gossipArch.md

Hyperledger fabric dissemination network

The goal for the dissemination fabric is allow information dissemination between different nodes in the Hyperledger fabric. This includes communication between consenters to peers, peers to peers etc.

As a first goal we focus on ledger dissemination from consenters to all peers in the system in a scalable way.
Other communication pattern like communication between submitters and endorsers will be addressed later.

The dissemination component's goal is to provide a reliable way of passing messages from the peers connected to the consensus service to the rest of the peers.

The communication is done using gossip: When a peer receives a message, it chooses a set of random peers and propagates the new message to them.
In addition we have an anti-entropy mechanism that is used to synchronize the state between peers. We keep full peer-memebership and liveness information available at each peer - this is needed at this stage to able to choose **K** random peers. Later on we can extend this mechanism for keep information about chaincode availability on peers etc.

The dissemination component consists of two entity types:

- GossipService: A component that initiates message dissemination from a fabric peer to the rest of the fabric peers that are members of the dissemination network.
- GossipMember: A component that interacts with the fabric peer itself, and:
  - Provides blocks sent by other fabric peers
  - Obtains blocks from the local ledger to be sent to other fabric peers that don't possess the respective blocks.

The Interface to the ledger is designed to operate in the following way:

• The user of the component (ledger) can convey its interest in a consequent series of ledger-generated messages (blocks) to the gossip component.
If such a message was received from a peer and stored internally it can be passed to the ledger.

• Assuming the user (ledger) possesses a message (block) with sequence number of i , the gossip component can fetch from the user (ledger) any message (block) with a sequence number **j<i**

If a peer's ledger falls behind too much from the rest of its peers, it doesn't use the state transfer mechanism described in this document, but rather falls back to a different specific mechanism of state transfer which will be added in the future.

Components API

GossipService

```
// Payload defines an object that contains a ledger block
type Payload struct {
    Data      []byte    // The content of the message, possibly encrypted
    Hash      string    // The message hash
    seqNum  uint64  // The message sequence number
}

// GossipService is used to publish new blocks to the gossip network
type GossipService interface {
    // payload: Holds the block's content, hash and seqNum
    Publish(payload Payload) error
}
```

GossipMember

The GossipMember's creation involves passing the following objects to the constructor:

```
// ReplicationProvider used by the GossipMember in order to obtain Blocks of
// certain seqNum range to be sent to the requester
type ReplicationProvider interface {
    GetData(startSeqNum uint64, endSeqNum uint64) ([]Payload, error)
}

// MessageCryptoVerifier verifies the message's authenticity,
// if messages are cryptographically signed
type MessageCryptoVerifier interface {
    // Verify returns nil whether the message and its identifier are authentic,
    // otherwise returns an error
    Verify(seqNum uint64, sender string, payload Payload) error
}

// MessagePolicyVerifier verifies whether the message conforms to all required policies the ledger defines,
// and can be safely delivered to the user.
type MessagePolicyVerifier interface {
    Verify(seqNum uint64, sender string, payload Payload) error
}
```

The GossipMember's API is the following:

```
// GossipMember is used to obtain new blocks from the gossip network
type GossipMember interface {
  // RegisterCallback registers a callback that is invoked on messages
  // from startSeq to endSeq and invokes the callback when they arrive
  RegisterCallback(startSeq uint64, endSeq uint64, callback func([]Payload))
}
```

Dissemination protocol

The gossip component consists of the following 2 modules:

- discovery module - maintains alive and non-responsive peer set
- communication module - maintains connections and disseminates messages

The discovery module uses the communication module to send its information, and the Communication module passes discovery associated messages to the discovery module.

discovery Protocol flow of a peer:

- Each peer is given a bootstrap peer set **B** at its creation.
- Each peer has a set **V** of `<id, endpoint, logical_timestamp>` triplet which represents the peer's known and responsive member set, and a similar set **H** of non-responsive peers, but **V[id]** will denote the the last Alive message from a peer with a certain id element in **V**.
- The messages can be cryptographically signed(verified) with a priv (pub) key issued by the network's CA.

The protocol definition is as follows:

discovery(p):

1. **H=B**, **V={}**
2. On startup and once in a while, try to connect to every $\mathbf{p_i}$ in **H** and if the connection is successful, add p_i to **V** and update the communication module.
3. For each $\mathbf{p_i}$ in **V**, obtain $\mathbf{V}^{p_i}$ and set **V** to be the union of all $\mathbf{V}^{p_i}$ obtained from the peers queried.
4. If **V** has changed, update the Communication module for creating connections to all new added peers and removing connections for non-responsive peers.
5. Once in every **T** seconds:
    - the peer **p** passes an **Alive** message to the Communication module for dissemination. The message includes:
        a. $\mathbf{p}^{\mathbf{id}}$
        b. $\mathbf{p}^{\mathbf{endpoint}}$
        c. $\mathbf{m}^{\mathbf{logical\_timestamp}}$ - a tuple of
            a. **inc_number** is the timestamp at which **p** started running
            b. **seq#** is a monotonously increasing number that increases by 1 on each **Alive** message generation.

6. for each member p_i in the discovery set **V**, keep track of the last time an Alive message originated from it.
   If more than **5T** seconds have passed without receiving an Alive message from p_i:
   i. Remove $\mathbf{p_i}$ from **V** and
   ii. Add $\mathbf{p_i}$ to **H**
   iii. Update the Communication module to close the connection to it.
7. On reception of an **Alive** message **m** from a peer $\mathbf{p_i}$:
   i. If **m.id** doesn't exist in **V**: **V[m.id] = m**.
   ii. Else, if both **m.timestamp.inc_number > V[id].timestamp.inc_number** and **m.timestamp.seq# > V[id].timestamp.seq#**: **V[m.id] = m**.

The Communication protocol flow of a peer:

- Holds an internal buffer for messages not passed to the ledger.
- Each message **m** received from a peer has an **id** that can be verified.
  That id is either the `<m.timestamp, p.id>` in case of an **Alive** message or the hash of the ledger block in case of a ledger disseminated block.
  Either way, the protocol communication protocol refers to the message's id as **m.id**.
- Each peer holds a set **R** of message ids it has previously encountered, along with their received time, and periodically discards old message ids that were received more than **5T** seconds ago.

communication protocol of p:

1. Upon receiving a message **m** from a peer $\mathbf{p_i}$ at time **t**:
   i. If the message is an **Alive** message:
      a. Pass it to the discovery module.
2. If **m.id** id was previously received (found in **R**), discard the message [1]
3. Else:
   i. add `<m.id, m.t>` to **R**
   ii. Select a set of **k=log(|V|)+3** random peers: $\{p_0,p_1,...,p_k\}$ at random out of **V** and for each $\mathbf{p_i}$:
      a. If $\mathbf{p_i}$ is non responsive, update the discovery module to add $\mathbf{p_i}$ to **H**
      b. Send **m** to $\mathbf{p_i}$ [2]
4. Once in every T seconds, select a set of **k=log(|V|)+3** random responsive peers out of **V** and for each such $\mathbf{p_i}$:
   i. Send **V** to $\mathbf{p_i}$
   ii. Obtain $\mathbf{V^{p_i}}$, $\mathbf{H^{p_i}}$ and add each **p** in $\mathbf{V^{p_i}}$ but not in **H** to **V** and add each **p** in $\mathbf{H^{p_i}}$ to **H**.
   iii. Send to $\mathbf{p_i}$ your highest ledger originated message sequence number **s** for which you received before all $\mathbf{m_i}$ **s.t ,i < s**.
   iv. If you have before received a message $\mathbf{m_j}$ that wasn't passed to the ledger yet such that **j>s** , send to $\mathbf{p_i}$ a bitmap **b** of all sequence numbers from **s** to **j** , And receive back from $\mathbf{p_i}$ a set of messages $\mathbf{M_i}$ and (optionally) its own $\mathbf{s_i}$, $\mathbf{b_i}$.
   v. Store the messages from **M** internally[3].
   vi. (Optionally) send back to $\mathbf{p_i}$ a crafted **M** of your own according to $\mathbf{p_i}$'s $\mathbf{s_i}$, $\mathbf{b_i}$ sent.
5. Upon receiving a message **m** from the user, or from the Discovery module:
   i. add `<m.id, m.t>` to **R**
   ii. Select a set of **k=log(|V|)+3** random peers and send **m** to them.

[1] - If m.id was previously received, we surely have handled and disseminated it ourselves before, so no need to do that again.
[2] - Sending messages may not be immediate, but rather a batching mechanism might be employed.
[3] - The communication module stores the messages in an internal data structure that enables efficiently delivering consequent sequence numbers to the ledger, but also inserting messages that arrive with out-of-order sequence numbers. This enables the gossip component to on the one hand keep blocks that it can't deliver yet because it's missing earlier blocks, and on the other hand fetch the missing blocks from other peers to obtain a consecutive set of sequenced blocks to deliver them to the ledger.