exp} = 10\,\%$		
$n_{\mathtt{A,B}}$	99 500	99 500	99 500	95 000	95 000	95 000
$\tilde{n}$	100 000	100 000	100 000	100 000	100 000	100 000
FR	0.0010000	0.10000	10.000	0.0010000	0.10000	10.000
b_k	45	38	31	45	38	31
fr'_t	0.0005684	0.07276	9.313	0.0005684	0.07276	9.313

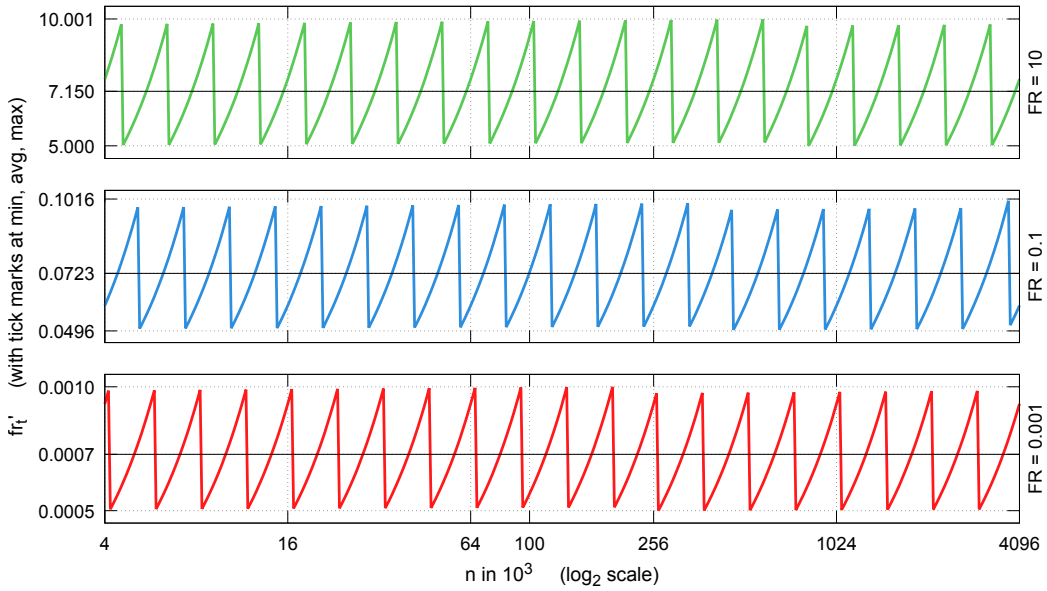


Figure 5.3: *Trivial* fr'_t in the *missing* scenario for different FR and n.

5.6 Evaluation

Based on the configured failure rate FR, the *trivial* reconciliation algorithm sets appropriate sizes for keys to exchange. The following sections evaluate the protocol under different aspects, starting with different δ and FR assuming correct $\delta_{exp} := \delta$. We then present the effects of incorrect $\delta_{exp} \neq \delta$ as well as different data and failure distributions. A scalability analysis for different n and FR each conclude this evaluation.

5.6.1 General Analysis for Different δ and FR

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand}$

Figures 5.4 and 5.5 show the actual failure rates of redundantly transferred items and missed differences as well as the transfer costs in our experimental evaluation for $\delta_{exp} := \delta \in (0, 10]\,\%$ and $\delta \in (0, 100]\,\%$, respectively. The average of the sum of the two failure types in the worst-case scenario should be equal to—or at least lower than—the configured FR. As shown, this bound holds. We do, however, observe that the *trivial* algorithm's accuracy may be higher than anticipated, partly because b_k needs to be integral (also ref. Section 5.5) but also due to eq. (5.4) (page 56) assuming the worst-case scenario of all differences being missing items. Especially in the *outdated* scenario, this

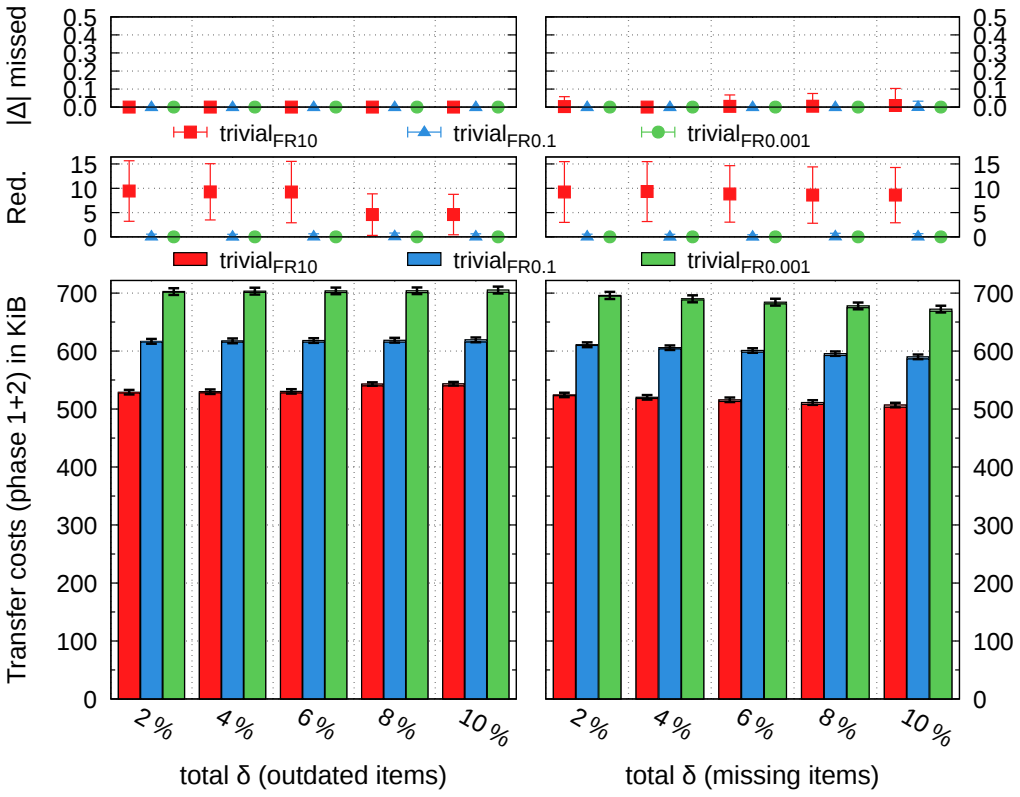


Figure 5.4: *Trivial* reconciliation with small δ and different FR.

leads to the *trivial* algorithm overestimating the number of different keys and thus an increasing accuracy, i.e. a lower failure rate, with increasing δ.

Note that unrecognised items of Δ are not present in the *outdated* scenario since only single collisions may cause them (ref. Section 5.3.1). If present in the *missing* scenario, however, the most likely result is only one unrecognised item of Δ as opposed to the two items accounted for. Additionally, recall that any missing item in a collision set with at least two items on any one node does not lead to a failure. Since the probability of this increases with δ, the number of redundant item transfers thus lowers in the *missing* scenario. Similarly, unrecognised items of Δ increase with the probability of a single collision.

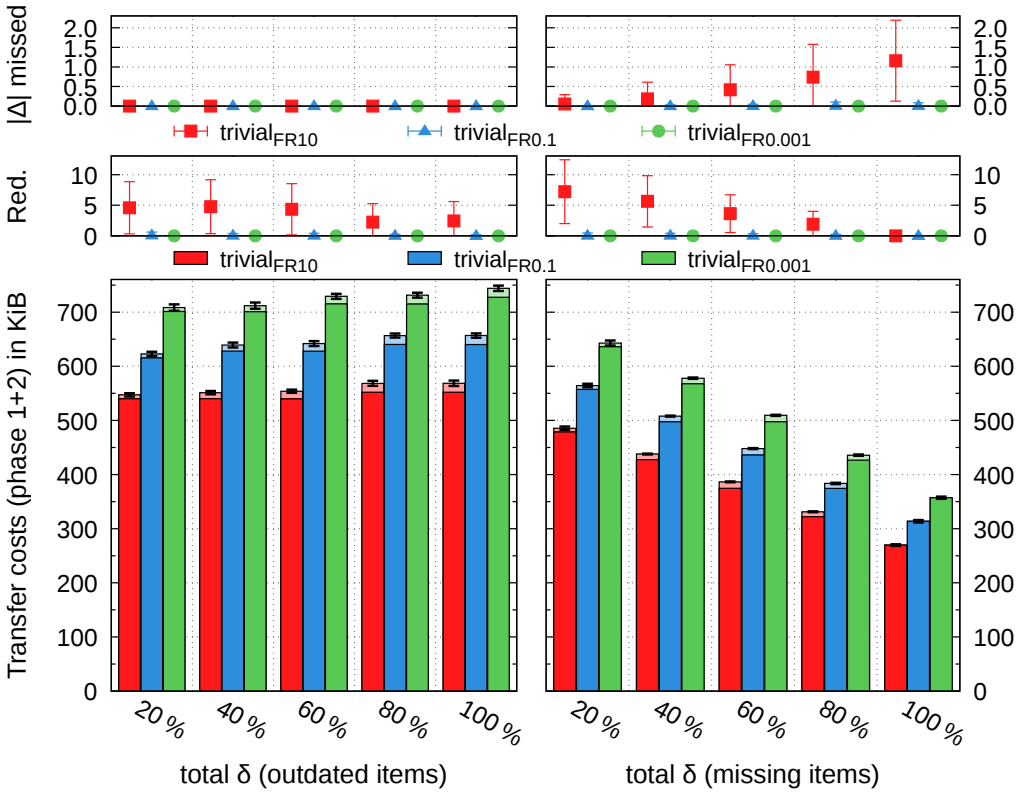


Figure 5.5: *Trivial* reconciliation with high δ and different *FR*.

Since the number of unique keys $\tilde{n}$ is estimated using δ_{exp}, b_k increases in steps with $\delta =: \delta_{exp}$ (ref. eq. (5.4), page 56) when this leads to the estimated $\tilde{n}$ increasing above $\sqrt{2} \cdot \tilde{n}$ (following the argumentation of Section 5.5.1). In the *outdated* scenario, for example, according to eq. (5.4), $\tilde{n}$ is 111 112 and 166 667 for $\delta_{exp} = 20\,\%$ and $80\,\%$, respectively. Depending on where b_k increased before, `CKV` costs (phase 1) thus increase at least once between these steps as is shown in the *outdated* scenario of Figure 5.5. Since in the *missing* scenario, each node's number of items decreases with higher δ, there, `CKV` costs decrease instead. Regarding $\texttt{CK}_{idx}$, the delta-encoded indices seem effective and keep phase 2 costs below 4 KiB and 17 KiB for $\delta \leq 10\,\%$ and $100\,\%$, respectively.

The given three values of FR all behave similarly here with differences in the failure rate and costs as expected. While the differences in the costs can already be estimated to increase logarithmically with FR^{-1} as expected (ref. eq. (5.7)), we show this fact in more detail in Section 5.6.5 below.

5.6.2 What if δ_{exp} is Wrong?

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$ vs. 1 %
$fail_{rand}$

For the *trivial* reconciliation protocol, δ_{exp} is used to estimate the (worst-case) number of unique keys $\tilde{n}$ in order to set b_k. This worst-case assumes that all differences are missing items and therefore, in the *outdated* scenario, where n is static, any δ_{exp} only results in a too high $\tilde{n} > n$: the higher δ_{exp}, the higher the accuracy and the transfer costs, irrespective of how close δ_{exp} is to δ. This effect can be observed in Figure 5.6 which shows the results of the *trivial* reconciliation with a perfect approximation of $\delta_{exp} = \delta$ and a wrong $\delta_{exp} = 1\,\%$ that violates our assumption of $\delta_{exp} \gtrapprox \delta$.

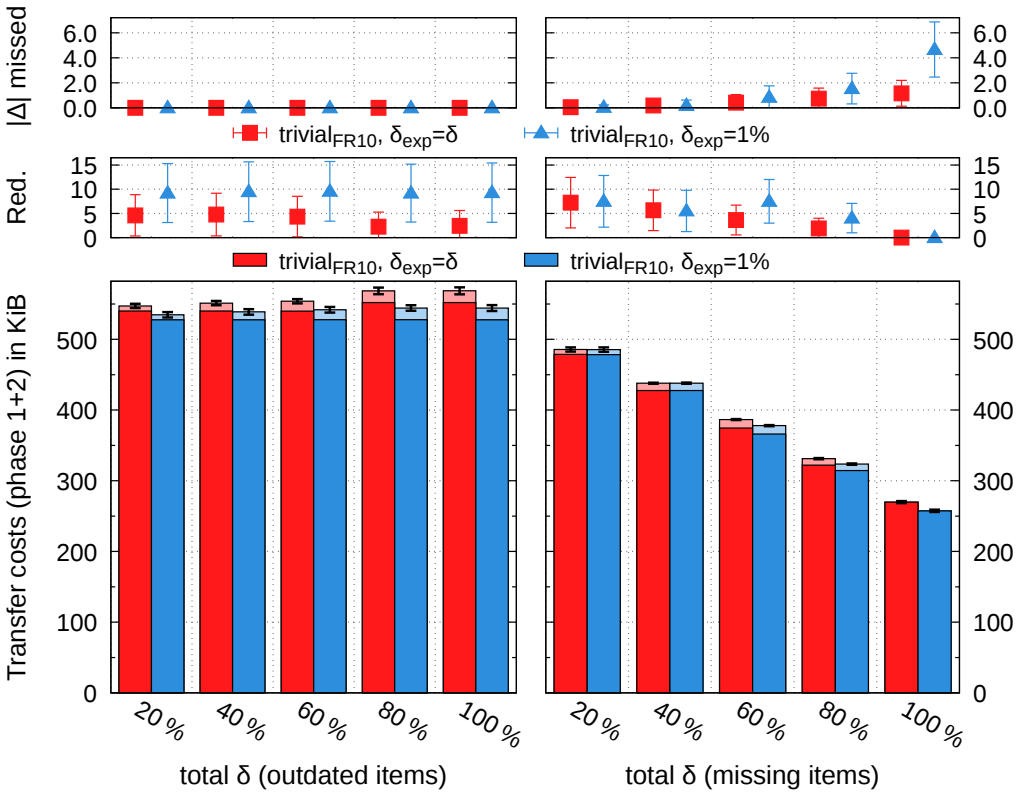


Figure 5.6: *Trivial* reconciliation with high δ and different δ_{exp}.

In the *missing* scenario, however, the number of errors increases the more the actual δ diverges from δ_{exp} since $fr_t \leq FR$ (eq. (5.6), page 57) does not hold anymore. Overall, the effects are still very limited even for $\delta_{exp} = 1\,\%$ and $\delta = 100\,\%$. Please note, however, that in this case with $n_A = n_B = 50\,000$,

$\tilde{n}$ is 100 000 for $\delta_{exp} = 100\,\%$ and 50 252 for $\delta_{exp} = 1\,\%$ and thus b_k is 31 and 29, respectively. The expected failure rate using the real $n = 100\,000$ roughly quadruples from 9.3 to 37.3 with these b_k (ref. eqs. (5.4) to (5.6)).

5.6.3 Data and Failure Distribution Sensitivity

$n = 100\,000$
$data_{rand,\ bin_{0.2}}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand,\ bin_{0.2}}$

Since the *trivial* reconciliation protocol encodes items in `CKV` using delta-encoding, it could be influenced by the data distribution. However, we additionally apply zlib compression which, together, results in only minor deviations due to the data distribution as shown by Figure 5.7. In fact, a few bytes may be saved when the compressed keys cluster which results in up to 0.6 % savings as shown by the blue bars. The failure distribution only influences the $\texttt{CK}_{idx}$ of phase 2 but these effects are also small to negligible.

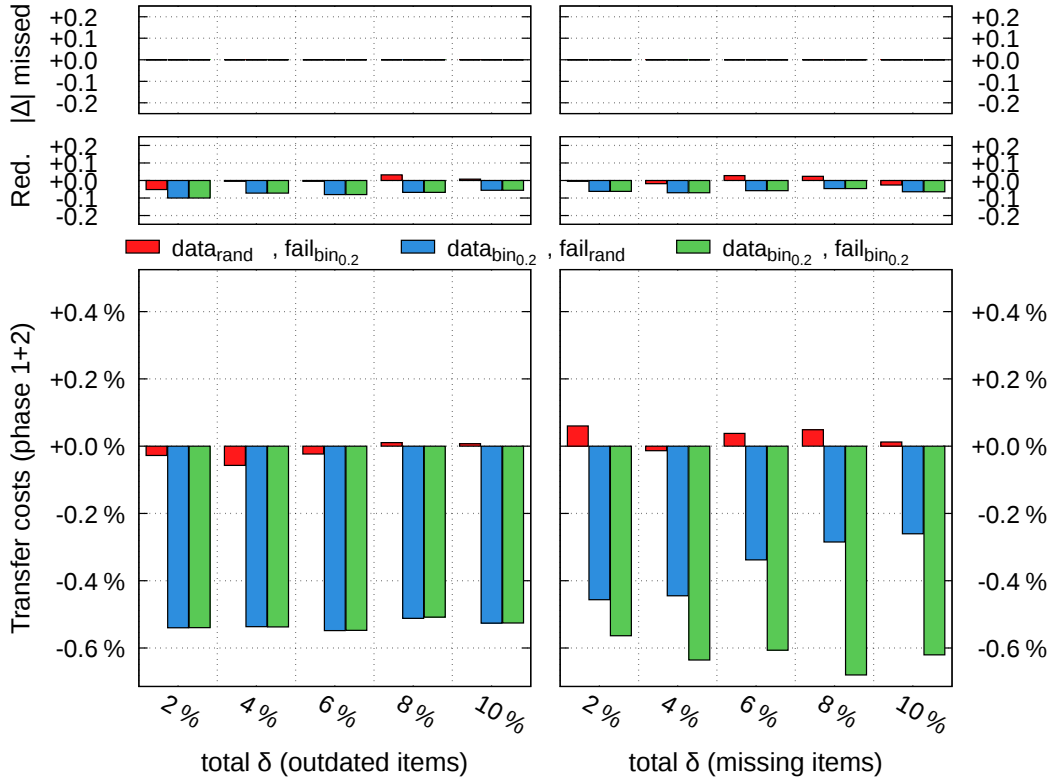


Figure 5.7: *Trivial* reconciliation with $FR = 0.1$ and different data and failure distributions compared to $data_{rand}, fail_{rand}$

5.6.4 Scalability with the System Size n

$n = variable$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 5.8 shows how the *trivial* reconciliation algorithm performs with smaller and larger numbers of items than the $n = 100\,000$ from above using a fixed $\delta = 3\,\% =: \delta_{exp}$. The two accuracy metrics evolve as expected. Regarding the transfer costs, we also plot an upper bound of the phase 1 costs of the $\mathcal{O}(n)$ naïve algorithm ($n \cdot (128 + 32)$ bits) for comparison. The results shown by Figure 5.8 support our theoretical $\mathcal{O}(n \cdot \log n)$ scalability for $n \to \infty$, $|\Delta| \in \mathcal{O}(n)$ and *FR* constant (ref. eq. (5.7) on page 57).

We also observe that the *trivial* transfer costs slowly approach the naïve transfer costs. Eventually, the number of bits needed to distinguish different items under *FR* is higher than the number of bits available for the keys. In that case, however, 128 bit keys may not be sufficient either to allow keys to be derived from a hash or even to distinguish this many items.

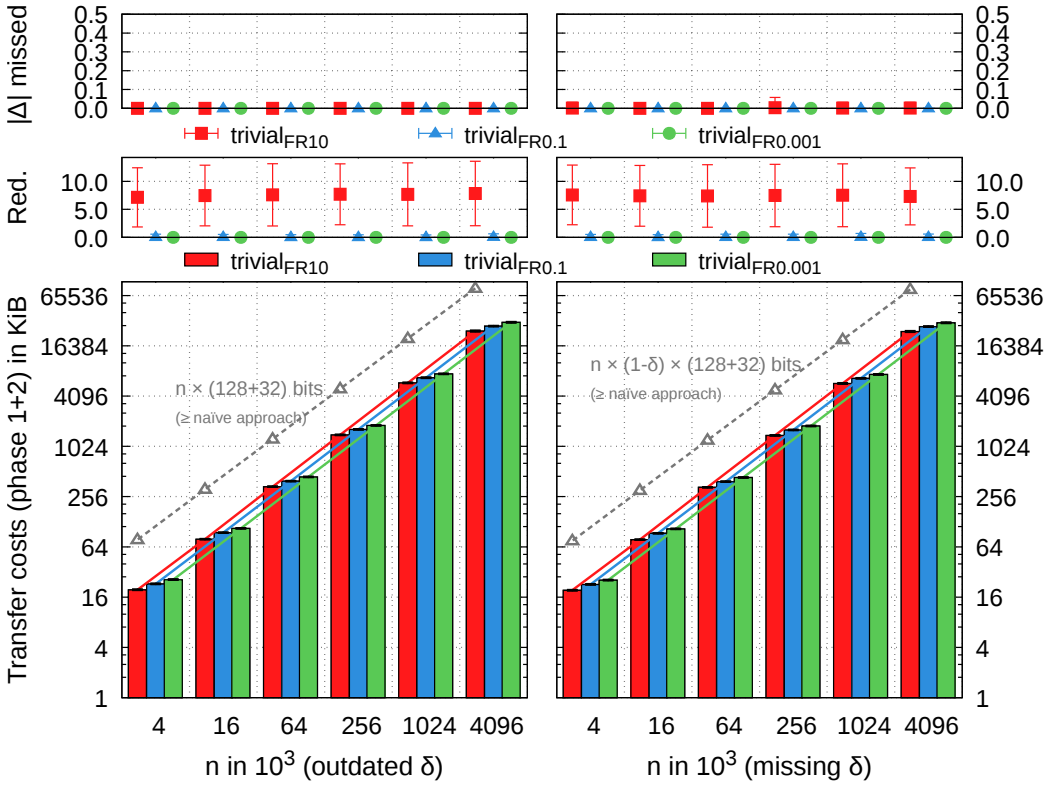


Figure 5.8: *Trivial* reconciliation scalability with data size n and different *FR*.

5.6.5 Scalability with the Target Failure Rate FR

$n = 100\,000$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 5.9 shows how the communication costs and the failures of the *trivial* reconciliation evolve with different target failure rates FR and constant n. While the smaller values for FR are not representative for counting failures with our simulation setup of 1 000 random simulations anymore, we can still derive the costs required for such a level of accuracy. Higher FR, however, evolve as expected. Regarding the costs, we observe a clear logarithmic increase with FR^{-1} as derived from eq. (5.7), page 57, and thus support the theoretical complexity of $\mathcal{O}(\log(1/FR))$ for n and $|\Delta|$ constant.

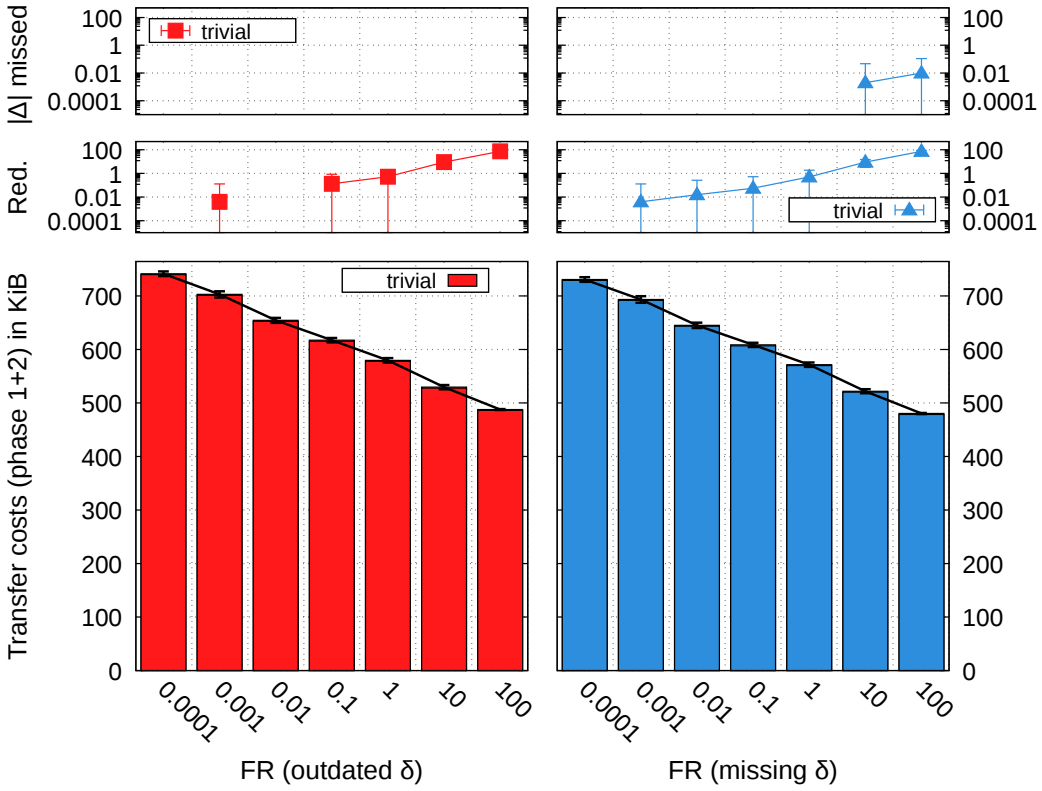


Figure 5.9: *Trivial* reconciliation scalability with the target failure rate FR ($\log_{10}$ scale on the x-axis and on the y-axis except for the transfer costs).

Chapter 6

SHash Reconciliation

By sending a 32-bit version number for each item at node B, the *trivial* algorithm creates a big overhead for items which are common between the two nodes. This is especially prominent if the number of differences is low. Therefore, we create a new algorithm, *SHash*, which hashes `{key, version}` *pairs* and does not send the version numbers at first.

6.1 Protocol

Similarly to the *trivial* reconciliation, at B we apply delta-encoding to a sorted hash list (ref. Section 5.2.1) but create hashes from the combined key and version. The result is the SH binary without explicit version numbers which is sent to A together with a few configuration parameters (Figure 6.1). In this first phase, SHash identifies $S'_A \supseteq \Delta'_B \cup Old'_A, S'_A \subseteq S_A$ on A by checking whether A's items are present in SH. Additionally, all unmatched hashes from SH are collected which represent $S'_B \supseteq \Delta'_A \cup Old'_B, S'_B \subseteq S_B$ and are sent to B using the delta-encoded CK_{idx} scheme from Section 5.3.2. *Phase 2* then executes a slightly modified *trivial* reconciliation, i.e. *trivial'*, on these sets to identify separate Δ'_B and Δ'_A. Please note that $CK_{idx}(S'_B)$ and $CKV(S'_A)$ overlap in the keys of outdated items at A or B but they are essential in CKV and we cannot remove them from CK_{idx} without further information.

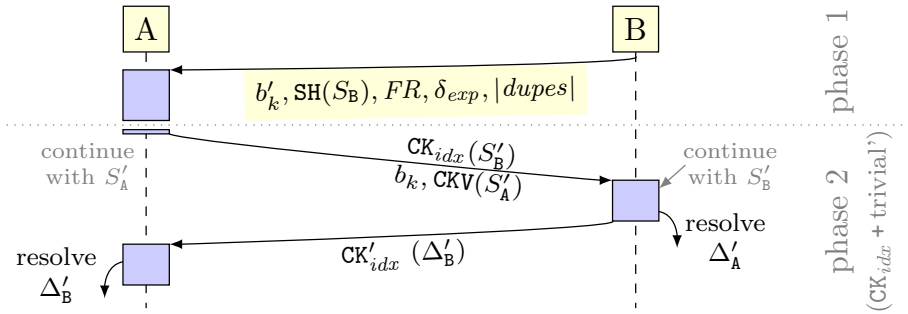


Figure 6.1: SHash reconciliation protocol.

The changes in the *trivial'* algorithm compared to the original algorithm from Chapter 5 are limited to a different representation of the keys to request, i.e. $\texttt{CK}'_{idx}$. Since we expect a request for all $|\Delta'_{\texttt{B}}|$ items of the items in $\texttt{CKV}(S'_{\texttt{A}})$ and $|S'_{\texttt{A}}| \gtrapprox |\Delta'_{\texttt{B}} \cup Old'_{\texttt{A}}| = |\Delta'_{\texttt{B}}| + |Old'_{\texttt{A}}|$, it is more efficient to use a simple binary for the representation of their positions where the bit at position `x` denotes whether the item at position `x` is requested (1) or not (0).

6.2 SH Data Structure Details

In contrast to the `CKV` structure of the *trivial* algorithm (ref. Section 5.2), item hashes are built on the concatenation of an item's key *and* version, i.e. $\ll i_k, i_v \gg$. The rest of the creation of `SH`, however, is similar (Algorithm 10): each $\ll i_k, i_v \gg$ is hashed to b'_k bits by using the least significant bits of an MD5 hash, the list of these hashes is sorted, and items with non-unique hashes are removed and added to phase 2, i.e. $S'_{\texttt{B}}$. Furthermore, the delta-encoding of Section 5.2.1 is applied and the result constitutes the `SH` binary which, together with the number of non-unique i_h ($|dupes|$ in Figure 6.1), makes $n_{\texttt{B}}$ known to `A`.

Algorithm 10 `SH` creation

```
function SH(Entries)
    hashes ← []                                   ▷ empty list for the hashes
    for all i ∈ Entries do                        ▷ hash all items' keys and versions
        i_h ← MD5(«i_k, i_v», b'_k)  ▷ use the b'_k least significant bits of an MD5 hash
        hashes.APPEND(i_h)
    end for
    SORT(hashes)                                  ▷ sort by item hashes i_h
    S'_B ← REMOVEDUPES(hashes)                    ▷ remove non-unique i_h, add to phase 2
    return DELTAENCODE(hashes, b'_k)              ▷ Delta-Encoding from Section 5.2.1
end function
```

6.3 Using SH for Set Reconciliation

Similarly to the *trivial* reconciliation, each item in $S_{\texttt{A}}$ is hashed the same way as those in `SH` and compared with each of the b'_k-bit hashes in `SH` (Algorithm 11). In contrast, if a match is found the item is assumed to exist on `B` with the same version and is thus ignored. If no match is found, the item from $S_{\texttt{A}}$ is added to $S'_{\texttt{A}}$, i.e. the set of differences phase 2 works on. Please recall that we do not have any version numbers at hand to decide whether `A` or `B` has the newer version of an item and thus a second phase is necessary. It would not be required if only *missing* items occurred and this was known a-priori.

At the end, after all items from $S_{\texttt{A}}$ have been checked, all indices of unmatched hashes in `SH` are added to $S'_{\texttt{B}}$ in order to be able to identify $Mis'_{\texttt{A}}$. In contrast to the *trivial* reconciliation, however, unmatched hashes also originate from

Algorithm 11 SHash reconciliation at node A (Phase 1)

```
function SHashSync(S_A, CKV, b'_k)        ▷ after receiving b'_k and SH (S_B) from B
    hashes ← DeltaDecode(SH, b'_k)        ▷ revert the Delta-Encoding from above
    for all i ∈ S_A do                                          ▷ hash all items' keys
        i_h ← MD5(«i_k, i_v», b'_k)  ▷ use the b'_k least significant bits of an MD5 hash
    end for
    S'_A ← RemoveDupes(S_A)              ▷ remove non-unique i_h, add them to S'_A
    for all i ∈ S_A do                                    ▷ for all remaining items of S_A
        if ¬ hashes.Remove(i_h) then          ▷ find and remove i_h from hashes
            S'_A.Append(i)                ▷ not present in SH ⇒ put i into phase 2
        end if
    end for
    S'_B.AppendAll(hashes)                ▷ request all items from unmatched hashes
end function
```

outdated items. Some of these could be removed by also checking an item with version numbers in the vicinity of the item's version at A but we do not employ such a technique since this is not universal to all scenarios and only practical for small version differences.

6.3.1 Phase 2 Details – $\mathtt{CK}_{idx}$ and *trivial'*

After the first phase, node A continues to work with items in S'_A. Similarly, upon receiving the $\mathtt{CK}_{idx}$ structure (ref. Figure 6.1), node B continues with items in S'_B which were encoded in $\mathtt{CK}_{idx}$. This encoding was already described in the *trivial* reconciliation and we refer to Section 5.3.2 for details.

Algorithm 12 $\mathtt{CK}'_{idx}$ creation

```
function CK'_idx(hashes, Δ'_B)   ▷ hashes from CKV, Δ'_B with item hashes to request
    CK'_idx ← empty bitstring                                  ▷ start an empty binary
    for all i_h ∈ hashes do                                    ▷ traverse hashes in order
        if i_h ∈ Δ'_B then
            CK'_idx.AddBits(1)                          ▷ add 1-bit for items to request
        else
            CK'_idx.AddBits(0)                          ▷ add 0-bit for items to ignore
        end if
    end for                                  ▷ note: every i_h ∈ Δ'_B exists in hashes
    return CK'_idx.Truncate( )                  ▷ remove trailing zero bits, return
end function
```

With S'_A on A and S'_B on B, we execute a *trivial'* set reconciliation which only differs from the *trivial* set reconciliation by the returned $\mathtt{CK}'_{idx}$ which does not use delta-encoded indices but lets each bit determine whether the item at its position is requested (1) or not (0) (Algorithm 12). Afterwards, we truncate trailing 0 bits and obtain $\mathtt{CK}'_{idx}$ which will be zlib-compressed during transfer.

6.3.2 Implications of Hash Collisions

Phase 1

As in the *trivial* set reconciliation, hashes of different items from $S_\mathtt{B}$ may collide, or hashes of different items from $S_\mathtt{A}$, or hashes of items which are different in $S_\mathtt{A}$ and $S_\mathtt{B}$. In contrast, since version numbers are included in the hashes, overall $n^* := |S_\mathtt{A}| + |Mis_\mathtt{A} \cup New_\mathtt{B} \cup Old_\mathtt{B}| = |S_\mathtt{A}| + |\Delta \setminus Mis_\mathtt{B}|$ hashes may collide, with every outdated item counting twice: once for each version. For each hash collision among items whose different $\langle\!\langle i_k, i_v \rangle\!\rangle$ and $\langle\!\langle j_k, j_v \rangle\!\rangle$ are hashed to the same value h, i.e. from $X_h := \{i \in S_\mathtt{A} \cup S_\mathtt{B} : i_h = h\}$ with $|X_h| > 1$, A needs to put $X_h \cap S_\mathtt{B}$ and $X_h \cap S_\mathtt{A}$ into $S'_\mathtt{B}$ and $S'_\mathtt{A}$ for phase 2 on node B and A, respectively, in order to allow phase 2 to identify all items of Δ. This follows from the worst-case scenario that $X_h \cap S_\mathtt{B} \subseteq \Delta_\mathtt{A}$ and $X_h \cap S_\mathtt{A} \subseteq \Delta_\mathtt{B}$.

With this, phase 1 only creates errors for the overall reconciliation if it fails to identify items from Δ or puts a common item of $S_\mathtt{A} \cap S_\mathtt{B}$ into only one node's phase 2. Such errors only originate from the following hash collisions (summarised by Table 6.1 and similar to the *trivial* algorithm in Section 5.3.1):

- *a single collision of $x \in S_A$ with $y \in S_B, y \neq x$, i.e. $X_h = \{x, y\}$*: We cannot distinguish this from a valid match with $y = x$ and thus assume that $x \in S_\mathtt{B}$. As a result, we fail to identify an item of Δ for each $i \in X_h$ that is a missing item. Each outdated/newer item $i \in X_h$ for which its counterpart is in a single collision, too, also becomes an unidentified item. Otherwise if i is the newer item, only the outdated item remains in phase 2 and thus becomes a redundant item transfer with the feedback message eventually updating it (ref. Section 3.1).

- *multiple collisions in X_h with at least two items in S_A or in S_B*: Items in X_h can be common items or actual differences and $X_h \cap S_\mathtt{B}$ and $X_h \cap S_\mathtt{A}$ are put into $S'_\mathtt{B}$ and $S'_\mathtt{A}$, respectively. Therefore, both types go into phase 2 which may have more items to process but does not miss any of them.

Contrary to the *trivial* reconciliation, the total number of errors caused by the first phase is thus bound by twice the expected number of single collisions with one item from $S_{\mathtt{A}_\Delta}$ and $S_{\mathtt{B}_\Delta}$ each and no other collision. Even lower bounds may be given only with further assumptions on the failure distribution.

Table 6.1: SHash reconciliation errors for hash value h with collision set X_h.

Failure Case	Result (after phase 2)
$X_h = \{x \in Mis_\mathtt{B}, y \in Mis_\mathtt{A}\}$	two unrecognised items of Δ
$X_h = \{x \in S_\mathtt{A}, y \in S_\mathtt{B}\}, x_k = y_k, x_v \neq y_v$	one unrecognised item of Δ
$X_h = \{x \in S_\mathtt{A}, y \in S_\mathtt{B}\}$ other cases	up to two failures (unrecognised items of Δ or redundant transfers)
$\lvert X_h \cap S_\mathtt{A}\rvert \geq 2$ or $\lvert X_h \cap S_\mathtt{B}\rvert \geq 2$	no errors

Similarly to the *trivial* reconciliation, we cannot detect the single collision case but detect multiple collisions on at least one node. If $|X_h \cap S_{\mathtt{B}}| \geq 2$, we remove all items of $X_h \cap S_{\mathtt{B}}$ from `SH` and add them to phase 2, i.e. $S'_{\mathtt{B}}$. Therefore, items from $S_{\mathtt{A}}$ with the same collision, i.e. $X_h \cap S_{\mathtt{A}}$, are not found in `SH` and added to phase 2, i.e. $S'_{\mathtt{A}}$. Analogously, if $|X_h \cap S_{\mathtt{A}}| \geq 2$, all items of $X_h \cap S_{\mathtt{A}}$ are added to $S'_{\mathtt{A}}$ and not matched with `SH`. Thus, any item in $X_h \cap S_{\mathtt{B}}$ is unmatched and added to $S'_{\mathtt{B}}$. Since these sets may also contain common items, phase 2 may work on more items than the differences alone which makes phase 1 beneficial only if it removes enough items from $S_{\mathtt{A}}$ and $S_{\mathtt{B}}$. Fortunately, b'_k influences both types of collisions similarly and—since single collisions cause failures in the reconciliation—is set appropriately to fulfil a failure rate FR (see below).

Phase 2

Since the `CKV` matching of phase 2 is identical to the original *trivial* set reconciliation, hash collisions in `CKV` have identical implications (ref. Section 5.3.1). The only difference is the set they work on and this is influenced by the actual differences as well as the hash collisions in the `SH` binary of phase 1.

6.4 Deducing SHash Parameters from FR

SHash reconciliation consists of two phases. Since each of these phases is approximate, both may cause failures and we need to set individual accuracy targets for each of them in order to fulfil a global FR. Due to the linearity of the expected number of failures (ref. Section 2.5.1), we can use $FR/2$ in each of the two phases in order to distribute the failure rate equally among them:

$$\underbrace{FR(p_1)}_{\text{target failure rate phase 1}} \coloneqq \frac{FR}{2} \geq \underbrace{fr\left(p_{X\in\{1,2\}}\right)}_{\text{actual failure rate phase } X} \quad \Rightarrow \quad fr = fr(p_1) + fr(p_2) \leq FR \tag{6.1}$$

This will be the accuracy target of phase 1 at whose end we will calculate its actual failure rate and use the remaining one for phase 2.

> ☞ **Splitting** FR
>
> Any process fulfilling FR can be split into x equally accurate (not necessarily independent) sub-phases, each using the same $FR(p_X) \coloneqq FR/x$ target failure rate due to its linearity (ref. Section 2.5.1):
>
> $$\overbrace{fr(p_i)}^{\text{actual failure rate of phase } i} \leq \frac{FR}{x} =: FR(p_X) \qquad \forall\, 1 \leq i \leq x$$
>
> $$\Rightarrow \quad fr = \sum_{i=1}^{x} fr(p_i) \leq \sum_{i=1}^{x} FR(p_X) = \sum_{i=1}^{x} \frac{FR}{x} = FR \tag{6.2}$$

Note that different distribution schemes may be preferred in some scenarios. Finding an optimal algorithm-specific and potentially scenario-dependent distribution of *FR*, however, is an interesting problem for future work.

6.4.1 Phase 1

As established in Section 6.3.2, the *failure rate $fr_s(p_1)$ of phase 1 of the SHash algorithm* is bound by $2 \cdot \mathrm{E}[\text{single collisions}]$ with a "single collision" being a colliding items set with one item from $S_{\mathtt{A}_\Delta}$ and $S_{\mathtt{B}_\Delta}$ each and no other collision. Similarly to the *trivial* reconciliation in Section 5.4.1, we use the linearity of the expected value with (non-independent) random variables representing one item's *single collision* probability to calculate the expected total number of single collisions. For this, let $n_{\mathtt{A}} \coloneqq |S_{\mathtt{A}}|$, $n_{\mathtt{A}_\Delta} \coloneqq |S_{\mathtt{A}_\Delta}|$, and $n_{\mathtt{B}_\Delta} \coloneqq |S_{\mathtt{B}_\Delta}|$:

$$\mathrm{E}[\text{single collisions}] = \underbrace{n_{\mathtt{A}_\Delta}}_{\substack{\text{items of } S_{\mathtt{A}} \\ \text{which may collide}}} \cdot \Bigg(\underbrace{\overbrace{\binom{n_{\mathtt{B}_\Delta}}{1}}^{\substack{\text{choices for the} \\ \text{colliding item}}} \cdot \underbrace{\frac{1}{2^{b'_k}}}_{\substack{\text{collision with one} \\ \text{of the } n_{\mathtt{B}_\Delta} \text{ items}}} \cdot \overbrace{\left(1 - \frac{1}{2^{b'_k}}\right)^{n_{\mathtt{B}_\Delta}-1}}^{\substack{\text{no collision with} \\ n_{\mathtt{B}_\Delta}-1 \text{ items}}} \cdot \underbrace{\left(1 - \frac{1}{2^{b'_k}}\right)^{n_{\mathtt{A}}-1}}_{\substack{\text{no collision with} \\ n_{\mathtt{A}}-1 \text{ items}}}}_{\substack{\text{probability of a "single collision" for a given hash of } S_{\mathtt{A}_\Delta} \subseteq S_{\mathtt{A}} \\ = \mathrm{E}[\text{"single collision" for a given hash of } S_{\mathtt{A}_\Delta}]}} \Bigg) \tag{6.3}$$

$$= \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{2^{b'_k}} \cdot \left(1 - \frac{1}{2^{b'_k}}\right)^{n_{\mathtt{B}_\Delta}+n_{\mathtt{A}}-2}$$

Please note that E[single collisions] is symmetric in the two nodes since $n_{\mathtt{A}} = |S_{\mathtt{A}} \cap S_{\mathtt{B}}| + n_{\mathtt{A}_\Delta}$ and thus $n_{\mathtt{B}_\Delta} + n_{\mathtt{A}} = |S_{\mathtt{A}} \cap S_{\mathtt{B}}| + n_{\mathtt{A}_\Delta} + n_{\mathtt{B}_\Delta}$.

Lemma 6.4.1. $(1+x)^n \leq \dfrac{1}{1-nx}$ *for* $x \in \mathbb{R}, -1 \leq x \leq 0$ *and* $n \in \mathbb{N}$

Proof (via induction). For $n = 0$, this inequality follows immediately. For $n = i + 1 > 0$, we assume $(1+x)^i \leq 1/(1-ix)$ from $n = i$ and prove that the inequality holds:

$$\begin{aligned}
& & (1+x)^i &\leq \frac{1}{1-ix} & & \\
\Leftrightarrow\ & (1+x)\cdot(1+x)^i &\leq (1+x)\cdot\frac{1}{1-ix} & & & \text{for } x \geq -1 \\
\Leftrightarrow\ & (1+x)^{i+1} &\leq (1+x)\cdot\frac{1}{1-ix} &\overset{?}{\leq} \frac{1}{1-(i+1)\cdot x} & \\
\Leftrightarrow\ & & (1+x)\cdot(1-ix-x) &\overset{?}{\leq} 1-ix & & \text{for } x \leq 0 \leq i \\
\Leftrightarrow\ & & -x^2\cdot(i+1) &\overset{?}{\leq} 0 & & \checkmark \text{ (for } i \geq 0)
\end{aligned}$$

□

With the help of Lemma 6.4.1, we calculate an appropriate b'_k so that the failure rate $fr_s(p_1)$ fulfils $FR(p_1)$, i.e. $fr_s(p_1) \leq FR(p_1)$:

$$fr_s(p_1) \leq 2 \cdot \mathrm{E}[\text{single collisions}] = 2 \cdot \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{2^{b'_k}} \cdot \left(1 - \frac{1}{2^{b'_k}}\right)^{n_{\mathtt{B}_\Delta}+n_{\mathtt{A}}-2} \tag{6.4}$$

$$\leq 2 \cdot \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{2^{b'_k}} \cdot \frac{1}{1 + \frac{(n_{\mathtt{B}_\Delta}+n_{\mathtt{A}}-2)}{2^{b'_k}}} \leq FR(p_1)$$

$$\Leftarrow \quad 2 \cdot \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{2^{b'_k}} \cdot \frac{2^{b'_k}}{2^{b'_k} + n_{\mathtt{B}_\Delta} + n_{\mathtt{A}} - 2} \leq FR(p_1)$$

$$\Leftrightarrow \quad 2 \cdot \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{FR(p_1)} \leq 2^{b'_k} + n_{\mathtt{B}_\Delta} + n_{\mathtt{A}} - 2$$

$$\Leftarrow \quad \log_2 \Bigg(\underbrace{2 \cdot \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{FR(p_1)} - n_{\mathtt{B}_\Delta} - n_{\mathtt{A}} + 2}_{=:x} \Bigg) \leq b'_k \qquad \text{if } x > 0$$

In case $x \leq 0$, any b'_k fulfils eq. (6.4) and we choose the lowest one, i.e. $b'_k = 1$:

$$\Leftarrow \quad \log_2 \left(\max \left(2 \cdot \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{FR(p_1)} - n_{\mathtt{B}_\Delta} - n_{\mathtt{A}} + 2,\ 2 \right) \right) \leq b'_k \tag{6.5}$$

An Upper Bound on the Size of the Collision Sets

In order to determine b'_k from eq. (6.5), we need the values of $n_{\mathtt{A}}$, $n_{\mathtt{A}_\Delta}$, and $n_{\mathtt{B}_\Delta}$. While $n_{\mathtt{A}}$ is known to B from the initial handshake (ref. Chapter 3), $n_{\mathtt{A}_\Delta}$ and $n_{\mathtt{B}_\Delta}$ are unknown but can be estimated using δ_{exp}. We assume the worst-case that $Mis_{\mathtt{A}}$ and $Mis_{\mathtt{B}}$ are empty and thus $n_{\mathtt{A}_\Delta} = n_{\mathtt{B}_\Delta} = n \cdot \delta$ which we estimate by using the upper bound $\tilde{n}$ on the number of unique keys (eq. (5.4), page 56):

$$\tilde{n}_{X_\Delta} \coloneqq \min \left(\left\lceil \delta_{exp} \cdot \max \left(\left\lceil \frac{n_{\mathtt{A}} + n_{\mathtt{B}}}{2 - \delta_{exp}} \right\rceil, |n_{\mathtt{A}} - n_{\mathtt{B}}| \right) \right\rceil, n_X \right) \tag{6.6}$$

$$= \min \left(\lceil \delta_{exp} \cdot \tilde{n} \rceil, n_X \right) \gtrapprox \min \left(\lceil \delta \cdot n \rceil, n_X \right) \geq n_{X_\Delta} \quad X \in \{\mathtt{A}, \mathtt{B}\}$$

Note that we use the upper bound of at most n_X items since $\tilde{n}$ uses a different—and contradicting—worst-case assumption and δ_{exp} may be incorrect.

Phase 1 Formulae Wrap-Up

A suitable b'_k which fulfils FR can be given from eq. (6.5) using the lowest possible integral value for b'_k with $\tilde{n}_{\mathtt{A}_\Delta}$ and $\tilde{n}_{\mathtt{B}_\Delta}$ from eq. (6.6):

$$b'_k \coloneqq \left\lceil \log_2 \left(\max \left(\frac{2 \cdot \tilde{n}_{\mathtt{A}_\Delta} \cdot \tilde{n}_{\mathtt{B}_\Delta}}{FR(p_1)} - \tilde{n}_{\mathtt{B}_\Delta} - n_{\mathtt{A}} + 2,\ 2 \right) \right) \right\rceil \tag{6.7}$$

$$= \left\lceil \log_2 \left(\max \left(\frac{4 \cdot \tilde{n}_{\mathtt{A}_\Delta} \cdot \tilde{n}_{\mathtt{B}_\Delta}}{FR} - \tilde{n}_{\mathtt{B}_\Delta} - n_{\mathtt{A}} + 2,\ 2 \right) \right) \right\rceil$$

An upper bound on the failure rate of phase 1 can be given similarly (ref. eq. (6.4)) and with this b'_k it is bound by $FR(p_1)$ as desired:

$$fr_s(p_1) \leq \frac{n_{\mathtt{A}_\Delta} \cdot n_{\mathtt{B}_\Delta}}{2^{b'_k-1}} \cdot \left(1 - \frac{1}{2^{b'_k}}\right)^{n_{\mathtt{B}_\Delta}+n_{\mathtt{A}}-2} \lessapprox \underbrace{\frac{\tilde{n}_{\mathtt{A}_\Delta} \cdot \tilde{n}_{\mathtt{B}_\Delta}}{2^{b'_k-1}} \cdot \left(1 - \frac{1}{2^{b'_k}}\right)^{\tilde{n}_{\mathtt{B}_\Delta}+n_{\mathtt{A}}-2}}_{=: fr'_s(p_1) \leq FR(p_1)} \tag{6.8}$$

Floating Point Precision Calculating $fr'_s(p_1)$

For high b'_k, $z := 2^{-b'_k}$ is small and there may be problems in the floating point representation of $1-z$ although $(1-z)^x$ could be suitably represented for $x := \tilde{n}_{\mathtt{B}_\Delta} + n_{\mathtt{A}} - 2$. Therefore, we use the $\mathtt{ln1p} := \ln(1+x)$ function by Goldberg [32] (Theorem 4) and calculate the following algebraic equivalent:

$$(1-z)^x = e^{\ln(1-z)^x} = e^{x \cdot \ln(1-z)} = e^{x \cdot \mathtt{ln1p}(-z)} \tag{6.8a}$$

6.4.2 Phase 2

Phase 2 consists of two combined parts: (a) sending $\mathtt{CK}_{idx}$ and (b) running a *trivial'* reconciliation. Since sending $\mathtt{CK}_{idx}$ is an exact step, we use the remaining failure rate for the identification of the resolve sets with *trivial'*. Due to the rounding of b'_k in phase 1, though, the actual worst-case failure rate $fr'_s(p_1)$ of phase 1 (eq. (6.8)) may not be as close to $FR(p_1)$ as intended. We thus calculate the remaining failure rate that phase 2 can use based on eq. (6.1):

$$fr_s \leq fr'_s(p_1) + fr_s(p_2) \leq FR \quad \Leftrightarrow \quad fr_s(p_2) \leq FR - fr'_s(p_1) =: \underbrace{FR(p_2)}_{\text{accuracy target of phase 2}} \tag{6.9}$$

Splitting FR iteratively

Any process fulfilling FR can be split iteratively into x *almost* equally accurate (not necessarily independent) sub-phases using the most of the available FR. For this, we re-distribute the difference of the target failure rate $FR(p_i)$ and the effective worst-case failure rate $fr'(p_i)$ of sub-phase i to the next sub-phases based on eq. (6.2). Thus, for $i = 1$:

$$fr(p_1) \lessapprox fr'(p_1) \leq \underbrace{FR(p_{i=1})}_{\text{target failure rate of phase } i} := \frac{FR}{x} \qquad \underbrace{fr(prev_{i=1})}_{\text{failure rate of } \textit{all} \text{ processes up to } i \text{ (inclusive)}} := fr'(p_1)$$

For $i > 1$, we target all remaining sub-phases at equal failure rates:

$$FR(p_i) := \frac{FR - fr(prev_{i-1})}{x - i + 1} \qquad fr(prev_i) := fr(prev_{i-1}) + fr'(p_i)) \tag{6.10}$$

$$\Rightarrow fr = \sum_{j=1}^{x} fr(p_j) \leq \underbrace{fr(prev_{i-1})}_{\text{sub-phases } 1,\ldots,i-1} + \underbrace{(x-i+1)\cdot FR(p_i)}_{\text{sub-phases } i,\ldots,x} \leq FR$$

It is obvious that the resulting overall failure rate fr is bound by FR and is closer to FR than by using x equally accurate sub-phases with target failure rates FR/x. Note, however, that we are not able to compensate the last sub-phase's difference in its target and effective worst-case failure rate.

Floating Point Precision Splitting FR iteratively

If FR and $fr(prev_{i-1})$ differ by several orders of magnitude their floating point difference in eq. (6.10) may become FR, especially for the first few sub-phases. Since $\frac{FR - fr(prev_{i-1})}{x-i+1} \leq \frac{FR}{x-i+1}$ for all $i > 1$, however, in these cases the target failure rate is too high but will eventually be compensated as $fr(prev_i)$ grows towards FR. For $fr(prev_i)$ itself, we use Kahan Summation (ref. Theorem 8 in [32]) which keeps a running compensation of the error when adding multiple small numbers.

The *trivial'* protocol in phase 2 works on items in $S'_{\mathtt{A}}$ (elements to send in `CKV`) and items in $S'_{\mathtt{B}}$ (elements to match with) in order to identify the resolve sets. We use $S'_{\mathtt{A}}$ and $S'_{\mathtt{B}}$ in the calculation of b_k (eqs. (5.4) and (5.5), page 56) as well as $\delta_{exp} = 100\,\%$ since they may be distinct, e.g. in the *missing* scenario. Note that $FR(p_2)$ is set as the target failure rate but the actual worst-case failure rate $fr'_t(S'_{\mathtt{A}}, S'_{\mathtt{B}})$ of the *trivial'* protocol may be lower due to rounding.

6.4.3 Overall Failure Rate

The overall effective worst-case failure rate fr'_s of the SHash reconciliation protocol may be derived by adding the two effective worst-case failure rates $fr'_s(p_1)$ and $fr'_t(S'_{\mathtt{A}}, S'_{\mathtt{B}})$ due to the linearity of the expected number of failures:

$$fr'_s := fr'_s(p_1) + \underbrace{fr'_t(S'_{\mathtt{A}}, S'_{\mathtt{B}})}_{=:fr'_s(p_2)} \leq FR \tag{6.11}$$

Differences between fr'_s and the target failure rate FR are caused by b_k and b'_k being integral and will be discussed in Section 6.5 below.

6.4.4 Overall Costs

With the previous considerations, we accumulate the costs C_s of the SHash reconciliation algorithm and provide upper bounds for the delta-encoded and zlib-compressed structures that are created in the two phases of the protocol:

$$C_s = \left|\mathtt{SH}(S_\mathtt{B})\right| + \left|\mathtt{CK}_{idx}(S'_\mathtt{B})\right| + \underbrace{C_{t'}}_{\text{trivial'}}\Big(\overbrace{S'_\mathtt{A}}^{\text{elements to send}}, \underbrace{|S'_\mathtt{B}|}_{\text{elements to match with}}, FR(p_2), \delta_{exp} = 100\,\%\Big)$$

$$= \overbrace{\left|\mathtt{SH}(S_\mathtt{B})\right|}^{\leq n_\mathtt{B}\cdot b'_k} + \overbrace{\left|\mathtt{CK}_{idx}(S'_\mathtt{B})\right|}^{\substack{\leq |\Delta|\cdot\lceil\log_2(n_\mathtt{B})\rceil + t \\ \approx |\Delta\setminus Mis_\mathtt{B}|\cdot s + t}} + \overbrace{\left|\mathtt{CKV}(S'_\mathtt{A})\right|}^{\lessapprox |\Delta|\cdot(b_k+32)} + \overbrace{\left|\mathtt{CK}'_{idx}(\Delta'_\mathtt{B})\right|}^{\lessapprox |\Delta|} \tag{6.12}$$

$$\in \mathcal{O}\left(n\cdot\log\frac{\tilde{n}_\Delta}{FR} + |\Delta|\cdot\log\frac{n\cdot|\Delta|}{FR}\right) \quad \text{(for } n,\tilde{n}_\Delta,|\Delta|\to\infty,\ FR\to 0\text{)}$$

Similarly to Section 5.4.2, inside the $\mathcal{O}$ notation, $\tilde{n}_\Delta \coloneqq \lceil\delta_{exp}\cdot\tilde{n}\rceil \geq \tilde{n}_{X\in\{\mathtt{A}_\Delta,\mathtt{B}_\Delta\}}$ (ref. eq. (6.6)) may be replaced by $n\cdot\delta_{exp} \approx |\Delta|$ or even n due to $\delta_{exp} \leq 100\,\%$.

Proving the Communication Complexity

The overall communication complexity of eq. (6.12) follows from the communication complexity of the two phases analysed below, i.e. eqs. (6.13) and (6.16).

Phase 1 From eq. (6.7), it is obvious, that $b'_k \leq \log_2(\max(^{4\cdot\tilde{n}_\Delta^2}/_{FR}+2, 2))$ and we deduce the communication complexity of $\mathtt{SH}(S_\mathtt{B})$ as:

$$|\mathtt{SH}(S_\mathtt{B})| \in \mathcal{O}\left(n\cdot\log\left(\frac{\tilde{n}_\Delta}{FR}\right)\right) \quad \text{(for } n,\tilde{n}_\Delta\to\infty,\ FR\to 0\text{)} \tag{6.13}$$

Phase 2 For the first $\mathtt{CK}_{idx}$, we refer to eq. (5.9) of Section 5.4.2 and derive:

$$|\mathtt{CK}_{idx}(S'_\mathtt{B})| \in \mathcal{O}\left(|\Delta|\cdot\log n\right) \quad \text{(for } n,|\Delta|\to\infty\text{)} \qquad \text{(with } S'_\mathtt{B}\approx\Delta\text{)} \tag{6.14}$$

Similarly, Section 5.4.2 analyses the $\mathtt{CKV}$ structure but there are different sets and parameters here, i.e. $|S'_\mathtt{A}|$ items in $\mathtt{CKV}$ (as $n_\mathtt{B}$), $|\Delta|$ different keys (as n), the upper bound $FR(p_2)$ on the failure rate, and $\delta_{exp} = 100\,\%$. With $\tilde{n} \leq 2\Delta$ in eq. (5.8) (page 58) due to the different n, we derive:

$$|\mathtt{CKV}(S'_\mathtt{A})| \in \mathcal{O}\left(|\Delta|\cdot\log\frac{|\Delta|}{FR(p_2)}\right) \subseteq \mathcal{O}\left(|\Delta|\cdot\log\frac{|\Delta|}{FR}\right) \tag{6.15}$$

$$\text{(for } |\Delta|\to\infty,\ FR(p_2)\to 0\text{)} \qquad \text{(for } |\Delta|\to\infty,\ FR\to 0\text{)}$$

The alternate $\mathtt{CK}'_{idx}$ representation requires $\mathcal{O}(|\Delta|)$ bits (for $|\Delta|\to\infty$) and we may thus derive the communication complexity of phase 2:

$$\mathcal{O}\left(|\Delta|\cdot\log(n) + |\Delta|\cdot\log\frac{|\Delta|}{FR} + |\Delta|\right) \subseteq \mathcal{O}\left(|\Delta|\cdot\log\frac{n\cdot|\Delta|}{FR}\right) \tag{6.16}$$

$$\text{(for } \tilde{n},|\Delta|\to\infty,\ FR\to 0\text{)} \qquad \text{(for } \tilde{n},|\Delta|\to\infty,\ FR\to 0\text{)}$$

6.5 Effective Worst-Case Accuracy

Similarly to the *trivial* reconciliation, b'_k in phase 1 and b_k in phase 2 must be integral (ref. eqs. (5.5) and (6.7), pages 56 and 73). Therefore, we cannot get arbitrarily close to FR but instead choose the closest possible value below. For $FR \in \{10, 0.1, 0.001\}$ and $\delta_{exp} \coloneqq \delta \in \{1\,\%, 10\,\%\}$, the following two sections show these b'_k and b_k as well as the resulting effective worst-case failure rates in both phases, i.e. $fr'_s(p_1)$ from eq. (6.8) (page 74) and $fr'_s(p_2) \coloneqq fr'_t(S'_{\mathtt{A}}, S'_{\mathtt{B}})$ from eq. (5.6) (page 57), based on the information the algorithm has, i.e. $n_{\mathtt{A}}$, $n_{\mathtt{B}}$, δ_{exp} and FR in phase 1 and $S'_{\mathtt{A}}$ and $S'_{\mathtt{B}}$ in phase 2. Note, however, that the *observed* failure rate in Section 6.6 may be lower than the effective worst-case failure rate presented here since $\tilde{n}_{X \in \{\mathtt{A}_\Delta, \mathtt{B}_\Delta\}}$ from eq. (6.6) is based on $\tilde{n}$ which uses a contradicting worst-case assumption.

6.5.1 Outdated Items Scenario

Table 6.2 shows the values of the parameters of the SHash reconciliation protocol in the *outdated* scenario. The number of bits b'_k per item is based on a target failure rate $FR(p_1) = {}^{FR}/_2$. Since b'_k is rounded up to the nearest integer, however, the effective worst-case failure rate $fr'_s(p_1)$ of phase 1 (ref. eq. (6.8)) is different to $FR(p_1)$ as shown, i.e. in the worst-case only ${}^{FR(p_1)}/_2$ since adding one bit to b'_k roughly halves the failure rate for large enough b'_k. Phase 2 may, however, re-use these left-overs and sets an appropriate target rate $FR(p_2)$.

Table 6.2: SHash bit sizes b'_k and b_k and effective worst-case failure rates in the *outdated* scenario with $n = n_{\mathtt{A}} = n_{\mathtt{B}} = 100\,000$ and $\delta = \delta_{exp}$.

	$\delta = 1\,\%$			$\delta = 10\,\%$		
FR	0.0010000	0.10000	10.000	0.0010000	0.10000	10.000
$\tilde{n}$	100 503	100 503	100 503	105 264	105 264	105 264
$\tilde{n}_{X \in \{\mathtt{A}_\Delta, \mathtt{B}_\Delta\}}$	1 006	1 006	1 006	10 527	10 527	10 527
	phase 1					
$FR(p_1)$	0.0005000	0.05000	5.000	0.0005000	0.05000	5.000
b'_k	32	26	19	39	33	26
$fr'_s(p_1)$	0.0004713	0.03012	3.184	0.0004032	0.02580	3.297
	phase 2					
$\lvert S'_{\mathtt{A},\mathtt{B}} \rvert$	1 000			10 000		
$FR(p_2)$	0.0005287	0.06988	6.816	0.0005968	0.07420	6.703
b_k	34	27	21	41	34	27
$fr'_s(p_2)$	0.0004654	0.05958	3.811	0.0003638	0.04656	5.961
fr'_s	0.0009367	0.08969	6.995	0.0007669	0.07237	9.258

In order to determine b_k, the *trivial'* algorithm in phase 2 needs to know the sizes of $S'_{\mathtt{A}}$ and $S'_{\mathtt{B}}$ at $\mathtt{A}$ which it both has. For the examples of Table 6.2, we assume that phase 1 did not create any errors and therefore, $|S'_{\mathtt{A}}| = |S'_{\mathtt{B}}| = |\Delta'| =: |S'_{\mathtt{A,B}}|$. In contrast, the evaluation in Section 6.6 below uses the real values of $S'_{\mathtt{A}}$ and $S'_{\mathtt{B}}$. These define b_k along with its worst-case effective failure rate $fr'_s(p_2)$. In some of the examples, e.g. $FR = 0.1, \delta = 1\,\%$, phase 2 is able to compensate the inaccuracy of phase 1 and $fr'_s(p_2) > FR(p_1)$ which brings the overall failure rate fr'_s closer to the target rate FR.

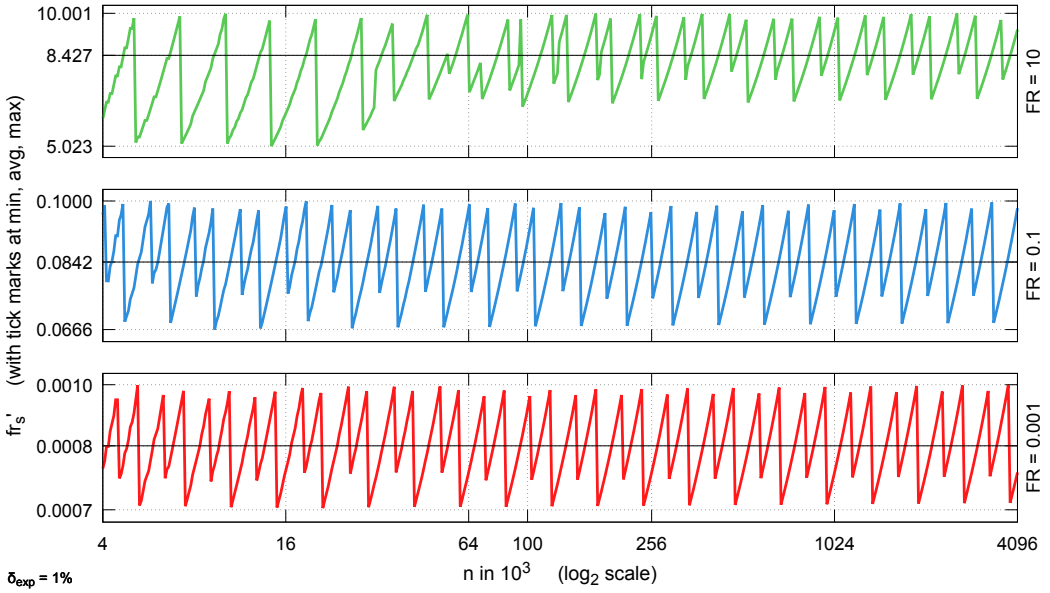


Figure 6.2: SHash fr'_s in the *outdated* scenario for different n ($\delta = \delta_{exp} = 1\,\%$).

More broadly, Figure 6.2 shows, how fr'_s behaves for different n. Similarly to the *trivial* algorithm alone (ref. Section 5.5) and except for low n with $FR = 10$ which we discuss below, we observe a repetitive pattern but here in the range of roughly $[2/3 \cdot FR, FR]$ due to the compensation phase 2 provides. Additionally, we observe two peaks of fr'_s in each repetition due to the rounding in each of the two phases. Further values for δ create the same patterns with the same amplitude and frequency but shifted along the x-axis due to the difference in $\tilde{n}$. For clarity, $\delta = 10\,\%$ has thus been omitted from the plot.

Figure 6.2 also shows some degradation of the effective failure rates for $FR = 10$ and low n that we want to draw attention to. In this constellation, any b'_k fulfils $FR(p_1)$ (ref. eq. (6.5)) and thus the minimum $b'_k = 1$ is used. With high probability, all items thus collide and enter phase 2 and SHash effectively becomes a *trivial'* reconciliation. Phase 1 is only able to remove common items between the nodes if b'_k is large enough. Until then, a transition period shows some intermediate behaviour. Note that this degradation to a *trivial'* reconciliation does not only influence the effective failure rate but also the communication costs as we will see in Section 6.6 below.

6.5.2 Missing Items Scenario

Table 6.3 shows the values of the parameters of the SHash protocol in the the *missing* scenario with a total number of $n = 100\,000$ unique items. The δ differences have been distributed equally among the nodes and thus $n_{\mathtt{A,B}} \coloneqq n_{\mathtt{A}} = n_{\mathtt{B}}$ is 99 500 for $\delta = 1\,\%$ and 95 000 for $\delta = 10\,\%$. Since the upper bound on the number of unique keys $\tilde{n}$ (eq. (5.4), page 56) accounts for this case, $\tilde{n}$ is equal to n. The size of the collision sets $\tilde{n}_{X \in \{\mathtt{A}_\Delta, \mathtt{B}_\Delta\}}$ follows accordingly (eq. (6.6), page 73) but, as expected, is too high in this scenario.

Table 6.3: SHash bit sizes b'_k and b_k and effective worst-case failure rates in the *missing* scenario with $\delta = \delta_{exp}$, $n = 100\,000$, and $n_{\mathtt{A,B}} \coloneqq n_{\mathtt{A}} = n_{\mathtt{B}}$ accordingly.

	$\delta = 1\,\%$, $n_{\mathtt{A,B}} = 99\,500$			$\delta = 10\,\%$, $n_{\mathtt{A,B}} = 95\,000$		
FR	0.0010000	0.10000	10.000	0.0010000	0.10000	10.000
$\tilde{n}$	100 000	100 000	100 000	100 000	100 000	100 000
$\tilde{n}_{X \in \{\mathtt{A}_\Delta, \mathtt{B}_\Delta\}}$	1 000	1 000	1 000	10 000	10 000	10 000
	phase 1					
$FR(p_1)$	0.0005000	0.05000	5.000	0.0005000	0.05000	5.000
b'_k	32	26	19	39	32	26
$fr'_s(p_1)$	0.0004657	0.02976	3.149	0.0003638	0.04656	2.976
	phase 2					
$\lvert S'_{\mathtt{A,B}} \rvert$	500			5 000		
$FR(p_2)$	0.0005343	0.07024	6.851	0.0006362	0.05344	7.024
b_k	32	25	19	39	32	25
$fr'_s(p_2)$	0.0004652	0.05954	3.807	0.0003638	0.04656	5.959
fr'_s	0.0009308	0.08930	6.957	0.0007276	0.09313	8.935

Similarly to the *outdated* scenario above, differences only arise from the different values of $\tilde{n}_{X \in \{\mathtt{A}_\Delta, \mathtt{B}_\Delta\}}$ and the different numbers of items participating in phase 2. The effective worst-case failure probabilities for different n, as shown by Figure 6.3, thus also exhibit the same pattern as above with minor deviations in the values. Note that here, the values of $\delta = 10\,\%$ have also been omitted for clarity since they only shift the values along the x-axis.

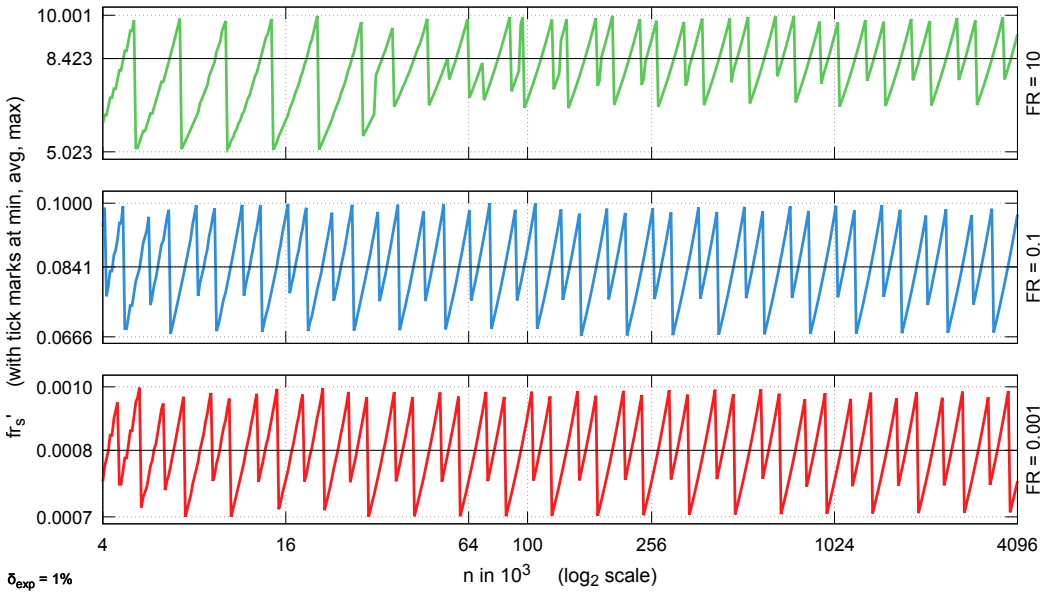


Figure 6.3: SHash fr'_s in the *missing* scenario for different n ($\delta = \delta_{exp} = 1\,\%$).

6.6 Evaluation

The SHash algorithm tries to improve on the *trivial* algorithm by only sending version numbers for different items. Similarly, phase 1 costs depend on n, δ_{exp}, and FR (ref. eq. (6.12), page 76, and eq. (6.7), page 73). Phase 2, however, includes both an own $\mathtt{CK}_{idx}$ (larger than inside a *trivial* reconciliation) and a *trivial'* reconciliation of the identified differences. These two phases eventually cause more overhead than by using the *trivial'* reconciliation alone.

Below, we first analyse the protocol's behaviour with different δ and FR assuming correct $\delta_{exp} := \delta$. We then evaluate the effects of an incorrect $\delta_{exp} \neq \delta$ as well as different data and failure distributions and conclude with an analysis of the scalability for varying n and FR.

6.6.1 General Analysis for Different δ and FR

For both $\delta \in (0, 10]\,\%$ and $\delta \in (0, 100]\,\%$, shown in Figures 6.4 and 6.5, the actual failure rate, i.e. the sum of the redundantly transferred items and the missed $|\Delta|$, is much lower than the configured FR despite the effective worst-case failure rate being closer to FR than a *trivial* reconciliation alone. This is not only due to b_k and b'_k being integral but, as argued above, also

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand}$

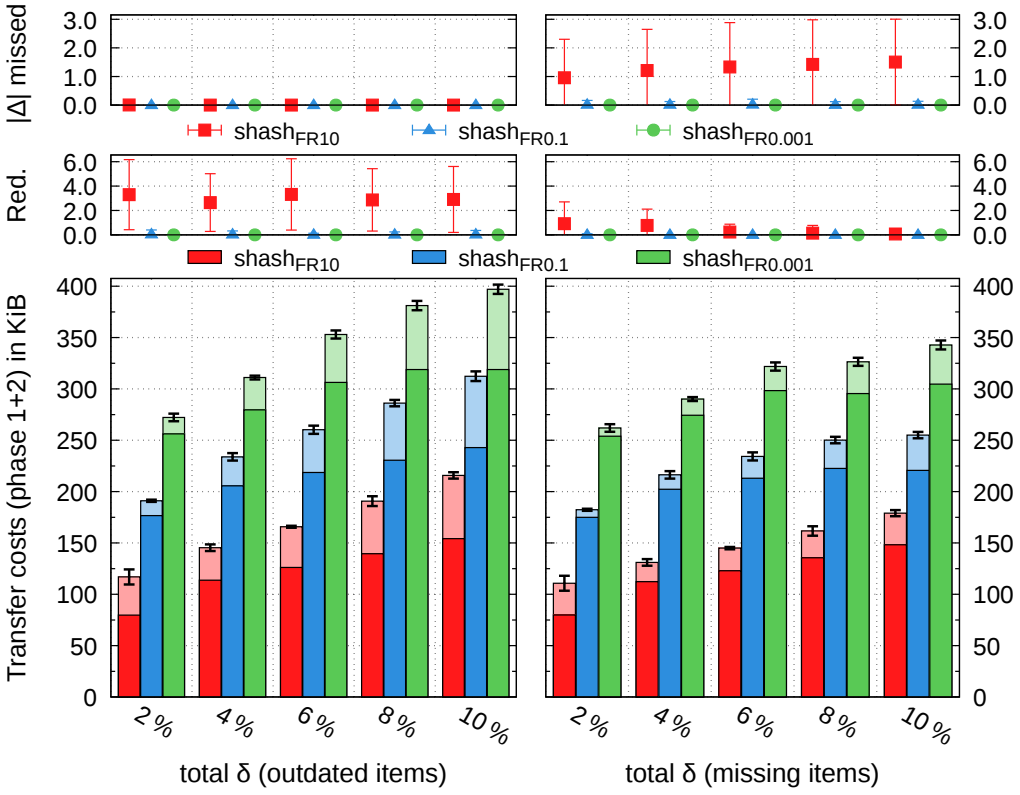


Figure 6.4: SHash reconciliation with small δ and different FR.

due to $\tilde{n}_{X \in \{\mathtt{A}_\Delta, \mathtt{B}_\Delta\}}$ using contradicting worst-case assumptions for the number of unique keys, i.e. only missing items in Δ, and for the size of the collision sets, i.e. only newer items in Δ, (ref. eq. (6.6)). Although this case never exists—and thus resources are wasted due to a too high accuracy—we cannot avoid it without further information on the type of Δ to expect.

Unidentified Δ practically never occurs in the *outdated* scenario due to its low probability and the only cause being in phase 1 (ref. Section 6.3.2). Redundant transfers, however, could by caused by any phase. In contrast, unidentified items of Δ in the *missing* scenario may result from either phase and are more likely but redundant transfers only result from phase 2.

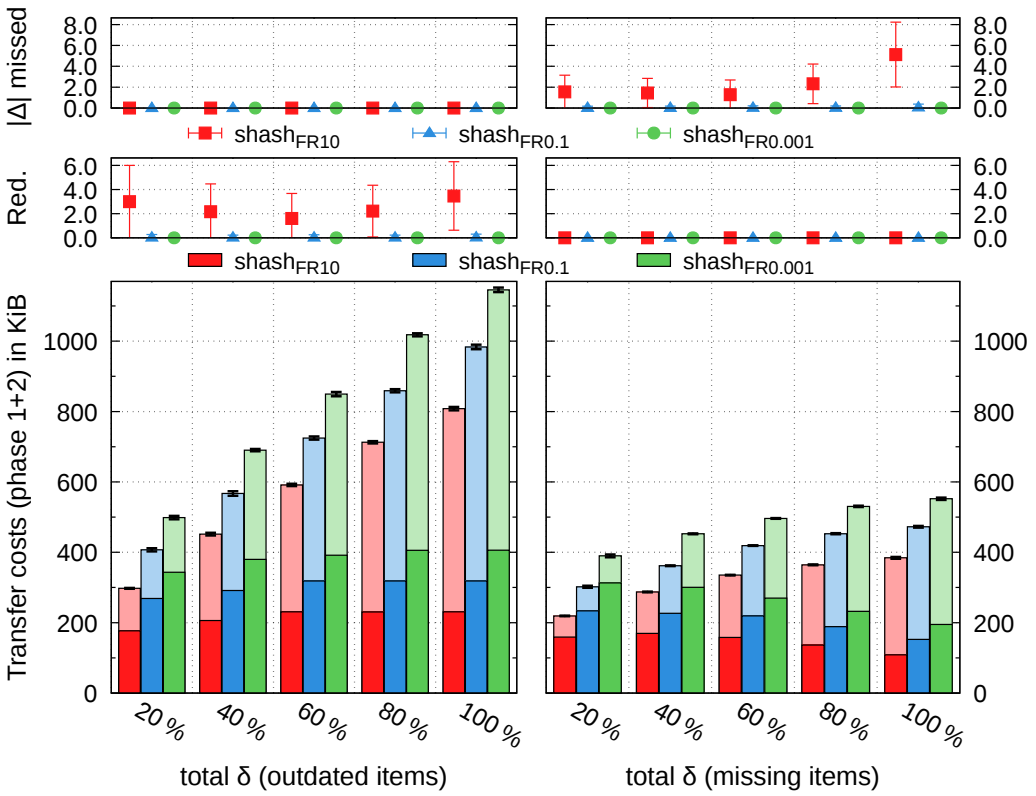


Figure 6.5: SHash reconciliation with high δ and different FR.

Both phases' transfer costs grow logarithmically with δ but phase 1 costs grow much slower towards $\delta = 100\,\%$ than those of phase 2 (also ref. Section 6.4.4). With these values, the upper bounds $n_\mathtt{A}$ and $n_\mathtt{B}$ on the size of the collision sets (eq. (6.6), page 73) limit the growth of $\tilde{n}_{\mathtt{A}_\Delta}$ and $\tilde{n}_{\mathtt{B}_\Delta}$, respectively. Please note that in the *missing* scenario, the growth of b'_k and b_k is compensated in the transfer costs by the reduced number of items on each of the nodes. Although discussed in more detail in Section 6.6.5 below, Figures 6.4 and 6.5 also support our theoretical bound on the communication costs complexity (ref. eq. (6.12), Section 6.4.4) regarding FR indicating that the phase 1 costs increase logarithmically with FR^{-1} and so do the total transfer costs.

SHash Degradation

As already seen in Section 6.5, the SHash reconciliation may degrade to a *trivial'* reconciliation if b'_k is not large enough to filter out enough items from S_A and S_B. This may happen if δ_{exp} or n is low or FR is large as shown by Figure 6.6 for $FR = 10$ and $\delta \leq 1\,\%$. The lower the δ, the lower b'_k until the minimum $b'_k = 1$ is used (ref. eq. (6.5), page 73). To eliminate this degradation, future work could set a larger lower bound for b'_k, e.g. for $FR = 10$, the b'_k calculated for $\delta = 2\,\%$. This would practically fix phase 1 costs for lower δ while phase 2 costs still decrease with δ as expected. Note, however, that despite its improvements, this solution still limits the scalability of the SHash reconciliation for low δ_{exp} or n.

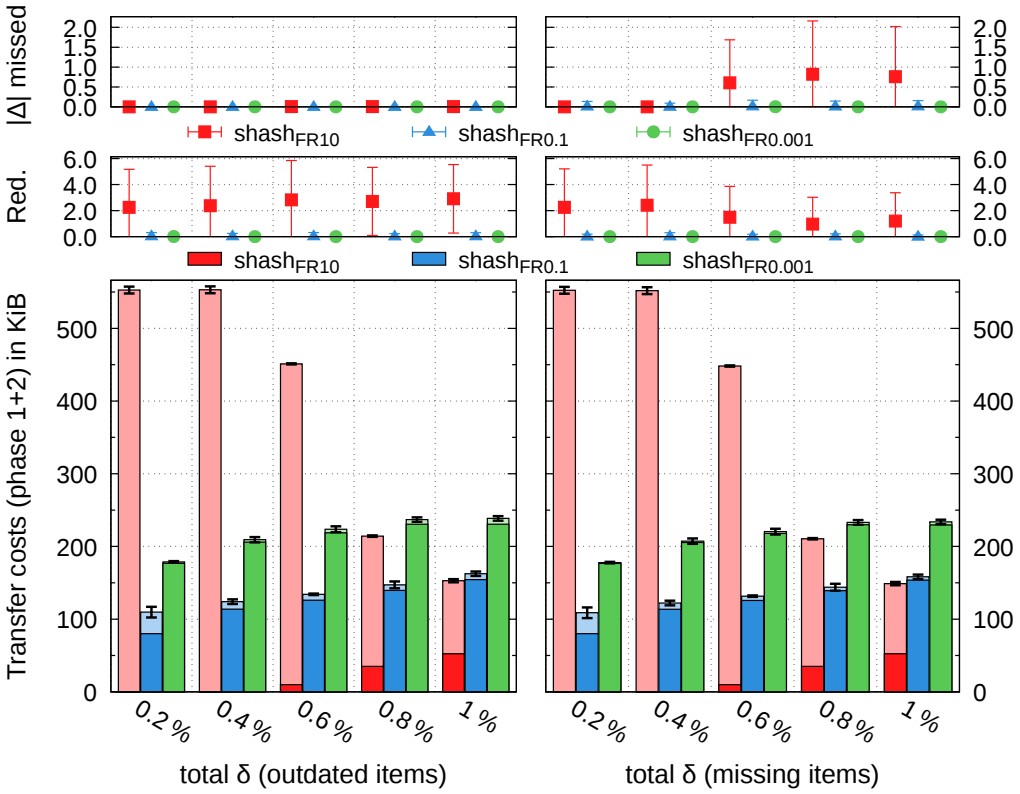


Figure 6.6: SHash reconciliation with very low δ and different FR.

6.6.2 What if δ_{exp} is Wrong?

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$ vs. 1 %
$fail_{rand}$

The SHash reconciliation protocol uses δ_{exp} to estimate the worst-case number of unique keys $\tilde{n}$ and derive the worst-case size of the collision sets $\tilde{n}_{X \in \{A_\Delta, B_\Delta\}}$ accordingly (eq. (6.6), page 73). As detailed above, however, these two use contradicting worst-case assumptions which result in $\tilde{n}_X$ and thus b'_k being too large for a correct $\delta_{exp} = \delta$ but if δ_{exp} is wrong, they will eventually be too small to fulfil FR, as shown by Figure 6.7.

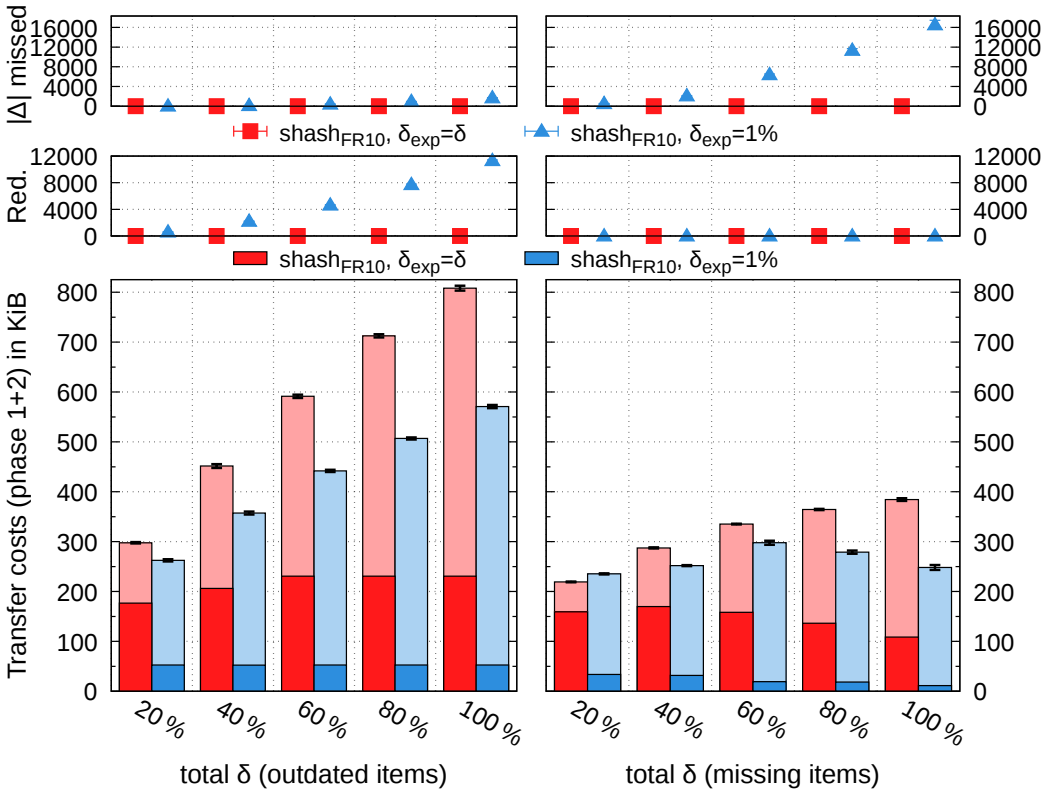


Figure 6.7: SHash reconciliation with high δ and different δ_{exp}.

For $\delta = 100\,\%$ in the *outdated* scenario, $n = 100\,000 = n_{X \in \{A_\Delta, B_\Delta\}}$, i.e. the real size of the collision sets of each node. This would result in $b'_k = 32$ and thus $fr(p_1) = 4.65$ (eq. (6.8), page 74) which is below the target $FR(p_1) = 5$. With $\delta_{exp} = 1\,\%$, however, $\tilde{n} = 100\,503$ and $\tilde{n}_{X \in \{A_\Delta, B_\Delta\}} = 1\,006$ which is much lower. This results in $b'_k = 19$ and, by using the actual $n_{X \in \{A_\Delta, B_\Delta\}}$, yields a phase 1 effective worst-case failure rate of $fr(p_1) = 26\,049$. Since our evaluation setup, however, distributes differences uniformly at random among the nodes, the most likely outcome of a single collision is one redundant item transfer (ref. Section 6.3.2) which causes a much lower observed failure rate in Figure 6.7.

Please also note that phase 2 is not directly affected by δ_{exp} since it assumes 100 % differences between S'_A and S'_B. It is thus only affected by the additional items from the multiple collisions cases of phase 1.

6.6.3 Data and Failure Distribution Sensitivity

$n = 100\,000$
$data_{rand,\ bin\,0.2}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand,\ bin_{0.2}}$

Similarly to the *trivial* algorithm, SHash only shows a minor sensitivity to different data or failure distributions due to our encoding schemes and the use of hashes. The differences shown by Figure 6.8 are very small and do not show a consistent trend for either variation. They are more likely due to normal variations in the simulations. The minor deviation that we observed in the *trivial* reconciliation when applied to a binomial data distribution (ref. Section 5.6.3) seems to be mitigated by either the much lower b'_k compared to b_k or the use of two phases during the reconciliation.

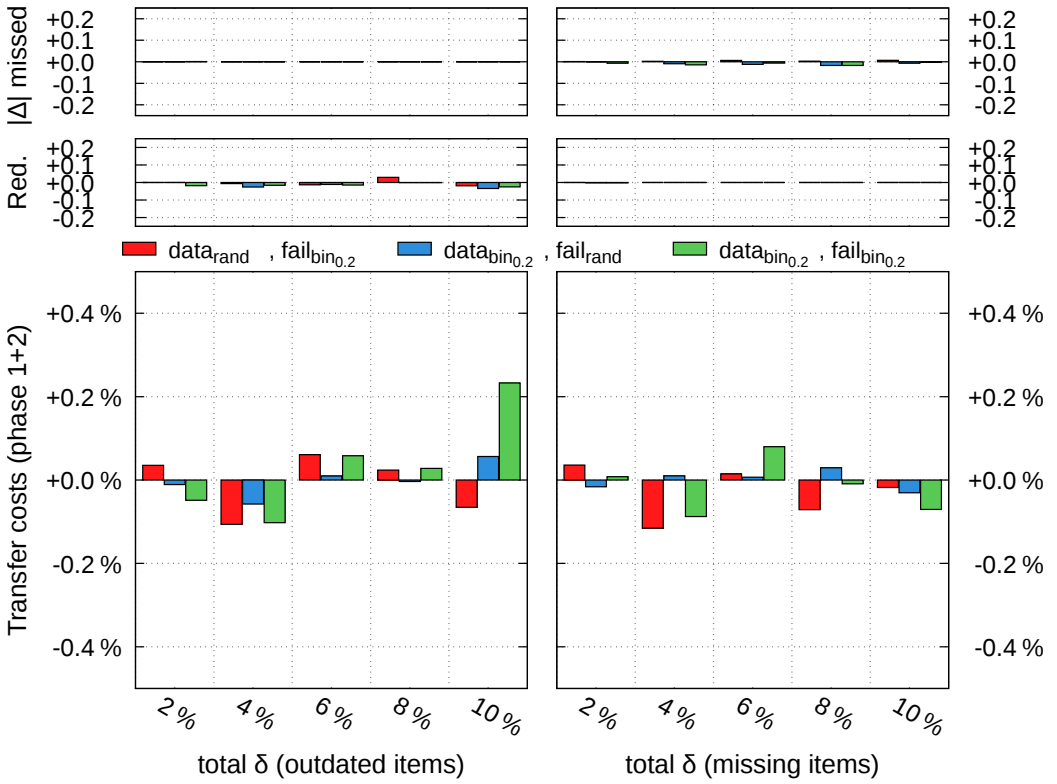


Figure 6.8: SHash reconciliation with $FR = 0.1$ and different data and failure distributions compared to $data_{rand}, fail_{rand}$

6.6.4 Scalability with the System Size n

$n = variable$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 6.9 shows how the SHash algorithm performs with smaller and larger numbers of items than above and a fixed $\delta = 3\,\% =: \delta_{exp}$. Since here we combine the techniques from the *trivial* reconciliation, the results are similar, too, and we find support for the theoretical $\mathcal{O}(n \cdot \log n)$ scalability for $n \to \infty$, $|\Delta| \in \mathcal{O}(n)$ and FR constant (ref. eq. (6.12), page 76).

The observed failure rate seems stable in the *outdated* scenario but in the *missing* scenario, redundant item transfers seem to switch to unrecognised items of Δ with the sum of the two slightly decreasing. This may result from the two contradicting worst-case assumptions for $\tilde{n}_{X \in \{A_\Delta, B_\Delta\}}$ which influence the occurrence of single collisions quadratically (ref. eq. (6.3), page 72).

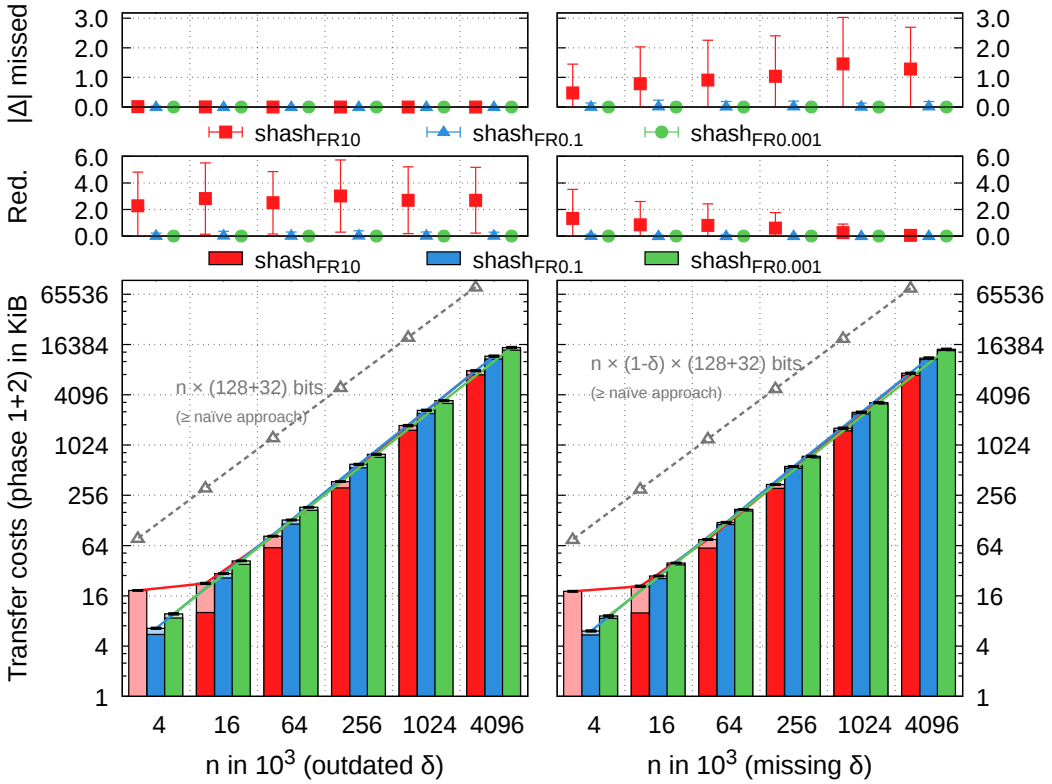


Figure 6.9: SHash reconciliation scalability with data size n and different FR.

As discussed above, the SHash algorithm may degrade to the *trivial'* reconciliation for low δ_{exp} or n or large FR. Here, we observe this degradation for low n with $FR = 10$. With increasing n, SHash eventually returns to its original form. As with the *trivial* algorithm, however, the transfer costs are also approaching the naïve transfer costs with the same consequences (please refer to Section 5.6.4 for details).

6.6.5 Scalability with the Target Failure Rate *FR*

$n = 100\,000$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 6.10 shows how the communication costs of the SHash reconciliation evolve with different target failure rates *FR* and constant n. The failure rate evolves as expected but please recall that the results of the lower values of *FR* are not representative in our 1 000 random simulations, e.g. if a failure should only occur only once every 10 000 simulations. For larger *FR*, again, we observe the start of the degradation of the SHash algorithm to the *trivial'* reconciliation by looking at the costs of the two phases. Other than this degradation, Figure 6.10 supports the communication costs complexity of $\mathcal{O}(\log(1/FR))$ for n and $|\Delta|$ constant (ref. eq. (6.12), page 76).

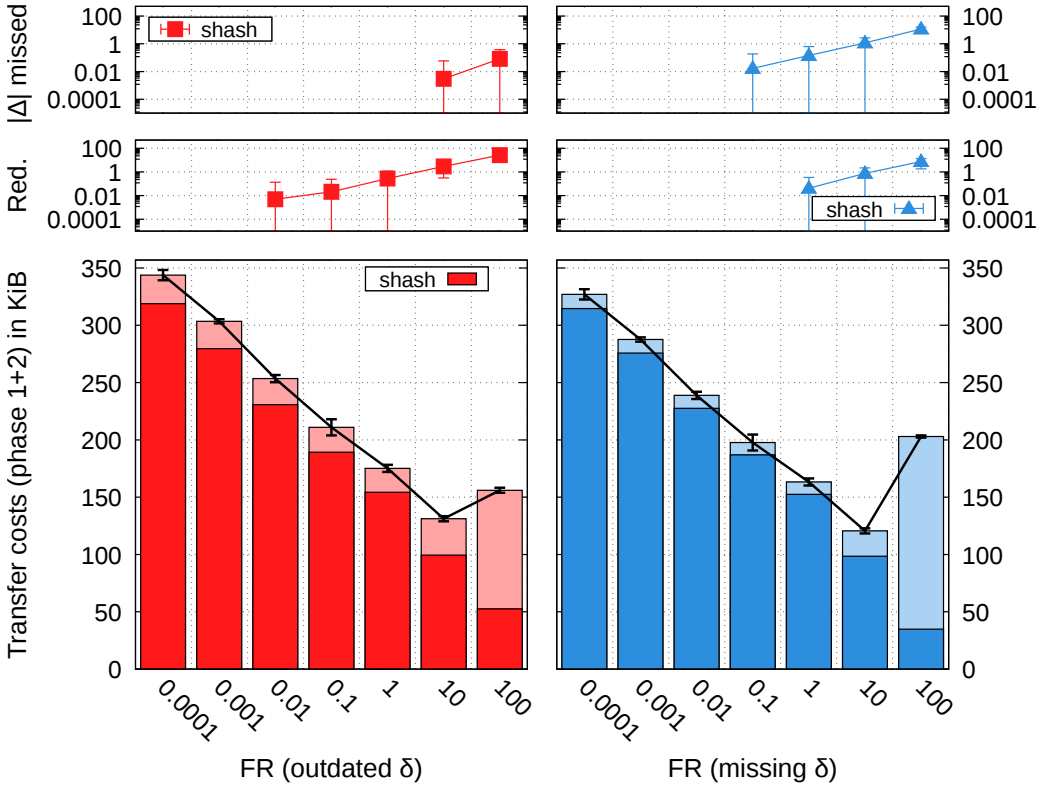


Figure 6.10: SHash reconciliation scalability with the target failure rate *FR* ($\log_{10}$ scale on the x-axis and on the y-axis except for the transfer costs).

Chapter 7

Bloom Filter Reconciliation

While SHash sends a single hash for each item, a more advanced data structure may save even more bandwidth. This section elaborates Bloom filters [3] for the set reconciliation problem. Bloom filters are an efficient probabilistic data structure for representing sets and have been intensely studied in past and current research. There are many extensions to the original algorithm as well as many applications in different systems and algorithms [8, 78].

7.1 Protocol

The set reconciliation protocol with Bloom filters is similar to the SHash algorithm. Instead of creating a hash list, though, B creates a Bloom filter BF [3] from the key and version of the items in S_B and sends BF to A (Figure 7.1), along with some metadata. With this BF, A checks whether its data items are present on B and identifies $S'_A \supseteq \Delta'_B \cup Old'_A, S'_A \subseteq S_A$. Since (standard) Bloom filters can neither be enumerated nor unmatched hashes/bits can be used as in the SHash protocol, Mis_A can not be identified. We therefore create another Bloom filter with the items of S_A and send the XOR of this Bloom filter and BF as Diff-BF(S_A) back to B. Analogously, B identifies $S'_B \supseteq \Delta'_A \cup Old'_B, S'_B \subseteq S_B$.

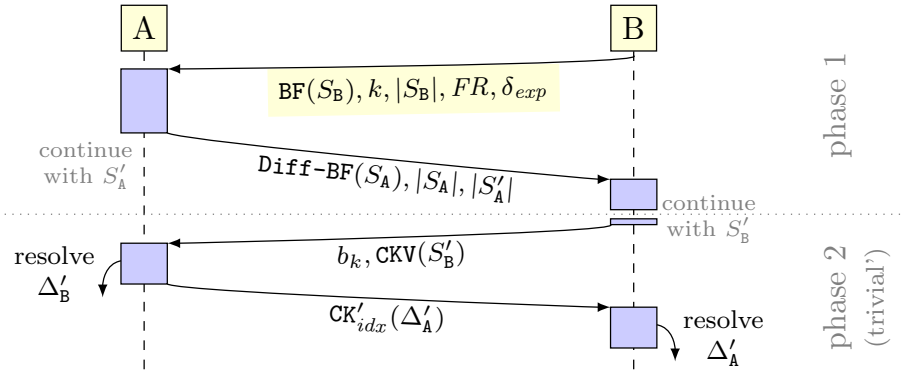


Figure 7.1: *Bloom filter* reconciliation protocol.

Phase 2 operates on $S'_{\texttt{A}}$ and $S'_{\texttt{B}}$ using the *trivial'* reconciliation protocol from the SHash algorithm (ref. Section 6.1) with a reversed direction, i.e. `B` sending `CKV` instead of `A`. At the end, $\Delta'_{\texttt{A}}$ and $\Delta'_{\texttt{B}}$ are pushed to `A` and `B`, respectively.

7.2 Bloom Filter Details

A Bloom filter [3] is an approximate data structure which encodes n items (here: `{key, version}` pairs) into m bits. Initially all bits are set to 0 and items are added by setting k bits to 1 (ref. Figure 7.2). These k positions are calculated using k *independent and uniformly distributed* hash functions $g_{i \in \{0,1,\ldots,k-1\}}$ (also see the box below). Membership queries for an item x check that all the $g_i(x)$ bits are set to 1. If any of these bits is 0 then $x \notin \texttt{BF}$, otherwise $x \in \texttt{BF}$ with a *false-positive probability* FP of a *single* membership query.

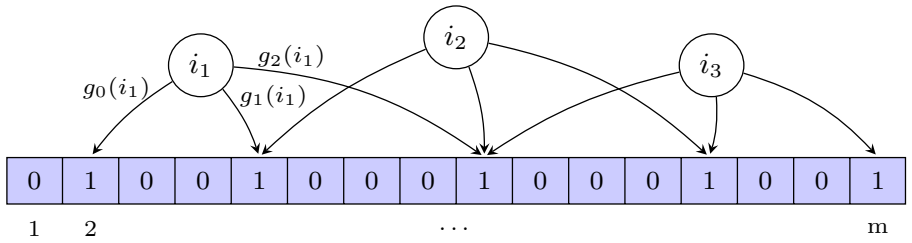


Figure 7.2: *Bloom filter* example for items i_1, i_2, and i_3 with $k = 3$.

Less Hashing, Same Performance

The computation of k hash functions may be expensive and thus, schemes such as *double hashing* [23] have been proposed. The idea is to derive the k independent and uniformly distributed hash functions g_i based on two hash functions h_1 and h_2 and $g_i(x) = h_1(x) + i \cdot h_2(x) \mod m$. Dillinger and Manolios [23] require that $h_2(x)$ be non-zero and relatively prime to m to ensure the uniqueness of the positions and thus a stable FP. In a later theoretical analysis, Kirsch and Mitzenmacher [42] claim the same *asymptotic* behaviour for g_i with *any* $h_2(x)$ and m (for $n \to \infty$).

Our first implementation using h_1 = MD5 and h_2 = SHA1 without restrictions on $h_2(x)$ or m, however, revealed a higher FP than given by eq. (7.1) below. This could be caused by m and k depending on the actual n in our algorithm (see below) and thus n not being large enough (in relation to m and k) for the asymptotics to kick in.

Therefore, instead of using double hashing or similar schemes, we hash the combined x and i and use $g_{i \in \{1,2,\ldots,k\}}(x) := \texttt{SHA1(x + i)} \mod m$. We then reduce the number of actual hash operations by splitting each 160 bit hash into $c := \lfloor 160 / \lceil \log_2(m) \rceil \rfloor$ smaller independent and uniformly distributed hash values and use the first k (unique) hashes in $\cup_{i \in \{1,2,\ldots,\lceil k/c \rceil\}} \texttt{split}(g_i(x))$.

Mullin [61] analyses Bloom filters mathematically and shows how FP is calculated and how m needs to be set to fulfil FP for a given k:

$$FP \coloneqq FP(n,m,k) = \overbrace{\left(1 - \underbrace{\left(1 - {}^1\!/_m\right)^{k\cdot n}}_{=:p_0,\ \text{the probability that any given bit remains 0}}\right)}^{k\ \text{positions to check}}{}^{k} \quad \Leftrightarrow \quad m = \frac{1}{1 - \sqrt[k\cdot n]{1 - \sqrt[k]{FP}}} \tag{7.1}$$

Two competing forces influence FP when k is increased: (a) p_0 decreases with k and thus $1 - p_0$ increases, and (b) the number of positions to check increases with k and thus $(1 - p_0)^k$ decreases for fixed $p_0 \in (0,1) \subset \mathbb{R}$. Figure 7.3 shows this correlation and the resulting m and $FP(n,m,k)$ for arbitrary k using fixed n and a target false positive probability FP_t from the examples of Section 7.5.

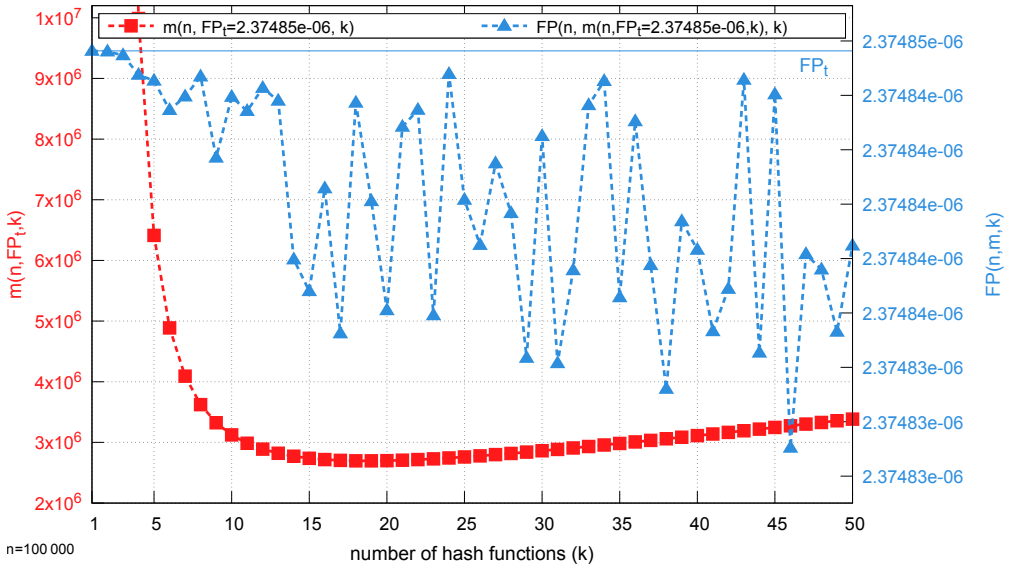


Figure 7.3: *Bloom filter* size m (left axis, red) and the resulting $FP(n,m,k)$ (right axis, blue) with this m based on eq. (7.1) for different k and fixed $n = 100\,000$ and $FP_t = 2.37485 \cdot 10^{-6}$.

Floating Point Precision Calculating $FP(n,m,k)$

In order to calculate $FP(n,m,k)$ in eq. (7.1) with high precision for large m, n, or k, we need to avoid problems with floating point numbers near 1. For this, we use the $\texttt{ln1p} \coloneqq \ln(1+x)$ function by Goldberg [32] (Theorem 4) and calculate the following (algebraically equal) equation for $FP(n,m,k)$:

$$\begin{aligned} FP(n,m,k) = \left(1 - \left(1 - {}^1\!/_m\right)^{k\cdot n}\right)^k &= e^{\ln\left(1 - e^{\ln\left(1-{}^1\!/_m\right)^{k\cdot n}}\right)^k} \\ = e^{k\cdot\ln\left(1 - e^{k\cdot n\cdot\ln\left(1-{}^1\!/_m\right)}\right)} &= e^{k\cdot\texttt{ln1p}\left(-e^{k\cdot n\cdot\texttt{ln1p}\left(-{}^1\!/_m\right)}\right)} \end{aligned} \tag{7.1a}$$

ℹ Floating Point Precision Calculating m

The calculation of m in eq. (7.1) also suffers from the floating point representation near 1. Since $\sqrt[k]{FP}$ is not too close to 0, however, we can use $z \coloneqq 1 - \sqrt[k]{FP}$ and work around the calculation of $1 - \sqrt[k \cdot n]{z}$ by using the series representation of the root:

$$1 - \sqrt[k \cdot n]{z} = 1 - \sum_{i=0}^{\infty} \frac{\ln^i(z)}{i! \cdot (k \cdot n)^i} = \sum_{i=1}^{\infty} -\frac{\ln^i(z)}{i! \cdot (k \cdot n)^i} \tag{7.1b}$$

We evaluate this sum as long as its floating point representation is changing and thus omit smaller terms. This usually happens early since $k \cdot n$ is large.

Fan et al. [27] and Mitzenmacher [56] further extend the Bloom filter mathematical analysis and derive the optimal number of hash functions $k = \ln(2) \cdot {}^{m}\!/_{n}$ from $FP(n, m, k) \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k$. Later, Broder and Mitzenmacher [8] also show that this k is optimal for the exact formula of $FP(n, m, k)$ (eq. (7.1)) and that this approximation is very close. Let $FP_k(n, m) \coloneqq FP\left(n, m, \ln(2) \cdot {}^{m}\!/_{n}\right)$ be the false positive probability for optimal k. By inserting the optimal k into $\left(1 - e^{-\frac{k \cdot n}{m}}\right)^k$, the following estimates can be derived:

$$FP_k(n, m) \approx 2^{-\ln(2) \cdot \frac{m}{n}} \quad \Leftrightarrow \quad m \approx \frac{n}{\ln(2)} \cdot \log_2 \frac{1}{FP_k(n, m)} \tag{7.2}$$

$$\Leftrightarrow \quad k \approx \log_2 \frac{1}{FP_k(n, m)} = -\log_2(FP_k(n, m)) \tag{7.3}$$

With the parameters from Figure 7.3, we verify an optimal $k \approx 19$ and the lowest $m \approx 2\,695\,726$. Please note, that in practice, k and m need to be integer parameters of the Bloom filter and we will go into this in the sections below.

ℹ On the false-positive rate of Bloom filters

Although $FP(n, m, k) = (1 - (1 - {}^{1}\!/_{m})^{k \cdot n})^k =: p^k$ (eq. (7.1)) is given in many publications on Bloom filters, Bose et al. [5] show that for $k \geq 2$, this is only a (strict) lower bound and only a good approximation for large Bloom filters. They show that

$$p^k < FP(n, m, k) \leq p^k \cdot \left(1 + \mathcal{O}\left(\frac{k}{p} \cdot \sqrt{\frac{\ln m - k \cdot \ln p}{m}}\right)\right) \tag{7.4}$$

under the condition that ${}^{k}\!/_{p} \cdot \sqrt{{}^{(\ln m - 2k \cdot \ln p)}\!/_{m}} \leq c$ for some constant $c < 1$.

We nevertheless continue with eq. (7.1) and the approximations of eqs. (7.2) and (7.3) since our values of m are large enough (depending on FR) in order to consider it a good approximation of p^k.

7.3 Set Reconciliation with Bloom filters

As briefly outlined above and pictured in Figure 7.1, upon receiving $\texttt{BF}(S_\texttt{B})$, node A checks each item $x \in S_\texttt{A}$ for existence in this Bloom filter. For any positive result, i.e. $x \in \texttt{BF}(S_\texttt{B})$, x is assumed to exist in $S_\texttt{B}$ with the same version and is thus discarded. Any other item $x \notin \texttt{BF}(S_\texttt{B})$ is put into $S'_\texttt{A}$ and enters phase 2. Note that any item $x \notin S_\texttt{B}$ may cause a false positive with probability $FP_\texttt{B}$ which will be detailed below. At the end of these checks, the set of items for further inspection in phase 2 is reduced to $S'_\texttt{A} \supseteq \Delta'_\texttt{B} \cup Old'_\texttt{A}$.

Similarly, A creates a Bloom filter of the items in $S_\texttt{A}$ from which B can identify $S'_\texttt{B} \supseteq \Delta'_\texttt{A} \cup Old'_\texttt{B}$ so that $Mis_\texttt{A}$ also enters phase 2. We will use the same size m and number of hash functions k for both Bloom filters for simplicity and to allow the following optimisation: instead of transmitting $\texttt{BF}(S_\texttt{A})$ as is, we XOR it with $\texttt{BF}(S_\texttt{B})$ and transfer this $\texttt{Diff-BF}(S_\texttt{A}) \coloneqq \texttt{BF}(S_\texttt{A})\ \texttt{XOR}\ \texttt{BF}(S_\texttt{B})$ instead. The lower the number of differences, the more bits are shared among $\texttt{BF}(S_\texttt{A})$ and $\texttt{BF}(S_\texttt{B})$ and the more zero-bits are in $\texttt{Diff-BF}(S_\texttt{A})$ which is thus easier to compress and leads to reduced communication costs.

Phase 2 then identifies $\Delta'_\texttt{B}$ and $\Delta'_\texttt{A}$ on nodes A and B using the *trivial'* reconciliation from above (ref. Section 6.3.1) and resolves the differences accordingly.

7.3.1 Implications of False Positives in Phase 1

Let $\texttt{BF}_Y \coloneqq \texttt{BF}(S_Y)$ be the Bloom filter with the items of node $Y \in \{\texttt{A}, \texttt{B}\}$ and a false positive probability of FP_Y. Also let the opposite node of Y be $\overline{Y}$, i.e. $\overline{Y} \coloneqq \texttt{B}$ if $Y = \texttt{A}$ and vice versa. A false positive during a membership query on node $\overline{Y}$ in Bloom filter $\texttt{BF}_Y$, i.e. $x \in \texttt{BF}_Y$ although $x \notin S_Y$, may occur for every $x \in S_{\overline{Y}} \setminus S_Y = Mis_Y \cup New_{\overline{Y}} \cup Old_{\overline{Y}} =: S_{\overline{Y}_\Delta}$, i.e. $S_{\texttt{A}_\Delta}$ or $S_{\texttt{B}_\Delta}$ on node A and B, respectively. In this case, x does not participate in phase 2 on $\overline{Y}$ and the following cases may arise for the reconciliation (summarised in Table 7.1):

- In the *missing* items scenario, x is missing on Y but without x participating in phase 2, the result is one unrecognised item of Δ.
- In the *outdated* items scenario, an older or newer item $y \in Y : y_k = x_k, y_v \neq x_v$ exists which may be a false positive in $\texttt{BF}_{\overline{Y}}$ and thus an unrecognised item of Δ or may not be a false positive in which case up to one item is redundantly transferred, i.e. if x is newer than y.

Overall, the total number of errors caused by the first phase is bound by once the expected number of false positives at either node.

7.3.2 Implications of Hash Collisions in Phase 2

As in the SHash algorithm (ref. Section 6.3.2), the results of hash collisions in CKV are the same as in the original *trivial* set reconciliation (ref. Section 5.3.1) but based on the difference sets phase 1 discovers: Since membership queries

Table 7.1: Bloom filter reconciliation errors caused by an item $x \in S_{\overline{Y}_\Delta}$ being a false positive in phase 1, i.e. $x \in \mathtt{BF}_Y, x \notin S_Y$.

Failure Case (in phase 1)	Result (after phase 2)
$x \in Mis_\mathtt{A} \cup Mis_\mathtt{B}$	one unrecognised item of Δ
$\exists y \in S_Y : y_k = x_k \wedge y \in \mathtt{BF}_{\overline{Y}} \wedge y_v \neq x_v$	$x, y \notin$ phase 2 $\Rightarrow$ one unrecognised item of Δ
$\exists y \in S_Y : y_k = x_k \wedge y \notin \mathtt{BF}_{\overline{Y}} \wedge y_v < x_v$	only the old $y \in$ phase 2 $\Rightarrow$ one redundant item transfer
$\exists y \in S_Y : y_k = x_k \wedge y \notin \mathtt{BF}_{\overline{Y}} \wedge y_v > x_v$	no errors (the newer y is sent)

in Bloom filters only exhibit false positives, phase 2 does not operate on any extraneous common items and thus $S'_\mathtt{A} \subseteq \Delta_\mathtt{B} \cup Old_\mathtt{A}$ and $S'_\mathtt{B} \subseteq \Delta_\mathtt{A} \cup Old_\mathtt{B}$. Therefore, redundant item transfers may only occur in the *outdated* scenario and due to collisions in the *trivial'* reconciliation sending outdated items.

7.4 Deducing Bloom Filter Parameters from FR

Similarly to the SHash algorithm in Chapter 6, Bloom filter reconciliation uses two approximate phases and we thus need to distribute the available FR. As above, we aim at using an equal part of FR in each phase by exploiting the expected value's linearity and set a target failure rate of $FR(p_1) \coloneqq {}^{FR}/_2$ (ref. eq. (6.1), page 71). Phase 2 then determines its own target failure rate based on the leftovers from the actual worst-case failure rate of phase 1.

7.4.1 Phase 1

Since the *failure rate $fr_b(p_1)$ of phase 1 of the Bloom filter reconciliation protocol* is bound by once the expected number of false positives at either node (ref. Section 7.3.1), and only items from $S_{\mathtt{A}_\Delta}$ and $S_{\mathtt{B}_\Delta}$ may cause them, we derive the following bound using the linearity of the expected value:

$$fr_b(p_1) \leq \underbrace{|S_{\mathtt{A}_\Delta}| \cdot \overbrace{1 \cdot FP_\mathtt{B}}^{\text{failure rate of a single item check in } \mathtt{BF}_\mathtt{B}}}_{\text{failure rate of all checks of } S_\mathtt{A} \text{ in } \mathtt{BF}_\mathtt{B}} + \underbrace{|S_{\mathtt{B}_\Delta}| \cdot \overbrace{1 \cdot FP_\mathtt{A}}^{\text{failure rate of a single item check in } \mathtt{BF}_\mathtt{A}}}_{\text{failure rate of all checks of } S_\mathtt{B} \text{ in } \mathtt{BF}_\mathtt{A}} \tag{7.5}$$

An Upper Bound on the Number of False Positive Candidates

An estimate of the sizes $n_{\mathtt{A}_\Delta} \coloneqq |S_{\mathtt{A}_\Delta}|$ and $n_{\mathtt{B}_\Delta} \coloneqq |S_{\mathtt{B}_\Delta}|$ has been derived in Section 6.4.1 (eq. (6.6)) which we use here as well:

$$\tilde{n}_{\Delta_X} \coloneqq \min\left(\left\lceil \delta_{exp} \cdot \max\left(\left\lceil \frac{n_\mathtt{A} + n_\mathtt{B}}{2 - \delta_{exp}} \right\rceil, |n_\mathtt{A} - n_\mathtt{B}|\right)\right\rceil, n_X\right) \gtrapprox n_{\Delta_X} \quad X \in \{\mathtt{A}, \mathtt{B}\} \tag{7.6}$$

Deriving a Common m and k for $\texttt{BF}_\texttt{A}$ and $\texttt{BF}_\texttt{B}$

In order to support a differencing Bloom filter `Diff-BF` with XOR-ed bits and save communication costs during the (compressed) transfer of $\texttt{BF}(S_\texttt{A})$, we need to use a common m and k for both Bloom filters. These can be derived from eq. (7.5) using $\tilde{n}_{\texttt{A}_\Delta}$ and $\tilde{n}_{\texttt{B}_\Delta}$ (eq. (7.6)) and limiting $fr_b(p_1)$ by $FR(p_1)$ but this solution requires finding new optima. For simplicity, however, we use a common target false positive probability FP_t for both Bloom filters, derive individual m_Y and k_Y for $\texttt{BF}_Y$ and select the most appropriate pair as the common m and k. Individual $FP_\texttt{A}$ and $FP_\texttt{B}$ can then be derived accordingly.

$$fr_b(p_1) \lessapprox n_{\texttt{A}_\Delta} \cdot FP_\texttt{B} + n_{\texttt{B}_\Delta} \cdot FP_\texttt{A} \lessapprox \overbrace{\tilde{n}_{\texttt{A}_\Delta} \cdot FP_\texttt{B} + \tilde{n}_{\texttt{B}_\Delta} \cdot FP_\texttt{A}}^{=:fr'_b(p_1)} \leq FR(p_1) \quad (7.7)$$

$$\Leftarrow \quad (\tilde{n}_{\texttt{A}_\Delta} + \tilde{n}_{\texttt{B}_\Delta}) \cdot FP_t \leq FR(p_1)$$

$$\Leftarrow \quad FP_t \coloneqq \frac{FR(p_1)}{\tilde{n}_{\texttt{A}_\Delta} + \tilde{n}_{\texttt{B}_\Delta}} = \frac{FR}{2 \cdot (\tilde{n}_{\texttt{A}_\Delta} + \tilde{n}_{\texttt{B}_\Delta})} \quad (7.8)$$

With this FP_t, the optimal number k of hash functions for both Bloom filters can be closely approximated using eq. (7.3). Similarly, individual optimal Bloom filter sizes m_Y can be approximated using n_Y and eq. (7.2):

$$k \approx -\log_2(FP_t) \qquad m_Y \approx \frac{n_Y}{\ln(2)} \cdot \log_2 \frac{1}{FP_t} \quad (7.9)$$

On the need for *integral* m and k

Regarding the fact that both m and k must be integral, Fan et al. [27] and Mitzenmacher [56] only consider k and simply choose the highest integer not greater than the optimal k to reduce computational costs. We consider both $k_1 \coloneqq \lfloor -\log_2(FP_t) \rfloor$ and $k_2 \coloneqq \lceil -\log_2(FP_t) \rceil$ and calculate appropriate $m_{Y,i\in\{1,2\}}$ based on the exact eq. (7.1) and round up to retain the chosen FP_t:

$$m_{Y,i\in\{1,2\}} \coloneqq \left\lceil \frac{1}{1 - \sqrt[n\cdot k_i]{1 - \sqrt[k_i]{FP_t}}} \right\rceil \quad (7.10)$$

From these two pairs, we choose the one that minimises m_Y and $FP_{Y,i} \coloneqq FP\left(n_Y, m_{Y,i}, k_i\right)$ from eq. (7.1) and satisfies $FP_{Y,i} \leq FP_t$:

$$(k_Y, m_Y) \coloneqq \begin{cases} (k_1, m_{Y,1}) & \text{if } m_{Y,1} < m_{Y,2} \text{ and } FP_{Y,i\in\{1,2\}} \leq FP_t \\ (k_1, m_{Y,1}) & \text{if } m_{Y,1} = m_{Y,2} \text{ and } FP_{Y,1} < FP_{Y,2} \leq FP_t \\ (k_1, m_{Y,1}) & \text{if } FP_t < FP_{Y,2} \text{ and } FP_{Y,1} < FP_{Y,2} \\ (k_1, m_{Y,1}) & \text{if } m_{Y,1} \leq m_{Y,2} \text{ and } FP_t < FP_{Y,1} = FP_{Y,2} \\ (k_2, m_{Y,2}) & \text{otherwise} \end{cases} \tag{7.11}$$

In theory, some of these cases, e.g. with $FP_t < FP_{Y,i}$, should not arise but may be possible due to floating point issues. In these cases, we may overshoot and have a slightly higher failure probability. We do, however, prefer the lowest error here and phase 2 will compensate the rest.

For each of the two integral pairs (k_Y, m_Y) ($Y \in \{\mathtt{A}, \mathtt{B}\}$), we determine the individual false positive probabilities $FP_{X,Y} \coloneqq FP(n_X, m_Y, k_Y)$ (eqs. (7.1) and (7.1a), page 91) of the Bloom filter checks on node $X \in \{\mathtt{A}, \mathtt{B}\}, X \neq Y$. By combining these two for each Y, we derive the resulting effective failure rates $fr'_{b,Y}(p_1)$ (eq. (7.7)) and choose the one that minimises m and $fr'_{b,Y}(p_1)$ and satisfies $fr'_{b,Y}(p_1) \leq FR(p_1)$:

$$X \coloneqq \begin{cases} \mathtt{A} & \text{if } m_\mathtt{A} < m_\mathtt{B} \text{ and } fr'_{b,Y\in\{\mathtt{A},\mathtt{B}\}}(p_1) \leq FR(p_1) \\ \mathtt{A} & \text{if } m_\mathtt{A} = m_\mathtt{B} \text{ and } fr'_{b,\mathtt{A}}(p_1) < fr'_{b,\mathtt{B}}(p_1) \leq FR(p_1) \\ \mathtt{A} & \text{if } FR(p_1) < fr'_{b,\mathtt{B}}(p_1) \text{ and } fr'_{b,\mathtt{A}}(p_1) < fr'_{b,\mathtt{B}}(p_1) \\ \mathtt{A} & \text{if } m_\mathtt{A} \leq m_\mathtt{B} \text{ and } FR(p_1) < fr'_{b,\mathtt{A}}(p_1) = fr'_{b,\mathtt{B}}(p_1) \\ \mathtt{B} & \text{otherwise} \end{cases}$$
$$k \coloneqq k_X \qquad m \coloneqq m_X \qquad fr'_b(p_1) \coloneqq fr'_{b,X}(p_1) \tag{7.12}$$

As above, some of these cases should not occur but may be possible due to floating point issues and will be compensated by phase 2.

Phase 1 Formulae Wrap-Up

Common, integral values of m and k for the target false positive rate $FP_t \coloneqq {FR(p_1)}/{(\tilde{n}_{\mathtt{A}_\Delta} + \tilde{n}_{\mathtt{B}_\Delta})}$ (eq. (7.8)) are determined with the rules above (eqs. (7.10) to (7.12)). For a simple classification, an estimate may be given by inserting into eq. (7.9):

$$k \approx -\log_2\left(\frac{FR(p_1)}{\tilde{n}_{\mathtt{A}_\Delta} + \tilde{n}_{\mathtt{B}_\Delta}}\right) \qquad m \approx \frac{\max(n_\mathtt{A}, n_\mathtt{B})}{\ln(2)} \cdot \log_2 \frac{\tilde{n}_{\mathtt{A}_\Delta} + \tilde{n}_{\mathtt{B}_\Delta}}{FR(p_1)} \tag{7.13}$$

From the exact common, integral values of m and k, however, the overall effective worst-case failure rate may be derived by inserting $FP(n_X, m, k)$

from eq. (7.1) (page 91) into eq. (7.7) for the appropriate Bloom filter:

$$fr_b(p_1) \lessapprox n_{\mathtt{A}_\Delta} \cdot FP_{\mathtt{B}} + n_{\mathtt{B}_\Delta} \cdot FP_{\mathtt{A}} \lessapprox \tilde{n}_{\mathtt{A}_\Delta} \cdot FP_{\mathtt{B}} + \tilde{n}_{\mathtt{B}_\Delta} \cdot FP_{\mathtt{A}} =: fr'_b(p_1)$$
$$fr'_b(p_1) = \tilde{n}_{\mathtt{A}_\Delta} \cdot \left(1 - (1 - {}^1\!/_m)^{k \cdot n_{\mathtt{B}}}\right)^k + \tilde{n}_{\mathtt{B}_\Delta} \cdot \left(1 - (1 - {}^1\!/_m)^{k \cdot n_{\mathtt{A}}}\right)^k \qquad (7.14)$$

7.4.2 Phase 2

Phase 2 of the Bloom filter reconciliation is identical to the *trivial'* protocol of phase 2 of the SHash reconciliation except for the reversed direction. We therefore refer to Section 6.4.2 for details of the summary here.

The targeted failure rate of phase 2 depends on the left-over failure rate after phase 1 (eq. (7.14)) and thus $FR(p_2) \coloneqq FR - fr'_b(p_1)$. This also compensates cases where the failure rate was overshot, i.e. $fr'_b(p_1) > FR(p_1)$, and makes sure that the overall reconciliation fulfils FR. The *trivial'* algorithm sends a `CKV` of $S'_{\mathtt{B}}$ with an appropriate number of bits b_k per item (eq. (5.5), page 56, with $\delta_{exp} = 100\,\%$) and matches them with $S'_{\mathtt{A}}$. The resulting effective worst-case failure rate $fr'_b(p_2) \coloneqq fr'_t(S'_{\mathtt{B}}, S'_{\mathtt{A}})$ may be lower than the target $FR(p_2)$.

7.4.3 Overall Failure Rate

Similarly to the SHash reconciliation, the overall effective worst-case failure rate fr'_b of the Bloom filter reconciliation protocol may be derived by adding the two effective worst-case failure rates $fr'_b(p_1)$ and $fr'_b(p_2)$ based on the linearity of the expected number of failures (differences between fr'_b and FR will be discussed in Section 7.5 below):

$$fr'_b \coloneqq fr'_b(p_1) + fr'_b(p_2) \leq FR \qquad (7.15)$$

7.4.4 Overall Costs

With the considerations from above and omitting the savings from delta-encoding or zlib message compression, we accumulate an upper bound of the overall Bloom reconciliation communication costs C_b as (with $\tilde{n}_\Delta \coloneqq \tilde{n}_{\Delta_{\mathtt{A}}} + \tilde{n}_{\Delta_{\mathtt{B}}}$):

$$\begin{aligned} C_b &= m + |\mathtt{Diff\text{-}BF}| + \underbrace{C_{t'}}_{\text{trivial'}}\Big(\overbrace{S'_{\mathtt{B}}}^{\text{elements to send}}, \underbrace{|S'_{\mathtt{A}}|}_{\text{elements to match with}}, FR(p_2), \delta_{exp} = 100\,\%\Big) \\ &\leq 2m + \underbrace{\left|\mathtt{CKV}(S'_{\mathtt{B}})\right|}_{\lessapprox |\Delta| \cdot (b_k + 32)} + \underbrace{\left|\mathtt{CK}'_{idx}(\Delta'_{\mathtt{A}})\right|}_{\lessapprox |\Delta|} \qquad \in \underbrace{\mathcal{O}\left(n \cdot \log \frac{\tilde{n}_\Delta}{FR} + |\Delta| \cdot \log \frac{|\Delta|}{FR}\right)}_{(\text{for } n, \tilde{n}_\Delta, |\Delta| \to \infty,\ FR \to 0)} \end{aligned} \qquad (7.16)$$

Similarly to the SHash reconciliation (Section 6.4.4), $\tilde{n}_\Delta$ may be replaced by n or $n \cdot \delta_{exp} \approx |\Delta|$ inside the $\mathcal{O}$ notation which results in $\mathcal{O}((n + |\Delta|) \cdot \log({}^{|\Delta|}\!/_{FR}))$.

Proving the Communication Complexity

The given communication complexity follows from calculating the Bloom filter size m as well as k (eq. (7.9)) by using FP_t from eq. (7.8). Our implementation, however, uses the exact eq. (7.10) for m but inside $\mathcal{O}$ this does not matter and we continue with the simpler analysis using eq. (7.9):

$$m_Y \approx \frac{n_Y}{\ln(2)} \cdot \log_2 \frac{2\tilde{n}_\Delta}{FR} \qquad Y \in \{\mathtt{A}, \mathtt{B}\} \tag{7.17}$$

It is obvious that this is in $\mathcal{O}\left(n \cdot \log(\tilde{n}_\Delta / FR)\right)$ for $n, \tilde{n}_\Delta \to \infty,\ FR \to 0$ and the given overall complexity follows with the $C_{t'}$ analysis of Section 6.4.4.

7.5 Effective Worst-Case Accuracy

In contrast to the *trivial* and SHash reconciliation protocols, the size m of the Bloom filter is more fine-grained with regards to a targeted failure rate since the rounding of m influences the whole reconciliation and not each item. We thus expect the effective worst-case failure rate of phase 1 to closely follow the target rate. On the other hand, we use a common m and k for both Bloom filters, i.e. on nodes `A` and `B`, and may thus experience differences if two differently-sized sets are represented.

The following two sections present the actual m, k, and b'_k as well as the resulting effective worst-case failure rates $fr'_b(p_1)$, $fr'_b(p_2)$, and fr'_b for examples with FR being 10, 0.1, or 0.001 and $n = 100\,000$. The given values are based on the information the algorithm has at the time when parameters are determined, i.e. $n_\mathtt{A}$, $n_\mathtt{B}$, δ_{exp} and FR in phase 1 and $S'_\mathtt{B}$ and $S'_\mathtt{A}$ in phase 2, and are presented separately for the two scenarios, i.e. outdated and missing items. As in the SHash reconciliation, however, note that the *observed* failure rate in Section 7.7 may be lower than the effective worst-case failure rate presented here since the same upper bound on the number of false positive candidates $\tilde{n}_{\Delta_{X \in \{\mathtt{A},\mathtt{B}\}}}$ (eq. (7.6), page 94) is used with effectively contradicting worst-case assumptions (ref. Section 6.5).

7.5.1 Outdated Items Scenario

For the *outdated* scenario with $\delta = 1\,\%$ and $\delta = 10\,\%$ differences, Table 7.2 shows the various parameters and failure rates of an exemplary reconciliation with failures distributed equally among the two nodes and an assumed full identification of all differences in phase 1. Since m, k, and b'_k all need to be integral and thus do not allow arbitrary precision, target failure rates may not be reached or may—in rare cases—overshoot due to floating point issues as described above. Based on δ_{exp}, the upper bound on the number of false positive candidates $\tilde{n}_{\Delta_{\mathtt{A},\mathtt{B}}} := \tilde{n}_{\Delta_\mathtt{A}} = \tilde{n}_{\Delta_\mathtt{B}}$ is determined (eq. (7.6), page 94) and together with n and $FR(p_1)$, a target false positive rate FP_t is set which leads

to the given integral values of m and k. This FP_t is closely matched by the effective false positive rates $FP_{\mathtt{A}} = FP_{\mathtt{B}} =: FP_{\mathtt{A,B}}$ of the two Bloom filters which are equal in the *outdated* scenario due to $n_{\mathtt{A}} = n_{\mathtt{B}}$. The differences between $FP_{\mathtt{A,B}}$ and FP_t are lower than 0.01 % and lead to the effective worst-case failure rate $fr'_b(p_1)$ being very close to the target rate $FR(p_1)$ as anticipated.

Table 7.2: Bloom filter size m, hash functions count k and effective worst-case failure rates in the *outdated* scenario with $n = n_{\mathtt{A}} = n_{\mathtt{B}} = 100\,000$ and $\delta = \delta_{exp}$.

	$\delta = 1\,\%$			$\delta = 10\,\%$		
FR	0.001000	0.1000000	10.00000	0.001000	0.100000	10.00000
$\tilde{n}_{\Delta_{\mathtt{A,B}}}$	1 006	1 006	1 006	10 527	10 527	10 527
	phase 1					
$FR(p_1)$	0.000500	0.0500000	5.00000	0.000500	0.050000	5.00000
FP_t	$2.5{\cdot}10^{-7}$	$2.5{\cdot}10^{-5}$	$2.5{\cdot}10^{-3}$	$2.4{\cdot}10^{-8}$	$2.4{\cdot}10^{-6}$	$2.4{\cdot}10^{-4}$
k	22	15	9	25	19	12
m	3 165 310	2 207 058	1 248 878	3 654 191	2 695 726	1 736 994
$FP_{\mathtt{A,B}}$	$2.5{\cdot}10^{-7}$	$2.5{\cdot}10^{-5}$	$2.5{\cdot}10^{-3}$	$2.4{\cdot}10^{-8}$	$2.4{\cdot}10^{-6}$	$2.4{\cdot}10^{-4}$
$fr'_b(p_1)$	0.000500	0.0499998	4.99999	0.000500	0.050000	4.99999
	phase 2					
$\lvert S'_{\mathtt{A,B}}\rvert$	1 000			10 000		
$FR(p_2)$	0.000500	0.0500002	5.00001	0.000500	0.050000	5.00001
b_k	34	28	21	41	34	28
$fr'_b(p_2)$	0.000465	0.0297875	3.81097	0.000364	0.046564	2.98019
fr'_b	0.000965	0.0797873	8.81096	0.000864	0.096564	7.98019

Phase 2 thus uses a target failure rate $FR(p_2) \approx FR(p_1)$ and exhibits the usual deviations due to b'_k being integral: the effective worst-case failure rate $fr'_b(p_2)$ of phase 2 is at least half of $FR(p_2)$ (ref. Section 5.5). The overall effective worst-case failure rate fr'_b is in the range $[3/4 \cdot FR, FR]$ which is closer to FR than the *trivial* or SHash reconciliations above (see Sections 5.5 and 6.5).

These fluctuations of fr'_b can also be seen in Figure 7.4 which provides more details on its dependency on different n. Since the deviations from FR only depend on how many items enter phase 2, it exhibits the pattern of the *trivial'* reconciliation alone (ref. Section 5.5). This is also why different values of δ simply shift the function along the x-axis.

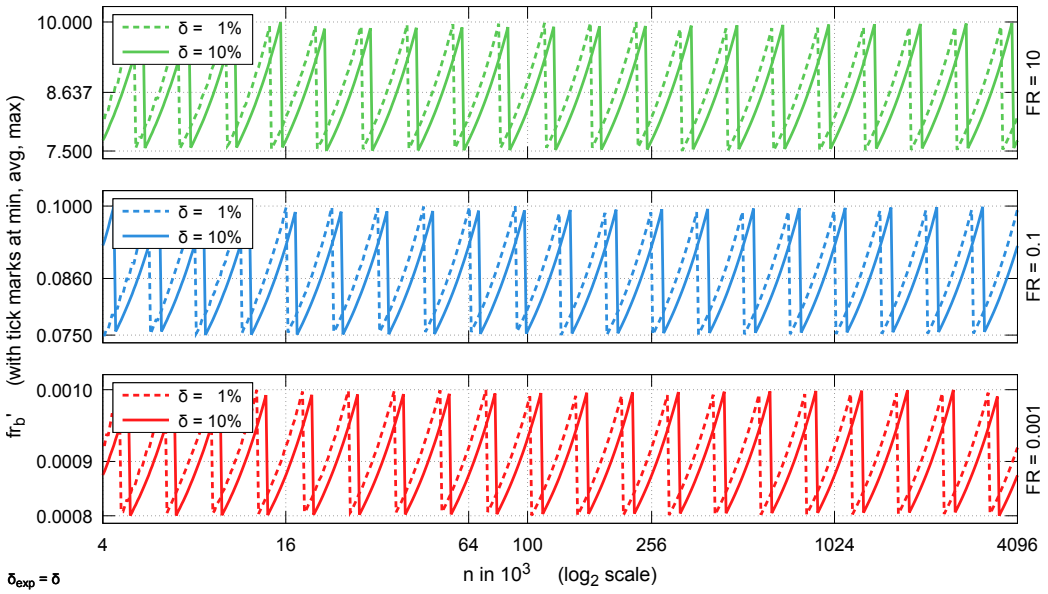


Figure 7.4: Bloom filter fr'_b in the *outdated* scenario for different FR, n, and $\delta = \delta_{exp}$.

7.5.2 Missing Items Scenario

Instead of showing the *missing* scenario with an equal distribution of failures to both nodes—which would just reflect the *trivial'* reconciliation of phase 2—, Table 7.3 and Figure 7.5 show a synchronisation between $S_\mathtt{A}$ and $S_\mathtt{B}$ where the δ percent differences are distributed so that node A misses $^3/_4$ of them and node B the remaining $^1/_4$. Therefore, the number of items in each Bloom filter is different and—due to the use of a common m and k—we expect some deviation in the false positive rate of one of them and thus in $fr'_b(p_1)$ as well.

In Table 7.3, $\delta = 1\,\%$ and $\delta = 10\,\%$ are shown with appropriately set $n_\mathtt{A}$ and $n_\mathtt{B}$. As shown, $\tilde{n}$ is estimated correctly but $\tilde{n}_{\Delta_{\mathtt{A,B}}} \coloneqq \tilde{n}_{\Delta_\mathtt{A}} = \tilde{n}_{\Delta_\mathtt{B}}$ is too large compared to the real number of unique items on each node, e.g. 250 on A with $\delta = 1\,\%$, due to the aforementioned effectively contradicting worst-case assumptions in the calculation of $\tilde{n}_{\Delta_{\mathtt{A,B}}}$. From these values, a common target false positive rate FP_t is determined (eq. (7.8), page 95) and the optimal values $k_{X\in\{\mathtt{A,B}\}}$ and $m_{X\in\{\mathtt{A,B}\}}$ are determined for each node.

After selecting the best common m and k to use for both Bloom filters (eq. (7.12), page 96), the individual false positive rates as well as the effective worst-case failure rate of phase 1 can be calculated. Since m is the optimal choice for at least one of the Bloom filters, one of them, i.e. $\mathtt{BF}(S_\mathtt{B})$ here, has a false positive rate close to the target rate FP_t. Conversely, the other one shows some deviation which results in $fr'_b(p_1)$ not being as close to $FR(p_1)$ as with equally distributed failures, e.g. in the *outdated* scenario above (see Table 7.2).

The remaining failure rate may be used in phase 2 which operates on differently-sized sets from the identified differences of phase 1, i.e. $S'_\mathtt{A}$ and $S'_\mathtt{B}$,

Table 7.3: Bloom filter size m, hash functions count k and effective worst-case failure rates in the *missing* scenario with $\delta = \delta_{exp}$, $n = 100\,000$, and 3/4 of the difference missing on node A ($n_{\mathtt{A}}, n_{\mathtt{B}}$ set accordingly).

	$\delta = 1\,\%$			$\delta = 10\,\%$		
FR	0.001000	0.1000000	10.00000	0.001000	0.100000	10.00000
$n_{\mathtt{A}}$	99 250	99 250	99 250	92 500	92 500	92 500
$n_{\mathtt{B}}$	99 750	99 750	99 750	97 500	97 500	97 500
$\tilde{n}$	100 000	100 000	100 000	100 000	100 000	100 000
$\tilde{n}_{\Delta_{\mathtt{A,B}}}$	1 000	1 000	1 000	10 000	10 000	10 000
	phase 1					
$FR(p_1)$	0.000500	0.0500000	5.00000	0.000500	0.050000	5.00000
FP_t	$2.5{\cdot}10^{-7}$	$2.5{\cdot}10^{-5}$	$2.5{\cdot}10^{-3}$	$2.5{\cdot}10^{-8}$	$2.5{\cdot}10^{-6}$	$2.5{\cdot}10^{-4}$
$k_{\mathtt{A}}$	22	15	9	25	19	12
$m_{\mathtt{A}}$	3 140 336	2 189 254	1 238 305	3 370 168	2 483 771	1 596 830
$k_{\mathtt{B}}$	22	15	9	25	19	12
$m_{\mathtt{B}}$	3 156 157	2 200 283	1 244 543	3 552 339	2 618 028	1 683 146
k	22	15	9	25	19	12
m	3 156 157	2 200 283	1 244 543	3 552 339	2 618 028	1 683 146
$FP_{\mathtt{A}}$	$2.3{\cdot}10^{-7}$	$2.4{\cdot}10^{-5}$	$2.4{\cdot}10^{-3}$	$9.9{\cdot}10^{-9}$	$1.2{\cdot}10^{-6}$	$1.6{\cdot}10^{-4}$
$FP_{\mathtt{B}}$	$2.5{\cdot}10^{-7}$	$2.5{\cdot}10^{-5}$	$2.5{\cdot}10^{-3}$	$2.5{\cdot}10^{-8}$	$2.5{\cdot}10^{-6}$	$2.5{\cdot}10^{-4}$
$fr'_b(p_1)$	0.000482	0.0487169	4.92394	0.000349	0.037479	4.10717
	phase 2					
$\lvert S'_{\mathtt{A,B}}\rvert$	$\lvert S'_{\mathtt{A}}\rvert = 250$, $\lvert S'_{\mathtt{B}}\rvert = 750$			$\lvert S'_{\mathtt{A}}\rvert = 2\,500$, $\lvert S'_{\mathtt{B}}\rvert = 7\,500$		
$FR(p_2)$	0.000518	0.0512831	5.07606	0.000651	0.062521	5.89283
b_k	32	26	19	39	32	26
$fr'_b(p_2)$	0.000465	0.0297723	3.80726	0.000364	0.046562	2.97971
fr'_b	0.000947	0.0784892	8.73120	0.000713	0.084041	7.08689

which we assume to be correct, here. Although the size of CKV depends on the number of items in $S'_{\mathtt{B}}$ only, b_k and the effective worst-case failure rate only depend on the number of unique keys (ref. eqs. (5.5) and (5.6), pages 56 and 57) and are thus not influenced by the different distribution of δ. Therefore, we expect the usual fluctuations of the *trivial'* protocol as above.

All together, due to the inaccuracies in phase 1, the overall effective worst-case failure rate here is lower on average than with equally distributed failures as shown by Figure 7.5 compared to Figure 7.4 above. The higher the δ, the higher the absolute differences in the item counts and the higher the difference of the target failure rate and the effective worst-case failure rate of phase 1. Although we re-use this difference in the target failure rate of phase 2, it can only compensate so much and—for $\delta = 10\,\%$ compared to $\delta = 1\,\%$—we already

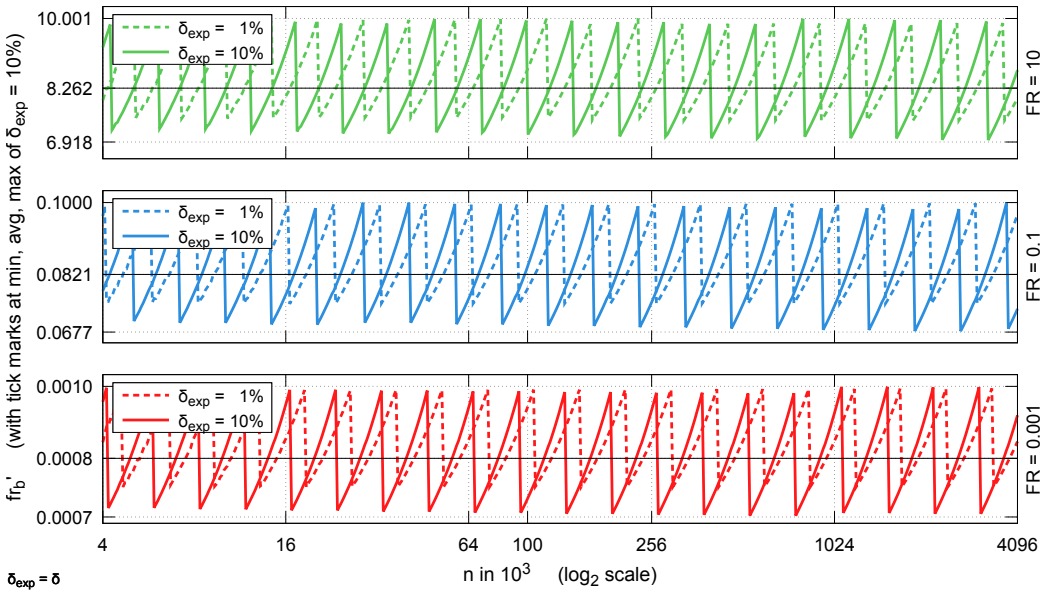


Figure 7.5: Bloom filter fr'_b in the *missing* scenario for different FR, n, and $\delta = \delta_{exp}$ with $^3/_4$ of the difference missing on node `A`.

see a bend in the ascend from the minimum to the maximum value of fr'_b and some fluctuations in reaching these two extremes. The higher the inequality in the number of items between the nodes, the more fr'_b will degrade to a distribution similar to the SHash protocol (ref. Figure 6.3, page 80).

7.6 Related Work

For an extensive overview of research on various Bloom filter variants improving compression rates, reducing maintenance overhead, or reducing some of their limitations, please refer to Luo et al. [51] and Tarkoma et al. [78] who extend a previous survey of Broder and Mitzenmacher [8]. These surveys also discuss related work with regards to hash function choice and several applications of the respective Bloom filter variants. To show the basic application of our accuracy model, however, we used the most-widely spread original `BF` in the presented set reconciliation algorithm, also because we focus on accuracy and network costs and ignore construction time. Nevertheless, our technique to fairly compare approximate set reconciliation methods by enforcing a fixed accuracy can be applied to any Bloom filter variant if desired.

Among these variants, Mitzenmacher [56] re-evaluates the theory around Bloom filters and the selection of appropriate parameters when a compression of the m bit array is allowed and tunes k for optimal *compressed* size. In practice, his *Compressed Bloom filters* achieve a 5 %–15 % reduction of transmission size. The un-compressed size, however, may grow to $m = {}^n/_{FP}$ which may make it unsuitable in bigger systems. Most other `BF` variants add support for

counting or deleting elements which we both do not need to identify missing or outdated key-value pairs but may be helpful in the maintenance of the Bloom filter structure. In our case, a Bloom filter which represents a multi-set (for key and version) would be preferable. Existing approaches like filter banks [15] or Bloomier filters [16] are, however, not as bandwidth friendly as the original Bloom filter. Filter banks and Bloomier filters use similar techniques and put an item into one of several Bloom filters based on the value associated with it. They are thus only suitable for small value spaces.

Invertible Bloom lookup tables (IBLT) [34] also associate a value with each key. They were discussed intensively in Section 2.3.6 along with invertible Bloom filters. Similarly, set reconciliation with counting Bloom filters was presented in Section 2.3.5 above. Both algorithms seem promising candidates for approximate set reconciliation but face their individual challenges discussed in their respective sections.

7.7 Evaluation

Bloom filter reconciliation uses a single binary of size m and sets k bits per item to 1. These bits may overlap and thus save bandwidth but also remove the ability to identify unmatched items from $Mis_{\mathtt{A}}$. Therefore, *both* nodes exchange Bloom filters with each other which we optimise by sending the second one as an XOR-ed `Diff-BF` for efficient compression with low δ.

During Bloom filter member checks, false negatives do not occur, false positives according to the probability FP. In the overall reconciliation, these false positives may become either unidentified differences, i.e. for missing items or if both versions of an item are false positives, or redundant item transfers, i.e. if the outdated item is sent (ref. Section 7.3.1).

7.7.1 General Analysis for Different δ and FR

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand}$

Figures 7.6 and 7.7 show the results for $\delta \in (0, 10]\,\%$ and $\delta \in (0, 100]\,\%$, respectively. The total number of errors is roughly only 1/3 to 1/2 as high as the configured FR which is partly due to (a) $\tilde{n}_{\Delta_{X \in \{\mathtt{A},\mathtt{B}\}}}$ being too high and thus phase 1 ending up with an—undetectedly—too high accuracy, (b) in the *outdated* scenario only half of the false positives leading

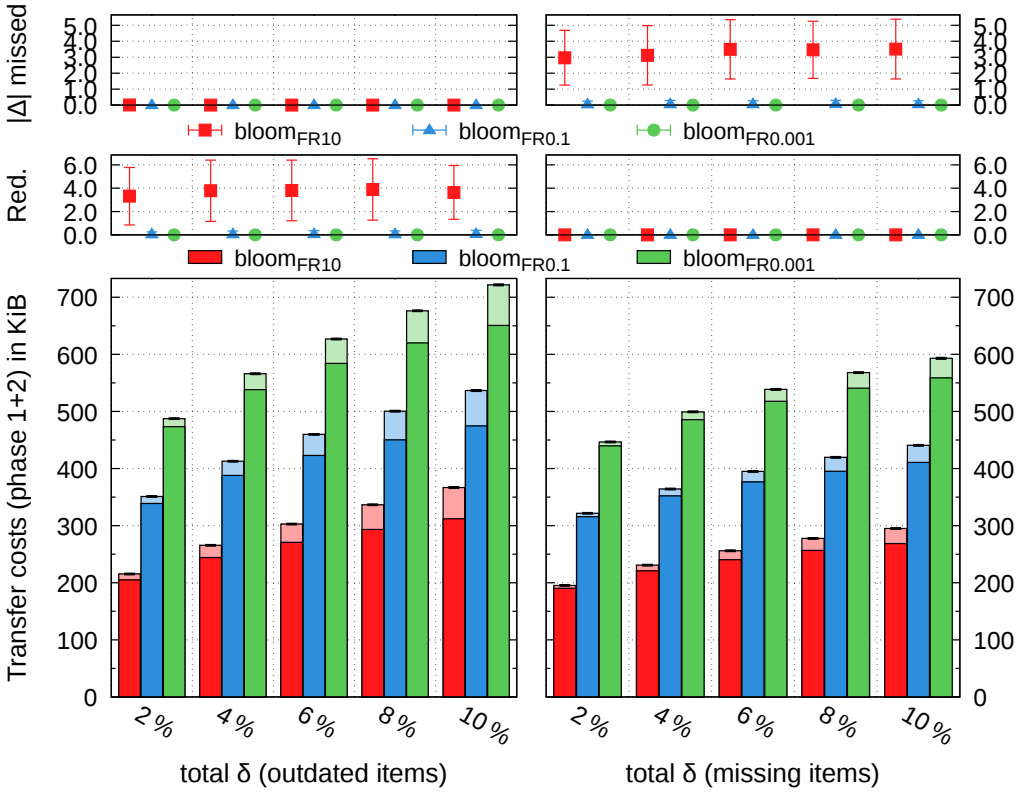


Figure 7.6: Bloom filter reconciliation with small δ and FR.

to redundant item transfers (ref. Section 7.3.1), and (c) phase 2 not getting as close to $FR(p_2)$ as desired due to b_k being integral. Additionally, and with regards to the different failure types, please recall that phase 2 does not operate on any extraneous common items and thus, there, any multiple collision in the *missing* scenario leads to no errors and in the *outdated* scenario to redundant item transfers of twice the number of colliding keys, as accounted for. The more unlikely case of a single collision only results in failures in the *missing* scenario and there, only one unrecognised item of Δ is most likely (ref. Section 5.3.1). Therefore, in the *missing* scenario, there may not be redundant item transfers. In the *outdated* scenario, however, unidentified differences only occur if both versions of an item are false positives in the respective Bloom filters and most errors are redundant item transfers.

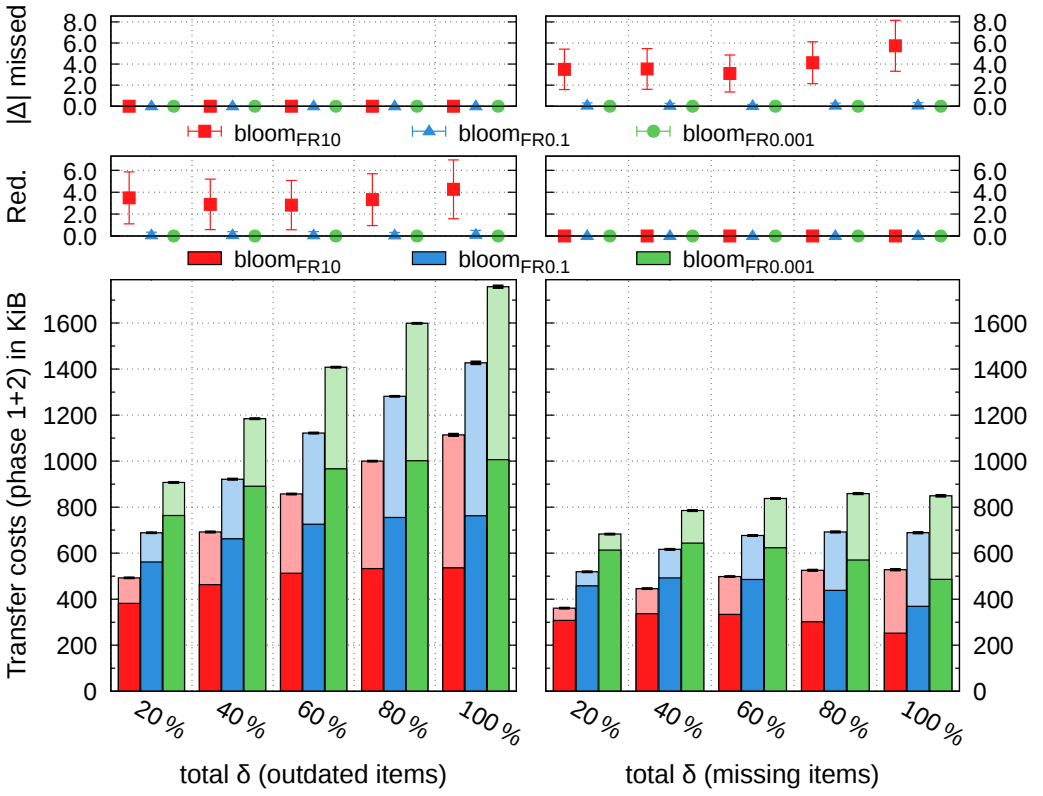


Figure 7.7: Bloom filter reconciliation with high δ and FR.

The size of the Bloom filter `BF` in phase 1 depends on n, $\tilde{n}_{\Delta_{X\in\{\mathtt{A},\mathtt{B}\}}}$, and FR (ref. eq. (7.13), page 96). It thus increases with δ_{exp} in the *outdated* scenario and decreases with δ in the *missing* scenario due to fewer items on the nodes. The size of the Bloom filter `Diff-BF`, however, has `BF`'s size as an upper bound but its compressibility depends on δ as well. Both sizes sum up to the phase 1 costs but the plots also roughly indicate each of them: $|\mathtt{BF}|$ in the phase 1 costs for $\delta = 2\,\%$ and $|\mathtt{BF}| + |\mathtt{Diff\text{-}BF}|$ for $\delta = 100\,\%$, due to the Bloom filter only growing logarithmically with $\tilde{n}_{\Delta_{X\in\{\mathtt{A},\mathtt{B}\}}}$ and thus δ_{exp} (ref. eq. (7.16), page 97).

The logarithmic growth of the combined phase 1 costs as well as the overall costs of phase 1 and 2 can also be seen in the plots, just as an indication of the logarithmic increase with FR^{-1} which is detailed in Section 7.7.5 below.

7.7.2 What if δ_{exp} is Wrong?

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$ vs. 1 %
$fail_{rand}$

In the Bloom filter reconciliation protocol, δ_{exp} is used to estimate the number of false positives candidates $\tilde{n}_{\Delta_{X\in\{A,B\}}}$ in phase 1 on which the false positive probability target FP_t depends (eq. (7.8), page 95). If δ_{exp} is too high, Bloom filters sizes will be larger than needed and the resulting FP will be lower which reduces the number of errors. If δ_{exp} is too low, FP will be higher than needed, Bloom filter sizes will reduce, and more failures will occur, as shown by Figure 7.8. Phase 2 is not directly affected by δ_{exp} except for items not entering this phase and thus slightly reducing its costs.

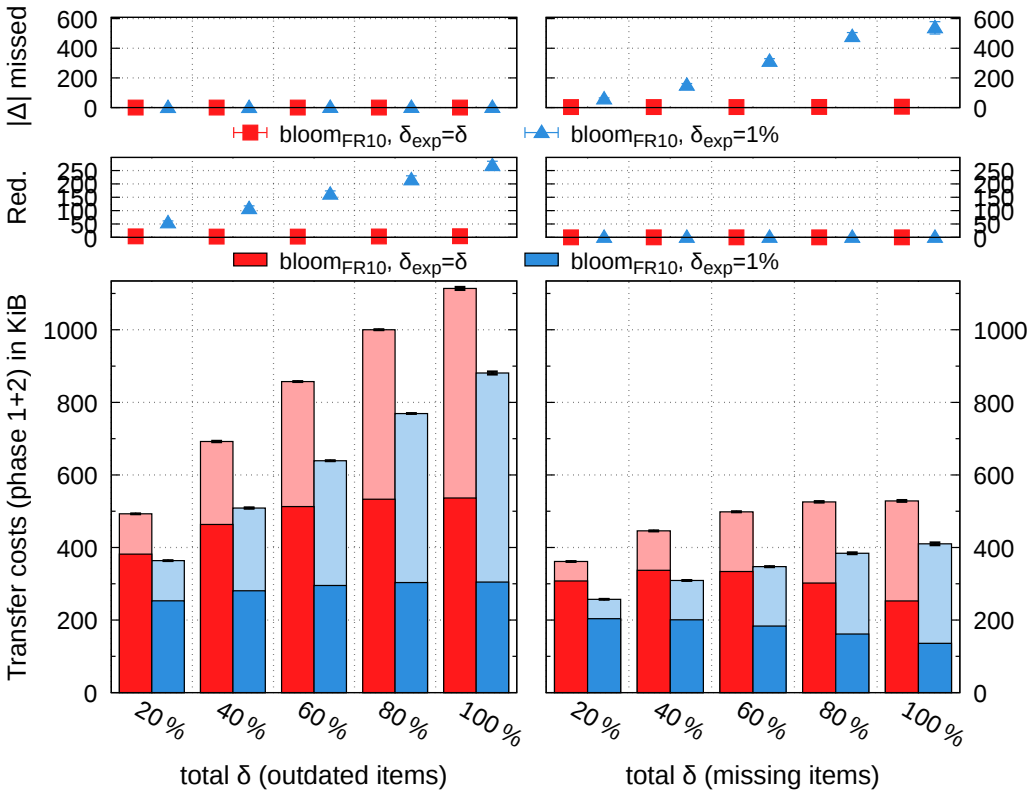


Figure 7.8: Bloom filter reconciliation with high δ and different δ_{exp}.

The extend of the increased numbers of failures can be seen by calculating $\tilde{n}_{\Delta_{X\in\{A,B\}}}$ (eq. (7.6), page 94): in the *outdated* scenario, for $\delta = 100\,\%, \delta_{exp} = 1\,\%$, $\tilde{n}_{\Delta_X}$ is 1 006 while the real n_{Δ_X} is 100 000. Similarly, in the *missing* scenario, $\tilde{n}_{\Delta_X} = 503$ vs. $n_{\Delta_X} = 50\,000$. Also recall that the number of failures in the *missing* scenario is higher than in the *outdated* scenario because the latter only

has a redundant item transfer in roughly half of the cases of a false positive (ref. Section 7.3.2). This was not clearly visible in the section above due to the influence of phase 2 while phase 1 errors are more prominent here.

7.7.3 Data and Failure Distribution Sensitivity

$n = 100\,000$
$data_{rand,\ bin\,0.2}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand,\ bin_{0.2}}$

Due to the hashing, Bloom filters should not be sensitive to either data or failure distribution. The same applies to phase 2 since it already operates on $\Delta' \subseteq \Delta$ and uses hashing, too (also ref. Section 5.6.3), which eliminates any effects of a different data or failure distribution. Figure 7.9 confirms this behaviour for both the failures as well as the costs.

Regarding the costs, please recall that the Bloom filter size m does not depend on the data (or failure) distribution assuming perfectly random hash functions. `Diff-BF` may, however, allow a better compression if the bits of the different items cluster. Apparently, though, by using more hash functions than in the previous algorithms, the bits are spread more uniformly (independent of the data or failure distribution) and thus do not influence the zlib compression in the transfer costs.

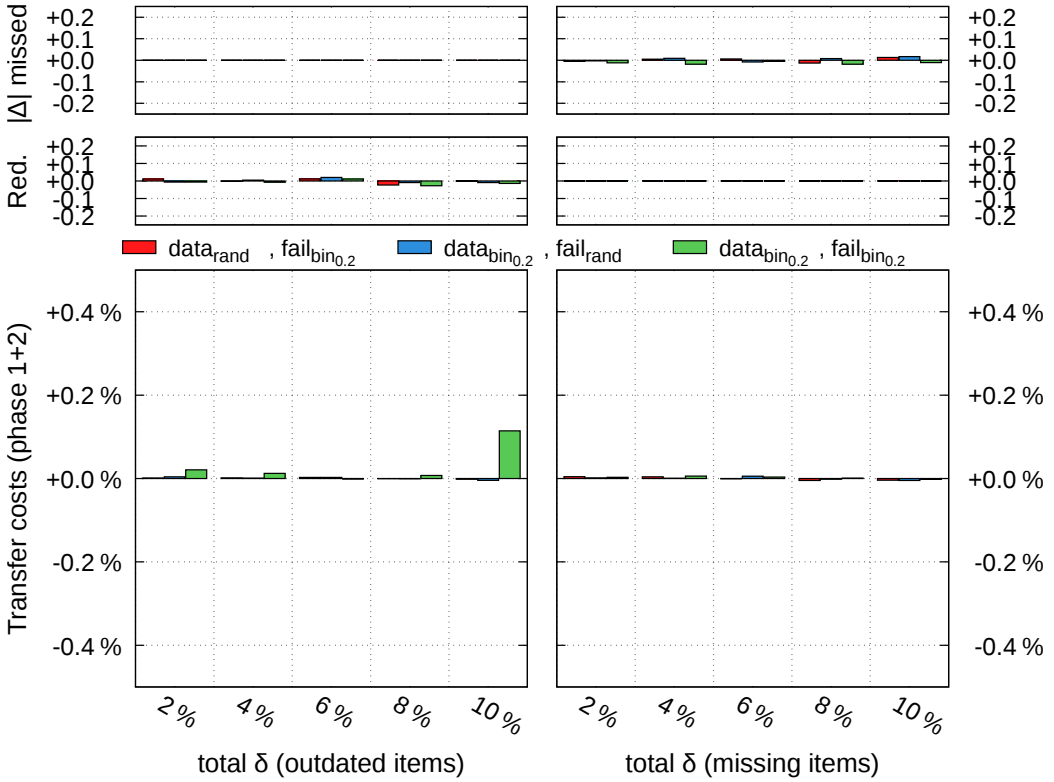


Figure 7.9: Bloom filter reconciliation with $FR = 0.1$ and different data and failure distributions compared to $data_{rand}, fail_{rand}$.

7.7.4 Scalability with the System Size n

$n = variable$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 7.10 shows the results of the Bloom filter reconciliation with smaller and larger numbers of items than above and a fixed $\delta = 3\,\% =: \delta_{exp}$. As expected, the accuracy is unaffected by the number of items and the transfer costs increase with $\mathcal{O}(n \cdot \log n)$ for $n \to \infty$, $|\Delta|, \tilde{n}_\Delta \in \mathcal{O}(n)$ and FR constant (ref. eq. (7.16), page 97). Similarly to the *trivial* and SHash reconciliations, Bloom transfer costs keep approaching the naïve transfer costs—which are in $\mathcal{O}(n)$—with higher n. Please refer to Section 5.6.4 for a discussion of this effect.

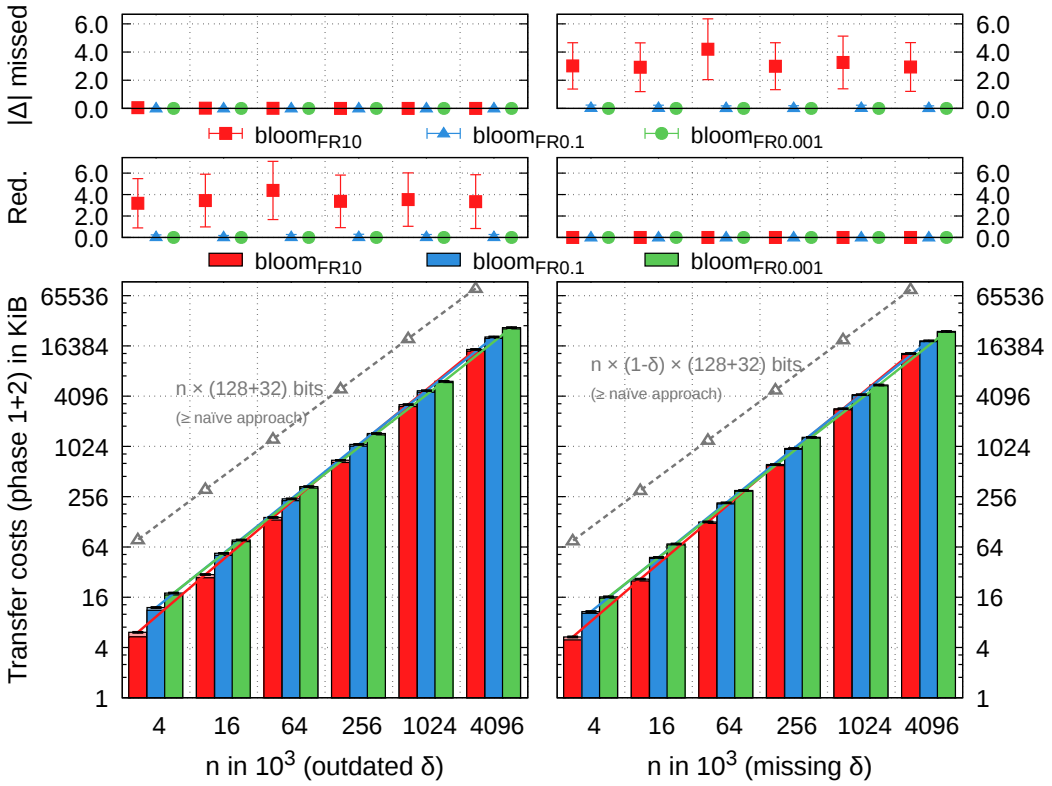


Figure 7.10: Bloom filter reconciliation scalability with data size n and different FR.

7.7.5 Scalability with the Target Failure Rate FR

$n = 100\,000$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

For fixed n, Figure 7.11 shows how the actual failures and the communication costs of the Bloom filter reconciliation evolve with different target failure rates FR. While the number of failures for lower values of FR is not representative for our 1 000 random simulations, higher values are and the costs are as well. The results are inconspicuous with regards to the communication costs complexity of $\mathcal{O}(\log(1/FR))$ for n and $|\Delta|$ constant (ref. eq. (7.16), page 97) as well as the progression of the redundant item transfers in the *outdated* scenario and the unidentified differences in the *missing* scenario. With an increased FR, however, we are also able to observe some unidentified differences in the *outdated* scenario for which the probability is very low in both phases of the reconciliation (ref. Section 7.3.1).

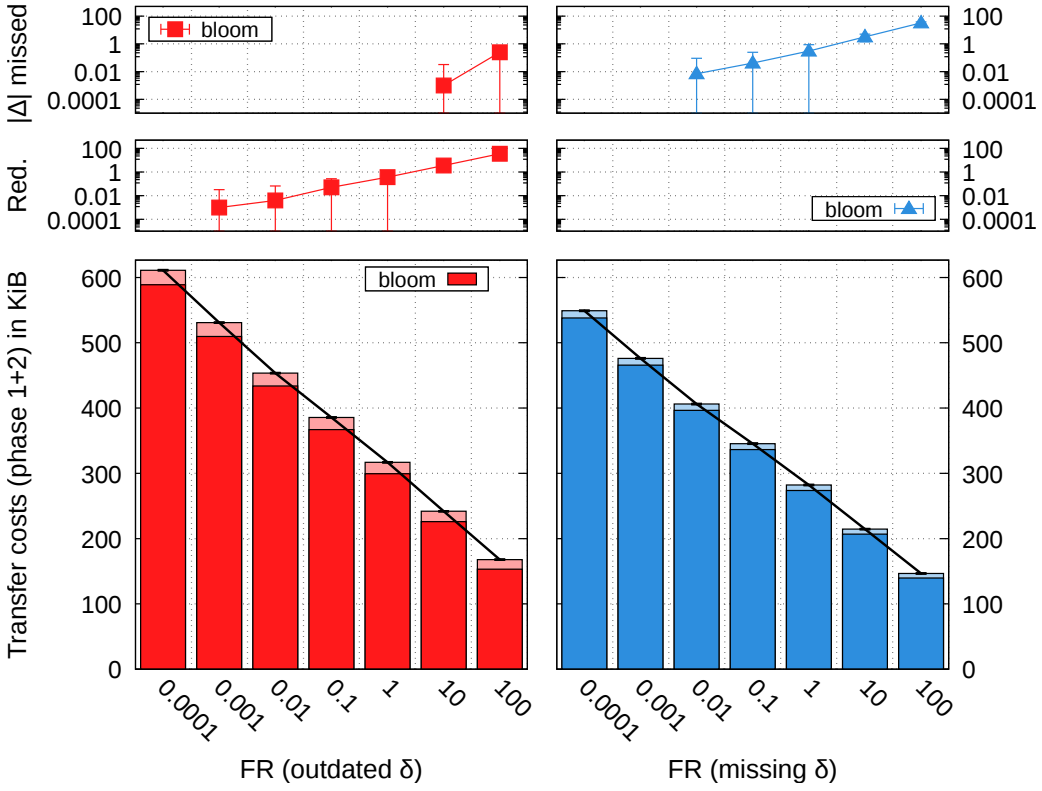


Figure 7.11: Bloom filter reconciliation scalability with the target failure rate FR ($\log_{10}$ scale on the x-axis and on the y-axis except for the transfer costs).

Chapter 8

Merkle Tree

In a Merkle tree [52], leaf nodes contain items and are labelled with the hash of the data they represent. Inner nodes group leaf nodes or other inner nodes and are labelled with the hash of the labels of their children. This allows an effective comparison of two distributed data sets with a recursive top-down algorithm by comparing the labels of these sets' Merkle trees [10, 14].

8.1 Protocol

In contrast to the algorithms above, Merkle reconciliation is a multi-round protocol with a dynamic number of rounds (Figure 8.1). At first, node B creates a Merkle tree of S_B and tells A to create one of S_A with the same parameters. The following synchronisation loops over the levels of the Merkle tree, starting with the root node: A sends the hashes of all relevant nodes in the current level, i.e. CH(cur lvl), to B which compares them with its own hashes at this level and returns the results. Any hash mismatch identifies some sort of difference and determines the tree nodes to continue with in the next iteration of the loop.

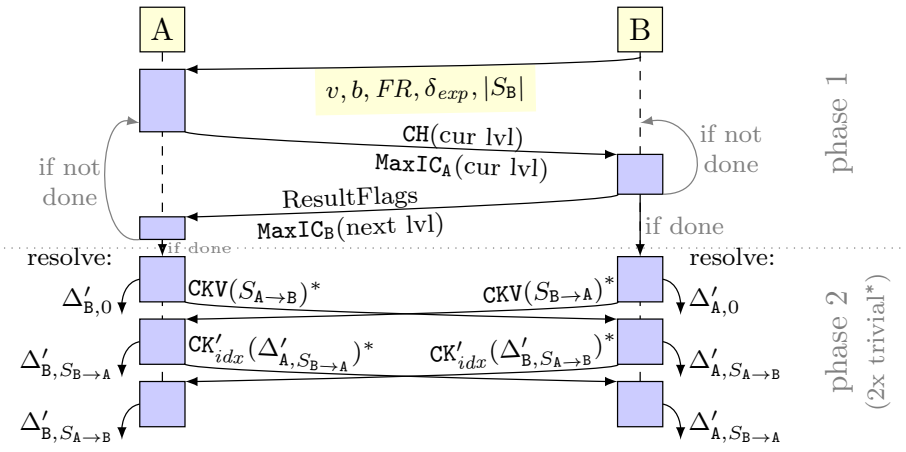


Figure 8.1: *Merkle tree* reconciliation protocol.

In phase 2, the differing items in the remaining tree nodes are identified with slightly modified *trivial* reconciliations, i.e. *trivial**, starting at `A` or `B` depending on the difference type which we detail later on. These two *trivial** reconciliations operate on different Merkle tree nodes and run in parallel. They will eventually identify the items to push.

8.2 Building Merkle Trees

A Merkle tree for reconciliation uses key-version pairs of a node's items and puts them into buckets of sizes up to $b \geq 1$. Each leaf node represents a single (potentially empty) bucket. Let $v \geq 2$ be the degree of the tree, i.e. the number of sub-trees of an inner node. We build a Merkle tree over an (uncollapsed) Patricia Trie [43] similar to Byers et al. [10] but omit the first re-hashing phase for a pristine Merkle tree (see below). Note that if the data is uniformly distributed, a pristine Merkle tree is similarly balanced as a Merkle tree with an initial re-hashing phase. We are thus able to provide a more generic evaluation of Merkle trees by omitting the re-hashing phase.

Algorithm 13 Merkle tree creation

```
function BuildMerkle(I, v, b)                    ▷ consecutive interval I of items
    if I.items.size( ) > b then
        MT ← NewInnerNode( )
        for all I' ∈ Split(I, v) do              ▷ for v equally long parts of I (sorted)
            child ← BuildMerkle(I', v, b)
            MT.AddChild(child)                   ▷ add child to the children of node MT
        end for
        MT.CalcInnerHash( )                      ▷ XOR of the children's hashes
    else
        MT ← NewLeafNode( )
        MT.AddItems(I.items)                     ▷ add items to the bucket of node MT
        sort(MT.bucket)                          ▷ sort by the items' keys
        MT.CalcLeafHash( )                       ▷ SHA-1 of the sorted item list
    end if
    return MT
end function
```

The top-down construction of such a Merkle tree is depicted in Algorithm 13. We start with the interval of keys the Merkle tree is built for and define a recursive algorithm: as long as the current interval contains more than b items, split it into v equidistant sub-intervals and build a Merkle (sub-) tree for each of these as children of the current node using all the items within the sub-interval. Note that in case of integer keys it is sufficient that the sub-intervals cover *nearly* the same number of keys, i.e. $\lfloor |I|/v \rfloor$ or $\lceil |I|/v \rceil$. If the number of items is less than or equal to b, a leaf node is created. Finally, the hashes are created bottom-up: At a leaf node, the bucket is hashed using SHA-1 on the binary

representation of the ordered list of key-version pairs in the bucket. Inner nodes XOR the hashes of their children for their own label [10].

Figure 8.2 shows an exemplary Merkle tree which results from data items with keys $k \in \{5, 10, 21, 26, 69, 74, 85, 101, 106, 122\}$ and version 1 in the synchronisation interval $I = (0, 128]$ with $v = 2$ and $b = 2$. We use this example for node `A` and omit the labels, i.e. hashes, for now.

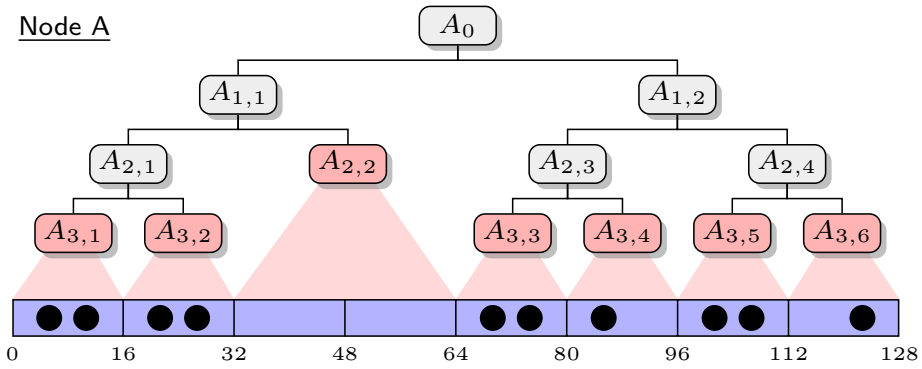


Figure 8.2: Exemplary binary *Merkle tree* ($v = 2$, $b = 2$) of node `A`.

8.3 Set Reconciliation with Merkle Trees

Before we go through the Merkle synchronisation protocol in detail, please note that for reasons which will become clear below, we tag each hash from `A` as being an inner hash (0) or a leaf hash (1) by adding one bit. For each leaf hash, we add one more bit to indicate an empty leaf (0) or not (1). In addition to these status bits, non-empty leaf hashes use S_ℓ bits during the synchronisation, inner hashes use S_i bits. Both are constant for all hashes of a tree level but may vary in different levels. Empty leaves only transmit the two status bits.

8.3.1 Phase 1

Phase 1 of the Merkle synchronisation protocol is depicted in Algorithm 14 and loops over the levels of the Merkle tree, starting with the root node. Node `A` sends its tree nodes' hashes to node `B` which compares them with its own hashes and responds with the 2-bit representations of the result codes of Table 8.1. Hash mismatches identify nodes with differences and the algorithm continues into the next Merkle tree level with only these tree nodes' children until no further mismatches occur or no tree nodes are left.

Note that some of the result codes' bit patterns of Table 8.1 are non-unique but both nodes can distinguish between all cases based on their local information and the status flags, e.g. bit pattern `11` can be distinguished at `A` by its own tree node type. Also note that the combination of an empty leaf node at both nodes `B` and `A` is always a match and thus not a candidate for mismatches.

Algorithm 14 Basic Merkle tree synchronisation protocol (phase 1)

```
function MerkleSyncA(MT_A)        ▷ executed on node A with Merkle tree MT_A
    S_{B→A}, S_{A→B}, Δ'_A, Δ'_B ← [], MN ← [MT_A.root]        ▷ MN = current node list
    repeat
        SendHashes(MN)                          ▷ encode & send Merkle tree hashes
        BMsg ← receive(result message from B)                    ▷ wait for msg
        for all code ∈ DecodeResponse(BMsg) do                   ▷ decode result bits
            N ← MN.PopFirst( )          ▷ remove and get first element in MN
            if code = cont_inner then   ▷ continue MerkleSync with all children
                MN.AppendAll(N.children)        ▷ append all children to the end
            else if code = stop_inner then
                S_{A→B}.Append(N)                                ▷ add N to the end
            else if code = stop_leaf then
                S_{B→A}.Append(N)                                ▷ add N to the end
            else if code = stop_empty_leaf3 or stop_empty_leaf4 then
                Δ'_B.Append(N.bucket)                  ▷ add items of N to the end
            end if                          ▷ drop nodes for all other result codes
        end for
    until MN = []
end function

function MerkleSyncB(MT_B)        ▷ executed on node B with Merkle tree MT_B
    S_{B→A}, S_{A→B}, Δ'_A, Δ'_B ← [], MN ← [MT_B.root]        ▷ MN = current node list
    repeat
        AMsg ← receive(message from A)                           ▷ wait for msg
        Response ← empty bitstring
        for all hash_A ∈ DecodeHashes(AMsg) do
            N ← MN.PopFirst( )          ▷ remove and get first element in MN
            code ← Compare(N, hash_A)   ▷ determine result code from Table 8.1
            if code = cont_inner then   ▷ continue MerkleSync with all children
                MN.AppendAll(N.children)        ▷ append all children to the end
            else if code = stop_inner then
                S_{A→B}.Append(N)                                ▷ add N to the end
            else if code = stop_leaf then
                S_{B→A}.Append(N)                                ▷ add N to the end
            else if code = stop_empty_leaf1 or stop_empty_leaf2 then
                Δ'_A.Append(N.bucket)                  ▷ add items of N to the end
            end if                          ▷ drop nodes for all other result codes
            Response.AddBits(code)
        end for
    until MN = []
end function
```

Table 8.1: Result codes and resolve sets for the different cases during Merkle tree node hash comparisons (✓= match, ↯= mismatch).

	node type at B	node type at A	set	code	bits
✓	$any_{\mathtt{B}}$	$any_{\mathtt{B}}$	-	`ok`	01
↯	$inner_{\mathtt{B}}$	$inner_{\mathtt{A}}$	-	`cont_inner`	00
↯		*non-empty* $leaf_{\mathtt{A}}$	$S_{\mathtt{A}\to\mathtt{B}}$	`stop_inner`	11
↯		*empty* $leaf_{\mathtt{A}}$	$\Delta'_{\mathtt{A},0}$	`stop_empty_leaf1`	11
↯	*non-empty* $leaf_{\mathtt{B}}$	$inner_{\mathtt{A}}$	$S_{\mathtt{B}\to\mathtt{A}}$	`stop_leaf`	10
↯		*non-empty* $leaf_{\mathtt{A}}$	$S_{\mathtt{B}\to\mathtt{A}}$	`stop_leaf`	10
↯		*empty* $leaf_{\mathtt{A}}$	$\Delta'_{\mathtt{A},0}$	`stop_empty_leaf2`	10
↯	*empty* $leaf_{\mathtt{B}}$	$inner_{\mathtt{A}}$	$\Delta'_{\mathtt{B},0}$	`stop_empty_leaf3`	11
↯		*non-empty* $leaf_{\mathtt{A}}$	$\Delta'_{\mathtt{B},0}$	`stop_empty_leaf4`	00

Example Synchronisation

We will now go through phase 1 of an exemplary synchronisation of two nodes `A` and `B` to make things clearer. Let node `B` have almost the same items as node `A` from Figure 8.2, with the following exceptions: item 10 has version 2, item 26 is missing, and there are additional items at 37, 42, and 53. The resulting Merkle tree and the differences in the items as well as in the Merkle tree hashes are shown by Figure 8.3.

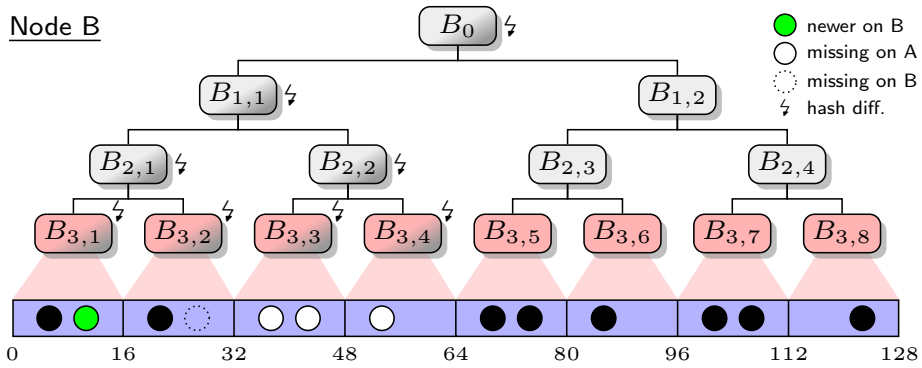


Figure 8.3: Exemplary binary *Merkle tree* ($v = 2$, $b = 2$) of node `B` with differences indicated.

The synchronisation between `A` and `B` (Figure 8.4) starts with `A` sending its root hash A_0 to `B`. Since $A_0 \neq B_0$ (↯) and both are inner nodes, `B` returns `cont_inner` and `A` sends all hashes from its children, i.e. $A_{1,1}, A_{1,2}$, to `B`. Since $A_{1,2} = B_{1,2}$, we assume that the trees below these nodes are equal and do not require further synchronisation. `B` thus returns `cont_inner` for $A_{1,1}/B_{1,1}$ and `ok` for $A_{1,2}/B_{1,2}$. `A` continues by sending the $A_{2,1}$ hash and, for $A_{2,2}$, the bits indicating an empty leaf (`10`). Both are mismatches but $A_{2,2}$ is an empty leaf hash which can be directly resolved with items from $S_{\mathtt{B}}$. Thus, any items below $B_{2,2}$ are added to $\Delta'_{\mathtt{A}}$ and `stop_empty_leaf1` is returned, together with the

`cont_inner` for the hash mismatch of $A_{2,1}$ and $B_{2,1}$. Finally, `A` sends $A_{3,1}$ and $A_{3,2}$ which both turn out as non-empty leaf mismatches at `B` (`stop_leaf`) and are added to $S_{\mathtt{B}\to\mathtt{A}}$ on both nodes for the following *trivial** reconciliation.

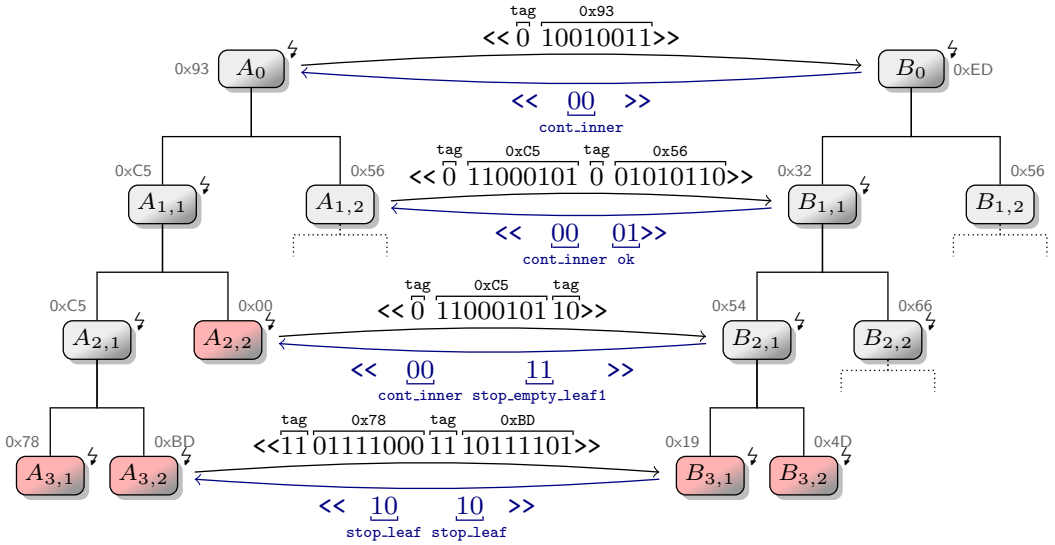


Figure 8.4: Merkle tree sync example based on the trees from Figures 8.2 and 8.3 with exemplary hash labels, inner/leaf node tags as described, result codes from Table 8.1, and fixed hash sizes $S_\ell = S_i = 8\,\text{bit}$.

8.3.2 Phase 2

After phase 1, both nodes already know some parts of the resolve sets from empty leaf mismatches, i.e. $\Delta'_{\mathtt{A},0} \subseteq \Delta'_{\mathtt{A}}$ and $\Delta'_{\mathtt{B},0} \subseteq \Delta'_{\mathtt{B}}$, which are resolved immediately (ref. Figure 8.1 and Table 8.1, pages 111 and 115). Additionally, the two sets $S_{\mathtt{A}\to\mathtt{B}}$ and $S_{\mathtt{B}\to\mathtt{A}}$ with non-empty leaves at `A` and `B`, respectively, are known to both nodes from which the rest of $\Delta'_{\mathtt{A}}$ and $\Delta'_{\mathtt{B}}$ is extracted in *phase 2*. The following two parallel *trivial** reconciliations are based on the *trivial'* algorithm from Section 6.3.1—a variation of the *trivial* algorithm of Chapter 5: we basically spawn a *trivial'* reconciliation for each non-empty leaf node's bucket and the corresponding items at the other node with additional changes to improve the efficiency with respect to this multiplicity.

`B` creates a compressed key-value binary $\mathtt{CKV}(S_{\mathtt{B}\to\mathtt{A}})^*$ from the individual `CKV` structures of nodes in $S_{\mathtt{B}\to\mathtt{A}}$ (details below) and, analogously, `A` creates $\mathtt{CKV}(S_{\mathtt{A}\to\mathtt{B}})^*$. Then, each $\mathtt{CKV}(\ldots)^*$ is sent to the opposite node. Since `A` and `B` both know both tree node sets, after receiving $\mathtt{CKV}(\ldots)^*$, they extract the original `CKV` structures, map them to appropriate Merkle tree nodes, and execute the ordinary *trivial* reconciliation (Algorithm 8, page 53) in the according key range. In this first step, $\Delta'_{\mathtt{B},S_{\mathtt{B}\to\mathtt{A}}} \subseteq \Delta'_{\mathtt{B}}$ from items in $S_{\mathtt{B}\to\mathtt{A}}$ and $\Delta'_{\mathtt{A},S_{\mathtt{A}\to\mathtt{B}}} \subseteq \Delta'_{\mathtt{A}}$ from items in $S_{\mathtt{A}\to\mathtt{B}}$ may be resolved by the two reconciliations.

As usual, in the second step of these *trivial** reconciliations, on each node, all unmatched keys as well as keys from newer items from $\texttt{CKV}(\dots)^*$ are requested via $\texttt{CK}'_{idx}(\dots)^*$ in order to resolve the remaining $\Delta'_{\texttt{A},S_{\texttt{B}\to\texttt{A}}} \subseteq \Delta'_{\texttt{A}}$ and $\Delta'_{\texttt{B},S_{\texttt{A}\to\texttt{B}}} \subseteq \Delta'_{\texttt{B}}$ from items of Merkle tree nodes in $S_{\texttt{B}\to\texttt{A}}$ and $S_{\texttt{A}\to\texttt{B}}$, respectively.

The $\texttt{CKV}(\dots)^*$ Data Structure

$\texttt{CKV}(\dots)^*$ is a tuple of binaries which each combine data from all non-empty Merkle leaf nodes of node A and B in the sets $S_{\texttt{A}\to\texttt{B}}$ and $S_{\texttt{B}\to\texttt{A}}$, respectively. The creation of $\texttt{CKV}(\dots)^*$ is shown in Algorithm 15 and described below. The need for all five of its components will become clear below and from Section 8.4 but note that by grouping different types of data into separate binaries instead of using a single one, we bundle similar items and thus allow a better overall zlib compression of the synchronisation messages.

Algorithm 15 $\texttt{CKV}^*$ creation for tree nodes in $S_{\texttt{A}\to\texttt{B}}$ on A and $S_{\texttt{B}\to\texttt{A}}$ on B

```
function CKV*(nodes)          ▷ at A and B for data in the given Merkle tree nodes
    HBin, VBin, BSizeBin, DPrefixBin, DupesBin ← empty bitstring
    for all N ∈ nodes do                                  ▷ note: all N are non-empty
        {H, V} ← CKV(N.bucket)          ▷ also adds to Δ'_A (Algorithm 7, page 52)
        b_d ← H.RemovePrefix(H)              ▷ delta-encoding prefix (Section 5.2.1)
        HBin.AddBits(H)                                                  ▷ hashes
        VBin.AddBits(V)                                                  ▷ versions
        BSizeBin.AddBits(|N.bucket| − 1, ⌈log2 b⌉)   ▷ bucket sizes, ⌈log2 b⌉ bits
        DPrefixBin.AddBits(b_d)                          ▷ delta-encoding prefixes
        DupesBin.AddBits(|Δ'_A|, ⌈log2(b + 1)⌉)   ▷ number of non-unique hashes
    end for
    return {HBin, VBin, BSizeBin, DPrefixBin, DupesBin}                  ▷ a 5-tuple
end function
```

The first two components of $\texttt{CKV}(\dots)^*$, i.e. *HBin* and *VBin*, are basically a concatenation of the CKV components (ref. Section 5.2) of the items in each leaf nodes' bucket, i.e. a binary of the delta-encoded hashes and a binary of the version numbers. We do, however, split off the prefix the delta-encoding uses to encode the actual number of bits b_d used per hash (ref. Section 5.2.1) and re-package it into a separate binary *DPrefixBin*. Similarly, we create separate binaries for the bucket sizes, i.e. *BSizeBin*, and the number of non-unique hashes, i.e. *DupesBin*, which is determined by the CKV creation algorithm. Additionally, since we operate on non-empty leaves, we encode $|N.bucket| - 1$ instead and only use $\lceil \log_2(b) \rceil$ bits each (with Merkle bucket size b). The number of non-unique hashes, however, still requires $\lceil \log_2(b+1) \rceil$ bits each.

The $\texttt{CK}'_{idx}(\dots)^*$ Data Structure

$\texttt{CK}'_{idx}(\dots)^*$ is a single binary that concatenates all $\texttt{CK}'_{idx}$ structures the *trivial'* algorithm creates (ref. Section 6.3.1) to request items inside a Merkle leaf

node's bucket. Instead of truncating the individual $\texttt{CK}'_{idx}$, however, we leave their sizes fixed at b bits to allow decoding the full binary on the other node. Algorithm 16 shows its creation based on the received $\texttt{CKV}(\dots)^*$ and a list of hash lists with the items to request from the individual *trivial'* reconciliations.

Algorithm 16 ${\texttt{CK}'_{idx}}^*$ creation for $\texttt{CKV}^*$ and a list of hash lists to request

```
function CK'_idx*(CKV*, hashes)   ▷ CKV* from the opposite node, hashes to request
    CK'_idx* ← empty bitstring                          ▷ start an empty binary
    for all CKV ∈ CKV* do
        req ← hashes.PopFirst( )                        ▷ extract matching hashes to request
        CK'_idx ← CK'_idx(CKV, req)                     ▷ see Algorithm 12, page 69
        CK'_idx.EnsureSize(b)                           ▷ revert truncate, append 0-bits if required
        CK'_idx*.AddBits(CK'_idx)                       ▷ append all bits to the end
    end for
    return CK'_idx*
end function
```

*Trivial** vs. *Trivial'* Reconciliation

With the additional information at hand, using *trivial** is more efficient in terms of transfer costs than a single *trivial'* reconciliation of all items in phase 2 which SHash and Bloom use. We benefit from the inherent partitioning into Merkle tree nodes which results in fewer items that may collide and thus lower bit sizes $b_k \in \mathcal{O}\left(\log\left({}^{\tilde{n}}/_{FR}\right)\right)$ (ref. eqs. (5.5) and (5.7), pages 56 and 57) when reconciling partition by partition. Additionally, by using a *trivial** reconciliation for $S_{\texttt{B}\to\texttt{A}}$ and $S_{\texttt{A}\to\texttt{B}}$ each instead of a single combined reconciliation, we only pack items from leaf nodes into $\texttt{CKV}(\dots)^*$ and thus further reduce transfer costs in the *missing* scenario where differences may occur between items in inner and leaf nodes. Furthermore, it is not required to send interval boundaries since these are inherent in the Merkle tree.

On the other hand, $\texttt{CKV}(\dots)^*$ contains additional binaries for the bucket sizes and the number of duplicates and needs to transmit multiple prefixes for the delta-encoding. Their contents, however, are likely to be similar which allows a good compression. Also, we omit sending individual b_k parameters since these may be derived from the bucket size, the number of duplicates and other local information (see below).

8.3.3 Implications of Hash Collisions

Phase 1

In phase 1, Merkle tree nodes are compared hierarchically using their hashes. This is done one-by-one and always compares two hashes whose nodes cover the same key range. In case the hashes are different, some difference *must* exist below these nodes since identical items always yield identical hashes.

Table 8.2: Merkle tree reconciliation errors caused by a hash collision of the hashes of tree node A_x and B_x.

Scenario	Node types	Worst-Case Result (after phase 2)
missing	leaf-leaf	$2b$ unrecognised items of Δ
	inner-inner	$A_x.\texttt{items} + B_x.\texttt{items}$ unrecognised items of Δ
outdated	leaf-leaf	b unrecognised items of Δ
	inner-inner	$A_x.\texttt{items}$ ($= B_x.\texttt{items}$) unrecognised items of Δ

If any of these two nodes is a leaf node, all items below the two will be passed on to phase 2. This includes any common items included in a differing bucket (if $b \geq 2$) or—only in the *missing* scenario—common items between a leaf node on one side and an inner node on the other side, e.g. if node `A` in the example above also contained an item with key 53 and version 1 in which case $A_{2,2}$ would contain the same item as $B_{3,4}$ but the synchronisation would stop at $A_{2,2}/B_{2,2}$. These additional common items in phase 2 increase the communication costs of phase 2 but do not create any errors in the overall reconciliation yet.

In case two hashes match, however, all items below these tree nodes may either be identical or not. In the latter case, a hash collision occurred and any differing item is not present in phase 2 on either node and thus unrecognised. For two colliding leaf nodes[a], in the worst case, $2b$ items are affected in the *missing* scenario and b items in the *outdated* scenario. Similarly, if two inner nodes collide, in the worst case, every item beneath each node is affected and missing in Δ'. Table 8.2 summarises these failure cases.

Phase 2

Since *trivial** is basically a multiply executed *trivial'* reconciliation, consequences of hash collisions in each `CKV` data structure are the same as in the SHash and Bloom algorithms (Sections 6.3.2 and 7.3.2) and thus up to twice the number of non-unique hashes in each instance (ref. Section 5.3.1). The sets *trivial** works on are based on the Merkle tree nodes phase 1 discovers as being different. The number of items may thus be larger than the actual number of differences due to bucketing (parameter b) or smaller due to hash collisions.

8.4 Deducing Merkle Tree Parameters from *FR*

To fulfil a given *FR* with the protocol of Figure 8.1 (page 111), we first split the approximate Merkle reconciliation into two phases, each targeting equal failure rates. Similarly to the previous algorithms, we thus use $FR(p_1) \coloneqq {}^{FR}/_{2}$ for phase 1 (ref. Sections 6.4 and 7.4).

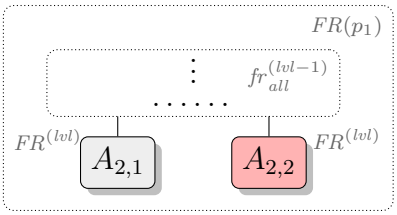
[a] We use the term of two Merkle tree nodes colliding equally to the collision of their respective hashes.

8.4.1 Phase 1

Recall that phase 1 of the Merkle tree synchronisation protocol consists of a top-down hash comparison protocol of the Merkle tree nodes among the data they represent, i.e. $S_{\mathtt{A}}$ and $S_{\mathtt{B}}$. These trees have leaf nodes with buckets of up to $b \geq 1$ items each and a degree of $v \geq 2$, i.e. v sub-trees per inner node.

During the reconciliation, there are two accuracy-influencing parameters which are set per tree level: the number of bits to use for hashes of inner nodes, i.e. S_i, and for hashes of leaf nodes, i.e. S_ℓ. We will distribute the available failure rate level by level and derive appropriate S_i and S_ℓ for the synchronisation protocol given by Algorithm 14 (page 114). Below, we describe the individual concepts being used which together make up the overall phase 1 parameter deduction and failure rate calculation.

Distributing FR among all Nodes of a Tree Level



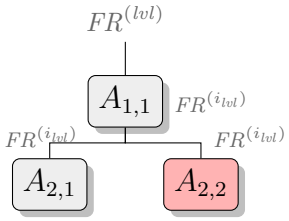
For each tree level $lvl \in \{1, 2, \ldots\}$, assuming *all* previous approximate processes have a combined failure rate $fr_{all}^{(lvl-1)}$, we derive the target failure rate $FR^{(lvl)}$ of *each* sub-tree of the current node list MN (ref. Algorithm 14) from eq. (6.10) (page 74), given that we need to fulfil $FR(p_1)$ in total:

$$FR^{(lvl)} := \frac{FR(p_1) - fr_{all}^{(lvl-1)}}{|MN|} \tag{8.1}$$

With this simple scheme, we allow each sub-tree to use an equal amount of the failure rate and require only minimal overhead (see below).

Distributing FR to a Node and its Children

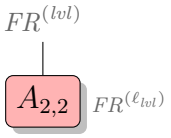
For each sub-tree of the current node list we derive new failure rate targets for all possible approximate processes it owns based on its overall target failure rate $FR^{(lvl)}$. We distinguish inner and leaf nodes:



Inner Nodes Inner nodes have up to $v + 1$ approximate sub-processes: v child trees and the hash comparison of the node itself. We aim at distributing $FR^{(lvl)}$ equally among these sub-processes and thus use the following target failure rate for the hash comparison of an inner node (ref. eq. (6.2), page 71):

$$FR^{(i_{lvl})} = \frac{FR^{(lvl)}}{v + 1} \tag{8.2}$$

Leaf Nodes Leaf nodes only have one process—the hash comparison itself—and do not require a further split of the available failure rate. Therefore, we use the following target failure rate for each leaf node's hash comparison:



$$FR^{(\ell_{lvl})} = FR^{(lvl)} \tag{8.3}$$

Deriving S_i and S_ℓ from $FR^{(i_{lvl})}$ and $FR^{(\ell_{lvl})}$

The actual number of bits to use for the hash comparisons, i.e. S_i for inner nodes and S_ℓ for leaf nodes, depends on the target failure rates $FR^{(i_{lvl})}$ and $FR^{(\ell_{lvl})}$ as well as the number of items affected by a hash collision. We derive appropriate sizes separately for all inner and leaf nodes of a tree level *lvl*.

Inner Nodes Since the basic Merkle tree synchronisation protocol of Section 8.3.1 does not allow an estimation of the number of failures from two inner nodes' hash collision, we would have to assume the worst case that all items are affected. Instead, we establish a smaller upper bound on the number of affected items by transmitting the maximum number of items below *any* node[b] in the current node list MN. `B` thus initially sends $\texttt{MaxIC}_\texttt{B} = |S_\texttt{B}|$ and sends updated values of $\texttt{MaxIC}_\texttt{B}$ with the result flags. `A` sends its $\texttt{MaxIC}_\texttt{A}$ values (initially $|S_\texttt{A}|$) with the compressed hashes (ref. Figure 8.1, page 111). Since these are just single values once per tree level, the overhead is negligible.

In the worst case, $\tau \coloneqq \texttt{MaxIC}_\texttt{A} + \texttt{MaxIC}_\texttt{B}$ items (on both nodes) are affected by a hash collision of two inner nodes and we derive an upper bound on the expected number of failures $fr^{(i_{lvl})}$ of two inner nodes' hash comparison from the hash collision probability 2^{-S_i} and τ. S_i can then be selected appropriately to fulfil $FR^{(i_{lvl})}$:

$$FR^{(i_{lvl})} \geq \underbrace{\frac{1}{2^{S_i}}}_{\text{hash collision probability}} \cdot \overbrace{\tau}^{\text{affected items (worst case)}} =: fr'^{(i_{lvl})} \geq fr^{(i_{lvl})} \quad \Leftarrow \quad S_i \coloneqq \left\lceil \log_2 \frac{\tau}{FR^{(i_{lvl})}} \right\rceil \tag{8.4}$$

Instead of using a common upper bound, we could have also sent individual $\texttt{MaxIC}_*$ for each Merkle node but unless the Merkle tree is extremely unbalanced and τ varies by multiple orders of magnitude (powers of 2), the effect on S_i is limited. The costs of transferring multiple $\texttt{MaxIC}_*$, however, outweigh the reductions on some S_i.

Leaf Nodes Similarly, the number of affected items due to hash collisions among leaf nodes can be given as $\min(\tau, 2b)$ in the worst case. Together with

[b] Note that although leaf nodes are included in $\texttt{MaxIC}_\texttt{A}$ and $\texttt{MaxIC}_\texttt{B}$, if there is an inner node in MN, the number of items it represents is always larger than a leaf node's number of items or else it would have been a leaf node, too.

the hash collision probability 2^{-S_ℓ}, we derive an upper bound on the expected number of failures $fr^{(\ell_{lvl})}$ and set S_ℓ appropriately in order to fulfil $FR^{(\ell_{lvl})}$:

$$FR^{(\ell_{lvl})} \geq \underbrace{\frac{1}{2^{S_\ell}}}_{\text{hash collision probability}} \cdot \overbrace{\min(\tau, 2b)}^{\text{affected items (worst case)}} =: fr'^{(\ell_{lvl})} \geq fr^{(\ell_{lvl})} \quad \Leftarrow \quad S_\ell \coloneqq \left\lceil \log_2 \frac{\min(\tau, 2b)}{FR^{(\ell_{lvl})}} \right\rceil \tag{8.5}$$

Accuracy Similarly to the *trivial* reconciliation, due to ceiling S_i and S_ℓ, we create more accurate versions of the algorithm than needed. In order to re-distribute these unused parts of the failure rate (see below), we would ideally use the exact expected failure rates $fr^{(i_{lvl})}$ and $fr^{(\ell_{lvl})}$. Since these are unknown, however, we will instead use the expected effective *worst-case* failure rates $fr'^{(i_{lvl})}$ and $fr'^{(\ell_{lvl})}$ from which S_i and S_ℓ were derived, respectively.

Using δ_{exp} to Reduce the Bound τ on the Number of Affected Items

So far, we have only been using $\texttt{MaxIC}_\texttt{A} + \texttt{MaxIC}_\texttt{B}$ as an upper bound on the number of affected items but we can get a much lower bound using δ_{exp}. Unfortunately, without further information on the failure distribution, we can only use δ_{exp} to calculate an estimate on the upper bound of the total number of affected items $\delta \cdot n$, i.e. for the whole reconciliation of $S_\texttt{A}$ and $S_\texttt{B}$. This follows similarly to the SHash algorithm (ref. eq. (6.6), page 73) and thus, for each tree level, τ may be given as:

$$\begin{aligned} \tau &\coloneqq \min\left(\left\lceil \delta_{exp} \cdot \max\left(\left\lceil \frac{n_\texttt{A} + n_\texttt{B}}{2 - \delta_{exp}} \right\rceil, |n_\texttt{A} - n_\texttt{B}|\right)\right\rceil, \texttt{MaxIC}_\texttt{A} + \texttt{MaxIC}_\texttt{B}\right) \\ &= \min\left(\lceil \delta_{exp} \cdot \tilde{n} \rceil, \texttt{MaxIC}_\texttt{A} + \texttt{MaxIC}_\texttt{B}\right) \gtrapprox \min\left(\lceil \delta \cdot n \rceil, \texttt{MaxIC}_\texttt{A} + \texttt{MaxIC}_\texttt{B}\right) \end{aligned} \tag{8.6}$$

If we assumed a uniform failure distribution—or employed a first re-hashing phase to create one—, we could use δ_{exp} recursively down the tree and use $\tau \coloneqq \lceil \delta_{exp} \cdot \lceil {(\texttt{MaxIC}_\texttt{A}+\texttt{MaxIC}_\texttt{B})}/{(2-\delta_{exp})} \rceil \rceil$ instead. However, as shown in Section 8.7.4 below, with regards to the communication costs, uniformly distributed failures are actually the worst case. We thus stay with τ from eq. (8.6) and note that the main effect of δ_{exp} is restricted to the upper levels of the tree.

Calculating $fr_{all}^{(lvl)}$ and Re-Distributing unused $FR^{(lvl)}$

Until now, we have reserved parts of the available target failure rate $FR^{(lvl)}$ for approximate hash comparisons as well as for potential approximate sub-processes. During the synchronisation, however, after sending hashes to node B, the following situations may be detected exactly or result in sub-processes being dropped and thus never create failures:

- *Comparing inner and leaf nodes*: This is always detected as a mismatch from the hash tags we add (ref. Section 8.3). Since a leaf node is involved, however, there are no further sub-processes in phase 1 but some target failure rate was reserved in case of the inner node being on node `A`.

- *Comparing an empty leaf node with any other node*: This is similar to the previous case and also covered by the hash tags.

- *Any hash mismatch*: Different hashes must originate from different items and no failures may be created here. If sub-processes exist, i.e. in case of inner nodes on both sides, they are executed in the next level.

- *Any hash match*: The hash match itself may originate from a collision and is thus an approximate process. Since the sub-trees are assumed to be identical, there are no further sub-processes. Any reserved target failure rate, i.e. in case of inner nodes on both sides, remains unused.

All of these cases are distinguishable by both nodes due to each node's own Merkle tree node types and the result codes from Table 8.1 (page 115).

The overall *used target failure rate* in this level can thus be derived from the number of inner-inner and leaf-leaf hash matches together with their expected worst-case failure rates $fr'^{(i_{lvl})}$ and $fr'^{(\ell_{lvl})}$ (ref. eqs. (8.4) and (8.5)) based on the maximum number of affected items τ and the chosen values for S_i and S_ℓ, respectively. Together, for the next iteration as described above, by using the linearity of the expected value, the effective worst-case failure rate $fr_{all}^{(lvl)}$ of *all* approximate processes up to the current tree level is given as (with $fr_{all}^{(0)} := 0$):

$$fr_{all}^{(lvl)} = \underbrace{fr_{all}^{(lvl-1)}}_{\text{previous levels}} + \underbrace{|matches_{\text{inner-inner}}| \cdot fr'^{(i_{lvl})} + |matches_{\text{leaf-leaf}}| \cdot fr'^{(\ell_{lvl})}}_{\text{current tree level}} \tag{8.7}$$

With this, any difference between the target failure rates $FR^{(i_{lvl})}$ and $FR^{(\ell_{lvl})}$ and their effective worst-case failure rates $fr'^{(i_{lvl})}$ and $fr'^{(\ell_{lvl})}$, respectively, can be re-distributed to the next tree level or—in the last one—to the next phase.

The last of these $fr_{all}^{(lvl)}$ thus reflects the effective worst-case failure rate $fr'_m(p_1)$ of the whole phase 1 which depends on the distribution of the data items as well as the distribution of the failures on both nodes:

$$fr_m(p_1) \lessapprox fr'_m(p_1) := fr_{all}^{(\text{last } lvl)} \leq FR(p_1) \tag{8.8}$$

Further Remarks

Note that during the whole phase 1, it is not necessary to transmit either S_ℓ, S_i, $FR^{(\ell_{lvl})}$, or $FR^{(i_{lvl})}$ between the nodes `A` and `B` since both nodes have all the information from the Merkle tree structure, the compressed hashes or the result flags, and the $\texttt{MaxIC}_{Y \in \{\texttt{A},\texttt{B}\}}$ values to calculate all of them themselves.

8.4.2 Phase 2

As in Sections 6.4.2 and 7.4.2 above, for a second approximate reconciliation phase, we will re-use any unused failure rate from phase 1. Therefore, in phase 2, we target an overall failure rate of $FR(p_2) = FR - fr'_m(p_1)$ and split it into $|S_{\mathtt{B}\to\mathtt{A}}| + |S_{\mathtt{A}\to\mathtt{B}}|$ almost equally accurate sub-processes $trivial_{sub}$ iteratively using eq. (6.10) (page 74). This process is outlined below and re-distributes accuracy based on the effective (worst-case) failure rates of each sub-process and thus accounts for rounding inefficiencies in each of them.

Node `A` initiates phase 2 with $|S_{\mathtt{A}\to\mathtt{B}}|$ *trivial* sub-processes $t^{(j)}_{S_{\mathtt{A}\to\mathtt{B}}}$ for $j \in \{1, 2, ..., |S_{\mathtt{A}\to\mathtt{B}}|\}$, node `B` initiates $|S_{\mathtt{B}\to\mathtt{A}}|$ *trivial* sub-processes $t^{(k)}_{S_{\mathtt{B}\to\mathtt{A}}}$ for $k \in \{1, 2, ..., |S_{\mathtt{B}\to\mathtt{A}}|\}$. Together, they have to fulfil $FR(p_2)$ and thus, according to eq. (6.10), both *trivial* sub-processes $t^{(1)}_{Y\in\{S_{\mathtt{A}\to\mathtt{B}},S_{\mathtt{B}\to\mathtt{A}}\}}$ start by using $FR(t^{(1)}_Y) = {FR(p_2)}/{(|S_{\mathtt{B}\to\mathtt{A}}| + |S_{\mathtt{A}\to\mathtt{B}}|)}$. Then, each sub-process uses its individual $\mathtt{MaxIC}_{Y\in\{\mathtt{A},\mathtt{B}\}}$ from the Merkle tree level where the mismatch occurred to calculate the upper bound $\tilde{n}$ on the number of affected items (with $\delta_{exp} = 100\,\%$ in eq. (5.4), page 56) and finally b_k (eq. (5.5), page 56) for the *trivial* reconciliation:

- Since $S_{\mathtt{A}\to\mathtt{B}}$ only originates from non-empty leaf nodes at `A` and inner nodes at `B` (ref. Table 8.1, page 115), a trivial sub-process at node `A` may have up to $\tilde{n} := |\mathtt{node.bucket}| + \mathtt{MaxIC}_{\mathtt{B}}$ different keys to hash.

- If the *trivial* sub-process from $S_{\mathtt{B}\to\mathtt{A}}$ originates from a non-empty leaf node at `B` and an inner node at `A`, we will continue similarly with $\tilde{n} := |\mathtt{node.bucket}| + \mathtt{MaxIC}_{\mathtt{A}}$.

- If the *trivial* sub-process from $S_{\mathtt{B}\to\mathtt{A}}$ originates from two non-empty leaf nodes, we will instead use $\tilde{n} := |\mathtt{node.bucket}| + \min(\mathtt{MaxIC}_{\mathtt{A}}, b)$ based on the maximum number of items a leaf node may have.

Following the iterative splitting of the target failure rate $FR(p_2)$ as described on page 74, each subsequent *trivial* sub-process $t^{(2)}_Y, t^{(3)}_Y, \ldots$ then adds its effective worst-case failure rate fr'_t (ref. eq. (5.6), page 57) to the sum of the used failure rate and calculates its own target failure rate appropriately. At the end of phase 2 on both nodes, each node may thus still have some unused failure rate which could not be re-distributed. We will evaluate this in Section 8.5.

8.4.3 Overall Failure Rate

With the effective worst-case failure rate $fr'_m(p_1)$ of phase 1 (ref. eq. (8.8)) and $fr'_m(p_2) := fr'_t$ of phase 2 (ref. Section 8.4.2 above) and by using the linearity of the expected value, the overall effective worst-case failure rate of the Merkle tree reconciliation is given as:

$$fr'_m = fr'_m(p_1) + fr'_m(p_2) \tag{8.9}$$

8.4.4 Example

Table 8.3 exemplifies each tree level's failure rates and hash sizes for a Merkle ($v = 4$, $b = 3$) reconciliation with $n = 100\,000$ randomly distributed items, $FR = 0.1$, and $\delta = \delta_{exp} = 10\,\%$ outdated items. Here, we will look at both the progression of the different reconciliation parameters with the tree level as well as the detailed calculations of these values in the exemplary tree level 7.

Table 8.3: Merkle ($v = 4$, $b = 3$) reconciliation parameters per tree level with τ from eq. (8.6) (other parameters from eqs. (8.1) to (8.5) and (8.7)).

lvl	$\lvert MN \rvert$	$\texttt{MaxIC}_{\texttt{A,B}}$	τ	$FR^{(lvl)}$, $FR^{(\ell_{lvl})}$	$FR^{(i_{lvl})}$	S_i	S_ℓ	$fr_{all}^{(lvl)}$
1	1	100 000	10 527	$5.00 \cdot 10^{-2}$	$1.00 \cdot 10^{-2}$	21	7	0
2	4	25 275	10 527	$1.25 \cdot 10^{-2}$	$2.50 \cdot 10^{-3}$	23	9	0
3	16	6 393	10 527	$3.13 \cdot 10^{-3}$	$6.25 \cdot 10^{-4}$	25	11	0
4	64	1 668	3 336	$7.81 \cdot 10^{-4}$	$1.56 \cdot 10^{-4}$	25	13	0
5	256	447	894	$1.95 \cdot 10^{-4}$	$3.91 \cdot 10^{-5}$	25	15	0
6	1 024	131	262	$4.88 \cdot 10^{-5}$	$9.77 \cdot 10^{-6}$	25	17	$7.81 \cdot 10^{-6}$
7	4 092	42	84	$1.22 \cdot 10^{-5}$	$2.44 \cdot 10^{-6}$	26	19	$4.32 \cdot 10^{-4}$
8	15 012	17	34	$3.30 \cdot 10^{-6}$	$6.60 \cdot 10^{-7}$	26	21	$7.69 \cdot 10^{-3}$
9	27 788	8	16	$1.52 \cdot 10^{-6}$	$3.04 \cdot 10^{-7}$	26	22	$2.77 \cdot 10^{-2}$
10	6 560	6	12	$3.40 \cdot 10^{-6}$	$6.81 \cdot 10^{-7}$	25	21	$3.57 \cdot 10^{-2}$
11	224	4	8	$6.38 \cdot 10^{-5}$	$1.28 \cdot 10^{-5}$	20	17	$3.98 \cdot 10^{-2}$
12	4	4	8	$2.54 \cdot 10^{-3}$	$5.08 \cdot 10^{-4}$	14	12	$3.98 \cdot 10^{-2}$
13	4	2	4	$2.54 \cdot 10^{-3}$	$5.08 \cdot 10^{-4}$	13	11	$4.38 \cdot 10^{-2}$

There are a total of 24 233 inner and 72 700 leaf nodes. 14 997 leaf nodes are empty and do not represent data items. For the reconciliation, FR was first split into the two phases with a target failure rate $FR(p_1) = 0.05$ for phase 1. Its effective worst-case failure rate, however, was $fr'_m(p_1) \approx 0.043754$, which leaves phase 2 with $FR(p_2) \approx 0.056247$. A total of 9 462 *trivial* sub-processes were used (all from $S_{\texttt{B}\to\texttt{A}}$ as expected in the *outdated* scenario where only non-empty leave hashes may mismatch; ref. Table 8.1, page 115) and showed an effective worst-case failure rate of $fr'_m(p_2) \approx 0.034640$. The overall effective worst-case failure rate of this example is thus $fr'_m := fr'_m(p_1) + fr'_m(p_2) \approx 0.078394$ (ref. Section 8.5 for a detailed discussion on the deviation from the target $FR = 0.1$).

Parameter Progression with the Tree Level

In this example, the number of nodes in tree levels 1 to 6 increases with the degree $v = 4$ which indicates that all hash comparisons in tree levels 1 to 5 are mismatches. In contrast, level 7 does not involve 4 096 nodes and therefore, either the tree is unbalanced or some hashes of level 6 match. Since the hash mismatches in levels 1 to 5 are exact comparisons, nothing of the available

target failure rate is used and $fr_{all}^{(lvl)}$ remains 0 (ref. eq. (8.7), page 123). Since then the target failure rate $FR^{(lvl)}$ for each node only depends on the total number of nodes per tree level (eq. (8.1), page 120), it thus quarters each level. Its further progression depends on the effective worst-case failure rate of the hash matches—and thus $fr_{all}^{(lvl)}$—as well as the number of remaining nodes $|MN|$ and may thus increase or decrease accordingly. Likewise, S_i and S_ℓ increase or decrease with $FR^{(lvl)}$ and the number of affected items. Also, $fr_{all}^{(lvl)}$ progresses towards $FR(p_1)$ as expected.

Parameter Calculations for Tree Level 7

We will now reconstruct the calculations the Merkle tree synchronisation protocol in tree level 7 which are based on eqs. (8.1) to (8.7) (pages 120 to 123). At first, $FR^{(7)}$ is determined from eq. (8.1):

$$FR^{(7)} = \frac{FR(p_1) - fr_{all}^{(6)}}{|MN|} = \frac{0.05 - 7.81 \cdot 10^{-6}}{4092} = 1.22 \cdot 10^{-5}$$

$FR^{(\ell_7)} = FR^{(7)}$ follows immediately from eq. (8.3). $FR^{(i_7)}$ is calculated from eq. (8.2) as follows:

$$FR^{(i_7)} = \frac{FR^{(7)}}{v+1} = \frac{1.22 \cdot 10^{-5}}{5} = 2.44 \cdot 10^{-6}$$

S_i and S_ℓ (eqs. (8.4) and (8.5)) follow with τ from eq. (8.6) which reflects the maximum number of affected items due to a hash collision:

$$\begin{aligned}
\tau &= \min\left(\left\lceil \delta_{exp} \cdot \tilde{n} \right\rceil, \texttt{MaxIC}_\texttt{A} + \texttt{MaxIC}_\texttt{B}\right) = \min\left(10\,527, 42 + 42\right) = 84 \\
S_i &= \left\lceil \log_2 \frac{\tau}{FR^{(i_7)}} \right\rceil = \left\lceil \log_2 \frac{84}{2.44 \cdot 10^{-6}} \right\rceil = \lceil 25.03 \rceil = 26 \\
S_\ell &= \left\lceil \log_2 \frac{\min(\tau, 2b)}{FR^{(\ell_7)}} \right\rceil = \left\lceil \log_2 \frac{\min(84, 6)}{1.22 \cdot 10^{-5}} \right\rceil = \lceil 18.91 \rceil = 19
\end{aligned}$$

With these values of S_i and S_ℓ, the effective worst-case failure probabilities of the Merkle tree node hash comparisons are derived from eqs. (8.4) and (8.5):

$$\begin{aligned}
fr'^{(i_7)} &= \frac{1}{2^{S_i}} \cdot \tau = \frac{1}{2^{26}} \cdot 84 = 1.25 \cdot 10^{-6} \\
fr'^{(\ell_7)} &= \frac{1}{2^{S_\ell}} \cdot \min(\tau, 2b) = \frac{1}{2^{19}} \cdot \min(84, 6) = 1.14 \cdot 10^{-5}
\end{aligned}$$

In order to calculate the failure rate of tree level 7 and thus the overall failure rate $fr_{all}^{(7)}$ up until and including this level from eq. (8.7), we need the number of inner-inner and leaf-leaf hash matches (not shown in Table 8.3). Assuming that all nodes are inner nodes, we derive the number of inner-inner hash matches from the number of nodes $|MN|$ in level 8 which should be $4\,092 \cdot 4 = 16\,368$ if

no match occurred. Since its actual value is 15 012, however, 1 356 nodes are missing due to $^{1\,356}/_{4} = 339$ matches. The overall failure rate of levels 1 to 7 combined can thus be calculated as:

$$\begin{aligned} fr_{all}^{(7)} &= fr_{all}^{(6)} + |matches_{\text{inner-inner}}| \cdot fr'^{(i_7)} + |matches_{\text{leaf-leaf}}| \cdot fr'^{(\ell_7)} \\ &= 7.81 \cdot 10^{-6} + 339 \cdot 1.25 \cdot 10^{-6} + 0 \cdot 1.14 \cdot 10^{-5} = 4.32 \cdot 10^{-4} \end{aligned}$$

8.4.5 Overall Costs

Deriving a generic costs formula for the Merkle tree reconciliation is difficult since its actual transfer costs heavily depend on the input data, more precisely: its distribution in the key range as well as the distribution of failures. Here, we provide a rough sketch of classifying the costs by using a uniform key distribution, a perfectly-balanced Merkle tree, and assuming that every leaf node differs so that every tree node's hash is transferred. It is therefore a worst-case synchronisation for a best-case tree which is close to any tree with a uniform key distribution. As we will see from the evaluation in Section 8.7.4 below, other (skewed) key and especially failure distributions actually perform even better in terms of communication costs due to the synchronisation being able to skip over whole sub-trees.

In a perfectly-balanced Merkle tree, the depth d is $\lceil \log_v(^n/_b) \rceil$ since the bucket size b reduces the number of leaf nodes and the key range is split recursively into v sub-intervals. In practice, though, the Merkle tree creation of Algorithm 13 (page 112) is based on fixed sub-interval sizes which do not only lead to buckets not being completely filled but may also create empty leaf nodes, depending on the size of v and b. More generally, thus $d \in \mathcal{O}(\log_v(^n/_b))$, assuming a uniform item key distribution. Please also note that the total number of tree levels is given as $d + 1$.

If there are no matches in inner tree nodes, $fr_{all}^{(lvl)} = 0$ for all tree levels (ref. eq. (8.7)) and the failure rates and signature sizes progress as shown by Table 8.4. Therefore, since $\tau = 2 \cdot {}^n/_{|MN|}$ (an upper bound to eq. (8.6) for perfectly-balanced trees) and $FR^{(i_{lvl})}$ both decrease with the tree level by a factor of $^1/_v$, the value of S_i remains the same throughout all levels (ref. eq. (8.4)). This effect may also be seen in levels 3 to 6 in the example of Table 8.3 above where τ and $FR^{(i_{lvl})}$ roughly quarter with each level.

Overall, there are at most $1 + v^1 + \ldots + v^d = {}^{(v^{d+1}-1)}/_{(v-1)}$ nodes in the tree with at most v^d leaf nodes and up to ${}^{(v^{d+1}-1)}/_{(v-1)} - v^d = {}^{(v^d-1)}/_{(v-1)}$ inner nodes. Therefore, the total costs are bound from above by:

$$\underbrace{\frac{v^d - 1}{v - 1}}_{\text{total number of inner nodes}} \cdot \overbrace{\left\lceil \log_2 \frac{2n \cdot (v+1)}{FR(p_1)} \right\rceil}^{S_i} + \underbrace{v^d}_{\text{leaf nodes at level } d+1} \cdot \overbrace{\left\lceil \log_2 \frac{2b \cdot v^d}{FR(p_1)} \right\rceil}^{S_\ell \text{ at level } d+1}$$

Table 8.4: Reconciling a perfectly-balanced Merkle tree with no matching inner nodes (according to eqs. (8.1) to (8.7), pages 120 to 123).

lvl	$\lvert MN\rvert$	τ	$FR^{(lvl)}$, $FR^{(\ell_{lvl})}$	$FR^{(i_{lvl})}$	S_ℓ	S_i
1	1	$2n$	$FR(p_1)$	$\frac{FR(p_1)}{v+1}$	$\left\lceil \log_2 \frac{2b}{FR(p_1)} \right\rceil$	$\left\lceil \log_2 \frac{2n\cdot(v+1)}{FR(p_1)} \right\rceil$
2	v	$\frac{2n}{v}$	$\frac{FR(p_1)}{v}$	$\frac{FR(p_1)}{v\cdot(v+1)}$	$\left\lceil \log_2 \frac{2b\cdot v}{FR(p_1)} \right\rceil$	$\left\lceil \log_2 \frac{2n\cdot(v+1)}{FR(p_1)} \right\rceil$
3	v^2	$\frac{2n}{v^2}$	$\frac{FR(p_1)}{v^2}$	$\frac{FR(p_1)}{v^2\cdot(v+1)}$	$\left\lceil \log_2 \frac{2b\cdot v^2}{FR(p_1)} \right\rceil$	$\left\lceil \log_2 \frac{2n\cdot(v+1)}{FR(p_1)} \right\rceil$
⋮	⋮	⋮	⋮	⋮	⋮	⋮

With $d \in \mathcal{O}(\log_v(n/b))$ from above, we establish the following complexity class for phase 1:

$$\mathcal{O}\left(\frac{n}{b\cdot v}\cdot\log\frac{n\cdot v}{FR(p_1)} + \frac{n}{b}\cdot\log\frac{n}{FR(p_1)}\right) \subseteq \mathcal{O}\left(\frac{n}{b}\cdot\log\frac{n}{FR}\right)$$

(for $n\to\infty,\ FR(p_1)\to 0$) (for $n\to\infty,\ FR\to 0$)

The latter follows from $FR(p_1) = FR/2$ and

$$\frac{n}{b\cdot v}\cdot\log\frac{n\cdot v}{FR(p_1)} = \underbrace{\frac{1}{v}}_{<1}\cdot\left(\frac{n}{b}\cdot\log\frac{n}{FR(p_1)}\right) + \frac{n}{b}\cdot\underbrace{\frac{\log v}{v}}_{\leq 1} \leq 2\cdot\left(\frac{n}{b}\cdot\underbrace{\log\frac{n}{FR(p_1)}}_{>1}\right)$$

Since the *trivial** reconciliation is an optimisation of the *trivial'* algorithm from the SHash reconciliation, it is bound from above by the communication costs of *trivial'* (ref. Section 6.4.4) and we thus derive the overall Merkle tree reconciliation communication costs complexity as:

$$C_m \in \mathcal{O}\left(\frac{n}{b}\cdot\log\frac{n}{FR} + |\Delta|\cdot\log\frac{|\Delta|}{FR}\right) \tag{8.10}$$

(for $n,|\Delta|\to\infty,\ FR\to 0$)

Note, however, that although v is not present in this class, it does effect practical reconciliations where inner nodes may match: Higher v (a) reduces the depth of the tree and thus avoids multiple transmissions of mismatching (inner node) hashes of differing key ranges, but (b) makes the tree wider and thus unnecessarily transfers multiple matching hashes where a previous tree level could have seen the match with lower v and (c) may introduce more empty leaf nodes. These trade-offs are further discussed in Section 8.7 below using the real communication costs from simulations.

8.4.6 Future Work

The Merkle tree synchronisation protocol presented above applies approximate hash comparisons with a defined accuracy to the standard, "quasi-exact" 160 bit Merkle tree synchronisation. The following techniques may further improve the results in terms of communication costs or number of message rounds. These were not deployed for the evaluation below and remain for future work.

To further decrease the communication costs at the cost of the number of message rounds, we could apply the rsync techniques described in Section 4.2.1 where two phases were introduced: a first phase with smaller checksums and an additional overall strong checksum, and a potential second phase using the full checksum size if the strong checksum indicates a failure. In the Merkle tree synchronisation, this could be applied to each tree level using smaller values for S_i and S_l in a first phase and a full 160 bit feedback hash over all matching hashes. If this feedback hash is equal to the one at node `A`, we continue as usual but adjust the failure rate accordingly. If not, this tree level's hashes need to be transmitted again using the full S_i and S_l. Similarly to rsync, this could be fine-tuned so that the second phase is only required on occasion but still, in the worst-case, twice as many message rounds are needed.

Alternatively, a more light-weight version of this technique could only add the overall hash of the matching hashes and adjust the failure rate accordingly. The expected savings are considerably lower, especially since S_i and S_l only depend logarithmically on the failure rate (eqs. (8.4) and (8.5), pages 121 and 122). On the other hand, this allows a better adjustment to a given FR.

Regarding the number of message rounds, the example in Table 8.3 above showed that in the upper tree levels, most comparisons are hash matches and thus transferred completely. This is most prominent for uniformly distributed failures but may also be relevant for other failure distributions. Instead of transmitting all these hashes, we could start at a lower tree level and thus reduce the number of message rounds (and some communication costs) for high δ at the cost of an increased overhead for very low δ. By starting at level 7 for $v = 4$, for example, we would always transmit 4 096 hashes instead of a single root hash first. More generally, we could split the original interval into any number of sub-intervals in any way, build a Merkle tree for each of them, and start with $|MN| > 1$ in the first tree level of Algorithm 14 (page 114).

8.5 Effective Worst-Case Accuracy

In contrast to the reconciliation algorithms above, the Merkle tree synchronisation heavily depends on the input data, i.e. its distribution in the key space and the distribution of failures among the data items. In order to analyse the effective worst-case accuracy, we therefore evaluate the expected number of failures empirically by using the value the algorithm calculates during the reconciliation of each of 100 random instances of different set reconciliation scenarios (reduced from 1 000 due to the vast number of data points) similar to the evaluation below and the example in Section 8.4.4.

We show the effective worst-case failure rate fr'_m and its compontents $fr'_m(p_1)$ and $fr'_m(p_2)$ (ref. eq. (8.9), page 124) of two different Merkle trees with $v = 4$ and $b \in \{3, 1\}$ to evaluate the influence of the bucket size b and the buckets' fill rate. The influence of the data and failure distribution is evaluated with a set of experiments using a uniform distribution and another set with a binomial distribution. Other combinations than $FR = 0.1$ and $\delta = \delta_{exp} = 1\,\%$ presented here show similar behaviour and are given in Appendix A.2.1 for reference.

Please note that the actual observed failure rates may be lower than fr'_m due to worst-case assumptions in both phases which may not occur, e.g. assuming a common τ in phase 1 which is based on $\texttt{MaxIC}_{X \in \{\texttt{A},\texttt{B}\}}$ (eq. (8.6), page 122) or assuming distinct items between the leaves for phase 2 (ref. Section 5.5.1).

8.5.1 Uniformly Distributed Item Keys and Failures

The following two sections show the calculated effective worst-case failure rates fr'_m of the Merkle tree reconciliation with a uniform item key and failure distribution, i.e. $data_{rand}$ and $fail_{rand}$.

Outdated Items Scenario

Figure 8.5 shows the failure rates of the Merkle tree reconciliation in different phases for $v = 4, b = 3$, as in the example of Section 8.4.4. Most of the overall fluctuations of fr'_m are caused by phase 1 but also by phase 2 not being able to compensate much of this underperformance. The latter is due to the buckets not being completely filled (also see the example of Section 8.4.4) which is in contrast to the worst-case assumptions of phase 2 regarding the total number of different keys to hash[c], i.e. b items per bucket (ref. Section 8.4.2). The bucket fill rate itself results from the structure of the Merkle tree which depends on the tree's degree v and—due to the fixed interval splitting of an inner node (ref. Algorithm 13, page 112)—is also influenced by the number of items n.

As opposed to $b = 3$, phase 2 of the Merkle tree reconciliation with $b = 1$ (Figure 8.6) is able to almost completely compensate any discrepancies of

[c]Note that although the Merkle tree reconciliation protocol can detect this deviation *after* receiving the $\texttt{CKV}(\ldots)^*$ structures on each node, it cannot account for this in advance without sending individual item counts for each leaf node.

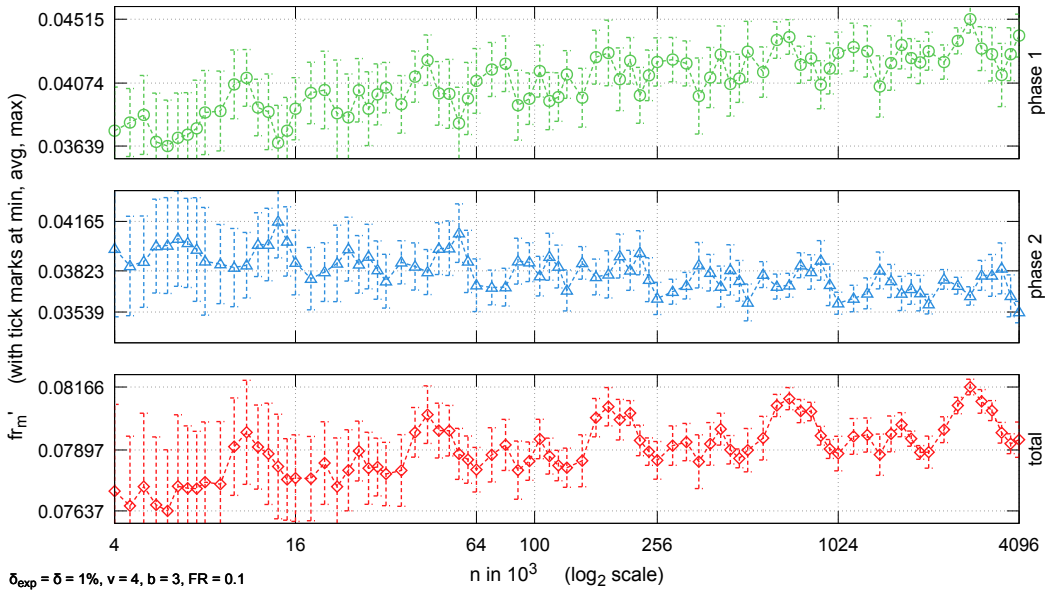


Figure 8.5: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m in the *outdated* scenario (error bars show the standard deviation).

phase 1. There, all non-empty leaf nodes are full as expected by the *trivial** algorithm and hence $fr'_m \approx FR$. Also, with $b = 1$, the standard deviation of $fr'_m(p_1)$ and $fr'_m(p_2)$ is much lower than for $b = 3$ since higher b cause more fluctuations in the tree structure and the bucket fill rates due to both lower trees as well as our fixed interval splits.

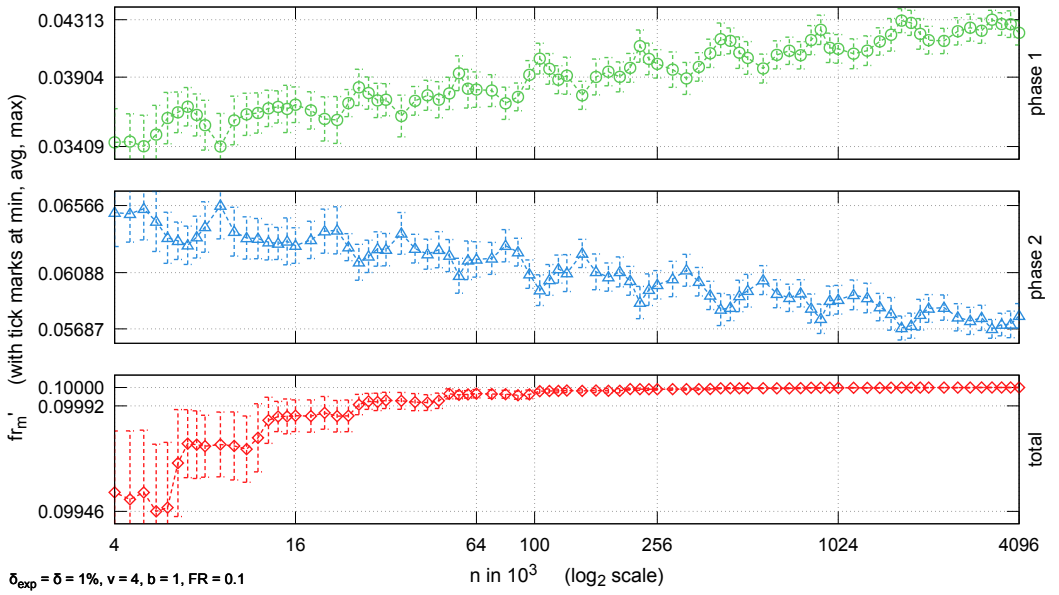


Figure 8.6: Merkle $v = 4, b = 1$ effective worst-case failure rate fr'_m in the *outdated* scenario (error bars show the standard deviation).

In general, in smaller trees, e.g. due to lower n, minor changes in the keys of the items create whole new sub-trees and thus cause more jitter than in larger trees. Large trees also re-use more of their target $FR(p_1)$ since, in the deepest level, any reserved parts of the available target failure rate from what turns out to be an exact sub-process cannot be re-distributed (ref. Section 8.4.1). The deeper the tree, the smaller these reserved parts and the sum of these may be smaller in total as long as there are enough approximate processes.

Missing Items Scenario

The effect of lower b resulting in better use of FR in the *outdated* scenario is inverted in the *missing* scenario as shown by Figures 8.7 and 8.8. The lower b, the more sub-trees are missing compared to all items being present and thus, the trees of nodes `A` and `B` diverge. In contrast, with higher b, the structure of the Merkle trees becomes more similar and only the buckets' fill rates vary. Therefore, phase 2 of $b = 1$ hosts more leaf-inner sub-processes than with $b = 3$ which, in return, contains more leaf-leaf sub-processes.

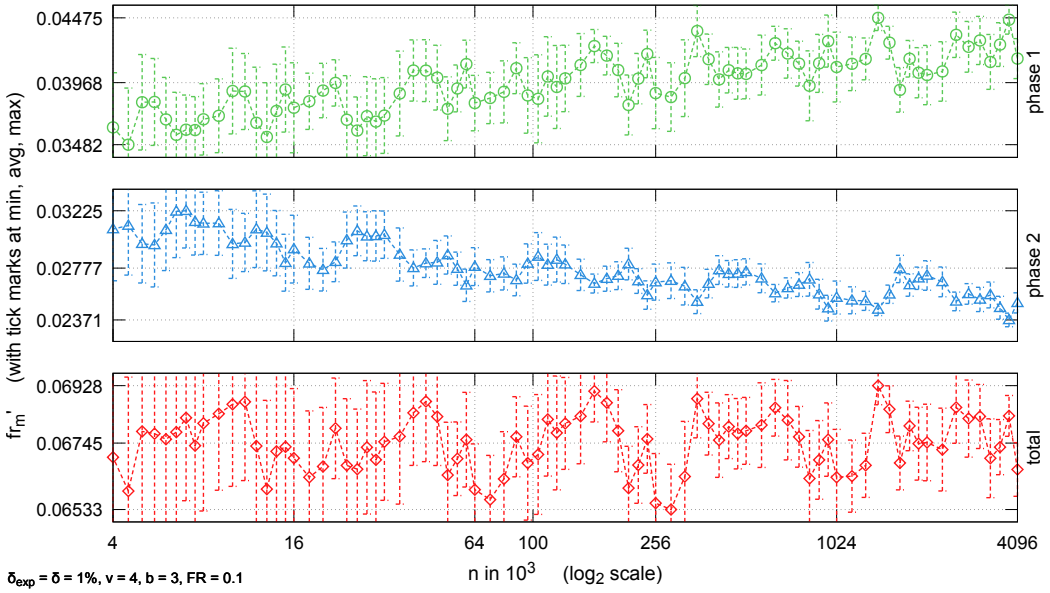


Figure 8.7: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m in the *missing* scenario (error bars show the standard deviation).

To estimate the number of unique keys to hash in phase 2, every leaf-inner *trivial* reconciliation relies on $\texttt{MaxIC}_{Y \in \{A,B\}}$, i.e. the *maximum* number of items below *any* inner node of its tree level, and may thus be larger than the actual value or the upper bound in leaf-leaf reconciliations (ref. Section 8.4.2). This does not only explain the different values of fr'_m for these two bucket sizes but also explains the differences between the *missing* scenario and the *outdated* scenario above where only leaf-leaf sub-processes are present.

Compared to the *outdated* scenario (Figures 8.5 and 8.6), we also observe higher standard deviations in the overall failure rate fr'_m which is due to (more) fluctuations in the tree structure when items are missing while the structure of the Merkle trees with outdated items does not change at all and only depends on the data distribution.

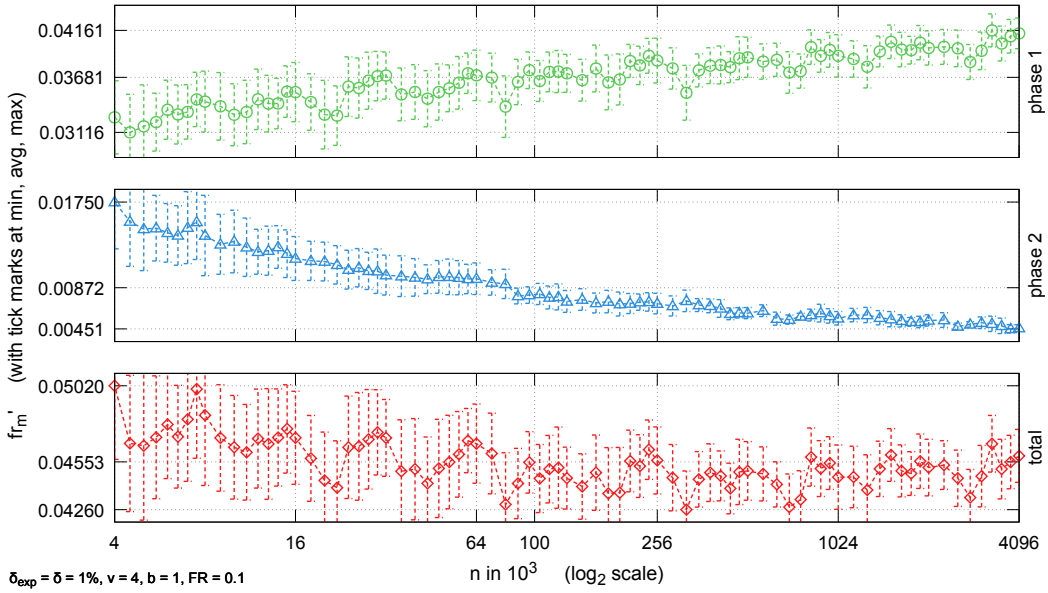


Figure 8.8: Merkle $v = 4, b = 1$ effective worst-case failure rate fr'_m in the *missing* scenario (error bars show the standard deviation).

8.5.2 Binomially Distributed Item Keys and Failures

In contrast to the fairly balanced Merkle trees above and a reconciliation that visits most of the tree nodes, here, we show the effective worst-case failure rates fr'_m in both phases in a binomially distributed scenario with $data_{bin\,0.2}$ and $fail_{bin_{0.2}}$, i.e. using $B(n, p = 0.2)$, for $v = 4, b = 3$.

A binomial data distribution may result in higher bucket fill rates but also leads to skewed Merkle trees and breaks the assumption that the maximum number of affected items is equal in all sub-trees which may influence the *actual* failure rates as shown by Section 8.7 below. However, the *worst-case* failure rates of phase 1—as calculated by the algorithm—are not affected by this since we do not exchange enough data to recognise this. In contrast, the given failure rates of phase 2 may be affected since $\texttt{MaxIC}_{Y\in\{A,B\}}$ is used to estimate the number of different keys to hash (ref. Section 8.4.2) and any discrepancy is recognised *after* receiving $\texttt{CKV}(\ldots)^*$ and is thus included in $fr'_m(p_2)$.

A binomial failure distribution clusters failures in one region of the tree and thus, during the reconciliation, more nodes may be dropped early. Therefore, compared to a uniform failure distribution, there may be fewer approximate

processes from hash matches in total. It also leads to fewer leaf nodes entering phase 2 with fewer common items among them, thus bringing phase 2 closer to its worst-case assumption of $\delta_{exp} = 100\,\%$ (ref. Section 8.4.2).

Overall, we expect an increased impact of phase 2 which, in the best case, compensates more of phase 1's unused FR and, in the worst case, compensates less, compared to $data_{rand}$ and $fail_{rand}$ above.

Outdated Items Scenario

The *outdated* binomial scenario of Figure 8.9 shows slightly higher average values of fr'_m compared to Figure 8.5 and with some n even values close to FR. The general pattern, however, is dominated by phase 2 while phase 1 failure rates, i.e. $fr'_m(p_1)$, are considerably smaller than in the uniform scenario above. The main reason for this behaviour is the inability to re-distribute any unused parts of FR from exact sub-processes such as hash mismatches of the last tree level (ref. Section 8.4.1). These parts are larger than above due to fewer approximate processes during the reconciliation but are re-distributed to phase 2 leading to $fr'_m \approx FR$ in some cases.

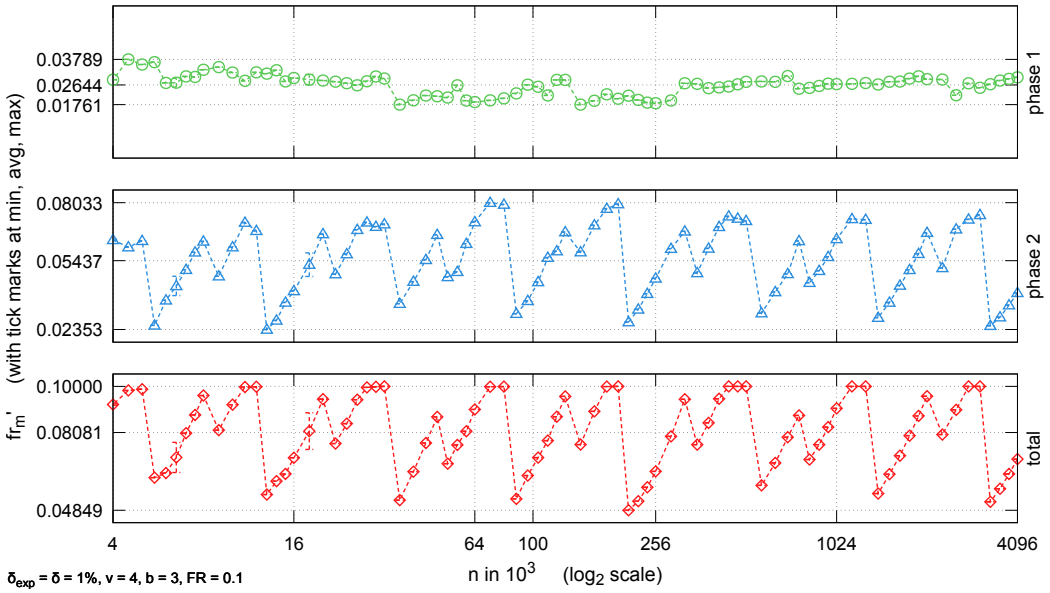


Figure 8.9: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m in the *outdated* scenario of $data_{bin\,0.2}$, $fail_{bin_{0.2}}$ (error bars show the standard deviation).

The pattern of the effective worst-case failure rates of phase 2, i.e. $fr'_m(p_2)$, is caused by the bucket fill rate of the (leaf) nodes participating in phase 2. It follows the difference of the actual number of items and $\min(\texttt{MaxIC}_{Y\in\{A,B\}}, b)$ (ref. Section 8.4.2). This effect seems to be amplified by fewer nodes participating (with higher impact) and, in some scenarios, is apparently not compensated by the overall higher bucket fill rate. As above, the bucket fill rate (also) depends on n due to our fixed interval splits which leads to the given results.

Missing Items Scenario

Similar to the uniformly distributed scenarios above, the binomial *missing* scenario shown by Figure 8.10 exhibits lower effective worst-case failure rates than the appropriate *outdated* scenario. Here, phase 2 also contains nodes from leaf-inner mismatches and thus discrepancies in $\texttt{MaxIC}_{Y \in \{A,B\}}$ become even more prominent since this is the main factor to estimate the number of unique keys to hash. The range of the values of $fr'_m(p_1)$ remains similar, though, with more fluctuations from the differences in the structure of the Merkle trees due to the missing items.

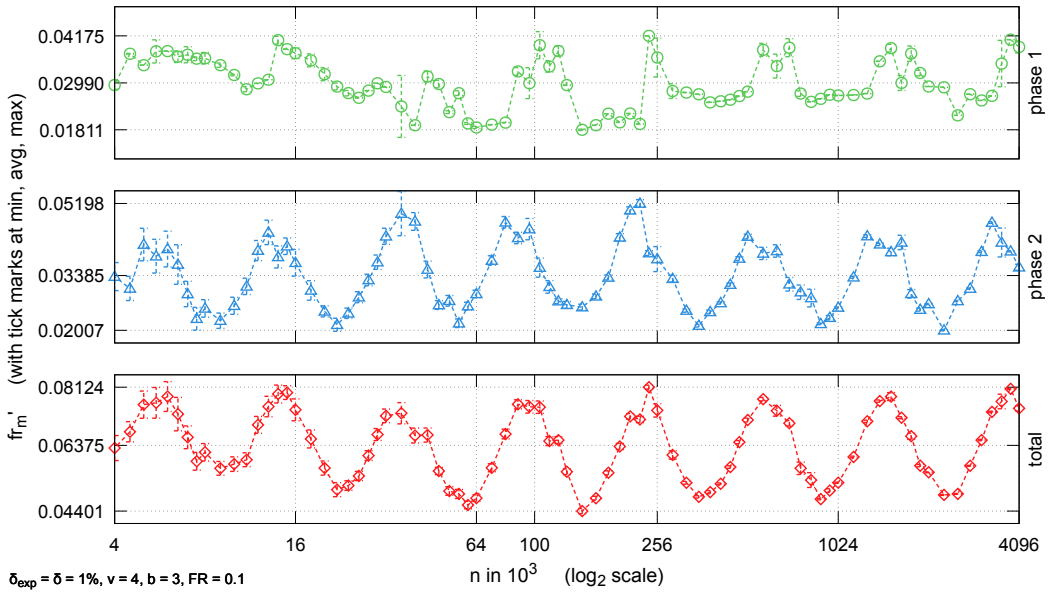


Figure 8.10: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m in the *missing* scenario of $data_{bin\,0.2}$, $fail_{bin_{0.2}}$ (error bars show the standard deviation).

8.6 Related Work

The multi-round set reconciliation protocol with Merkle trees efficiently identifies small differences and is used in distributed storage systems like Dynamo [22], Cassandra [63], and DHash [14]. Until now, however, Merkle trees have only been optimised with the focus of minimising their construction time and maintenance [83] or message rounds [11, 10] while we focus on reducing the transfer costs. Trutna et al. [83], for example, describe the efficient embedding and maintenance of a Merkle tree over an existing storage back-end in the SCADS Distributed Storage Toolkit.

Byers et al. [10] propose two techniques which we omit for a broader evaluation or for simplicity: (a) adding a first randomisation phase, e.g. via re-hashing

the keys, and (b) using collapsed Patricia Tries. Randomising the keys' distribution produces better balanced trees which reduce the tree's (worst case) height and thus the number of synchronisation rounds in the reconciliation. For our evaluations, however, we focus on the amount of transmitted bytes but please note that the $data_{rand}$ scenario effectively represents a Merkle tree reconciliation protocol with a first randomisation phase, as opposed to $data_{bin\,0.2}$ (ref. Section 8.7.4) which shows the impact of leaving it out. Collapsed Patricia Tries are mainly beneficial with unbalanced trees, saving memory due to fewer (inner) nodes. Merkle tree reconciliation over collapsed Patricia Tries, however, is more complex as additional information would need to be exchanged so that both nodes know which node to compare with. It is unclear whether this additional data outweighs the benefits of fewer nodes to exchange.

Additionally to these enhancements, Byers et al. [11, 10] also propose an approximate reconciliation tree (ART) based on putting the Merkle tree into two Bloom filters, one for the inner nodes and one for the leaf nodes. During reconciliation, both are sent in a single round and thus remove the $\mathcal{O}(\log_v(n/b))$ message round complexity of Merkle tree reconciliation. In contrast to Bloom reconciliation and as opposed to [47, 45], ART is able to identify $Mis_{\mathtt{A}}$ due to the underlying structure of the Merkle tree if we include the range into the hash[d]. Compared to Bloom filters, in scenarios with low differences, ART improves the computational overhead due to fewer lookups. In general, though, ART can never be more accurate than an equally-sized Bloom filter [10]. This follows from the observation that the leaf Bloom filter needs to have the same accuracy as an ordinary Bloom filter in the Bloom reconciliation.

A two-phase approach to set reconciliation which is similar to ours has been presented by Lin and Levis [49] who use a (hybrid) Merkle tree and Bloom filter reconciliation protocol in sensor networks. There, too, the first phase merely detects that a difference exists, i.e. by using a hash tree and Bloom filters, while the second phase identifies the actual differences similarly to our trivial reconciliation. They also work with key-version pairs but assume a full coverage of the key space, i.e. every possible key reflects an actual sensor and has a version associated with it, and build the hash tree on versions only. Each of their "summary messages" contains a salt for the hash functions[e] and a list of elements, each with start key index, end key index, a summary hash of the versions in this range, and a Bloom filter of the key-version tuples of all items in this range (with $k = 1$ hash function). With this information, their protocol *DIP* estimates the probability that an item differs and uses this and the estimated costs to decide which algorithm to continue with, i.e. searching further in the tree or trivially reconciling the data.

[d] Note that with ART, as opposed to Merkle reconciliation, each tree node's hash is essentially compared to all tree node hashes of the other node. Therefore, mismatches with sub-trees of another tree level need to be avoided in order to identify missing sub-trees.

[e] Salting hash functions may reduce hash collisions or at least avoid the same ones over and over again in repeated set reconciliation attempts.

8.7 Evaluation

Reconciliation with Merkle trees aims at low transfer sizes for low δ at the cost of having multiple message rounds. We aim at further improving transfer costs by reducing hash sizes according to FR. Two further parameters influence the costs but not the failure rate: the bucket size b and the degree v of the tree.

8.7.1 Parameter Space Exploration

$n = 100\,000$
$data_{rand}$
$\delta = 2\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

In order to select appropriate parameters for both v and b for further evaluations, we present the combined average transfer costs of both phases during a parameter sweep of $v \in [2, 16]$ and $b \in [1, 16]$ in Figure 8.11. Transfer costs are presented as a heatmap with warmer and brighter colours showing higher costs; numbers inside each field show the number of communication messages for the two phases and thus, indirectly, the Merkle tree height.

In general, by increasing v and b, the depth of the Merkle tree—and thus the number of message rounds—can be reduced at the cost of transferring more hashes per level and leaf node, respectively. Additional costs may arise from higher v where more empty leaf nodes are created due to the static interval splits which are independent of the actual key distribution. Since in our implementation hash signature sizes mostly increase per level, depending

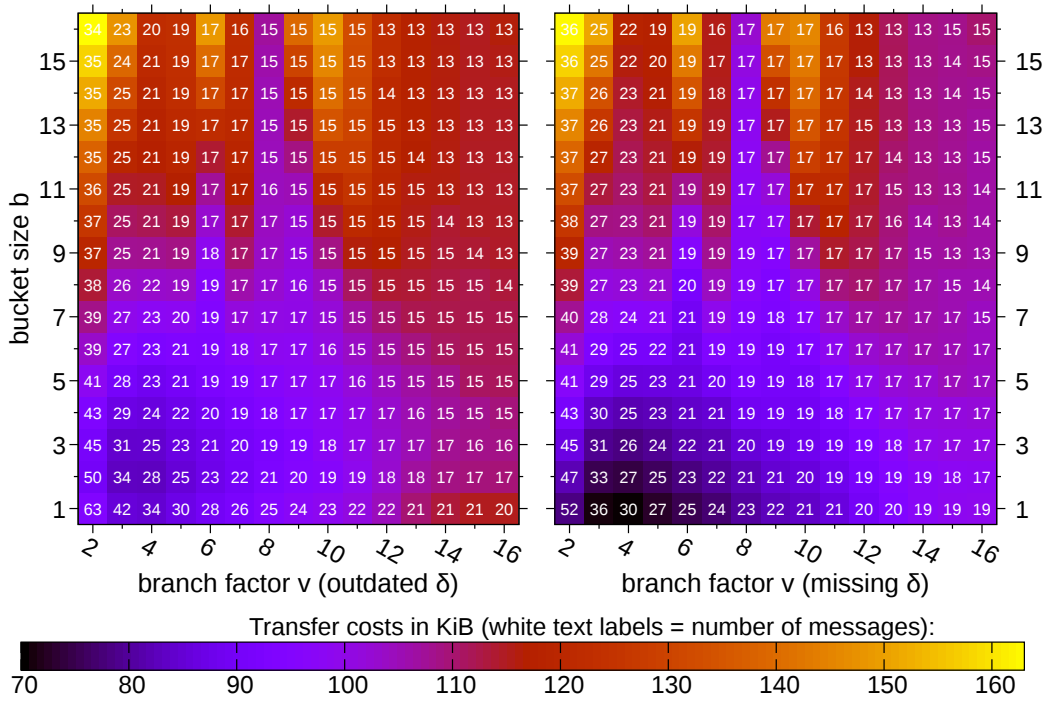


Figure 8.11: Merkle reconciliation transfer costs (heatmap) and number of messages (in white) with $FR = 0.1$ and different v and b for $\delta_{exp} = \delta = 2\,\%$.

on the participating processes and nodes (ref. Table 8.3, page 125), there is a trade-off between transmitting more hashes at lower sizes or fewer hashes with bigger sizes in deeper tree levels.

Transfer Costs

Regions with the lowest total cost in Figure 8.11 reside around $v \in \{3, 4\}$ with $b = 2$ and $b = 1$ for the *outdated* and *missing* scenarios, respectively, with slightly lower costs for $v = 3$ in both scenarios. From there on, higher or lower values of v or b generally increase transfer costs.

Figure 8.11 also shows some interesting patterns of the costs: For any fixed v, with increasing b, costs first strive towards an optimum where the savings in phase 1 (smaller trees) outweigh the additional costs of phase 2 (more items not in Δ) and then degrade again. For a fixed b, though, cost progressions show some irregularities, with multiple local optima. It seems that our protocol is more sensitive to v than to b due to the tree construction as outlined above. Also, since the actual tree structure varies with v, the fill grade among the leaf nodes—and thus phase 2 costs—may be affected, too.

Message Rounds

Since most of the protocol messages originate from the Merkle tree synchronisation (ref. Section 8.1) and the rest is in $\mathcal{O}(1)$, the number of message rounds may be given as roughly half the number of communication messages (numbers plotted into the heatmap of Figure 8.11), depending on the processes and the parallelism in phase 2 (ref. Section 8.1). This also reflects the Merkle tree height and, from the averages of 1 000 simulations as shown, we can confirm that both v and b reduce the height as expected. However, similarly to the transfer costs above, the interval-based construction of our Merkle tree dampens the height reduction of the two parameters. Also, since the height is in $\mathcal{O}(\log_v(n/b))$ (ref. Section 8.4.5), the biggest improvements between v and $v + 1$ or b and $b + 1$ exist with low b and v and, in general, the influence of v is bigger than the influence of b.

Different FR and δ

Other values of δ and FR do not seem to influence the region of parameters optimal with regards to transfer costs and are presented in Appendix A.2.2. Further details may also be extracted from the experiments below which shed some more light on the individual costs of the two phases and the actual failure rates of selected values of v and b.

8.7.2 General Analysis for Different δ and FR

Parameters v and b revisited

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 8.12 shows Merkle tree reconciliations with a fixed bucket size $b = 2$ and selected degrees v which extend the details compared to the parameter sweep above. In the simulations shown, the actual average failure rate is below the set FR in most cases but exposes an increased standard deviation that originates from hash collisions where whole sub-trees are wrongly identified as being equal. The *missing* scenario for $v = 2, \delta = 8\,\%$ even includes a case of a hash collision of the root hash which causes nothing to be reconciled. These events, however, are to be expected from the Merkle tree reconciliation. For the given values of δ, minimal transfer costs are achieved with $v = 4$. The smaller δ, the more $v = 3$ is achieving similar transfer costs but with more message rounds as shown above. For even lower δ, Figure A.12 in Appendix A.2.3 is showing that this trend persists. The larger δ, the more $v = 8$ becomes competitive since the additional costs for more—and in particular empty—leave nodes in the tree of phase 1 are compensated by phase 2 having fewer buckets completely filled and thus fewer items being transferred. This becomes more prominent in the *missing* scenario. For $\delta \to 100\,\%$, please refer to Figure A.13 in Appendix A.2.3 but we will also evaluate higher δ below.

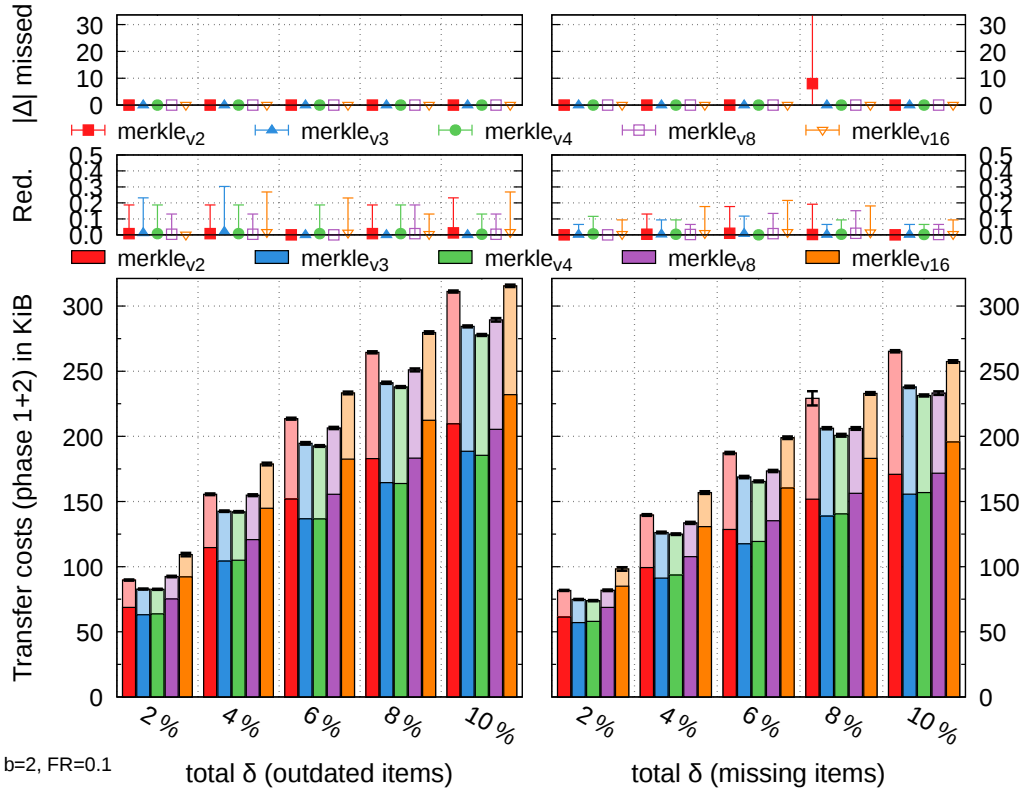


Figure 8.12: Merkle reconciliation for small δ, fixed $b = 2$ but varying v.

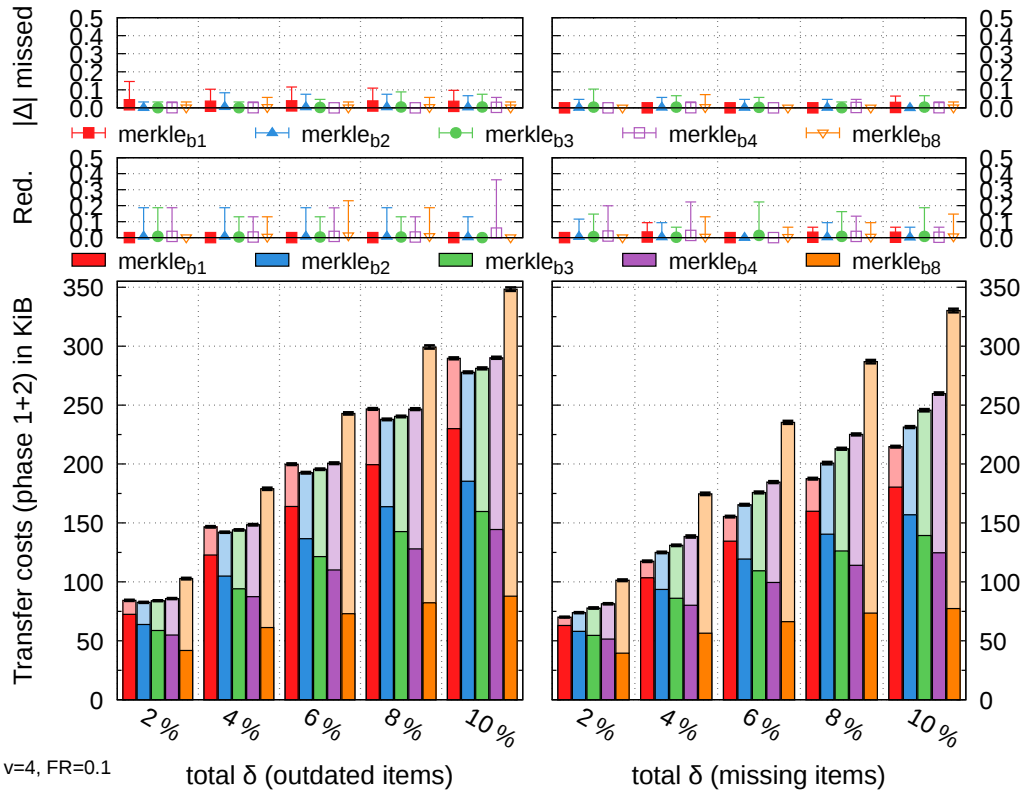


Figure 8.13: Merkle reconciliation for small δ, fixed $v = 4$ but varying b.

In contrast to above, we now fix v to the transfer-cost optimum $v = 4$ and vary the bucket size b, i.e. the maximum number of elements per leaf node, as shown by Figure 8.13. Higher b lead to smaller Merkle trees and reduced phase 1 costs (and message rounds) in all scenarios shown. The *trivial** reconciliations of phase 2, though, need to handle more items per leaf node at a higher cost. In the *missing* scenario, any $b > 1$ increases the overall transfer costs although the savings in phase 1 are similar to the *outdated* scenario. In contrast to the *outdated* scenario, however, if items are missing with $b = 1$ and are matched with an empty leaf node, we may directly resolve them without phase 2. This explains the overall lower phase 2 costs. Thus, the higher b, the fewer of these cases exist and the more phase 2 costs become similar to the *outdated* scenario. In the *outdated* scenario itself, $b = 2$ achieves the lowest overall transfer costs. Further values of δ are presented in Figures A.14 and A.15 in Appendix A.2.3 and confirm that this is also optimal for $\delta < 2\,\%$ and $\delta > 10\,\%$ with an exception for very high $\delta > 40\,\%$ in the *outdated* scenario. There, phase 2 costs can not be reduced much with higher b since (almost) all items are already being transferred.

☞ For all evaluations below, we will continue with $v = 4, b = 2$ which seems to be the best compromise from the experiments above.

Different target failure rates FR

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand}$

The influence of different values of FR on a Merkle reconciliation with $v = 4$ and $b = 2$ for typical $\delta \in (0, 10]\,\%$ and higher $\delta \in (0, 100]\,\%$ is shown in Figures 8.14 and 8.15, respectively, focussing on an overview of the costs and the progression with δ. Phase 1 as well as phase 2 costs increase logarithmically with FR^{-1} as predicted by eq. (8.10) (page 128) and indicated in the plots for any fixed δ (note the constant increase of costs with FR^{-1}; more details in Section 8.7.6 below). Also, except for a few hash collisions causing outliers with an impact based on the tree level of the collision, the accuracy is relatively stable, too, and below FR as expected[f]. This is due to our worst-case assumptions accounting for bigger-impact hash collisions that are compensated on average by other scenarios.

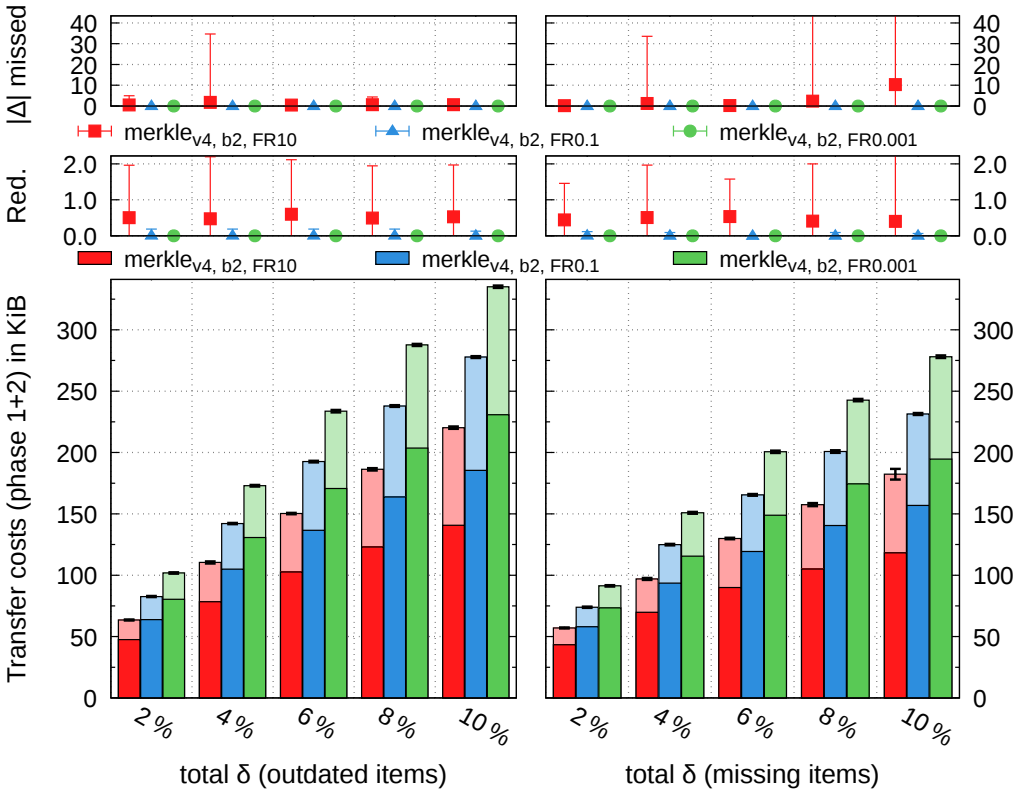


Figure 8.14: Merkle reconciliation ($v = 4$, $b = 2$) with small δ and varying FR.

To indicate the savings achieved by using an approximate Merkle reconciliation over the original one, Figure 8.16 compares the costs of the values of FR from above and the costs if instead full 160 bit hashes were used in both phases (merkle_{s160}). As shown, the 160-bit Merkle reconciliation is roughly 5 times as

[f] Please recall that hash collisions influence costs and accuracy similarly as shown and note that different relative sizes of the error bars only originate from different scales.

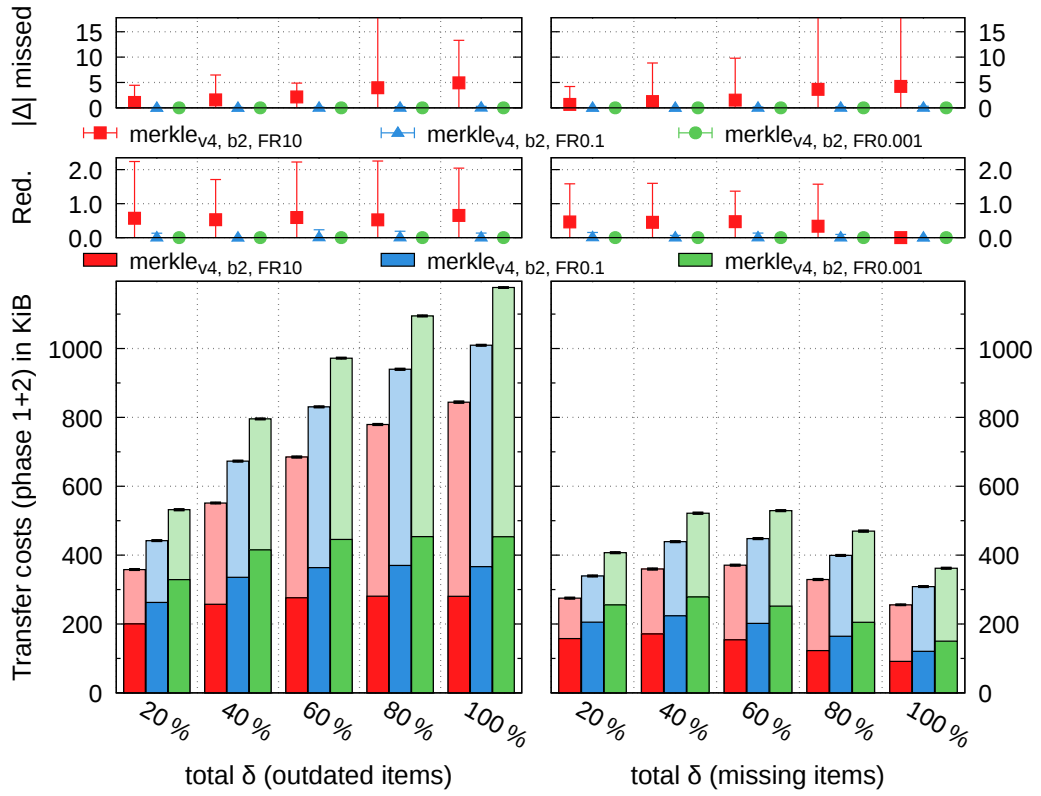


Figure 8.15: Merkle reconciliation ($v = 4$, $b = 2$) with high δ and varying FR.

expensive as our approximate Merkle tree with $FR = 0.001$. We could leverage this in a real system by reconciling more often using the approximate version but still having lower costs and shorter reaction times to failures.

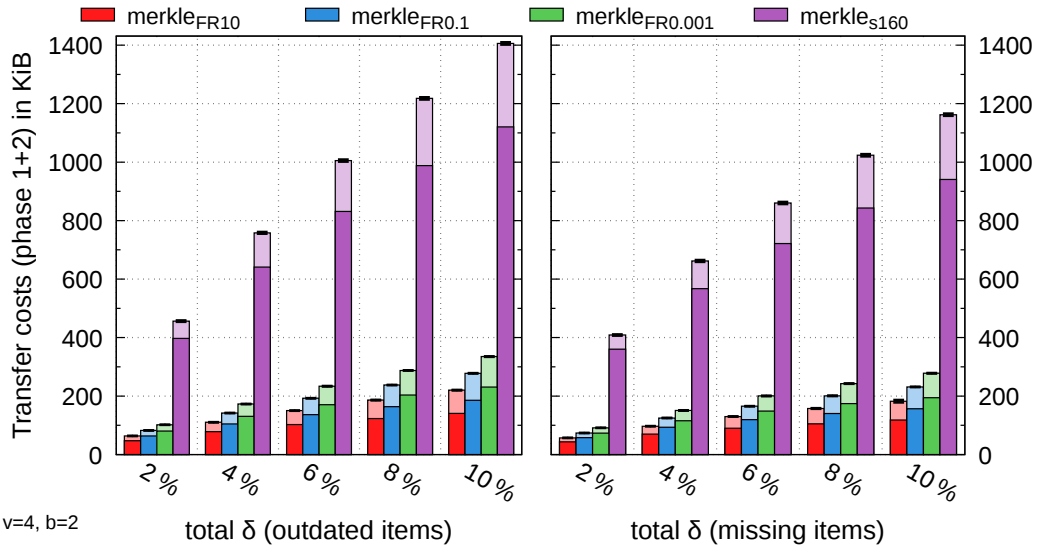


Figure 8.16: Approximate vs. original Merkle reconciliation ($v = 4$, $b = 2$).

8.7.3 What if δ_{exp} is Wrong?

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$ vs. 1 %
$fail_{rand}$

In the Merkle tree reconciliation protocol, δ_{exp} is used to reduce the bound τ on the number of items affected by a hash collision (ref. Section 8.4.1). Since we cannot assume a uniform distribution of the failures though, this only reduces τ for the upper levels of the tree (ref. Table 8.3, page 125). With δ_{exp} being lower than δ, as shown by Figure 8.17, the upper tree levels will use too few bits compared to what is necessary for FR. The chance of hash collisions increases and if they happen, $fr_{all}^{(lvl)}$ is lower than the actual value which causes further signature sizes to be too low. The biggest impact, however, is in the first tree levels and we expect minor cost savings as shown due to those tree levels only containing a small portion of the total number of tree nodes. As the difference between δ_{exp} and δ grows, so do the actual failure rates which eventually become larger than the set FR. Similarly, if δ_{exp} is higher than δ, costs marginally increase and failure rates decrease. Please refer to Appendix A.2.4 for further values of δ.

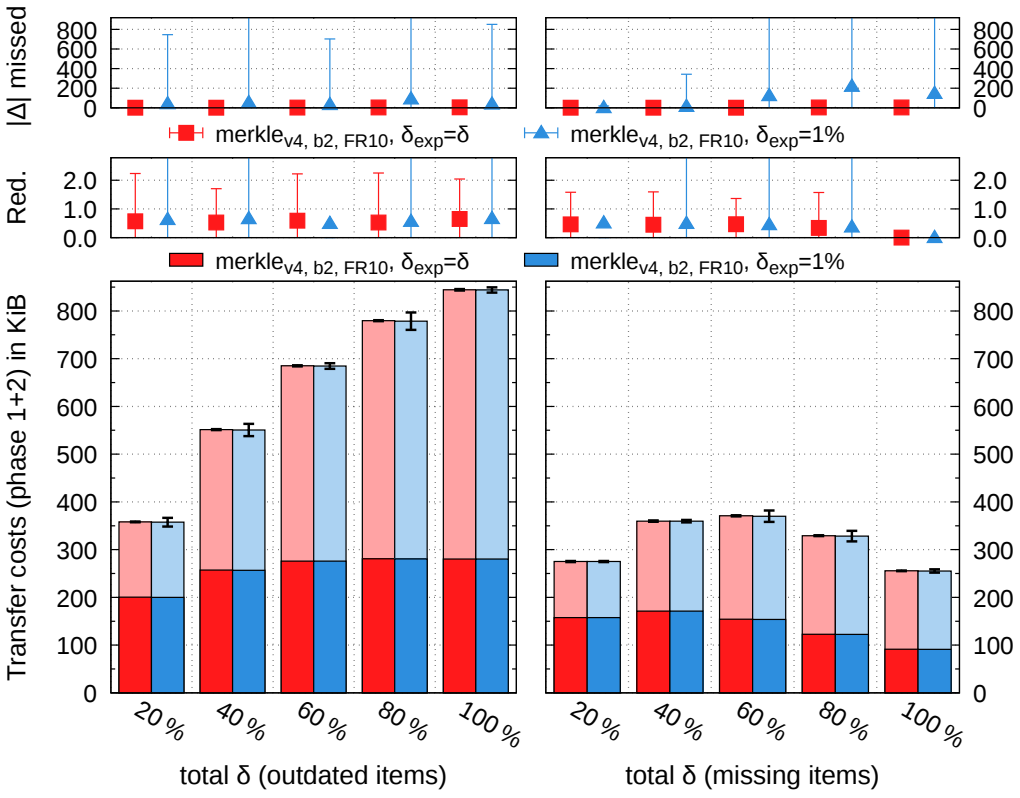


Figure 8.17: Merkle reconciliation with high δ and different δ_{exp}.

8.7.4 Data and Failure Distribution Sensitivity

$n = 100\,000$
$data_{rand,\ bin\ 0.2}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand,\ bin_{0.2}}$

In Section 8.5.2 above, we already saw the effects of a skewed data *and* failure distribution. Here, we will show the detailed differences of each individual distribution as well and present the results of our simulations including costs and observed failure rates. Since Merkle reconciliation skips over sub-trees with matching hashes, a transfer costs sensitivity to skewed failure distributions is expected. Figure 8.18 shows these differences (with slightly differently-plotted accuracy metrics than usual to which we will come below) and, consequently, presents up to 90 % lower transfer costs for the binomial failure distribution.

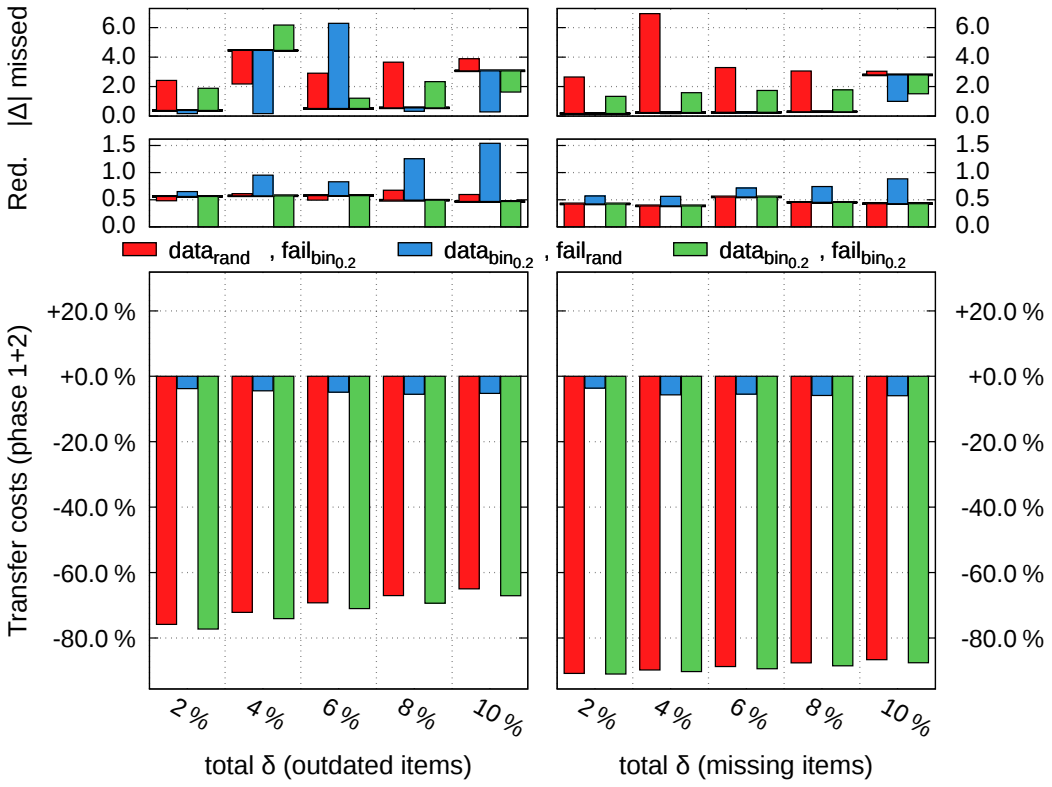


Figure 8.18: Merkle reconciliation with $FR = 10$ and different data and failure distributions compared to $data_{rand}, fail_{rand}$ ($v = 4$, $b = 2$).

A skewed data distribution influences the structure of the Merkle trees and in particular its height but since we re-distribute any unused FR, the costs of this in the reconciliation are negligible. In fact, we observe a roughly 5 % reduction of the costs that is caused by phase 1 being cheaper—while phase 2 is slightly more expensive—despite the violation of our assumption of equal number of items in each sub-tree of a level for τ (ref. Section 8.4). The changes in the costs of the two phases both originate from the binomial data distribution leading to higher bucket fill rates as discussed in Section 8.5.2.

As for the accuracy, Section 8.5.2 argued that we may observe reduced failure rates due to the broken uniformity assumption for τ which is why we account for more items being affected by a hash collision than actually present in some sub-trees. Although we did already choose $FR = 10$ for the evaluation here in order to be able to actually see any changes[g], Figure 8.18 does not show a general trend for the accuracy and the fluctuations are more likely caused by outliers. Also note that all observed failure rates (unidentified differences plus redundantly transferred items) are below the target FR.

Message Rounds

Although not in the focus of this thesis, we will briefly discuss the data and failure distribution's influence on the number of message rounds in addition to the discussion in Section 8.7.1 above. From there, please recall that the number of communication messages as logged by our simulations and given in Figure 8.19 is roughly double the number of message rounds and—except for 2 and 4 messages from phase 2 in the *outdated* and *missing* scenarios, respectively—is based on the tree height and reconciliation path in the Merkle tree. The influence of the reconciliation path may be seen from different values of δ for a fixed data distribution. With a uniform data and failure distribution, for example, lower δ cause slightly fewer messages being sent on average since the failures may not actually be present in the lowest tree level, depending on the interval splits of our construction algorithm (Algorithm 13, page 112).

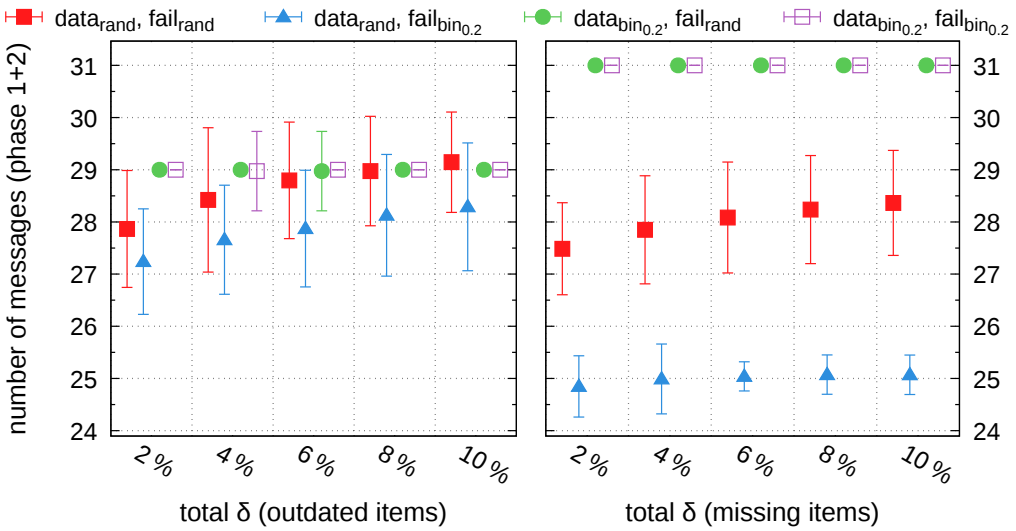


Figure 8.19: Message rounds in the Merkle reconciliation with $FR = 10$ and different data and failure distributions ($v = 4$, $b = 2$).

[g]Please also note that we slightly re-arranged the plot compared to the usual data and failure distribution plots above in order to be able to also see the absolute averages of the two accuracy metrics in the $data_{rand}, fail_{rand}$ scenario from which the others deviate.

A binomial data distribution is more stable than a uniform data distribution with regards to the standard deviation and the reconciliation path due to more items being present in the lowest tree levels. The fact that a binomial data distribution may lead to deeper trees may not be observed in the *outdated* scenario but only in the *missing* scenario. It seems that only the buckets' fill ratio of the lowest tree levels is skewed (0, 1, or 2 items per bucket with $b = 2$). In the *missing* scenario, this may, however, lead to more tree levels remaining compared to the uniform data distribution.

A change in the failure distribution only influences the reconciliation path in the *outdated* scenario and may also influence the tree structure in the *missing* scenario. It has a bigger impact on the uniform data distribution since there, compared to the uniform failure distribution, the possibilities for tree nodes vanishing from the lowest tree level or not being visited is higher. This effect is especially prominent in the *missing* scenario due to the bigger impact of the failure distribution. With a binomial data distribution, however, we do not observe any change since, there, a binomial failure distribution only affects how many buckets go into phase 2 because of the higher bucket fill rates in the failure region.

Remarks

Note that by adding a first randomisation phase as Byers et al. [10] propose, we could remove any differences from different data and failure distributions and effectively achieve the same results as with $data_{rand}$ and $fail_{rand}$.

8.7.5 Scalability with the System Size n

$n = variable$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 8.20 shows the results of the Merkle reconciliation for $v = 4$ and $b = 2$ with smaller and larger numbers of items than above and a fixed δ. Since we include n into the hash size calculations (ref. Section 8.4), there should be no changes in the accuracy for different n. For small n, however, hash collisions in the first tree levels, i.e. with the biggest impact as shown by $|\Delta|$ *missed*, are more likely to occur in the 1 000 simulations shown. For higher n, they cause more failures and—to fulfil FR—they thus need to be less likely and are apparently not caught by our number of simulations. Costs increase with $\mathcal{O}(n \cdot \log n)$ for $n \to \infty$, $|\Delta| \in \mathcal{O}(n)$ and FR, b constant as expected by eq. (8.10) (page 128) and indicated by a slight relative increase towards the naïve transfer costs[h] for $n = 4\,096\,000$ compared to $n = 4\,000$. Eventually, the number of bits needed to distinguish different items under FR is higher than the number of bits available for the keys. In that case, however, we may need larger item key hashes to distinguish the items anyway.

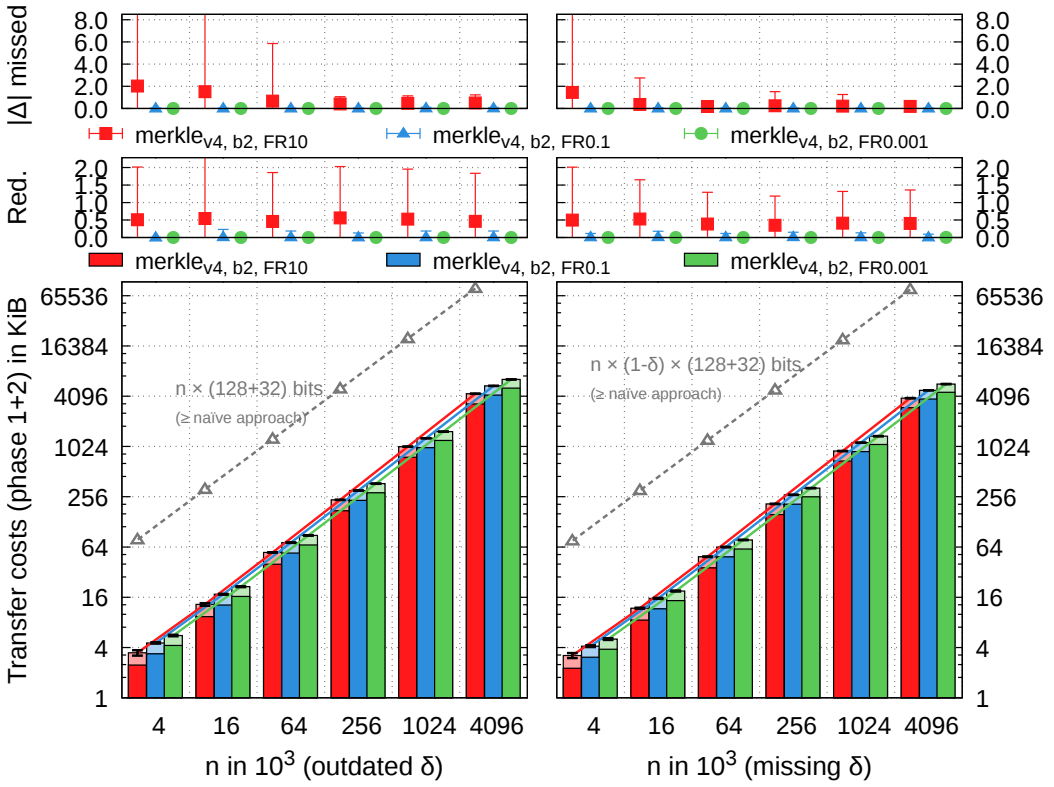


Figure 8.20: Merkle reconciliation ($v = 4$, $b = 2$) scalability with data size n and different FR.

[h] Note that Merkle reconciliation uses a maximum of 160 bits for its tree hashes as opposed to the 128 bit hashes used by the algorithms above and the naïve transfer costs as presented here for comparison.

8.7.6 Scalability with the Target Failure Rate FR

$n = 100\,000$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Figure 8.21 shows how the communication costs of the Merkle reconciliation ($v = 4$, $b = 2$) evolve with different target failure rates FR and fixed n which extends the observations from the evaluations above. Regarding the accuracy, we verify that the target FR holds and that the accuracy is, on average, much lower than FR due to our worst-case assumptions and, especially, the inability to make use of δ_{exp} in lower levels of the tree, using a common τ for all nodes of a tree level, or assuming distinct items between the leaves of phase 2 (also ref. Section 8.5). For FR below a certain threshold, though, the given values are not representative anymore since the probability of any failure occurring at all is too low to always be visible in our 1 000 simulations. The results for the costs, however, are representative in all scenarios and show a progression with $\mathcal{O}(\log(1/FR))$ for n, $|\Delta|$, and b constant as expected by eq. (8.10) (page 128).

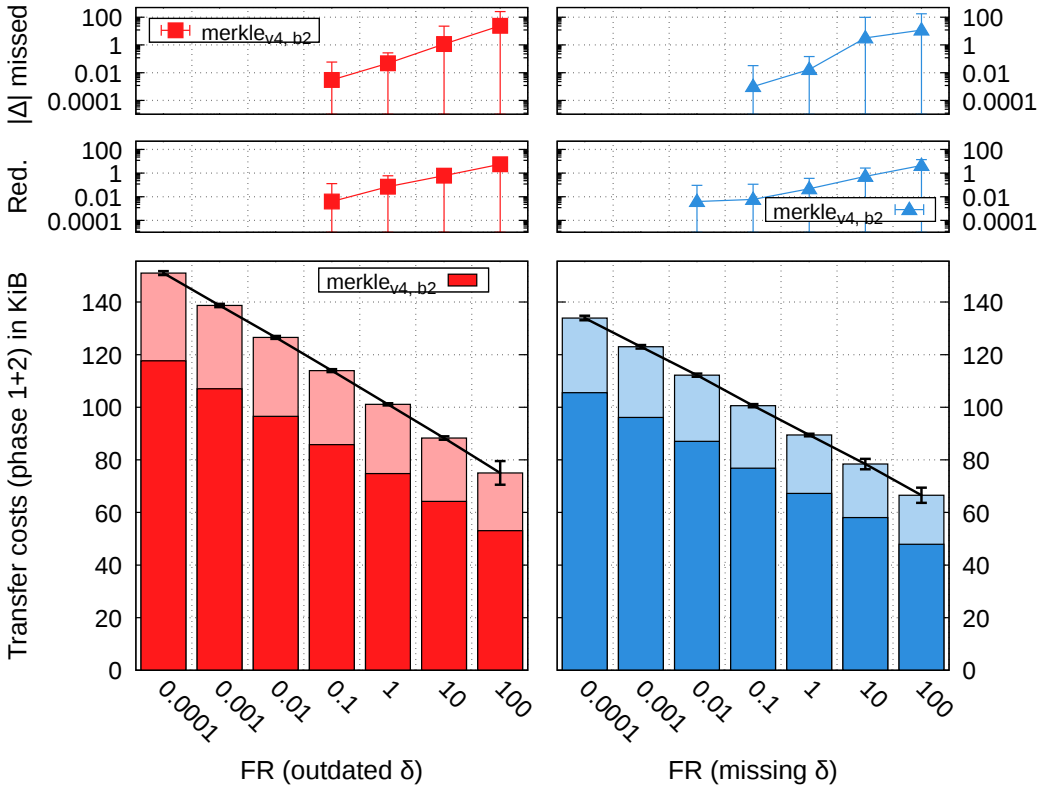


Figure 8.21: Merkle reconciliation ($v = 4$, $b = 2$) scalability with the target failure rate FR ($\log_{10}$ scale on the x-axis and on the y-axis except for the transfer costs).

Chapter 9

Comparative Evaluation

Chapters 5 to 8 applied our accuracy model from Section 2.5 to four different set reconciliation algorithms and individually evaluated their performance with respect to costs and accuracy in different scenarios, i.e. different δ and FR, inaccuracies in δ_{exp}, different data and failure distributions, and the scalability in both the system size n and the target failure rate FR. This chapter aims at comparing these four algorithms altogether and verifying that the accuracy model's main goal of offering a method to fairly compare different set reconciliation algorithms is achieved.

From a theoretical point of view, the complexity of the communication costs of both phases, as summarised by Table 9.1, is only slightly different. After considering $|\Delta| \in \mathcal{O}(n)$, we may even subsume $\mathcal{O}\left(n \cdot \log({}^{n}/_{FR})\right)$ for all of them. The following sections will thus evaluate the differences of the empirical evaluations described above. We will also compare our algorithms with the unoptimised rsync implementation of Chapter 4 that uses a blocksize of 64 bits and full 128 bit hashes, i.e. $\text{rsync}_{64,cs128}$, to put the results into perspective[a].

Table 9.1: Communication costs complexity for $n, |\Delta| \to \infty,\ FR \to 0$.

Rsync	$\mathcal{O}(n)$	(ref. Section 4.3.3)
Trivial	$\mathcal{O}\left(n \cdot \log \frac{n}{FR} + \lvert\Delta\rvert \cdot \log n\right)$	(ref. eq. (5.7), page 57)
SHash	$\mathcal{O}\left(n \cdot \log \frac{\lvert\Delta\rvert}{FR} + \lvert\Delta\rvert \cdot \log \frac{n \cdot \lvert\Delta\rvert}{FR}\right)$	(ref. eq. (6.12), page 76)
Bloom	$\mathcal{O}\left(n \cdot \log \frac{\lvert\Delta\rvert}{FR} + \lvert\Delta\rvert \cdot \log \frac{\lvert\Delta\rvert}{FR}\right)$	(ref. eq. (7.16), page 97)
Merkle (perfectly balanced)	$\mathcal{O}\left(\frac{n}{b} \cdot \log \frac{n}{FR} + \lvert\Delta\rvert \cdot \log \frac{\lvert\Delta\rvert}{FR}\right)$	(ref. eq. (8.10), page 128)

[a] Please recall that we could apply rsync's optimisations to our algorithms as well and then compare against an ordinary rsync implementation with similar results.

9.1 General Analysis for Different δ and FR

$n = 100\,000$
$data_{rand}$
$\delta = variable$
$\delta_{exp} = \delta$
$fail_{rand}$

Figures 9.1 and 9.2 show the results for all algorithms with $\delta \in (0, 10]\,\%$ and $\delta \in (0, 100]\,\%$, respectively, with a common $FR = 10$ as their target rate[b]. For very low δ, we can observe the degradation of SHash reconciliation costs into a *trivial* reconciliation. The failures, however, are considerably reduced compared to the *trivial* reconciliation since here, in phase 2, we wrongly assume $\delta = 100\,\%$ and thus choose higher hash sizes based on the assumption of phase 1 filtering out common items. This also explains the slightly increased costs compared to the *trivial* reconciliation alone. Bloom and Merkle reconciliation do not expose a degradation and thus scale better for lower δ, even if the proposed SHash degradation fix (ref. page 83) was applied.

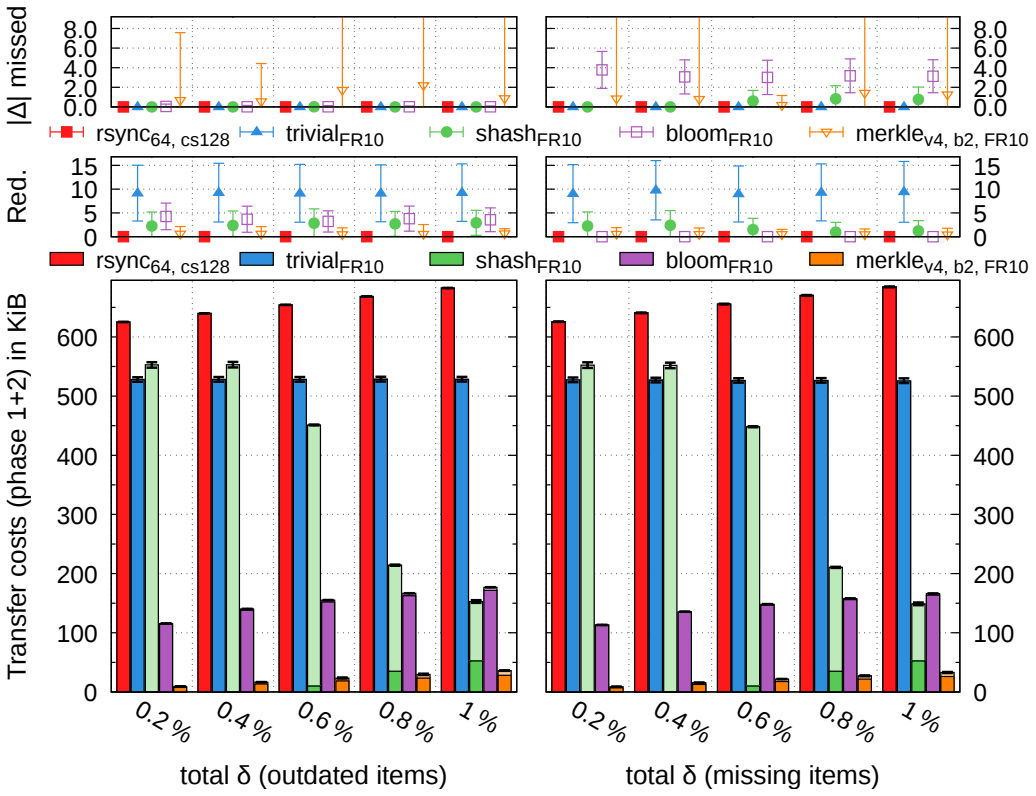


Figure 9.1: Comparison for $FR = 10$ with small differences $0.2\,\% \leq \delta \leq 1\,\%$.

Only the *trivial* reconciliation is able to reach the target FR on average and thus does not waste resources, for both ranges of δ. For the other algorithms, the worst-case(s) seem to occur more rarely but might have bigger impacts when present. Since we always account for them, we thus waste resources in

[b] Other target rates than $FR = 10$ lead to similar results than the ones given. Plots showing their results for $2\,\% \leq \delta \leq 10\,\%$ are given in Appendix A.3.1. We chose to present $FR = 10$ here to show the fluctuations of the Merkle reconciliation in the 1 000 simulations.

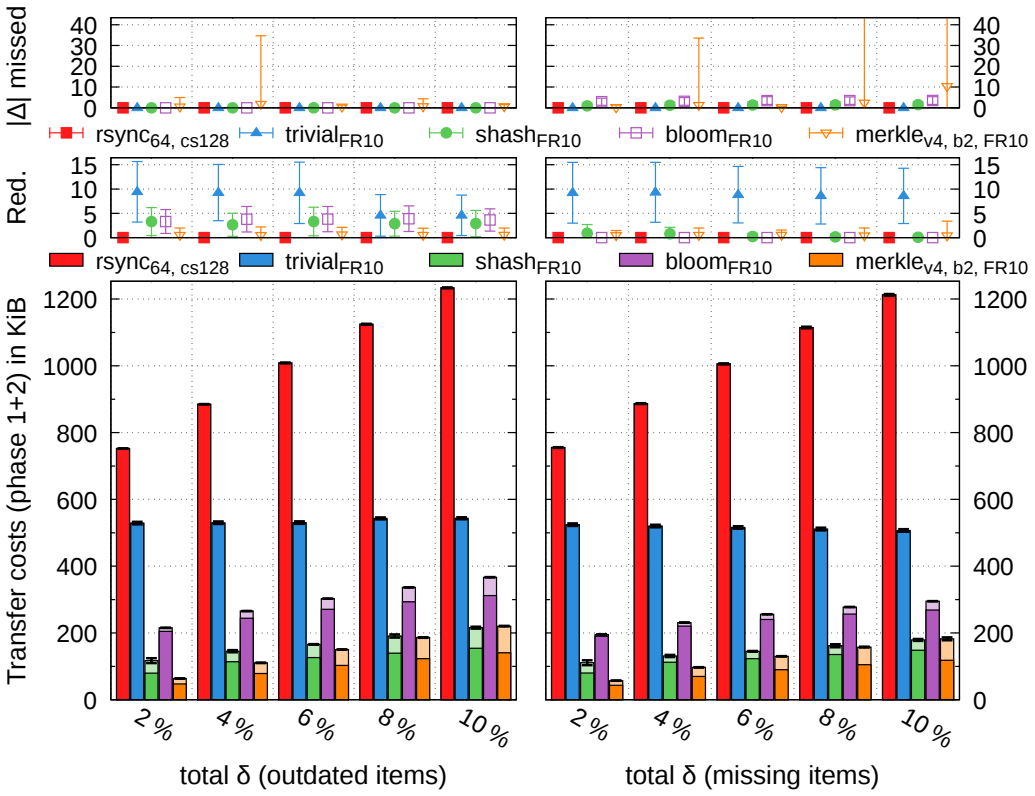


Figure 9.2: Comparison for $FR = 10$ with differences $2\,\% \leq \delta \leq 10\,\%$.

the average case. Merkle reconciliation has the highest fluctuations (shown by the standard deviation) due to hash collisions in higher tree levels leading to many more failures than a single hash collision in a Bloom filter, for example.

Regarding the costs, the lower δ, the higher the advantage of the Merkle reconciliation over the other algorithms (at the cost of an increased number of message rounds, ref. Section 8.7). SHash is the cheapest for any $\delta \gtrapprox 9\,\%$ although the difference to Merkle reconciliation is small (also ref. Figure 9.3 below). Among the $\mathcal{O}(1)$-round algorithms, SHash has the lowest costs, except for $\delta < 1\,\%$ where Bloom filters are better and SHash degrades. For $\delta \leq 10\,\%$, the *trivial* reconciliation costs are always lower than the unoptimised $\text{rsync}_{64,cs128}$ but also always higher than our other approximate set reconciliation algorithms and 3-5 times more expensive than the cheapest $\mathcal{O}(1)$-round one. The higher δ, the more blocks are different for any rsync-based algorithm and the more expensive it is. Eventually, it degrades to sending the whole key-version list plus all block hashes and is more expensive than the naïve algorithm transferring only the key-version list which is roughly 2 MiB here (also ref. Section 4.3).

For even higher δ—which may be considered an uncommon case—, SHash remains the cheapest algorithm until the *trivial* algorithm achieves lower costs around $\delta \approx 50\,\%$ in the *outdated* scenario and $\delta \approx 70\,\%$ in the *missing* scenario (ref. Figure 9.3). Eventually, depending on the sizes of the payload behind the

set reconciliation, it may even be cheaper to simply transfer all values. With more items being different, hash collisions leading to redundant item transfers as in the *trivial* reconciliation may actually transmit an item to update/regenerate and not cause a redundant transfer after all. Therefore, especially in the *missing* scenario, failures are reduced with higher δ. For Merkle reconciliation, with higher δ the worst-case assumption of a common τ for all nodes of a tree level is closer to the actual presence of failures and may lead to failure rates closer to the target FR.

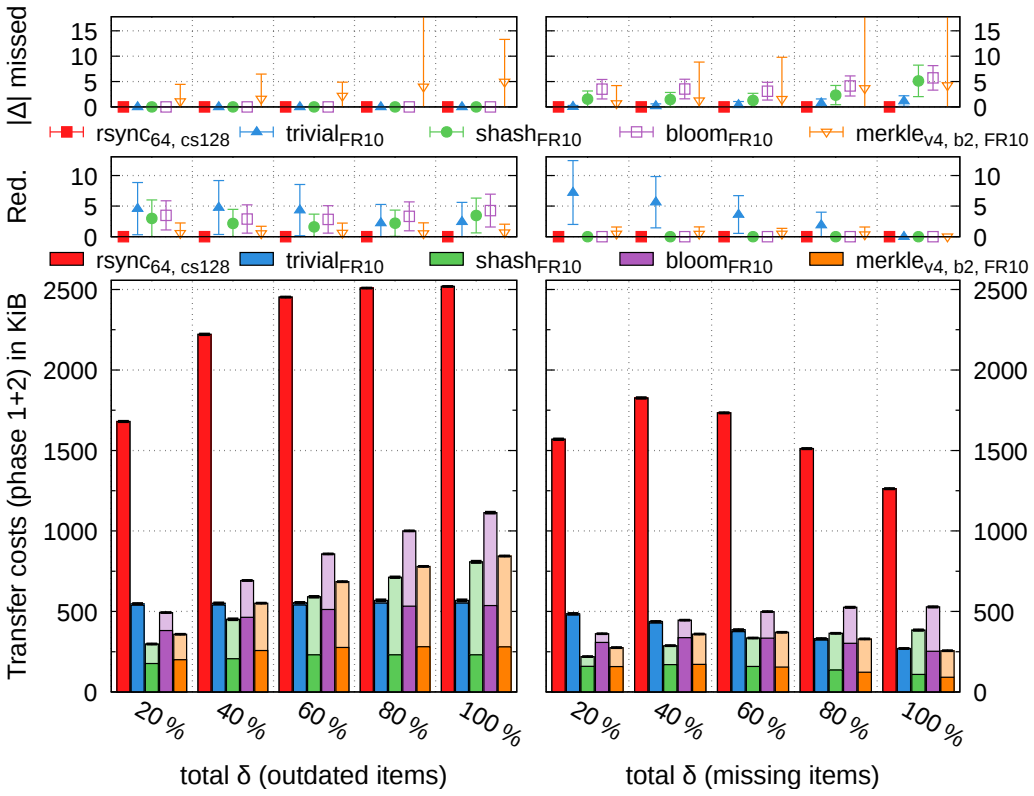


Figure 9.3: Comparison for $FR = 10$ with high differences $20\,\% \leq \delta \leq 100\,\%$.

9.2 What if δ_{exp} is Wrong?

$n = 100\,000$
$data_{rand}$
$\delta = 100\,\%$
$\delta_{exp} = \delta$ vs. $1\,\%$
$fail_{rand}$

A crucial part in each of our approximate algorithms is the correctness of δ_{exp} which we assume based on Section 2.4. To see the effects of an incorrect δ_{exp}, Figure 9.4 presents both accuracy metrics (with a $\log_{10}$ scale) as well as the transfer costs (linear scale) of the difference of $\delta_{exp} = 100\,\%$ versus $\delta_{exp} = 1\,\%$ in scenarios with $\delta = 100\,\%$ (further values of δ may be found in the individual evaluations above). Results are given grouped by algorithm on the x-axis (without rsync since it does not use δ_{exp}).

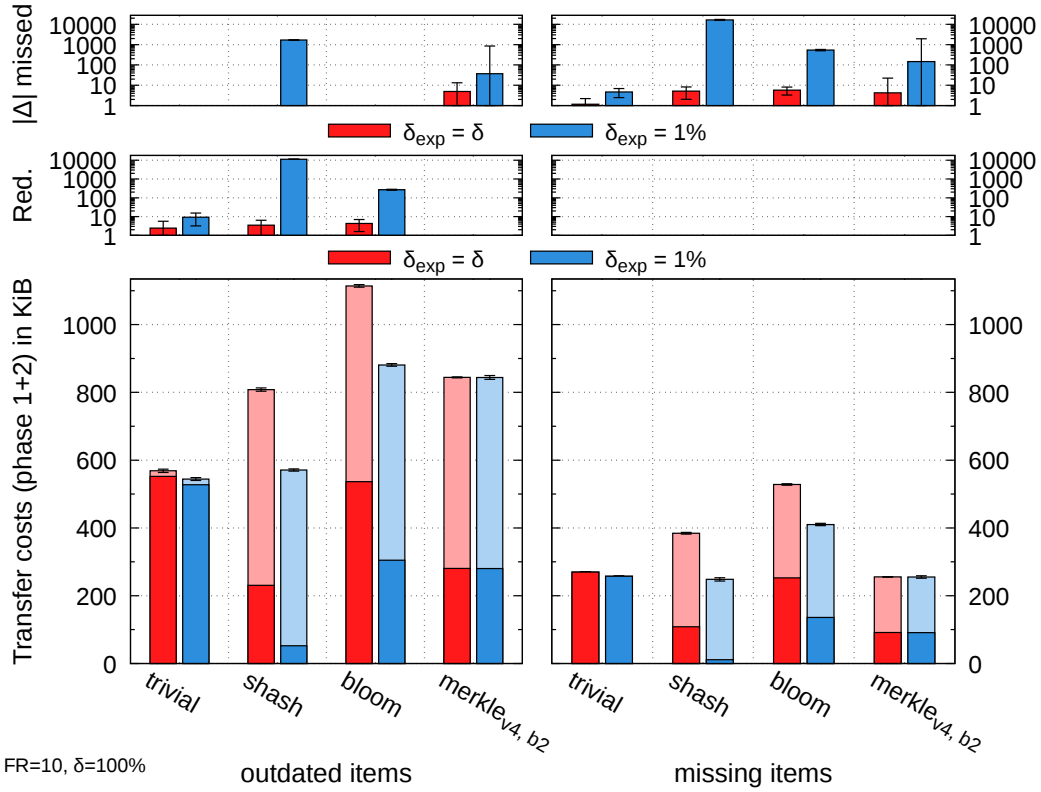


Figure 9.4: Comparison for correct $\delta_{exp} = 100\,\%$ vs. too low $\delta_{exp} = 1\,\%$ for $\delta := 100\,\%$ and each of our four approximate set reconciliation methods.

All of our approximate set reconciliation methods have reduced costs for lower δ_{exp} but also a lower accuracy. *Trivial* reconciliation is the least affected one due to $\tilde{n}$ only changing from 100 000 to 50 252 (ref. Section 5.6.2) while SHash's is most affected with its collision sets' sizes $\tilde{n}_{X \in \{\mathtt{A}_\Delta, \mathtt{B}_\Delta\}}$ changing from 100 000 to 1 006 (ref. Section 6.6.2) for $\delta_{exp} = 100\,\%$ and $\delta_{exp} = 1\,\%$, respectively. Although having the same collision sets, Bloom filters retain a higher accuracy than SHash due to a more sophisticated data structure with more fine-grained control (number of bits in total vs. number of bits per item). Merkle tree reconciliation seems even less sensitive to wrong δ_{exp} but please recall that

changes in δ_{exp} mostly affect the upper tree levels where hash collisions cause many errors but are made unlikely. With even more simulations than 1 000, more collisions may manifest and thus possibly also show a higher average number of failures. At the same time, however, the savings in the costs are negligible since the upper tree levels do not have too many hashes (ref. Section 8.7.3).

9.3 Data and Failure Distribution Sensitivity

$n = 100\,000$
$data_{rand,\ bin\,0.2}$
$\delta = 10\,\%$
$\delta_{exp} = \delta$
$fail_{rand,\ bin_{0.2}}$

As shown by Figure 9.5, skewed data and failure distributions only marginally affect the accuracy of our approximate set reconciliation algorithms, mostly caused by the normal fluctuations due to the randomness of our simulations. At least the target FR should never be exceeded since we always accounted for the worst case in the setup of the algorithms.

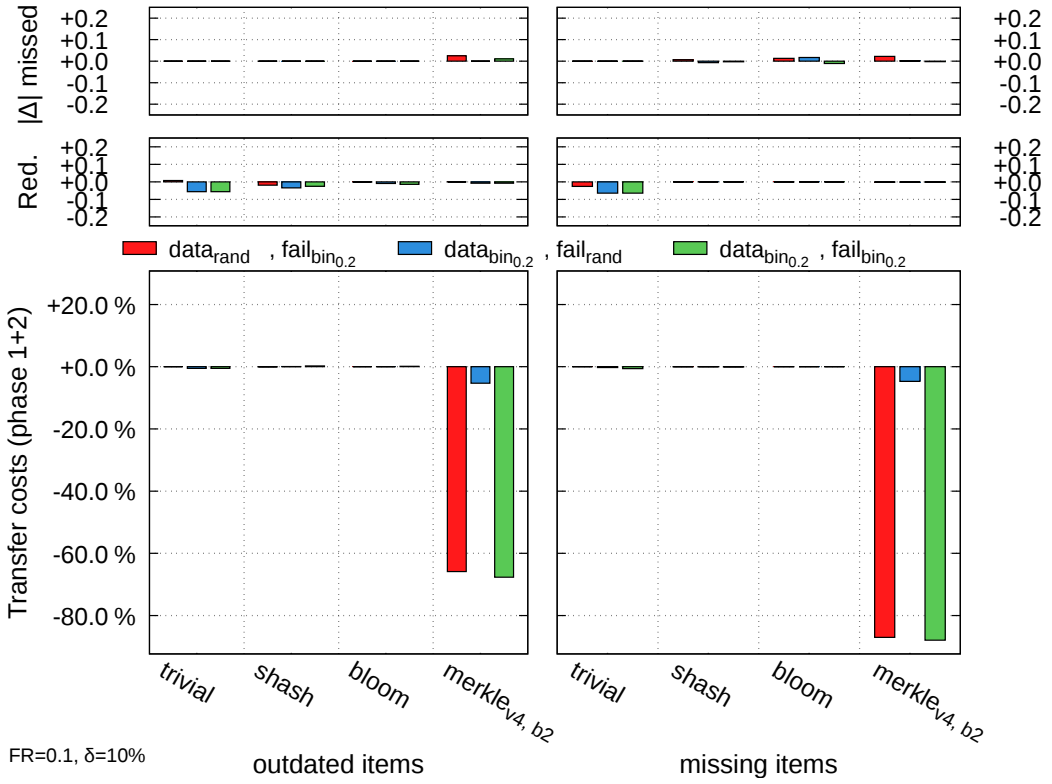


Figure 9.5: Influence of different data and failure distributions on each algorithm relative to $data_{rand}, fail_{rand}$.

On the other hand, skewed data and failure distributions may have a bigger effect on the costs as shown for the Merkle tree reconciliation. There, the structure of the Merkle tree depends on the data distribution resulting in a roughly 5 % *decrease* of the costs compared to a uniform distribution due to

higher bucket fill rates and despite an increased tree height (ref. Section 8.7.4, also for a discussion on the implications on the number of message rounds). Since skewed failure distributions—as the binomial distribution shown here—result in failures clustering in certain regions of the tree, more sub-trees with common items may be skipped during the reconciliation. The fewer the number of failures, the higher the effect and here, for $\delta = 10\,\%$, costs are reduced by almost 70 % and 90 % for the *outdated* and *missing* scenarios, respectively. In contrast, *trivial*, SHash, and Bloom filter reconciliations are only marginally affected due to their hash-based data structure which effectively compensates for different data or failure distributions.

9.4 Scalability with the System Size n

$n = variable$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Although the communication costs complexity for all our set reconciliation algorithms is the same, i.e. $\mathcal{O}\left(n \cdot \log(n/FR)\right)$ for $n \to \infty,\ FR \to 0$ (ref. Table 9.1, page 149 with $|\Delta| \in \mathcal{O}(n)$), Figure 9.6 shows the practical differences of these algorithms with increasing n but keeping FR and δ constant. Firstly, the accuracy seems unaffected by increasing n as expected, since we include n into the hash size calculations. Please note that although missed

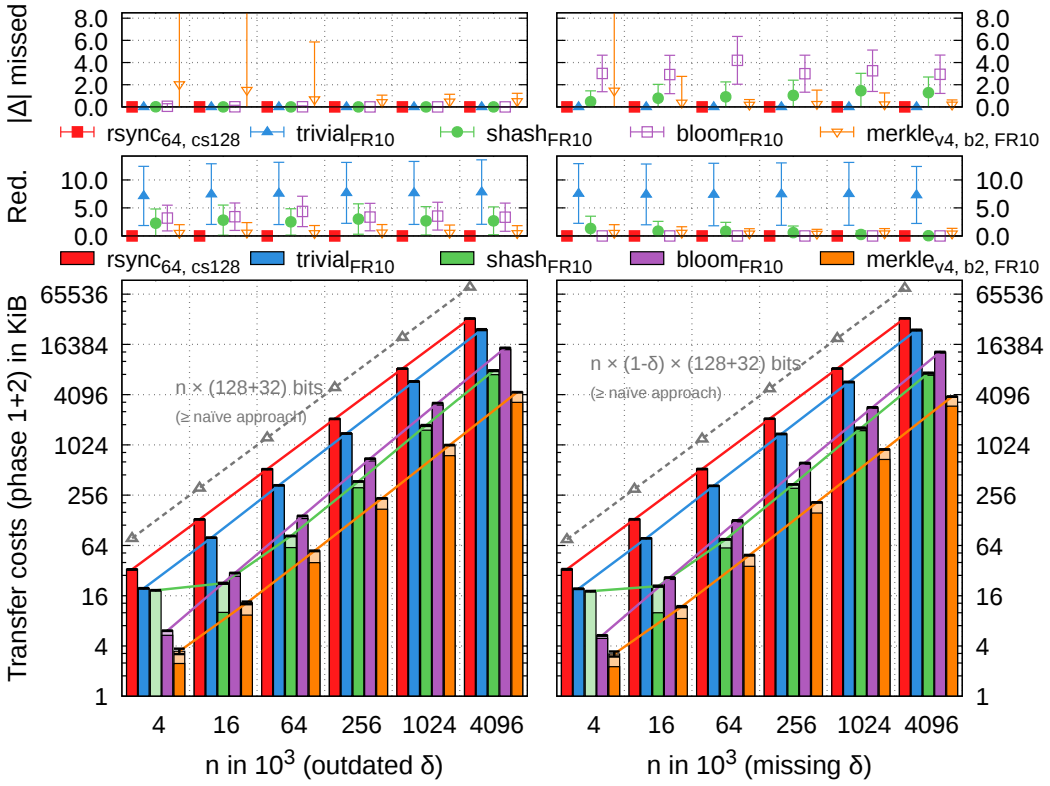


Figure 9.6: Scalability for $FR = 10$ with data size n ($\delta = 3\,\%$).

Δ for Merkle reconciliation seems to decrease with n, this is just an artefact of bigger-impact hash collisions being too unlikely to occur within our 1 000 simulations (also ref. Section 8.7.5).

Regarding the costs, we do see a linear progression from the naïve and rsync reconciliations (please note the $\log_4$ scale on both axes). The costs of all approximate set reconciliations is increasing towards these and thus has a steeper slope due to the $\mathcal{O}(n \cdot \log(n))$ costs complexity here. *Trivial* and Merkle reconciliation exhibit the lowest slope and thus scale better with n than Bloom filters and SHash who progress more quickly towards the naïve algorithm. Also note the SHash degradation for low n as discussed in Chapter 6. Table 9.2 presents a subset of this plot in numbers, i.e. the overall transfer costs of the *outdated* scenario, to make this effect more visible. With these numbers and the factors between different n, we are able to see that *trivial* scales even slightly better than the Merkle reconciliation, albeit at a 5-6 times higher costs, still valid for $n = 4\,096\,000$. They also reveal a sub-linear increase of costs for the Merkle reconciliation for $n = 4\,000$ vs. $n = 16\,000$ which is, however, only caused by smaller trees not being as efficient as larger trees caused by—not only—our interval splits (also ref. Sections 8.5 and 8.7).

Table 9.2: Total transfer costs (in KiB) with data size n (*outdated* items with $\delta = 3\,\%$, $FR = 10$).

$n/10^3$	Naïve	*Trivial*	SHash	Bloom	Merkle$_{v4,b2}$
4	$\leq$ 78.1	19.6	18.6	6.1	3.5
	x 4.00	x 4.00	x 4.03	x 1.21	x 4.92	x 3.80
16	$\leq$ 312.5	79.1	22.6	29.9	13.1
	x 4.00	x 4.00	x 4.25	x 3.68	x 4.83	x 4.20
64	$\leq$ 1250.0	336.2	82.9	144.4	55.1
	x 4.00	x 4.00	x 4.18	x 4.50	x 4.85	x 4.27
256	$\leq$ 5000.0	1404.7	373.6	700.1	235.1
	x 4.00	x 4.00	x 4.20	x 4.69	x 4.64	x 4.33
1024	$\leq$ 20000.0	5894.7	1753.0	3248.1	1019.4
	x 4.00	x 4.00	x 4.18	x 4.54	x 4.58	x 4.31
4096	$\leq$ 80000.0	24625.0	7955.2	14874.3	4388.9

9.5 Scalability with the Target Failure Rate FR

$n = 100\,000$
$data_{rand}$
$\delta = 3\,\%$
$\delta_{exp} = \delta$
$fail_{rand}$

Similarly to the scalability with the system size n, if keeping n fixed and looking at the algorithms for $FR \to 0$, all our approximate reconciliation algorithms have the same communication costs complexity of $\mathcal{O}(\log(1/FR))$ (ref. Table 9.1, page 149 with $|\Delta| \in \mathcal{O}(n)$). The practical differences are shown by Figure 9.7 where we observe Merkle reconciliation costs increasing the least from $FR = 100$ towards $FR = 0.0001$. Bloom filters costs, on the other hand, increase the most and do not scale well with $FR \to 0$. Except for the SHash degradation for high FR (and/or low n) as shown, SHash and *trivial* costs scale similarly due to similar techniques being used.

Regarding accuracy, we verify that all our approximate set reconciliation algorithms fulfil the configured target FR with some deviations for low FR in the *trivial* reconciliation. Please note, however, that in order to be representative even for low FR, we would need more than 1 000 simulations. Additionally to the low FR, this is also algorithm-specific since failures may have different severity and thus even lower probabilities of occurring each. As a result, the average (absolute) number of failures—as shown—may get too high or too low depending on whether failures occur or not.

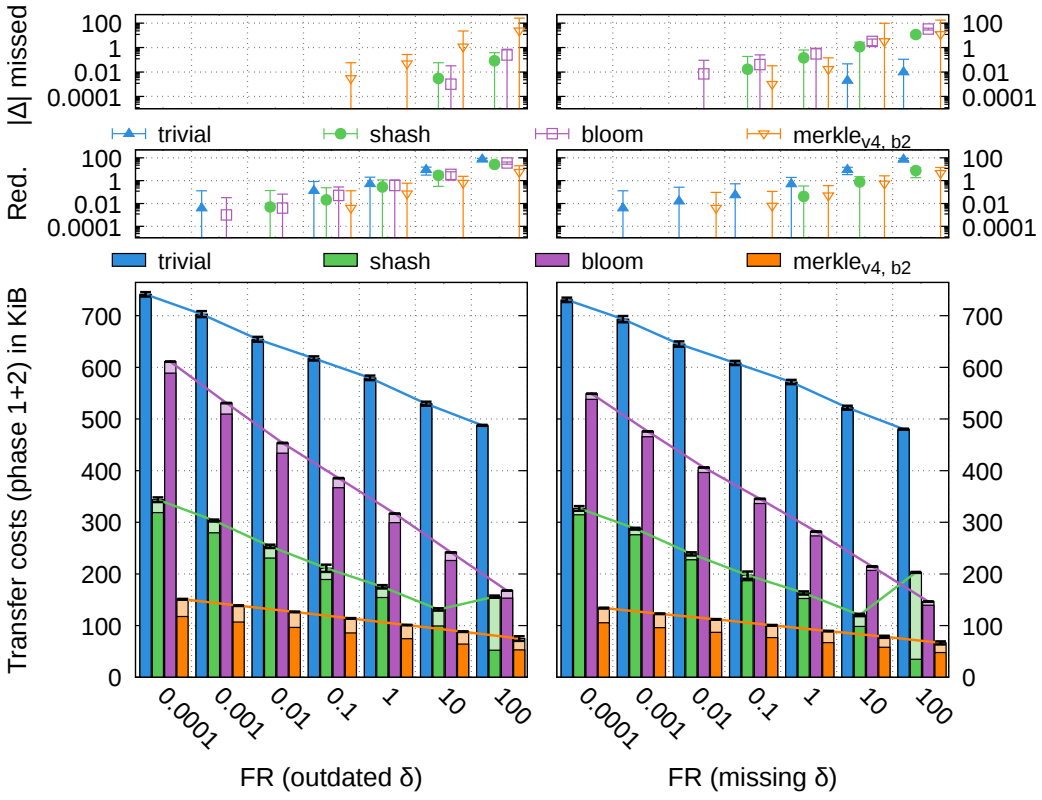


Figure 9.7: Scalability with the target failure rate FR ($\log_{10}$ scale on the x-axis and on the y-axis except for the transfer costs).

9.6 Applicability and Limitations of our Accuracy Model

In the analysis and evaluations above, we have shown that our accuracy model of enforcing a fixed (and close) upper bound FR on the failure rate (per set reconciliation; ref. Section 2.5) works well for the given set reconciliation algorithms and reduces the number of accuracy-influencing parameters to just one, i.e. FR. This common parameter allows the user to define a scenario-independent accuracy that an algorithm should obey. It thus allows a fair comparison of different algorithms as shown above. Without FR, a fair comparison would be difficult since each data point may require different parameters for the algorithms in order to achieve a similar accuracy. This is especially true for the evaluation of the scalability in the system size n but also applies to fixed n where different values of δ may require different parameters.

The scenario-independence of our upper-bound accuracy model, however, also implies using worst-case assumptions in many cases which lead to the actual average failure rates being well below FR in various scenarios. This effect becomes more apparent the more advanced the algorithms and their data structure become and is most visible in the Merkle tree based set reconciliation (ref. Figure 9.7 above). For any given scenario, e.g. a specific item or failure distribution, we may be able to tune the algorithms' accuracy-influencing parameters in a way such that the *average* failure rate fulfils FR, instead. This would be a fair comparison for a *specific* scenario, while our approach is more generic and fair for *any* scenario. With the scenario-specific tuning, we can expect reduced costs for otherwise too accurate algorithms as shown above.

We have shown how to apply our accuracy model to four different approximate set reconciliation algorithms but are certain it can be applied to a greater set of (approximate) algorithms (ref. Section 2.3) due to its generality. However, more complex formulae for the calculations of the accuracy as for the counting Bloom filters (eqs. (2.11) and (2.12), page 23), for example, may hinder a direct analysis and calculations of optimal parameters. In these cases, we may use other techniques such as sampling to find accuracy-influencing parameters that closely fulfil FR.

Chapter 10

Conclusion

In this thesis, we presented a novel and generic accuracy model for approximate set reconciliation algorithms that allows a fair comparison. This model accounts for both types of failures during set reconciliation, i.e. false-negatives (missing differences) and false-positives (superfluous differences which lead to unnecessary item transfers when resolved), and enforces an upper bound on these failures.

We have shown our accuracy model's generality by applying it to four different approximate set reconciliation algorithms: two simple hash-based approaches, one based on Bloom filters and one based on Merkle trees. For each of these algorithms, we derived any accuracy-influencing parameter(s) from a common failure bound FR and thus reduced the number of accuracy-influencing parameters to just one which is also easier to handle. By applying our accuracy model, we also created a new (approximate) variant of a Merkle tree based set reconciliation algorithm with adjustable accuracy. Since this variant includes a second phase for reconciling Merkle tree buckets, it also sheds some new light on the bucketing of items in the Merkle tree's leaf nodes. Normally, grouping more than one item there results in increased costs if only a few of them actually differ. With our implementation, bucketing savings in the Merkle tree do not only reduce the number of message rounds but may also outweigh the costs of the second phase.

We analysed and evaluated each of the resulting four algorithms extensively in terms of their accuracy, bandwidth usage, and scalability to verify the correctness and usefulness of our accuracy model. As expected, all algorithms achieve an accuracy within the given bound but may not get too close to this bound in a specific scenario since we decided to follow this bound even in worst case scenarios. However, if desired—and more details about the application scenario of an algorithm, e.g. a failure distribution, are known—we may be able to adjust our accuracy model and the algorithm to generate a less generic but more efficient variant.

Regarding the transfer costs, all algorithms are within $\mathcal{O}\left(n \cdot \log(n/FR)\right)$ with Bloom filter reconciliation even achieving $\mathcal{O}\left(n \cdot \log(|\Delta|/FR)\right)$ and SHash being between these two (ref. Table 9.1 on page 149 for more details). In the practical

evaluation, however, the multi-round Merkle tree reconciliation achieves an order of magnitude lower transfer costs in low-difference scenarios which eventually level out with the SHash costs around $\delta \approx 10\,\%$. Two more results stand out: (a) as opposed to an ordinary (almost-exact) Merkle tree reconciliation which was said not to be efficient beyond $\delta = 5\,\%$ compared to the naïve approach [14], our optimised approximate Merkle tree reconciliation has lower transfer costs for up to $\delta \approx 50\,\%$ differences compared to its counterpart, the approximate *trivial* reconciliation, and (b) except for the SHash degradation for low δ and/or high FR, SHash is more efficient than the more advanced (and more CPU-intensive) Bloom filter. Further investigations may find a variant that attenuates the degradation but ultimately, only Bloom filters are able to efficiently encode each item with less than a few bytes (or even one byte) each. Figure 10.1 shows a concise summary of which algorithms perform best depending on different values of δ.

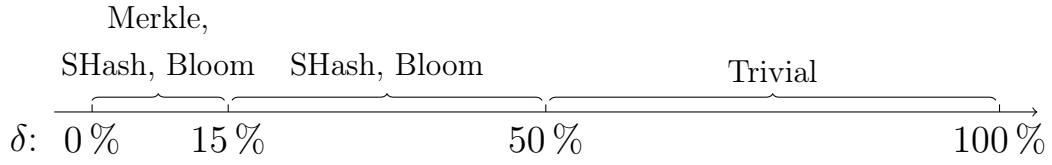


Figure 10.1: Best overall algorithms for different δ based on their empirical communication costs.
(Please recall that Merkle tree reconciliation uses more round-trips than the others and that SHash degrades for low n, δ, and/or high FR.)

A final note about the usefulness of having accuracy-limited approximate set reconciliation algorithms: these (cheap) algorithms may be used at high frequencies by a set reconciliation service which may also combine them with exact reconciliations at lower frequencies to achieve quick reactions to failures at low costs and catch any missing items from the approximate algorithms at defined intervals. It may however already be sufficient to only execute the approximate algorithms at higher frequencies. To avoid running into the same collisions among different executions of the same algorithm, though, each execution should use a different salt for their hash function which may be added to all contents before being hashed.

Appendix A

Additional Plots and Code

A.1 Naïve and Rsync Reconciliation

```
crypto:start().
Random = fun(Min, Max) -> crypto:rand_uniform(Min, Max) end.
Stats =
 fun([]) -> {0, 0, 0, 0};
    ([_|_] = TList) ->
     {Len, Sum, Sum2, Min, Max} =
      lists:foldl(fun(E, {L, X1, X2, Min, Max}) ->
                    {L + 1, X1 + E, X2 + E * E,
                     min(E, Min), max(E, Max)}
                  end, {0, 0, 0, 0, 0}, TList),
     % pay attention to possible loss of precision here:
     {Sum / Len, math:sqrt((Len * Sum2 - Sum * Sum) / (Len *
        Len)), Min, Max}
 end.
ToInt = fun(S) -> list_to_integer([X || X <- S, X =/= $,]) end.

KMax = 16#10000000000000000000000000000000. % 128 bits
VMax = 16#100000. VDiffMax = 512. % 20 bits
N = 100000. Deltas = [0,2,4,6,8,10]. Repeats = 100.
{ok, Pat} = re:compile("sent ([0-9,]+) bytes\s+received ([0-9,]
   +) bytes\s+[0-9,\.]+ bytes/sec"),

A = lists:ukeysort(2, [{I, Random(0, KMax), Random(1, VMax)}
                       || I <- lists:seq(1, N)]), N = length(A).
[spawn(fun() ->
  receive {go, Src} -> ok end,
  BName= lists:flatten(io_lib:format("B_~w~B",[FType, Delta])),
  AName= lists:flatten(io_lib:format("A_~w~B",[FType, Delta])),
  {Naive, RSDef, RSRest} = lists:unzip3(
   [begin
     io:format("~B ", [J]),
     {Bin0, Bin1} = lists:unzip(
      [begin
        KV1 = <<K:128, V1:32>>,
        if
```

```
          I >= (Delta * N) div 100 + 1 ->
           {KV1, KV1};
          FType =:= upd ->
           V2 = max(0, V1 - Random(1, VDiffMax)),
           KV2 = <<K:128, V2:32>>,
           case Random(0, 2) of 0 -> {KV1, KV2}; 1 -> {KV2, KV1}
               end;
          FType =:= reg ->
           KV2 = <<>>,
           case Random(0, 2) of 0 -> {KV1, KV2}; 1 -> {KV2, KV1}
               end
        end
       end || {I, K, V1} <- A]),
     BBin = list_to_binary(Bin1), ABin = list_to_binary(Bin0),
     Naive = byte_size(zlib:compress(BBin)),
     [RS32, RS64, RS128, RSDef] =
      [begin
        file:write_file(BName, BBin),
        file:write_file(AName, ABin),
        RSync = os:cmd("rsync -Iz --no-W " ++ BS ++ " --stats "
            ++ BName ++ " " ++ AName),
        {match, [_, Sent0, Rcv0]} = re:run(RSync, Pat, [{
           capture, all, list}]),
        ToInt(Sent0) + ToInt(Rcv0)
       end || BS <- ["-B32", "-B64", "-B128", ""]],
     {Naive, RSDef, {RS32, RS64, RS128}}
    end || J <- lists:seq(1, Repeats)]),
  {RS32, RS64, RS128} = lists:unzip3(RSRest),
  Src ! {Delta, FType, Naive, RSDef, RS32, RS64, RS128},
  file:delete(AName), file:delete(BName)
 end) ! {go, self()}
 || Delta <- Deltas, FType <- [upd, reg]], ok.

[receive {Delta, FType, Naive, RSDef, RS32, RS64, RS128} ->
  io:format("~n~w ~B~n", [FType, Delta]),
  [begin
    {ok, File} = file:open(io_lib:format("~w_~w_~3..0B.dat", [
       Alg, FType, Delta]), [write]),
    io:fwrite(File, "~p ~p ~B ~B", tuple_to_list(Stats(RC))),
    file:close(File)
   end || {Alg, RC} <- [{naive, Naive}, {rsync, RSDef}, {
      rsync32, RS32}, {rsync64, RS64}, {rsync128, RS128}]]
 end || Delta <- Deltas, FType <- [upd, reg]], ok.
```

Listing A.1: Naïve and rsync simulation script (Erlang 18.0 code).

A.2 Merkle Tree

A.2.1 Effective Worst-Case Accuracy

Uniformly Distributed Item Keys and Failures – *Outdated* Scenario

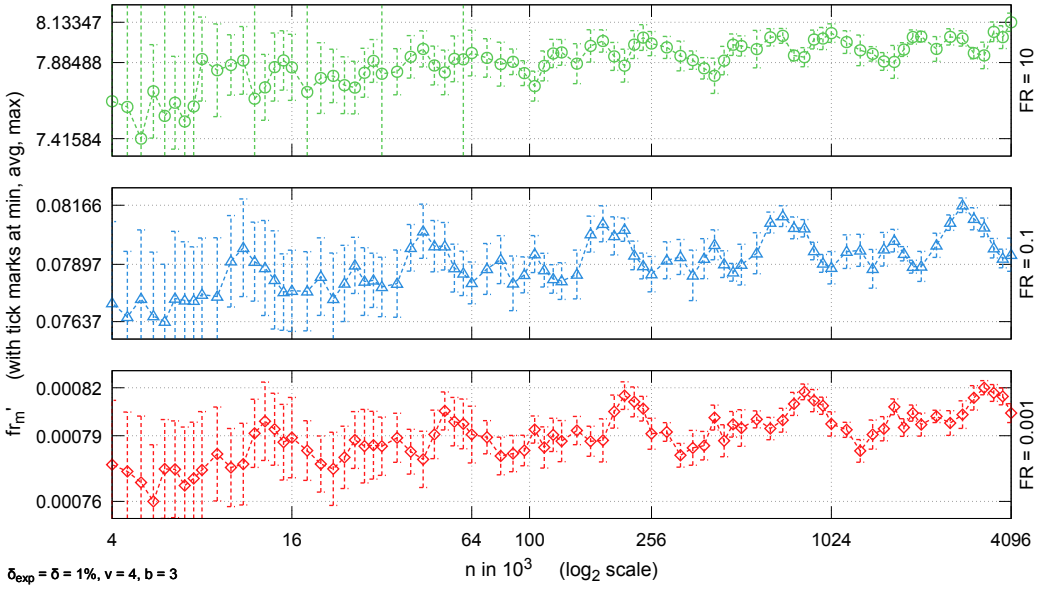


Figure A.1: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m, *outdated* scenario, $\delta = 1\,\%$, different FR and n (error bars = standard deviation).

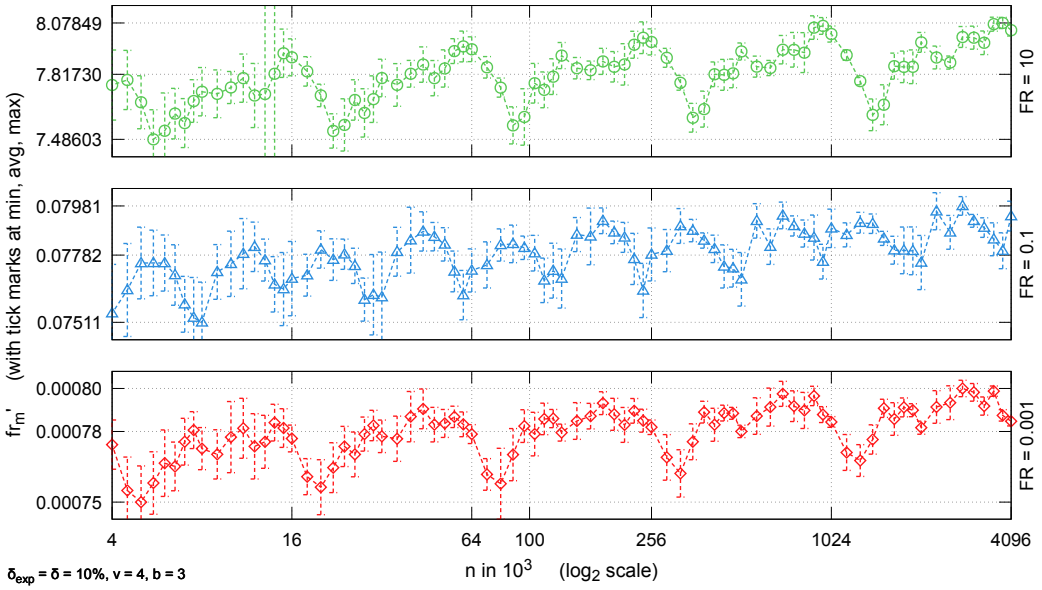


Figure A.2: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m, *outdated* scenario, $\delta = 10\,\%$, different FR and n (error bars = standard deviation).

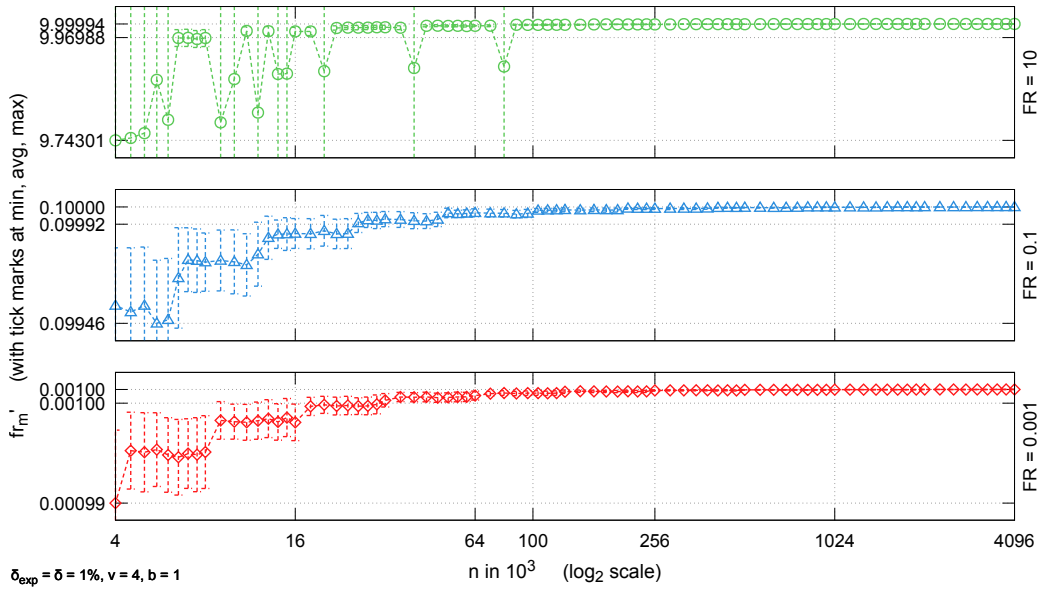


Figure A.3: Merkle $v = 4, b = 1$ effective worst-case failure rate fr'_m, *outdated* scenario, $\delta = 1\,\%$, different FR and n (error bars = standard deviation).

Uniformly Distributed Item Keys and Failures – *Missing* Scenario

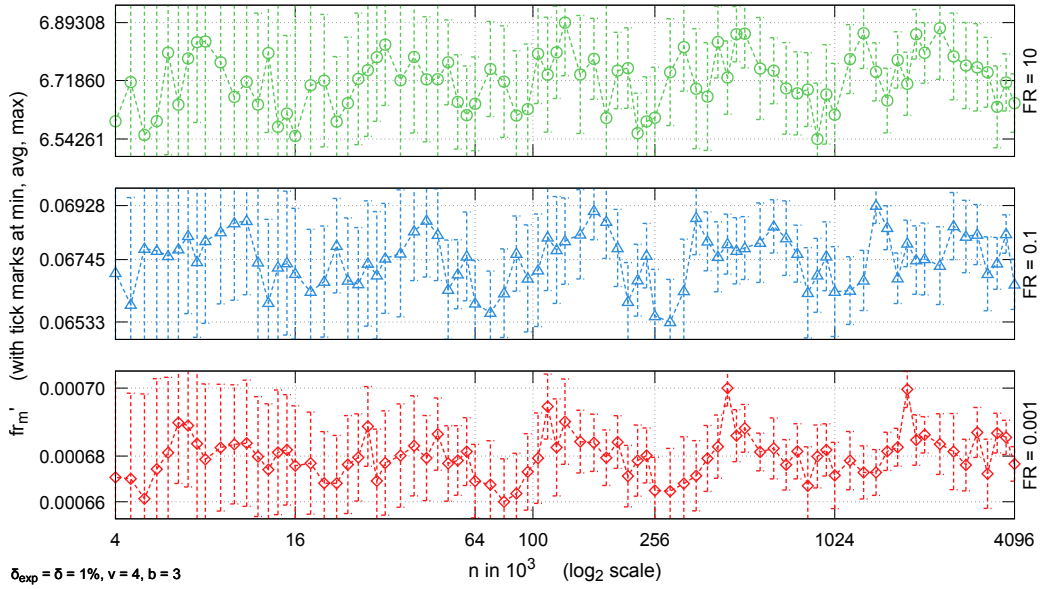


Figure A.4: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m, *missing* scenario, $\delta = 1\,\%$, different FR and n (error bars = standard deviation).

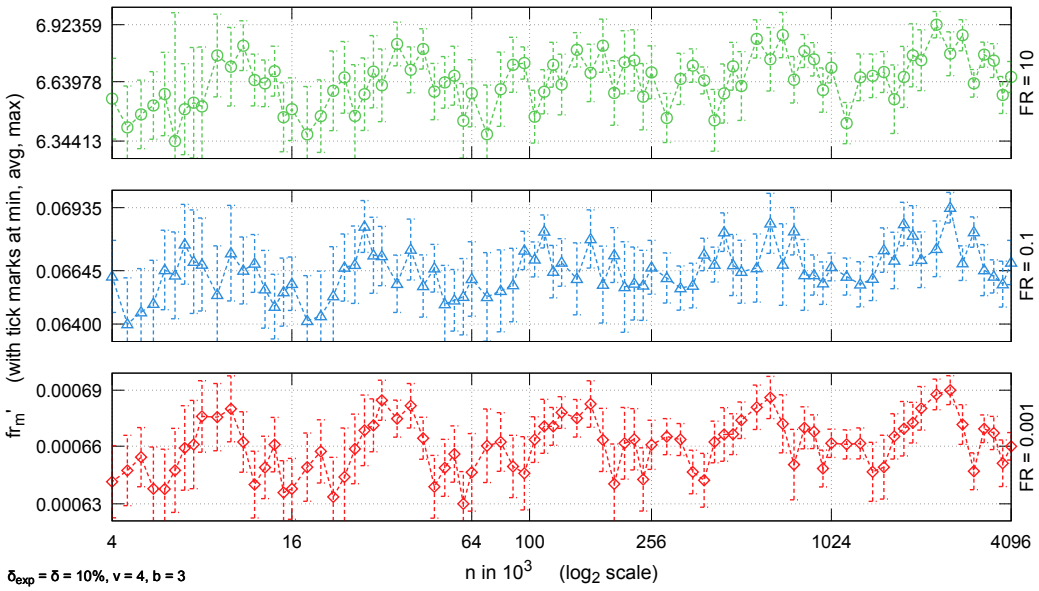


Figure A.5: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m, *missing* scenario, $\delta = 10\,\%$, different FR and n (error bars = standard deviation).

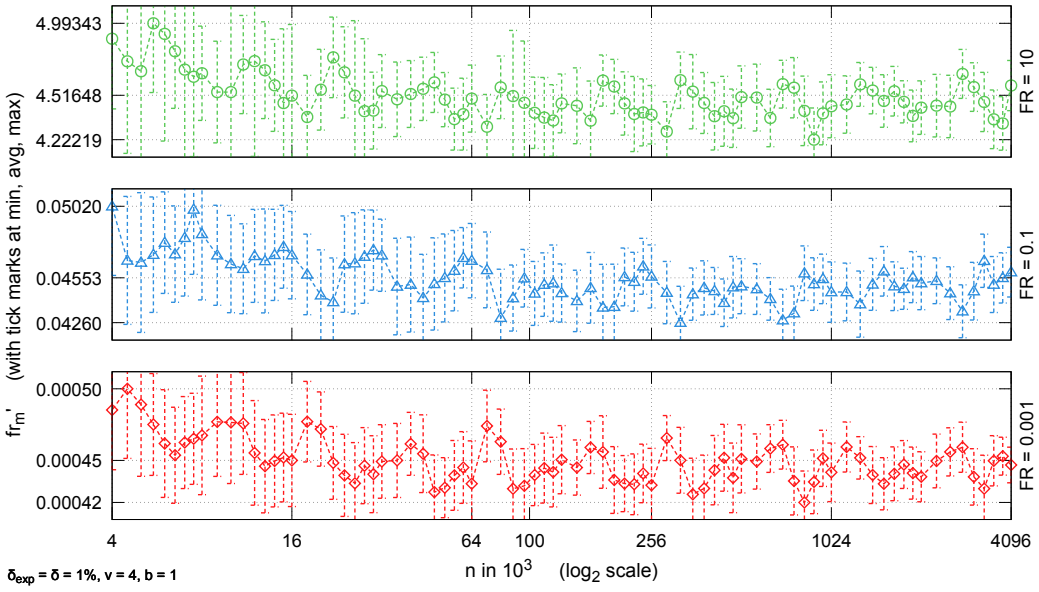


Figure A.6: Merkle $v = 4, b = 1$ effective worst-case failure rate fr'_m, *missing* scenario, $\delta = 1\,\%$, different FR and n (error bars = standard deviation).

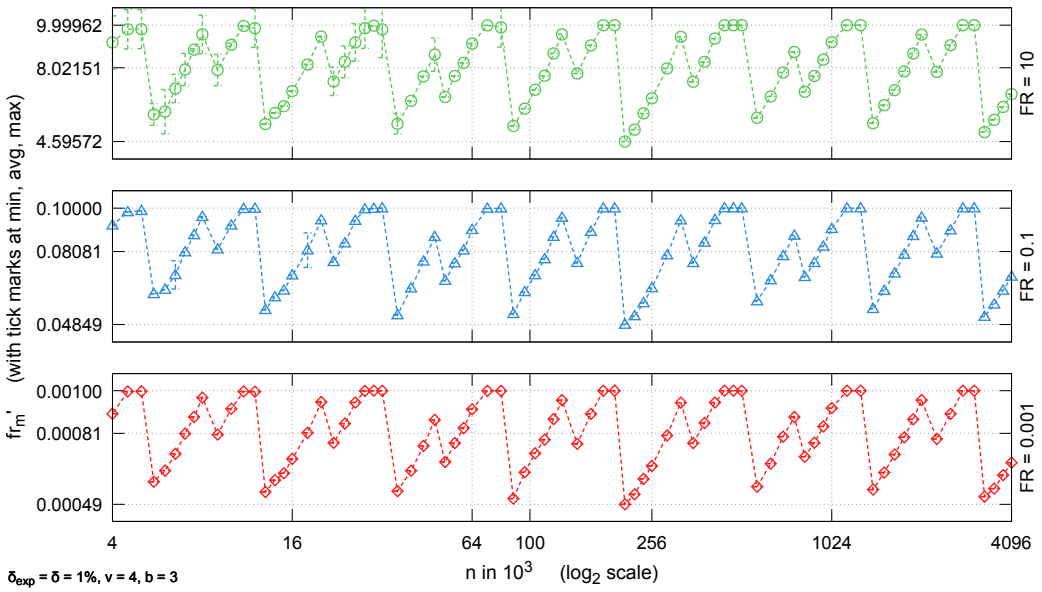


Figure A.7: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m, *outdated* scenario for different FR and n with a binomial data and failure distribution (error bars = standard deviation).

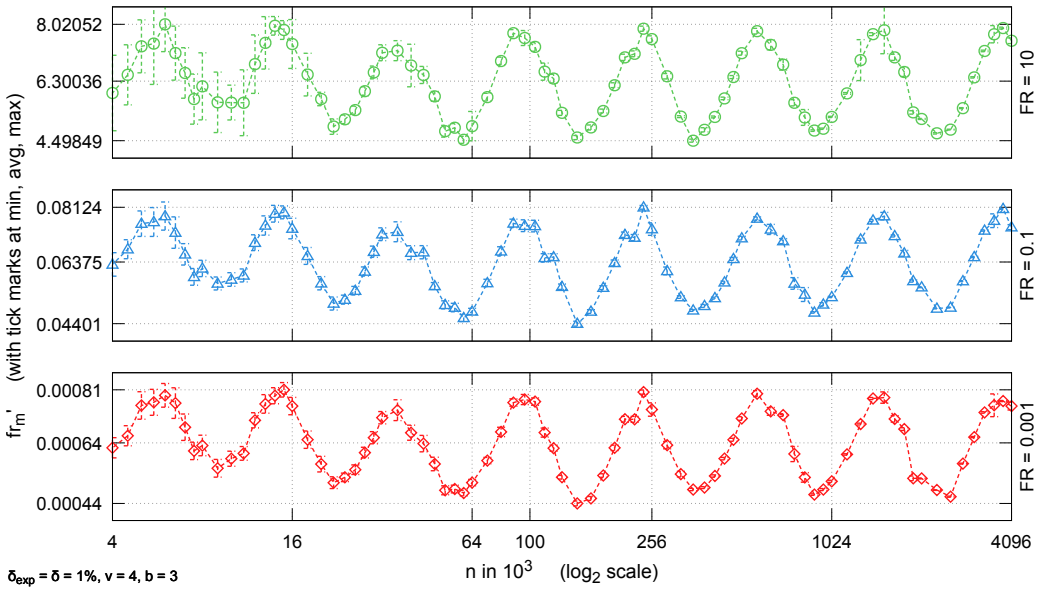


Figure A.8: Merkle $v = 4, b = 3$ effective worst-case failure rate fr'_m, *missing* scenario for different FR and n with a binomial data and failure distribution (error bars = standard deviation).

A.2.2 Evaluation

Parameter Space Exploration

Figures A.9 to A.11 show Merkle tree parameter sweeps of $v \in [2, 16]$ and $b \in [1, 16]$ with different δ and FR than Figure 8.11 on page 137. As shown, differences in FR do not seem to change much in neither the relative transfer costs nor the absolute numbers of message rounds. Different values of δ, however, do change both metrics but not the region of parameters optimal with regards to transfer costs, i.e. $v \in \{3, 4\}$ with $b = 2$ and $b = 1$ for the *outdated* and *missing* scenarios, respectively.

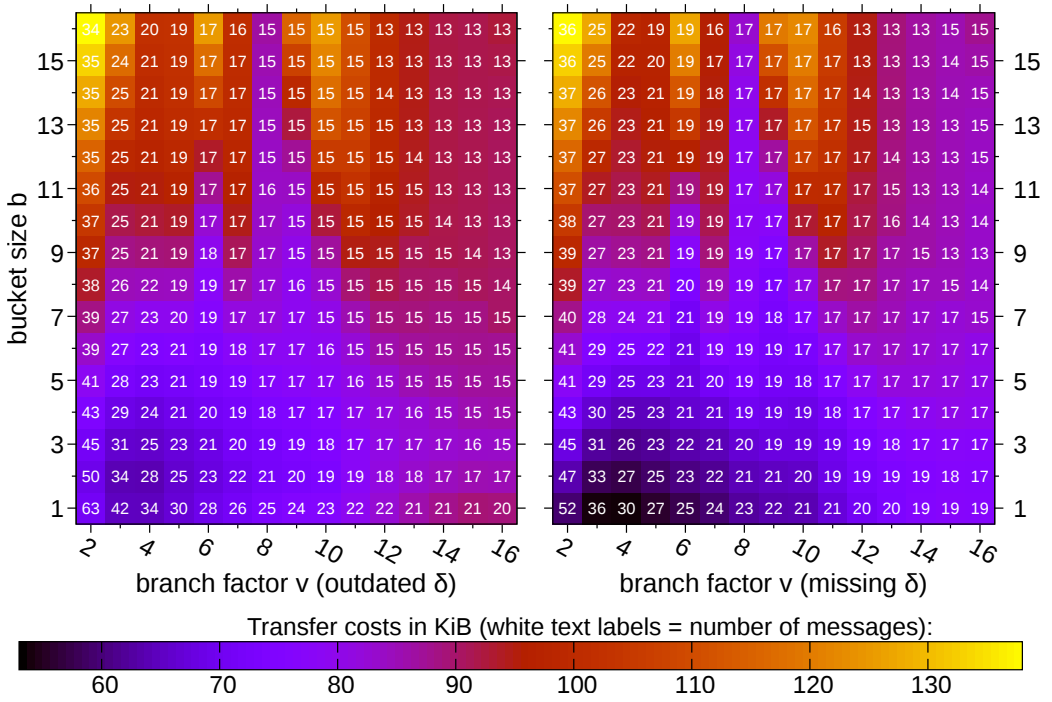


Figure A.9: Merkle reconciliation transfer costs (heatmap) and number of messages (in white) with $FR = 10$ and different v and b for $\delta_{exp} = \delta = 2\,\%$.

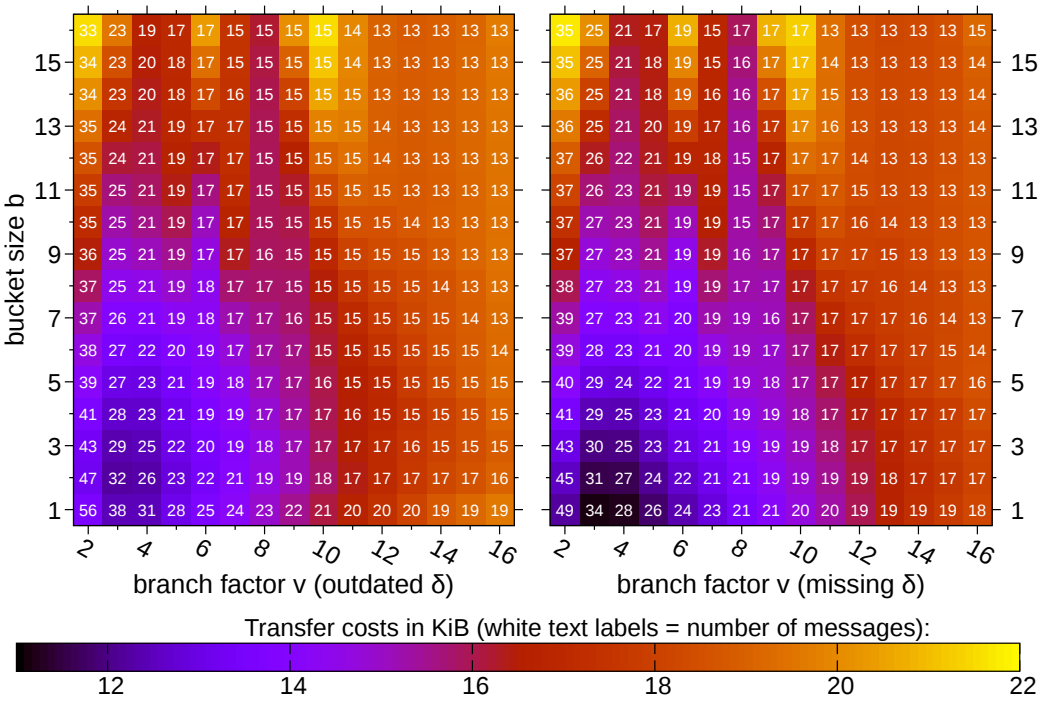


Figure A.10: Merkle reconciliation transfer costs (heatmap) and number of messages (in white) with $FR = 0.1$ and different v and b for $\delta_{exp} = \delta = 0.2\,\%$.

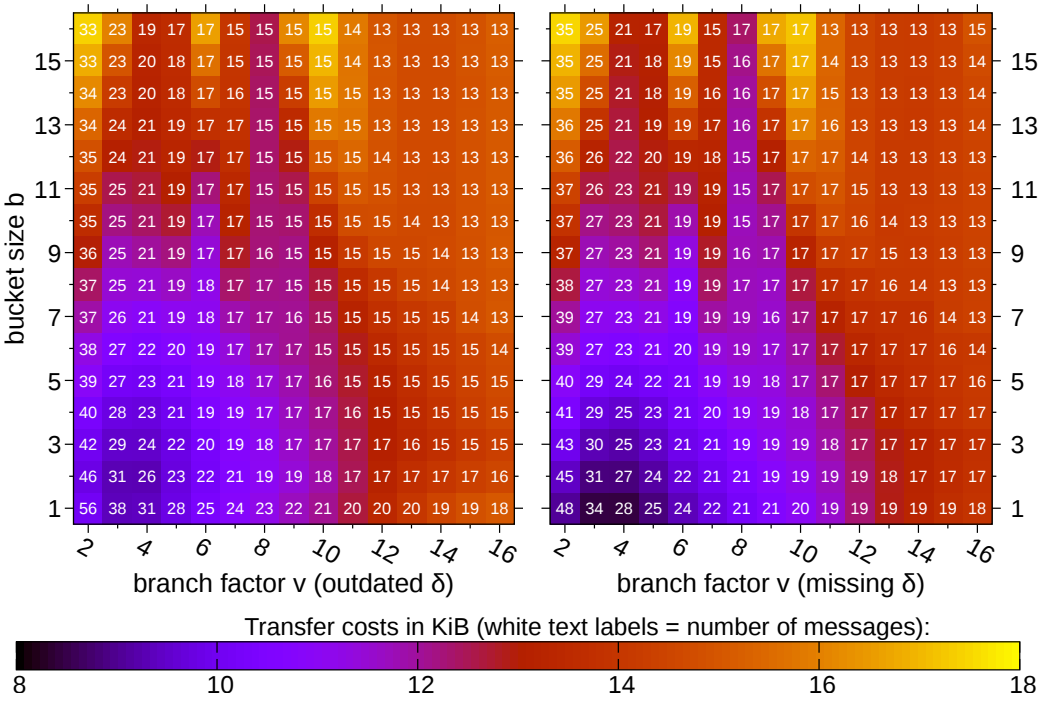


Figure A.11: Merkle reconciliation transfer costs (heatmap) and number of messages (in white) with $FR = 10$ and different v and b for $\delta_{exp} = \delta = 0.2\,\%$.

A.2.3 General Analysis for Different δ and FR

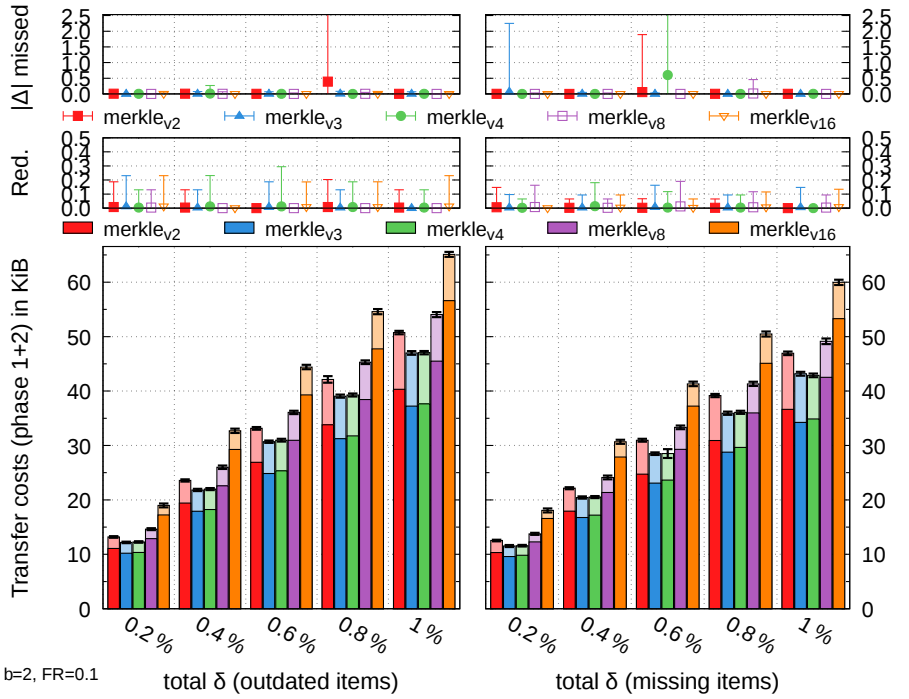


Figure A.12: Merkle reconciliation for small δ, fixed b but varying v.

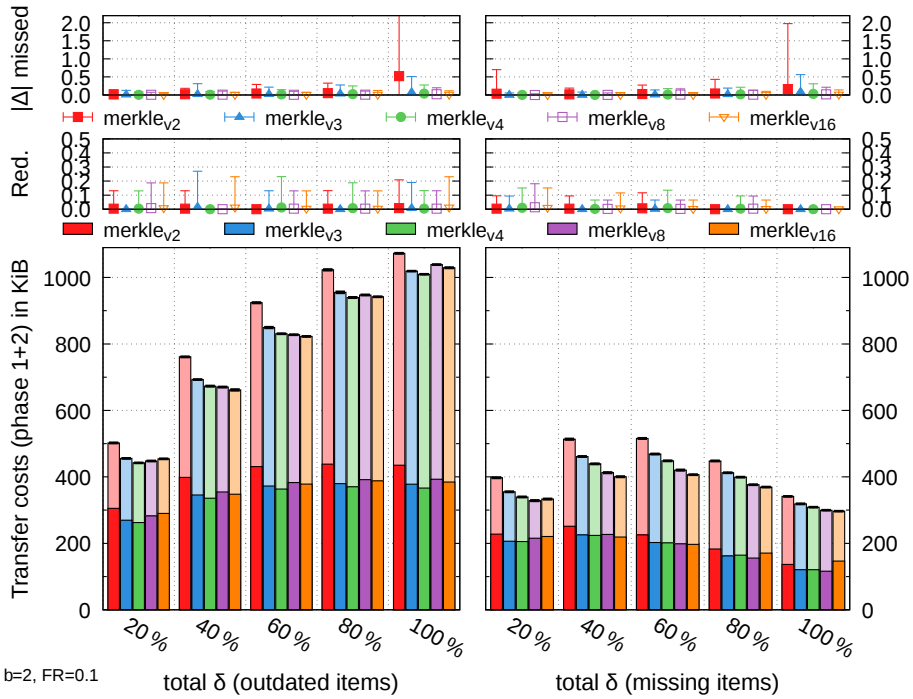


Figure A.13: Merkle reconciliation for high δ, fixed b but varying v.

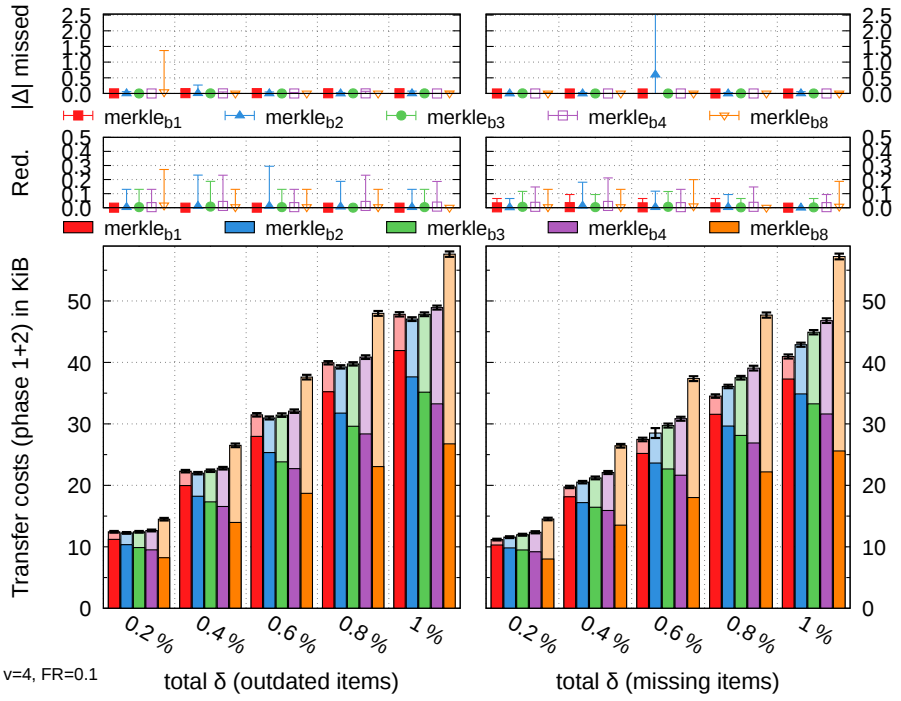


Figure A.14: Merkle reconciliation for small δ, fixed v but varying b.

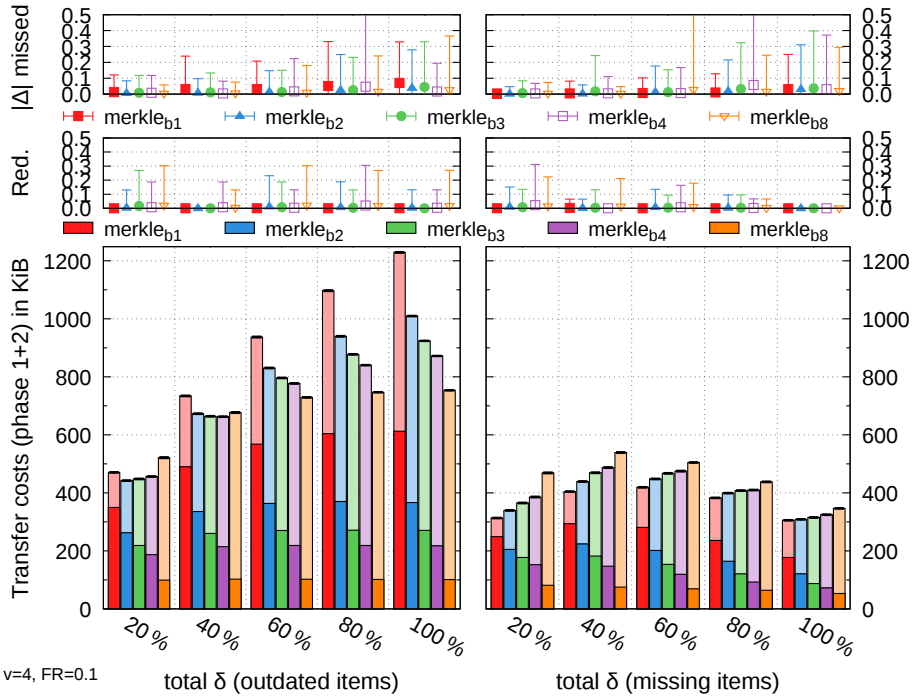


Figure A.15: Merkle reconciliation for high δ, fixed v but varying b.

A.2.4 What if δ_{exp} is Wrong?

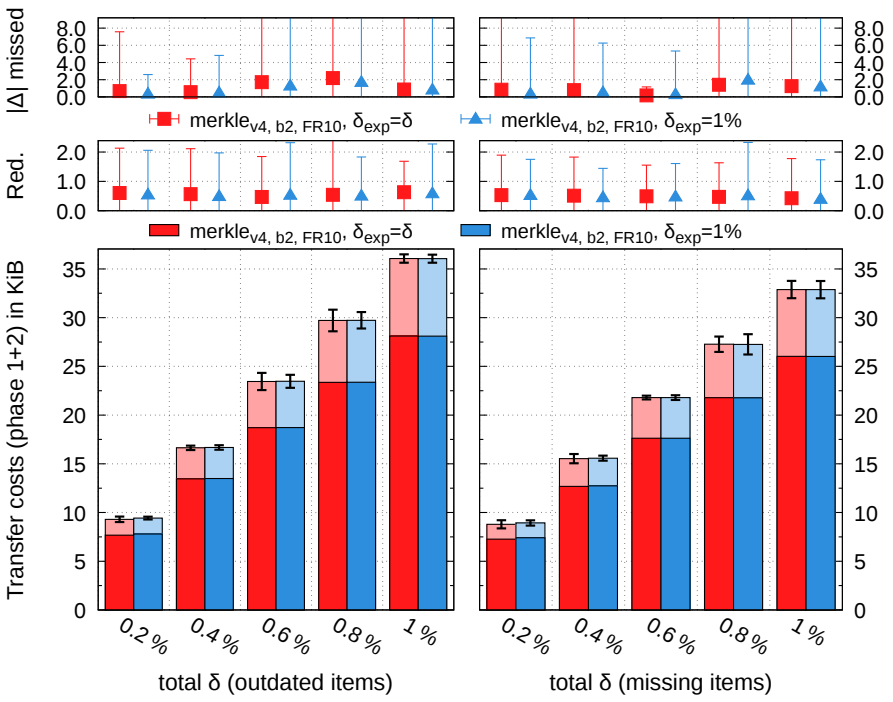


Figure A.16: Merkle reconciliation with very small δ and different δ_{exp}.

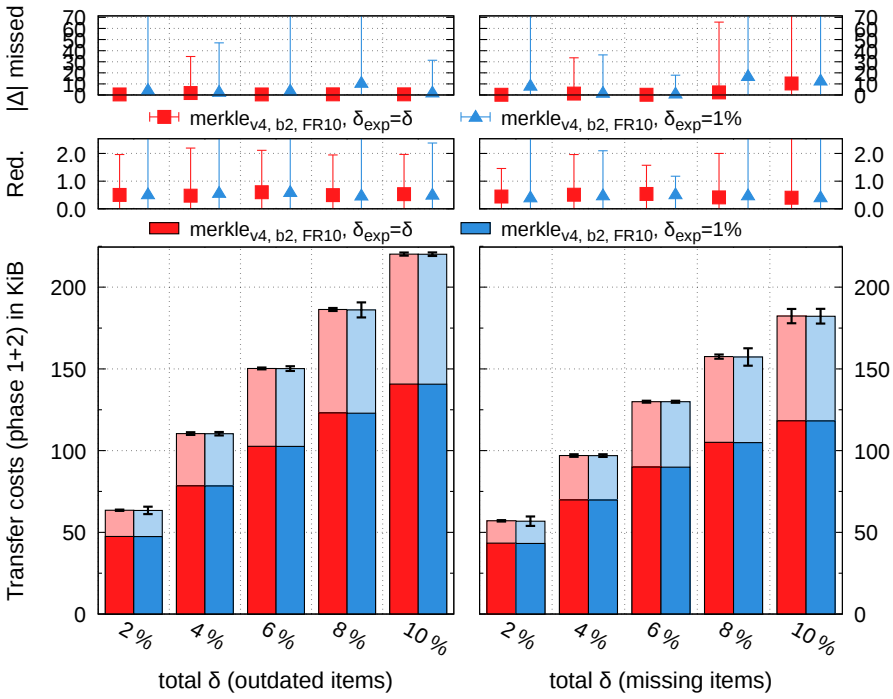


Figure A.17: Merkle reconciliation with small δ and different δ_{exp}.

A.3 Comparative Evaluation

A.3.1 General Analysis for Different δ and FR

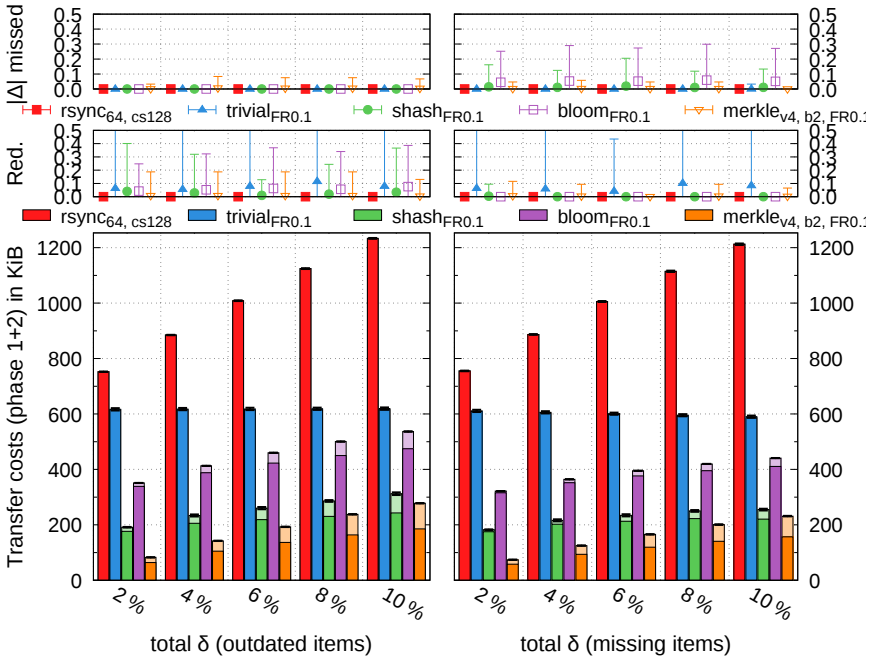


Figure A.18: Comparison for $FR = 0.1$ with small δ.

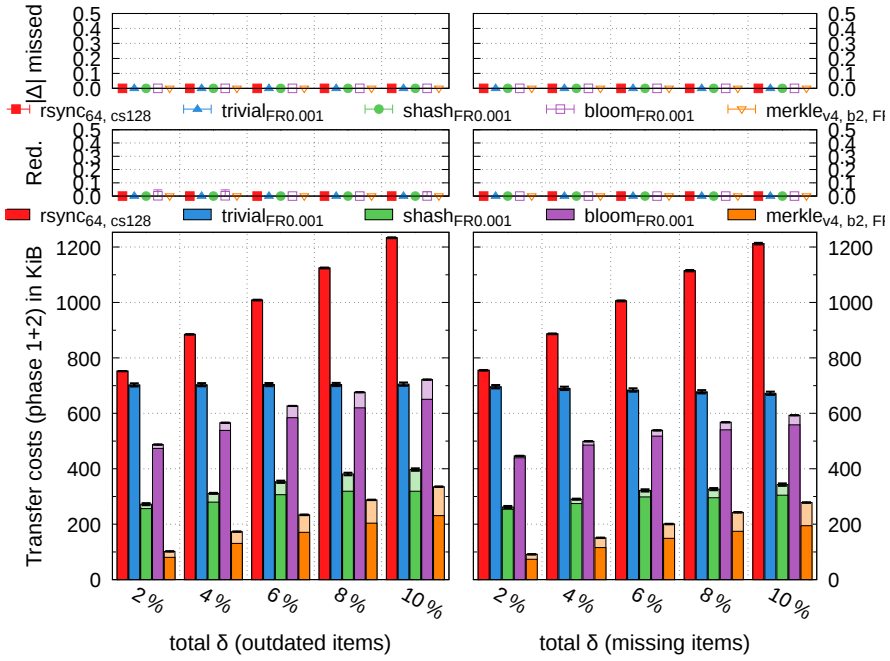


Figure A.19: Comparison for $FR = 0.001$ with small δ.

Bibliography

[1] K. A. S. Abdel-Ghaffar and A. El Abbadi. „An optimal strategy for comparing file copies“. In: *IEEE Transactions on Parallel and Distributed Systems* 5.1 (Jan. 1994), pp. 87–93. ISSN: 1045-9219. DOI: 10.1109/71.262591.

[2] Aftab Ali and Farrukh Aslam Khan. „A Broadcast-Based Key Agreement Scheme Using Set Reconciliation for Wireless Body Area Networks“. In: *Journal of Medical Systems* 38.5 (2014), pp. 1–12. ISSN: 1573-689X. DOI: 10.1007/s10916-014-0033-1.

[3] Burton H. Bloom. „Space/Time Trade-offs in Hash Coding with Allowable Errors“. In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692.

[4] Anudhyan Boral and Michael Mitzenmacher. „Multi-party set reconciliation using characteristic polynomials“. In: *2014 52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. Sept. 2014, pp. 1182–1187. DOI: 10.1109/ALLERTON.2014.7028589.

[5] Prosenjit Bose et al. „On the false-positive rate of Bloom filters“. In: *Information Processing Letters* 108.4 (2008), pp. 210–213. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2008.05.018.

[6] A. Z. Broder. „On the resemblance and containment of documents“. In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. June 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[7] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. „Min-Wise Independent Permutations“. In: *Journal of Computer and System Sciences* 60.3 (2000), pp. 630–659. ISSN: 0022-0000. DOI: 10.1006/jcss.1999.1690.

[8] Andrei Broder and Michael Mitzenmacher. „Network Applications of Bloom Filters: A Survey“. In: *Internet Math.* 1.4 (2003), pp. 485–509.

[9] Michiel Buddingh. *The distribution of hash function outputs*. URL: https://michiel.buddingh.eu/distribution-of-hash-values (visited on Dec. 8, 2018).

[10] John Byers, Jeffrey Considine, and Michael Mitzenmacher. *Fast Approximate Reconciliation of Set Differences*. Tech. rep. Boston University Computer Science Department, 2002.

[11] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. „Informed Content Delivery Across Adaptive Overlay Networks“. In: *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '02. Pittsburgh, Pennsylvania, USA: ACM, 2002, pp. 47–60. ISBN: 1-58113-570-X. DOI: 10.1145/633025.633031.

[12] Brad Calder et al. „Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency“. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 143–157. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043571.

[13] J.Lawrence Carter and Mark N. Wegman. „Universal classes of hash functions“. In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 143–154. ISSN: 0022-0000. DOI: 10.1016/0022-0000(79)90044-8.

[14] Josh Cates. „Robust and efficient data management for a distributed hash table“. MA thesis. Massachusetts Institute of Technology, May 2003.

[15] Francis Chang, Wu-chang Feng, and Kang Li. „Approximate caches for packet classification“. In: *23rd IEEE INFOCOM Proceedings*. Vol. 4. IEEE, Mar. 2004, pp. 2196–2207. DOI: 10.1109/INFCOM.2004.1354643.

[16] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. „The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables“. In: *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '04. New Orleans, Louisiana: SIAM, 2004, pp. 30–39. ISBN: 0-89871-558-X.

[17] Di Chen, Christian Konrad, Ke Yi, Wei Yu, and Qin Zhang. „Robust Set Reconciliation“. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 135–146. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2610528.

[18] Matt Corallo. *BIP 152: Compact Block Relay*. Apr. 2016. URL: https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki (visited on Sept. 28, 2018).

[19] Graham Cormode and S. Muthukrishnan. „What's Hot and What's Not: Tracking Most Frequent Items Dynamically“. In: *ACM Trans. Database Syst.* 30.1 (Mar. 2005), pp. 249–278. ISSN: 0362-5915. DOI: 10.1145/1061318.1061325.

[20] Graham Cormode and S. Muthukrishnan. „What's New: Finding Significant Differences in Network Data Streams“. In: *IEEE/ACM Trans. Netw.* 13.6 (Dec. 2005), pp. 1219–1232. ISSN: 1063-6692. DOI: 10.1109/TNET.2005.860096.

[21] Graham Cormode, S. Muthukrishnan, and Irina Rozenbaum. „Summarizing and Mining Inverse Distributions on Data Streams via Dynamic Inverse Sampling“. In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 25–36. ISBN: 1-59593-154-6.

[22] Giuseppe DeCandia et al. „Dynamo: Amazon's Highly Available Key-value Store“. In: *21st ACM SIGOPS Proceedings*. Vol. 41. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281.

[23] Peter C. Dillinger and Panagiotis Manolios. „Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings“. In: ed. by Alan J. Hu and Andrew K. Martin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. Chap. Bloom Filters in Probabilistic Verification, pp. 367–381. ISBN: 978-3-540-30494-4. DOI: 10.1007/978-3-540-30494-4_26.

[24] D. Eastlake 3rd and P. Jones. *US Secure Hash Algorithm 1 (SHA1)*. Tech. rep. RFC3174. United States, 2001.

[25] David Eppstein and Michael T. Goodrich. „Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters“. In: *IEEE Transactions on Knowledge and Data Engineering* 23.2 (Feb. 2011), pp. 297–306. ISSN: 1041-4347. DOI: 10.1109/TKDE.2010.132.

[26] David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. „What's the Difference? Efficient Set Reconciliation Without Prior Context“. In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: ACM, Aug. 2011, pp. 218–229. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018462.

[27] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. „Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol“. In: *IEEE/ACM Transactions on Networking* 8.3 (June 2000), pp. 281–293. ISSN: 1063-6692. DOI: 10.1109/90.851975.

[28] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. „An Approximate L^1-Difference Algorithm for Massive Data Streams“. In: *SIAM Journal on Computing* 32.1 (2002), pp. 131–151. DOI: 10.1137/S0097539799361701.

[29] Philippe Flajolet and G. Nigel Martin. „Probabilistic counting algorithms for data base applications“. In: *Journal of Computer and System Sciences* 31.2 (1985), pp. 182–209. ISSN: 0022-0000. DOI: 10.1016/0022-0000(85)90041-8.

[30] Sumit Ganguly and Anirban Majumder. „Deterministic K-set Structure“. In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '06. Chicago, IL, USA: ACM, 2006, pp. 280–289. ISBN: 1-59593-318-2. DOI: 10.1145/1142351.1142392.

[31] Ali Ghodsi, LucOnana Alima, and Seif Haridi. „Symmetric Replication for Structured Peer-to-Peer Systems“. In: *Databases, Information Systems, and Peer-to-Peer Computing*. Ed. by Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin, and ArisM. Ouksel. Vol. 4125. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 74–85. ISBN: 978-3-540-71660-0. DOI: 10.1007/978-3-540-71661-7_7.

[32] David Goldberg. „What Every Computer Scientist Should Know About Floating-point Arithmetic“. In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163.

[33] Richard Andrew Golding. „Weak-consistency Group Communication and Membership“. UMI Order No. GAX93-12435. PhD thesis. Santa Cruz, CA, USA, 1992.

[34] Michael T. Goodrich and Michael Mitzenmacher. „Invertible Bloom Lookup Tables“. In: *Communication, Control, and Computing, 2011 49th Annual Allerton Conference on*. Sept. 2011, pp. 792–799. DOI: 10.1109/Allerton.2011.6120248.

[35] Caidong Gu, Wei Gong, and Amiya Nayak. „Identifying Discrepant Tags in RFID-enabled Supply Chains“. In: *Wireless Algorithms, Systems, and Applications*. Ed. by Qing Yang, Wei Yu, and Yacine Challal. Cham: Springer International Publishing, 2016, pp. 162–173. ISBN: 978-3-319-42836-9. DOI: 10.1007/978-3-319-42836-9_15.

[36] Deke Guo and Mo Li. „Set Reconciliation via Counting Bloom Filters“. In: *IEEE Transactions on Knowledge and Data Engineering* 25.10 (Oct. 2013), pp. 2367–2380. ISSN: 1041-4347. DOI: 10.1109/TKDE.2012.215.

[37] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. „Resource Discovery in Distributed Networks“. In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '99. Atlanta, Georgia, USA: ACM, 1999, pp. 229–237. ISBN: 1-58113-099-6. DOI: 10.1145/301308.301362.

[38] Christian Henke, Carsten Schmoll, and Tanja Zseby. „Empirical Evaluation of Hash Functions for Multipoint Measurements“. In: *SIGCOMM Comput. Commun. Rev.* 38.3 (July 2008), pp. 39–50. ISSN: 0146-4833. DOI: 10.1145/1384609.1384614.

[39] A Horton and R Adams. „Standard for interchange of USENET messages“. In: *Network Working Group, IETF* (1987). RFC 1036.

[40] Zhiyao Hu et al. „Comparing set reconciliation methods based on bloom filters and their variants“. In: *Tsinghua Science and Technology* 21.2 (Apr. 2016), pp. 157–167. ISSN: 1007-0214. DOI: 10.1109/TST.2016.7442499.

[41] M. G. Karpovsky, L. B. Levitin, and A. Trachtenberg. „Data verification and reconciliation with generalized error-control codes“. In: *IEEE Transactions on Information Theory* 49.7 (July 2003), pp. 1788–1793. ISSN: 0018-9448. DOI: 10.1109/TIT.2003.813498.

[42] Adam Kirsch and Michael Mitzenmacher. „Less hashing, same performance: Building a better Bloom filter“. In: *Random Structures & Algorithms* 33.2 (2008), pp. 187–218. ISSN: 1098-2418. DOI: 10.1002/rsa.20208.

[43] Donald E. Knuth. „Sorting and searching.(The art of computer programming, vol. 3) Addison-Wesley“. In: *Reading, MA* (1973), pp. 551–575.

[44] Nico Kruber, Mikael Högqvist, and Thorsten Schütt. „The Benefits of Estimated Global Information in DHT Load Balancing“. In: *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 382–391. ISBN: 978-0-7695-4395-6. DOI: 10.1109/CCGrid.2011.11.

[45] Nico Kruber, Maik Lange, and Florian Schintke. „Approximate Hash-Based Set Reconciliation for Distributed Replica Repair“. In: *34th IEEE Symposium on Reliable Distributed Systems (SRDS)*. Sept. 2015, pp. 166–175. DOI: 10.1109/SRDS.2015.30.

[46] John Kubiatowicz et al. „OceanStore: An Architecture for Global-scale Persistent Storage“. In: *SIGPLAN Not.* 35.11 (Nov. 2000), pp. 190–201. ISSN: 0362-1340. DOI: 10.1145/356989.357007.

[47] Maik Lange. „Effiziente Reparatur von Repliken in Distributed Hash Tables“. German. MA thesis. Humboldt-Universität zu Berlin, Sept. 2012.

[48] Jin Li et al. „Using index partitioning and reconciliation for data deduplication“. US 9110936. Aug. 2015.

[49] Kaisen Lin and Philip Levis. „Data Discovery and Dissemination with DIP“. In: *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*. IPSN '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 433–444. ISBN: 978-0-7695-3157-1. DOI: 10.1109/IPSN.2008.17.

[50] Eric Lombrozo, Johnson Lau, and Pieter Wuille. *BIP 141: Segregated Witness (Consensus layer)*. Dec. 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki (visited on Sept. 28, 2018).

[51] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. „Optimizing Bloom Filter: Challenges, Solutions, and Comparisons“. In: *CoRR* abs/1804.04777 (2018). arXiv: 1804.04777.

[52] Ralph C. Merkle. „A Certified Digital Signature“. In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. Vol. 435. Lecture Notes in Computer Science. Springer New York, 1990, pp. 218–238. ISBN: 978-0-387-97317-3. DOI: 10.1007/0-387-34805-0_21.

[53] Yaron Minsky and Ari Trachtenberg. *Efficient Reconciliation of Unordered Databases*. Tech. rep. Ithaca, NY, USA, 1999.

[54] Yaron Minsky and Ari Trachtenberg. *Practical Set Reconciliation*. Tech. rep. BU-ECE-2002-01. Boston University, Department of Electrical and Computer Engineering, Feb. 2002.

[55] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. „Set reconciliation with nearly optimal communication complexity“. In: *Information Theory, IEEE Transactions on* 49.9 (Sept. 2003), pp. 2213–2218. ISSN: 0018-9448. DOI: 10.1109/TIT.2003.815784.

[56] Michael Mitzenmacher. „Compressed Bloom Filters“. In: *IEEE/ACM Transactions on Networking* 10.5 (Oct. 2002), pp. 604–612. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.803864.

[57] Michael Mitzenmacher and Tom Morgan. „Reconciling Graphs and Sets of Sets“. In: *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18. Houston, TX, USA: ACM, 2018, pp. 33–47. ISBN: 978-1-4503-4706-8. DOI: 10.1145/3196959.3196988.

[58] Michael Mitzenmacher and Rasmus Pagh. „Simple Multi-Party Set Reconciliation“. In: *CoRR* abs/1311.2037 (2013).

[59] Michael Mitzenmacher and George Varghese. „Biff (Bloom filter) codes: Fast error correction for large data sets“. In: *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. July 2012, pp. 483–487. DOI: 10.1109/ISIT.2012.6284236.

[60] Michael Mitzenmacher and George Varghese. „The complexity of object reconciliation, and open problems related to set difference and coding“. In: *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*. Oct. 2012, pp. 1126–1132. DOI: 10.1109/Allerton.2012.6483345.

[61] James K. Mullin. „A Second Look at Bloom Filters“. In: *Commun. ACM* 26.8 (Aug. 1983), pp. 570–571. ISSN: 0001-0782. DOI: 10.1145/358161.358167.

[62] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Mar. 2009. URL: https://bitcoin.org/bitcoin.pdf (visited on Sept. 21, 2018).

[63] Nishant Neeraj. *Mastering Apache Cassandra*. Second. Community experience distilled. Packt Publishing Ltd., Mar. 2015. ISBN: 9781784396251.

[64] A. Orlitsky. „Communication Issues in Distributed Computing“. UMI Order No. GAX87-07717. PhD thesis. Stanford, CA, USA, 1987.

[65] A. Pinar Ozisik, Gavin Andresen, George Bissias, Amir Houmansadr, and Brian N. Levine. *A Secure, Efficient, and Transparent Network Architecture for Bitcoin*. Tech. rep. Amherst: University of Massachusetts, 2016.

[66] A. Pinar Ozisik, Gavin Andresen, George Bissias, Amir Houmansadr, and Brian N. Levine. „Graphene: A New Protocol for Block Propagation Using Set Reconciliation“. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí. Cham: Springer International Publishing, 2017, pp. 420–428. ISBN: 978-3-319-67816-0. DOI: 10.1007/978-3-319-67816-0_24.

[67] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. „Cardinality estimation and dynamic length adaptation for Bloom filters“. In: *Distributed and Parallel Databases* 28.2 (Dec. 2010), pp. 119–156. ISSN: 1573-7578. DOI: 10.1007/s10619-010-7067-2.

[68] J. A. Pouwelse et al. „TRIBLER: a social-based peer-to-peer system“. In: *Concurrency and Computation: Practice and Experience* 20.2 (2008), pp. 127–138. ISSN: 1532-0634. DOI: 10.1002/cpe.1189.

[69] Robbert van Renesse, Yaron Minsky, and Mark Hayden. „A Gossip-Style Failure Detection Service“. In: *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Ed. by Nigel Davies, Seitz Jochen, and Kerry Raymond. London: Springer London, 1998, pp. 55–70. ISBN: 978-1-4471-1283-9. DOI: 10.1007/978-1-4471-1283-9_4.

[70] Patrick Reynolds and Amin Vahdat. „Efficient Peer-to-peer Keyword Searching“. In: *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Middleware '03. Rio de Janeiro, Brazil: Springer-Verlag New York, Inc., 2003, pp. 21–40. ISBN: 3-540-40317-5.

[71] Sean Rhea, Russ Cox, and Alex Pesterev. „Fast, Inexpensive Content-addressed Storage in Foundation“. In: *USENIX 2008 Annual Technical Conference*. ATC'08. Boston, Massachusetts: USENIX Association, 2008, pp. 143–156.

[72] Ronald Linn Rivest. *The MD5 Message-Digest Algorithm*. Tech. rep. RFC1321. United States, 1992.

[73] *Scalaris, a distributed, transactional key-value store*. URL: https://github.com/scalaris-team/scalaris (visited on Dec. 9, 2018).

[74] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. „Scalaris: Reliable Transactional P2P Key/Value Store“. In: *ACM SIGPLAN Erlang Workshop*. ERLANG '08. Victoria, BC, Canada: ACM, Sept. 2008, pp. 41–48. ISBN: 978-1-60558-065-4. DOI: 10.1145/1411273.1411280.

[75] Robert Schweller et al. „Reversible Sketches: Enabling Monitoring and Analysis over High-speed Data Streams“. In: *IEEE/ACM Trans. Netw.* 15.5 (Oct. 2007), pp. 1059–1072. ISSN: 1063-6692. DOI: 10.1109/TNET.2007.896150.

[76] *Secure Hash Standard (SHS)*. Federal Information Processing Standard 180-4. Washington: National Institute of Standards and Technology, 2015. DOI: 10.6028/NIST.FIPS.180-4.

[77] *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Federal Information Processing Standard 202. Washington: National Institute of Standards and Technology, 2015. DOI: 10.6028/NIST.FIPS.202.

[78] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. „Theory and Practice of Bloom Filters for Distributed Systems“. In: *Communications Surveys & Tutorials, IEEE* 14.1 (2012), pp. 131–155. ISSN: 1553-877X. DOI: 10.1109/SURV.2011.031611.00024.

[79] Flemming Topsøe. „Some Bounds for the Logarithmic Function“. In: *Inequality Theory and Applications* 4 (2007). Ed. by Yeol Je Cho, Jong Kyu Kim, and Sever S. Dragomir, pp. 137–151.

[80] A. Trachtenberg, D. Starobinski, and S. Agarwal. „Fast PDA synchronization using characteristic polynomial interpolation“. In: *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. June 2002, 1510–1519 vol.3. DOI: 10.1109/INFCOM.2002.1019402.

[81] Andrew Tridgell. „Efficient algorithms for sorting and synchronization“. PhD thesis. 1999.

[82] Andrew Tridgell and Paul Mackerras. *The rsync algorithm*. Tech. rep. TR-CS-96-05. Canberra 0200 ACT, Australia: Department of Computer Science, The Australian National University, June 1996.

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

[84] Peter Tschipper. *BUIP010: Xtreme Thinblocks*. Jan. 2016. URL: https://bitco.in/forum/threads/buip010-passed-xtreme-thinblocks.774/ (visited on Sept. 21, 2018).

[85] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. „CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays“. In: *Journal of Network and Systems Management* 13.2 (June 2005), pp. 197–217. ISSN: 1064-7570 (Print) 1573-7705 (Online). DOI: 10.1007/s10922-005-4441-x.

[86] M. N. Wegman and J. L. Carter. „New classes and applications of hash functions“. In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. Oct. 1979, pp. 175–182. DOI: 10.1109/SFCS.1979.26.

[87] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. Byzantium Version 94ebda - 2018-06-05. June 2018. URL: https://github.com/ethereum/yellowpaper (visited on Sept. 21, 2018).

[88] Bo Yu and Fan Bai. „PYRAMID: Informed content reconciliation for vehicular peer-to-peer systems“. In: *Vehicular Networking Conference (VNC), 2015 IEEE*. Dec. 2015, pp. 212–219. DOI: 10.1109/VNC.2015.7385579.

[89] Benjamin Zhu, Kai Li, and Hugo Patterson. „Avoiding the Disk Bottleneck in the Data Domain Deduplication File System“. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. FAST'08. San Jose, California: USENIX Association, 2008, 18:1–18:14.

Publications of the Author

Mikael Högqvist, Seif Haridi, Nico Kruber, Alexander Reinefeld, and Thorsten Schütt. „Using Global Information for Load Balancing in DHTs“. In: *Workshop on Decentralized Self Management for Grids, P2P, and User Communities*. Vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2008, pp. 236–241. ISBN: 978-0-7695-3553-1. DOI: 10.1109/SASOW.2008.50.

Mikael Högqvist and Nico Kruber. „Passive/Active Load Balancing with Informed Node Placement in DHTs“. In: *Self-Organizing Systems: 4th IFIP TC 6 International Workshop (IWSOS 2009). Proceedings*. Ed. by Thrasyvoulos Spyropoulos and Karin Anna Hummel. Vol. 5918. Zurich, Switzerland: Springer Berlin Heidelberg, Dec. 2009, pp. 101–112. ISBN: 978-3-642-10865-5. DOI: 10.1007/978-3-642-10865-5_9.

Nico Kruber. „DHT Load Balancing with Estimated Global Information“. Diploma Thesis. Humboldt-Universität zu Berlin, Sept. 2009. URN: urn:nbn:de:0297-zib-11514.

G. Birkenheuer et al. „MoSGrid – a molecular simulation grid as a new tool in computational chemistry, biology and material science“. In: *Journal of Cheminformatics* 3.1 (2011). Free Software Session and poster, pp. 1–1. ISSN: 1758-2946. DOI: 10.1186/1758-2946-3-S1-P14.

Nico Kruber, Mikael Högqvist, and Thorsten Schütt. „The Benefits of Estimated Global Information in DHT Load Balancing“. In: *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 382–391. ISBN: 978-0-7695-4395-6. DOI: 10.1109/CCGrid.2011.11.

Nico Kruber, Florian Schintke, and Michael Berlin. „A relational database schema on the transactional key-value store scalaris“. In: *Proceedings of 2nd Workshop on Scalable Cloud Data Management*. Oct. 2014, pp. 29–37. DOI: 10.1109/BigData.2014.7004441.

Nico Kruber, Maik Lange, and Florian Schintke. „Approximate Hash-Based Set Reconciliation for Distributed Replica Repair“. In: *34th IEEE Symposium on Reliable Distributed Systems (SRDS)*. Sept. 2015, pp. 166–175. DOI: 10.1109/SRDS.2015.30.

Scientific Talks

26 May 2011 The Benefits of Estimated Global Information in DHT Load Balancing. 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. Newport Beach, CA, USA.

05 Jun 2012 Scalaris: Scalable Web Applications with a Transactional Key-Value Store. Berlin Buzzwords 2012. Berlin, Germany

20 Nov 2012 Scalable Data Models with the Transactional Key-Value Store Scalaris. INGI Fall 2012 Doctoral School Day in Cloud Computing. Louvain-la-Neuve, Belgium.

27 Oct 2014 A Relational Database Schema on the Transactional Key-Value Store Scalaris. 2nd Workshop on Scalable Cloud Data Management. Washington DC, USA.

30 Sept 2015 Approximate Hash-Based Set Reconciliation for Distributed Replica Repair. 34th International Symposium on Reliable Distributed Systems. Montreal, Canada.