Byzantine Fault Tolerant Set Reconciliation

Bachelor Thesis

Degree programme:	BSc in Computer science
Author:	Elias Summermatter
Thesis advisor:	Prof. Dr. Christian Grothoff
Expert:	Han van der Kleij
Date:	June, 17th 2021

Abstract

The goals of the thesis are to improve the existing implementation of the Byzantine Fault Tolerant Set Reconciliation in the GNUnet project regarding performance and security and to create an RFC documenting the protocol to publish as a draft to the IETF Datatracker. The improvements made in the protocol are validated by logical reasoning and measurement. The existing implementation in GNUnet is used for revocation in the GNU Name System, but could be used in the future for decentralised e-voting systems to achieve consensus on the ballots between voting authorities.

The goal of reconciliation protocols is to find an efficient way in a peer-to-peer network to compute the union of two sets of elements over a network. Since the sets can have large overlaps and small differences, it could be inefficient to transfer all elements of one peer to the other and then compute the union on one site. Eppstein demonstrated a better way to reconcile sets with small differences using advanced probabilistic data types. Dold provided an original implementation of Eppstein's ideas.
The enhancement made to the protocol for this thesis have improved the performance of matching sets with small differences by 30 to 42 percent. Also, several serious implementation errors that made the previous protocol unstable, e.g., for larger sets, have been fixed. Furthermore, through the exact analysis and documentation of the individual protocol steps and messages, various security improvements in the protocol and also basic sanity checks in the code could be implemented. A complete binary level specification in form of an RFC is published on the IETF Datatracker.

Acknowledgements

I would like to thank all those who supported and motivated me during the preparation of this bachelor thesis.

First of all, I would like to thank Prof. Christian Grothoff, who supervised my bachelor thesis. I would like to thank him for his helpful suggestions and constructive criticism during the preparation of this thesis.

For evaluating and dealing with the thesis I would like to thank Mr. Han van der Kleij as the examining expert.

I thank BFH for providing me with access to computing resources needed for the simulations taking hundreds of thousands of CPU hours.

I would also like to thank Mr. Michael Scheurer who supported me in my company throughout my studies.

Finally, I would like to thank Maria Summermatter, my mother, who made my studies possible with her support and also assisted with proofreading.

Table of Contents

Contents

1 Introduction

A way is sought to reconcile the differences between two sets over a network as simply and resource-efficiently as possible. This is important for sensitive applications, such as e-voting, when the votes are reconciled in a decentralised system between the electoral authorities. With an efficient set reconciliation protocol, it is possible to achieve consensus on the ballots the votes efficiently[1, 2]. Especially in this field, precision and security are extremely important.

An efficient way to reconcile two sets of elements over a network was first described by Eppstein[3]. This work is primarily concerned with matching small differences in very large sets, because with large set differences it is usually more resource-efficient to transfer the entire sets instead of eliciting the difference between the two sets and only transferring the resulting difference.

A first implementation was developed and implemented by Dold and Grothoff ([4]: src/setu in 2016) . The implementation has not yet been specified nor has it been tested more closely for performance and security. Currently, the implementation is mainly used in the GNU Name System (GNS)[5] to synchronise revocations across all peers from the P2P network.

This thesis addresses security and performance limitations of the current implementation. These improvements are quantified by measurements or logical reasoning. A purely mathematical analysis of the problem would have been prohibitively complex, as numerous variables influence the optimal performance. Thus, a combination of logical reasoning backed by experimental measurements was performed. A server with large computing power was available, which significantly reduced the time of the simulations. Logical reasoning was chosen for the analysis and improvement of the security aspects.

In addition, an RFC-like binary level specification of the optimised implementation was written. With this specification, the described protocol for set reconciliation could be ported to other programming languages and thus become more widely available.

This work is divided into two parts. In the RFC the implementation, specification and the background to the topic are described. The RFC is in the Appendix B - RFC. Parts and details that are not important for an implementation are described directly in the main document.

Privacy and security on the Internet are of concern to me. This work helps establish GNS, make it more secure and stable, which could lead to increased use of GNS, replacing the insecure name resolution protocols used today. Furthermore, I am contributing to the development of Free Software.

2 Related Work

This paper builds on the work of Florian Dold and Christian Grothoff (2017) entitled "Byzantine set-union consensus using efficient set reconciliation"[1]. The set union protocol developed by Dold is extended and improved by this work. Security and performance aspects were looked at on the protocol level, while the work of Dold and Grothoff based the experimental measurements mainly on the overlying layers of GNUnet. Many measurements in the work were made directly by CADET [6].

The theoretical basis for determining the set difference comes from a paper by Prof. David Eppstein. "What's the difference? Efficient Set Reconciliation without Prior Context". [3].

In his work, Eppstein showed a way for two peers to synchronise their sets in a single round with communication overhead proportional to the set size difference. The method shown by Eppstein does not require prior knowledge such as logs of the last synchronisation. In his paper, Eppstein tried to answer two questions that are closely related to this thesis:

- "What are the optimal parameters for an IBF?" (Section 6.1 of Eppstein's paper): Eppstein found that the relevant parameters for tuning an IBF are the number of buckets in the IBF and the number of times an element is mapped to the IBF. These two values are critical in determining how often the decoding of an IBF fails. He has found that the number of times an element is mapped to the IBF in his experiments is optimal at the value 4.

- "How should one tune the Strata Estimator to balance accuracy and overhead?"(Section 6.2 of Eppstein's paper): The paper argues, that if the Strata Estimator's estimate is too low, there is a risk of creating an IBF that is too small, and this brings a risk of having to transfer a new, larger IBF. This leads to a lot more bandwidth being used, so care should be taken that the set estimate needed to determine the IBF is rather set too high than too low. Eppstein compares in his work the "min-wise" independent permutations locality sensitive hashing scheme"[7] (min-wise) with Strata Estimators. He comes to the conclusion that the min-wise estimator is very susceptible to variances and therefore it can come to large inaccuracies with small set differences.With large set differences the min-wise estimator is superior to the Strata Estimators. In his work he combined the min-wise estimator with the Strata Estimator and achieved a new hybrid estimator that combines both advantages in one. However, this is not interesting for this work, since differential synchronisation is used for which a precise estimation is important only for small set differences.

The protocol proposed by Eppstein uses Invertible Bloom Filters (IBFs) and Strata Estimators (SE) to perform efficient differential synchronisation without prior knowledge. An Invertible Bloom Filter is a propabilistic data type that supports three operations: Add elements, Remove elements and determine the element difference between two Invertible Bloom Filters by an XOR operation. A Strata Esimator is a construct consisting of several Invertible Bloom Filters, a Strata Estimator allows to estimate the set difference. The protocol proposed by Eppstein is very simple and uses intially a Strata Estimator to calculate the set difference and then creates an IBF of the optimal size based on this calculated set difference. In this way, the elements that are missing from one peer or the other can be identified and exchanged.

Eppsteins work is the basis for the implementation and the basis for the improvements achieved by this thesis.

There are a few other studies that address the issue. One recent study is created by Michael Mitzenmacher and Tom Morgan, titled "Robust Set Reconciliation via Locality Sensitive Hashing" from 2018[8].

However, Mitzenmacher's paper deals with points in a metric space that are close together. To achieve this, well-known set reconciliation techniques were combined with locality-sensitive hashing[8]. This thesis, in contrast to Mitzenmacher's work, does not deal directly with points in metric space. Another key difference is that Mitzenmacher's work is concerned with finding points that are close together, but in this thesis the goal is to synchronise the sets precisely. The work of Mitzenmacher combines for the first time IBF with Locality-sensitive hashing. This work is interesting but cannot contribute much to our work.

Michael Mitzenmacher published a secound paper together with Rasmus Pagh under the title "Simple Multi-Party Set Reconciliation", which was published in 2015 [9]. The second work by Mitzenmacher, on the other hand, deals with the synchronisation of sets across multiple peers. In Mitzenmacher's work, emphasis was placed on the fact that set reconciliation can take place efficiently not only between two peers but between many peers. The developed technique should also work efficiently if it is not known exactly how many peers participate in the set reconciliation. For this purpose, network coding techniques in conjunction with Invertible Bloom Lookup Tables (IBLT)[10, 11] were used, which enabled a more efficient network usage. The reason why Mitzenmacher's proposed protocol cannot be used for implementation on the GNUnet is that it relies on a central place, a relay, to process the IBLT. GNUnet would have to operate a relay, but this needs to be avoided in a decentralised peer to peer setting.

3 Code Improvements

This section describes improvements that are neither directly related to the performance nor to the security of the protocol.

3.1 Determining the Optimal IBF Size after Failed Decoding

The original implementation stipulated that after failed decoding the size of the IBF was doubled. This is not optimal because the size of the IBF is always doubled. If in the extreme case with an initial set difference of 10'000 elements all but two elements could be decoded, the size of the IBF is still doubled, although the remaining set difference contains only 2 elements. It is therefore desirable that the new size of the IBF should also take into account how many elements were successfully decoded.

The heuristic that determines how big the IBF is after a failed decoding has been adjusted as follows:

Algorithm 1 Determinate next IBF size

```
Inputs:
i_factor: The factor by which the difference has
been multiplied to get the ibf bucket number
decoded_elements: Number of elements that have been
successfully decoded
last_ibf_size: The number of buckets of the last
IBF
Output:
next_size: Size of the next IBF
FUNCTION get_next_ibf_size(i_factor, d_elements, l_ibf_size)


    next_size =(l_ibf_size * 2) - ( i_factor * d_elements )
    RETURN MAX(37, next_size)

END FUNCTION
```

The new heuristic always doubles the size of the IBF, but also takes into account the decoded elements by subtracting the successfully decoded elements from the new doubled IBF size.

The theoretical benefit can be determined mathematically by a simple calculation.

If there is initially a set difference of 10'000 elements and therefore an IBF with 20'000 buckets and the decoding fails after 9500 elements, a new IBF of size 20’000 would be sent according to the old implementation. If one calculates 13 bytes per bucket (packet header is neglected) this is 260kb (13*20k) of bandwidth. With the new heuristic there would be 1000 buckets (10’000*2 - 2 * 9500) which is 13kb (13*1k). This means a potential bandwidth saving of factor 20 on the new attempt. Adding the 130kb (10k*13) initial transfer, there are 390kb with simple doubling and 143kb with the new heuristic. Increasing the IBF with the new heuristic leads to a saving factor of 2.5. Naturally, this simple analysis neglects that a smaller IBF may fail to decode a second time, causing additional round trips.

Thus, in order to measure the effective difference in practice, experiments were carried out to substantitate the previous logical plausibility argument. The aim was to find out whether the new heuristic produces better results, and to quantify the improvement. To determine whether multiplying the size of the last IBF by a factor of 2 is the optimal value, the heuristic was also generated with a factor of 1.5 and 2.5.

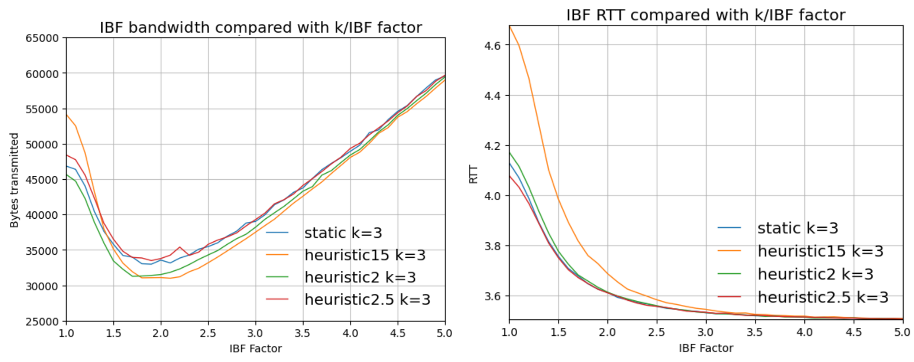


Figure 1: Performance improvements by new introduced IBF size heuristic

On the graphs, clearly can be seen that the heuristic is superior to the previous implementation in terms of the number of bytes transferred. With the heuristic that produces a doubling of the not decoded elements, one sees with an IBF factor of 2 an improvement compared with the static variant of 34k $\rightarrow$ 32k = 7%. The improvement is much smaller than in the calculation example above. However, it must be taken into account that with an IBF factor of 2, a decoding error occurs only in 20% of the cases and this heuristic can bring an improvement only in these cases. The effect on transmissions that effectively fail is likely to be up to 5 times higher, i.e., about 35%. This 35% bandwidth saving compared to the old implementation is significant.

The second graph shows the average number of RTTs required for the corresponding IBF factors. Here it can be clearly seen that the number of RTTs required for the smallest factor measured (1.5) is disproportionately high compared to the other larger factors. As expected, larger factors also produce a better result. This is not significant and shows no differences above a base IBF factor of 1.5.

3.2 Variable IBF Counter Size

Another uncovered problem of the original implementation was, that the counter of the IBF was always a signed 1-byte number. This 1-byte number allowed a maximum counter of 126 (127 means, that the counter has overflowed, see Appendix B - RFC.

This means that if a single bucket is hit more than 126 times, the IBF cannot be decoded anymore. In the "worst case" this means that already after 127 hits an IBF cannot be decoded successfully. This is the case, if all elements are mapped to the same bucket. The best case is when each bucket is hit the same number of times; then the maximum number of elements is calculated with the following formula:

$$elements_{max} = \frac{126 \times NumberOfElements}{NumberOfBucketsAnElementIsMapped} \tag{1}$$

Figure 2: Formula to calculate the max elements in an IBF bucket for the old implementation

This means that in an IBF with 32 buckets (original minimum number of buckets in an IBF), at least one counter overflows between 126 and 4032 (126 * 32) hits. This is only true if it is assumed that an element is mapped into only one bucket. In the new implementation, an element is not mapped into one but three buckets, in the old implementation even to four buckets. This means that these calculated numbers (126 and 4032) must be divided by three and four respectively. As long as the set from which the IBF was created is sufficiently small, this might not be a problem.

Since the number of IBF buckets in the existing implementation is only dependent on the set difference (IBF factor), but not on the size of the sets, it would quickly become a problem with small set differences but large sets. With a set difference of <16 (with IBF factor of 2, 2*16=32) elements and a k-value (number of buckets to which a bucket is mapped) of 3, it would lead to a set size of 127 elements at the earliest and after 345 (4032/3 + 1) elements at the latest, that the first counter overflows and the IBF can no longer decode.

For this problem, there are two possible solutions. The first possibility is to make the minimum size of the IBF dependent on the absolute set size in addition to the dependence on the set difference. This would cause the size of the IBF to grow linearly with the absolute set size. Calculating the minimum size would be complicated, as one would need to find a formula to calculate the probability of 127 elements mapping to the same bucket given an equally distributed mapping function. This problem could be seen as a modified form of the Birthday Paradox. [1] to be described, finding such a formula or deriving it would be difficult.

The second and better possibility is to select the counter size for the transmission variable when encoding the message for transmission over the network on the basis of the largest counter in the IBF, so that the counter for the transmission becomes maximally compact. This has the advantage that one can calculate internally with an 8-byte unsigned integer, but the bytes for transmission do not grow linearly, but logarithmically to the absolute set size and one can guarantee that it never (or at least not before 18446744073709551615 bucket hits) comes to a decoding failure because of the overflow of a counter.

The exact implementation can be found in the Git[4] of the GNUnet project as C code or in the RFC definition as pseudocode.

3.3 Statemachine

In order to make it easier to understand how the implementation of the Set Reconciliation Protocol works and to make the protocol more comprehensible for third parties, a statemachine diagram was created in this work that depicts the protocol. It was documented which messages can be received and sent in which phases and which messages can lead to which phase changes.

The initializing peer starts in the "Initializing Connection" phase and the peer expecting the connection starts in the "Expecting Connection" phase. In the first step after the connection is established, the receiving peer transmits the Strata Estimator to the initializing peer, so that it can estimate the set differences. As soon as the estimator has made the estimation, the peer decides whether to use the full synchronisation mode of operation or the differential mode of operation. If it chooses the full synchronisation mode of operation, The peer decides whether to transmit his set first or to request the set from the other peer. In the full mode of operation first one peer transmits the set to the other peer and then the other peer responds with the elements that are missing in the set of the first peer. If the sets are identical the full synchronisation is finished. If the initializing peer decides to use the differential mode of operation, the initializingpeer transmits an IBF to the receiving peer and becomes the passive decoding peer. As soon as the receiving peer receives the IBF, it starts decoding the elements in the IBF and becomes an active decoding peer. The active peer now requests all elements that are missing in his set and sends to the passive peer all elements that are missing in his set. It is now possible that the decoding of the IBF fails. In this case there is a role change when the active peer sends an IBF to the passive peer. By sending the IBF a role change occurs, the active decoding peer becomes the passive decoding peer and the passive decoding peer becomes the active decoding peer. This active/passive switch is repeated until the sets are identical and the operation is completed.

A detailed walkthrough and explanation in detail can be found in the Appendix B - RFC in the section “Mode of Operation”.

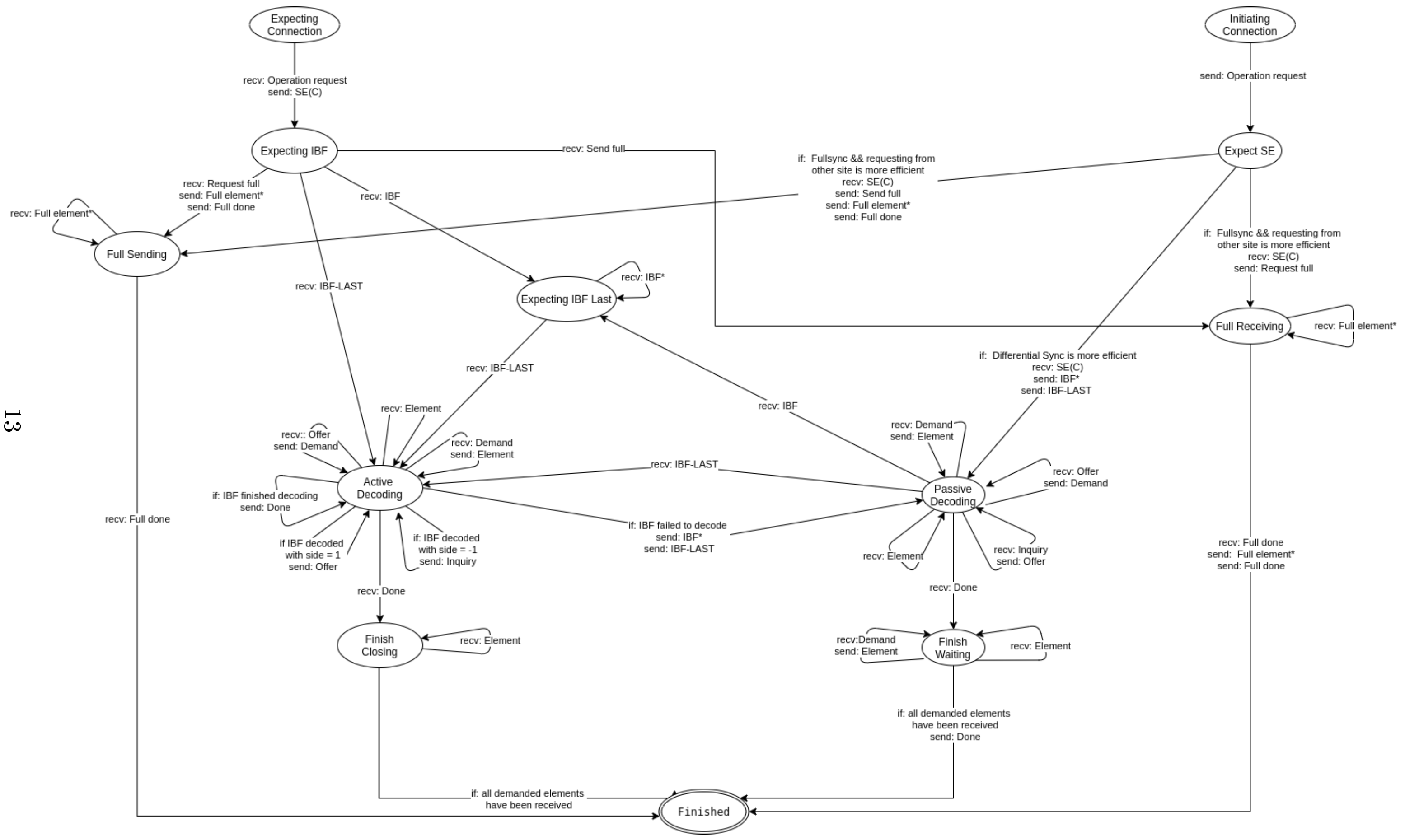


Figure 3: Statemachine diagram set union operation

3.4 Improve Strata Estimator prediction precision

A central point of the protocol is the initial estimation of the set difference at the beginning of the protocol. The set difference is an important input parameter to determine which mode of operation should be used so that the sets can be matched with as little effort as possible. The fact that the estimated difference is as precise as possible is also central to making stricter security assumptions and being able to exclude peers that do not follow the protocol more quickly.

3.4.1 Added Support for Multiple Strata Estimators

The simplest and most obvious way to improve an estimate is to carry it out several times and to average it. This is based on the "Monte Carlo methods" and the underlying stochastic law "of large numbers" [1] which applies to random experiments:

> “Many quantitative problems in science, engineering, and economics are nowadays solved via statistical sampling on a computer. Such Monte Carlo methods can be used in three different ways: (1) to generate random objects and processes in order to observe their behavior, (2) to estimate numerical quantities by repeated sampling, and (3) to solve complicated optimisation problems through randomized algorithms.“[12]

Here, the Monte Carlo methods are applied in a weakened variant, as described in the quoted part under point 2.

In the original implementation, only one Strata Estimator is exchanged in each case and the estimate calculated from this is the estimate used for the further protocol. The new improved implementation should improve this and send 1,2,4 or 8 Strata Estimators (n) depending on the local set size. Based on these estimates, the average can be formed and an estimate with higher precision can be achieved.

To create statistically independent Strata Estimators, the elements must be salted with the salting function known for salting the IBF.

The elements added to the IBFs of different Strata Estimators must be seeded differently. So that the elements for different Strata Estimators are mapped to different buckets. If the IBFs are not salted, then identical Strata Estimators are created. In this case the estimation is the same for all Strata Estimators and there is no added value by transferring multiple IBFs. The implementation of salting as a bit rotation has the advantage that it is very simple, very easy to undo and gives a good distribution over the buckets of the IBFs. The shift by 7 bits was chosen because 7 is a prime number and therefore also a generator of the group 64[13].

[1] `https://www.investopedia.com/terms/l/lawoflargenumbers.asp`

The salting function:

```
Algorithm 2 Salting function IBF/Strata Estimator
    Inputs:
    value: Input value to salt (needs to be 64 bit unsigned)
    salt: Salt to salt value with
    Output:
    Returns: Salted value
    FUNCTION salt(value, salt)
        s = (salt * 7) % 64
        RETURN (value >> s) | (value << (64 - s))
    END FUNCTION
```

When the implementation was adjusted for multiple Strata Estimators, the salts 0,1,2,3,4,5,6 and 7 were applied to the 8 different Strata Estimators. In a first attempt to adjust the implementation, the average estimates did not become more precise as expected, but the estimates became less precise. The reason for this was that the 8 Strata Estimators were statistically dependent and not independent as expected. The statistical independence could be improved by multiplying the salt by 7 (hence the "salt * 7" in the pseudocode of the salt function). As the table below shows, this led to a significantly better estimate:

	salt	salt * 7	Improvements
Experiments	200’000	200’000	-
Strata Count	4	4	-
Mean	+1	0	100%
StdDev	173	93	54%
Media	-18	-4	450%
Min	-494	-472	4.6%
Max	+1314	+820	60%
Percentiles 99	+512	+252	200%
Percentiles 75	+102	+58	76%
Percentiles 50	-18	-4	450%
Percentiles 25	-120	-62	94%
Percentiles 1	-328	-200	64%

Table 1: SE improvements through multiplying salt by 7

The graphical representation of the numbers shows this even more clearly: In the plot on the right, the salt was multiplied by 7:

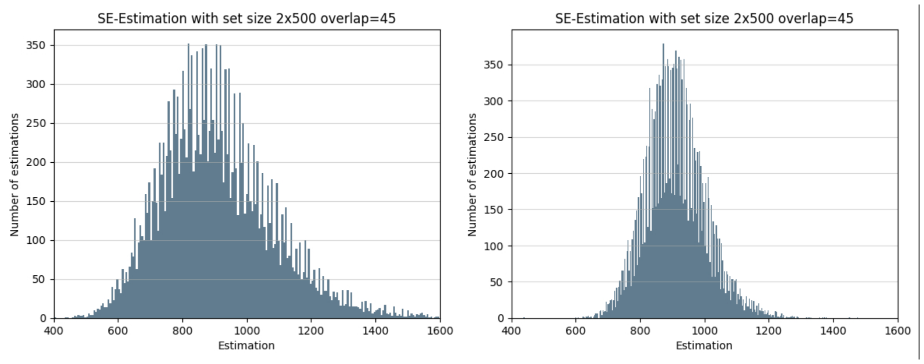


Figure 4: SE improvements through multiplying salt by 7 / Actual set difference is 910

These plots show that the distribution of the estimates is closer together and the estimates have become more precise. This suggests that the statistical independence of the Strata Estimators has been significantly improved by this small change.

3.4.2 IBF Size in Strata Estimator

As a further change, an attempt was made to set the number of buckets of the IBFs in the Strata Estimators to prime numbers, namely 61, 67, 71, 73, 79, 83, 89, 97. This only led to a small improvement. In the original implementation, the number of buckets of the IBFs in the Strata Estimator was set to 80. In the new implementation, the number of buckets was set to the prime number closest to 80, that is to 79. A small positive effect is nevertheless measurable. The explanation why using a prime number as the number of buckets improves the rate of successful decodes, is that the hash of the elements is distributed among the buckets by an operation modulo the number of buckets. This means that the distribution of the elements is best , when the modulo operator is a generator of the whole group and not only of a part of it. It follows from Lagrange's theorem[13] that the order of any element (i.e. the size of the group generated by any element) of a group of size p must be either 1 or p. Since the improvement is very simple and also provides better results, the number of buckets for IBF was set to the minimum of 79 in the new implementation.

	IBF Bucket count 80	IBF Bucket counts primes	Improvements
Experiments	20'000	20'000	-
Strata Count	4	4	-
Mean	+2	0	-
StdDev	154	146	5%
Media	-14	-12	17%
Min	-434	-458	-6%
Max	+1178	+1038	13%
Percentiles 99	+438	+414	6%
Percentiles 75	+94	+86	9%
Percentiles 50	-14	-12	17%
Percentiles 25	-106	-102	4%
Percentiles 1	-290	-290	0%

Table 2: Performance impact IBF bucket number is a prime in Strata Estimator

Although the improvement in the estimate is in the single-digit percentage range for most values, the change does not increase the implementation complexity as it can also be implemented without effort and is thus worthwhile.

3.4.3 Number of Strata Estimator dependant on Set Size

The most important disadvantage of sending multiple Strata Estimators is that additional bandwidth is consumed by the additional Strata Estimators. Since with small sets the set difference estimation plays a much smaller role and the additional bandwidth consumed plays a correspondingly larger role, it makes sense to make the number of Strata Estimators sent variable and dependent on the local set size.

An important parameter to determine how many Strata Estimators should be transmitted is the effective size of the Strata Estimators. As can be seen from the table below, the size is linearly increasing. It can be assumed that approximately 4221 bytes (8441/2) are used per Strata Estimator.

Number of Strata Estimators	Compressed Size	Enlargement factor
1	4266	-
2	8441	1.97
4	16828	1.99
8	33658	2.00

Table 3: Compressed Strata Estimator sizes

Based on these values, a heuristic was defined when 1, 2, 4 or even 8 Strata Estimators should be sent. The heuristic is defined in the following table. The rules are based on the average element size multiplied by the number of elements in the set. This value is referred to as "b" in the table:

Number of Strata Estimators	Implementation Rule	Avg Strata Estimator Size
1	b < 68k (4221 * 16)	4k
2	b > 68k (4221 * 16)	8k
4	b > 269k (4221 * 64)	17k
8	b > 1'077k (4221 * 256)	34k

Table 4: Determinating optimal number of Strata Estimators

3.4.4 Result

By adding support for a variable number of Strata Estimators, the precision could be significantly improved for larger sets. This can be seen in the following plot, where the deviation from the optimal value was plotted:

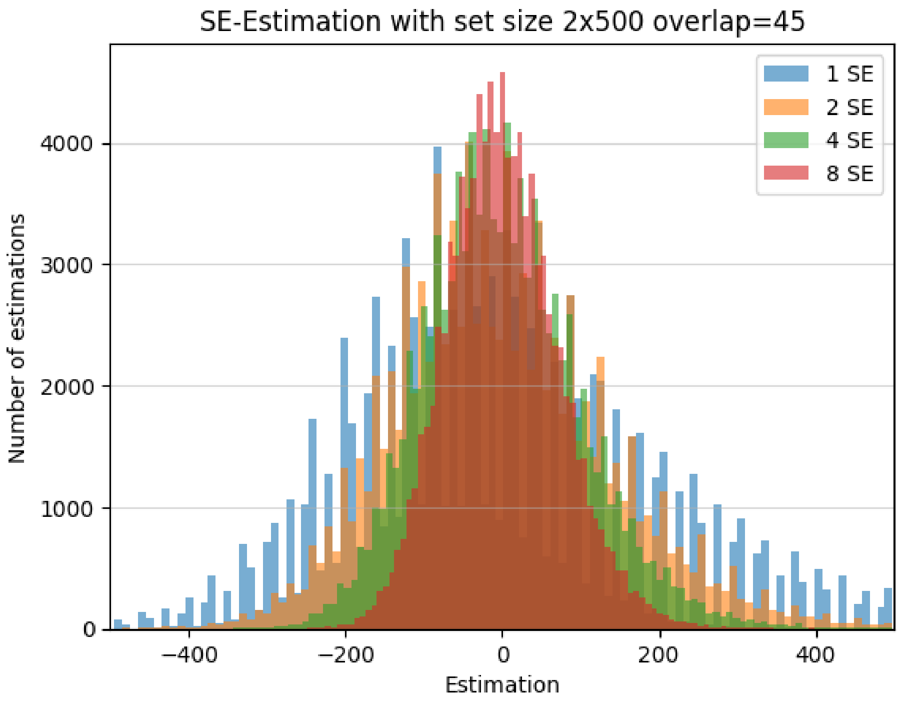


Figure 5: Improving estimates with multiple IBFs

It can be clearly seen that, as expected, the estimation becomes more and more precise the more Strata Estimators (SE) are applied to obtain an estimate. The spread is significantly reduced by each additional Strata Estimator and an estimate becomes much more precise. Precise estimates help in the security analysis to get tighter bounds for the likelihood of decoding failures with benign participants, and can thus terminate the protocol with malicious participants earlier.

3.5 Message Changes

During the implementation of the performance and security improvements, it became necessary to adapt the contents of various messages and to define new messages:

3.5.1 Send Full Message

In the original implementation, in the full synchronisation mode of operation, either a "Request Full" message was used to announce that the other peer should transmit his set or the sending of the "Full Element" messages implicitly communicated that the peer would transmit its own set first.

In this case, for security hardening, it became necessary to introduce a new message that allows meta data about the sets to be exchanged before all elements are sent, in particular the calculated set differences. More details on the new message structure can be found in the RFC in the Appendix B - RFC.

3.5.2 IBF Message

In the IBF message, new fields were added and existing fields were adapted, as these were needed for the validation of the Mode of Operation:

- An existing 8-bit field, an 8-bit padding field (reserved1) and a second 16-bit padding field (reserved2) have been converted into a 32-bit IBF size field. This is necessary because in the old implementation the IBF size was given using a logarithmic order (2^order=size), but now the size of the IBF is no longer given in orders of magnitude, but as a discrete number.

- With the introduction of the variable counter size, the maximum counter in the IBF also had to be transferred; for this purpose, an unsigned 16-bit value was inserted.Strata Estimator Message

During the optimisation of the Strata Estimator, a new field was introduced in the Strata Estimator Message, because a variable number of Strata Estimators can now be transmitted. The new field defines how many Strata Estimators are in the message.

3.5.3 Operation Request Message

In the existing implementation, an "Operation Type" was included in the "Operation Request". This field could take the following values: NONE, INTERSECTION and UNION. Since only the UNION operation is supported by this protocol, this field is superfluous and can be removed. In this way, 32 bits can be saved.

3.5.4 Inquiry Message

In the "Inquiry" message, the old implementation had a "reserved padding" field that was filled with zeros and did not contribute to the alignment. This has been removed, thus saving another 32 bits per "Inquiry" message.

3.6 Error in Salting of IBF

One of the more serious bugs in the original implementation was that if the first IBF did not decode, in the majority of cases the next IBF (doubled in size) did not decode either. This was because the salt used to create the IBF was always the same for both peers. This means that if, for example, one peer created an IBF with the salt "1" and the other peer could not decode it, the other peer also created an IBF with the salt "1" in the first round. Only when a new switch was made, did the first peer change its salt to "2". In the case of differential synchronisation, this meant that in practically every case when the decoding of the first IBF failed, an additional half round trip had to be spent. This could be solved relatively easily by starting the salts of the peers in a staggered manner; for example, one starts at 0, the other at 31. The improvement becomes clear when one looks at the table below, which shows how many simulations (all differential mode of operation) required how many active/passive switches:

Active/Passive Switches	Buggy Implementation	Fixed Implementation
0	47498	40457
1	0	8841
2	2493	656
3	1	29
4	7	17
5	0	0
6	0	0
7	0	0
8	1	0
9	0	0
Total Simulations	50000	50000

Table 5: Buggy vs. fixed IBF salt implementation

One can clearly see that the improved implementation shows a better pattern that corresponds to what one would expect. It is noticeable that the fixed implementation seems to need more active/passive switches even though they are better distributed. This is to be expected, as some unrelated parameters of the original implementation were chosen much more aggressively. However, this is reflected negatively in the required bandwidth, which cannot be taken from this table.

3.7 Additional Phases

In the original implementation, phases were missing and the implementation was extended by these phases.

3.7.1 Full Receiving

The original implementation did not have the "Full Receiving" phase, which was added in the revision. In the old implementation, all messages newly processed in the "Full Receiving" phase were processed in the "Expect SE" phase. This new phase is necessary to make a more fine-grained message validation. It makes the state diagram clearer and easier to understand.

3.7.2 Renamed Phases

Various phases in the code have been renamed to correspond to the state diagram:

- The "Done" phase has been renamed "Finished".
- The "Expect IBF" phase has been renamed "Expecting IBF".
- The "Inventory Acitve" phase has been renamed "Active Decoding".
- The "Inventory Passive" phase has been renamed "Passive Decoding".
- The "ibf cont" phase has been renamed "Expecting IBF Last".

3.8 Minimal Number of IBF Buckets

In the original implementation, the minimum number of buckets in an IBF was limited to 16. This lower limit was increased to a prime number, 37, due to various experiments and experiences. This reduces the risk of an active/passive switch for small set differences. Although this requires additional bandwidth, it still pays off, as synchronisations with small set differences often occur in a peer-to-peer network.

3.9 Determinate Average Element Size

For many applications, it can be assumed that all elements in the set have the same size. However, there are also areas of application where the size of the elements differs.

The average element size is a central parameter for performance optimisation, also for security assumptions. Therefore, an adaptation to the actual element sizes is desirable. A simple function was written that functions as an iterator and can be iterated over each element in a hash table:

Algorithm 3 Function to calculate average element size

```
Input:
elements: Hashtable which contains all elements
Output:
Returns the average size over all elements
FUNCTION get_avg_size(elements)
    total_size = 0
    FOR element in elements DO
        total_size  += size(element)
    END FOR
    RETURN total_size / len(elements)
END FUNCTION
```

3.10 Determinate Maximal IBF Counter

To compress the IBF counters, a function was needed that could determine the minimum number of bits needed to represent the highest counter, based on the uncompressed IBF. The following pseudocode implementation was developed for this purpose:

Algorithm 4 Function to determinate minimal IBF counter size

```
Input:
ibf: The IBF
Output:
Return the minimal required bit size of the counter to
represent all counters
FUNCTION ibf_get_min_counter_size (ibf)
    max_counter = 0
    FOR (counter in ibf.counters) DO
        IF (counter > max_counter) THEN
            max_counter = counter
        END IF
    END FOR
    RETURN 64 - COUNT_TRAILING_ZEROS(max_counter)
END FUNCTION
```

As “COUNT_TRAILING_ZEROS“ function a fast internal compiler function to count the trailing Zeros can be used (in GCC for example __builtin_clzll [2])

[2]https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

3.11 Improve IBF Size

The following graph shows the probability that the first IBF does not decode. The simulation was done against the original code without any changes to the code concerning the performance.

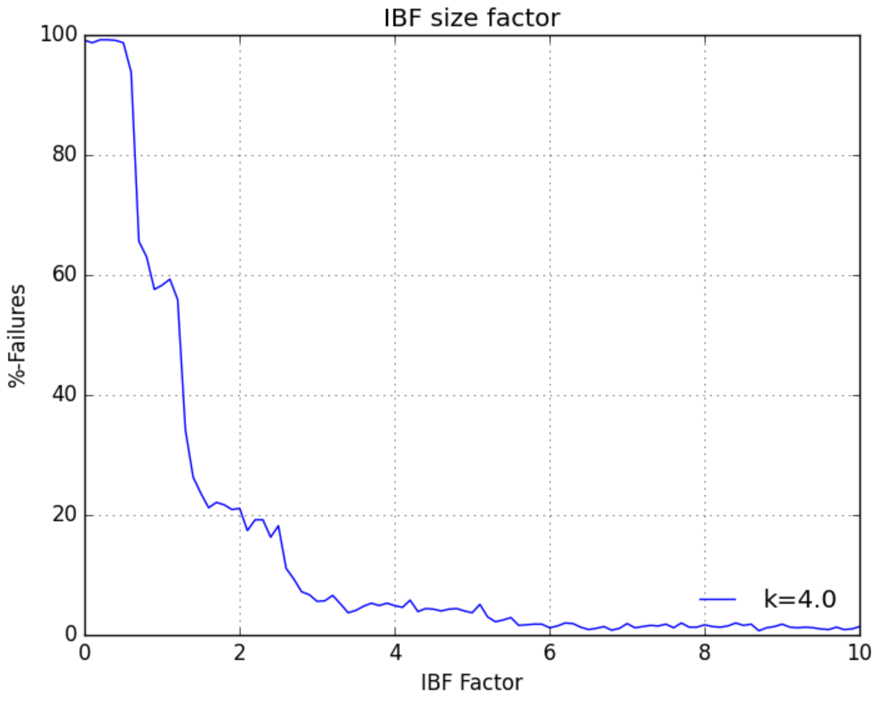


Figure 6: Graphical representation of the original IBF decoding failure rate

The most obvious items in this graph are the four plateaus in the graph. The plateaus can be explained by the nature of how the number of IBF buckets is defined in the original implementation. The number of buckets is defined as estimated set size difference multiplied by the IBF factor. This is rounded up to the next larger power of two (...1024, 2048, 4096...). This logarithmic gradation was probably implemented in the initial implementation for simplicity and needed to be corrected to allow a more granular definition of the IBF sizes. It is unclear why this was implemented this way.

The implementation was adjusted so that the IBF factors could be passed as exact discrete numbers.

Plotting the new implementation that does not round up to the next power of two results in the following graph:

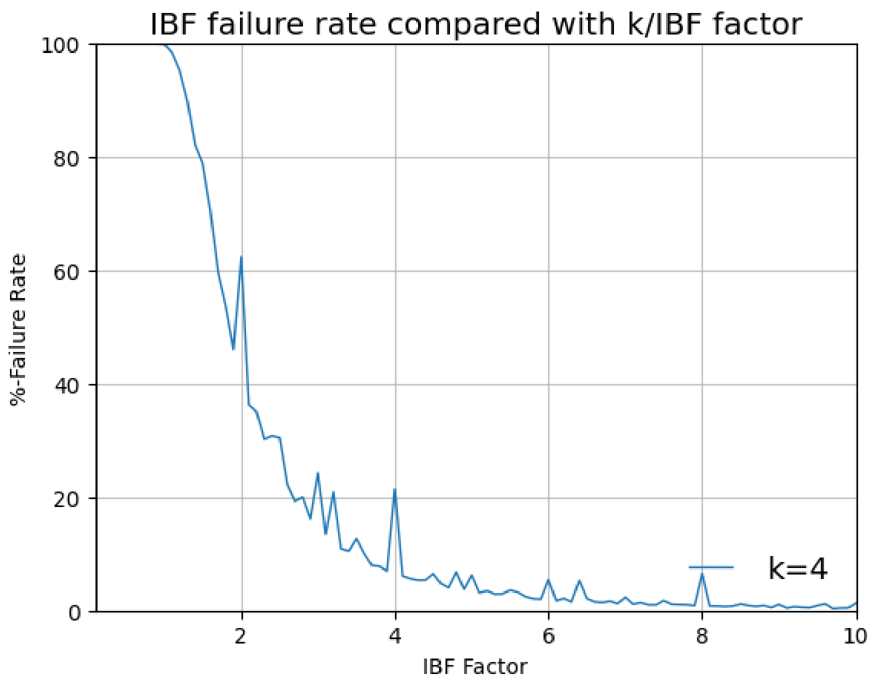


Figure 7: Graphical representation of the IBF decoding failure rate without logarithmic steps

This plot looks more like what would be expected. What can be clearly seen are the significantly increased error rates for factors 2, 3, 4, 5, 6, 7, and 8. This is best explained by the fact, that when the estimate is multiplied by integer factors, there is an increased probability and for numbers two, four, six, eight (multiplicative of two), there is even exclusively an even sized IBF. For IBFs of even size the probability that the IBF multiple is not decoded is increased. This assumption is supported by an additional experiment in which the size of the IBFs was made odd by a binary or (| 1):

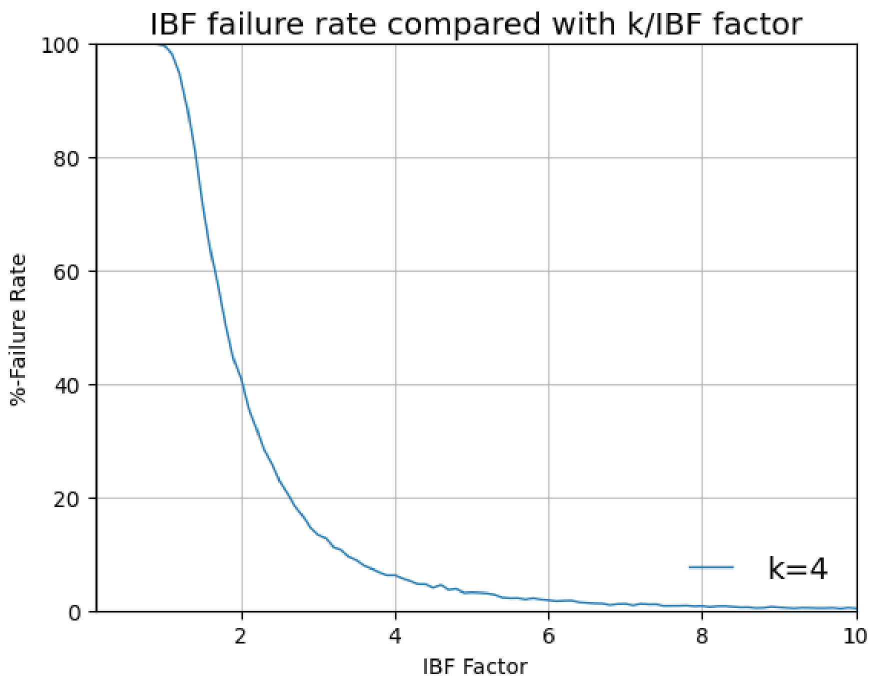


Figure 8: Graphical representation of the IBF decoding failure with an odd number of buckets

This graph looks exactly as expected and supports the assumption that IBFs of odd sizes give significantly better results.

The implementation was adapted by the two changes described, so that the performance parameter study in the section 4 can be carried out with discrete numbers.

3.12 Check Bucket Falsely Classified as Pure

An additional check was introduced to ensure that in case a bucket is falsely classified as pure when processing the Inquiry message, a role change must take place and an IBF must be received.

4 Performance Tests

The performance measurement is focusing primarily on the "differential synchronisation" operation mode, because the "full synchronisation" mode always has an easily predictable cost given the number of elements and their average size.

There are two tuning parameters for which a suitable value must be found:

- The factor that the set size difference estimate is multiplied with, which then defines the number of buckets the IBF is created with.
- The IBFs k value, which defines the number of buckets an element is mapped to.

Given these tuning parameters, the protocol then needs to additionally compute the relative set difference. The set difference must be as precise as possible and is calculated with a Strata Estimator. Improvements on the Strata Estomator can be found in section 3.4.

For all of these values, the following performance indicators should be measured:

- The probability in percentage the IBF does not decode successfully.
- The round trips used for the protocol.
- The bytes transmitted for the synchronisation.
- Accuracy of the set size difference estimation.

4.1 Goals

The main goal of the performance tests is to find the optimal parameter (number of buckets and number of buckets per element) for the IBF generation and find the optimal threshold to decide whether full synchronisation or differential synchronisation is to be prefered.

The number of buckets in an IBF should be defined by a factor multiplied by the estimated set size difference, so the number of buckets should be defined as a linear function. Due to the structure of the algorithm, it seemed obvious from the beginning that it could be a linear function. However, the hypothesis that it is a linear factor was to be proven experimentally.

The number of buckets per element should be defined as a discrete positive integer.

The threshold defining the mode of operation should be adjustable by a user input defining a tradeoff between bandwidth and round trips (bytes/RTT).

4.2 Test Sets Generation

To do performance tests it is vital to be able to randomly generate two sets of elements. The set generation function needs to take as input the size of set 1, the size of set 2, the numbers of elements that are the same in both sets (overlap) and the size of an element. This function should then return two sets of elements which fulfill the defined parameter.

The functions definition:

```
Algorithm 5 Code to generate single random element
    Input:
    number_of_bytes: The desired size of the element
    Output:
    element: Pointer to the random generated element
    FUNCTION generate_random_element (number_of_bytes)
        *element = MALOC(number_of_bytes)
        SECURE_RANDOM_BYTE_GEN (element, number_of_bytes)
        RETURN element
    END FUNCTION
```

Algorithm 6 Code to generate two random sets with given overlap

```
Input:
overlap: Number of elements that overlap in the set
s1_size: Number of elements total in set 1
s2_size: Number of elements total in set 2
e_size: Byte size of a single element
Output:
sets to run set reconciliation algorithm against
FUNCTION initRandomSets (overlap, s1_size, s2_size, e_size)
    set1 = {}
    set2 = {}
    // Add overlapping elements to both sets
    WHILE o_ctr < overlap DO
        o_ctr = o_ctr + 1
        element = generate_random_element (e_size)
        add_element_to_set (set1, element)
        add_element_to_set (set2, element)
        s1_size = s1_size - 1
        s2_size = s2_size - 1
    END FOR
    // Add other elements to set 1
    WHILE set1_size > 0 DO

        element  = generate_random_element (e_size)
        add_element_to_set (set1, element)
        s1_size = s1_size - 1
    END WHILE
    // Add other elements to set 2
    WHILE set2_size > 0 DO
       element = generate_random_element (e_size)
        add_element_to_set (set2, element)
        s2_size = s2_size - 1
    END WHILE
    RETURN (set1, set2)
END FUNCTION
```

For the generation of the elements a secure pseudorandom number generator (CSPRNG) is required to ensure the statistically independence of each element. The full function can be found in the GNUnet Git[3] repository.

[3]the GNUNET_CRYPTO_random_block function in the file /src/util/crypto_random.c in the [4]

4.3 Measure the Round Trip Time (RTT)

For the protocol, the calculation of round trips for the protocol can be easily achieved by looking at the state diagram and counting the side switches of the messages:

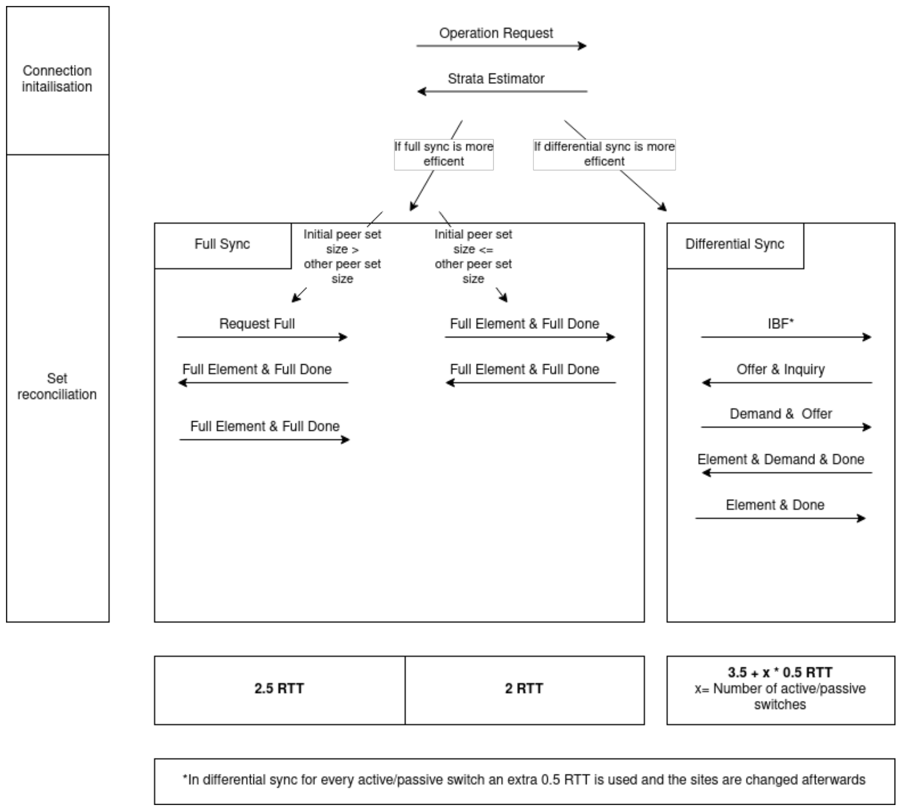

Connection initailisation
Set reconciliation
Operation Request
Strata Estimator
If full sync is more efficent
If differential sync is more efficent
Full Sync
Initial peer set size > other peer set size
Initial peer set size <= other peer set size
Request Full
Full Element & Full Done
Full Element & Full Done
Full Element & Full Done
Full Element & Full Done
Differential Sync
IBF*
Offer & Inquiry
Demand & Offer
Element & Demand & Done
Element & Done
2.5 RTT
2 RTT
3.5 + x * 0.5 RTT
x= Number of active/passive switches
*In differential sync for every active/passive switch an extra 0.5 RTT is used and the sites are changed afterwards

Figure 9: Analysis of the protocol steps based on the protocol statemachine.

The amount of RTTs used for a reconciliation depends on the synchronisation mode used and which side has the bigger set.

In optimal conditions the differential operation mode needs 3.5 RTTs and every IBF that fails to decode adds another 0.5 RTT (3.5 + x * 0.5). When x is the number of failed IBF decoding tries.

In case it is more favourable to request the initializing peer's set, the full synchronisation mode needs a total of 2.5 RTTs, and 2 RTTs otherwise.

With this knowledge a simple pseudocode function can be used to calculate the RTT used:

Algorithm 7 Code to calculate RTTs used

```
Input:
moo: Which mode of operation is used
switch_number: Active/passive switches required in
differential mode
Output:
Returns the number of RTTs used
FUNCTION get_rtt(moo, switch_number)
    IF moo == "FULL_SYNC_LOCAL_SENDING_FIRST" THEN
        RETURN 2.5
    ELSE IF moo == "FULL_SYNC_REMOTE_SENDING_FIRST" THEN
        RETURN 2.5
    ELSE
        RETURN 3.5 + (0.5 * switch_number)
    END IF
END FUNCTION
```

4.4 Measure the Bandwidth

Measuring the bandwidth is straight forward: the size of every message that is transmitted needs to be stored and then summed up to a total.

For some analysis it is interesting to filter just outgoing-, incoming or messages of a specific type, so it is a good idea to store the messages individually and sum them up in the end.

To determine the bandwidth, it is necessary to record how many messages of which type are sent and received by both peers. In addition, the variable share, if any, is added up for each message. If this data is available, the bandwidth can be easily determined with the following formula.

$$bandwidth = (number_{messages} \times size_{header}) + variable_{bytes} \tag{2}$$

Figure 10: Formula to determine bandwidth of a single message

To capture the entire bandwidth, the bandwidth of all messages can be added up. The header sizes are static and can be taken from the table below:

Message Type	Bytes
IBF	16
OFFER	4
DEMAND	4
INQUIRY	12
REQUEST_FULL	20
SEND_FULL	20
FULL_ELEMENT	8
STRATA_ESTIMATOR	13
DONE	4
FULL_DONE	4
ELEMENTS	8

Table 6: Static header sizes

4.5 Performance Tests

In the following section the performance tests are described in detail:

4.5.1 Strata Estimator Estimation Distribution

It is important to understand how good the estimation of the set difference is, because a good estimation is the basis of all performance optimisations. See section 4.5.3.

To measure the precision of the Strata Estimator implementation, the estimation is run 12 000 000 iterations with two sets which contain 5000 elements and have an overlap of 450 elements and plot the distribution:

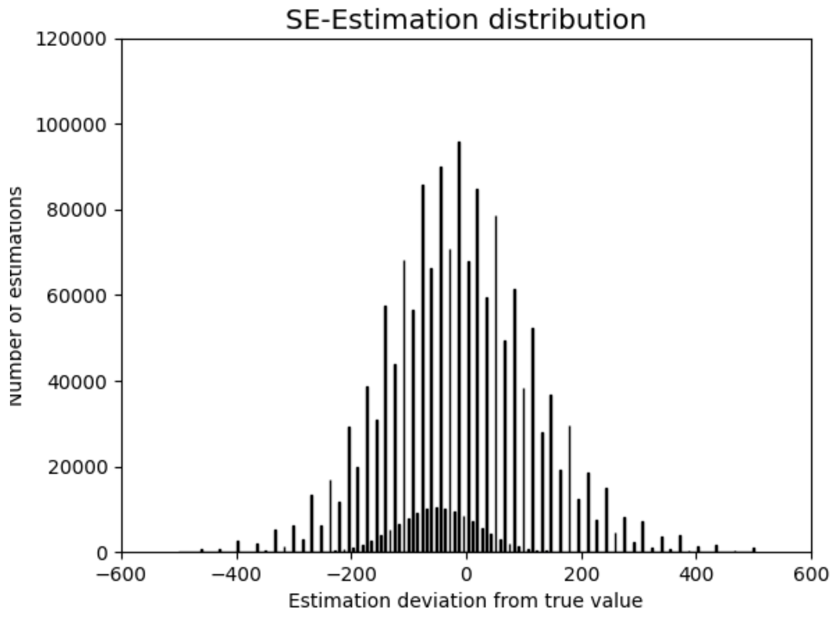


Figure 11: Graphical representation of the distribution of the Strata Estimator estimation

The distribution graph shows the deviation of the estimation from the real set difference.

The actual set difference in this case was 9100 (2x5000-2x450) elements. The mean over the 12 000 000 iterations shows a slight left shift and three normal distributed looking curves stacked over each other. This is what was expected:

The left shift means that the Strata Estimator has the tendency making a slight underestimation of the set difference. This can be explained by the inner working of the decoding of the Strata Estimator. Since starting decoding the IBFs, in the Strata Estimator which contains the smallest share of the elements and continuing decoding with IBFs containing more elements until no further IBF can be decoded. Then the set size difference can be estimated on basis of the successfully decoded IBFs. It is easy to understand that the set difference is often underestimated because bigger, only partly decoded IBFs are not honoreed. This explains why there are three visible sub curves (Actually, even more, but they are not visible in the graph because they are to flat) and every curve is shifted less to the left. The curves originate from the fact that not for every experiment the last successfully decoded IBF is of the same order. In this case most of the iterations were distributed over three different orders. The smallest and densest curve is the IBF with the smallest order.

4.5.2 IBF Parameter Study

An IBF should be generated to be as small as possible to save bandwidth. The failure rate should be low to save RTT and bandwidth. These two targets are partly contradictory (larger IBF $\Rightarrow$ lower failure rate but larger IBF $\Rightarrow$ more bandwidth).

The size (number of buckets) an IBF needs, is defined by a factor (we evaluated the range 0-10) of the estimated set size difference.

The number of buckets mapped per element (k) (we evaluated the range 1-10) defines how often a single element is mapped to the IBF. A higher value can increase the chances to get a pure bucket, but a too high value decreases the probability of getting pure buckets. This sounds counter intuitive, but it is easy to understand. Ex.: When mapping an element just once on the IBF, the chance that it is not possible to decode the IBF is very high, because only one other element is needed to be mapped on the same bucket and the IBF remains undecodable. The other extreme would be to map every element 10 times to the IBF, in this case the probability every bucket will be hit at least two times is very high and this leaves the IBF undecodable too. So to find a good k value is key.

The other value that is very important to optimize, is the IBF Factor. This factor is multiplied with the estimated set size difference, which results in the number of buckets the IBF is build with.

The following graph shows the probability that the first IBF does not decode for k=2-10 (k=1 was omitted on the following graphs because the protocol hardly worked anymore and made the graphs unreadable).

The performance of the original implementation corresponds to the point on the plot below, line k=4 and IBF factor 2. Subject to changes in the section 3.11.

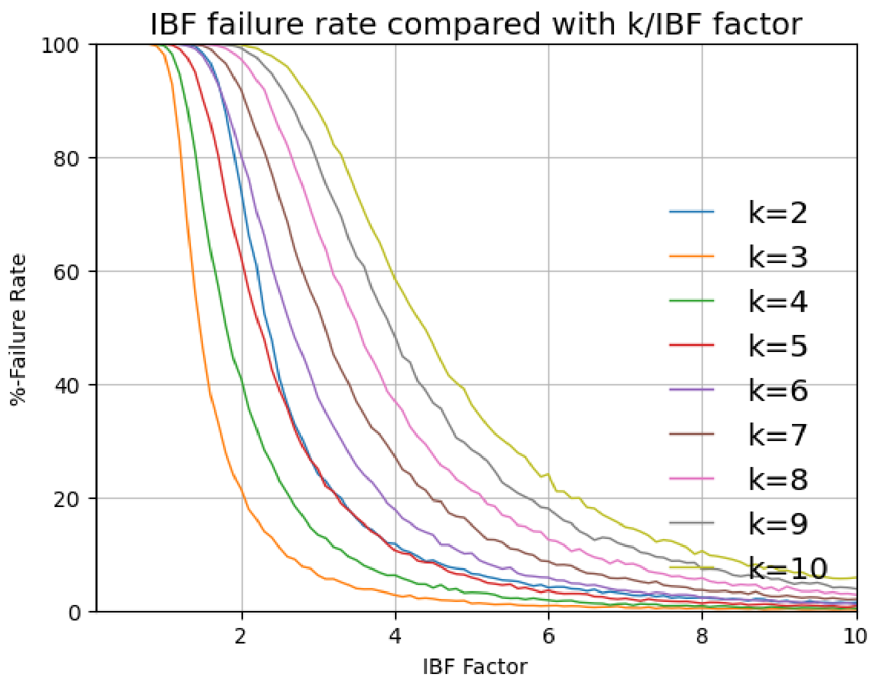


Figure 12: Graphical representation of the IBF decoding failure with odd number of buckets

It clearly shows that the k value of 3 is the best for each factor in terms of error probability. Regarding the IBF hash factor, as expected, the larger the factor, the smaller the error probability.

In addition to the error probability, the consumed bandwidth is also a decisive criterion for the transmission of the IBF.

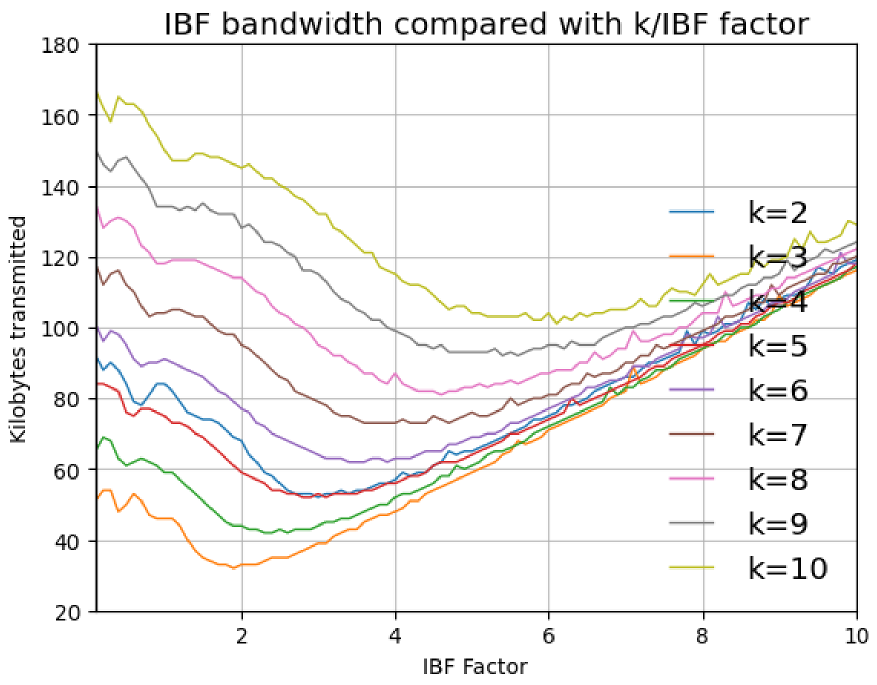


Figure 13: Graphical representation of bytes transmitted by different IBF sizes (IBF only)

In this graph, it can be seen that the k value of 3 is always optimal for the bandwidth. In addition, it can clearly be seen that a factor of around 2 is optimal when k is 3 for the bandwidth.

For connections with increased latency, the average round trips used are also decisive for the performance in addition to the bandwidth.

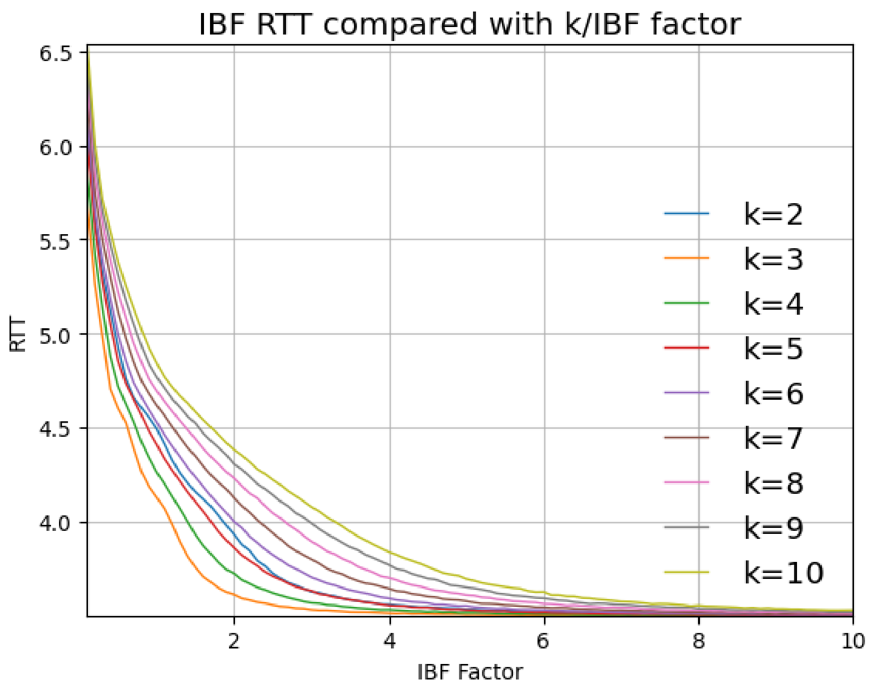


Figure 14: Graphical representation of RTTs required by different IBF sizes

This graph shows the average number of round trips per k value and IBF factor. There is no surprise here either: the k value of 3 seems to be optimal in terms of round trips. For the IBF factor, the larger the IBF, the fewer round trips are needed. This is easy to explain:

For the k value to generate the IBF, after looking at these graphs, it clearly shows that the best value is 3. For the IBF factor, it is not so clear. The question has to be answered, if a static IBF factor is sufficient in all situations or if it has to be adjusted due to the quality of the connection between the two peers.

To answer this question, five possible peer-to-peer connections scenarios were defined.

To calculate the latency, a physical distance, the distance between Bern and Frankfurt was assumed, 420km[14].

According to pingman.com, data packets can be assumed to move forward at about 192,000km/s:

> "The distance the data is traveling. Data travels at (very roughly) 120,000 miles (or 192,000 kilometers) per second, or 120 miles (192 km) per ms (millisecond) over a network connection. With traceroute, we have to send the data there and back again, so roughly 1 ms of latency is added for every 60 miles (96km, although with the level of accuracy we're using here, we should say '100km') of distance between you and the target"[15]

Using this example to calculate the latency between Bern and Frankfurt, it can be seen that a data packet takes about 4 ms.

In addition, the latency varies depending on the modem type:

> "The latency of the connecting device. For a cable modem, this can normally be between 5 and 40 ms. For a DSL modem this is normally 10 to 70ms. For a Dial-Up modem, this is normally anywhere from 100 to 220ms. For a cellular link, this can be from 200 to 600 ms. For a T1, this is normally 0 to 10 ms"[15]

The following graphs were generated with the following parameters:

Type	Bandwidth	Modem latency[2]	Total latency	Bandwidth-latency-tradeoff
Datacenter	10Gb/s	5ms	9ms	1,111,111,111
Fiber to the home (FTTH)	1Gb/s	10ms	14ms	71',428,571
DSL Fast	100mb/s	35ms	39ms	2,564,102
DSL Slow	10mb/s	35ms	39ms	256,410
Dial-Up	128kb/s	151ms	155ms	825

Table 7: Testcases defined to determinate practical impact of measurements

To calculate the optimal IBF factor for all five connection types, a graph is drawn for each type with the milliseconds required to execute the protocol plotted on the Y-axis

The milliseconds spent are calculated for each slice of 0.1 IBF factor using the following formula:

ar: Average round trip time used
lr: Total latency as stated in the table above
ab: Average bytes required for reconciliation
bs: Bytes transmitted per ms

$$t = ar \times lr + \frac{ab}{bs} \tag{3}$$

Figure 15: Function to calculate milliseconds used for different testcases

Three different lines are plotted, the red line showing how much time was spent transferring the data, the green line showing how much time was spent on round trips, and a blue line showing the data from the red and green lines added together.

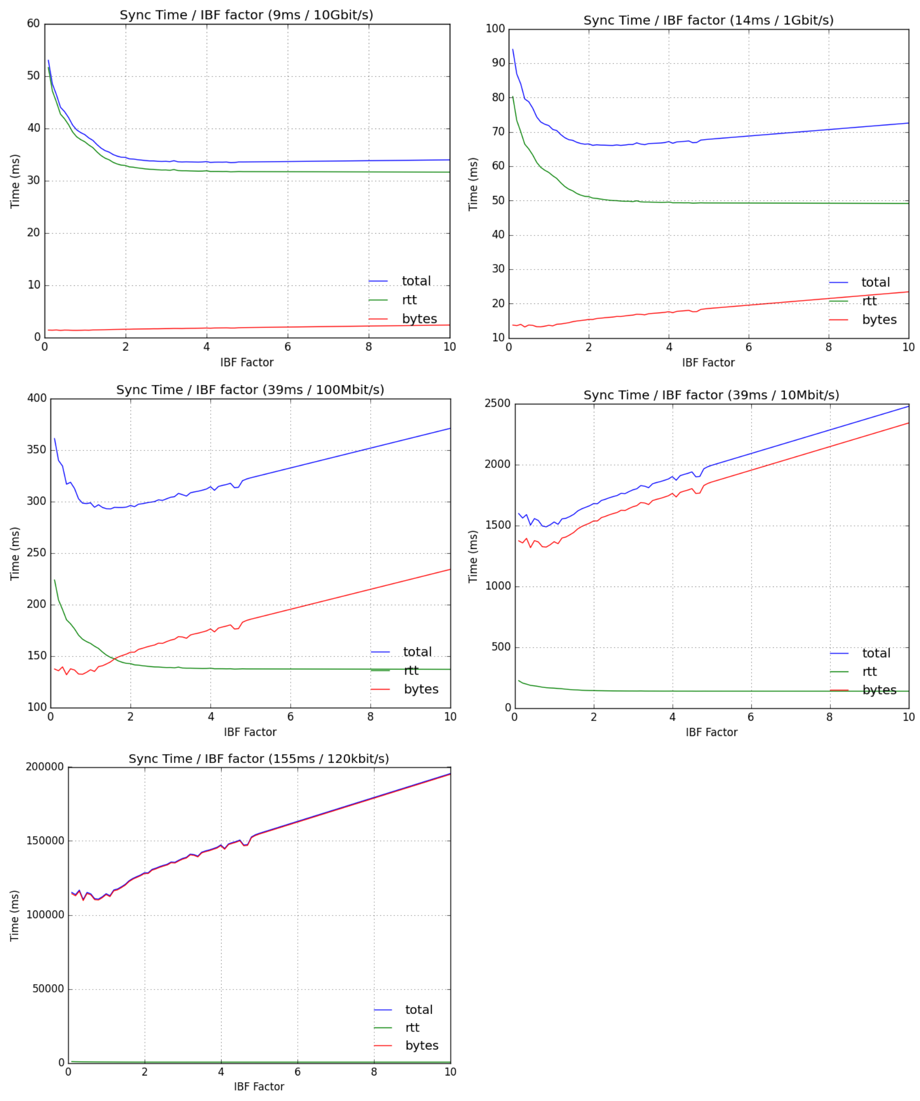


Figure 16: Graphical representation of milliseconds used for reconciliation in Datacenters, FTTH, DSL Fast, DSL Slow and Dial-Up (from left to right, from top to bottom)

The plot for the data center clearly shows that the bandwidth is so high that the number of bytes transferred has practically no influence on the total time spent and practically only the number of round trips counts. Therefore, the larger the IBF, the better. Fewer RTTs are needed, which have the greatest influence on the time spent. From an IBF factor greater than 2, no significant improvement can be seen.

With a home fiber connection, it can be seen that a larger proportion of the total time must be spent on transferring the data. The optimal IBF factor for the fiber is approximately 2, since the total time spent is lowest with this factor.

With DSL Fast connections, it becomes apparent for the first time that for a factor of less than ten, the time that has to be spent on transferring data exceeds the time consumed by the round trips. This results in an optimal plateau between the factors 1.5 and two.

This trend continues with DSL Slow connection. The used round trips become less important compared to the time that must be spent to transfer the data. Here it becomes clear that a value of 2 and less is optimal to achieve the shortest possible transmission time.

With a Dial-Up connection, this trend continues, the time for the RTTs becomes less important and the time to transmit data becomes practically the sole influencing factor. This type of connection is very uncommon today and can be neglected for our considerations.

What can be said about the IBF factor, is that the larger the bandwidth, the more crucial it becomes to save round trips. In a peer-to-peer network today, it can be assumed that the average internet connectivity in developed countries for home internet connections is between 10mb/s and 100mb/s [3] and it can be assumed that most servers have 1Gb/s connectivity. Therefore, this implementation focuses on FTTH, DSL Fast and DSL Slow type connections. It is clear from the graphs that a value between 1.5 and 2 is optimal for these.

According to the IT trade magazine IT-Daily, the bandwidth for private connections has increased by up to 147% since 2017.

> "Bei dem Vergleich dieser Top-25 Geschwindigkeitslaender zur Veränderung der Geschwindigkeit seit 2017 schafft es Deutschland immerhin auf Platz 18 - 31% Besserung sind hier zu verzeichnen, waehrend Spitzenreiter Taiwan eine 147% Verbesserung verbuchen kann."[16]

Quote translation:

> "In the comparison of these top 25 internet speed countries on the change in speed since 2017, Germany still makes it to 18th place -31% improvement is recorded here, while frontrunner Taiwan can record a 147% improvement."

Since bandwidth is expected to continue to increase in the future and it is not a short-term trend, the influence of bandwidth will become less and less the decisive factor and round trip time will become more important.
Based on all these measurements, an IBF factor of 2 is future-proof and behaves almost optimally for all relevant connection types.

The original work of Eppstein largely aligns with the results of our experiments with the optimized implementation. In these experiments, in contrast to the original work of Eppstein, it could not be determined that for set differences smaller than 300 elements a k value of 4 would be better:

"In practice, a simple rule of thumb is to construct an IBF in Phase 2 with twice the number of cells as the estimated difference to account for both under-estimation and IBF decoding overheads. For estimates greater than 200, 3 hashes should be used and 4 hashes otherwise."[3]

4.5.3 Differential vs. Full Mode

One of the most important performance influencing factors is the choice between differential and full mode of operation.

This question is about whether it is more efficient to transmit the entire set or to determine the set difference and transmit only the missing elements.

An additional possibility with full synchronisation is, that it is more efficient to request the elements from the other side, instead of transmitting the elements to the other peer. This makes sense when the additional half round trip that is required to request the set from the other peer. This is especially beneficial if round trips are much cheaper than bandwidth and more elements of the estimated difference are at the other peer.

The previous implementation of the Strata Estimator in GNUnet did not have the functionality to estimate on which side of the peer-to-peer operation how many elements of the difference are.

Therefore, the implementation had to be extended accordingly:

When decoding the IBFs of the Strata Estimator, it had to be additionally stored on which side the element has decoded (1/-1). This ratio can be used to estimate how many elements are missing on one side and on the other side.
To decide which mode is optimal, a function must be written that can estimate how much bandwidth will be used for the operation, based on the input parameters: Average element size, local and remote set sizes and local and remote set difference to estimate how much bandwidth will be used for the operation. Furthermore it should be possible to specify a bandwidth latency tradeoff which defines how many bytes an additional round trip should cost. This is necessary so that the protocol can be adapted and optimized for different applications.
In order to be able to carry out these calculations, various constants are also required. For example the message sizes, which can be taken from the RFC in the Appendix B - RFC.
The exact implementation can be found in the RFC as pseudocode, or the concrete implementation can be found in the GNUnet Git [4] repositories. The developed pseudocode is included as a Python script in this work.

To verify the estimates of the written algorithm and to estimate how precise they are, 10'000 simulations were performed, and the actual bytes transferred were compared with the number of bytes estimated by the algorithm.

Five experiments were made with different set size differences:

		Ex 1	Ex 2	Ex 3	Exp 4	Ex 5
Set size 1		5000	5000	5000	5000	5000
Set size 2		5000	5000	5000	5000	5000
Sets Overlap		0	1250	2500	3750	4500
Sets Overlap in %		0	25	50	75	90
Difference		10000	7500	5000	2500	1000
Strata		9850	7367	4929	2470	984
RTT		3.656	3.649	3.628	3.619	3.614
Strata/RTT		39.033	38.065	36.217	35.416	34.995
Total bytes	Simulation	2'372kb	1'708kb	1'177kb	584kb	233kb
	Estimated	2'309kb	1'732kb	1155kb	577kb	231kb
IBF bytes	Simulation	384kb	280kb	178kb	87kb	34kb
	Estimated	349kb	262kb	175kb	87kb	35kb
Deviation predict./sim.		63kb	-24kb	22kb	7kb	2kb
%-Deviation predic./sim.		2.713	-1.366	1.932	1.201	0.904

Table 8: Accuracy comparison table of new algorithm

The deviation of the algorithm's estimate from the average across all simulations is very precise, ranging from 2.71% to -1.366%.

It is noticeable that the number of RTTs needed for the operations is very constant at 3.6, no matter how large or small the set difference is.

The fact that the strata estimation is always slightly lower than the effective difference is due to the way the Strata Estimator works and is expected. However, this bias is compensated by the experimentally derived factors used in the implementation.

4.6 Results

Since part of the work is to measure and quantify performance improvements. At the beginning of the project the initial performance of the implementation was measured and at the end, the achieved improvement was measured by an identical simulation. The results were presented in the following plots.

All graphs used for measuring the code improvements were made with two sets, each containing 500 elements at 32 bytes and having an overlap applied on the X-axis. The resolution of the graph is such, that the less improved "Full Synchronisation Mode of Operation" contains measurement points in increments of 100 (0,100,200,300,400). The interesting section, which was improved by applying mainly the "Differential Synchronisation Mode of Operation" contains measuring points in steps of 10 (410,420,430,440,450,460,470,480,490). Each of these measurement points is the average value over 10'000 simulations.

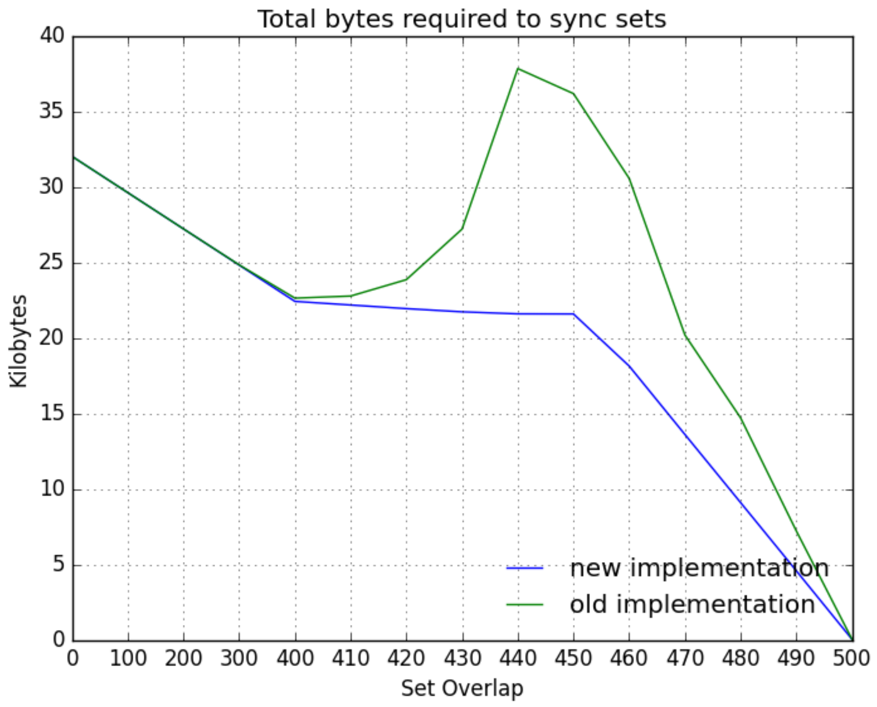


Figure 17: Graphical representation of bandwidth improvements through new implementation with different set differences. Bandwidth-latency-tradeoff set to 10,000 bytes/round trip.

In this first graph, it can be seen in green the original implementation and in blue the improved implementation. This graph shows on the Y-axis the total bandwidth of the operation required at a certain set overlap to synchronise the two sets of peers.

It is easy to observe that the new implementation of differential synchronisation has significant bandwidth savings over the old implementation, and approaches the optimal "linear" behavior across the entire spectrum of set differences. This is quantified again in the table below in exact number of bytes and percentages.

An indication that the algorithmic estimation of the set sizes is very precise, is that the graph of the new implementation does not show a significant increase, when switching between full and differential synchronisation, as can be clearly seen with the old implementation.

Set Overlap	Old Implementation	New Implementation	Difference in %
0	32008	32010	0%
100	29608	29610	0%
200	27208	27210	0%
300	24825	24817	0%
400	22623	22451	-1%
410	22761	22251	-3%
420	23845	22044	-8%
430	27194	21910	-20%
440	37821	22090	-42%
450	36161	22924	-37%
460	30549	20115	-35%
470	20182	15033	-25%
480	14691	10053	-32%
490	7206	5047	-30%

Table 9: Numerical representation of bandwidth improvements through new implementation with different set differences

The new implementation results in bandwidth savings of 20% to 42% in cases, where the set difference is small, which is a massive improvement of the protocol in terms of the bandwidth used.

In addition, in this simulation the difference of the elements were equally distributed to both peers, if this would not have been the case (as it would be very often in reality) the new implementation would be superior to the old one, even in full synchronisation in 50% of the cases, because it can estimate in which cases it is worth requesting the set from the other peer.

The second important aspect of measuring performance is the road trips needed to synchronise the two sets. The following graph shows the required RTTs for the complete synchronisation of the two sets.

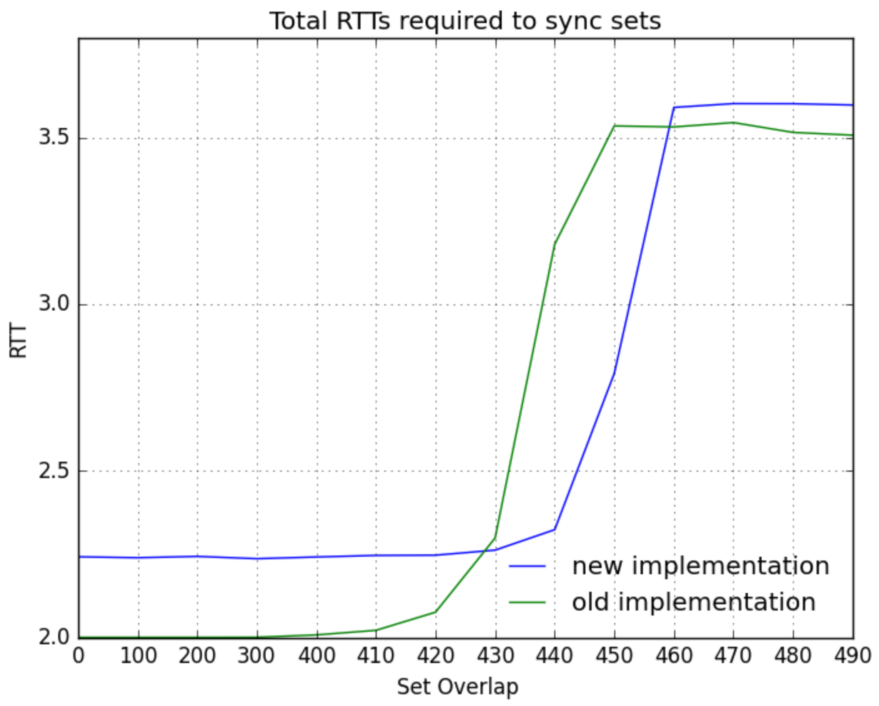


Figure 18: Graphical representation of RTTs improvements through new implementation with different set differences. Bandwidth-latency-tradeoff set to 0 bytes/round trip.

What might be surprising at first glance, is that the new implementation is not always better. For example, it always seems to be 0.25 RTTs worse in full synchronisation. But this can be explained very easily, because for these tests the algorithm for the choice of the "Mode of Operation" was configured so, that the "bandwidth-latency-tradeoff" was 0. Therefore, in 50% of the simulations the protocol needs an additional 0.5 RTTs, which could easily be avoided by setting the bandwidth latency tradeoff differently.

If the "Bandwidth Latency Tradeoff" is increased from 0 to 10'000, which means that 1 roundtrip is worth 10'000 bytes, a different picture emerges:

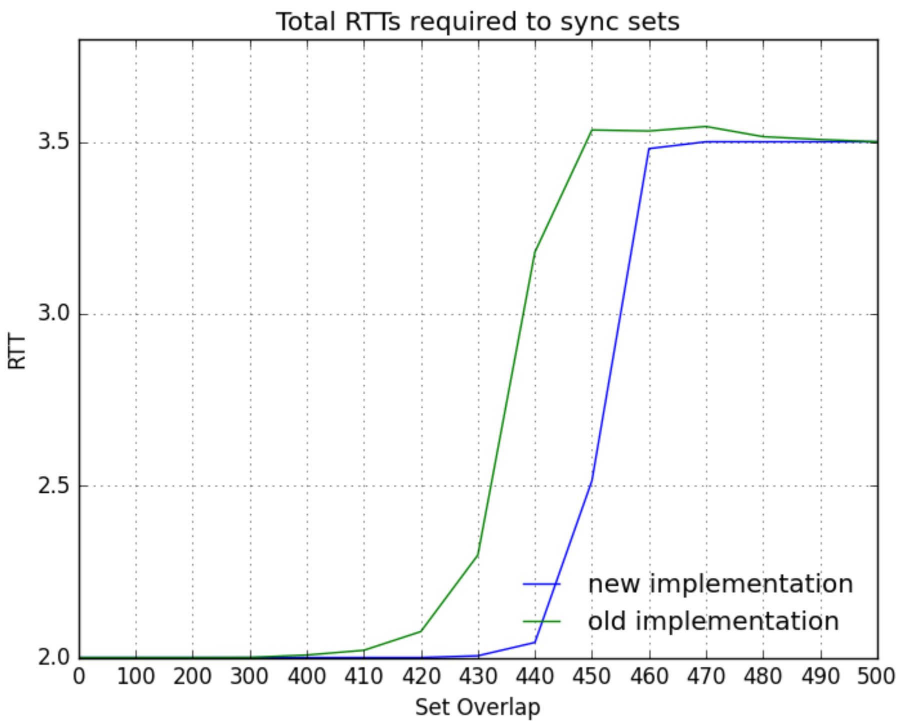


Figure 19: Graphical representation of the round trips. Bandwidth-latency-tradeoff set to 10,000 bytes/round trip

As can be clearly seen in the graph above, the new implementation is in any case superior to the old one in terms of the number of round trips and consistently delivers better results than the old implementation.

Also, the optimisation of the parameters to generate the IBF with the new algorithm is less aggressive, this saves a lot of bandwidth on one side but on the other side it consumes a bit more RTTs. This deterioration is worth it in the vast majority of cases, because the bandwidth savings are correspondingly large.

What is surprising is that the new algorithm seems to be better in the middle range (430-450) despite the very lax IBF parameters regarding RTTs. This can be explained by the fact that in the old implementation the switch between full and differential synchronisation was forced too early and way too aggressive.

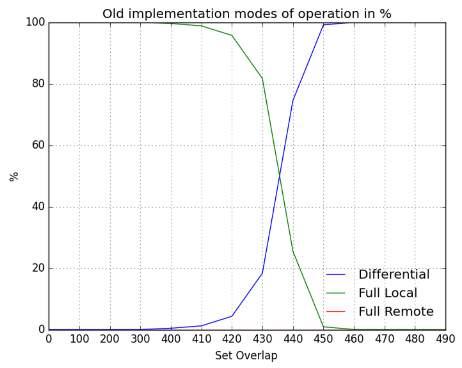


Figure 20: Old implementation probability for choosing between mode of operations by set difference.

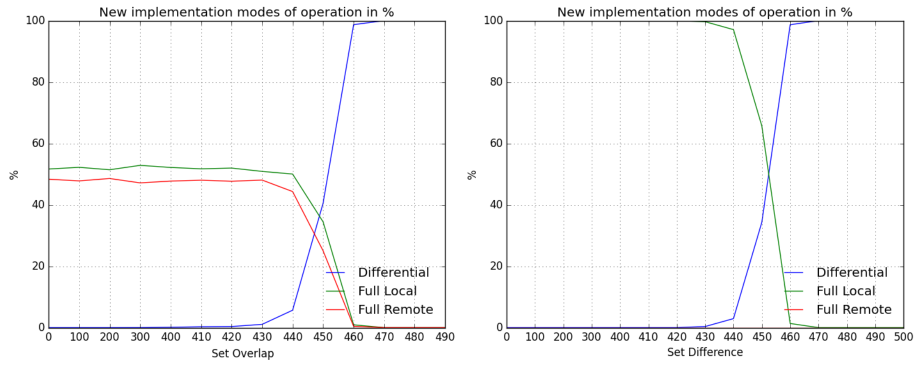


Figure 21: New implementation probability for choosing between mode of operations by set difference. Bandwidth-latency-tradeoff set to 0 bytes/round trip (left) and 10,000 bytes/round trip (right)

To support the assumption regarding the additional 0.25 round trips, the percentages ratio of the used mode of operations were also plotted in two graphs. In these graphs, as in the previous graphs, the set difference is shown on the X-axis and the percentage distribution is shown on the Y-axis. The blue line shows the percentage of differential synchronisations, the green line shows the percentage of full synchronisations, where the local elements were sent to the remote peer first and the red line shows the percentage, where the elements were requested by the remote peer.

This confirms the assumption that the additional 0.25 RTTs come from the two different full modes.

5 Security

In the security section of this document, the most important security improvements of the protocol are described and justified.

5.1 Attacker

Before considering how to make a protocol more secure, one must consider what the targets of an attacker might be. Only when these targets are known appropriate protective measures can be considered. The following targets were identified for this work:

- The attacker wants to consume maximum bandwidth.
- The attacker wants to consume maximum round trips.

In both cases the attacker basically wants to ensure that the synchronisation cannot be completed, but strings along the other party to maximize resources spent. This is an attack, as the other party may have limited resources and may miss deadlines or other more productive interactions as a result. Thus, in this work, the attacks in which the attacker tries to waste the peer's resources are to be made more difficult. Note that the attacker can always force the protocol to fail by simply stopping to participate or detectably not following the rules. Alas, in that case, the other peer can simply give up and will not waste its resources on the interaction with the attacker.

5.2 Validate Message Received in Correct Phases

To harden the protocol against attacks, controls were introduced in the improved implementation that check for each message whether the message was received in the correct phase. This is central so that an attacker finds as little attack surface as possible and makes it more difficult for the attacker to send the protocol into an endless loop, for example.

To achieve this, when a message is received, it is checked whether this message is allowed in the current phase. For this purpose, a function was written that functionally corresponds to the following pseudocode:

Algorithm 8 Algorithm determinating if a message is received in valid phase

```
Input:
allowed_phases: A list containing all phases in which
the message can be received
phase: The phase in which the protocol is in
Output:
Returns 0 if message is valid in phase and -1 if not
FUNCTION check_valid_phase (allowed_phases, phase)
    FOR allowed_phase IN allowed_phases DO
        IF allowed_phase == phase THEN
            RETURN 0
        END IF
    END FOR
    RETURN -1
END FUNCTION
```

To determine which message is allowed in which phase, the state diagram described in section 3.3 can be consulted.

5.3 Message Control Flow

Another key point to minimise the attack surface on the differential mode of operation of the protocol is to check that no message has been received twice, that no replies to messages are missing or that messages are received that do not follow the protocol message flow. The four possible checking conditions can be seen in the schematic diagram below:

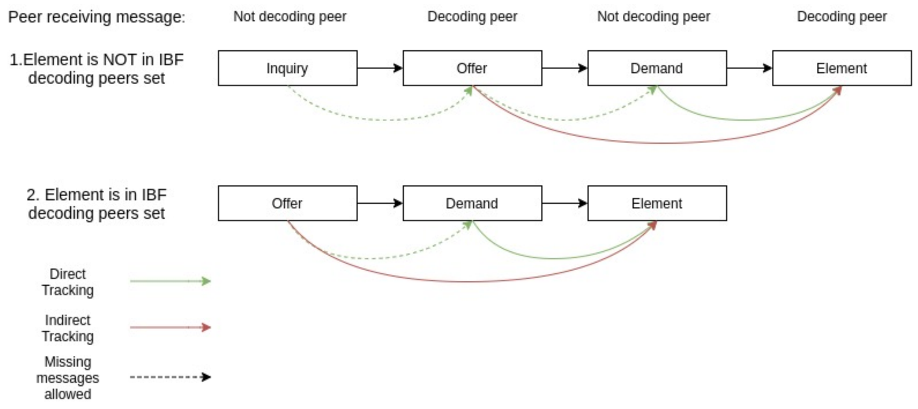


Figure 22: Message Flow Control schematic diagram

The following rules are checked by the Message Control Flow:

1. For each "Demand" message sent, exactly one "Element" message must be received. This applies because the remote peer has committed itself in advance with its "Offer" message that the offered element is in his set.
2. For each "Inquiry" message sent, a maximum of one "Offer" message shall normally be received. However, since the "Inquiry" message is an 8-byte hash, collisions may occur. But this should be rather rare. Nevertheless, it can not be assumed that collisions never occur and therefore not that an inquiry is unique.
3. For each "Offer" message sent, a maximum of one "Demand" message shall be received.
4. No "Element" message shall be received or requested for which no Offer (red arrows in the diagram) or Demand has been sent or received.

The chaining of the messages is realised for the Offer, Demand and Element messages via the hash. The hash is contained in the messages and is unique (Except the hash in the "Inquiry" message, because it is only 8-byte (64-bit) long and not like the hashes in the other messages, which are 64-byte (SHA-512[17]) long and therefore with cryptographic security unique). It is more complicated with the "Inquiry" message, as this only contains the IBF key of the requested message, but not the hash. With the "Inquiry" message, the requested IBF key must be stored and as soon as an Offer message is received, it must be determined which IBF key the Offer represents. The following pseudocode is used to check this chaining (next page).

Algorithm 9 Function to control the message flow

```
Available message states and numerical representation:

0 - MESSAGE_EMPTY: This message has never been sent or received
1 - MESSAGE_SENT: This message has been sent to other peer
2 - MESSAGE_EXPECTED: This message is expected to be received
                      from other peer
3 - MESSAGE_RECEIVED: This message has been received

Inputs:
hash_map : A hashmap containing for every processed hash an dictionary
with entries for all message types (mt) and the given state (new_state)
mt : Message type that needs to be updated. Eg. OFFER_MESSAGE
new_state : New message state to set the dictionary of the hash to
hash_code : The hashcode that needs to be processed
Outputs:
Returns -1 if action is not allowed and 0 if action is in
accordance with the protocol
FUNCTION message_control_flow (hash_map, mt, new_state, hash_code)
    cfe = hash_map.find(hash_code)
    IF (OFFER_MESSAGE == mt) THEN
        state = cfe.offer
    ELSE IF (DEMAND_MESSAGE == mt) THEN
        state = cfe.demand
    ELSE IF (ELEMENT_MESSAGE == mt) THEN
        state = cfe.element
        IF ((new_mcfs != MESSAGE_SENT) &&

            (MESSAGE_RECEIVED != cfe.offer)) THEN
            RETURN -1
        END IF
        IF ((new_mcfs != MESSAGE_SENT) &&
            (cfe.demand != MESSAGE_SENT)) THEN
            RETURN -1
        END IF
    ELSE
        RETURN -1
    IF new_state <= state THEN
        RETURN -1
    END IF
    hash_map.insertOrUpdate(hash_code,cfe)
    RETURN 0
END FUNCTION
```

5.4 Limit Active/Passive Switches in Differential Synchronisation

One of the most important security improvements of the protocol is the limitation of the maximum allowed active/passive switches during differential synchronisation. The original implementation did not have a limit. By limiting the maximum rounds, an attacker is strongly limited. This is central to the security of the protocol as an adversary could easily send an IBF that fails to decode, forcing the victim into an endless decoding cycle.

Since it is a question of probabilities how many active/passive switches are needed to match the sets, it is important to determine at the beginning how large the probability is in each round that the decoding of the IBFs fails and another round is needed. In order to determine this probability, ~10,000,000 simulations were carried out with the improved implementation and parameters found in this work.

Active/Passive switches	Number of experiments	share in %	share in % compared last round
Total	10099993	100%	-
0	7,849,529	78%	-
1	1,926,568	19%	24.5%
2	261,521	2.6%	13.6%
3	47,253	0.47%	18%
4	14,972	0.15%	31%
5	149	0.0015%	1%
6	1	-	<1%

Table 10: Measurement data for determinating average IBF-Decoding error rate

This table shows that the probability of an active/passive switch is 1%, 1%, 13%, 18%, 25%, 31% the average of these value is 14.83%. This is consistent with the measurements shown in the 4 section of this paper. It can be assumed that the probability of decoding each round is around 15%, i.e. the probability of failing to decode an IBF for n rounds can be approximated with the following formula:

$$probability = 0.15^n \tag{4}$$

Figure 23: Formula to calculate probability of failure in the n-th round

Through probability calculation, one can detect that one is interacting with an attacker by showing that the number of rounds of decoding failures is too improbable to happen in benign situations.

The number of rounds after which the operation can be aborted (and it can be assumed with a defined security level that the other peer is not performing the protocol correctly) can be calculated with the following formulae.

$$(0.15)^{numberOfRounds} = \frac{1}{2^{securityLevel}} \tag{5}$$

This can then be transformed to calculate the required rounds:

$$numberOfRounds = 0.3653681 \times securityLevel \tag{6}$$

Or transformed to determine the safety level for a given number of rounds:

$$securityLevel = 2.73697 \times numberOfRounds \tag{7}$$

Figure 24: Formulae to determinate level of confidence to detect malicious peer

From these formulae, the numbers of rounds for the given safety levels are shown in the table below:

Security Level in Bit	Rounds
0	0
10	4
20	8
30	11
80	30
128	47
256	94

Table 11: Table representing level of confidence to detect a malicious peer

The definition and rationale for the security level chosen for the implementation can be found in the section5.8.

5.5 Full Synchronisation Plausibility Check

If the differential synchronisation mode of operation has been limited by restricting the maximum number of active/passive switches, the full synchronisation mode remains open for an attack. Although this is less demanding in terms of CPU time, full synchronisation of large sets can place a significant load on the network.

An attacker can try to use up a peer's bandwidth by pretending that the peer needs full synchronisation even if the set difference is very small and the attacker only has a few (or even zero) elements that are not already synchronised.

In such a case, it would be ideal if the plausibility could already be checked during full synchronisation as to whether the other peer was honest or not with regard to the estimation of the set size difference and thus the choice of Mode of Operation.

In order to calculate this plausibility, the formula (9) was developed, which depicts the probability with which one can calculate the corresponding plausibility based on the number of new and repeated elements after each received element.

rs: Estimated remote set difference
lis: Local initial set size / Number of elements in set
rd: Received duplicates / Number of duplicated element received
rf: Received fresh / Number of fresh elements received

$$probability = (1 - \frac{rs}{lis + rs})^{rd - rf(\frac{lis}{rs})} \tag{8}$$

Figure 25: Formula to determinate if elements received are plausible

This formula is plausibilised with the following example calculation, for which "Estimated Remote Set Difference" 490 and for the "Locale Initial Set Size" : 5 was chosen.

Step	Duplicate received elements	Fresh received elements	Probability
1	0	0	1
2	0	10	1.6
3	1	98	1
4	2	150	0.11
5	3	290	0.82
6	5	490	1
attack	13	0	$1.13 \times 10^{25} \approx (\frac{1}{2^{80}})$

Table 12: Simulation of the plausibility check formula

In this calculation example, one can see in steps 1 and 6 that both at the beginning and at the conclusion of the operation, the "probability" is 1. This is correct because the values at this point are 100% plausible. This is also reflected in step 3, where the plausibility is also 1, since 98 is exactly 1/5 of 490. The remaining steps also behave as expected. They are above 1 if a disproportionately large number of new elements were received and below 1 if a disproportionately large number of duplicated elements were received. The attack example shows that an attack can already be detected after 13 received elements with a probability of two to the power of 80. One problem with this approach is that it relies on the elements being received in randomised order. This was not the case in the original implementation. For this reason, the existing implementation had to be modified so that the elements are now sent in randomised order. A corresponding security level was determined for the probabilistic approach. The discussion on this can be found in Section 5.8.

Besides this approach from probability theory, there is an additional check that can be made. After the entire set has been transferred to the other peer, no known elements may be returned by the second peer, since the second peer should only return the elements that are missing from the initial peer's set. These two approaches are given here in the pseudocode:

```
Algorithm 10 Function to verify the plausibility of full
syncronisation mode of operation

    Input:
    SECURITY_LEVEL: The security level used e.g. 2^80
    phase: The statemachine phase
    rs: Estimated Remote Set Difference
    lis: Number of elements in Set
    rd: Number of duplicated element received
    rf:  Number of fresh elements received
    Output:
    Returns 1 if full sync is plausible and -1 otherwise
    FUNCTION full_sync_plausibility_check (phase,rs,lis,rd,rf)
        security_level_lb = 1 / SECURITY_LEVEL
        IF (PHASE_FULL_SENDING == phase) THEN
            IF (rd > 0) THEN
                RETURN -1
            END IF
        END IF
        IF (PHASE_FULL_RECEIVING == phase) THEN
            IF (0 <= rs) THEN
                rs = 1
            END IF
            base = (1 - ( rs / (lis + rs)))
            exponent = (rd - (rf * (lis/rs)))
            value = POWER(base, exponent)
            IF ((value < security_level_lb) ||
                (value > SECURITY_LEVEL) THEN
                RETURN -1
            END IF
        END IF
        RETURN 1
    END FUNCTION
```

5.6 Validate Mode of Operation

Another weakness in the original protocol was that one peer alone can determine which mode of operation is selected for the operation. An attacker can take advantage of this by choosing a mode that is as unfavourable as possible. The other peer then has no way of checking whether this decision was correct.

To solve this problem, the algorithm described in section 4.5.3 and in the RFC in the Appendix B - RFC is executed on both sides with the same parameters. This ensures that an attacker can only choose a mode of operation that is compatible with the previously defined values (which are then used for all further plausibility checks).

This limits an attacker's ability to waste resources by influencing the mode of operation.

5.7 Byzantine Boundaries

Another way to restrict an attacker is to define an upper and lower bound, based on prior knowledge, for the number of elements that the attacker could have. These two values depend on the purpose of the application and should be passed by the application via the API interface.

The lower byzantine bound can be, for example, the number of elements the other peer had in his set at the last contact. It is unusual in many cases for the set to become smaller, at least within a short time.

The upper byzantine barrier can be a practical maximum e.g. the number of e-voting votes; in Switzerland, no more than 5.5 million votes can ever be counted (2020) [4] A time-dependent value is also possible, e.g. a maximum of 500 elements are added every day, in which case the maximum would be $ByzantineUpperBound = DaysSinceLastSync \times 500 + SetSizeLastSync$.

Due to these two barriers, several checks can be implemented:

- The number of elements that the remote peer declares to have in his set must never be smaller than the lower byzantine bound.
- The estimated local set size difference together with the local set size must never be higher than the upper byzantine bound.
- The estimated remote set size difference together with the remote set size must also never be higher than the upper byzantine bound.

One problem is that the estimates of the set size difference can legitimately be higher than the real value (in the case of an incorrect estimate). In this case, the upper Byzantine limit may be exceeded, although in reality this limit was never exceeded. To prevent this, the implementation has been adapted so that it is not possible for the remote/local set sizes together with the remote/local estimated set size difference to exceed the upper Byzantine bound.

[4] `https://www.bfs.admin.ch/bfs/de/home/statistiken/politik/abstimmungen/stimmbeteiligung.html`

The three checks described were implemented as follows:

```
Algorithm 11 Function to check byzantine boundaries
    Input:
    rec: Number of elements in remote set
    rsd: Number of elements differ in remote set
    lec: Number of elements in local set
    lsd: Number of elements differ in local set
    UPPER_BOUND: Given byzantine upper bound
    LOWER_BOUND: Given byzantine lower bound
    Output:
    returns 1 if parameters in byzantine bounds otherwise
    returns -1
    FUNCTION check_byzantine_bounds (rec,rsd,lec,lsd)
        IF (rec + rsd > UPPER_BOUND) THEN
            RETURN -1
        END IF
        IF (lec + lsd > UPPER_BOUND) THEN
            RETURN -1
        END IF
        IF (rec < LOWER_BOUND) THEN
            RETURN -1
        END IF
        RETURN 1
    END FUNCTION
```

5.8 Security Level

An important parameter in the probability-based security algorithms described above is the security level measured in bits:

> “Security Bits estimate the computational steps or operations (not machine instructions) required to find a solution to the problem in the problem’s domain (FF, IF, or EC). For example, if someone says, ’My system uses 1024 Diffie Hellman’, they are really stating their system has a security level of 80 bits (and because its Diffie Hellman, the problem domain is finite field). It will take a computer, on average, approximately 2^80 operations to find a solution (think Big-Oh notation). To break Diffie-Hellman via classical discrete logarithms, a number of methods could be employed: Index calculus, modified Pollard’s rho, or Baby-step giant-step to name a few.”[18]

The minimum security level required to achieve cryptographic security is shifting as computing power becomes cheaper and cheaper. NIST currently recommends a minimum security level (by 2030) of 112-bits.[19].

In contrast to the above-mentioned definition of security, our definition is not about cryptographic security, but about the probability of a user who adheres to the protocol being falsely identified as an attacker, i.e. the false positive rate of the probabilistic algorithms.

For practically all applications, a security level of 1/2^80 should be sufficient, even if this is no longer sufficient for cryptographic applications. The probability here is 1 / 1208925819614629174706176 (2^80). For comparison, 2^80 is 17 times higher than the number of stars in the universe estimated by astronomers. Which is estimated nearly 70 quadrillions[20]. Thus, a security level of 80 will be used to flag interactions with other peers as likely malicious in our implementation.

If this level of security is not sufficient for an application, it can be adapted very easily.

5.9 Results

The security improvements aim to limit the possibilities of an attacker to deviate from the protocol as much as possible, and the false positive dedection rate is as low as possible. Attacks cannot be prevented completely, but they can be detected as early and as reliably as possible. It was not the goal to completely prevent attacks, which is very difficult in a probability-based protocol either way.

The improvements made effectively limit an attack to a minimum, because it is no longer possible for an attacker to use up either infinite rounds in differential synchronisation by manipulating IBFs or to manipulate the protocol by sending false messages in the wrong phases or to trigger a full synchronisation, even though the attacker does not have enough new elements.

6 Conclusion

6.1 Summary

The aim of this work was to improve the existing implementation in terms of security and performance and to document it in an RFC.

Important performance improvements achieved through this work are, for example, massive bandwidth savings in set reconciliation of up to 42%. In the adapted implementation, it is possible to define a "bandwidth-latency-tradeoff" depending on the application area, which makes it possible to define a trade-off between latency and bandwidth. The newly developed algorithm, which determines the mode of operation based on various factors, has made it possible to save bandwidth and round trips for many scenarios.

Importantly, the new implementation limits an attacker more. The attack resistance has been improved through various checks, validations and probability-based algorithms. An attacker who does not adhere to the protocol can thus be quickly recognised and excluded in many cases.

The protocol was improved in many places. Some bugs were found and fixed. For example, the old implementation would not have worked at all for larger sets with small set differences, as the counter of the IBF would have overflowed. To prevent the counter from overflowing, an algorithm was developed that makes the counter as compact as possible when it is transmitted over the network.

Another important part of the work was documenting the protocol as an RFC. This was achieved through the work, the RFC can be found in the Appendix B - RFC.

6.2 Addressees of the Improvement

The work is primarily intended to contribute to the free software project GNUnet. The improvements brought in should especially support GNUnet in establishing GNS[5] as an alternative to DNS.

The RFC created should allow alternative implementations to be created for the Secure Set Reconciliation protocol used at GNS for distributing revocations that are compatible with the existing implementation.

The improvements and the documentation should help the Secure Set Reconciliation Protocol to be extended and used in other application areas.

[5]GNU Name System

6.3 Future Work

Towards the end of the work, time was running out, not all ideas could be implemented. There are possibilities to continue that could be addressed in a future work:

- An improvement that would save round trips and bandwidth, especially for very small set differences (the set difference being very small is a very common case when the sets are often matched). The improvement would be: Instead of using the elements contained in the IBFs from the Strata Estimators, only to calculate the set difference. One could request the decoded elements directly from the remote peer or deliver them to the remote peer, depending on which set they are missing. This would minimise the set difference already before the IBF is created and thus reduce the probability that the decoding fails. If the set difference is very small and therefore the stratum 0 of the Strata Estimator is decoded, the rest of the protocol could be omitted completely and the sets would be synchronised between the peers. This improvement has a great potential to have high additional savings.
- A possible future extension could be to transfer the elements of a very small set to the other peer directly after the "Operation Request". This would save bandwidth, round trips and computing time if transmitting the full set is close in size to the Strata Estimator.
- The implementation for compressing the IBF counter works with 1-byte words, this could be rewritten to work with 8-byte (64-bit) words to increase efficiency.
- In the existing implementation, a performance improvement was achieved by making the size of the IBFs always odd. In a future work, it could be investigated how the use of prime numbers would affect the performance. It is conceivable that by using prime numbers, the failure rate during decoding could be further reduced.
- Another way to improve the protocol could be to add a SHA-512 XOR sum of the reconciled set to the "Done" and "Full done" messages . This could increase the security that the sets are identical after completion of the protocol. Whether this is necessary and whether the additional bandwidth required is worthwhile must be carefully examined.
- It would also be useful to re-perform the measurements made in this thesis with asymmetric set sizes and larger elements in the set, to validate that the assumptions made in this thesis are also valid for other set constellations.
- To introduce a check that ensures, that in case of multiple IBF messages the offset sent in the IBF-Message is monotonic increasing and a multiple of the maximal buckets per element.

- When receiving an IBF-Last message, three additional plausibiliy checks could be introduced: One is to ensure that after each active/passive switch the IBF can never be more than double in size. Another plausibility check is that an IBF probably never will be larger than the byzantine upperbound multiplied by two. The third plausibility check is to take successfully decoded IBF keys (received offers and demands) into account and to validate the size of the received IBF with the in Appendix B - RFC in the "Operation Mode" section defined function "get_next_ibf_size()".
- When receiving an IBF message, a sanity check can be introduced to ensure that the "OFFSET" message field is never higher than the "IBF SIZE" field in the IBF message.

7 Indices and References

List of Figures

List of Tables

List of Algorithms

References

[1] F. Dold and C. Grothoff, "Byzantine set-union consensus using efficient set reconciliation," vol. 2017, no. 1, p. 14. [Online]. Available: https://doi.org/10.1186/s13635-017-0066-3 1, 2

[2] F. Dold, "The GNU taler system: practical and provably secure electronic payments. (le systeme GNU taler: Paiements electroniques pratiques et securises)." 1

[3] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? efficient set reconciliation without prior context," in *Proceedings of the ACM SIGCOMM 2011 conference*, ser. SIGCOMM '11. Association for Computing Machinery, pp. 218–229. [Online]. Available: https://doi.org/10.1145/2018436.2018462 1, 2, 4.5.2

[4] gnunet.git - GNUnet core repository. [Online]. Available: https://git.gnunet.org/gnunet.git/ 1, 3.2, 3, 4.5.3

[5] M. Wachs, M. Schanzenbach, and C. Grothoff, "A censorship-resistant, privacy-enhancing and fully decentralized name system," in *Cryptology and Network Security*, ser. Lecture Notes in Computer Science, D. Gritzalis, A. Kiayias, and I. Askoxylakis, Eds. Springer International Publishing, pp. 127–142. 1

[6] C. Grothoff, B. Polot, and C. Loesch, *The Internet is Broken: Idealistic Ideas for Building a NEWGNU Network*. 2

[7] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," vol. 60, pp. 630–659. 2

[8] M. Mitzenmacher and T. Morgan, "Robust set reconciliation via locality sensitive hashing." [Online]. Available: http://arxiv.org/abs/1807.09694 2

[9] M. Mitzenmacher and R. Pagh, "Simple multi-party set reconciliation." [Online]. Available: http://arxiv.org/abs/1311.2037 2

[10] What are invertible bloom lookup tables? | dash news. [Online]. Available: https://dashnews.org/what-are-invertible-bloom-lookup-tables/ 2

[11] M. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," pp. 792–799. 2

[12] D. P. Kroese and R. Y. Rubinstein, "Monte carlo methods," vol. 4, no. 1, pp. 48–58, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/wics.194. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/wics.194 3.4.1

[13] Group theory - lagrange's theorem. [Online]. Available: https://crypto.stanford.edu/pbc/notes/group/lagrange.html 3.4.1, 3.4.2

[14] Entfernung bern > frankfurt - luftlinie, fahrstrecke, mittelpunkt. [Online]. Available: https://www.luftlinie.org/Bern/Frankfurt 4.5.2

[15] What's "normal" for latency and packet loss? [Online]. Available: https://www.pingman.com/kb/42 4.5.2

[16] S. Parthier. IT studien und analysen. [Online]. Available: https://www.it-daily.net/analysen/16102-internet-geschwindigkeit-weltweit-deutschland-auf-platz-25 4.5.2

[17] S. Gueron, S. Johnson, and J. Walker, "SHA-512/256," in *2011 Eighth International Conference on Information Technology: New Generations*, pp. 354–358. 5.3

[18] Security level - crypto++ wiki. [Online]. Available: https://www.cryptopp.com/wiki/Security_Level 5.8

[19] P. B. B. OBE. NIST recommendation for key management. [Online]. Available: https://billatnapier.medium.com/nist-recommendation-for-key-management-9cb53fd72f6e 5.8

[20] How many stars are there in the universe? [Online]. Available: https://skyandtelescope.org/astronomy-resources/how-many-stars-are-there/ 5.8

8 Declaration of Authorship

I confirm with my signature that I have carried out my Bachelor Thesis independently. All sources of information (technical literature, discussions with experts, etc.) that have contributed significantly to my work are listed in my work report.
First name and last name: Elias Summermatter
Date: 17.06.2021
Signature: ______________________________

9 Appendix A - Project Management

The following tools were used for this project :

- For project planning the project management tool Jira.[6]
- For code version management the GNUnet git [7] repository.
- For the management, versioning and backup of the thesis deliverables, the simulation raw data and the simulation evaluation scripts a public GitLab[8] repository.
- For discovered improvements not yet implemented, the GNUnet bug tracker[9] (bugs discovered during the course of the work and not implemented in this work are additionally listed in the Future Work section).

The task can be divided into six main tasks:

1. Creation of the RFC
2. Determining the initial situation
3. Finding possible performance improvements
4. Finding possible security improvements
5. Implementing the found performance improvements
6. Implement the found security improvements

These six tasks were recorded as milestones in Jira and scheduled as shown in the time line section. For every milestone various subtasks have been initially defined and during the project newly defined subtask have been added. Details about the subtasks can be found in the Jira project management tool.

In addition, there are the tasks that are required for the formal thesis but are not directly related to the task.

- Documentation of the thesis
- Code improvements
- Video
- Poster for the exhibition
- TechDay presentation
- Bachelor defense

[6] `https://byzantine-fault-tolerant-set-reconciliation.atlassian.net`
[7] `https://git.gnunet.org/gnunet.git`
[8] `https://gitlab.seccom.ch/elias/Byzantine-Fault-Tolerant-Set-Reconciliation`
[9] `https://bugs.gnunet.org`

9.1 Risk Analysis

The risk analysis should determine which are the biggest risks for the project and which countermeasures can be taken against these risks. The risks are always to be seen in the light of the fact that the Bachelor Thesis could be judged as insufficient

ID	Title	Description of the risk	Risk mitigation
1	Scope	The biggest risk identified for this work is the size and scope of the work. The work was originally planned as work for a two person team. Unfortunately, since no second person could be found to do the work, it was decided to do the project as a solo effort.	If the project cannot be completed in full, it is important that partial results are documented in such a way that it is also possible to submit partial results.
2	Technical knowledge	At the beginning of this project there was very little experience in C programming and no knowledge of GNUnet.	This risk could be mitigated by the author having studied both the GNUnet project and C programming in detail before starting the project.
3	Complex subject matter	The subject involves mathematical complexities that need to be understood for a thorough analysis. Since the mathematical background in computer science studies is not as well-founded as in mathematics studies, this can become a challenge.	To minimise this risk, it will be tried to avoid a complex mathematical approach wherever possible and reasonable.

Table 13: Risk evaluation table

At the end of the project it can be stated that the risks identified at the beginning of the project corresponded well with the problems that were identified. The risk reductions have prevented insurmountable problems from occurring. Especially risks 2 and 3 were problematic at the beginning, as they slowed down progress at times. In the end, a good balance was found and the project was successfully completed. Risk 1 was largely cushioned by an increased workload. Significantly more hours were invested than planned.

9.2 Time-Line

At the beginning of the project, a corresponding timeline was created, which contains the appropriate epics to plan the project that the progress can be traced and, in case of deviations, the appropriate measures can be taken.

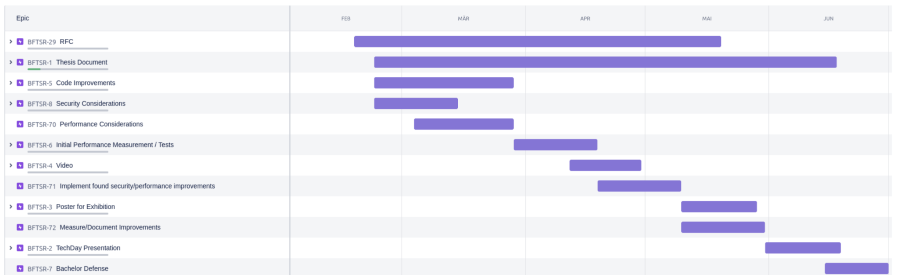


Figure 26: Initial time line

This is the initial schedule that was created at the beginning of the project. Care was taken to ensure that the parts that are interdependent were scheduled one after the other.

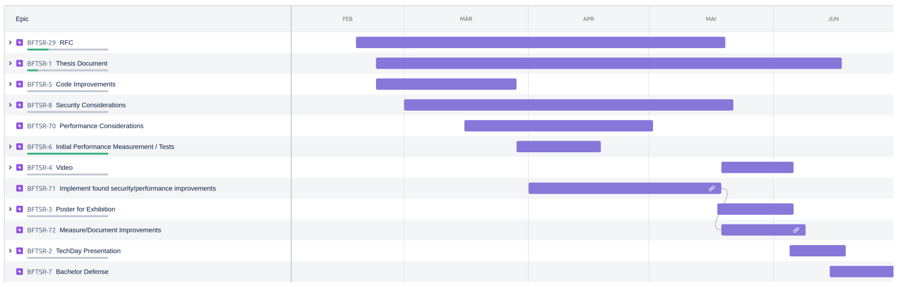
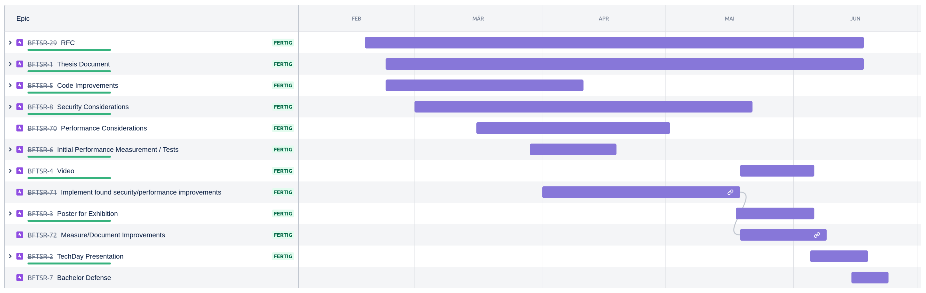


Figure 27: Adapted timeline

The schedule had to be adjusted on the 08.04.2021, because the initial schedule assumed that it is possible to determine the performance mathematically. When it was defined that the performance would be determined by simulations, the schedule had to be adjusted.

Another change was that originally the "performance considerations" were scheduled after the "security considerations". It turned out that the security considerations were largely based on the findings of the performance considerations. For this reason, the security considerations were extended to include the performance considerations.

It would have been difficult to create the poster and the video before the results of the work were available. Therefore, this work has been pushed back.



Epic	
~~BFTSR-29~~ RFC	FERTIG
~~BFTSR-1~~ Thesis Document	FERTIG
~~BFTSR-5~~ Code Improvements	FERTIG
~~BFTSR-8~~ Security Considerations	FERTIG
~~BFTSR-70~~ Performance Considerations	FERTIG
~~BFTSR-6~~ Initial Performance Measurement / Tests	FERTIG
~~BFTSR-4~~ Video	FERTIG
~~BFTSR-71~~ Implement found security/performance improvements	FERTIG
~~BFTSR-3~~ Poster for Exhibition	FERTIG
~~BFTSR-72~~ Measure/Document Improvements	FERTIG
~~BFTSR-2~~ TechDay Presentation	FERTIG
BFTSR-7 Bachelor Defense	

Figure 28: Effective timeline

As can be seen on the effective schedule, I often worked on different tasks at the same time, as they were interdependent and difficult to separate.

The creation of the RFC took more time than planned, some changes had to be made after the scheduled time window.

The code improvements took slightly more time than planned.

The adapted timeline could be largely adhered to. As can be seen from the effective timeline, all tasks could be completed by the submission of this document. Only the epic "Bachelor Defense" is still open, because the Bachelor Defense will take place after the submission of this document.

9.3 Time Management

The thesis was written over a period of 22 weeks. In total I worked about 650 hours on the thesis. A lot of time was spent learning C programming and running and evaluating well over a hundred simulations. The special form, how a RFC has to be created, cost me some effort. The execution of the various tasks was planned in working days, not hours, see Jira roadmaps above.

9.4 Communication/Meetings

Fixed meetings took place weekly on Thursdays at 16.30h with Professor. Dr. Christian Grothoff. These regular meetings were held remotely via the GNUnet Mumble[10] server due to the Covid-19 pandemic. Outside of these regular meetings we communicated via mail and phone.

Two online meetings with Mr. Han van der Kleij (expert) and Professor. Dr. Christian Grothoff took place. (04.March 2021 and 29.April 2021).

The date for the bachelor defense was set on April 26th, June 23rd and it was decided that it should take place live.

[10]gnunet.org

9.5 Decisions

This section will justify and document the key decisions made during the project.

9.5.1 Performance Analysis: Mathematical or through Simulations

At the beginning of the performance analysis, the question arose whether the performance analysis should be created by simulations or by mathematical derivation.

It quickly became apparent that a mathematical analysis would be very complicated, as the reasoning would have involved a large number of variable parameters and such a model would have become very complex. Therefore, it was decided to rely on "approximations" and thus on simulations. In addition, a server with sufficient computing power was available for this work. For the required simulations, 100'000de of CPU hours could be spent on simulations.

9.5.2 IBF Factor Static or Variable

At the beginning of the project, it was assumed that the IBF factor must be defined variably and determined on the basis of the set differences, set sizes. During the experiments it was found that a static IBF factor is sufficient. For details see the performance section of this document.

9.5.3 Improvements

Since the work was very extensive and time was running out towards the end of the work, it was necessary to decide which improvements should be saved for future work and which improvements should be implemented in this work. The deferred improvements are listed in Future Work.

9.6 Conclusion

All tasks were completed as planned. In the last four weeks, I have worked many hours, including weekends, to achieve the defined goals.

10 Appendix B - RFC

Workgroup:	Independent Stream	
Internet-Draft:	draft-summermatter-set-union-01	
Published:	16 June 2021	
Intended Status:	Informational	
Expires:	18 December 2021	
Authors:	E. Summermatter	C. Grothoff
	Seccom GmbH	*Berner Fachhochschule*

Byzantine Fault Tolerant Set Reconciliation

Abstract

This document contains a protocol specification for Byzantine fault-tolerant Set Reconciliation.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes a byzantine fault tolerant set reconciliation protocol used to efficient and securely compute the union of two sets across a network.

This byzantine fault tolerant set reconciliation protocol can be used in a variety of applications. Our primary envisioned application domain is the distribution of revocation messages in the GNU Name System (GNS) [GNS]. In GNS, key revocation messages are usually flooded across the peer-to-peer overlay network to all connected peers whenever a key is revoked. However, as peers may be offline or the network might have been partitioned, there is a need to reconcile revocation lists whenever network partitions are healed or peers go online. The GNU Name System uses the protocol described in this specification to efficiently distribute revocation messages whenever network partitions are healed. Another application domain for the protocol described in this specification are Byzantine fault-tolerant bulletin boards, like those required in some secure multiparty computations. A well-known example for secure multiparty computations are various E-voting protocols [CryptographicallySecureVoting] which use a bulletin board to share the votes and intermediate computational results. We note that for such systems, the set reconciliation protocol is merely a component of a multiparty consensus protocol, such as the one described in Dold's "Byzantine set-union consensus using efficient set reconciliation" [ByzantineSetUnionConsensusUsingEfficientSetReconciliation].

The protocol described in this report is generic and suitable for a wide range of applications. As a result, the internal structure of the elements in the sets MUST be defined and verified by the application using the protocol. This document thus does not cover the element structure, except for imposing a limit on the maximum size of an element.

The protocol faces an inherent trade-off between minimizing the number of network round-trips and the number of bytes sent over the network. Thus, for the protocol to choose the right parameters for a given situation, applications using an implementation of the protocol SHOULD provide a parameter that specifies the cost-ratio of round-trips vs. bandwidth usage. Given this trade-off factor, an implementation CAN then choose parameters that minimize total execution cost. In particular, there is one major choice to be made, namely between sending the complete set of elements, or computing the set differences and transmitting only the elements in the set differences. In the latter case, our design is basically a concrete implementation of a proposal by Eppstein.[Eppstein]

We say that our set reconciliation protocol is Byzantine fault-tolerant because it provides cryptographic and probabilistic methods to discover if the other peer is dishonest or misbehaving. Here, the security objective is to limit resources wasted on malicious actors. Malicious actors could send malformed messages, including malformed set elements, claim to have much larger numbers of valid set elements than they actually hold, or request the retransmission of elements that they have already received in previous interactions. Bounding resources consumed by malicous actors is important to ensure that higher-level protocols can use set reconciliation and still meet their resource targets. This can be particularly critical in multi-round synchronous consensus protocols where peers that cannot answer in a timely fashion would have to be treated as failed or malicious.

To defend against some of these attacks, applications SHOULD remember the number of elements previously shared with a peer, and SHOULD provide a way to check that elements are well-formed. Applications MAY also provide an upper bound on the total number of valid elements that exist. For example, in E-voting, the number of eligible voters MAY be used to provide such an upper bound.

A first draft of this RFC is part of Elias Summermatter's bachelor thesis. Many of the algorithms and parameters documented in this RFC are derived from experiments detailed in this thesis. [byzantine_fault_tolerant_set_reconciliation]

This document defines the normative wire format of resource records, resolution processes, cryptographic routines and security considerations for use by implementors. SETU requires a bidirectional secure communication channel between the two parties. Specification of the communication channel is out of scope of this document.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Background

2.1. Bloom Filter

A Bloom filter (BF) is a space-efficient probabilistic datastructure to test if an element is part of a set of elements. Elements are identified by an element ID. Since a BF is a probabilistic datastructure, it is possible to have false-positives: when asked if an element is in the set, the answer from a BF is either "no" or "maybe".

A BF consists of L buckets. Every bucket is a binary value that can be either 0 or 1. All buckets are initialized to 0. A mapping function M is used to map each ID of each element from the set to a subset of k buckets. In the original proposal by Bloom, M is non-injective and can thus map the same element multiple times to the same bucket. The type of the mapping function can thus be described by the following mathematical notation:

```
              ------------------------------------
              # M: E->B^k
              ------------------------------------
              # L = Number of buckets
              # B = 0,1,2,3,4,...L-1 (the buckets)
              # k = Number of buckets per element
              # E = Set of elements
              ------------------------------------
              Example: L=256, k=3
              M('element-data') = {4,6,255}
```

Figure 1

A typical mapping function is constructed by hashing the element, for example using the well-known Section 2 of HKDF construction [RFC5869].

To add an element to the BF, the corresponding buckets under the map M are set to 1. To check if an element may be in the set, one tests if all buckets under the map M are set to 1.

In the BF the buckets are set to 1 if the corresponding bit in the bitstream is 1. If there is a collision and a bucket is already set to 1, the bucket stays at 1.

In the following example the element e0 with M(e0) = {1,3} has been added:

```
                  bucket-0     bucket-1       bucket-2      bucket-3
              +-------------+-------------+-------------+-------------+
              |      0      |      1      |      0      |      1      |
              +-------------+-------------+-------------+-------------+
```

Figure 2

It is easy to see that an element e1 with M(e1) = {0,3} could have been added to the BF below, while an element e2 with M(e2) = {0,2} cannot be in the set represented by the BF below:

```
                bucket-0     bucket-1       bucket-2      bucket-3
            +-------------+-------------+-------------+-------------+
            |      1      |      0      |      0      |      1      |
            +-------------+-------------+-------------+-------------+
```

Figure 3

The parameters L and k depend on the set size and MUST be chosen carefully to ensure that the BF does not return too many false-positives.

It is not possible to remove an element from the BF because buckets can only be set to 1 or 0. Hence it is impossible to differentiate between buckets containing one or more elements. To remove elements from the BF a Counting Bloom Filter is required.

2.2. Counting Bloom Filter

A Counting Bloom Filter (CBF) is a variation on the idea of a Bloom Filter. With a CBF, buckets are unsigned numbers instead of binary values. This allows the removal of an element from the CBF.

Adding an element to the CBF is similar to the adding operation of the BF. However, instead of setting the buckets to 1 the numeric value stored in the bucket is increased by 1. For example, if two colliding elements M(e1) = {1,3} and M(e2) = {0,3} are added to the CBF, bucket 0 and 1 are set to 1 and bucket 3 (the colliding bucket) is set to 2:

```
                bucket-0     bucket-1       bucket-2      bucket-3
            +-------------+-------------+-------------+-------------+
            |      1      |      1      |      0      |      2      |
            +-------------+-------------+-------------+-------------+
```

Figure 4

The counter stored in the bucket is also called the order of the bucket.

To remove an element form the CBF the counters of all buckets the element is mapped to are decreased by 1.

For example, removing M(e2) = {1,3} from the CBF above results in:

```
              bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
          |      1      |      0      |      0      |      1      |
          +-------------+-------------+-------------+-------------+
```

Figure 5

In practice, the number of bits available for the counters is often finite. For example, given a 4-bit counter, a CBF bucket would overflow 16 elements are mapped to the same bucket. To handle this case, the maximum value (15 in our example) is considered to represent "infinity". Once the order of a bucket reaches "infinity", it is no longer incremented or decremented.

The parameters L and k and the number of bits allocated to the counters SHOULD depend on the set size. A CBF will degenerate when subjected to insert and remove iterations of different elements, and eventually all buckets will reach "infinity". The speed of the degradation will depend on the choice of L and k in relation to the number of elements stored in the IBF.

3. Invertible Bloom Filter

An Invertible Bloom Filter (IBF) is a further extension of the Counting Bloom Filter. An IBF extends the Counting Bloom Filter with two more operations: decode and set difference. This two extra operations are key to efficiently obtain small differences between large sets.

3.1. Structure

An IBF consists of an injective mapping function M mapping elements to k out of L buckets. Each of the L buckets stores a signed COUNTER, an IDSUM and an XHASH. An IDSUM is the XOR of various element IDs. An XHASH is the XOR of various hash values. As before, the values used for k, L and the number of bits used for the signed counter and the XHASH depend on the set size and various other trade-offs.

If the IBF size is too small or the mapping function does not spread out the elements uniformly, the signed counter can overflow or underflow. As with the CBF, the "maximum" value is thus used to represent "infinite". As there is no need to distinguish between overflow and underflow, the most canonical representation of "infinite" would be the minimum value of the counter in the canonical 2-complement interpretation. For example, given a 4-bit counter a value of -8 would be used to represent "infinity".

```
                 bucket-0     bucket-1       bucket-2      bucket-3
            +-------------+-------------+-------------+-------------+-------
      count |   COUNTER   |   COUNTER   |   COUNTER   |   COUNTER   |  C...
            +-------------+-------------+-------------+-------------+------
      idSum |    IDSUM    |    IDSUM    |    IDSUM    |     IDSUM   |  I...
            +-------------+-------------+-------------+-------------+------
    hashSum |   HASHSUM   |   HASHSUM   |   HASHSUM   |    HASHSUM  |  H..
            +-------------+-------------+-------------+-------------+-------
```

Figure 6

3.2. Salted Element ID Calculation

IBFs are a probabilistic data structure, hence it can be necessary to recompute the IBF in case operations fail, and then try again. The recomputed IBF would ideally be statistically independent of the failed IBF. This is achieved by introducing an IBF-salt. Given that with benign peers failures should be rare, and that we need to be able to "invert" the application of the IBF-salt to the element IDs, we use an unsigned 32 bit non-random IBF-salt value of which the lowest 6 bits will be used to rotate bits in the element ID computation.

64-bit element IDs are generated from a Section 2 of HKDF construction [RFC5869] with HMAC-SHA512 as XTR and HMAC-SHA256 as PRF with a 16-bit KDF-salt set to a unsigned 16-bit representation of zero. The output of the KDF is then truncated to 64-bit. Finally, salting is done by calculating the IBF-salt modulo 64 (effectively using only the lowest 6-bits of the IBF-salt) and doing a bitwise right rotation of the output of KDF. We note that this operation was chosen as it is easily inverted, allowing applications to easily derive element IDs with one IBF-salt value from element IDs generated with a different IBF-salt value.

In case the IBF does not decode, the IBF-salt can be changed to compute different element IDs, which will (likely) be mapped to different buckets, likely allowing the IBF to decode in a subsequent iteration.

```
# INPUTS:
# key: Pre calculated and truncated key from id_calculation function
# ibf_salt: Salt of the IBF
# OUTPUT:
# value: salted key
FUNCTION salt_key(key,ibf_salt):
  s = (ibf_salt * 7) modulo 64;
  /* rotate key */
  return (key >> s) | (key << (64 - s))
END FUNCTION


# INPUTS:
# element: element for which we are to calculate the element ID
# ibf_salt: Salt of the IBF
# OUTPUT:
# value: the ID of the element
FUNCTION id_calculation (element,ibf_salt):
    kdf_salt = 0 // 16 bits
    XTR=HMAC-SHA256
    PRF=HMAC-SHA256
    key = HKDF(XTR, PRF, kdf_salt, element) modulo 2^64
    return salt_key(key, ibf_salt)
END FUNCTION
```

Figure 7

3.3. HASH calculation

The HASH of an element ID is computed by calculating the CRC32 checksum of the 64-bit ID value, which returns a 32-bit value.CRC32 is well-known and described in Section 4.1 of the RFC [RFC3385].

3.4. Mapping Function

The mapping function M decides which buckets a given ID is mapped to. For an IBF, it is beneficial to use an injective mapping function M.

The first index is simply the CRC32 of the ID modulo the IBF size. The second index is calculated by creating a new 64-bit value by shifting the previous 32-bit value left and setting the lower 32 bits to the number of indices already processed. From the resulting 64-bit value, another CRC32 checksum is computed. The subsequent index is the modulo of this CRC32 output. The process is repeated until the desired number of indices is generated. In the case the process computes the same index twice, which would mean this bucket could not get pure again, the second hit is just skipped and the next iteration is used instead, creating an injective mapping function.

```
# INPUTS:
# key: the ID of the element calculated
# k: numbers of buckets per element
# L: total number of buckets in the IBF
# OUTPUT:
# dst: Array with k bucket IDs
FUNCTION get_bucket_id (key, k, L)
  bucket = CRC32(key)
  i = 0 // unsigned 32-bit index
  filled = 0
  WHILE filled < k DO

    element_already_in_bucket = false
    j = 0
    WHILE j < filled DO
      IF dst[j] == bucket modulo L THEN
        element_already_in_bucket = true
      END IF
      j++
    END WHILE

    IF !element_already_in_bucket THEN
        dst[filled] = bucket modulo L
        filled = filled + 1
    END IF

    x = (bucket << 32) | i // 64 bit result
    bucket = CRC32(x)
    i = i + 1
  END WHILE
  return dst
END FUNCTION
```

Figure 8

3.5. Operations

When an IBF is created, all counters and IDSUM and HASHSUM values of all buckets are initialized to zero.

3.5.1. Insert Element

To add an element to an IBF, the element is mapped to a subset of k buckets using the injective mapping function M as described in section Mapping Function. For the buckets selected by the mapping function, the counter is increased by one and the IDSUM field is set to the XOR of the element ID computed as described in section Salted Element ID Calculation and the previously stored IDSUM. Furthermore, the HASHSUM is set to the XOR of the previously stored HASHSUM and the hash of the element ID computed as described in section HASH calculation.

In the following example, the insert operation is illustrated using an element with the ID 0x0102 mapped to {1,3} with a hash of 0x4242, and a second element with the ID 0x0304 mapped to {0,1} and a hash of 0x0101.

Empty IBF:

```
            bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
   count  |      0      |      0      |      0      |      0      |
          +-------------+-------------+-------------+-------------+
   idSum  |    0x0000   |    0x0000   |    0x0000   |    0x0000   |
          +-------------+-------------+-------------+-------------+
 hashSum  |    0x0000   |    0x0000   |    0x0000   |    0x0000   |
          +-------------+-------------+-------------+-------------+
```

Figure 9

Insert first element with ID 0x0102 and hash 0x4242 into {1,3}:

```
            bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
   count  |      0      |      1      |      0      |      1      |
          +-------------+-------------+-------------+-------------+
   idSum  |    0x0000   |   0x0102    |    0x0000   |   0x0102    |
          +-------------+-------------+-------------+-------------+
 hashSum  |    0x0000   |   0x4242    |    0x0000   |   0x4242    |
          +-------------+-------------+-------------+-------------+
```

Figure 10

Insert second element with ID 0x0304 and hash 0101 into {0,1}:

```
            bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
   count  |      1      |      2      |      0      |      1      |
          +-------------+-------------+-------------+-------------+
   idSum  |    0x0304   |   0x0206    |    0x0000   |   0x0102    |
          +-------------+-------------+-------------+-------------+
 hashSum  |    0x0101   |   0x4343    |    0x0000   |   0x4242    |
          +-------------+-------------+-------------+-------------+
```

Figure 11

3.5.2. Remove Element

To remove an element from the IBF the element is again mapped to a subset of the buckets using M. Then all the counters of the buckets selected by M are reduced by one, the IDSUM is replaced by the XOR of the old IDSUM and the ID of the element being removed, and the HASHSUM is similarly replaced with the XOR of the old HASHSUM and the hash of the ID.

In the following example the remove operation is illustrated.

IBF with two encoded elements:

```
              bucket-0     bucket-1       bucket-2      bucket-3
            +-------------+-------------+-------------+-------------+
      count |      1      |      2      |      0      |      1      |
            +-------------+-------------+-------------+-------------+
      idSum |    0x0304   |   0x0206    |    0x0000   |   0x0102    |
            +-------------+-------------+-------------+-------------+
    hashSum |   0x0101    |   0x4343    |    0x0000   |   0x4242    |
            +-------------+-------------+-------------+-------------+
```

Figure 12

After removal of element with ID 0x0304 and hash 0x0101 mapped to {0,1} from the IBF:

```
              bucket-0     bucket-1       bucket-2      bucket-3
            +-------------+-------------+-------------+-------------+
      count |      0      |      1      |      0      |      1      |
            +-------------+-------------+-------------+-------------+
      idSum |    0x0000   |   0x0102    |    0x0000   |   0x0102    |
            +-------------+-------------+-------------+-------------+
    hashSum |    0x0000   |   0x4242    |    0x0000   |   0x4242    |
            +-------------+-------------+-------------+-------------+
```

Figure 13

Note that it is possible to "remove" elements from an IBF that were never present in the IBF in the first place. A negative counter value is thus indicative of elements that were removed without having been added. Note that an IBF bucket counter of zero no longer guarantees that an element mapped to that bucket is not present in the set: a bucket with a counter of zero can be the result of one element being added and a different element (mapped to the same bucket) being removed. To check that an element is not present requires a counter of zero and an IDSUM and HASHSUM of zero --- and some certainty that there was no collision due to the limited number of bits in IDSUM and HASHSUM. Thus, IBFs are not suitable to replace BFs or IBFs.

Buckets in an IBF with a counter of 1 or -1 are crucial for decoding an IBF, as they MIGHT represent only a single element, with the IDSUM being the ID of that element. Following Eppstein [Eppstein], we will call buckets that only represent a single element *pure buckets*. Note that due to the possibility of multiple insertion and removal operations affecting the same bucket, not all buckets with a counter of 1 or -1 are actually pure buckets. Sometimes a counter can be 1 or -1 because N elements mapped to that bucket were added while N-1 or N+1 different elements also mapped to that bucket were removed.

3.5.3. Extracting elements

Extracting elements from an IBF yields IDs of elements from the IBF. Elements are extracted from an IBF by repeatedly performing a decode operation on the IBF.

A decode operation requires a pure bucket, that is a bucket to which M only mapped a single element, to succeed. Thus, if there is no bucket with a counter of 1 or -1, decoding fails. However, as a counter of 1 or -1 is not a guarantee that the bucket is pure, there is also a chance that the decoder returns an IDSUM value that is actually the XOR of several IDSUMs. This is primarily detected by checking that the HASHSUM is the hash of the IDSUM. Only if the HASHSUM also matches, the bucket could be pure. Additionally, one MUST check that the IDSUM value actually would be mapped by M to the respective bucket. If not, there was a hash collision and the bucket is also not pure.

The very rare case that after all these checks a bucket is still falsely identified as pure MUST be detected (say by determining that extracted element IDs do not match any actual elements), and addressed at a higher level in the protocol. As these failures are probabilistic and depend on element IDs and the IBF construction, they can typically be avoided by retrying with different parameters, such as a different way to assign element IDs to elements (by varying the IBF-salt), using a larger value for L, or a different mapping function M. A more common scenario (especially if L was too small) is that IBF decoding fails because there is no pure bucket. In this case, the higher-level protocol generally MUST also retry using different parameters (except if an attack is detected).

Suppose the IBF contains a pure bucket. In this case, the IDSUM in the bucket is the ID of an element. Furthermore, it is then possible to remove that element from the IBF (by inserting it if the counter was negative, and by removing it if the counter was positive). This is likely to cause other buckets to become pure, allowing further elements to be decoded. Eventually, decoding ought to finish with all counters and IDSUM and HASHSUM values reach zero. However, it is also possible that an IBF only partly decodes and then decoding fails due to the lack of pure buckets after extracting some element IDs.

In the following example the successful decoding of an IBF containing the two elements previously added in our running example.

We begin with an IBF with two elements added:

```
             bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
   count  |      1      |      2      |      0      |      1      |
          +-------------+-------------+-------------+-------------+
   idSum  |   0x0304    |   0x0206    |    0x0000   |   0x0102    |
          +-------------+-------------+-------------+-------------+
 hashSum  |   0x0101    |   0x4343    |    0x0000   |   0x4242    |
          +-------------+-------------+-------------+-------------+
```

Figure 14

In the IBF are two pure buckets to decode (buckets 0 and 3) we choose to start with decoding bucket 0. This yields the element with the hash ID 0x0304 and hash 1010. This element ID is mapped to buckets {0,1}. Subtracting this element results in bucket 1 becoming pure:

```
              bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
    count |      0      |      1      |      0      |      1      |
          +-------------+-------------+-------------+-------------+
    idSum |    0x0000   |   0x0102    |    0x0000   |   0x0102    |
          +-------------+-------------+-------------+-------------+
  hashSum |    0x0000   |   0x4242    |    0x0000   |   0x4242    |
          +-------------+-------------+-------------+-------------+
```

Figure 15

We can now decoding bucket 2 and extract the element with the ID 0x0102 and hash 0x4242. Now the IBF is empty. Extraction completes with the status that the IBF has been successfully decoded.

```
              bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
    count |      0      |      0      |      0      |      0      |
          +-------------+-------------+-------------+-------------+
    idSum |    0x0000   |    0x0000   |    0x0000   |    0x0000   |
          +-------------+-------------+-------------+-------------+
  hashSum |    0x0000   |    0x0000   |    0x0000   |    0x0000   |
          +-------------+-------------+-------------+-------------+
```

Figure 16

3.5.4. Set Difference

Given addition and removal as defined above, it is possible to define an operation on IBFs that computes an IBF representing the set difference. Suppose IBF1 represents set A, and IBF2 represents set B. Then this set difference operation will compute IBF3 which represents the set A - B. Note that this computation can be done only on the IBFs, and does not require access to the elements from set A or B. To calculate the IBF representing this set difference, both IBFs MUST have the same length L, the same number of buckets per element k and use the same map M. Naturally, all IDs must have been computed using the same IBF-salt. Given this, one can compute the IBF representing the set difference by taking the XOR of the IDSUM and HASHSUM values of the respective buckets and subtracting the respective counters. Care MUST be taken to handle overflows and underflows by setting the counter to "infinity" as necessary. The result is a new IBF with the same number of buckets representing the set difference.

This new IBF can be decoded as described in section 3.5.3. The new IBF can have two types of pure buckets with counter set to 1 or -1. If the counter is set to 1 the element is missing in the secondary set, and if the counter is set to -1 the element is missing in the primary set.

To demonstrate the set difference operation we compare IBF-A with IBF-B and generate as described IBF-AB

IBF-A contains the elements with ID 0x0304 and hash 0x0101 mapped to {0,1}, and ID 0x0102 and hash 0x4242 mapped to {1,3}:

```
              bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
    count |      1      |      2      |      0      |      1      |
          +-------------+-------------+-------------+-------------+
    idSum |    0x0304   |   0x0206    |    0x0000   |   0x0102    |
          +-------------+-------------+-------------+-------------+
  hashSum |    0x0101   |   0x4343    |    0x0000   |   0x4242    |
          +-------------+-------------+-------------+-------------+
```

Figure 17

IBF-B also contains the element with ID 0x0102 and and another element with ID 0x1345 and hash 0x5050 mapped to {1,2}.

```
              bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
    count |      0      |      1      |      1      |      1      |
          +-------------+-------------+-------------+-------------+
    idSum |    0x0000   |    0x1447   |    0x1345   |    0x0102   |
          +-------------+-------------+-------------+-------------+
  hashSum |    0x0000   |    0x9292   |    0x5050   |    0x4242   |
          +-------------+-------------+-------------+-------------+
```

Figure 18

IBF-A minus IBF-B is then:

```
              bucket-0     bucket-1       bucket-2      bucket-3
          +-------------+-------------+-------------+-------------+
    count |      1      |      0      |     -1      |      0      |
          +-------------+-------------+-------------+-------------+
    idSum |    0x0304   |    0x1049   |    0x1345   |    0x0000   |
          +-------------+-------------+-------------+-------------+
  hashSum |    0x0101   |    0x5151   |    0x5050   |    0x0000   |
          +-------------+-------------+-------------+-------------+
```

Figure 19

After calculating and decoding the IBF-AB shows clear that in IBF-A the element with the hash 0x5050 is missing (-1 in bucket 2) while in IBF-B the element with the hash 0101 is missing (1 in bucket 0). The element with hash 0x4242 is present in IBF-A and IBF-B and is removed by the set difference operation. Bucket 2 is not empty.

3.6. Wire format

For the counter field, we use a variable-size encoding to ensure that even for very large sets the counter should never reach "infinity", while also ensuring that the encoding is compact for small sets. Hence, the counter size transmitted over the wire varies between 1 and 64 bits, depending on the maximum counter in the IBF. A range of 1 to 64 bits should cover most areas of application and can be efficiently implemented on most contemporary CPU architectures and programming languages. The bit length for the transmitted IBF will be communicated in the header of the *IBF* message in the "IMCS" field as unsigned 8-bit integer. For implementation details see section Variable Counter Size.

For the "IDSUM", we always use a 64-bit representation. The IDSUM value MUST have sufficient entropy for the mapping function M to yield reasonably random buckets even for very large values of L. With a 32 bit value the chance that multiple elements may be mapped to the same ID would be quite high, even for moderately large sets. Using more than 64 bits would at best make sense for very large sets, but then it is likely always better to simply afford additional round trips to handle the occasional collision. 64 bits are also a reasonable size for many CPU architectures.

For the "HASHSUM", we always use a 32-bit representation. Here, it is most important to avoid collisions, where different elements are mapped to the same hash, possibly resulting in a bucket being falsely classified as pure. While with 32 bits there remains a non-negligible chance of accidental collisions, our protocol is designed to handle occasional collisions. Hence, at 32 bit the chance is believed to be sufficiently small enough for the protocol to handle those cases efficiently. Smaller hash values would safe bandwidth, but also substantially increase the chance of collisions. 32 bits are also again a reasonable size for many CPU architectures.

4. Strata Estimator

Strata Estimators help estimate the size of the set difference between two sets of elements. This is necessary to efficiently determinate the tuning parameters for an IBF, in particular a good value for L.

Basically a Strata Estimator (SE) is a series of IBFs (with a rather small value of L=79) in which increasingly large subsets of the full set of elements are added to each IBF. For the n-th IBF, the function selecting the subset of elements MUST sample to select (probabilistically) 1/(2^n) of all elements. This can be done by counting the number of trailing bits set to "1" in an element ID, and then inserting the element into the IBF identified by that counter. As a result, all elements will be mapped to one IBF, with the n-th IBF being statistically expected to contain 1/(2^n) elements.

Given two SEs, the set size difference can be estimated by attempting to decode all of the IBFs. Given that L is set to a fixed and rather small value, IBFs containing large strata will likely fail to decode. For IBFs that failed to decode, one simply extrapolates the number of elements by scaling the numbers obtained from the other IBFs that did decode. If none of the IBFs of the SE decoded (which given a reasonable number of IBFs in the SE should be highly unlikely), one can theoretically retry using a different IBF-salt.

When decoding the IBFs in the strata estimator, it is possible to determine on which side which part of the difference is. For this purpose, the pure buckets with counter 1 and -1 must be distinguished and assigned to the respective side when decoding the IBFs.

5. Mode of Operation

Depending on the state of the two sets the set union protocol uses different modes of operation to efficiently determinate missing elements between the two sets.

The simplest mode is the *full synchronisation mode*. If the difference between the sets of the two peers exceeds a certain threshold, the overhead to determine which elements are different would outweigh the overhead of simply sending the complete set. Hence, the protocol may determine that the most efficient method is to exchange the full sets.

The second possibility is that the difference between the sets is relatively small compared to the set size. In this case, the *differential synchronisation mode* is more efficient. Given these two possibilities, the first steps of the protocol are used to determine which mode MUST be used.

Thus, the set union protocol always begins with the following operation mode independent steps:

The initiating peer begins in the **Initiating Connection** state and the receiving peer in the **Expecting Connection** state. The first step for the initiating peer in the protocol is to send an *Operation Request* to the receiving peer and transition into the **Expect SE** state. After receiving the *Operation Request* the receiving peer transitions to the **Expecting IBF** state and answers with the *Strata Estimator* message. When the initiating peer receives the *Strata Estimator* message, it decides with some heuristics which operation mode is likely more suitable for the estimated set difference and the application-provided latency-bandwidth tradeoff. The detailed algorithm used to choose between the Full Synchronisation Mode and the Differential Synchronisation Mode is explained in the section Combined Mode below.

5.1. Full Synchronisation Mode

When the initiating peer decides to use the full synchronisation mode and it is better that the other peer sends his set first, the initiating peer sends a *Request Full* message, and transitions from **Expecting SE** to the **Full Receiving** state. If it has been determined that it is better that the initiating peer sends his set first, the initiating peer sends a *Send Full* message followed by all set elements in *Full Element* messages to the other peer, followed by the *Full Done* message, and transitions into the **Full Sending** state.

A state diagram illustrating the state machine used during full synchronization is provided here.

The behavior of the participants the different state is described below:

Expecting IBF:

If a peer in the **Expecting IBF** state receives a *Request Full* message from the other peer, the peer sends all the elements of his set followed by a *Full Done* message to the other peer, and transitions to the **Full Sending** state. If the peer receives an *Send Full* message followed by *Full Element* messages, the peer processes the element and transitions to the **Full Receiving** state.

Full Sending: While a peer is in **Full Sending** state the peer expects to continuously receive elements from the other peer. As soon as a the *Full Done* message is received, the peer transitions into the **Finished** state.

Full Receiving: While a peer is in the **Full Receiving** state, it expects to continuously receive elements from the other peer. As soon as a the *Full Done* message is received, it sends the remaining elements (those it did not receive) from his set to the other peer, followed by a *Full Done*. After sending the last message, the peer transitions into the **Finished** state.

5.2. Differential Synchronisation Mode

The message format used by the protocol limits the maximum message size to 64 kb. Given that L can be large, an IBF will not always fit within that size limit. To deal with this, larger IBFs are split into multiple messages.

When the initiating peer in the **Expected SE** state decides to use the differential synchronisation mode, it sends an IBF, which may consist of several *IBF* messages, to the receiving peer and transitions into the **Passive Decoding** state.

The receiving peer in the **Expecting IBF** state receives the first *IBF* message from the initiating peer, and transitions into the **Expecting IBF Last** state if the IBF was split into multiple *IBF* messages. If there is just a single *IBF* message, the receiving peer transitions directly to the **Active Decoding** state.

The peer that is in the **Active Decoding**, **Finish Closing** or in the **Expecting IBF Last** state is called the active peer, and the peer that is in either the **Passive Decoding** or the **Finish Waiting** state is called the passive peer.

A state diagram illustrating the state machine used during differential synchronization is provided here.

The behavior of the participants the different states is described below:

Passive Decoding: In the **Passive Decoding** state the passive peer reacts to requests from the active peer. The action the passive peer executes depends on the message the passive peer receives in the **Passive Decoding** state from the active peer and is described below on a per message basis.

Inquiry message: The *Inquiry* message is received if the active peer requests the SHA-512 hash of one or more elements (by sending the 64 bit element ID) that are missing from the active peer's set. In this case the passive peer answers with *Offer* messages which contain the SHA-512 hash of the requested element. If the passive peer does not have an element with a matching element ID, it MUST ignore the inquiry (in this case, a bucket was falsely classified as pure, decoding the IBF will eventually fail, and roles will be swapped). It should be verified that after an falsely classified pure bucket a role change is made. If multiple elements match the 64 bit element ID, the passive peer MUST send offers for all of the matching elements.

Demand message: The *Demand* message is received if the active peer requests a complete element that is missing in the active peers set in response to an offer. If the requested element is known and has not yet been transmitted the passive peer answers with an *Element* message which contains the full, application-dependent data of the requested element. If the passive peer receives a demand for a SHA-512 hash for which it has no corresponding element, a protocol violation is detected and the protocol MUST be aborted. Implementations MUST also abort when facing demands without previous matching offers or for which the passive peer previously transmitted the element to the active peer.

Offer message: The *Offer* message is received if the active peer has decoded an element that is present in the active peers set and is likely be missing in the set of the passive peer. If the SHA-512 hash of the offer is indeed not a hash of any of the elements from the set of the passive peer, the passive peer MUST answer with a *Demand* message for that SHA-512 hash and remember that it issued this demand. The demand thus needs to be added to a list with unsatisfied demands.

Element message: When a new *Element* message has been received the peer checks if a corresponding *Demand* for the element has been sent and the demand is still unsatisfied. If the element has been demanded the peer checks the element for validity, removes it from the list of pending demands and then saves the element to the set. Otherwise the peer ignores the element.

IBF message: If an *IBF* message is received, this indicates that decoding of the IBF on the active site has failed and roles will be swapped. The receiving passive peer transitions into the **Expecting IBF Last** state, and waits for more *IBF* messages. There, once the final *IBF Last* message has been received, it transitions to **Active Decoding**.

IBF Last message: If an *IBF Last* message is received this indicates that there is just one IBF slice left and a direct state and role transition from **Passive Decoding** to **Active Decoding** is initiated.

Done message: Receiving the *Done* message signals the passive peer that all demands of the active peer have been satisfied. Alas, the active peer will continue to process demands from the passive peer. Upon receiving this message, the passive peer transitions into the **Finish Waiting** state.

Active Decoding: In the **Active Decoding** state the active peer decodes the IBFs and evaluates the set difference between the active and passive peer. Whenever an element ID is obtained by decoding the IBF, the active peer sends either an offer or an inquiry to the passive peer, depending on which site the decoded element is missing.

If the IBF decodes a positive (1) pure bucket, the element is missing on the passive peers site. Thus, the active peer sends an *Offer* to the passive peer. A negative (-1) pure bucket indicates that an element is missing in the active peers set, so the active peer sends a *Inquiry* to the passive peer.

In case the IBF does not successfully decode anymore, the active peer sends a new IBF computed with a different IBF-salt to the passive peer and changes into **Passive Decoding** state. This initiates a role swap. To reduce overhead and prevent double transmission of offers and elements, the new IBF is created on the local set after updating it with the all of the elements that have been successfully demanded. Note that the active peer MUST NOT wait for all active demands to be satisfied, as demands can fail if a bucket was falsely classified as pure.

As soon as the active peer successfully finished decoding the IBF, the active peer sends a *Done* message to the passive peer.

All other actions taken by the active peer depend on the message the active peer receives from the passive peer. The actions are described below on a per message basis:

Offer message: The *Offer* message indicates that the passive peer received a *Inquiry* message from the active peer. If a inquiry has been sent and the offered element is missing in the active peers set, the active peer sends a *Demand* message to the passive peer. The demand needs to be added to a list of unsatisfied demands. In case the received offer is for an element that is already in the set of the peer, the offer MUST BE ignored.

Demand message: The *Demand* message indicates that the passive peer received a *Offer* from the active peer. The active peer satisfies the demand of the passive peer by sending an *Element* message if a offer request for the element was sent earlier. Otherwise, the protocol MUST be aborted, as peers must never send demands for hashes that they have never been offered.

Element message: If element is received that was not demanded or for which the application-specific validation logic fails, the protocol MUST be aborted. Otherwise, the corresponding demand is marked as satisfied. Note that this applies only to the differential synchronization mode. In full synchronization, it is perfectly normal to receive Full Element messages for elements that were not demanded and that might even already be known locally.

Done message: Receiving the message *Done* indicates that all demands of the passive peer have been satisfied. The active peer then changes into the **Finish Closing** state. If the IBF has not finished decoding and the *Done* is received, the other peer is not in compliance with the protocol and the protocol MUST be aborted.

Expecing IBF Last In this state the active peer continuously receives *IBF* messages from the passive peer. When the last *IBF Last* message is received, the peer changes into the **Active Decoding** state.

Finish Closing / Finish Waiting In this states the peers are waiting for all demands to be satisfied and for the synchronisation to be completed. When all demands are satisfied the peer changes into **Finished** state.

5.3. Combined Mode

In the *combined mode* the protocol decides between Full Synchronisation Mode and the Differential Synchronisation Mode to minimize resource consumption. Typically, the protocol always runs in combined mode, but implementations MAY allow applications to force the use of one of the modes for testing. In this case, applications MUST ensure that the respective options to force a particular mode are used by both participants.

The Differential Synchronisation Mode is only efficient on small set differences or if the byte-size of the elements is large. If the set difference is estimated to be large the Full Synchronisation Mode is more efficient. The exact heuristics and parameters on which the protocol decides which mode MUST be used are described in the Performance Considerations section of this document.

There are two main cases when a Full Synchronisation Mode is always used. The first case is when one of the peers announces having an empty set. This is announced by setting the SETSIZE field in the *Strata Estimator* to 0. The second case is if the application requests full synchronisation explicitly. This is useful for testing and MUST NOT be used in production.

The state diagram illustrating the combined mode can be found here.

6. Messages

This section describes the various message formats used by the protocol.

6.1. Operation Request

6.1.1. Description

This message is the first message of the protocol and it is sent to signal to the receiving peer that the initiating peer wants to initialize a new connection.

This message is sent in the transition between the **Initiating Connection** state and the **Expect SE** state.

If a peer receives this message and is willing to run the protocol, it answers by sending back a *Strata Estimator* message. Otherwise it simply closes the connection.

6.1.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |     ELEMENT COUNT     |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                      APX
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /  APPLICATION DATA                             /
        /                                               /
```

Figure 20

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included.

MSG TYPE is the type of SETU_P2P_OPERATION_REQUEST as registered in GANA Considerations, in network byte order.

ELEMENT COUNT is the number of the elements the requesting party has in its set, as a 32-bit unsigned integer in network byte order.

APX is a SHA-512 hash that identifies the application.

APPLICATION DATA is optional, variable-size application specific data that can be used by the application to decide if it would like to answer the request.

6.2. IBF

6.2.1. Description

The IBF message contains a slice of the IBF.

The *IBF* message is sent at the start of the protocol from the initiating peer in the transaction between **Expect SE** -> **Expecting IBF Last** or when the IBF does not decode and there is a role change in the transition between **Active Decoding** -> **Expecting IBF Last**. This message is only sent if there is more than one IBF slice to be sent. If there is just one slice, then only the IBF Last message is sent.

6.2.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |       IBF SIZE        |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |         OFFSET        |    SALT   |   IMCS    |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                    IBF-SLICE
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 21

where:

MSG SIZE is a 16-bit unsigned integer in network byte orderwhichdescribes the message size in bytes with the header included.

MSG TYPE the type of SETU_P2P_REQUEST_IBF as registered in GANA Considerations in network byte order.

IBF SIZE is a 32-bit unsigned integer which signals the total number of buckets in the IBF. The minimal number of buckets is 37.

OFFSET is a 32-bit unsigned integer which signals the offset of the following IBF slices in the original.

SALT is a 16-bit unsigned integer that contains the IBF-salt which was used to create the IBF.

IMCS is a 16-bit unsigned integer, which describes the number of bits that are required to store a single counter. This is used for the unpacking function as described in the Variable Counter Size section.

IBF-SLICE are variable numbers of slices in an array. A single slice contains multiple 64-bit IDSUMS, 32-bit HASHSUMS and (1-64)-bit COUNTERS of variable size. All values are in the network byte order. The array of IDSUMS is serialized first, followed by an array of HASHSUMS. Last comes an array of unsigned COUNTERS (details of the COUNTERS encoding are described in section Section 7.2). The length of the array is defined by MIN(SIZE - OFFSET, MAX_BUCKETS_PER_MESSAGE). MAX_BUCKETS_PER_MESSAGE is defined as 32768 divided by the BUCKET_SIZE which ranges between 97-bits when counter uses bit 1 (IMCS=1) and 160-bit when counter size uses 64 bit (IMCS=64).

To get the IDSUM field, all IDs (computed with the IBF-salt) hitting a bucket under the map M are added up with a binary XOR operation. See Salted Element ID Calculation details about ID generation.

The calculation of the HASHSUM field is done accordingly to the calculation of the IDSUM field: all HASHes are added up with a binary XOR operation. The HASH value is calculated as described in detail in section HASH calculation.

The algorithm to find the correct bucket in which the ID and the HASH have to be added is described in detail in section Mapping Function.

Test vectors for an implementation can be found in the Test Vectors section

```
                           IBF-SLICE
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                    IDSUMS                     |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                    IDSUMS                     |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |        HASHSUMS       |        HASHSUMS       |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |        COUNTERS*      |        COUNTERS*      |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
* Counter size is variable. In this example the IMCS is 32 (4 bytes).
```

Figure 22

6.3. IBF Last

6.3.1. Description

This message indicates to the remote peer that this is the last slice of the Bloom filter. The receiving peer MUST check that the sizes and offsets of all received IBF slices add up to the total IBF SIZE that was given.

Receiving this message initiates the state transmissions **Expecting IBF Last** -> **Active Decoding**, **Expecting IBF** -> **Active Decoding** and **Passive Decoding** -> **Active Decoding**. This message can initiate a peer the roll change from **Active Decoding** to **Passive Decoding**.

6.3.2. Structure

The binary structure is exactly the same as the Structure of the message IBF with a different "MSG TYPE" which is defined in GANA Considerations "SETU_P2P_IBF_LAST".

6.4. Element

6.4.1. Description

The *Element* message contains an element that is synchronized in the Differential Synchronisation Mode and transmits a full element between the peers.

This message is sent in the state **Active Decoding** and **Passive Decoding** as answer to a *Demand* message from the remote peer. The *Element* message can also be received in the **Finish Closing** or **Finish Waiting** state after receiving a *Done* message from the remote peer. In this case the peer changes to the **Finished** state as soon as all demands for elements have been satisfied.

This message is exclusively used in the Differential Synchronisation Mode.

6.4.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |   E TYPE  |  PADDING  |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |   E SIZE  |              DATA
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 23

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included.

MSG TYPE is SETU_P2P_ELEMENTS as registered in GANA Considerations in network byte order.

E TYPE is a 16-bit unsigned integer which defines the element type for the application.

PADDING is 16-bit always set to zero.

E SIZE is a 16-bit unsigned integer that signals the size of the elements data part.

DATA is a field with variable length that contains the data of the element.

6.5. Offer

6.5.1. Description

The *Offer* message is an answer to an *Inquiry* message and transmits the full hash of an element that has been requested by the other peer. This full hash enables the other peer to check if the element is really missing in his set and eventually sends a *Demand* message for that element.

The offer is sent and received only in the **Active Decoding** and in the **Passive Decoding** state.

This message is exclusively sent in the Differential Synchronisation Mode.

6.5.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |         HASH 1
        +-----+-----+-----+-----+-----+-----+-----+-----+
       ...                                              ...
        +-----+-----+-----+-----+-----+-----+-----+-----+
                 HASH 1         |         HASH 2
        +-----+-----+-----+-----+-----+-----+-----+-----+
       ...                                              ...
        +-----+-----+-----+-----+-----+-----+-----+-----+
                 HASH 2         |         HASH n
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 24

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes header included.

MSG TYPE is SETU_P2P_OFFER as registered in GANA Considerations in network byte order.

HASH n contains n (one or more) successive SHA 512-bit hashes of the elements that are being requested with *Inquiry* messages.

6.6. Inquiry

6.6.1. Description

The *Inquiry* message is exclusively sent by the active peer in **Active Decoding** state to request the full hash of an element that is missing in the active peers set. This is normally answered by the passive peer with *Offer* message.

This message is exclusively sent in the Differential Synchronisation Mode.

6.6.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |          SALT         |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                   IBF KEY 1                   |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                   IBF KEY 2                   |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        ...                                           ...
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                   IBF KEY n                   |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 25

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included.

MSG TYPE is SETU_P2P_INQUIRY as registered in GANA Considerations in network byte order.

IBF KEY contains n (one or more) successive ibf keys (64-bit unsigned integer) for which the inquiry is sent.

6.7. Demand

6.7.1. Description

The *Demand* message is sent in the **Active Decoding** and in the **Passive Decoding** state. It is an answer to a received *Offer* message and is sent if the element described in the *Offer* message is missing in the peers set. In the normal workflow the answer to the *Demand* message is an *Element* message.

This message is exclusively sent in the Differential Synchronisation Mode.

6.7.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |         HASH 1
        +-----+-----+-----+-----+-----+-----+-----+-----+
       ...                                             ...
        +-----+-----+-----+-----+-----+-----+-----+-----+
                 HASH 1         |         HASH 2
        +-----+-----+-----+-----+-----+-----+-----+-----+
       ...                                             ...
        +-----+-----+-----+-----+-----+-----+-----+-----+
                 HASH 2         |         HASH n
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 26

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes and the header is included.

MSG TYPE the type of SETU_P2P_DEMAND as registered in GANA Considerations in network byte order.

HASH n contains n (one or more) successive SHA 512-bit hashes of the elements that are being demanded.

6.8. Done

6.8.1. Description

The *Done* message is sent when all *Demand* messages have been successfully satisfied and from the perspective of the sender the set is completely synchronized.

This message is exclusively sent in the Differential Synchronisation Mode.

6.8.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |     FINAL CHECKSUM
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 27

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included. The value is always 4 for this message type.

MSG TYPE is SETU_P2P_DONE as registered in GANA Considerations in network byte order.

FINAL CHECKSUM a SHA-512 hash XOR sum of the full set after synchronization. This should ensure that the sets are identical in the end!

6.9. Full Done

6.9.1. Description

The *Full Done* message is sent in the Full Synchronisation Mode to signal that all remaining elements of the set have been sent. The message is received and sent in the **Full Sending** and in the **Full Receiving** state. When the *Full Done* message is received in **Full Sending** state the peer changes directly into **Finished** state. In **Full Receiving** state receiving a *Full Done* message initiates the sending of the remaining elements that are missing in the set of the other peer.

6.9.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |      FINAL CHECKSUM
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 28

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included. The value is always 4 for this message type.

MSG TYPE the type of SETU_P2P_FULL_DONE as registered in GANA Considerations in network byte order.

FINAL CHECKSUM a SHA-512 hash XOR sum of the full set after synchronization. This should ensure that the sets are identical in the end!

6.10. Request Full

6.10.1. Description

The *Request Full* message is sent by the initiating peer in **Expect SE** state to the receiving peer, if the operation mode "Full Synchronisation Mode" is determined to be the superior Mode of Operation and that it is the better choice that the other peer sends his elements first. The initiating peer changes after sending the *Request Full* message into **Full Receiving** state.

The receiving peer receives the *Request Full* message in the **Expecting IBF**, afterwards the receiving peer starts sending his complete set in Full Element messages to the initiating peer.

6.10.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |    REMOTE SET DIFF    |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |   REMOTE SET SIZE     |    LOCAL SET DIFF     |
        +-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 29

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included. The value is always 16 for this message type.

MSG TYPE is SETU_P2P_REQUEST_FULL as registered in GANA Considerations in network byte order.

REMOTE SET DIFF is a 32-bit unsigned integer in network byte order, which represents the remote (from the perspective of the sending peer) set difference calculated with strata estimator.

REMOTE SET SIZE is a 32-bit unsigned integer in network byte order, which represents the total remote (from the perspective of the sending peer) set size.

LOCAL SET DIFF is a 32-bit unsigned integer in network byte order, which represents the local (from the perspective of the sending peer) set difference calculated with strata estimator.

6.11. Send Full

6.11.1. Description

The *Send Full* message is sent by the initiating peer in **Expect SE** state to the receiving peer if the operation mode "Full Synchronisation Mode" is determined as superior Mode of Operation and that it is the better choice that the peer sends his elements first. The initiating peer changes after sending the *Request Full* message into **Full Sending** state.

The receiving peer receives the *Send Full* message in the **Expecting IBF** state, afterwards the receiving peer changes into **Full Receiving** state and expects to receive the set of the remote peer.

6.11.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |    REMOTE SET DIFF    |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |   REMOTE SET SIZE     |    LOCAL SET DIFF     |
        +-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 30

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included. The value is always 16 for this message type.

MSG TYPE is SETU_P2P_REQUEST_FULL as registered in GANA Considerations in network byte order.

REMOTE SET DIFF is a 32-bit unsigned integer in network byte order, which represents the remote (from the perspective of the sending peer) set difference calculated with strata estimator.

REMOTE SET SIZE is a 32-bit unsigned integer in network byte order, which represents the total remote (from the perspective of the sending peer) set size.

LOCAL SET DIFF is a 32-bit unsigned integer in network byte order, which represents the local (from the perspective of the sending peer) set difference calculated with strata estimator.

6.12. Strata Estimator

6.12.1. Description

The strata estimator is sent by the receiving peer at the start of the protocol, right after the Operation Request message has been received.

The strata estimator is used to estimate the difference between the two sets as described in section Strata Estimator.

When the initiating peer receives the strata estimator, the peer decides which Mode of Operation to use for the synchronisation. Depending on the size of the set difference and the Mode of Operation the initiating peer changes into **Full Sending**, **Full Receiving** or **Passive Decoding** state.

The *Strata Estimator* message can contain one, two, four or eight strata estimators with different salts, depending on the initial size of the sets. More details can be found in section Multi Strata Estimators.

The IBFs in a strata estimator always have 79 buckets. The reason why can be found in [byzantine_fault_tolerant_set_reconciliation] in section 3.4.2.

6.12.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE | SEC |     SETSIZE
        +-----+-----+-----+-----+-----+-----+-----+-----+
              SETSIZE                 |     SE-SLICES
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 31

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included.

MSG TYPE is SETU_P2P_SE as registered in GANA Considerations in network byte order.

SEC is a 8-bit unsigned integer in network byte order, which indicates how many strata estimators with different salts are attached to the message. Valid values are 1,2,4 or 8, more details can be found in the section Multi Strata Estimators.

SETSIZE is a 64-bit unsigned integer that is defined by the size of the set the SE is

SE-SLICES are variable numbers of slices in an array. A slice can contain one or more Strata Estimators which contain multiple IBFs as described in IBF-SLICES in Section 6.2.2. A SE slice can contain one to eight Strata Estimators which contain 32 (Defined as Constant SE_STRATA_COUNT) IBFs. Every IBF in a SE contains 79 Buckets.

The different SEs are built as in detail described in Section 7.3. Simply put, the IBFs in each SE are serialized as described in Section 6.2.2 starting with the highest stratum. Then the created SEs are appended one after the other starting with the SE that was created with a salt of zero.

```
                          SE-SLICE
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                  SE_1 -> IBF_1
        +-----+-----+-----+-----+-----+-----+-----+-----+
       ...                                               ...
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                  SE_1 -> IBF_30
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |                  SE_2 -> IBF_1
        +-----+-----+-----+-----+-----+-----+-----+-----+
       ...                                               ...
        /                                               /
        /                                               /
```

Figure 32

6.13. Strata Estimator Compressed

6.13.1. Description

The Strata Estimator can be compressed with gzip as described in [RFC1951] to improve performance. This can be recognized by the different message type number from GANA Considerations.

6.13.1.1. Structure

The key difference between the compressed and the uncompressed Strata Estimator is that the SE slices are compressed with gzip ([RFC1951]) in the compressed SE. But the header remains uncompressed with both.

Since the content of the message is the same as the uncompressed Strata Estimator, the details are not repeated here. For details see section 6.12.

6.14. Full Element

6.14.1. Description

The *Full Element* message is the equivalent of the Element message in the Full Synchronisation Mode. It contains a complete element that is missing in the set of the peer that receives this message.

The *Full Element* message is exclusively sent in the transitions **Expecting IBF** -> **Full Receiving** and **Full Receiving** -> **Finished**. The message is only received in the **Full Sending** and **Full Receiving** state.

After the last *Full Element* message has been sent, the *Full Done* message is sent to conclude the full synchronisation of the element sending peer.

6.14.2. Structure

```
        0     8     16    24    32    40    48    56
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |  MSG SIZE |  MSG TYPE |   E TYPE  |  PADDING  |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |    SIZE   |   AE TYPE |         DATA
        +-----+-----+-----+-----+-----+-----+-----+-----+
        /                                               /
        /                                               /
```

Figure 33

where:

MSG SIZE is a 16-bit unsigned integer in network byte order, which describes the message size in bytes with the header included.

MSG TYPE is SETU_P2P_REQUEST_FULL_ELEMENT as registered in GANA Considerations in network byte order.

E TYPE is a 16-bit unsigned integer which defines the element type for the application.

PADDING is 16-bit always set to zero

E SIZE is a 16-bit unsigned integer that signals the size of the elements data part.

AE TYPE is a 16-bit unsigned integer that is needed to identify the type of element that is in the data field

DATA is a field with variable length that contains the data of the element.

7. Performance Considerations

7.1. Formulas

7.1.1. Operation Mode

The decision which Mode of Operation is used is described by the following code. More detailed explanations motivating the design can be found in the accompanying thesis in section 4.5.3. [byzantine_fault_tolerant_set_reconciliation]

The function takes as input the average element size, the local set size, the remote set size, the set differences as estimated from the strata estimator for both the local and remote sets, and the bandwidth/roundtrip tradeoff. The function returns the exact Mode of Operation that is predicted to be best as output: FULL_SYNC_REMOTE_SENDING_FIRST if it is likely cheapest that the other peer transmits his elements first, FULL_SYNC_LOCAL_SENDING_FIRST if it is likely cheapest that the elements are transmitted to the other peer directly, and DIFFERENTIAL_SYNC if the differential synchronisation is likely cheapest.

The constant IBF_BUCKET_NUMBER_FACTOR is always 2 and IBF_MIN_SIZE is 37. The method for deriving this can be found in the IBF parameter study in [byzantine_fault_tolerant_set_reconciliation] in section 4.5.2.

```
# CONSTANTS:
# IBF_BUCKET_NUMBER_FACTOR = 2: The amount the IBF gets increased if
#                               decoding fails
# RTT_MIN_FULL = 2: Minimal round trips used for full Synchronisation
# IBF_MIN_SIZE = 37: The minimal size of an IBF
# MAX_BUCKETS_PER_MESSAGE: Custom value depending on the underlying
#                          protocol
# INPUTS:
# avg_es: The average element size
# lss: The initial local set size
# rss: The remote set size
# lsd: the estimated local set difference calculated by the SE
# rsd: the estimated remote set difference calculated by the SE
# rtt: the tradeoff between round trips and bandwidth
# OUTPUT:
# FULL_SYNC_REMOTE_SENDING_FIRST, FULL_SYNC_LOCAL_SENDING_FIRST or
# DIFFERENTIAL_SYNC

FUNCTION decide_operation_mode(avg_es,
                               lss,
                               rss,
                               lsd
                               rsd,
                               rtt)

    # If a set size is zero always do full sync
    IF 0 == rss THEN
        RETURN FULL_SYNC_LOCAL_SENDING_FIRST
    END IF
    IF 0 == lss THEN
        RETURN FULL_SYNC_REMOTE_SENDING_FIRST
    END IF

    # Estimate required transferred bytes when doing a full
    # synchronisation and transmitting local set first.
    semh = sizeof(ELEMENT_MSG_HEADER)
    estimated_total_diff = rsd + lsd
    total_elements_local_send = rsd + lss
    cost_local_full_sync = avg_es * total_elements_local_send
                         + total_elements_local_send * semh
                         + sizeof(FULL_DONE_MSG_HEADER) * 2
                         + RTT_MIN_FULL * rtt

    # Estimate required transferred bytes when doing a full
    # synchronisation and transmitting remote set first.
    total_elements_remote_send = lsd + rss
    cost_remote_full_sync = avg_es * total_elements_remote_send
                          + total_elements_remote_send * semh
                          + sizeof(FULL_DONE_MSG_HEADER) * 2
                          + (RTT_MIN_FULL + 0.5) * rtt
                          + sizeof(REQUEST_FULL_MSG)

    # Estimate required transferred bytes when doing a differential
    #  synchronisation

    # Estimate messages required to transfer IBF
    ibf_bucket_count = estimated_total_diff * IBF_BUCKET_NUMBER_FACTOR
```

```
     IF ibf_bucket_count <= IBF_MIN_SIZE THEN
         ibf_bucket_count = IBF_MIN_SIZE
     END IF
     ibf_message_count = ceil (ibf_bucket_count / MAX_BUCKETS_PER_MESSAGE)

     # Estimate average counter length with variable counter
     estimated_counter_bits = MIN (2 * LOG2(lss / ibf_bucket_count),
                                   LOG2(lss))
     estimated_counter_bytes = estimated_counter_bits / 8

     # Sum up all messages required to do differential synchronisation
     ibf_bytes = sizeof(IBF_MESSAGE) * ibf_message_count
               + ibf_bucket_count * sizeof(IBF_KEY)
               + ibf_bucket_count * sizeof(IBF_KEYHASH)
               + ibf_bucket_count * estimated_counter_bytes
     # Add 20% overhead to cover IBF retries due to decoding failures
     total_ibf_bytes = ibf_bytes * 1.2

     # Estimate other message sizes to be transfered in diff sync
     # Note that we simplify by adding the header each time;
     # if the implementation combines multiple INQUIRY/DEMAND/OFFER
     # requests in one message, the bandwidth would be lower.
     done_size = sizeof(DONE_HEADER)
     element_size = (avg_es + sizeof(ELEMENT_MSG_HEADER))
                  * estimated_total_diff
     inquery_size = (sizeof(IBF_KEY) + sizeof(INQUERY_MSG_HEADER))
                  * estimated_total_diff
     demand_size  = (sizeof(HASHCODE) + sizeof(DEMAND_MSG_HEADER))
                  * estimated_total_diff
     offer_size   = (sizeof(HASHCODE) + sizeof(OFFER_MSG_HEADER))
                  * estimated_total_diff

     # Estimate total cost
     diff_cost = element_size + done_size + inquery_size
               + demand_size + offer_size + total_ibf_bytes
               + DIFFERENTIAL_RTT_MEAN * rtt

     # Decide for a optimal mode of operation
     full_cost_min = MIN (cost_local_full_sync,
                          cost_remote_full_sync)
     IF full_cost_min < diff_cost THEN
         IF cost_remote_full_sync > cost_local_full_sync THEN
             RETURN FULL_SYNC_LOCAL_SENDING_FIRST
         ELSE
             RETURN FULL_SYNC_REMOTE_SENDING_FIRST
         END IF
     ELSE
         RETURN DIFFERENTIAL_SYNC
     END IF
 END FUNCTION
```

Figure 34

7.1.2. IBF Size

The functions, described in this section, calculate a good initial size (initial_ibf_size) and in case of decoding failure, a good next IBF size (get_next_ibf_size).

These algorithms are described and justified in more details in [byzantine_fault_tolerant_set_reconciliation] in the parameter study in section 3.5.2, the max IBF counter in section 3.10 and the Improved IBF size in section 3.11.

```
# CONSTANTS:
# IBF_BUCKET_NUMBER_FACTOR = 2: The amount the IBF gets increased
                                if decoding fails
# Inputs:
# sd: Estimated set difference
# Output:
# next_size: Size of the initial IBF

FUNCTION initial_ibf_size(sd)
    # We do not go below 37, as 37 buckets should
    # basically always be below one MTU, so there is
    # little to be gained, while a smaller IBF would
    # increase the chance of a decoding failure.
    RETURN MAX(37, IBF_BUCKET_NUMBER_FACTOR * sd);
END FUNCTION

# CONSTANTS:
# IBF_BUCKET_NUMBER_FACTOR = 2: The amount the IBF gets increased if
#                               decoding fails
# Inputs:
# de: Number of elements that have been successfully decoded
# lis: The number of buckets of the last IBF
# Output:
# number of buckets for the next IBF

FUNCTION get_next_ibf_size(de, lis)
    next_size = IBF_BUCKET_NUMBER_FACTOR * (lis - de)
    # The MAX operation here also ensures that the
    # result is positive.
    RETURN MAX(37, next_size);
END FUNCTION
```

Figure 35

7.1.3. Number of Buckets an Element is Hashed into

The number of buckets an element is hashed to is hardcoded to 3. Reasoning and justification can be found in [byzantine_fault_tolerant_set_reconciliation] in the IBF parameter performance study in section 4.5.2.

7.2. Variable Counter Size

The number of bits required to represent the counters of an IBF varies due to different parameters as described in section 3.2 of [byzantine_fault_tolerant_set_reconciliation]. Therefore, a packing algorithm has been implemented. This algorithm encodes the IBF counters in their optimal bit-width and thus minimizes the bandwidth needed to transmit the IBF.

A simple algorithm is used for the packing. In a first step it is determined, which is the largest counter. The the base 2 logarithm then determines how many bits are needed to store it. In a second step for every counter of every bucket, the counter is stored using this many bits. The resulting bit sequence is then simply concatenated.

Three individual functions are used for this purpose. The first one is a function that iterates over each bucket of the IBF to get the maximum counter in the IBF. The second function packs the counters of the IBF, and the third function that unpacks the counters.

As a plausibly check to prevent the byzantine upper bound checks in Section 8.1.2 to fail, implementations must ensure that the estimates of the set size difference added together never exceed the set byzantine upper bound. This could for example happen in case the strata estimator overestimates the set difference.

```
# INPUTS:
# ibf: The IBF
# OUTPUTS:
# returns: Minimal amount of bits required to store the counter

FUNCTION ibf_get_max_counter(ibf)
    max_counter=1 # convince static analysis that we never take log2(0)
    FOR bucket IN ibf DO
        IF bucket.counter > max_counter THEN
            max_counter = bucket.counter
        END IF
    END FOR
    # next bigger discrete number of the binary logarithm of the
    # max counter
    RETURN CEILING( LOG2( max_counter ) )
END FUNCTION

# INPUTS:
# ibf: The IBF
# offset: The offset which defines the starting point from which bucket
#         the pack operation starts
# count: The number of buckets in the array that will be packed
# OUTPUTS:
# returns: A byte array of packed counters to send over the network

# INPUTS:
# ibf: The IBF
# offset: The offset which defines the starting point from which bucket
#         the pack operation starts
# count: The number of buckets in the array that will be packed
# OUTPUTS:
# returns: A byte array of packed counters to send over the network

FUNCTION pack_counter(ibf, offset, count)
    counter_bytes = ibf_get_max_counter(ibf)
    store_bits = 0
    store = 0
    byte_ctr = 0
    buf=[]

    FOR bucket IN ibf[offset] TO ibf[count] DO
        counter = bucket.counter
        byte_len = counter_bytes

        WHILE byte_len + store_bits < 8 DO
            bit_to_shift = 0

            IF store_bits > 0 OR byte_len > 8 THEN
                bit_free = 8 - store_bits
                bit_to_shift = byte_len - bit_free
                store = store << bit_free
            END IF
            buf[byte_ctr] = (( counter >> bit_to_shift) | store) & 0xFF
            byte_ctr = byte_ctr + 1
            byte_len -= 8 - store_bits
            counter = counter & ((1 << byte_len) - 1)
```

```
              store = 0
              store_bits = 0
          END WHILE
          store = (store << byte_len) | counter
          store_bits = store_bits + byte_len
          byte_len = 0
      END FOR

      # Write the last partial packed byte to the buffer
      IF store_bits > 0 THEN
          buf[byte_ctr] = store << (8 - store_bits)
          byte_ctr = byte_ctr + 1
      END IF

      RETURN buf
  FUNCTION END

  # INPUTS:
  # ibf: The IBF
  # offset: The offset which defines the starting point from which bucket
            the packed operation starts
  # count: The number of buckets in the array that will be packed
  # cbl: The bit length of the counter can be found in the
         ibf message in the ibf_counter_bit_length field
  # pd: A byte array which contains the data packed with the pack_counter
        function
  # OUTPUTS:
  # returns: Nothing because the unpacked counter is saved directly
             into the IBF

  FUNCTION unpack_counter(ibf, offset, count, cbl, pd)
      ibf_bucket_ctr = 0
      store = 0
      store_bits = 0
      byte_ctr = 0

      WHILE TRUE
          byte_read = pd[byte_ctr]
          bit_to_pack_left = 8
          byte_ctr++

          WHILE bit_to_pack_left >= 0 DO

              # Prevent packet from reading more than required
              IF ibf_bucket_ctr > (count - 1) THEN
                  RETURN
              END IF

              IF store_bits + bit_to_pack_left >= cbl THEN
                  bit_use = cbl - store_bits

                  IF store_bits > 0 THEN
                      store = store << bit_use
                  END IF
                  bytes_to_shift = bit_to_pack_left - bit_use
                  counter_partial = byte_read >> bytes_to_shift
                  store = store | counter_partial
                  ibf.counter[ibf_bucket_ctr + offset] = store
                  byte_read = byte_read & (( 1 << bytes_to_shift ) - 1)

                  bit_to_pack_left -= bit_use
                  ibf_bucket_ctr++
                  store = 0
                  store_bits = 0
              ELSE
                  store_bits = store_bits + bit_to_pack_left

                  IF 0 == store_bits THEN
                      store = byte_read
                  ELSE
                      store = store << bit_to_pack_left
                      store = store | byte_read
                  END IF
                  BREAK
              END IF
          END WHILE
      END WHILE
  END FUNCTION
```

Figure 36

7.3. Multi Strata Estimators

In order to improve the precision of the estimates not only one strata estimator is transmitted for larger sets. One, two, four or eight strata estimators can be transferred. Transmitting multiple strata estimators has the disadvantage that additional bandwidth will be used, so despite the higher precision, it is not always optimal to transmit eight strata estimators. Therefore, the following rules are used, which are based on the average element size multiplied by the number of elements in the set. This value is denoted as "b" in the table:

SEs	Rule
1	b < 68kb
2	b > 68kb
4	b > 269kb
8	b > 1'077kb

When creating multiple strata estimators, it is important to salt the keys for the IBFs in the strata estimators differently, using the following bit rotation based salting method:

```
# Inputs:
# value: Input value to salt (needs to be 64 bit unsigned)
# salt: Salt to salt value with; Should always be ascending and start
#       at zero
    i.e. SE1 = Salt 0; SE2 = Salt 1 etc.
# Output:
# Returns: Salted value

FUNCTION se_key_salting(value, salt)
    s = (salt * 7) modulo 64
    RETURN (value >> s) | (value << (64 - s))
END FUNCTION
```

Figure 37

Performance study and details about the reasoning for the used methods can be found in [byzantine_fault_tolerant_set_reconciliation] in section 3.4.1 under the title "Added Support for Multiple Strata Estimators". [byzantine_fault_tolerant_set_reconciliation]

8. Security Considerations

The security considerations in this document focus mainly on the security goal of availability. The primary goal of the protocol is to prevent an attacker from wasting computing and network resources of the attacked peer.

To prevent denial of service attacks, it is vital to check that peers can only reconcile a set once in a predefined time span. This is a predefined value and needs to be adapted per use basis. To enhance reliability and to allow for legitimate failures, say due to network connectivity issues, applications SHOULD define a threshold for the maximum number of failed reconciliation attempts in a given time period.

It is important to close and purge connections after a given timeout to prevent draining attacks.

8.1. General Security Checks

In this section general checks are described which should be applied to multiple states.

8.1.1. Input validation

The format of all received messages needs to be properly validated. This is important to prevent many attacks on the code. The application data MUST be validated by the application using the protocol not by the implementation of the protocol. In case the format validation fails the set operation MUST be terminated.

8.1.2. Byzantine Boundaries

To restrict an attacker there should be an upper and lower bound defined and checked at the beginning of the protocol, based on prior knowledge, for the number of elements. The lower byzantine bound can be, for example, the number of elements the other peer had in his set at the last contact. The upper byzantine bound can be a practical maximum e.g. the number of e-voting votes, in Switzerland.

```
# Input:
# rec: Number of elements in remote set
# rsd: Number of elements differ in remote set
# lec: Number of elements in local set
# lsd: Number of elements differ in local set
# UPPER_BOUND: Given byzantine upper bound
# LOWER_BOUND: Given byzantine lower bound
# Output:
# returns TRUE if parameters in byzantine bounds otherwise returns FALSE
FUNCTION check_byzantine_bounds (rec,rsd,lec,lsd)
    IF rec + rsd > UPPER_BOUND THEN
        RETURN FALSE
    END IF
    IF lec + lsd > UPPER_BOUND THEN
        RETURN FALSE
    END IF
    IF rec < LOWER_BOUND THEN
        RETURN FALSE
    END IF
    RETURN TRUE
END FUNCTION
```

Figure 38

8.1.3. Valid State

To harden the protocol against attacks, controls were introduced in the improved implementation that check for each message whether the message was received in the correct state. This is central so that an attacker finds as little attack surface as possible and makes it more difficult for the attacker to send the protocol into an endless loop, for example.

8.1.4. Message Flow Control

For most messages received and sent there needs to be a check in place that checks that a message is not received multiple times. This is solved with a global store (message) and the following code

The sequence in which messages are received and sent is arranged in a chain. The messages are dependent on each other. There are dependencies that are mandatory, e.g. for a sent "Demand" message, an "Element" message must always be received. But there are also messages for which a response is not mandatory, e.g. the *Inquiry* message is only followed by an "Offer" message, if the corresponding element is in the set. Due to this fact, checks can be installed to verify compliance with the following chain.

```
Chain for
elements    +---------+    +---------+    +---------+    +---------+
NOT in IBF  | INQUIRY |--->|  OFFER  |===>|  DEMAND |===>| ELEMENT |
decoding    +---------+    +---------+    +---------+    +---------+
peers set

Chain for
elements    +---------+    +---------+    +---------+
in IBF      |  OFFER  |--->| DEMAND  |===>| ELEMENT |
decoding    +---------+    +---------+    +---------+
peers set

            --->: Answer not mandatory
            ===>: Always answer needed.
```

Figure 39

In the message control flow its important to ensure that no duplicated messages are received (Except inquiries where collisions are possible) and only messages are received which are compliant with the flow in Figure 39. To link messages the SHA-512 element hashes, that are part of all messages, except in the *Inquiry* messages, can be used. To link an *Inquiry* message to an *Offer* message the SHA-512 hash from the offer has to be salted and converted to the IBF-Key (as described in Figure 7). The IBF-Key can be matched with the received *Inquiry* message.

At the end of the set reconciliation operation after receiving and sending the *Done* message, it should be checked that all demands have been satisfied and all elements have been received.

This is based on [byzantine_fault_tolerant_set_reconciliation], section 5.3 (Message Control Flow).

8.1.5. Limit Active/Passive Decoding changes

To prevent an attacker from sending a peer into an endless loop between active and passive decoding, a limitation for active/passive roll switches is required. Otherwise, an attacker could force the victim to waste unlimited amount of resources by just transmitting IBFs that do not decode. This can be implemented by a simple counter which terminates the operation after a predefined number of switches. The maximum number of switches needs to be defined in such a way that it is very improbable that more switches are required in a legitimate interaction, and hence the malicious behavior of the other peer is assured.

The question after how many active/passive switches it can be assumed that the other peer is not honest, depends on the various tuning parameters of the algorithm. Section 5.4 of [byzantine_fault_tolerant_set_reconciliation] demonstrates that the probability of decoding failure is less than 15% for each round. The probability that there will be n legitimate active/

passive changes is thus less than 0.15^{round number}. Which means that after about 30 active/passive switches it can be said with a certainty of 2^80 that one of the peers is not following the protocol. Hence, participants MUST impose a maximum of 30 active/passive changes.

8.1.6. Full Synchronisation Plausibility Check

An attacker can try to use up a peer's bandwidth by pretending that the peer needs full synchronisation, even if the set difference is very small and the attacker only has a few (or even zero) elements that are not already synchronised. In such a case, it would be ideal if the plausibility could already be checked during full synchronisation as to whether the other peer was honest or not with regard to the estimation of the set size difference and thus the choice of mode of operation.

In order to calculate this plausibility, section 5.5 of [byzantine_fault_tolerant_set_reconciliation] describes a formula, which depicts the probability with which one can calculate the corresponding plausibility based on the number of new and repeated elements after each received element.

Besides this approach from probability theory, there is an additional check that can be made. After the entire set has been transferred to the other peer, no known elements may be returned by the second peer, since the second peer should only return the elements that are missing from the initial peer's set.

This two approaches are implemented in the following pseudocode:

```
# Input:
# SECURITY_LEVEL: The security level used e.g. 2^80
# state: The statemachine state
# rs: Estimated remote set difference
# lis: Number of elements in set
# rd: Number of duplicated elements received
# rf: Number of fresh elements received
# Output:
# Returns TRUE if full synchronisation is plausible and FALSE otherwise

FUNCTION full_sync_plausibility_check (state,rs,lis,rd,rf)
    security_level_lb = -1 * SECURITY_LEVEL

    # Make sure that no element is received double when
    # all elements already are transmitted to the oder side.
    IF FULL_SENDING == state AND rd > 0 THEN
        RETURN FALSE
    END IF

    # Probabilistic algorithm to check for plausible
    # element distribution
    IF FULL_RECEIVING == state THEN

        # Prevent division by 0
        IF  0 <= rs THEN
            rs = 1
        END IF

        # Formula to verify plausibility
        base = 1 - (rs / (lis + rs))
        exponent = rd - rf * lis / rs
        value = exponent * (LOG2(base)/LOG2(2))
        IF value < security_level_lb OR value > SECURITY_LEVEL THEN
            RETURN FALSE
        END IF
    END IF
    RETURN TRUE
END FUNCTION
```

Figure 40

8.2. States

In this section the security considerations for each valid message in all states is described, if any other message is received the peer MUST terminate the operation.

8.2.1. Expecting IBF

Security considerations for received messages:

Request Full

It needs to be checked that the full synchronisation mode with receiving peer sending first is plausible according to the algorithm deciding which operation mode is applicable as described in Section 7.1.1.

IBF
: It needs to be checked that the differential synchronisation mode is plausible according to the algorithm deciding which operation mode is applicable as described in Section 7.1.1.

Send Full
: It needs to be checked that the full synchronisation mode with initiating peer sending first is plausible according to the algorithm deciding which operation mode is applicable as described in Section 7.1.1.

8.2.2. Full Sending

Security considerations for received messages:

Full Element
: When receiving full elements there needs to be checked, that every element is a valid element, that no element has been received more than once, and that not more elements have been received than the other peer has committed to at the beginning of the operation. The plausibility should also be checked with an algorithm as described in Section 8.1.6.

Full Done
: When receiving the *Full Done* message, it is important to check that not fewer elements have been received than the other peer has committed to send at the beginning of the operation. If the sets differ (the FINAL CHECKSUM field in the Full Done message does not match to the SHA-512 hash XOR sum of the local set), the operation has failed and the reconciliation MUST be aborted. It is a strong indicator that something went wrong (eg. some hardware bug). This should never occur!

8.2.3. Expecting IBF Last

Security considerations for received messages:

IBF
: The application should check that the overall size of the IBF that is being transmitted is within its resource bounds, and abort the protocol if its resource limits are likely to be exceeded, or if the size is implausible for the given operation.

  It needs to be checked that the offset (message field "OFFSET") for every received *IBF* message is strictly monotonic increasing and is a multiple of the MAX_BUCKETS_PER_MESSAGE defined in the Constants section, otherwise the connection MUST be aborted.

  Another sanity check is to ensure that the "OFFSET" message field never is higher than the "IBF SIZE" field in the *IBF* message.

IBF Last
: When all *IBF* messages have been received an *IBF Last* message should conclude the transmission of the IBF and a change to the **Active Decoding** phase should be ensured.

To verify that all IBFs have been received, a simple validation can be made. The number of buckets in the *IBF Last* message added to the value in the message OFFSET field should always be equal to the "IBF SIZE".

Further plausibility checks can be made. One is to ensure that after each active/passive switch the IBF can never be more than double in size. Another plausibility check is that an IBF probably never will be larger than the byzantine upperbound multiplied by two. The third plausibility check is to take successfully decoded IBF keys (received offers and demands) into account and to validate the size of the received IBF with the in Figure 35 described function get_next_ibf_size(). If any of these three checks fail the operation must be aborted.

8.2.4. Active Decoding

In the **Active Decoding** state it is important to prevent an attacker from generating and transmitting an unlimited number of IBFs that all do not decode, or to generate an IBF constructed to send the peers in an endless loop. To prevent an endless loop in decoding, loop detection MUST be implemented. A solution to prevent endless loop is to limit the number of elements decoded from an IBF. This limit is defined by the number of buckets in the IBF. It is not possible that more elements are decoded from an IBF than an IBF has buckets. If more elements than buckets are in an IBF it is not possible to get pure buckets. An additional check that should be implemented, is to store all element IDs that were prior decoded. When a new element ID is decoded from the IBF it should always be checked that no element ID is repeated. If the same element ID is decoded more than once, this is a strong indication for an invalid IBF and the operation MUST be aborted. Notice that the decoded element IDs are salted as described in Figure 7 so the described bit rotation needs to be reverted before the decoded element ID is stored and compared to the previous decoded element IDs.

If the IBF decodes more elements than are plausible, the operation MUST be terminated. Furthermore, if the IBF decoding successfully terminates and fewer elements were decoded than plausible, the operation MUST also be terminated. The upper thresholds for decoded elements from the IBF is the remote set size the other peer has committed too (Case if the complete remote set is new). The lower threshold for decoding element is the absolute value of the difference between the local and remote set size (Case the set difference is only in the set of a single peer). The other peer's committed set sizes is transmitted in the the **Expecting IBF** state.

Security considerations for received messages:

Offer
: If an offer for an element, that never has been requested by an inquiry or if an offer is received twice, the operation MUST be terminated. This requirement can be fulfilled by saving lists that keep track of the state of all sent inquiries and offers. When answering offers these lists MUST be checked. The sending and receiving of Offer messages should always be protected with an Message Flow Control to secure the protocol against missing, duplicated, out-of-order or unexpected messages.

Element
: If an element that never has been requested by a demand or is received twice, the operation MUST be terminated. The sending and receiving of Element messages should always be protected with an Message Flow Control to secure the protocol against missing, duplicated, out-of-order or unexpected messages.

Demand
: For every received demand an offer has to be sent in advance. If a demand for an element is received, that never has been offered or the offer already has been answered with a demand, the operation MUST be terminated. It is required to implement a list which keeps track of the state of all sent offers and received demands. The sending and receiving of *Demand* messages should always be protected with an Message Flow Control to secure the protocol against missing, duplicated, out-of-order or unexpected messages.

Done
: The *Done* message is only received if the IBF has finished decoding and all offers have been sent. If the *Done* message is received before the decoding of the IBF is finished or all open demands have been answered, the operation MUST be terminated. If the sets differ (the FINAL CHECKSUM field in the Done message does not match to the SHA-512 hash XOR sum of the local set), the operation has failed and the reconciliation MUST be aborted. It is a strong indicator that something went wrong (eg. some hardware bug). This should never occur!

  When a *Done* message is received the "check_if_synchronisation_is_complete()" function from the Message Flow Control is required to ensure that all demands have been satisfied successfully.

8.2.5. Finish Closing

In the **Finish Closing** state the protocol waits for all sent demands to be fulfilled.

In case not all sent demands have been answered in time, the operation has failed and MUST be terminated.

Security considerations for received messages:

Element
: When receiving Element messages it is important to always check the Message Flow Control to secure the protocol against missing, duplicated, out-of-order or unexpected messages.

8.2.6. Finished

In this state the connection is terminated, so no security considerations are needed.

8.2.7. Expect SE

Security considerations for received messages:

Strata Estimator

In case the strata estimator does not decode, the operation MUST be terminated to prevent to get to an unresolvable state. The set difference calculated from the strata estimator needs to be plausible, which means within the byzantine boundaries described in section Byzantine Boundaries.

8.2.8. Full Receiving

Security considerations for received messages:

Full Element
: When receiving full elements there needs to be checked, that every element is a valid element, no element has been received more than once and not more elements are received than the other peer committed to sending at the beginning of the operation. The plausibility should also be checked with an algorithm as described in Section 8.1.6.

Full Done
: When the *Full Done* message is received from the remote peer, it should be checked that the number of elements received matches the number that the remote peer originally committed to transmitting, otherwise the operation MUST be terminated. If the sets differ (the FINAL CHECKSUM field in the Full Done message does not match to the SHA-512 hash XOR sum of the local set), the operation has failed and the reconciliation MUST be aborted. It is a strong indicator that something went wrong (eg. some hardware bug). This should never occur!

8.2.9. Passive Decoding

Security considerations for received messages:

IBF
: In case an IBF message is received by the peer a active/passive role switch is initiated by the active decoding remote peer. A switch into active decoding mode MUST only be permitted for a predefined number of times as described in Section 8.1.5

Inquiry
: A check needs to be in place that prevents receiving an inquiry for an element multiple times or more inquiries than are plausible. The upper thresholds for sent/received inquiries is the remote set size the other peer has committed too (Case if the complete remote set is new). The lower threshold for for sent/received inquiries is the absolute value of the set difference between the local and remote set size (Case the set difference is only in the set of a single peer). The other peer's committed set sizes is transmitted in the the **Expecting IBF** state. Beware that it is possible to get key collisions and an inquiry for the same key can be transmitted multiple times, so the threshold should take this into account. The sending and receiving of *Inquiry* messages should always be protected with an Message Flow Control to secure the protocol against missing, duplicated, out-of-order or unexpected messages.

Demand
: Same action as described for *Demand* message in section Active Decoding.

Offer
: Same action as described for *Offer* message in section Active Decoding.

Done
: Same action as described for *Done* message in section Active Decoding.

Element
: Same action as described for *Element* message in section Active Decoding.

8.2.10. Finish Waiting

In the **Finish Waiting** state the protocol waits for all transmitted demands to be fulfilled.

In case not all transmitted demands have been answered at this time, the operation has failed and the protocol MUST be terminated with an error.

Security considerations for received messages:

Element
: When receiving Element messages it is important to always check the Message Flow Control to secure the protocol against missing, duplicated, out-of-order or unexpected messages.

9. Constants

The following table contains constants used by the protocol. The constants marked with a * are validated through experiments in [byzantine_fault_tolerant_set_reconciliation].

Name	Value	Description
SE_STRATA_COUNT	32	Number of IBFs in a strata estimator.
IBF_HASH_NUM*	3	Number of times an element is hashed to an IBF. (from section 4.5.2)
IBF_FACTOR*	2	The factor by which the size of the IBF is increased in case of decoding failure or initially from the set difference. (from section 4.5.2)
MAX_BUCKETS_PER_MESSAGE	1120	Maximum bucket of an IBF that are transmitted in single message.
IBF_MIN_SIZE*	37	Minimal number of buckets in an IBF. (from section 3.8)
DIFFERENTIAL_RTT_MEAN*	3.65145	The average RTT that is needed for a differential synchronisation.
SECURITY_LEVEL*	2^80	Security level for probabilistic security algorithms. (from section 5.8)
PROBABILITY_FOR_NEW_ROUND*	0.15	The probability for a IBF decoding failure in the differential synchronisation mode. (from section 5.4)
DIFFERENTIAL_RTT_MEAN*	3.65145	The average RTT that is needed for a differential synchronisation. (from section 4.5.3)
MAX_IBF_SIZE	1048576	Maximal number of buckets in an IBF.
AVG_BYTE_SIZE_SE*	4221	Average byte size of a single strata estimator. (from section 3.4.3)
VALID_NUMBER_SE*	[1,2,4,8]	Valid number of SE's (from section 3.4)

Figure 41

10. GANA Considerations

GANA is requested to amend the "GNUnet Message Type" [GANA] registry as follows:

Type	Name	References	Description
559	SETU_P2P_REQUEST_FULL	[This.I-D]	Request the full set of the other peer.
710	SETU_P2P_SEND_FULL	[This.I-D]	Signals to send the full set to the other peer.
560	SETU_P2P_DEMAND	[This.I-D]	Demand the whole element from the otherpeer, given only the hash code.
561	SETU_P2P_INQUIRY	[This.I-D]	Tell the other peer to send a list of hashes that match an IBF key.
562	SETU_P2P_OFFER	[This.I-D]	Tell the other peer which hashes match a given IBF key.
563	SETU_P2P_OPERATION_REQUEST	[This.I-D]	Request a set union operation from a remote peer.
564	SETU_P2P_SE	[This.I-D]	Strata Estimator uncompressed.
565	SETU_P2P_IBF	[This.I-D]	Invertible Bloom Filter slices.
566	SETU_P2P_ELEMENTS	[This.I-D]	Actual set elements.
567	SETU_P2P_IBF_LAST	[This.I-D]	Invertible Bloom Filter Last Slices.
568	SETU_P2P_DONE	[This.I-D]	Set operation is done.
569	SETU_P2P_SEC	[This.I-D]	Strata Estimator compressed.
570	SETU_P2P_FULL_DONE	[This.I-D]	All elements in full synchronisation mode have been sent is done.
571	SETU_P2P_FULL_ELEMENT	[This.I-D]	Send an actual element in full synchronisation mode.

Figure 42

11. Contributors

The GNUnet implementation of the byzantine fault tolerant set reconciliation protocol was originally implemented by Florian Dold.

12. Normative References

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3385] Sheinwald, D., Satran, J., Thaler, P., and V. Cavanna, "Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations", RFC 3385, DOI 10.17487/RFC3385, September 2002, <https://www.rfc-editor.org/info/rfc3385>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <https://www.rfc-editor.org/info/rfc1951>.

[byzantine_fault_tolerant_set_reconciliation] Summermatter, E., "Byzantine Fault Tolerant Set Reconciliation", 2021, <https://summermatter.net/byzantine-fault-tolerant-set-reconciliation-summermatter.pdf>.

[GANA] GNUnet e.V., "GNUnet Assigned Numbers Authority (GANA)", April 2020, <https://gana.gnunet.org/>.

[CryptographicallySecureVoting] Dold, F., "Cryptographically Secure, Distributed Electronic Voting", <https://git.gnunet.org/bibliography.git/plain/docs/ba_dold_voting_24aug2014.pdf>.

[ByzantineSetUnionConsensusUsingEfficientSetReconciliation] Dold, F. and C. Grothoff, "Byzantine set-union consensus using efficient set reconciliation", <https://doi.org/10.1186/s13635-017-0066-3>.

[Eppstein] Eppstein, D., Goodrich, M., Uyeda, F., and G. Varghese, "What's the Difference? Efficient Set Reconciliation without Prior Context", <https://doi.org/10.1145/2018436.2018462>.

[GNS] Wachs, M., Schanzenbach, M., and C. Grothoff, "A Censorship-Resistant, Privacy-Enhancing and Fully Decentralized Name System", 2014, <https://doi.org/10.1007/978-3-319-12280-9_9>.

Appendix A. Test Vectors

A.1. Map Function

INPUTS:

```
k: 3
ibf_size: 300

key1: 0xFFFFFFFFFFFFFFFF (64-bit)
key2: 0x0000000000000000 (64-bit)
key3: 0x00000000FFFFFFFF (64-bit)
key4: 0xC662B6298512A22D (64-bit)
key5: 0xF20fA7C0AA0585BE (64-bit)
```

Figure 43

OUTPUT:

```
key1: ["122","157","192"]
key2: ["85","243","126"]
key3: ["208","101","222"]
key4: ["239","269","56"]
key5: ["150","104","33"]
```

Figure 44

A.2. ID Calculation Function

INPUTS:

```
element 1: 0xFFFFFFFFFFFFFFFF (64-bit)
element 2: 0x0000000000000000 (64-bit)
element 3: 0x00000000FFFFFFFF (64-bit)
element 4: 0xC662B6298512A22D (64-bit)
element 5: 0xF20fA7C0AA0585BE (64-bit)
```

Figure 45

OUTPUT:

```
element 1: 0x5AFB177B
element 2: 0x64AB557C
element 3: 0xCB5DB740
element 4: 0x8C6A2BB2
element 5: 0x7EC42981
```

Figure 46

A.3. Counter Compression Function

INPUTS:

```
counter serie 1: [1,8,10,6,2] (min bytes 4)
counter serie 2: [26,17,19,15,2,8] (min bytes 5)
counter serie 3: [4,2,0,1,3] (min bytes 3)
```

Figure 47

OUTPUT:

```
counter serie 1: 0x18A62
counter serie 2: 0x3519BC48
counter serie 3: 0x440B
```

Figure 48

Authors' Addresses

Elias Summermatter
Seccom GmbH
Brunnmattstrasse 44
CH-3007 Bern
Switzerland
Email: elias.summermatter@seccom.ch

Christian Grothoff
Berner Fachhochschule
Hoeheweg 80
CH-2501 Biel/Bienne
Switzerland
Email: grothoff@gnunet.org