A Flexible Peer Management Architecture for Blockchain Systems

Alexander Chepurnoy and Bruno Woltzenlogel Paleo

No Institute Given

Abstract. For a blockchain network to be secure and efficient, it is essential that nodes follow a disciplined peer management strategy, trying to avoid malicious peers and seeking to discover and retain reliable honest peers. This paper defines an abstract peer management architecture, relying on flexible ranking and clustering of peers. A concrete implementation of this architecture in the Scorex framework is also described.

1 Introduction

Blockchain systems have been increasingly adopted in recent years. For instance, the total market cap of Bitcoin [**?**], the most popular cryptocurrency and blockchain application, has grown to more than 130 billion US dollars[1] since its inception in 2009. And Bitcoin is not an isolated phenomenon. Fifteen other cryptocurrencies have market caps larger than 1 billion US dollars. With so much at stake, the security of these systems is critical. Consensus protocols, such as Bitcoin's Proof of Work (PoW) [**?**] and variants of Proof of Stake (PoS) [**?**] (e.g. Cardano's Ouroboros [**?**] and Ethereum's TODO [**?**]), ensure that the communicating peers agree on a single blockchain and that the blockchain remains valid as long as honest nodes detain the majority of computation power or stake. However, the network itself is vulnerable to attacks that may prevent the peers from communicating effectively. A recent study [**?**] found, for example, that Bitcoin's peer management is significantly vulnerable to eclipse attacks, where an attacker running several nodes attempts to monopolize all incoming and outgoing connections of an honest node. If the attacker succeeds, the mining power or stake of that honest node, is effectively removed from the network. Furthermore, the attacker may fork the chain and lead the honest node to waste mining power (and profits) on the forked chain. The possibility of such network layer attacks highlights the importance of resilient peer management strategies and motivates the work presented here.

Bitcoin's rigid and hard-coded peer management strategy [**?**] distributes known peers in two tables, one for new peers and the other for tried peers, and groups them in buckets so that peers belonging to the same subnet (and hence more likely to be controlled by an attacker) are more likely to belong to

[1] Data obtained from `http://www.worldcoinindex.com` in 2017 on the 24th of November 2017.

the same bucket. A bitcoin node then tries to connect to peers from different buckets and from different tables.

The goal of the peer management architecture proposed here is to generalize Bitcoin's approach and make it more flexible, because there may be many other characteristics, besides IP address and being new or tried, that may be relevant for a node to consider when deciding whether to connect to a peer. The proposed architecture, described abstractly in Section **??**, depends on a peer ranking algorithm (cf. Section **??**) and a peer clustering algorithm (cf. Section **??**); and procedures and protocols for manipulating peers (cf. Section **??**), from peer banning to selection of peers for connection, may use the ranking and the clusters. Finally, an implementation of the proposed architecture in the Scala-based Scorex framework for blockchains is presented in Section **??**.

2 Abstract Peer Management Systems

A *peer* is a pair $(\text{addr}, \text{port})$, where addr is its IP address and port is its port. The set of all peers is denoted $\mathcal{P}$. Any characteristic of a peer (e.g. number of past connections, date of last connection, reputation, delivery delays) is a *feature* and the finite set of all relevant features is denoted $\mathcal{F}$.

TODO: do you think we need to separate ingoing / outgoing connections?

A *peer management structure* is a tuple (G, B, C), where G is the set of *good peers*, B is the set of *banned peers*, and C is the set of *connected peers*, satisfying the following conditions: $G \cap B = \emptyset$, $C \subseteq G$, $G \subseteq \mathcal{P}$, $B \subseteq \mathcal{P}$. A *peer management system* is a tuple $(M, \phi, \rho_\phi, \xi_\phi, \Pi)$ where:

- M is a peer management structure;
- ϕ is the *feature map* such that $\phi(p, f)$ is the value of feature f for peer p;
- $\rho_\phi : \forall S : 2^{\mathcal{P}}.S \to \mathbb{N}$ is a *peer ranking* such that $\rho_\phi(S, p)$ is the *rank* of peer p in the set of peers S using the feature map ϕ;
- $\xi_\phi : 2^{\mathcal{P}} \to 2^{2^{\mathcal{P}}}$ is a *peer clusterizer* using the feature map ϕ that takes a set of peers S and returns a set of *clusters* $\xi_\phi(S) \equiv \{S_1, \ldots, S_m\}$ such that $S_i \cap S_j = \emptyset$ when $i \neq j$ and $\bigcup_{1 \leq i \leq m} S_i = S$;
- Π is a set of *protocols* and *procedures* that take the current M and the current feature map ϕ and, possibly using ρ_ϕ and ξ_ϕ, return a modified peer management structure M' or a modified feature map ϕ'.

I'm using a dependent type notation here, in a sloppy way. This could be improved.

Example 1. Examples of peer rankings include: a function that sorts the set of peers S according to any combination of features in ϕ and returns the index of p in the sorted ordered set; or a function that returns the number of days since the last connection with p.

Example 2. Bitcoin's usage of separate tables for *new* peers, with whom there has never been a connection, and already *tried* peers, can be seen as a simple and rigid example of clusterizer. Bitcoin's bucket system, which partition peers according to their subnets, can also be seen as a clusterizer.

TODO: More examples, not related to bitcoin, here.

3 A Concrete Peer Ranking

The peer ranking function ρ_ϕ is defined by:

$$\rho_\phi(S, p) \equiv \sum_{f \in F_r} (\text{indexOf}(p, \text{sort}(S, f, \text{isIncreasing}(f)))) * \texttt{\$\{f\}Weight:Int}$$

where:

- F_r is the set of features relevant for ranking, which has `averageDeliveryDelay`, `numberOfPastConnections` and `elapsedTimeSinceLastConnection` as elements.
- $\text{indexOf}(p, L)$ is the index of p in a list L.
- $\text{sort}(S, f, b)$ is the list of elements in S sorted by their values on feature f. Sorting is increasing if $b = \texttt{true}$, and decreasing otherwise.
- $\text{isIncreasing}(f)$ is `true` iff the sorting of peers according to feature f should be increasing. isIncreasing is `true` for `numberOfPastConnections` and `elapsedTimeSinceLastConnection` and `false` for `averageDeliveryDelay`.
- For each feature f, `${f}Weight:Int` is a configurable parameter indicating the importance of feature f for the ranking.

Peers having a higher rank are considered to be better than peers having a lower rank. By sorting peers decreasingly with respect to their average delivery delay, faster peers are prioritized. By sorting peers increasingly with respect to their number of past connections, reliable peers that have already been tried and used many times are preferred. By sorting peers increasingly with respected to their elapsed time since last connection, rotation of peers is encouraged.

4 A Concrete Clusterizer

The peer clusterizer ξ_ϕ is defined by:

$$\xi_\phi(S) \equiv \texttt{k-means}\|_{\texttt{numberOfClusters}}(\text{map}(\lambda p.\text{toVector}^\phi_{F_c}(p) * \texttt{weightVector}, S))$$

where:

TODO: There is a minor typing issue here that I need to fix. kmeans is returning clusters of vectors, but it should be returning clusters of peers corresponding to those vectors.

- `k-means‖` is the parallel scalable k-means++ clustering algorithm [**?**]
- `numberOfClusters` is the configurable parameter specifying how many clusters should be generated. It defaults to `maxConnections`.
- $\text{toVector}^\phi_{F_c}(p)$ transforms a peer p into its vector of feature values $\text{map}(\lambda f.\phi(p, f), F_c)$.
- F_c is the vector of features relevant for clustering, which has `ipOctetOne`, `ipOctetTwo`, `ipOctetThree`, `ipOctetFour`, `numberOfPastConnections`.
- `weightVector` is the vector of weights $\text{map}(\lambda f.\texttt{\$\{f\}Weight}, F_c)$.

dividing an IP address into 4 dimensions in these 4 octets takes care of clustering in the same cluster/bucket peers that belong to the same subnet when IP allocation follows the old *classfull network* scheme. It probably works well for *variable-length subnet masking* as well. But, in this case, maybe something better could be done if we knew the routing prefix. But I think we don't. Or do we? More info: https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

By having IP octets as features relevant for clustering, peers that belong to the same subnet will tend to be in the same cluster. Since the subnet masking convention considers the leftmost bits of an IP address to be more significant (i.e.

the more leftmost bits are shared by two IP addresses, the more likely they are to belong to the same subnet), it should be ensured that `ipOctetOneWeight` >> `ipOctetTwoWeight` >> `ipOctetThreeWeight` >> `ipOctetFourWeight`, where >> means "significantly greater than".

By taking `numberOfPastConnections` into account, peers that differ in how known or new they are will tend to belong to different clusters.

5 Peer Management Protocols and Procedures

Peers are inserted and removed from the sets of good peers, banned peers and connected peers through protocols and procedures. The following subsections describe the concrete protocols and procedures that are implemented in Scorex.

will be

5.1 Peer Discovery

A peer discovery protocol requests new peers from a source and inserts them into the set of good peers G. Generally, a source may be another peer, a trusted central server or an untrusted communication channel (e.g. IRC, Twitter, ...).

In Scorex, when the size of G is below `minGoodPeers:Int`, the peer requests new peers from $p^* = \arg\min_p \rho_\phi(G, p)$. p^* replies by sending a set of new peers N. G is then updated to $G' \equiv G \cup (N \setminus B)$.

TODO: have "referencedBy" column in the peers table

monitor and record âĂIJnumber of incoming connectionsâĂİ and âĂIJnumber of outgoing connectionsâĂİ in the feature table. Then it is up to the ranking and clustering functions to use this information as they wish.

TODO: request from a central server? download from an untrusted communication channel?

5.2 Peer Selection

The peer selection protocol is executed whenever the size of C is smaller than `minConnections:Int`. Its goal is to select a subset N of $n \equiv$ `maxConnections:Int` $- |C|$ peers from $G \setminus C$ and then update C to $C' \equiv C \cup N$.

This is achieved through the following steps:

1. Partition G into the clusters $\{G_1, \ldots, G_m\} \equiv \xi_\phi(G)$.
2. Initialize C' with C.
3. For each cluster G_k, let $h(G_k) \equiv |C' \cup G_k|$.
4. Then, for i from 1 to n:
   (a) Let $H = \{G_k \in \xi_\phi(G) | h(G_k) = \min_{g \in \xi_\phi(G)} h(g)\}$
   (b) Randomly choose a cluster G^* from H with uniform probability.
   (c) Randomly choose an integer r in the $[0, 99]$ integer interval and then choose p^* from G^* with one of the following probability distributions (for all $p \in G^*$):
      - $P(p) \equiv 1/|G^*|$, if $r <$ `peerExplorationProbability:Int`.
      - $P(p) \equiv (\rho_\phi(G^*, p))/(\sum_{q \in G^*} \rho_\phi(G^*, q))G^*$, otherwise.
   (d) Let $C' \equiv C' \cup \{p^*\}$.

Finally, the occasional selection of peers with uniform probability, irrespectively of their rank, gives a chance of connection even to new peers that might have a low rank.

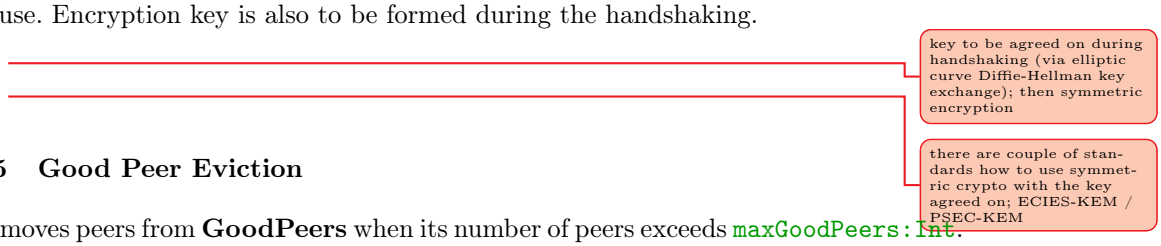
As the peer selection procedure chooses to connect to a set of peers that belong to different clusters, it will tend to choose a mix of old and new peers that belong to different subnets.

5.3 Feeler Connections

Bitcoin does the following: // * Choose a random address from new and attempt to connect to it if we can connect // successfully it is added to tried. // * Start attempting feeler connections only after node finishes making outbound // connections.

5.4 Handshake

During a handshake protocol a peer tries to get information on the availability of another peer and its capabilities. For example, if another peer is working in SPV mode, so does not have full blocks, they should not being asked from it. Or if the peer is interested in encrypted traffic exchange only, it would like to refuse. Encryption key is also to be formed during the handshaking.



5.5 Good Peer Eviction

Removes peers from **GoodPeers** when its number of peers exceeds `maxGoodPeers:Int`.

TODO: use ranking and cluster

5.6 Banned Peer Eviction

Removes peers from **BannedPeers** when its number of peers exceeds `maxBannedPeers:Int`.

TODO: use ranking and cluster

5.7 Peer Banning

Moves a peer from **GoodPeers** to **BannedPeers**.

5.8 Peer Rotation

Replaces a peer in **ConnectedPeers** by another peer from **GoodPeers**.

Speeds up recovery from eclipse attacks.

6 Information Exchange Protocols

7 Encryption and Authentication

ECIES, AES, MAC

8 Configuration Advice

In order to increase the peer-to-peer protocol's resilience against eclipse attacks by a group of malicious peers that pretend to be good until the moment when they decide to attack, `maxGoodPeers:Int` should be set sufficiently high, in order to keep the proportion of malicious peers in **GoodPeers** low compared to the proportion of honest peers. Then the probability that *all* connections are made with malicious peers, as required for an eclipse attack to succeed, is low as well.

Would we ever want to forgive banned peers?

As long as we do not want to ever forgive banned peers, `maxBannedPeers:Int` should be set as high as possible without exceeding the available memory.

I'm assuming that we want to load **BannedPeers** entirely into memory to be able to check quickly whether an incoming connection request is from a banned peer. Is this assumption correct?

9 Implementation in Scorex

In order to allow a high degree of flexibility, the proposed peer management architecture has many configurable parameters. Throughout the paper, configurable parameters and their types are formatted as "`anExampleParameter:ItsType`".

TODO: Parallel implementation of clustering

The peer management system maintains persistent and mutually disjoint tables of **GoodPeers**[2] and **BannedPeers**. There are at most `maxGoodPeers:Int` peers in **GoodPeers** and at most `maxBannedPeers:Int` in **BannedPeers**.

Example 3. TODO: Show an example table

The peer management system maintains a non-persistent table of **ConnectedPeers** to which the peer is currently connected. There are at most `maxConnections:Int` peers in **ConnectedPeers**.

10 Conclusion

Future work: reputation system

Future work: Anomaly detection: Consider a peer anomalous if it is the only element of a cluster?

References

[2] In Bitcoin, presumably good peers are split in two tables: **New** and **Tried**. Here a new peer is simply a good peer with zero past connections.