Workgroup:	Independent Stream	
Internet-Draft:	draft-summermatter-set-union-01	
Published:	10 June 2021	
Intended Status:	Informational	
Expires:	12 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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 12 December 2021.

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][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 F.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 the protocol SHOULD provide a parameter that specifies the cost-ratio of round-trips vs. bandwidth usage. Given this trade-off factor, the protocol will then choose parameters that minimize the total execution costs. In particular, there is one major choice to be made, namely between sending the complete set of elements, or sending only the elements that differ. 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.

The objective here 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 need to remember the number of elements previously shared with a peer, and provide a way to check that elements are well-formed. Applications may also be able to provide an upper bound on the total number of valid elements that may exist. For example, in E-voting, the number of eligible voters could be used to provide such an upper bound.

Initially, this RFC was created as part of Elias Summermatter's bachelor thesis. Many of the algorithms and parameters documented in this RFC are derived in detail 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 Filters

A Bloom filter (BF) is a space-efficient 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. 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.

Further in this document a bitstream output by the mapping function is represented by a set of numeric values for example (0101) = (2,4). 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 1.

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

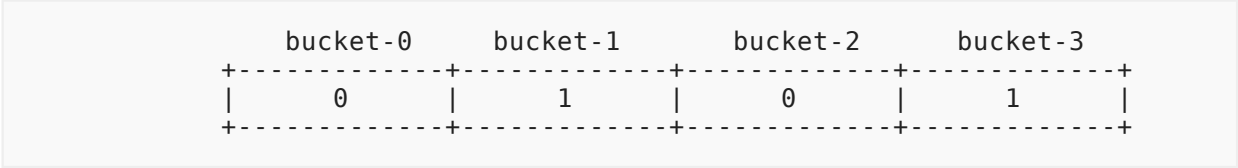


Figure 2

It is easy to see that the M(element) = (0,3) could be in the BF below and M(element) = (0,2) cannot be in the BF below:

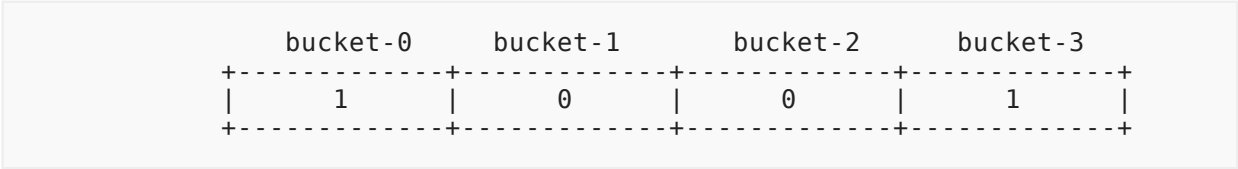


Figure 3

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 an extension of the Bloom Filters. In the 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 bucket on hit to 1 the numeric value stored in the bucket is increased by 1. For example if two colliding elements M(element1) = (1,3) and M(element2) = (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:

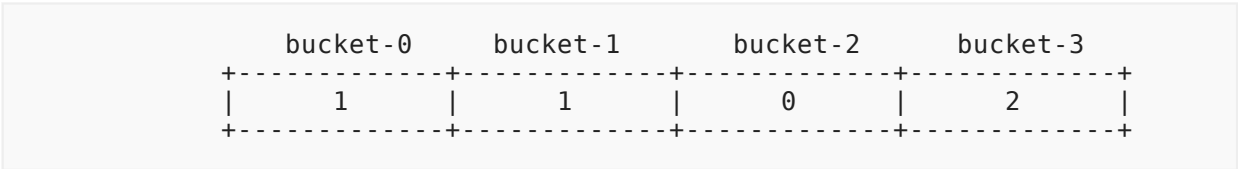


Figure 4

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.

Removing M(element2) = (1,3) from the CBF above:

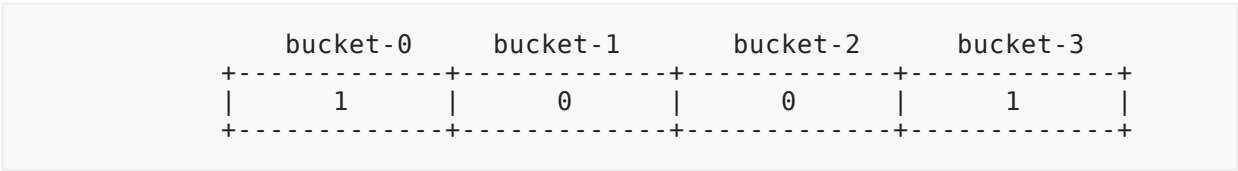


Figure 5

In practice, the number of bits available for the counters is usually finite. For example, given a 4-bit counter, a CBF bucket would overflow 16 elements are mapped to the same bucket. To efficiently 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 depend on the set size. An IBF 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 useful to efficiently extract small differences between large sets.

3.1. Structure

An IBF consists of a mapping function M and L buckets that each store a signed counter and an XHASH. 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, including the CPU architecture.

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".

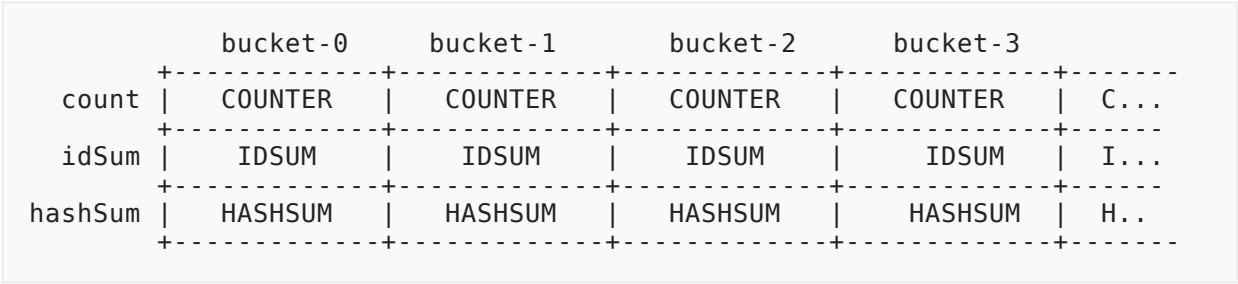


Figure 6

3.2. Operations

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

3.2.1. Insert Element

To add an element to an IBF, the element is mapped to a subset of k buckets using the mapping function M as described in the Bloom Filters section introducing BFs. 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 and the previously stored IDSUM. Furthermore, the HASHSUM is set to the XOR of the hash of the element ID and the previously stored HASHSUM.

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

Empty IBF:

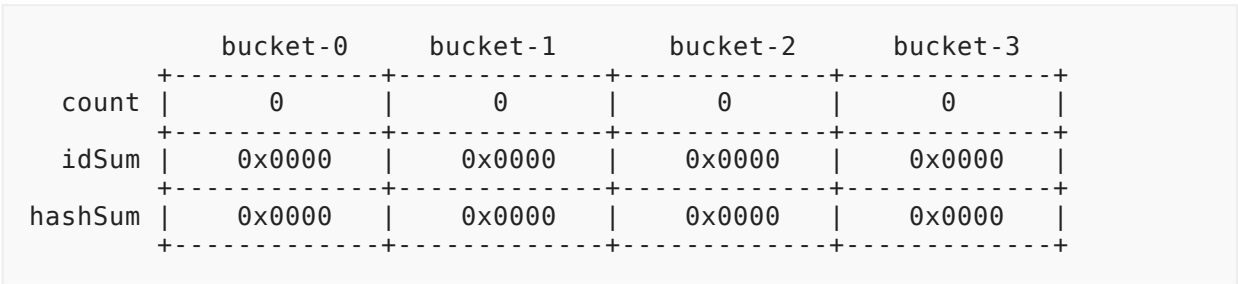


Figure 7

Insert first element: [0101] with ID 0x0102 and hash 0x4242:

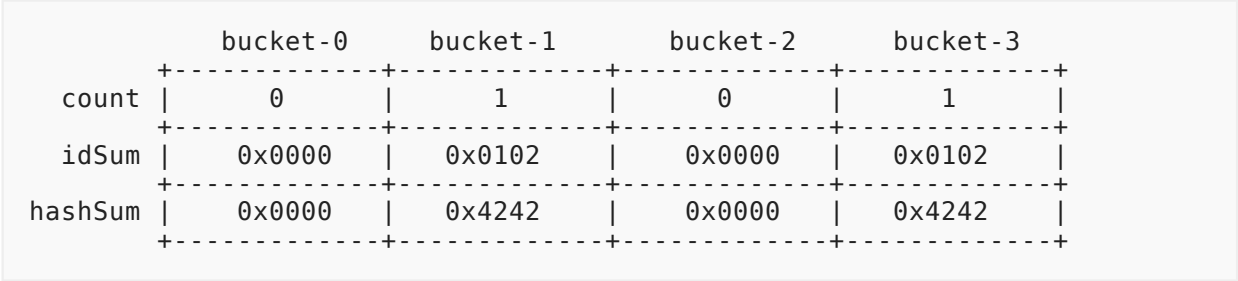


Figure 8

Insert second element: [1100] with ID 0x0304 and hash 0101:

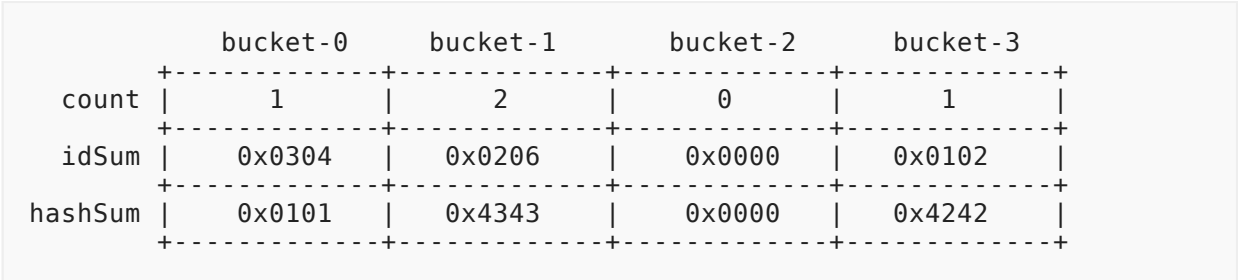


Figure 9

3.2.2. 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 for the element [1100] with the hash 0x0101 is demonstrated.

IBF with encoded elements:

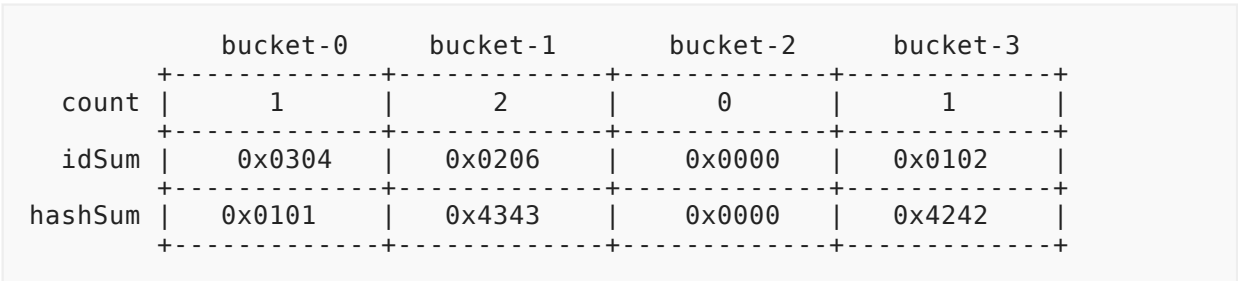


Figure 10

Remove element [1100] with ID 0x0304 and hash 0x0101 from the IBF:

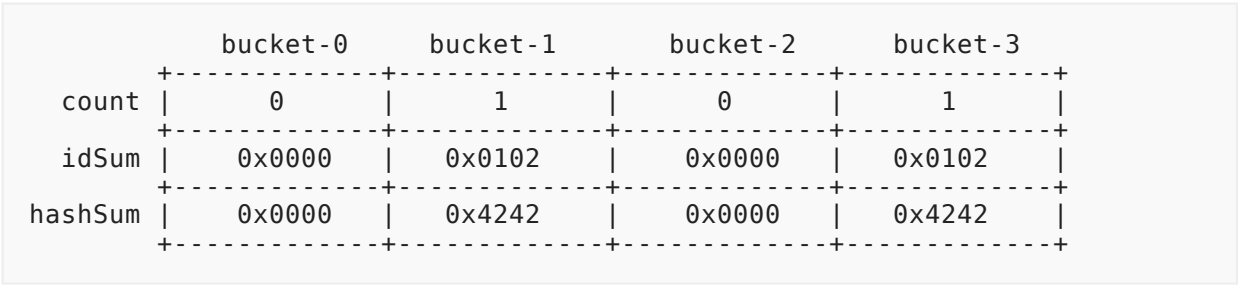


Figure 11

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 (CITE), 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.

3.2.3. Decode IBF

Decoding an IBF yields the HASH of an element from the IBF, or failure.

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.

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, 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 SHOULD also retry using different parameters.

Suppose the IBF contains a pure bucket. In this case, the IDSUM in the bucket identifies a single 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 succeed 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 after obtaining some elements.

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

IBF with the two encoded elements:

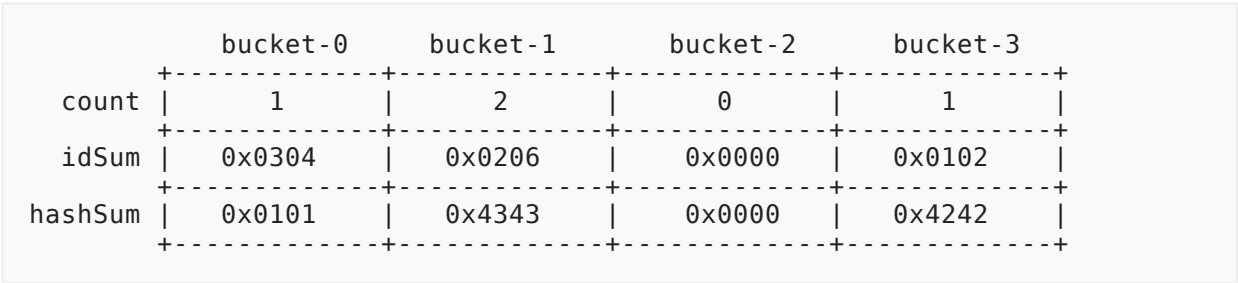


Figure 12

In the IBF are two pure buckets to decode (bit-1 and bit-4) we choose to start with decoding bucket 1, we decode the element with the hash 1010 and we see that there is a new pure bucket created (bit-2)

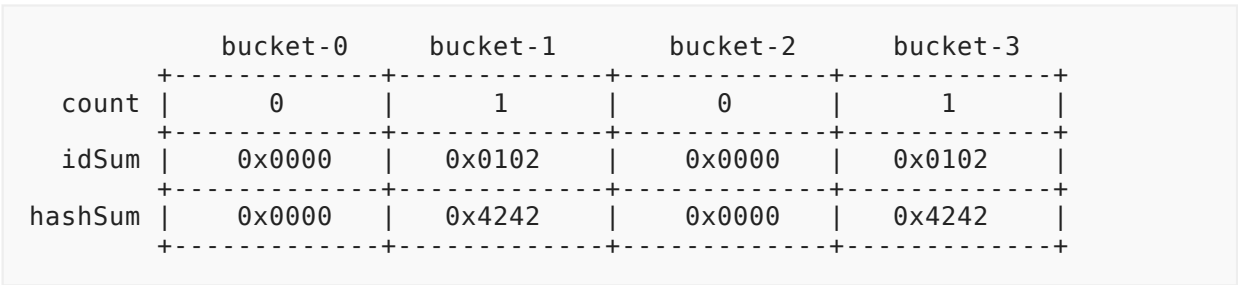
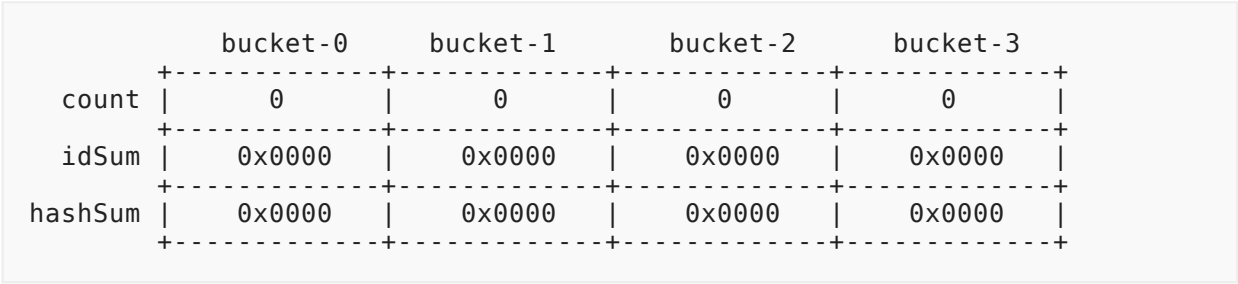


Figure 13

In the IBF only pure buckets are left, we choose to continue decoding bucket 2 and decode element with the hash 0x4242. Now the IBF is empty (all buckets have count 0) that means 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 14

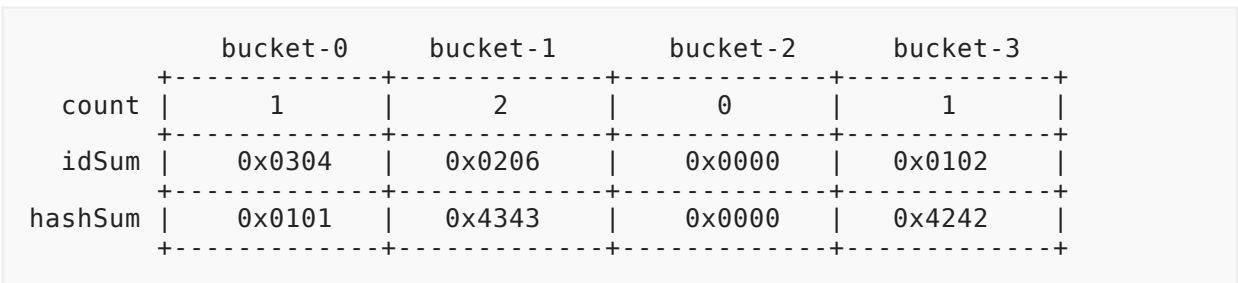
3.2.4. Set Difference

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 --- without having to transfer 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. 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 3.2.3. 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 containing elements with hashes 0x0101 and 0x4242:



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

Figure 15

IBF-B containing elements with hashes 0x4242 and 0x5050

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

Figure 16

IBF-AB XOR value and subtract count:

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

Figure 17

After calculating and decoding the IBF-AB shows clear that in IBF-A the element with the hash 0x5050 is missing (-1 in bit-3) while in IBF-B the element with the hash 0101 is missing (1 in bit-1 and bit-2). The element with hash 0x4242 is present in IBF-A and IBF-B and is removed by the set difference operation (bit-4).

3.3. Wire format

The counter size transmitted over the wire varies between 1 and 64 bit, depending on the maximum counter in the IBF. This variable counter should cover most areas of application. The bit length for the transmitted IBF is defined 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. However, we note that by design only a few elements (certainly less than 127) should ever be mapped to the same bucket, a

small number of bits should suffice. Furthermore, our protocol is designed to handle occasional collisions, so while with 32-bits there remains a chance of accidental collisions, at 32 bit the chance is generally believed to be sufficiently small for the protocol to handle those cases efficiently for a wide range of use-cases. Smaller hash values would safe bandwidth, but also drastically increase the chance of collisions. 32 bits are also again a reasonable size for many CPU architectures.

3.3.1. ID Calculation

The ID is generated as 64-bit output from a Section 2 of HKDF construction [RFC5869] with HMAC-SHA512 as XTR and HMAC-SHA256 as PRF and salt is set to the unsigned 64-bit equivalent of 0. The output is then truncated to 64-bit. It is important that the elements can be redistributed over the buckets in case the IBF does not decode. That is why the ID is salted with a random salt given in the SALT field of this message. Salting is done by calculating a random salt modulo 64 (using only the lowest 6-bits of the salt) and doing a bitwise right rotation of the output of KDF by the 6-bit salts numeric representation.

Representation in pseudocode:

```
# 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 % 64;
  k = key

  /* rotate ibf key */
  k = (k >> s) | (k << (64 - k))
  return key


# INPUTS:
# element: Element to calculated id from.
# salt: Salt of the IBF
# OUTPUT:
# value: the ID of the element

FUNCTION id_calculation (element,ibf_salt):
    salt = 0
    XTR=HMAC-SHA256
    PRF=HMAC-SHA256
    key = HKDF(XTR, PRF, salt, element)
    key = key modulo 2^64 // Truncate
    return salt_key(key,ibf_salt)
```

Figure 18

3.3.2. Mapping Function

The mapping function M as described above in the figure Figure 1 decides in which buckets the ID and HASH have to be binary XORed to. In practice the following algorithm is used:

The first index is simply the HASH modulo the IBF size. The second index is calculated by creating a new 64-bit value by shifting the 32-bit value left and setting the lower 32-bit to the number of indexes already processed. From the resulting 64-bit value a CRC32 checksum is created. The second index is now the modulo of the CRC32 output, this is repeated until the predefined amount of indexes is generated. In the case a index is hit twice, which would mean this bucket could not get pure again, the second hit is just skipped and the next iteration is used.

```
# INPUTS:
# key: Is the ID of the element calculated in the id_calculation
function above.
# number_of_buckets_per_element: Pre-defined numbers of buckets elements
are inserted into
# ibf_size: the size of the ibf (count of buckets)
# OUTPUT:
# dst: Array with bucket IDs to insert ID and HASH

FUNCTION get_bucket_id (key, number_of_buckets_per_element, ibf_size)
  bucket = CRC32(key)

  i = 0
  filled = 0
  WHILE filled < number_of_buckets_per_element

    element_already_in_bucket = false
    j = 0
    WHILE j < filled
      IF dst[j] == bucket modulo ibf_size THEN
        element_already_in_bucket = true
      ENDIF
      j++
    ENDWHILE

    IF !element_already_in_bucket THEN
        dst[filled++] = bucket modulo ibf_size
    ENDIF

    x = (bucket << 32) | i
    bucket = CRC32(x)

    i++
  ENDWHILE
  return dst
```

Figure 19

3.3.3. HASH calculation

The HASH is calculated by calculating the CRC32 checksum of the 64-bit ID value which returns a 32-bit value.

4. Strata Estimator

4.1. Description

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) 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 trying to decode all of the IBFs. Given that L was set to a rather small value, IBFs containing large strata will likely fail to decode. For those 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 choice of L should be highly unlikely), one can retry using a different mapping function M.

In addition, 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

The set union protocol uses IBFs and SEs as primitives. Depending on the state of the two sets there are different strategies or operation modes how to efficiently determinate missing elements between the two sets.

The simplest mode is the "full" synchronisation mode. The idea is that if the difference between the sets of the two peers exceeds a certain threshold, the overhead to determine which elements are different outweighs the overhead of sending the complete set. In this case, the most efficient method can be just to exchange the full sets.

Link to the statemachine diagram

The second possibility is that the difference of the sets is small compared to the set size. In this case, an efficient "differential" synchronisation mode is more efficient. These two possibilities given, the first steps of the protocol are used to determine which mode MUST be used.

Thus, the set synchronisation 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 tradeoff between the Full Synchronisation Mode and the Differential Synchronisation Mode is explained in the section Combined Mode.

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.

Link to the statemachine diagram

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

When the initiating peer in the **Expected SE** state decides to use the differential synchronisation mode, it sends a *IBF* to the receiving peer and transitions into the **Passive Decoding** state.

The receiving peer in the **Expecting IBF** state receives the *IBF* message from the initiating peer and transitions into the **Expecting IBF Last** state when there are multiple *IBF* messages to sent, when 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.

Link to the statemachine diagram

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. 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. If the requested element is valid 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 element, a protocol violation is detected and the protocol MUST be aborted. Implementations MAY strengthen this and forbid demands without previous matching offers.

Offer message: The *Offer* message is received if the active peer has decoded an element that is present in the active peers set and may 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 send demand need 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 rejects 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 or the final *IBF Last* message to be received.

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 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 new complete set after all demands and inquiries have been satisfied.

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 sent demand needs to be added to a list with unsatisfied demands. In case the received offer is for an element that is already in the set of the peer the offer is 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 *Element* message if a offer request for the element has been sent. In case the demanded element does not exist in the set, there was probably a bucket decoded that was not pure. Potentially all *Offer* and *Demand* messages sent later are invalid. In this case a role change active -> passive with a new IBF is easiest.

Element message: An element that is received is marked in the list of demanded elements as satisfied, validated and saved and no further action is taken. Elements that are not demanded or already known are discarded.

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 set reconciliation MUST be aborted.

Expecing IBF Last In the **Expecing IBF Last** state the active peer continuously receives *IBF* messages from the passive peer. When the last *IBF Last* message is received the active peer changes into **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 Full Synchronisation Mode and the Differential Synchronisation Mode are combined to minimize resource consumption.

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.

Link to statemachine diagram

6. Messages

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

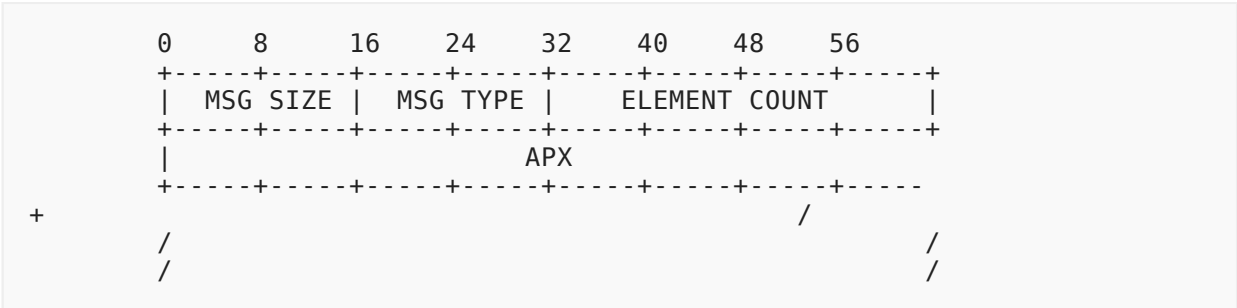


Figure 20

where:

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

MSG TYPE 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.

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 are more than one IBF slice to be sent, in case there is just one slice the IBF Last message is sent.

6.2.2. Structure

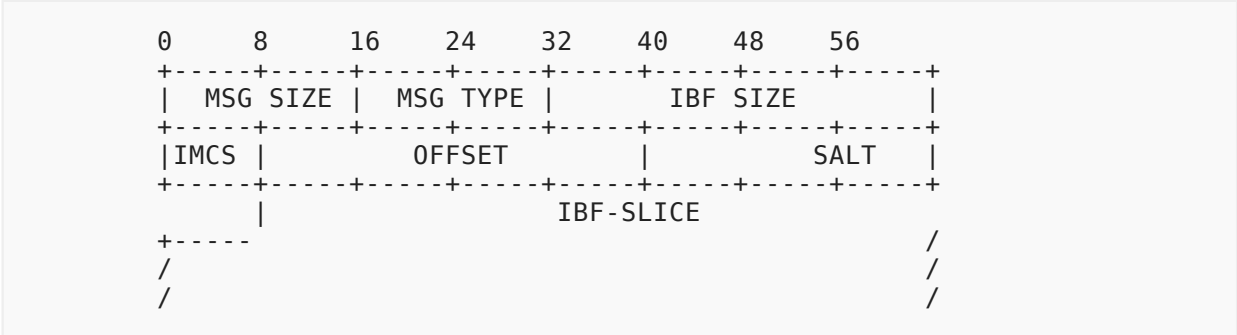


Figure 21

where:

MSG SIZE is a 16-bit unsigned integer in network byte orderwhichdescribes the message size in bytes and 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 number of buckets in the IBF.

IMCS IBF max counter size is a 8-bit unsigned integer, which describes the number of bit that is required to store a single counter. This is used for the unpacking function as described in the Variable Counter Size section.

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

SALT is a 32-bit unsigned integer that contains the salt which was used to create the IBF.

IBF-SLICE are variable numbers of slices in an array. A single slice contains multiple 64-bit IDSUMS, 32-bit HASHSUMS and 1-64bit COUNTERS of variable size. In the network order the array of IDSUMS is first, followed by an array of HASHSUMS and ended with an array of COUNTERS (details are described in section Section 7.2). 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 is 13-byte (104-bit). The minimal number of buckets in a single IBF is 79.

To get the IDSUM field, all IDs hitting a bucket are added up with a binary XOR operation. See 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

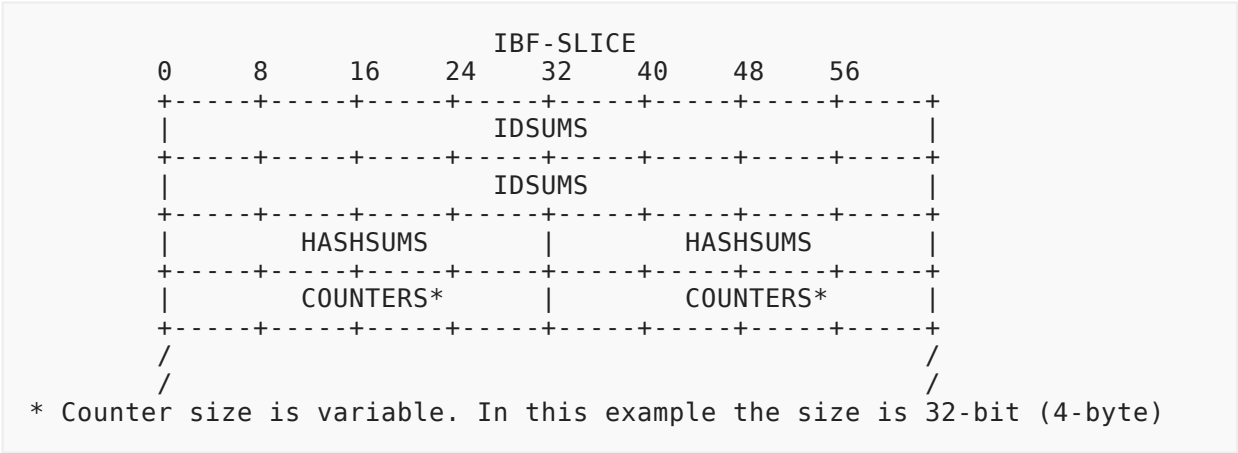


Figure 22

6.3. IBF Last

6.3.1. Description

This message indicates the remote peer that all slices of the bloom filter have been sent. 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".

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**.

6.4. 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 sent in the Differential Synchronisation Mode.

6.4.2. Structure

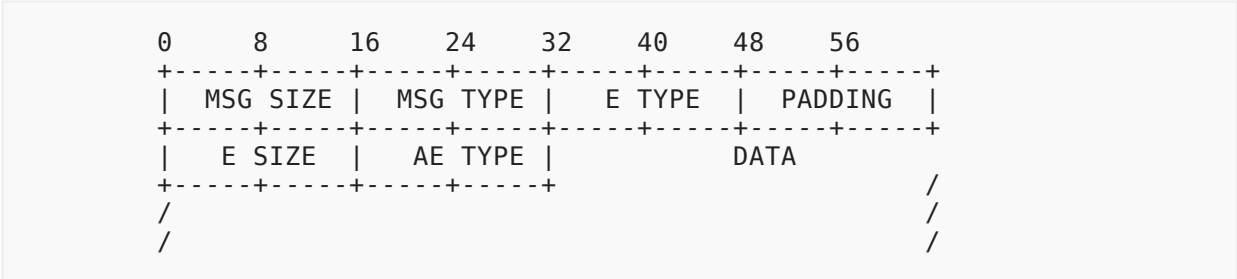


Figure 23

where:

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

MSG TYPE the type of SETU_P2P_ELEMENTS as registered in GANA Considerations in network byte order.

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

AE TYPE application specific element 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.

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

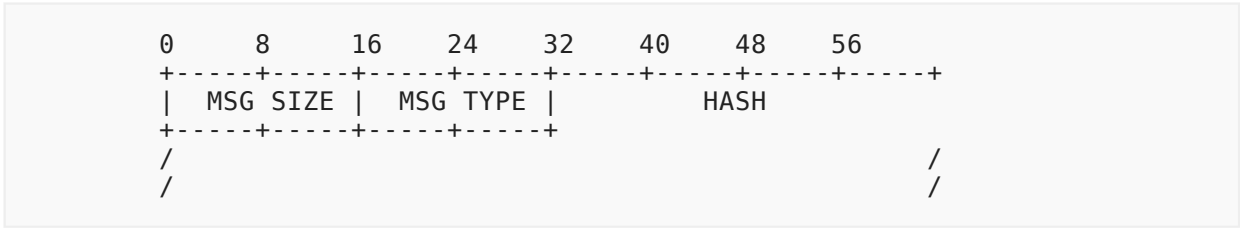


Figure 24

where:

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

MSG TYPE the type of SETU_P2P_OFFER as registered in GANA Considerations in network byte order.

HASH is a SHA 512-bit hash of the element that is requested with a *Inquiry* message.

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

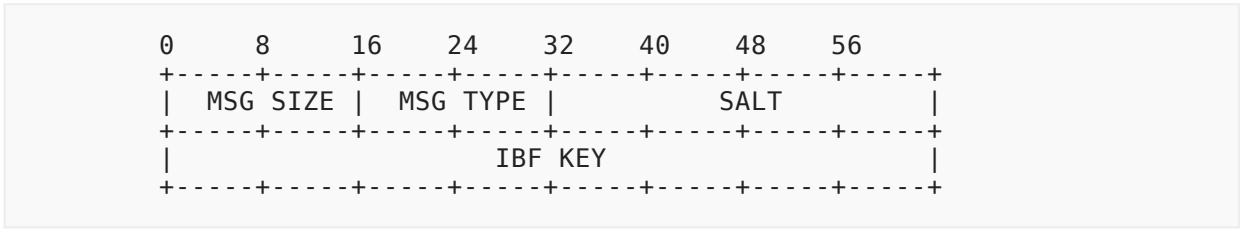


Figure 25

where:

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

MSG TYPE the type of SETU_P2P_INQUIRY as registered in GANA Considerations in network byte order.

IBF KEY is a 64-bit unsigned integer that contains the key 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

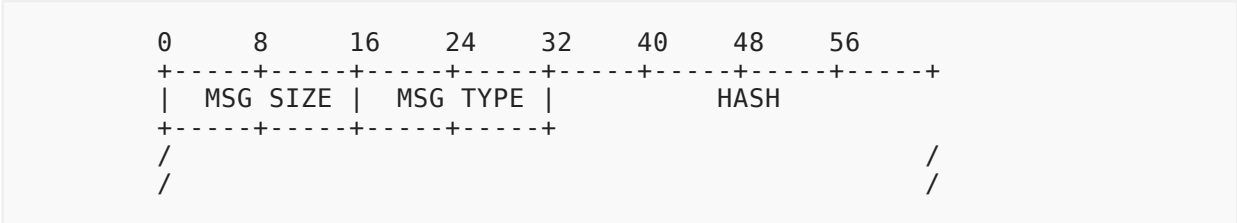


Figure 26

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 is a 512-bit Hash of the element that is demanded.

6.8. Done

6.8.1. Description

The *Done* message is sent when all *Demand* messages have been successfully satisfied and the set is complete synchronized.

This message is exclusively sent in the Differential Synchronisation Mode.

6.8.2. Structure

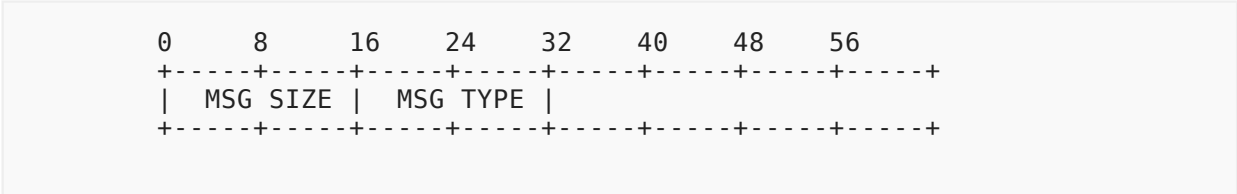


Figure 27

where:

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

MSG TYPE the type of SETU_P2P_DONE as registered in GANA Considerations in network byte order.

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

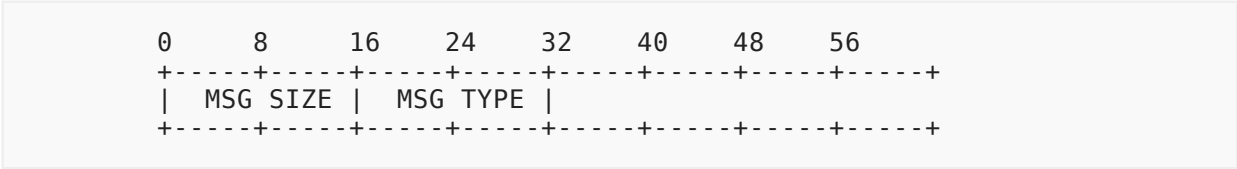


Figure 28

where:

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

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

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

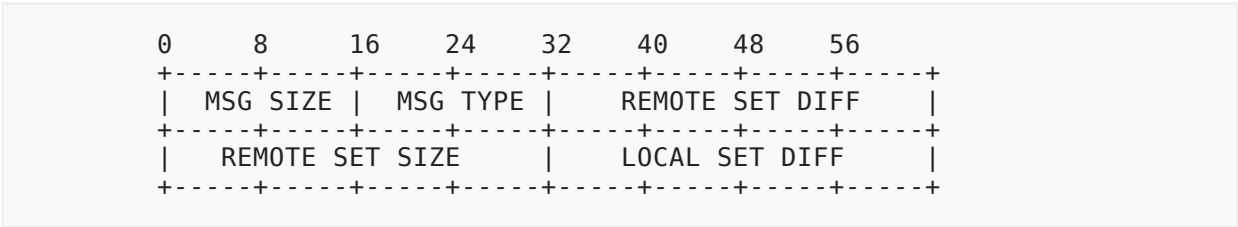


Figure 29

where:

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

MSG TYPE the type of 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

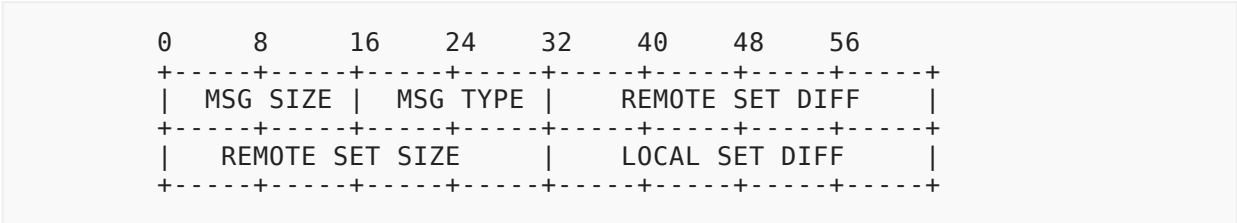


Figure 30

where:

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

MSG TYPE the type of 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 4.

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 Summermatter's work. [byzantine_fault_tolerant_set_reconciliation]

6.12.2. Structure

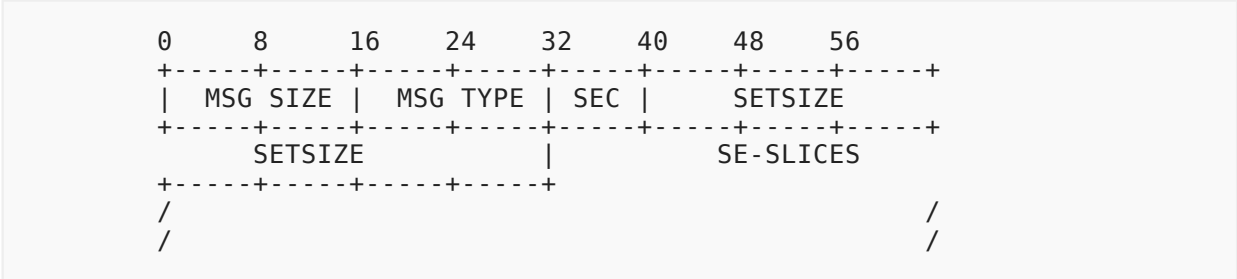


Figure 31

where:

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

MSG TYPE the type of SETU_P2P_SE as registered in GANA Considerations in network byte order.

SEC is a 8-bit unsigned integer in networkf 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 is variable in size and contains the same structure as the IBF-SLICES field in the IBF message.

6.13. Strata Estimator Compressed

6.13.1. Description

The Strata estimator can be compressed with gzip to improve performance. This can be recognized by the different message type number from GANA Considerations.

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

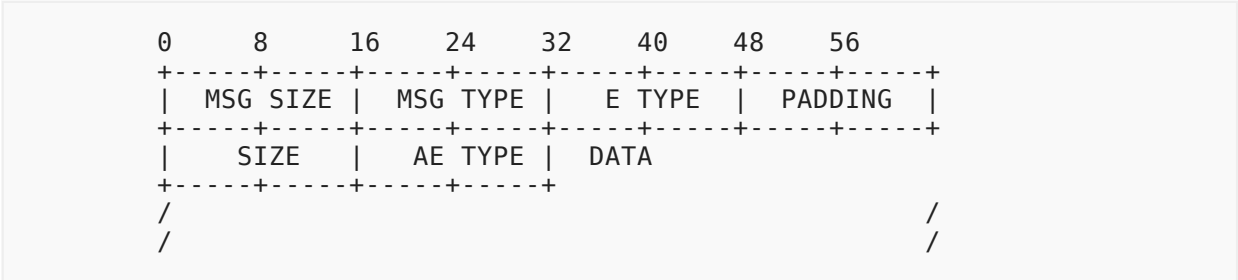


Figure 32

where:

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

MSG TYPE the type of SETU_P2P_REQUEST_FULL_ELEMENT as registered in GANA Considerations in network byte order.

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

AE TYPE application specific element 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. The function is complex. More detailed explanations can be found in the accompanying thesis. [byzantine_fault_tolerant_set_reconciliation]

The function takes as input the average element size, the local setsize, the remote setsize, the by the strata estimator calculated difference for local and remote set and the bandwith/roundtrips tradeoff. The function returns the exact Mode of Operation as output: FULL_SYNC_REMOTE_SENDING_FIRST if it is optimal that the other peer transmits his elements first, FULL_SYNC_LOCAL_SENDING_FIRST if it is optimal that the elements are transmitted to the other peer directly and DIFFERENTIAL_SYNC if the differential synchronisation is optimal.

The constant IBF_BUCKET_NUMBER_FACTOR is always 3 and IBF_MIN_SIZE is 37. The method for deriving this can be found in Summermatter's work.
[byzantine_fault_tolerant_set_reconciliation]

```
# 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 syncronisation
(always 2 or 2.5)
# IBF_MIN_SIZE = 37: The minimal size of an IBF
# MAX_BUCKETS_PER_MESSAGE: Custom value depending on the underlying
protocol
# INPUTS:
# avg_element_size: The average element size
# local_set_size: The initial local setsize
# remote_set_size: The remote setsize
# est_local_set_diff: the estimated local set difference calculated by
the strata estimator
# est_remote_set_diff: the estimated remote set difference calculated by
the strata estimator
# rtt_tradeoff: the tradeoff between round trips and bandwidth defined
by the use case
# OUTPUTS:
# returns: the decision (FULL_SYNC_REMOTE_SENDING_FIRST,
FULL_SYNC_LOCAL_SENDING_FIRST, DIFFERENTIAL_SYNC)

FUNCTION decide_operation_mode(avg_element_size,
                               local_set_size,
                               remote_set_size,
                               est_local_set_diff
                               est_remote_set_diff,
                               rtt_tradeoff)
    IF (0 == local_set_size)
        RETURN FULL_SYNC_REMOTE_SENDING_FIRST
    IF END
    IF (0 == remote_set_size)
        RETURN FULL_SYNC_LOCAL_SENDING_FIRST
    IF END

    estimated_total_diff = est_set_diff_remote + est_local_set_diff

    total_elements_to_send_local_send_first = est_remote_set_diff +
local_set_size;

    total_bytes_full_local_send_first = (avg_element_size *
total_elements_to_send_local_send_first)
                                                +
(total_elements_to_send_local_send_first * sizeof(ELEMENT_MSG_HEADER))
                                                +
(sizeof(FULL_DONE_MSG_HEADER) * 2)
                                                + RTT_MIN_FULL *
rtt_tradeoff

    total_elements_to_send_remote_send_first = est_local_set_diff +
remote_set_size

    total_bytes_full_remote_send_first = (avg_element_size *
total_elements_to_send_remote_send_first)
                                                 +
( total_elements_to_send_remote_send_first * sizeof(ELEMENT_MSG_HEADER))
                                                 +
```

```
(sizeof(FULL_DONE_MSG_HEADER) * 2)
                                                    + (RTT_MIN_FULL + 0.5) *
rtt_tradeoff
                                                    +
sizeof(REQUEST_FULL_MSG)

    ibf_bucket_count = estimated_total_diff * IBF_BUCKET_NUMBER_FACTOR

    IF (ibf_bucket_count <= IBF_MIN_SIZE)
        ibf_bucket_count = IBF_MIN_SIZE
    END IF

    ibf_message_count = ceil ( ibf_bucket_count /
MAX_BUCKETS_PER_MESSAGE);

    estimated_counter_size = MIN (
                                  2 * LOG2(local_set_size /
ibf_bucket_count),
                                  LOG2(local_set_size)
                                 )
    counter_bytes = estimated_counter_size / 8

    ibf_bytes =  sizeof(IBF_MESSAGE) * ibf_message_count * 1.2
               + ibf_bucket_count * sizeof(IBF_KEY) * 1.2
               + ibf_bucket_count * sizeof(IBF_KEYHASH) * 1.2
               + ibf_bucket_count * counter_bytes * 1.2

    element_size = (avg_element_size + sizeof(ELEMENT_MSG_HEADER)) *
estimated_total_diff
    done_size = sizeof(DONE_HEADER)
    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;

    total_bytes_diff = (element_size + done_size + inquery_size
                                  + demand_size + offer_size + ibf_bytes)
                                  + DIFFERENTIAL_RTT_MEAN * rtt_tradeoff

    full_min = MIN (total_bytes_full_local_send_first,
                             total_bytes_full_local_send_first)
    IF (full_min < total_bytes_diff)
        IF (total_bytes_full_remote_send_first >
total_bytes_full_local_send_first)
            RETURN FULL_SYNC_LOCAL_SENDING_FIRST
        ELSE
            RETURN FULL_SYNC_REMOTE_SENDING_FIRST
        END IF
    ELSE
        RETURN DIFFERENTIAL_SYNC
    IF END
```

Figure 33

7.1.2. IBF Size

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

These algorithms are described and justified in more details in the following work [byzantine_fault_tolerant_set_reconciliation].

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

FUNCTION initial_ibf_size(set_difference)
    return MAX(37, IBF_BUCKET_NUMBER_FACTOR * set_difference );
FUNCTION END

# CONSTANTS:
# IBF_BUCKET_NUMBER_FACTOR = 2: The amount the IBF gets increased if
decoding fails
# Inputs:
# 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(decoded_elements, last_ibf_size)
    next_size =(last_ibf_size * IBF_BUCKET_NUMBER_FACTOR) -
( IBF_BUCKET_NUMBER_FACTOR * decoded_elements )
    return MAX(37, next_size);
FUNCTION END
```

Figure 34

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 the following work [byzantine_fault_tolerant_set_reconciliation].

7.2. Variable Counter Size

Since the optimal number of bytes a counter in the IBF contains is very variable and varies due to different parameters. Details are described in the BSC thesis by Summermatter [byzantine_fault_tolerant_set_reconciliation]. Therefore a compression algorithm has been implemented, which always creates the IBF counter in optimal size. and thus minimizes the bandwidth needed to transmit the IBF.

A simple algorithm is used for the compression. In a first step it is determined, which is the largest counter and how many bits are needed to store it. In a second step for every counter of every bucket, the counter is stored in the bit length, determined in the first step and these are 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. As second it needs a function that compresses the counter of the IBF and thirdly a function that decompresses the counter.

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

FUNCTION ibf_get_max_counter(ibf)
    max_counter=0
    FOR bucket IN ibf
        IF bucket.counter > max_counter
            max_counter = bucket.counter

    RETURN CEILING( log2 ( max_counter ) ) # next bigger discrete number
of the binary logarithm of the max counter

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

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

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

        WHILE byte_len > 0
            byte_to_write = 0

            IF counter_bytes + store_bits >= 8
                bit_to_shift=0

                IF store_bits > 0 OR counter_bytes > 8
                    bit_free = 8 - store_bits
                    bit_to_shift = counter_bytes - bit_free
                    store = store << bit_free

                byte_to_write = (( counter >> bit_to_shift) | store) &
0xFF
                bit_to_shift -= 8 - store_bits
                counter = counter & (( 1 << counter_bytes ) - 1)
                store = 0
                store_bits = 0

            ELSE
                IF 0 == store_bits
                    store = counter
                ELSE
                    store = (store << counter_bytes) | counter
```

```
                store_bits = store_bits + byte_len
                byte_len = 0
                BREAK

            buffer[byte_ctr] = byte_to_write
            byte_ctr++
        NEXT_BUCKET

    # Write the last partial compressed byte to the buffer
    buffer[byte_ctr] = store << (8 - store_bits)
    byte_ctr++

    RETURN buffer

# INPUTS:
# ibf: The IBF
# offset: The offset which defines the starting point from which bucket
the compress operation starts
# count: The number of buckets in the array that will be compressed
# counter_bit_length: The bit length of the counter can be found in the
ibf message in the ibf_counter_bit_length field
# packed_data: 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, counter_bit_length,
packed_data)
    store = 0
    store_bits = 0
    byte_ctr = 0
    ibf_bucket_ctr = 0

    number_bytes_read = CEILING((count * counter_bit_length) / 8)

    WHILE ibf_bucket_ctr <= (count -1)
        byte_to_read = packed_data[byte_ctr]
        byte_ctr++
        bit_to_pack_left = 8

        WHILE bit_to_pack_left >= 0

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

            IF  ( store_bits + bit_to_pack_left ) >= counter_bit_length
                bit_use = counter_bit_length - store_bits

                IF store_bits > 0
                    store = store << bit_use

                bytes_to_shift = bit_to_pack_left - bit_use
                counter_partial = byte_to_read >> bytes_to_shift
                store = store | counter_partial
                ibf.counter[ibf_bucket_ctr] = store
```

```
                byte_to_read = byte_to_read & (( 1 << bytes_to_shift ) -
1)

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

            ELSE
                store_bits += bit_to_pack_left

                IF 0 == store_bits
                    store = byte_to_read
                ELSE
                    store = store << bit_to_pack_left
                    store = store | byte_to_read
                BREAK
```

Figure 35

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) % 64
    return (value >> s) | (value << (64 - s))
```

Figure 36

Performance study and details about the reasoning for the used methods can be found in the work of Summermatter. [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 cpu 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 failed decoding attempts in the protocol, it is possible to introduce a threshold for max failed reconciliation ties.

The formal format of all 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.

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. This can be implemented by a simple counter which terminates the operation after a predefined number of switches. The number of switches needs to be defined in such a way that it is very unprobable that more switches are required an the malicious intend of the other peer can be assumed.

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

8.1. General Security Check

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

8.1.1. 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)
        RETURN FALSE
    IF END
    IF (lec + lsd > UPPER_BOUND)
        RETURN FALSE
    IF END
    IF (rec < LOWER_BOUND)
        RETURN FALSE
    IF END
        RETURN TRUE
FUNCTION END
```

Figure 37

For the byzantine upper bound checks to function flawlessly, it needs to be ensured that the estimates of the set size difference added together never exceed the set byzantine upper bound. This could for example happen if the strata estimator overestimates the set difference.

8.1.2. 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.

The following function is executed every time a message is processed. The array allowed_state, which contains a list of allowed messages for the current state of the application.

```
# Input:
# allowed_states: A array containing all states in which the message can
be received
# state: The state in which the protocol is in
# Output:
# Returns TRUE if message is valid in state and FALSE if not

FUNCTION check_valid_state (allowed_states, state)
    FOR (allowed_state in allowed_states)
        IF (allowed_state == state)
            RETURN TRUE
        END IF
    FOR END
    RETURN FALSE
FUNCTION END
```

Figure 38

8.1.3. 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.

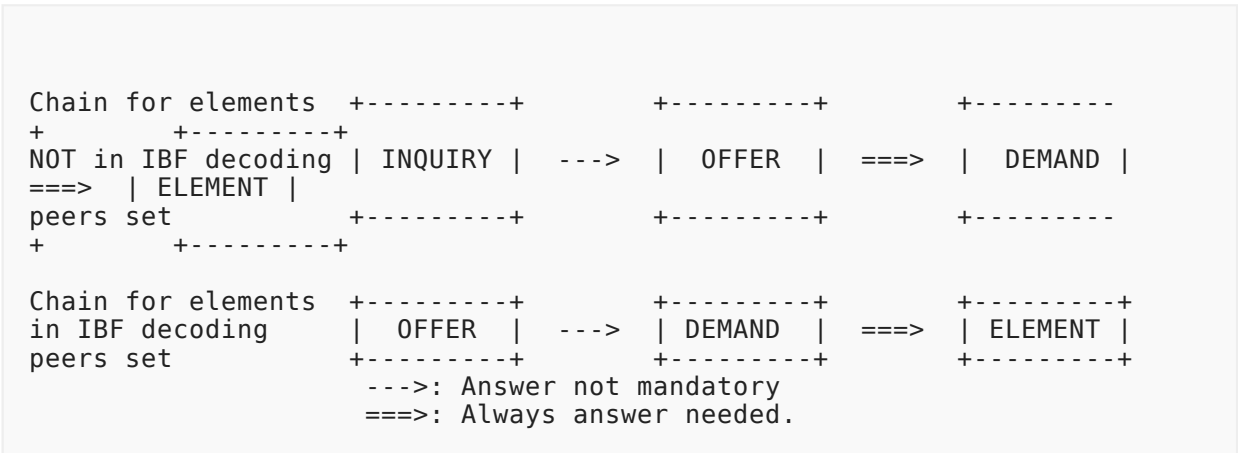


Figure 39

A possible implementation of the check in pseudocode could look as follows:

```
# ValidStates:
# The following message states are used to track the message flow.
# - NOT_INITIALIZED: Fresh initialized value
# - SENT: Element has been sent
# - EXPECTED: Element is expected
# - RECEIVED: Element is received

# Function to initialize new store
# Output:
# Returns empty store
FUNCTION initialize_store()
    RETURN {}
FUNCTION END

# Function to initialize a store element
# Output:
# Returns an empty element for the store
FUNCTION initialize_element()
    RETURN {
        INQUIRY: NOT_INITIALIZED,
        OFFER: NOT_INITIALIZED,
        DEMAND: NOT_INITIALIZED,
        ELEMENT: NOT_INITIALIZED
    }
FUNCTION END

# Function called every time a new message is transmitted to other peer
# Input:
# store: Store created by the initialize_store() function
# message_type: The message that was sent type e.g. INQUIRY or DEMAND
# hash: The hash of the element which is sent
# Output:
# Returns TRUE if the message flow was followed, otherwise FALSE
FUNCTION send(store, message_type, hash)

    IF NOT store.contains(hash)
        store_element = initialize_element()
    ELSE
        store_element = store.get(hash)
    END IF

    CASE based on message_type
        CASE INQUIRY
            IF store_element.INQUIRY == NOT_INITIALIZED
                store_element.INQUIRY = SENT
            ELSE
                RETURN FALSE
            END IF
        CASE OFFER
            IF store_element.OFFER == NOT_INITIALIZED
                store_element.OFFER = SENT
                store_element.DEMAND = EXPECTED
            ELSE
                RETURN FALSE
            END IF
        CASE DEMAND
            IF store_element.DEMAND == NOT_INITIALIZED AND
```

```
             (store_element.INQUIRY == SENT OR
              store_element.INQUIRY == NOT_INITIALIZED)
                store_element.DEMAND = SENT
                store_element.ELEMENT = EXPECTED
            ELSE
                RETURN FALSE
            END IF
        CASE ELEMENT
            IF store_element.ELEMENT == NOT_INITIALIZED AND
              store_element.OFFER == SENT
                store_element.ELEMENT = SENT
            ELSE
                RETURN FALSE
            END IF
        DEFAULT
            RETURN FALSE
        CASE END
   store.put(hash,store_element)
   RETURN TRUE
FUNCTION END

# Function called every time a new message is received from the other
peer
# Input:
# store: Store created by the initialize_store() function
# message_type: The message that was received type e.g. INQUIRY or DEMAND
# hash: The hash of the element which is received
# Output:
# Returns TRUE if the message flow was followed, otherwise FALSE
FUNCTION receive (store, message_type, hash)
    IF NOT store.contains(hash)
        store_element = initialize_element()
    ELSE
        store_element = store.get(hash)
    END IF

    CASE based on message_type
        CASE INQUIRY
            IF store_element.INQUIRY == NOT_INITIALIZED
                store_element.INQUIRY = RECEIVED
            ELSE
                RETURN FALSE
            END IF
        CASE OFFER
            IF store_element.OFFER == NOT_INITIALIZED
                store_element.OFFER = RECEIVED
            ELSE
                RETURN FALSE
            END IF
        CASE DEMAND
            IF store_element.DEMAND == EXPECTED AND
               store_element.OFFER == SENT
                store_element.DEMAND = RECEIVED
            ELSE
                RETURN FALSE
            END IF
        CASE ELEMENT
            IF store_element.ELEMENT == EXPECTED AND
```

```
              store_element.DEMAND == SENT
                store_element.ELEMENT = RECEIVED
            ELSE
                RETURN FALSE
            END IF
        DEFAULT
            RETURN FALSE
        CASE END
   store.put(hash,store_element)
   RETURN TRUE
FUNCTION END

# Function called when the union operation is finished to ensure that
all demands have
# been fulfilled
# Input:
# store: Store created by the initialize_store() function
# Output:
# Returns TRUE if all demands have been fulfilled otherwise FALSE
FUNCTION check_if_synchronisation_is_complete(store):
    FOR element in store.getAll()
        IF element.ELEMENT == EXPECTED OR
         element.DEMAND == EXPECTED
            RETURN FALSE
        IF END
    FOR END
    RETURN TRUE
FUNCTION END
```

Figure 40

This is based on the work of Summermatter. More details can be found in his thesis [byzantine_fault_tolerant_set_reconciliation].

8.1.4. Limit Active/Passive Decoding changes

The limitation of the maximum allowed active/passive changes during differential synchronisation is key to security. By limiting the maximum rounds in differential synchronisation an attacker can not waste unlimited amount of resources by just pretending an IBF does not decode.

The question after how many active/passive switches it can be assumed that the other peer is not honest, depends on many factors and can only be solved probabilistically. In the work of Summermatter [byzantine_fault_tolerant_set_reconciliation] this is described in detail. From this work it is concluded that the probability of decoding failure is about 15% for each round. The probability that there will be n active/passive changes is given by 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. It is reasonable that a maximum of 30 active/passive changes should be set.

8.1.5. 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, in Summmermatters paper [byzantine_fault_tolerant_set_reconciliation] a formula 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.

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 syncronisation is plausible and FALSE otherwise

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

    IF (FULL_SENDING == state)
        IF (rd > 0)
            RETURN FALSE
        IF END
    IF END

    IF (FULL_RECEIVING == state)
        IF (0 <= rs)
            rs = 1
        IF END
        base = (1 - ( rs / (lis + rs)))
        exponent = (rd - (rf * (lis/rs)))
        value = POWER(base, exponent)
        IF ((value < security_level_lb) || (value > SECURITY_LEVEL)
            RETURN FALSE
        IF END
    IF END
    RETURN TRUE
FUNCTION END
```

Figure 41

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 not more or less elements are received, as the other peer has committed to in the beginning of the operation. The plausibility should also be checked with an algorithm as described in Section 8.1.5.

Full Done When receiving the *Full Done* message, it is important to check that not less elements are received as the other peer has committed to send. If the sets differ, a resynchronisation is required. The number of possible resynchronisation MUST be limited, to prevent resource exhaustion attacks.

8.2.3. Expecting IBF Last

Security considerations for received messages:

IBF No special safety measures are necessary in this state. The maximum of IBF messages should be limited to a reasonable amount.

8.2.4. Active Decoding

In the **Active Decoding** state it is important to prevent an attacker from generating and passing an unlimited amount of IBFs, that do not decode or even worse, generate an IBF constructed to send the peers in an endless loop. To prevent an endless loop in decoding, a loop detection should be implemented. The simplest solution would be to prevent decoding of more than a given amount of elements. A more robust solution is to implement a algorithm that detects a loop by analyzing past partially decoded IBFs. This can be achieved by saving the hash of all prior partly decoded IBFs hashes in a hashmap and check for every inserted hash, if it is already in the hashmap.

If the IBF decodes more or less elements than are plausible, the operation MUST be terminated. The upper and lower threshold for the decoded elements can be calculated with the peers set sizes and the other peer committed set sizes from 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, doubled, not in order or unexpected messages.

Element If an element that never has been requested by a demand or is received double, 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, doubled, not in 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, doubled, not in order or unexpected messages.

Done The *Done* message is only received, if the IBF has been 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 offers and demands have been answered, the operation MUST be terminated. If the sets differ, a resynchronisation is required. The number of possible resynchronisation MUST be limited to prevent resource exhaustion attacks.

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, doubled, not in 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.

In case of compressed strata estimators the decompression algorithm needs to be protected against decompression memory corruption (memory overflow).

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 or less elements are received, as the other peer has committed to in the beginning of the operation. The plausibility should also be checked with an algorithm as described in Section 8.1.5.

Full Done When the *Full Done* message is received from the remote peer, all elements, that the remote peer has committed to, need to be received, otherwise the operation MUST be terminated. After receiving the *Full Done* message, future elements MUST NOT be accepted. If the sets differ, a resynchronisation is required. The number of possible resynchronisation MUST be limited to prevent resource exhaustion attacks.

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. In this moment the peer MUST wait for all open offers and demands to be fulfilled, to prevent retransmission before switching into active decoding operation mode. A switch into active decoding mode MUST only be permitted for a predefined number of times as described in Section 8.1.4

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 sending and receiving of *Inquiry* messages should always be protected with an Message Flow Control to secure the protocol against missing, doubled, not in 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 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, doubled, not in 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 Summermatter's work [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
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
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
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
PROBABILITY_FOR_NEW_ROUND*    | 0.15       | The probability for a IBF
decoding failure in the differential
                                             synchronisation mode
DIFFERENTIAL_RTT_MEAN*        | 3.65145    | The average RTT that is
needed for a differential synchronisation
MAX_IBF_SIZE                  | 1048576    | Maximal number of buckets in
an IBF
AVG_BYTE_SIZE_SE*             | 4221       | Average byte size of a single
strata estimator
VALID_NUMBER_SE*              | [1,2,4,8]  | Valid number of SE in
```

Figure 42

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 other peer, 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 Slice.
 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 43

11. Contributors

The original GNUnet implementation of the byzantine fault tolerant set reconciliation protocol has mainly been written by Florian Dold and Christian Grothoff.

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>.

[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:

```
number_of_buckets_per_element: 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 44

OUTPUT:

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

Figure 45

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 46

OUTPUT:

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

Figure 47

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 48

OUTPUT:

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

Figure 49

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