Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 9 November 2025

T. Przygienda
T. Li
Juniper Networks

8 May 2025

ISIS Hierarchical SNPs
draft-prz-lsr-hierarchical-snps-00

Abstract

The document introduces an optional, new type of SNPs called Hierarchical SNP (HSNP) that can compress the traditional CSNPs exchange into a variant of merkle tree, hence allowing to support very large databases and adjacency numbers. Such an approach should lead in case of inconsistencies to much faster re-synchronization since only a subset of packets compared to full scale CSNP exchange is necessary to correct the entropy present.

Table of Contents

1. Introduction

The document introduces an optional, new type of SNPs called Hierarchical SNP (HSNP) that can compress the traditional CSNPs exchange into a variant of merkle tree [MERKLE] and skip lists [SKIP], hence allowing to support very large databases and adjacency numbers. Such an approach should lead in case of inconsistencies to much faster re-synchronization since only a subset of packets compared to full scale CSNP exchange is necessary to correct the entropy present.

Although the scheme can be applied recursively to the point where a single merkle hash represents the whole database, for practical purposes a two level tree allows compression for LSDB sizes of the 1E5 order and hence the document limits itself in examples to such magnitudes. Further considerations mentioned in Section 6 seem to make this limit practically prudent.

We call CSNP entries of LSPs the zero level merkle hash (where we basically use lsp id, seq# and checksum as "hash") and the hash summarizing a set of those first level merkle hash and so on recursively.

2. Suggested Dynamic Leaf Partitioning

Practically speaking, the most interesting problem is the correct subdivision of the database into first level collection of leaf nodes (CSNP entries) that on one hand can provide a good compression, on the other hand the subdivision changes as little as possible. Otherwise the neighbors receiving the information may have to recompute the hashes rather than relying on a cache representing its own merkle tree. The subdivision should also produce enough "bins" no matter the distribution of the fragment IDs in the network. This is important to prevent such things as packing of all the fragments into a single checksum when e.g. a hash function degenerates. And ideally, a hash mismatch should produce not more than a single packet or two with lower level checksums or CSNPs to optimize re-convergence while minimizing amount of packets exchanged.

To start with, in IS-IS networks we can fit into the prevailing 1500 bytes somewhat less than 100 of zero level entries for the foreseeable future. This is the consequence of CSNP entries consuming LSP ID + Fragment + Seq# + CSUM + Lifetime length which amounts to 8 + 2 + 2 + 2 = 14 bytes each.

Subsequently, first and higher level hashes will occupy (as shown in Figure 2) the length of 7 + 7 + 4 = 18 bytes per hash and hence around 80 of those hashes fit into a packet.

Those considerations lead to the suggested partitioning and packing scheme below.

To start with, as stated, it is desirable to produce one packet on a miss on the merkle hash of first level leaf and hence such leaves will pack initially 80 LSP fragments with exceptions following later. We do not try to maximize the "initial packing". LSDB may grow and to maximize the chances of same "leaf packing" on both sides of an adjacency, even during flooding transitions, some "slack" is advisable.

At higher levels, to begin with 60 hashes should be summarized by a higher level hash for the same reason.

The packing will always put all fragments of a system into same leaf (which of course can exceed the advisable 80 fragments sometimes) and a first order leaf will be considered "full" if addition of next System ID fragments would exceed this size (except obviously, when the leaf is empty).

2.1. Repacking

During flooding transitions the databases on different nodes may be obviously different for some period of time. As stated, even in such situations it is beneficial for efficient implementations using caching to agree on both sides on the ranges of the Merkle hashes advertised. Or alternately, have a fast way to reconstruct the internal merkle hash for a range. E.g., in the suggested packing scheme, since the nodes will always agree on the "system-ID" boundary, a merkle hash per system ID can be easily kept and if the received ranges do not agree with cached ranges the necessary merkle hash recomputed very quickly. As example, in first level the worst case of 2 leaves containing ~80 nodes each and "intersecting" the cached internal ranges an implementation will have to merkle hash about 160 hashes (one per System ID) to validate the received value.

Obviously, it is still highly preferred for the ranges on advertised merkle hashes to agree on their system ID ranges precisely and it is of most importance at the top level. Under stable conditions those are the merkle hashes reducing the CSNP exchange to minimal amount of packets and processing effort.

To settle on the same ranges in HSNPs an implementation of the suggested packing should let a leaf that drops under 50% occupancy "start robbing" system IDs from "left" of the next leaf until the current leaf meets the "full condition". This is of course a recursive action that may ultimately generate less leaves, remove some and in a recursive fashion lead to the same "greedy robbery from the left" in the next level up. The "left" is colloquial here for starting with lowest System IDs under normal sorting criteria. On the other end of the spectrum a leaf that holds more than 150% of usual capacity (i.e. 80 * 1.5 LSPs or 60 * 1.5 Hashes) should be preferably split into two leaves unless it holds a single System ID with more than 80 fragments. Splitting the leaves may cause a repacking at a higher level again in a recursive fashion.

The rebalancing of ranges to agree across all nodes and hence reduce hashing load is a trade-off in terms of possibly large recomputation vs. suffering a penalty of recomputing some hashes on disagreeing

ranges on every exchange. Other solutions are of course possible such as internal caches that keep the recomputed hashes for the neighbor’s ranges.

Precise splitting/merging algorithms agreed upon increase the likelihood of nodes ending up on precisely same ranges. A possibly simpler idea to discuss is to simply "repack" the whole thing on some balance violations or periodically. Another idea is to simply use ISIS fragment sliding but this may lead in worst case to first level checksumming a single fragment over time.

Overall, different partitioning and packing approaches are possible but if system ID as natural partition is not used, this will likely change the packet format since the partition boundaries will necessarily reflect which of the fragments are covered by the hashes. Although, given that ordering of fragments has to be preserved it is hard to imagine anything else but start and end consisting of fragment IDs.

3. Fast, incremental, self-inverse hashing function

Since we need to generate massive amount of hashes over sets of ever-changing fragments it is very desirable to use a specialized hash function that provides means for very fast incremental adjustment of hash result on fragments arriving, aging out or changing their checksums. Also, splitting or merging of leaves should allow to generate first and higher order hashes very quickly.

Deeper considerations on such hashes can be found in [HASHES] but our design space is simplified due to irrelevance of security involved.

We use a large prime number constant to prime all hashes.

The hash function is a very fast XOR operation which incorporates the system IDs, checksums and numbering of fragments. It is 64 bits long for simplicity. Symbol << is used for rotation (not shift) to the left.

A hash of a first order leaf is nothing but (where all used values are converted to 64 bit values first with any remaining high order bits zero’ed out) the usual XOR(hash, (system-id-of-fragment + 1) << fragment#) followed by XOR(hash, (fragment# + 1) << 32 | (fragment-seq# + 1) << 16 | (fragment-csum + 1)) for all contained fragments . Obviously, such an operation can be easily reversed to yield the previous hash with the fragment removed which allows for very fast removal, addition of fragments and updates of the relevant fields.

Appendix A includes reference code of the introduced hashing.

Example of adding multiple fragments and removing those (in different order) which yields ultimately the same initial hash is found below:

```
Initial Hash: CDDC72CF
Adding Fragment 0100.0000.0000.00 #: 0 CSUM: 1 SEQNR: 0
Hash after Addition 10001CDDD72CC
Adding Fragment 0100.0000.0000.00 #: 1 CSUM: 2 SEQNR: 1
Hash after Addition 1010003CDDF72CE
Adding Fragment 0200.0000.0000.00 #: 1 CSUM: 3 SEQNR: 2
Hash after Addition 3010001CDDC72CB
Removing Fragment 0100.0000.0000.00 #: 0 CSUM: 1 SEQNR: 0
Hash after Removal 3000000CDDD72C8
Removing Fragment 0100.0000.0000.00 #: 1 CSUM: 2 SEQNR: 1
Hash after Removal 2000002CDDF72CA
Removing Fragment 0200.0000.0000.00 #: 1 CSUM: 3 SEQNR: 2
Hash after Removal CDDC72CF
```

Figure 1

A hash of second order leaf is nothing but XOR of all the hashes of all the contained first order leaves. Again, this allows to update the hash when adding, removing a leaf or changing its checksum in a very fast and simple manner. The packing of second order leaves is determined by how many first order hashes can be fitted into a PDU normally and based on Section 4 we can set a second order leaf size to 70. This will cause a mismatch in the hash of a 2nd order leaf to advertise roughly a PDU full of first order leaves.

Ultimately, and fairly obviously, third order hash uses second order hash logic to keep its hash. This all means that every time a first order leaf changes the contained system IDs for some reason the merkle hashes will have to be readjusted recursively in according 2nd and third order leaves. This is in itself nothing particular since _any_ change on first order leaf hash forces change on second order and consequently third order leaf hash. This is how Merkle trees work after all.

A first example will serve well here. We limit ourselves in the examples to consideration of a LSDB with 512 nodes with system identifiers of 1000.0000.00 <2 digits node-id> each holding 32 fragments numbered 0 to 31. We leave the uneven node identifiers out to have some "holes" in the numbering to hit some corner cases in further examples. We disregard the pseudo node byte as simply another byte of system identifier since it does not contribute further details to the scheme and use value 0 in further text.

In a stable state we can expect the following 128 first order leaves (each holding 2 systems worth of fragments) generating three packets. And as logical consequence two single second order leaves. First of the three first order packets will look roughly like this

```
      ...

+-------------------------------------------+
|  Start System ID: 0000.0000.0000.00       |
+-------------------------------------------+
|  End System ID:   0000.0000.00A0.00       | // 80 hashes for 160 systems
+-------------------------------------------+
|  HSNP Level: 0                            |
+-------------------------------------------+
|  Start System ID: 1000.0000.0000.00       |
+-------------------------------------------+
|  End System ID:   1000.0000.0002.00       | // 64 fragments over 2 systems
+-------------------------------------------+
|              Merkle Hash                  |
+-------------------------------------------+
..
|  Start System ID: 1000.0000.008E.00       |
+-------------------------------------------+
|  End System ID:   1000.0000.00A0.00       |
+-------------------------------------------+
|              Merkle Hash                  |
+-------------------------------------------+
```

Figure 2

The 2nd order packet will look like this

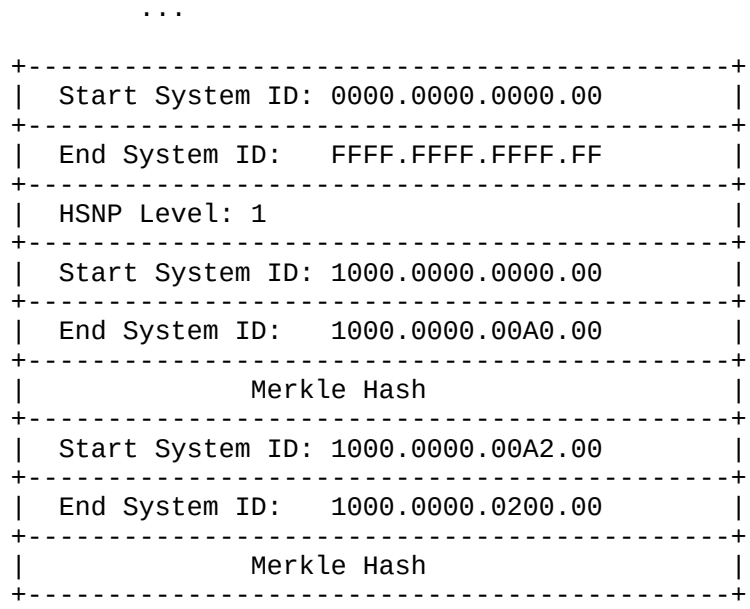
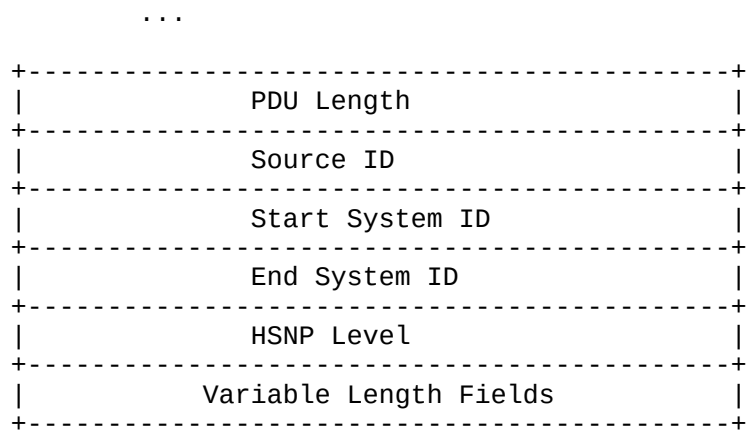


Figure 3

4. HSNP PDU Format

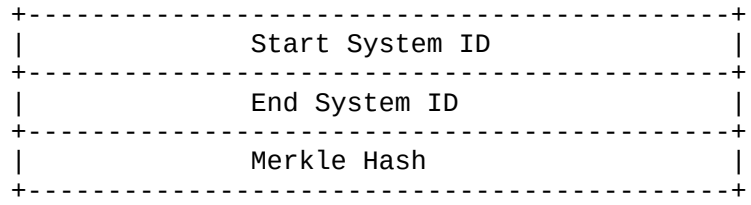
HSNP PDU Format follows closely CSNP format where instead of CSNP entries the according merkle hashes are propagated.



Start and End System IDs are or the usual ID Length + 1 byte length and indicate, just like CSNP do, the range that the HSNP covers.

HSNP level consists of 1 byte of level with 0 indicating "first level hashes".

The variable length fields are a sorted sequence of covered ranges in the following format



End System ID LSPs are included in the hash.

Merkle hash consists of 4 bytes of the 64-bit computed hash with upper and lower 4 bytes XOR'ed together.

This makes an entry in typical deployment scenarios 7 + 7 + 4 = 18 bytes long.

5. Procedures

5.1. Maximum supported level negotiation in IIH

IIH of nodes supporting this extension MUST include in IIH a new TLV that will indicate support for reception of HSNPs. Additionally, the TLV will carry the maximum level of HSNPs that the node will advertise. Each node MUST pick the minimum of advertised levels on the adjacency and use that for the HSNPs it will advertise. All nodes on the adjacency MUST advertise the TLV on their IIHs, otherwise HSNPs are not used.

5.2. Advertising HSNPs

Advertising normal CSNPs is replaced with advertisement of HSNPs at highest negotiated level. Under normal stable network condition this will be enough to maintain database integrity across all nodes with a minimum of transmission and processing.

In case a node receives HSNPs where the merkle hash ranges are not the same, the node MUST either to compute and verify the hashes over the ranges indicated or disaggregate the overlapped ranges to a lower level HSNPs. The disaggregation is less preferred since in case of range mismatches over all levels and both sides using this strategy this can lead to a 'ping-pong' ending with CSNPs ultimately.

A node receiving an HSNP where the hash received does not match on recomputation or comparison the result on its own LSDB SHOULD send immediately HSNPs of one level below with the Merkle hashes for the ranges where the hash mismatch was detected. Alternately, a node MAY choose to immediately send according CSNPs, PSNPs or flood the LSPs that have been detected as not matching the merkle hashes. Sending CSNPs may be preferable if the mismatch covers relatively few LSPs.

In case a node detects that it holds Merkle hashes for LSPs that are not covered by the received HSNP (start and end range and "holes" between ranges can be used to detect that condition), it MUST trigger the same behavior as triggered by CSNP with this condition, i.e. flood the missing LSPs.

6. Further Considerations

6.1. Impact of Packet Losses

Many levels of aggregation will be more susceptible to packet losses since each loss covers a much larger part of the LSDB. Additionally, during disaggregation where multiple HSNP levels must be triggered, the loss will force a delay until the higher level HSNP is regenerated.

6.2. Decompression and Caching/Comparison Optimizations

As mentioned above a node may apply many strategies to speed up decompression. LSPs missing in HSNPs as not covered by ranges are clearly "missing in action" and can be reflooded, small ranges where merkle mismatched can generate CSNPs, PSNPs or lead to flooding.

Caching of hashes can be applied at many levels since the merkle hashes suggested here are easily computed, even if certain elements must be removed from them, so e.g. on receiving a range <A -- B> while the node already holds < A -- B & next-after-B> can be simply adjusted by removing ’next-after-B’ merkle hash defined in this document from the cached result.

7. Security Considerations

TBD

8. IANA Section

TBD

9. Contributors

TBD

10. Acknowledgement

The discussions about "compressing CSNPs" go very long way back, allegedly to the times Radia Perlman was stalking the halls with insomniac Dave Katz. Recent efforts to push the scale of the protocol to new heights made the effort worth the effort (pun intended) to codify it into a standardized, practical engineering solution.

11. References

11.1. Normative References

[HASHES] Phan, C.-W., "Security considerations for incremental hash functions based on pair block chaining", Computers and Security 25 , 2006.

[MERKLE] Merkle, R.C., "A Digital Signature Based on a Conventional Encryption Function", Advances in Cryptology CRYPTO ’87 , 1988.

[SKIP] Phan, C.-W., "Skip lists: A probabilistic alternative to balanced trees", Communications of the ACM , 1990.

11.2. Informative References

Appendix A. Reference Implementation of Fragment Hashing

```
<CODE BEGINS>
#[derive(Eq, PartialEq, Hash)]
struct IncrementalHash(u64);

const SEED: u64 = 0xCDDC72CF;

impl Default for IncrementalHash {
    fn default() -> Self {
        IncrementalHash(SEED)
    }
}

impl IncrementalHash {
    // takes arbitrary number of byte arrays
    fn hash_in(&mut self, add: &[u64]) {
        for f in add {
            self.0 ^= f;
        }
    }

    fn hash_in_fragment(&mut self, sid: &SystemID, fragment_number: FragmentNr,
                        fragment:  &Arc<FragmentContent>) {
        let len = 8.min(sid.id.len());
        let mut sidc: [u8; 8] = [0; 8];
        sidc[(8-len) ..].copy_from_slice(&sid.id);
        sidc.rotate_left(fragment_number as _);

        self.hash_in(&[u64::from_be_bytes(sidc) + 1]);

        let frnr = fragment_number as u64 + 1;
        let frsn = fragment.seqnr as u64 + 1;
        let frcs = fragment.csum as u64 + 1;

        let hin = frnr << 32 | frsn << 16 | frcs;
        self.hash_in(&[hin]);
    }
}
<CODE ENDS>
```

Authors' Addresses

Tony Przygienda
Juniper Networks
Email: prz@juniper.net

Tony Li
Juniper Networks

Email: tli@juniper.net