Byzantine environment

Authors: Benny Mandler, Vita Bortnikov, Yacov Manevich, Artem Barger, Gennady Laventman, Gregory Chockler, Marko Vukolic (additional contributors please add you name to the list)

This document contains the current proposal for a scalable data dissemination infrastructure for the Hyperledger Fabric V1.0. All comments are welcomed.

The new Hyperledger Fabric architecture separates between orderers and peers; orderers provide an ordering service, and peers maintain the ledger and world state. One of the main goals of this new design is to significantly increase scalability of the fabric blockchain, partly by separating chaincode execution from ordering - In contrast to the old architecture, in the new architecture- the chaincode invocations run concurrently on a portion of the peers and not on all the nodes in the network. Increasing both scalability and trust of the fabric in the new design requires supporting a large number of peers.

The motivation of the currently proposed work is to provide efficient data dissemination and scalable communication between all various entities in the fabric network. This is currently needed since previously all functions and guarantees were concentrated at the validating nodes, and currently these are spread throughout different entities. Scale of peers is expected to be larger than in fabric V0.6. As the entities are separated there is a need for a way to disseminate information between all peers in a scalable and reliable manner supporting Byzantine failures

The distributed nature and the requirement for distributed trust does not work well for any centralized approach, and thus we put forward a Peer-to-peer proposal, which adapts better to the dynamic and decentralized nature of the blockchain.

In particular, the proposed next fabric architecture enlarges the scale and diversity of entities involved, which calls for a data dissemination infrastructure with diverse capabilities.

The data dissemination infrastructure will support multi channels by ensuring that the path a data item takes through the network abides by the restrictions of the specific channel a data item is associated with. In particular a data item should nThe proposed dissemination network will disseminate information to all peers (i.e., ordering service output to reach all peers), and in addition provide a continuous background activity to ensure that all peers receive the same data and reach the same state. This will enable support a dynamic set of peers.

The intention is for this mechanism to be scalable and elastic thus work well for small numbers of peers as well, and take into consideration the Byzantine environment in which it operates. Moreover, the intention is for additional fabric components to make use and interact with this infrastructure for data dissemination purposes.

ever reach a node that is not a member of the channel to which the data item is associated with.

Usage within Hyperledger Fabric

The separation of the ordering service from the peers maintaining the ledger introduces complex data dissemination and verification issues in a Byzantine environment. The motivation of the currently

proposed work is to provide efficient data dissemination and synchronization between all various nodes in the hyperledger network, taking into consideration a Byzantine environment. The data dissemination mechanism proposed is mostly for ledger management with the goal of having all ledger holders to have identical copies of the entire ledger.

The proposed communication layer serves to connect between different nodes comprising the blockchain network. A distribution network for handling a variety of inter-node communication patterns is proposed. The proposed communication infrastructure needs to be scalable and secure, such that security, privacy, and integrity constraints are maintained.

The main goals of gossip within Hyperledger Fabric are:

1) Enable all peers to have the same copy of the ledger while alleviating the need that all peers connect directly to the ordering service to receive ledger blocks;
2) Transfer of ledger (and consequently, latest state) to newly connecting peers without involving the ordering service.

Gossip component can be used with different implementations of the ordering service.

1.1 Orderers updates to peers

Information dissemination from the orderers to the peers is required. Namely, the ordered sequence of transactions that is the output of the consensus service needs to reach all the peers for the state updates to be committed and the ledger be updated in all peers. In this case the proposed dissemination structure will connect between the ordering service and the (committing) peers, and will disseminate the information of ordered transactions.

Selected peers will connect to the ordering service via the standard Deliver() protocol, and these peers will be in charge of distributing the received batches further among the other peers. Each organization will select a peer which will connect to the ordering service on its behalf and will disseminate the batches among the rest of the peers belonging to the same organization. The peer election process is further detailed later in this document.

Facebook

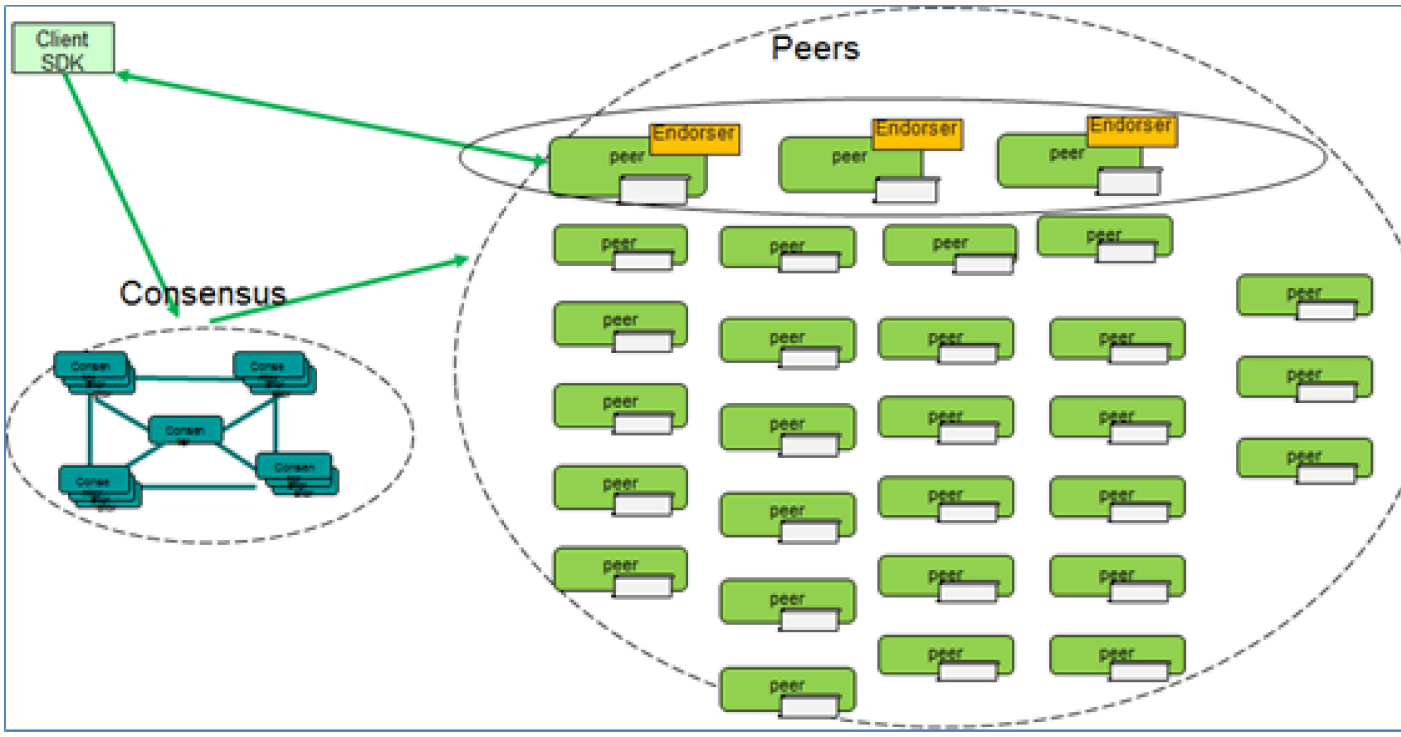

Figure 1: Main communication paths

1.2 Dynamic set of peers: State transfer / synchronization

The proposed scheme shall serve as a backbone for a state transfer / synchronization mechanism, with two flavors. First, for peers that have missed a small amount of batch updates (due for example to network hiccups or temporary overload) a gossip based state synchronization mechanism will ensure that the peer who is lagging behind will obtain the missing blocks from its peers. Second, to support peers joining the network after the network has been operating for a while, an anti-entropy based state synchronization shall be made available, via which potentially large data items will be transmitted among peers.

1.3 Gossiped Ledger block verification

In this subsection, we describe how the blocks gossiped are verified in the gossip layer

On reception of gossiped data block, a peer can verify the proof attached to the block by the ordering service. This proof may be implemented by a given ordering service as an k-out-of-n (multi)signature verification policy. For instance, SOLO and Kafka ordering service implementations require one a gossip peer to verify a single digital signature to verify the integrity of the gossiped block. In contrast,

SBFT ordering service requires f+1 out of n policy, so peers have to verify f+1 orderer signatures per block when using SBFT.

Other ordering service implementations may specify other verification policies or different k in a k-out-of-n policy.

1.4 Multichannel support

A channel is created to define the scope of information sharing, and is associated with a ledger. Every transaction is associated with a single channel, which clearly defines who are the entities (organizations and peers within them) that are aware of this transaction. For more information please refer to Multichannel Consensus and Ledgers.

A new channel is created by having the client SDK send for endorsement a CONFIGURATION transaction and pass it through the ordering service. The channel creation specifies the list of organizations that constitute the channel, namely the organizations that are eligible to join the channel.

Once the channel has been created, the client SDK can instruct specific peers belonging to the allowed organizations to join the newly created channel. The gossip component within the joining peer starts broadcasting within all organizations that belong to the channel, its participation in the channel.

For gossip to work correctly in a multichannel environment membership needs to be maintained within the channel members, namely all peers that have joined a channel need to know about each other. The peer connecting to the ordering service on behalf of an organization will receive the information as to which peers belong to each channel, and will invoke the Deliver() protocol on their behalf. The ordering service in turn shall ensure that the peer indeed belongs to an organization which is allowed to participate in the channel, and will enable the delivery of batches to the connected peer. The connected peer in turn will distribute newly received blocks among all the peers who joined the channel. Once the channel membership has been established the gossip component will operate according to its regular algorithms, with the only restriction that the membership of the channel is used for each piece of information that needs to be gossiped within that channel. Thus, no data items belonging to a specific channel will be delivered outside the channel; that's true both for transactions as well as gossip related membership information. The existence of a channel in any form will not be disclosed to peers outside of the channel.

To achieve that, once a peer joins a channel it needs to obtain the latest channel configuration, to discern which are the participating organizations. Upon joining a channel the peer is provided by the client sdk with the latest configuration transaction of that channel, which includes the participating organizations. For a newly created channel that would be the genesis block, and for older channels which have changed since inception that would be the latest reconfiguration block.

Once a peer has that information, the gossip component can share the information within the boundaries of the allowed organizations. In that manner membership of a channel is maintained by the gossip

service only within the allowed bounds. Once that information has been established gossip can work in a per channel basis based on that information.

The gossip per channel will work in an identical manner to a standard gossip, with the caveat that membership is per channel, thus all operations will be performed within the membership of the channel. In particular a peer will select only other peers from this channel to forward a block to, and will perform state reconcile only with additional peers of the same channel. Note that state synchronization may take place among peers that belong to different organizations as long as they are members of the channel in question.

When organizations are removed from a channel, a similar process to channel creation takes place. Namely, the client sdk will send for endorsement a CONFIGURATION transaction. Once endorsed, the transaction will flow to the orderers. In addition remaining peers will receive a corresponding join Channel invocation with the revised list of authorized organizations.. Each gossip component receiving this information will exclude the removed peers from its membership list, and thus channel related information will only flow to remaining channel peers.

Peers who have been removed from a channel will learn of that fact only eventually. The gossip membership maintenance component will keep on sending general alive messages from all peers, but specific alive messages for that channel will not flow anymore to the removed peer. Thus, after a while the peer will realize that it is not a member of a channel anymore.

1.5 Authentication and membership management

Each peer in the gossip network has its own certificate(identity) given by a Membership Service Provider (MSP) from the ones recognized within the blockchain network). It represents itself to other peers by an identifier that is deterministically derived from its certificate (identity). That identifier is called a peer's PKI-ID.
Each message that is gossiped (sent from a peer to a group of peers non-necessarily in a point-to-point manner, meaning- it can be relayed between peers) and asserts something about the peer:

- Must contain the peer's PKI-ID
- Must be signed by the peer
- Can be verified given the peer's certificate.

Messages that are sent point-to-point between peers and are not gossiped, are not signed because of the assumption that in a production environment, the peer's TLS layer is activated and takes care of security considerations (prevention of traffic hijacking, replay attacks, etc.)
The only message that isn't signed by a peer and also doesn't travel point-to-point is a message that contains a ledger block, which is signed by the ordering service.

The membership view of a peer (the peers it knows of) is built in the following way:
Each peer periodically gossips a special message called an **AliveMessage**, that contains the peer's:

- PKI-ID

- Endpoint (concatenation of host + ":" + port)
- Metadata (a byte array, kept for future use)
- PeerTime that consists of:
    - Peer's incarnation time (startup time)
    - Monotonously increasing counter that is incremented at each dissemination of **AliveMessage**
- Signature over the fields above
- Certificate of the peer (optional)

The certificate of the peer is included only for a limited period of time when the peer starts up and isn't added after that period of time.
When a peer receives any message from any peer that is signed by that peer, it validates it using the peer's certificate it should have received prior to receiving the message.
That's why the peer's certificate is included in the **AliveMessage** for a certain period of time after a peer starts-up, in order for peers to be able to verify its signature even though they haven't learned of its certificate yet.

Peers that haven't received other peer's certificates, can learn of other peer's certificates by a periodical dissemination mechanism that takes care of disseminating certificates to peers that are missing these certificates. This dissemination mechanism is pull-based only and has a very low overhead (discussed in a different section)

When a peer receives an **AliveMessage** from a remote peer either for the first time or if that peer was considered dead (hasn't sent an **AliveMessage** for a long time, or the connection to that peer has failed), the peer (re)connects to that remote peer by looking at the endpoint field in the **AliveMessage**.

Authentication:
In a production environment, it is assumed that the peers employ mutual TLS and each sides of the TLS connection have a valid TLS certificate.
When a peer contacts another peer for the first time, it performs a handshake with that peer that proves that the remote peer has the private key of the TLS certificate, thus binding the TLS session with the fabric membership identity.
The handshake is symmetrical and simple: The first message each peer sends to the remote peer is a **ConnEstablish** message containing:

- Signature over the hash of the TLS certificate of the peer
- PKI-ID of the peer
- Fabric membership service provider certificate

Each peer, in turn:

1. Receives that message
2. Extracts the TLS certificate of the remote peer and hashes it
3. Verifies the signature of the **ConnEstablish** message over the hash

If the verification failed, the peer rejects the connection.

1.5.1 Handling channel membership changes

When the gossip component initializes, it is given an implementation of a channel validation policy, which is able to determine a mapping between a remote peer and the organization it belongs to; thus establishing for which channels it is permitted to send blocks to.

This works by having a mapping of each peer's PKI-ID and its root org CA. The local ledger has the latest configuration of which organizations (denoted by the root org CA) are members of the channel.

Batches sent by the ordering service contain the latest configuration block sequence number of that channel. This way- when a peer receives a block from a fellow peer, it can look at the block's sequence number and at its own last sequence number in the ledger, and know whether it is safe to forward that block to other peers by looking at the sequence number of the last configuration block: if the sequence number of the last configuration block written in the data block received is higher than the last data block committed into the ledger, the data block is not forwarded but is delayed and kept in-memory. Otherwise, the last configuration the ledger has is up-to-date and it is safe to forward that block to whatever peers the channel validation policy of the gossip component allows.

1.6 Resulting Capabilities Needed

- **Efficient Information dissemination from one source to many destinations**
- **Peer-to-peer gossip**
- **Pub / sub like features to support multi-channels**
- **Membership – keep an updated list of nodes that are alive and connected with associated metadata**

1.7 Initial API

- **Gossip() – send a message to all nodes in the specific gossip group**
- **Accept() –reception of a message from other peers**
- **GetPeers() – obtain current membership view**
- **UpdateMetadata() – update the metadata of this node**
- **Stop() - stops the gossip component**

2 Implementation

The basic foundation is an unstructured Peer-to-Peer Gossip based data dissemination network.

The main goals of the implementation are to achieve efficient message dissemination from a single source to all members, while synchronizing the state of different nodes in the background to tackle Byzantine behavior, dynamic addition of peers, and network partitions.

Capabilities provided:

1. Discovery and Membership – keeping track of members who are alive and connected
2. Broadcast – deliver a message from a single source to all members
3. Background activity to sync state of all nodes

Gossip-based broadcast complementary modes of operation:

1. Push: At first distribute the message aggressively
2. Push-pull: Exchange information with neighbors to sync on delivered messages
3. Anti-entropy: Once in a while exchange information with neighbors to sync the global state

All peers are organized in a gossip network and maintain membership information about all available peers. Both ledger, state and membership information are disseminated using the gossip infrastructure.

Gossip based broadcast is achieved by having a peer receiving a message, choosing a set of random peers and propagating the newly received message to these peers. In addition there is a state reconciliation process (based on anti-entropy) that is used to synchronize the state between peers. That is achieved by periodically having a peer compare information with other peers and reconcile their states (pull). Full peer membership is maintained at each peer, to enable it to choose the random peers it will send the message to. In this scheme there is no need to maintain fixed connectivity, thus it is very robust, (relatively) easy to implement, while tolerating crash and Byzantine failures

Demos can be seen in https://www.youtube.com/watch?v=Di4y1gauoKc, https://www.youtube.com/watch?v=qpFzZRNKKp0.

2.1 Bootstrap

2.2 Membership - discovery

Each node upon startup receives as a configuration parameter a set of peers it should try to connect to. This list is known as the bootstrap set. The node periodically tries to establish a connection to peers in

the bootstrap set which do not appear in its current membership view. Upon successful establishment of a connection with a new peer, the full view of both peers is exchanged, and the local membership list on each peer is updated accordingly. Note, that these connections will not necessarily be maintained beyond this initial discovery phase. Each peer maintains a list of known peers that are alive and a set of peers considered to be dead. Each such entry includes the latest alive message received from each peer.

Periodically each peer broadcasts an **Alive** message to the network. In turn, periodically each peer scans its membership list and if an entry in there has not refreshed its alive status in a long enough time, that peer is considered to be dead and will be removed from the alive list.

Each peer decides by itself about dead peers based on the latest alive message received; this is the simplest way to provide Byzantine tolerance, and takes advantage of the fact that each alive message is cryptographically signed by the peer and thus- cannot be forged by attackers – thus, an attacker cannot alter the membership state of another peer by sending information.

Peers periodically attempt to re-connect to peers that appear dead.

2.3 Data dissemination (broadcast)

The underlying supporting structure for a data dissemination mechanism is gossip-based communication among nodes. In such a scheme no fixed overlay is constructed but rather a random set of peers is chosen for message propagation independently at each time at each peer. Information flows throughout the system by having peers exchanging information with the random peers selected at a certain point in time. This kind of interaction is usually more robust and easier to maintain in the face of churn and Byzantine behavior, than a structured overlay. Moreover, such a structure contains multiple disjoint paths between each pair of nodes, but the message complexity can be larger. Thus, it can used to satisfy previously described requirements taking into account the nature of data that is to be communicated across the network.

In such a scheme a peer connects to a random sample of its peers and exchanges information with them. Later on, messages can be transmitted throughout the network, each node propagating the messages via the selection of a random sample of peers. Note that in order to choose a random peer to connect to a peer can refer to its current membership view, a history of nodes that were alive at a previous point in time, its bootstrap set, and a list of all possible nodes as (if such information exists).

The main capability provided by this approach at first is message broadcast by flooding. Namely, a robust and efficient manner to have a message reach all connected nodes. An inherent tradeoff exists between robustness to Byzantine failures and the overhead of the communication system in terms of amount of messages sent overall and amount of peer nodes selected for message propagation at each round. A high outgoing degree from each node, as the underlying backbone to the dissemination structures detailed above, ensures that there are independent paths between different nodes such that the existence of a Byzantine peer along the route does not cut off a whole portion of the network from receiving certain messages. The higher the amount of neighbors we allow a peer to forward a message to, the more robust the system shall be while facing Byzantine peers while naturally, more messages overall will be sent within the network. Our theoretical work and resulting simulations have shown that a

rather small out degree from each peer can assure proper propagation even in the presence of a high percentage of Byzantine peers.

The dissemination component interacts with peers, by enabling them to broadcast a message to the network. In addition it interacts with the peer / ledger component in order to provide it with newly received ordered blocks, and enable the ledger to request missing blocks.

Push stage: Upon receiving a message a peer will Select a set of k random peers and forward the message to them. If a selected peer is not responsive, update the discovery module accordingly.

Pull stage: Periodically, select a set of random peers and exchange membership and block reception information with them. State reconciliation will follow by having each peer update its neighbor with missing information.

The interaction with the consensus component is achieved by the standard orderer API.

2.4 Leader election

Leader election capability is needed to determine which of the peers belonging to the same organization will connect to the ordering service on behalf of that peer. The leader election process will take place within the gossip layer, and assumes that more than one such leader can exist simultaneously for some periods of time, and that non-Byzantine behavior is expected.

The leader election process is based on the peers' metadata, which is distributed in the correct scope. For the remaining of this section we'll define the task and related scope as selected a leader within an organization. When a peer comes up it sets a "leader" metadata item to false, and this information is propagated to all other members of the organization. The aim is to have eventually one node with the "leader" item set to true.

Peers wait for a period of time for membership (including metadata information) information to propagate throughout the group. Once that timeout has expired, if there's no designated leader and the current peer has the lowest lexicographical name (or any other tie breaking algorithm), it declares itself as leader by setting the "leader" attribute to "true". If before timeout expiration another peer declared itself to be leader than it is accepted as the leader. If multiple peers declare themselves as leaders simultaneously, the one with the lowest name will prevail and the rest will give up.

All peers reports the leader to be the one with the lowest name among the ones that declared themselves to be leaders. Upon leader failure, the lowest remaining node declares itself to be the leader. Once again upon detection of another declared leader with a lower name, the current node will give up.

3 Staged Plan

The plan is structured in a way that will enable to progress towards the end goals, while integrating with additional fabric components whenever possible. The intention is to try to ingest incremental code changes rather than one large gulp that is difficult to swallow at once.

The first capability we plan to implement is the efficient information dissemination capability (broadcast) that can be used to disseminate newly created transaction batches to all peers.

1. First, the API has been spelled out, such that it provides external entities with the required functionality, and internal design is spelled out to support the API.
2. At a second stage a simplistic approach is implemented such that other components relying on this capability can start using it early.
3. A first full implementation of the information dissemination capabilities is produced, taking into account Byzantine faults.
4. Integration with additional fabric components, mainly the orderers as producers of information and the peers as the target of information fed by the orderers.
5. Integration with peer and ledger code such that state synchronization is enabled.
6. Add leader election support.
7. Add support for multi-channels.

4 Relevant papers

1. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing (http://arxiv.org/pdf/1602.06997v1.pdf)
2. Efficient Byzantine Broadcast in Wireless Ad-Hoc Networks (http://www.cs.technion.ac.il/~dvadim/pics/TR.pdf)
3. Brahms: Byzantine resilient random membership sampling (http://www.cs.technion.ac.il/~gabik/publications/Brahms-COMNET.pdf)
4. Fireflies: A Secure and Scalable Membership and Gossip Service (http://dl.acm.org/citation.cfm?id=2701418)
5. Self stabilizing and byzantine tolerant overlay network (http://www.cs.huji.ac.il/~dolev/pubs/opodis07-DHR-fulltext.pdf)
6. Separating Agreement from Execution for Byzantine Fault Tolerant Services (https://www.cs.utexas.edu/users/lorenzo/papers/sosp03.pdf)
7. On Scaling Decentralized Blockchains (http://fc16.ifca.ai/bitcoin/papers/CDE+16.pdf)
8. PeerReview: Practical Accountability for Distributed Systems (https://www.cis.upenn.edu/~ahae/papers/peerreview-sosp07.pdf)
9. Assessing Data Availability of Cassandra in the Presence of non-accurate Membership (http://midlab.diag.uniroma1.it/articoli/discco2013.pdf)
10. BAR Fault Tolerance for Cooperative Services (https://www.cs.utexas.edu/~lorenzo/papers/sosp05.pdf)
11. BAR Gossip (http://www.cs.utexas.edu/users/dahlin/papers/bar-gossip-apr-2006.pdf)