Routing Graph Reconcilation 0.0.2

Piotr Mikulski

1 Overview

In P2P networks, like NEAR Protocol [1], each node stores a graph of nodes and edges representing peers and connections between them. We will call such graph a **routing graph**. Whenever a peer joins the network or peer reconnect, they exchange their routing tables. The process of exchanging tables involves adding edges, which are present in one's peers view of the graph, but not in the others. For the same of simplicity we will describe only the process of adding new edges.

That operation gets more expensive as **routing graph** grows larger. Exchanging full routing tables on every reconnect would be expensive and we would like to avoid it.

In this article, we will discuss ways of exchanging routing tables by echanging the amount of information proportional to size of edges that need to be added.

2 Assumptions

Total number of edges in the graph won't be much bigger than order of 1 milion. Currently it's edge takes 360 bytes, it takes 2ms to very each edge. So sending that number of edges would require 360mb, and veryfing them would take half an hour on a single thread.

3 Definitions

3.1 Routing Graph

Graph, where peers are represented by nodes, and edges by connections between them.

3.2 Node

Each node is uniquely defined by it's public key.

```
pub struct PeerId {
    /// Peer is defined by it's public key
    public_key: PublicKey,
}
```

3.3 Edges

Each edge is defined by pair of peer ids and signatures from both peers. This data structure can be as large as 360 bytes in case of NEAR Protocol network. Therefore we would like to minimize the number of edges we transfer.

```
pub struct Edge {
    /// Since edges are not directed `peer0 < peer1` should hold.
    pub peer0: PeerId,
    pub peer1: PeedId,
    /// Signature from parties validating the edge.
    These are signature of the added edge.
    signature0: Signature,
    signature1: Signature,
    /// some other data
    /// ...
}
```

3.4 IBF - Inverse Bloom Filter

Invertible Bloom Filter (IBF) is a type of bloom filter, which can be used to simultaneously calculate D_{A-B} and D_{B-A} using $O(d)$ space. This data structure encodes sets in such a way that given two encodings of two different sets, you can recover symetric difference of those two sets. Detailed analysis can be found at ESRwPC [2].

3.4.1 important properties

- Two IBFS structures can be added/substacted from each other creating a new structure representing sum/difference of sets.
- It can be shown that given two IBF structures of size g, you can recover with high probability up to $(2/3) * g$ elements if g big enough. (above 1000) [2]
- IBF is a propabilisting approach is can fail, we have to that into account. For example by sending IBF encoded with different hash function or bigger size.
- decoding can lead to partial result. We can recover some edges even during failure.
- sizeof IBF with 2^k elements is $2^k(s_i + s_h)$, where s_i is size of item, s_h is size of hash.

4 Algorithm overview

There are many ways in which synchronization of **routing graphs** can be done. You have to consider a few factors: number of bytes send, number of round trips, computation time, total time. Later I'll discuss each variant, it's strong and weak points, and one which one I think we should use. In general the algorithm I'm proposing in as follows:

1. Using a probabilisting approach estimate number of edges that differ. Let call it d_e . (details in section 5)
2. Choose k such that IBF with 2^k elements would be enough to decode d_e elements.
3. If sending computing/sending/decoding IBF of size 2^k is more expensive than sending all elements, send all elements and exit
4. Otherwise send IBF of size 2^k if fully successful exit, otherwise increase k by 1, and go back to step 3. If recovery was partially successful add recovered edges.

4.1 Analysis

We have to send each edge in mutual difference at least once. We have to send at least $b_o = d * s_e$ bytes, where s_e is size of each edge.

Assuming we start with $k = 0$, it can be proven that total number of IBF buckets sent won't be bigger than $6d$ (TODO Proof 1). Therefore the total number of bytes won't be bigger than $6(s_e + s_h) * d$ plus metadata, s_h is the size of hash. So the total number of bytes sent won't be bigger than $6.13b_o$.

4.2 Improvement

We can improve the algorithm above. We know that each edge takes 360 bytes. Sending edges directly can be expensive. Instead for each edge e we can compute 8 bytes hash h_e. Instead of computing set of egdes are missing, we can compute set of hashes of edges which are missing. Given list of hashes of edges that differ, we can then send list of edges that differ. It can be shown that the total bytes send won't be higher than $6(s_h + s_h) * d + s_e * d = (96 + s_e) * d$ (TODO Proof 2). So the solution is at most 1.26 times worse than minimum. Using hashes could introduce potencial vurnelabilities, explanation on mitagion can be found in section 8.

5 Estimate size of difference between two sets

A few variants of estimating set differences are possible.

5.1 Don't do estimation

It's a simple and preferred approach. We can start with size of $k = 10$. Decoding IBF for 2^k elements takes less than 1ms, it can be represented using only 16kb. It can be shown that this if we had a perfect estimation we wound send only 1.13 times minimum bytes

5.2 Strata Estimator

Defined in [2]. This is a probabilistic estimator. If every node keeps the same Strate Estimator generated with the same hash function then it's possible to for malicious actor to generate one edge such that would cause Stata Estimator to large number of edges differ. This would cause full graph to be sent.

This could be mitigated for example by keeping a Stata Estimator computed with different hash for each connection, but this would require to use more memory. You would need to recompute this structure on each connection. Further reasearch is needed if that can be avoided. Though there are better solutions.

5.3 Send hashes of random m edges

Let's say we have Alice and Bob. They send to each other random m hashes of edges choosen with uniform distribution. They record the number of those edges that the other side doesn't have. Let's choose m to be 2000. Assuming the worst case where $e = 1000000$. We would get information about random sample of 0.05 percent of edges. This would allow us to accurately estimate the difference as long as the $d > 1000$. However, if $d < 1000$ then we could send start with $k = 11$ and send IBF with 2^k elements. Sending IBF with $k <= 11$ takes less than 1 ms, and its not expensive.

6 Routing Graph Reconcilation Algorithm

Let's assume Alice and Bobs are peers trying to exchange their routing tables.

```
// this is Alice code, B is Bob's peeer
// step 1
// step 1 can be merged into one rpc

// ask B for number of edges he knows about
let B_edges = B.get_num_edges();

// ask B about how many of my edges he knows about
let known_b   = B.how_many_hashes_are_know(self.get_random_e_hashes(2000));

let known_a = self.how_many_hashes_are_known(B.get_random_e_hashes(2000));
let d = self.estimate_d(known_a, known_b, self.get_num_edges(), B_edges);

// step 2
// compute k
let mut k = compute_k(d, B_edges , self.get_num_edges());
// estimate e
let e_est = B_edges  +  self.get_num_edges() - d;

// step 3
for k in 10..21 { // can be written as one round trip for every iteration
        if k == 20 or (2^k) > e_est / 10 {
                //  A sends all hashes to B, B returns all edges unknown
                //  A asks B to give all hashes known, A gets result, A send to B all
                ...
        }

        // step 4
        // ask B for IBF[k]
        let ibf_b = B.get_IBF(k);

        // try to recover
        let (success, recovered_edges) = self.get_IBF(k).recover_edges( ibf_b);

        // figure out which edges we known about
         let known_edges, unknown_edges = self.split(recovered_edges)

        // send to B missing edges
        B.send_edges( self.get_edges_from_hashes(known_edges));
        // ask B about unknown edges
        let edges_from_b = B.get_edges_by_hash(unknown_edges);

        self.add_edges(edges_from_b);

        if (success) {
                // we are done
                break;
        }

        k += 1;
}

\subsection{Improvements}
\begin{enumerate}
\item step 1 can be skipped to simplify the code. We can always start with k = 10
\item The algorithm can be rewritten to reduce the number of our trips. Each iteratio
Details can be left as an exercise to the reader.
\end{enumarate}
```

7 Performance results

TODO

7.1 adding 1000000 edges to IBF with size of 2^{21}

19ms

7.2 decoding IBF of size 2^{14} with 10000 edges recovered

7ms

7.3 decoding IBF of size 2^{18} with 100000 edges recovered

70ms

7.4 decoding IBF of size 2^{21} with 1000000 edges recovered

150ms

8 Mitigation

TODO

9 Proofs

TODO

10 References

References

[1] NEAR Protocol *https://near.org/*

[2] Efficient Set Reconciliation without Prior Context. *https://www.ics.uci.edu/ eppstein/pubs/EppGooUye-SIGCOMM-11.pdf*

[3] Minisketch: a library for BCH-based set reconciliation *https://github.com/eupn/minisketch-rs*