Babble Documentation

Release 0

Mosaic Networks

Jul 10, 2021

Contents

Babble is based on our own interpretation of Hashgraph, but also builds upon other techniques that facilitate coordination within distributed systems. Here, we give a high-level overview of the most important concepts that inspired the development of Babble and how they all fit together. This document is also intended for a non-technical audience.

CHAPTER 1

Common Knowledge

Roughly speaking, attaining common knowledge within a group means "everyone knows that everyone known that everyone knows..." to infinity. It is a necessary and sometimes even sufficient condition for reaching agreement and for coordinating actions. This connection was perhaps first drawn by David Lewis in his work on conventions, which led to the original definition. It is a fascinating topic that goes far beyond computer systems. We highly recommend the book Reasoning About Knowledge for a very thorough treatment of the subject.

To get an intuition about the link between common knowledge and agreement, we can look at the well know 'coordinated attack' problem. Two generals and their respective armies are posted on opposite sides of an enemy city perched on top of a hill. They must decide to attack together, at the same time, or not at all. Indeed, if one general attacks alone, he will lose the battle. The only means of communication is a messenger on horseback (always at risk of being intercepted by the enemy). How do they coordinate their attack?

One general, having made the decision to attack, could send a messenger to the other general. Upon receiving that message, the second general knows that the first general wants to attack, but he doesn't know that the first general knows that he received the message. So he sends an acknowledgment. Upon receiving the acknowledgment, the first general, knows that the second general knows that he wants to attack, but he doesn't know that the second general knows that he received the acknowledgment... There is always this element of doubt preventing either general from committing to a decision. It quickly becomes apparent that what is needed is common knowledge.

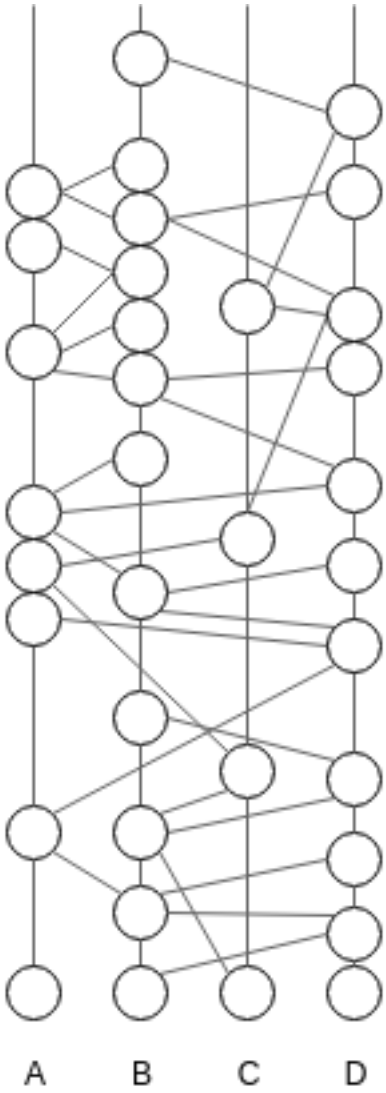
The dilemma is that pure common knowledge is not attainable in practical situations; particularly in asynchronous message passing systems with unreliable transports (like the two generals). Hence, we have to relax our requirements and rely on approximations of common knowledge. In Babble, we drop the simultaneity and allow participants to decide at different times.

CHAPTER 2

Gossip About Gossip

One way to approximate common knowledge in this context is to use a communication protocol where participants regularly tell each other everything they know about what everyone else knows. These are usually referred to as Full Information Protocols, aka ‘gossip about gossip’.

Members locally record the history of the gossip protocol in a directed acyclic graph, a DAG, where each vertex represents a gossip event and the edges connect a vertex to the immediately-preceding vertices. Roughly speaking, a member, say Alice, will repeatedly choose another member at random, say Bob, and attempt to learn what he knows that she doesn’t know. She will send him a sync request saying ‘Hey, here is what I know; what do you know that I don’t know?’. Bob will compute the difference and respond with a set of events that he knows and Alice doesn’t yet know. Alice will insert these events in her DAG, and create a new event to record this sync. The newly created event includes the hashes of her last event, and Bob’s last event. Hence, the DAG is connected by a succession of recursive cryptographic hashes; like a blockchain, but two-dimensional. Each event contains the hashes of the events below it and is digitally signed by its creator. So the entire graph of hashes is cryptographically secure.



The communication graph is a very rich data structure from which we can extract all sorts of information about the history of gossip, and also derive a consistent ordering of the events, even in the presence of faulty participants. But let's take it step by step.

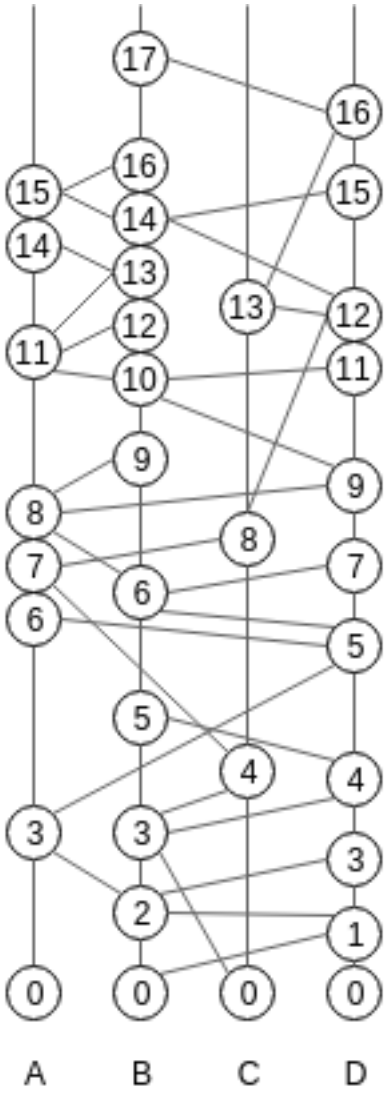
CHAPTER 3

Lamport Timestamps

Leslie Lamport introduced a seminal paper in 1978, entitled "Time, Clocks, and the Ordering of Events in a Distributed System". In this paper he describes a distributed algorithm for extracting a consistent total ordering of the events in an asynchronous message passing system, using a concept of Logical Clocks.

The algorithm follows some simple rules:

1. A process increments its counter before each event in that process;
2. When a process sends a message, it includes its counter value with the message;
3. On receiving a message, the counter of the recipient is updated, if necessary, to the greater of its current counter and the timestamp in the received message. The counter is then incremented by 1 before the message is considered received.
4. Ties are broken using an arbitrary function (eg. sort by hash)



This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. Synchronization is achieved because all processes order the commands according to their timestamps, so each process uses the same sequence of commands. A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T.

However, the resulting algorithm requires the active participation of all the processes. A process must know all the commands issued by other processes, so that the failure of a single process will make it impossible for any other process to execute commands, thereby halting the system. Babble implements Lamport Timestamps on top of the hashgraph, but with added steps for Byzantine Fault Tolerance.

This paper triggered a wave of research on BFT consensus algorithms. Some famous solutions are Paxos, PBFT, and Tendermint. Ultimately most of them are variations of a very well known paradigm in computer science: two-phase commit.

CHAPTER 4

Two-Phase Commit

We are not necessarily aware of it, but we all solve the consensus problem in real life situations on a daily basis. This is illustrated in the following quote from a blog:

"Simple solutions to the consensus problem seem obvious. Think about how you would solve a real world consensus problem - perhaps trying to arrange a game of bridge for four people with three friends. You'd call all your friends in turn and suggest a game and a time. If that time is good for everybody you have to ring round and confirm, but if someone can't make it you need to call everybody again and tell them that the game is off. You might at the same time suggest a new day to play, which then kicks off another round of consensus."

Most distributed consensus protocols are special adaptations of this concept. There is a theoretical result that says one can't attain BFT, in the same conditions, with of malicious participants. So, with the assumption that at least of participants are good, the usual solution resembles something like this:

1) Someone proposes a value
2) Everyone votes on the proposal and broadcasts their vote
3) Every one confirms they have received of votes for the same proposal, and broadcasts this confirmation.
4) When a participant collects of such confirmations, it commits the value.

Usually, the solutions vary around who gets to propose the value - aka the leader - and how this leader is elected or changed.

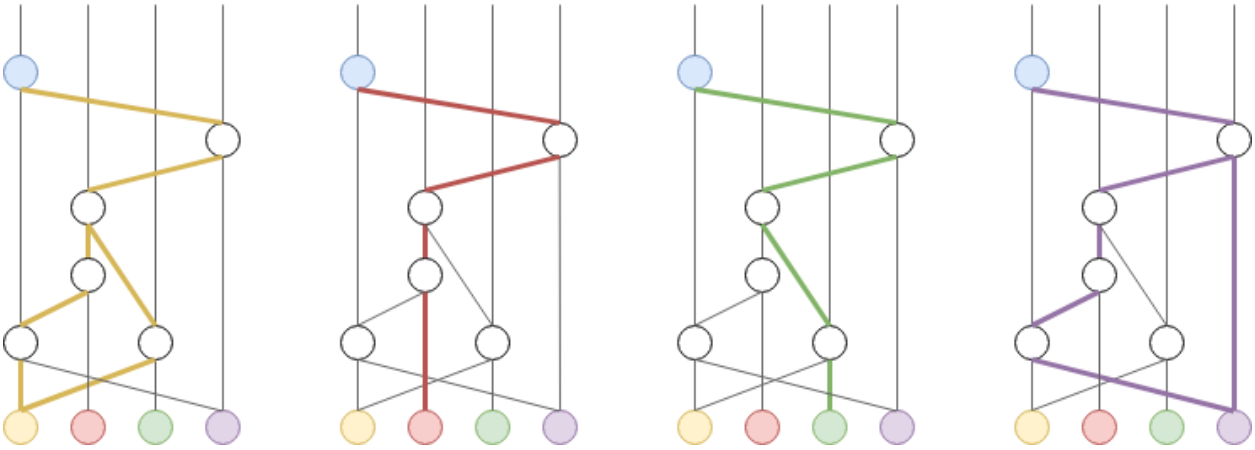
CHAPTER 5

Virtual Voting

A similar algorithm can be run internally thanks to the communication graph by using the concept of virtual voting. Instead of exchanging votes directly, we compute what other participants would have voted, based on our knowledge of what they know.

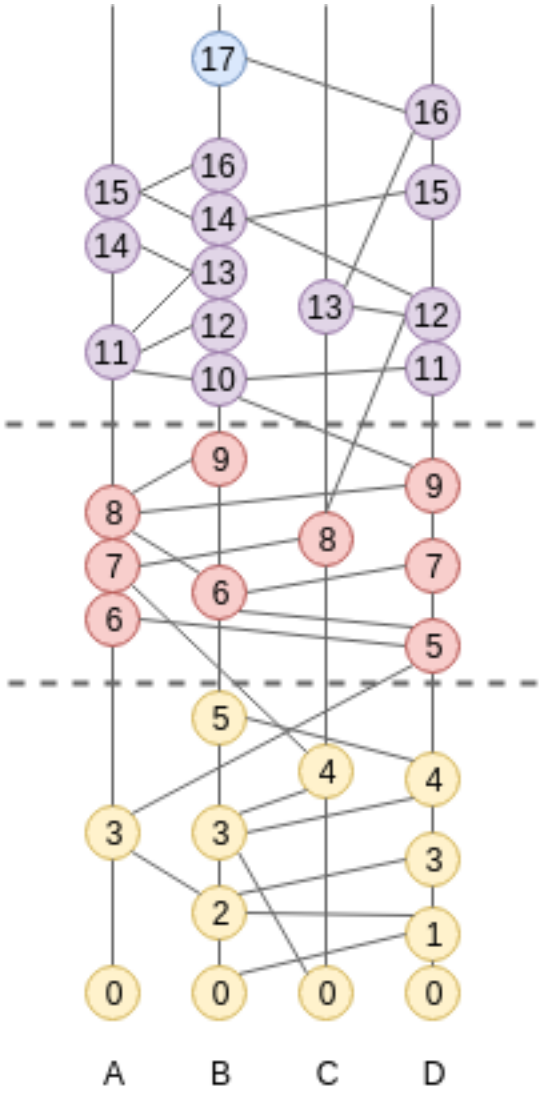
First, the Hashgraph defines a concept of *Strongly Seeing*:

"If there are n members, then an event w can strongly see an event x, if w can see more than 2n/3 events by different members, each of which can see x".



Strongly Seeing is analogous to receiving votes from two thirds of participants in the first phase of the two-phase commit.

Also, we do not need a leader to propose a value. Instead, participants compute virtual cuts in the hashgraph, called rounds, which allow processing events in batches. This is also a distributed algorithm where all members end up with the same rounds. Roughly speaking, starting at round 0, when we reach a point when of members can strongly see the cut from the previous rounds, we start a new round. When there is common knowledge about a round, attested by *Strongly Seeing*, we can decide on the order of event below that cut. The details of the algorithm are best described in the original hashgraph whitepaper.

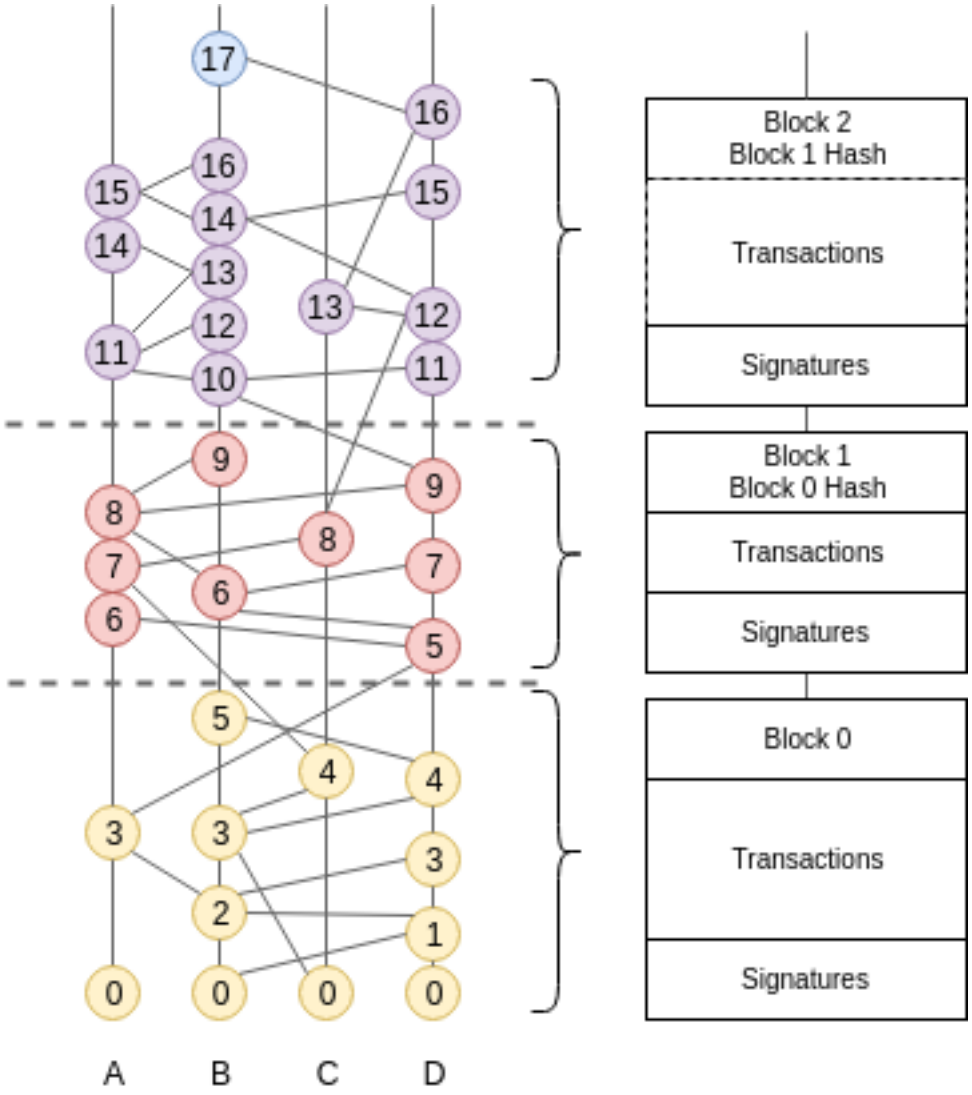


So this algorithm doesn't need a leader. All participants run the algorithm locally, process rounds at their own speed, and end up outputting the same batches of ordered events. Babble takes these batches of events and projects them onto a blockchain.

CHAPTER 6

Blockchain

A blockchain is a one-dimensional data-structure made of cryptographically chained blocks. It is convenient to map our two-dimensional hashgraph onto a blockchain because the blockchain is much easier to work with when it comes to consuming and verifying the output of the consensus algorithm. The concatenation of blocks, and the transactions they contain, is recursively secured by digital signatures. A block that obtains enough signatures (>1/3) can immediately be considered valid, along with all the blocks that precede it, because it contains a signed fingerprint of the list of blocks so far. The projection method is described in *From Hashgraph to Blockchain*.



So the output of Babble is a sequence of blocks; the interface between the app and Babble is a blockchain interface. This makes it convenient for developers to plug into Babble, and provides a base for building light-clients and cross-chain communication protocols. We believe that the p2p internet is moving towards a landscape of interconnected blockchains, the so called internet of blockchains, an Babble is built with this in mind.

CHAPTER 7

From Hashgraph to Blockchain

This document describes a technique for projecting a hashgraph onto a blockchain, which is better suited for representing an immutable ordered list of transactions. In this system, the order is governed by the Hashgraph consensus algorithm but the transactions are mapped onto a linear data structure composed of blocks; each block containing an ordered list of transactions, a hash of the resulting application state, a hash of the corresponding section of the hashgraph (Frame), a hash of the current peer-set, and a collection of signatures from the set of validators. This method enables hashgraph-based systems to implement any Inter-Blockchain Communication protocol and integrate with an Internet of Blockchains.

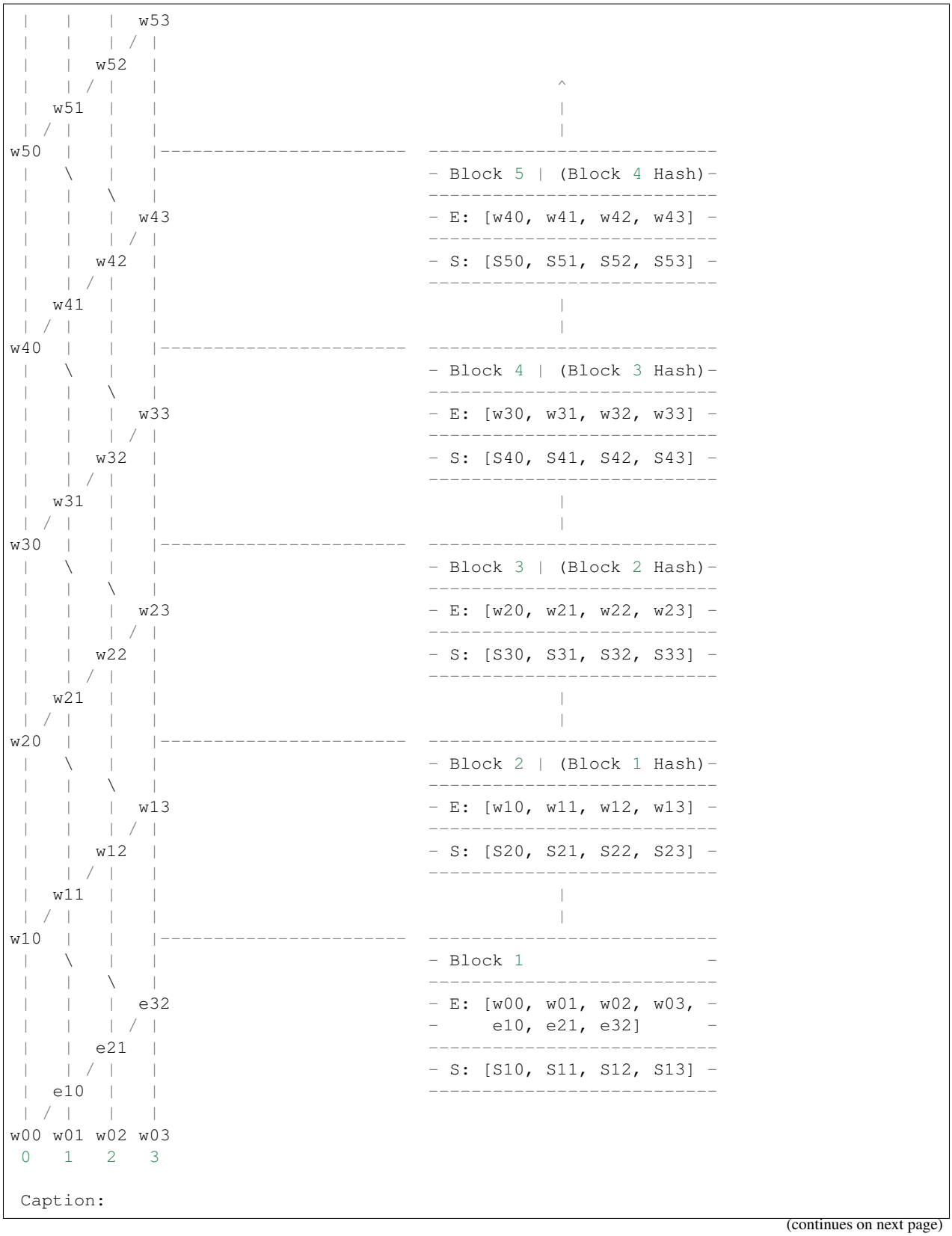
7.1 Motivation

The consumable output of any consensus system is an ordered list of transactions. Developers have been using blockchains to model such lists because they are efficient to work with. A linear data structure composed of batches of transactions, hashed and signed together, enabling easy verification of any transaction, is the right tool for the job. Although the word blockchain is now used in a much broader sense, it originally designated a data structure. Consensus algorithms, public/private networks, cryptocurrencies, etc., are independent concepts.

Hashgraph is a beautiful consensus algorithm based on a homonymous data structure. The hashgraph data structure, however, is not easy to work with when it comes to representing a linear sequence of transactions. It is a Directed Acyclic Graph (DAG) from which the order must be extracted via some complex consensus functions. To verify the consensus index of a given transaction, one has to re-compute the consensus methods on a subset of the hashgraph. On the other hand, blockchains do not need any further processing to extract the ordered list of transactions and simple cryptographic primitives are sufficient to validate blocks.

The "hashgraph vs blockchain" debate is a red herring. Blockchain is just a data structure; the engine is the underlying consensus algorithm. The projection method exposes an easy-to-work-with blockchain powered by the efficient Hashgraph consensus algorithm.

7.2 Implementation



(continues on next page)

(continued from previous page)

```
--------

E: List of Events contained in Block. Here, we mention Events because it is
easier to represent than transactions. Blocks would actually contain only
the transactions of the Events, but that is complicated to represent in this
diagram.

Sij: Signature of Block i by validator j
```

The Hashgraph algorithm always commits Events in batches. Indeed, when the fame of a super-majority of witnesses from a given round is decided, all the Events that are seen by all these famous witnesses (but not from an earlier round) get assigned the same *Round Received* and sorted according to a deterministic function. At that point, the consensus order of these Events is decided and will not change.

We gather the transactions of all the Events from the same *Round Received* into blocks. When Events get assigned a *Round Received* and sorted, we package their transactions (in canonical order) into a block and commit that block to the application. The application returns a hash of the state obtained by applying the block's transactions sequentially and we append this hash to the block's body before signing it. Block signatures will be exchanged as part of the regular gossip routine and appended to their corresponding blocks as they are received from other peers if they match the local block. Once a block has collected signatures from at least 1/3 of validators, it is deemed accepted because, by hypothesis, at least one of those signatures originates from an honest peer.

We extend the Event data structure to contain a set of block-signatures by the Event's creator. Having assigned a *RoundReceived* to a set of Events and produced a corresponding block, a member will append the block's signature in the next Event it defines. Hence, block-signatures piggy-back on the regular gossip messages and propagate at the same speed. Upon receiving Events from an other peer, a member will verify their block-signatures against its own version of the blocks; if the signatures match, they are recorded with the block. With this extended gossip routine, nodes simultaneously build up the hashgraph and the corresponding blockchain. It preserves the simplicity of the hashgraph system, which is one of its most valuable features, by not adding new types of messages; it only extends the existing Event data-structure.

By construction, the fame of a round R witness can only be decided by a witness in round R+2 or above. Hence, when a block is created for a *Round Received* R (block R), the hashgraph already contains Events at round R+2 or more; the signatures for block R, will be gossiped at the same time as Events of round R+2 or more. It follows that the signatures of block R will arrive with a lag of 2 or more consensus rounds.

7.3 Block Structure

```
Block: {
    Body:{
        Index                       int                            // block index
            RoundReceived               int                            // round␣
↪received of corresponding hashgraph frame
            Timestamp                   int64                          // unix␣
↪timestamp (median of timestamps in round-received famous witnesses)
            StateHash                   []byte                         // root hash of␣
↪the application after applying block payload; to be populated by application Commit
            FrameHash                   []byte                         // hash of␣
↪corresponding hashgraph frame
            PeersHash                   []byte                         // hash of peer-
↪set
            Transactions                [][]byte                       // transaction␣
↪payload
            InternalTransactions        []InternalTransaction          // internal␣
↪transaction payload (add/remove peers)
```

(continues on next page)

(continued from previous page)

```
            InternalTransactionReceipts []InternalTransactionReceipt // receipts for␣
↪internal transactions; to be populated by application Commit
    }
    Signatures: map[string]string
}
```

Blocks contain a body and a set of signatures. Signatures are based on the hash of the body; which is enough to verify the entire block because it contains a digital fingerprint of the body.

The Body's *RoundReceived* corresponds to the *RoundReceived* of the hashgraph Events who's transactions are included in the block; it serves the purpose of tying back to the underlying hashgraph. We do not produce a block when all the Events of a *Round Received* are empty. Hence, two consecutive blocks may have non-consecutive RoundReceived values and we use an additional property to index the blocks.

The 'Timestamp' is a Unix timestamp (number of seconds since January 1st, 1970) corresponding to the median of the timestamps of the famous witnesses in the frame's round-received. Upon creating a hashgraph Event, a Unix timestamp is automatically added to it using the creator's system clock. The Block's timestamp is the median of the timestamps included in the famous witnesses of the block's round-received. Note that nodes may have non-synchronised clocks, and may purposefuly tinker with their clocks to bias the block timestamp.

The FrameHash corresponds to the Frame in the hashgraph at RoundReceived. It is used in the FastSync protocol to verify the relationship between the Block and the Frame returned in a FastForwardResponse.

The body also contains a hash of the application's state resulting from applying the block's transactions sequentially. Thus, with the consenus algorithm and the necessary assumption that at least two thirds of participants are not compromised, collecting signatures from at least one third of validators provides sufficient evidence that all honest nodes have applied the same transactions in the same order, and computed the same state.

With the new Dynamic Membership protocol, which enables adding and removing peers dynamically, we added a PeersHash field to the body, to keep track of the validator-set. We can check the Frame's peer-set against the block's PeersHash to ensure that we are counting signatures from the appropriate peer-set.

InternalTransactions and InternalTransactionReceipts are used to track attempts to update the peer-set. InternalTransactions encode requests to join or leave the peer-set. Upon receiving a CommitBlock message, the application can accept or refuse InternalTransactions by returning correponding InternalTransactionReceipts.

7.4 Enhancements

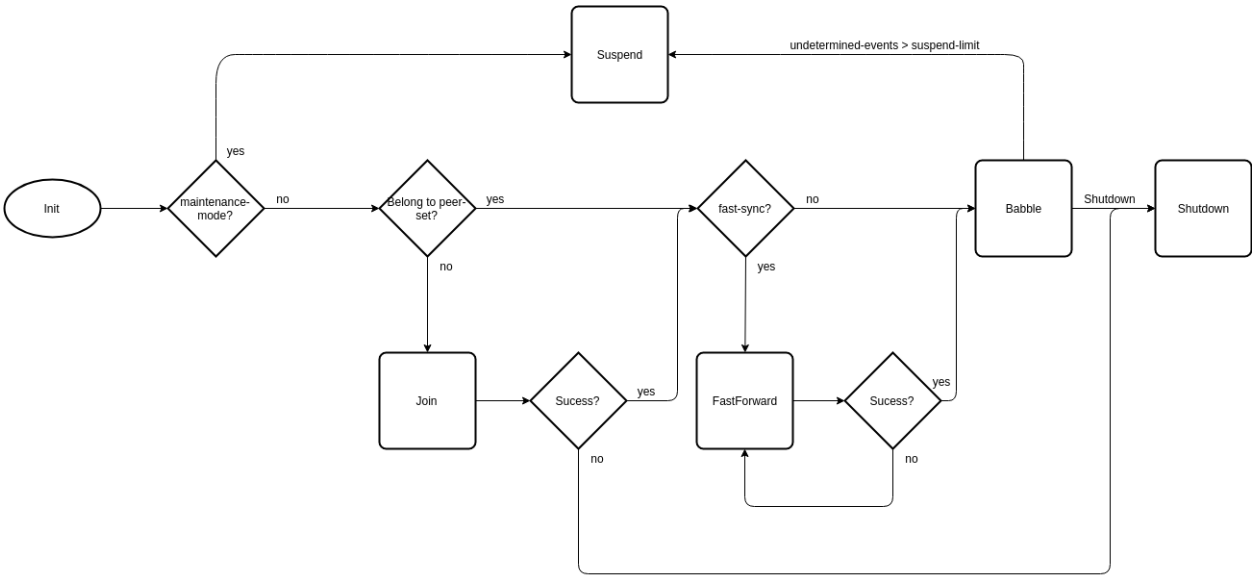
7.4.1 Inter-Blockchain Communication

Inter-Blockchain Communication (IBC) is about verifying on one chain that a transaction happened on another chain; one blockchain acts as a light-client to another blockchain. It is much simpler to build a light-client for a blockchain than for a hashgraph. In an effort to enable interoperability between blockchains, several initiatives have been proposed to build protocols for IBC like Cosmos, Polkadot and EOS. The projection method allows hashgraph-based systems to integrate with these network architectures.

CHAPTER 8

Node State-Machine

8.1 Flow

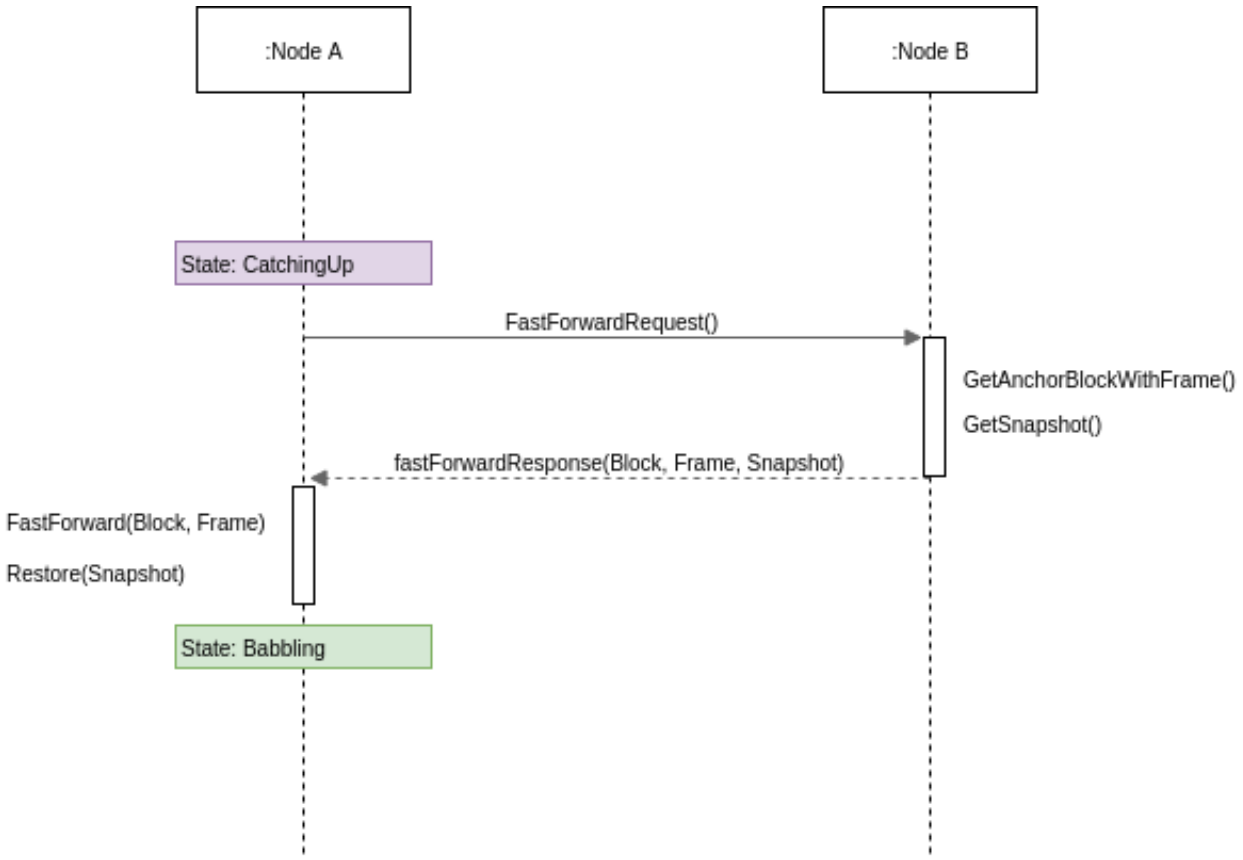


CHAPTER 9

FastSync

FastSync is an element of the Babble protocol which enables nodes to catch up with other nodes without downloading and processing the entire history of gossip (Hashgraph + Blockchain). It is important in the context of mobile ad hoc networks where users dynamically create or join groups, and where limited computing resources call for periodic pruning of the underlying data store. The solution relies on linking snapshots of the application state to independent and self-contained sections of the Hashgraph, called Frames. A node that fell back too far may fast-forward straight to the latest snapshot, initialize a new Hashgraph from the corresponding Frame, and get up to speed with the other nodes without downloading and processing all the transactions it missed. Of course, the protocol maintains the BFT properties of the base algorithm by packaging relevant data in signed blocks; here again we see the benefits of using a blockchain mapping on top of Hashgraph. Although implementing the Snapshot/Restore functionality puts extra strain on the application developer, it remains entirely optional; FastSync can be activated or deactivated via configuration.

9.1 Overview



The Babble node is implemented as a state-machine where the possible states are: **Babbling**, **CatchingUp**, **Joining**, **Leaving**, and **Shutdown**. When a node is started and belongs to the current validator-set, it will either enter the **Babbling** state, or the **CatchingUp** state, depending on whether the **fast-sync** flag was passed to Babble.

In the **CatchingUp** state, a node determines the best node to fast-sync from (the node which has the longest hashgraph) and attempts to fast-forward to their last consensus snapshot, until the operation succeeds. Hence, FastSync introduces a new type of command in the communication protocol: *FastForward*.

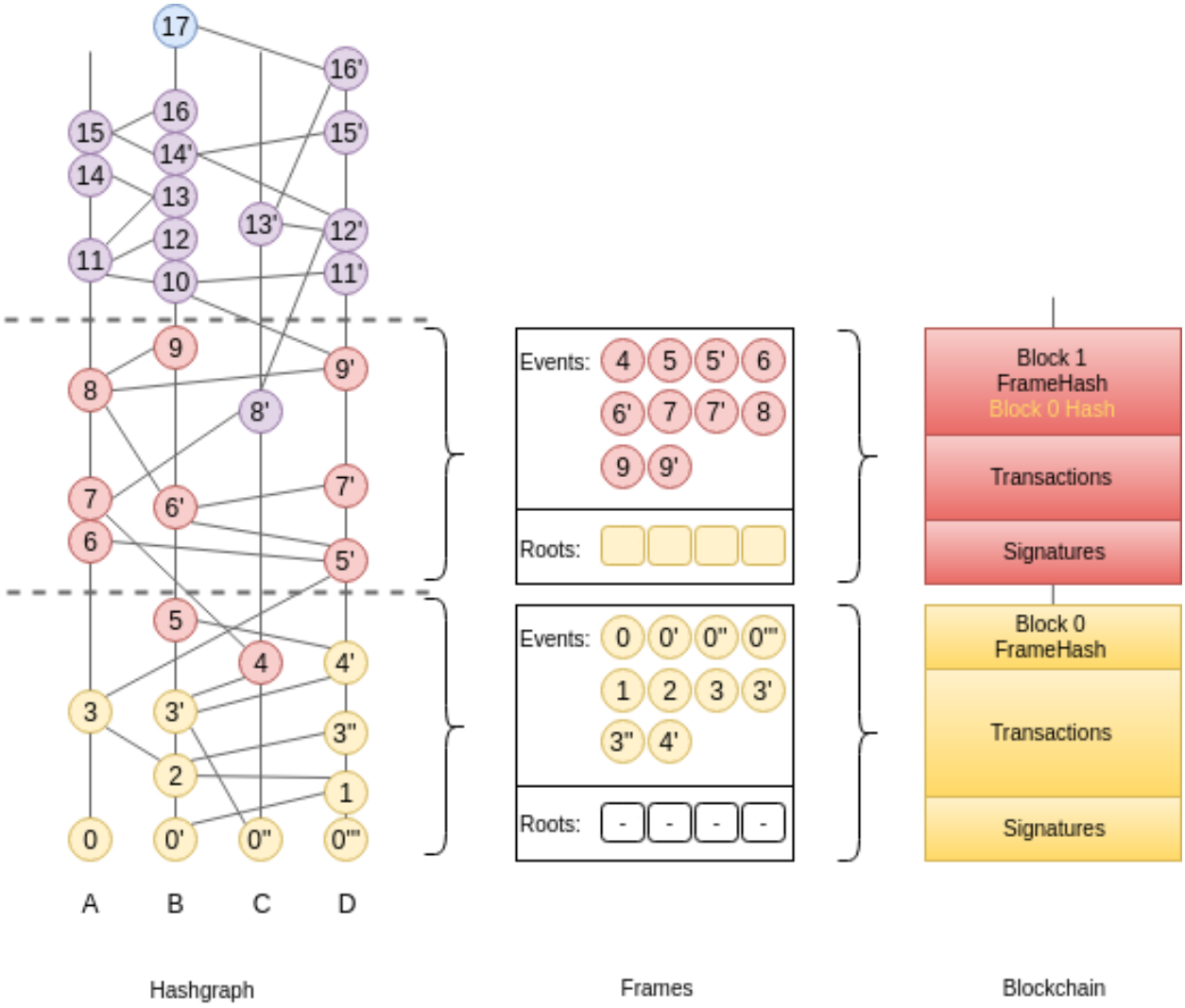
Upon receiving a FastForwardRequest, a node must respond with the last consensus snapshot, as well as the corresponding Hashgraph section (the Frame) and Block. With this information, and having verified the Block signatures against the other items as well as the known validator set, the requesting node attempts to reset its Hashgraph from the Frame, and restore the application from the snapshot. The difficulty resides in defining what is meant by *last consensus* snapshot, and how to package enough information in the Frames as to form a base for a new/pruned Hashgraph.

9.2 Frames

Frames are self-contained sections of the Hashgraph. They are composed of FrameEvents which wrap regular Hashgraph Events along with precomputed values for Round, Witness, and LamportTimestamp. Usually, these values would be calculated by every node locally but since FrameEvents belong to Blocks, which eventually collect enough signatures (>1/3), they can be used directly. Basically, Frames form a valid foundation for a new Hashgraph, such that gossip-about-gossip routines are not discontinued, while earlier records of the gossip history are ignored.

```
type Frame struct {
    Round    int //RoundReceived
    Peers    []*peers.Peer
    Roots    map[string]*Root
    Events   []*FrameEvent         //Events with RoundReceived = Round
    PeerSets map[int][]*peers.Peer //[round] => Peers
}
```

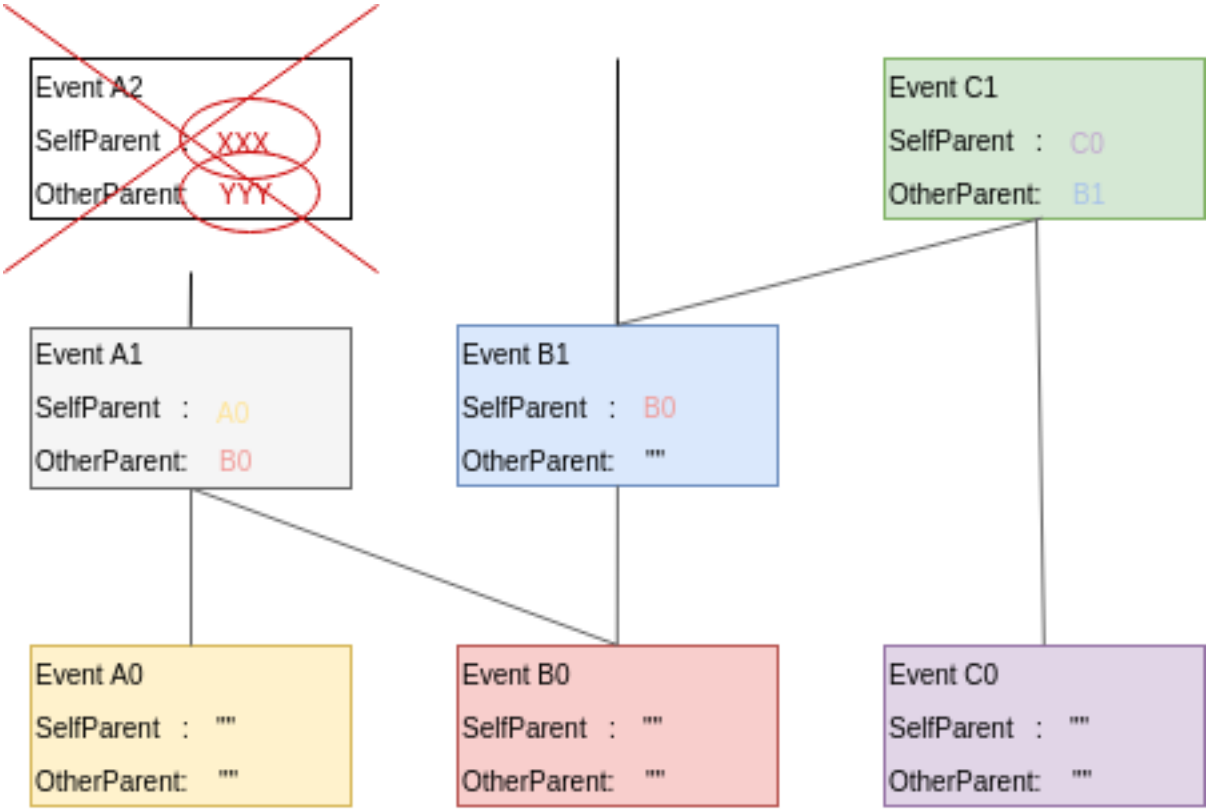
A Frame corresponds to a Hashgraph consensus round. Indeed, the consensus algorithm commits Events in batches, which we map onto Frames, and finally onto a Blockchain. This is an evolution of the previously defined *blockchain mapping*. Block headers now contain a Frame hash. As we will see later, this is useful for security. The Events in a Frame are the Events of the corresponding batch, in consensus order.



9.3 Roots

Frames also contain Roots, with a certain number of past FrameEvents for each particpant. Intuitively, "replanting" a Hashgraph requires deep enough roots. The Hashgraph is an interlinked chain of Events, where each Event contains two references to anterior Events (SelfParent and OtherParent). Upon inserting an Event in the Hashgraph, we check

that its references point to existing Events (Events that are already in the Hashgraph) and that at least the SelfParent reference is not empty. This is partially illustrated in the following picture where Event A2 cannot be inserted because its references are unknown.



If, when resetting from a Frame, we only used the Events associated with the corresponding round-received, we would quickly run into a situation where future Events reference unknown/older Events, and fail to be inserted. Hence, we need to package a "history" of past Events in the Frame, as a base layer for resetting the Hashgraph; this is contained in the Roots.

What matters is that every participant computes the same Roots, and that Roots contain sufficient information to keep inserting Events in a Reset hashgraph and compute a consensus order.

As of today, the number of FrameEvents in each Root (the root depth) is hard-coded to 10, to avoid nodes from using different values, which would result in different Blocks, and forks (partitions).

Note that there is still a possibility for an Event's OtherParent to refer to an Event "below" the Frame. This is possible due to the asynchronous nature of the gossip routines, but is an unlikely scenario. The Frame design tries to find a compromise between the size and the amount of useful information they contain. A root depth of 10 offers a high-enough probability of success.

9.4 FastForward

Frames may be used to initialize or reset a Hashgraph to a clean state, with indexes, rounds, blocks, etc., corresponding to a capture of a live run, such that further Events may be inserted and processed independently of past Events. Hashgraph Frames are loosely analogous to IFrames in video encoding, which enable fast-forwarding to any point in the video.

To avoid being tricked into fast-forwarding to an invalid state, the protocol ties Frames to the corresponding Blockchain by including Frame hashes in affiliated Block headers. A *FastForwardResponse* includes a Block and a Frame, such

that, upon receiving these objects, the requester may check the Frame hash against the Block header, and count the Block signatures against the **known** set of validators, before resetting the Hashgraph from the Frame.

Note the importance for the requester to be aware of the validator set of the Hashgraph it wishes to sync with; it is fundamental when it comes to verifying a Block. With a dynamic validator set, however, an additional mechanism will be necessary to securely track changes to the validator set.

9.5 Snapshot/Restore

It is one thing to catch-up with the Hashgraph and Blockchain, but nodes also need to catch-up with the application state. we extended the Proxy interface with methods to retrieve and restore snapshots.

```
type AppProxy interface {
  SubmitCh() chan []byte
  CommitBlock(block hashgraph.Block) (CommitResponse, error)
  GetSnapshot(blockIndex int) ([]byte, error)
  Restore(snapshot []byte) error
}
```

Since snapshots are raw byte arrays, it is up to the application layer to define what the snapshots represent, how they are encoded, and how they may be used to restore the application to a particular state. The *GetSnapshot* method takes a *blockIndex* parameter, which implies that the application should keep track of snapshots for every committed block. As the protocol evolves, we will likely link this to a *FrameRate* parameter to reduce the overhead on the application caused by the need to take all these snapshots.

So together with a Frame and the corresponding Block, a FastForward request comes with a snapshot of the application for the node to restore the application to the corresponding state. If the snapshot was incorrect, the node will immediately diverge from the main chain because it will obtain different state hashes upon committing new blocks.

9.6 Improvements and Further Work

The protocol is not entirely watertight yet; there are edge cases that could quickly lead to forks and diverging nodes.

1) Although it is unlikely, Events above the Frame that reference parents from "below" the Frame. These Events will fail to be inserted into the Hashgraph, and the node would stop making progress.

2) The snapshot is not directly linked to the Blockchain, only indirectly through resulting StateHashes.

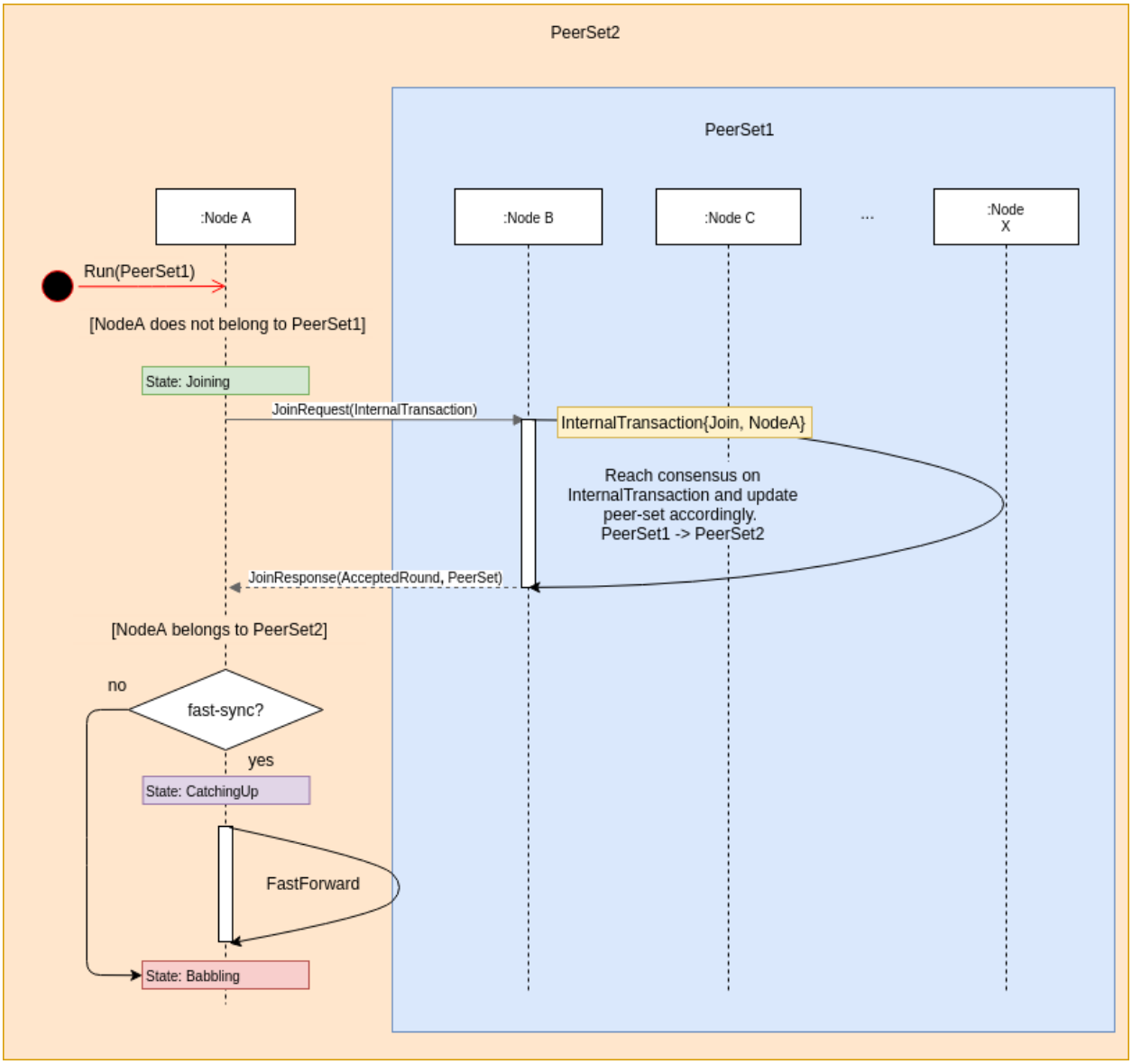
Both these issues could be addressed with a general retry mechanism, whereby the FastForward method is made atomic by working on a temporary copy of the Hashgraph. If an error or a fork are detected, try to FastSync again from another Frame. This requires further work and design on fork detection and self-healing protocols.

CHAPTER 10

Dynamic Membership

Dynamic Membership is an extension to the Babble protocol, which enables peers to join or leave a live cluster via consensus. Until now, we had only considered fixed peer-sets, where the list of participants was predetermined and unchanged throughout the life of a Babble network. This was an important limitation, hindering the formation of ad-hoc blockchains as we envision them. Here we present our solution, and its implementation, which relies on previous work regarding the Hashgraph-to-Blockchain projection, and FastSync.

10.1 Overview



A Babble node is started with a genesis peer-set (genesis.peers.json) and a current peer-set (peers.json). If its public-key does not belong to the current peer-set, the node will enter the `Joining` state, where it will attempt to join the peer-set. It does so by signing an `InternalTransaction` and submitting it for consensus via a `JoinRequest` to one of the current nodes.

The InternalTransaction is added to an Event, and goes through Babble consensus, until it is added to a block and committed. However, unlike regular transactions, the InternalTransaction is actually interpreted by Babble to modify the peer-set, if the application-layer accepts it. We shall see that, according to Hashgraph dynamics, an accepted InternalTransaction, committed with round-received R, only affects peer-sets for rounds R+6 and above.

If the JoinRequest was successful, and the new node makes it into the peer-set, it will either enter the `Babbling` state directly, or the `CatchingUp` state, depending on whether the `fast-sync` flag was provided. In the former case, the node will have to retrieve the entire history of the hashgraph and commit all the blocks it missed one-by-one. This is where it is important to have the genesis peer-set, to allow the joining node to validate the consensus decisions and peer-set changes from the beginning of the hashgraph.

The functionality for removing peers is almost identical, with the difference that there will be an automatic way of deciding when nodes should be removed, based on a minimum level of activity (ex: 10 rounds with no witnesses). As of today, a node submits a LeaveRequest for itself upon capturing a SIGINT signal when the Babble process is terminated cleanly.

10.2 InternalTransaction

In contrast with regular transactions, which only affect the application layer, InternalTransactions are internal to Babble. Babble acts upon InternalTransactions to modify part of its own state, the peer-set, rather than modifying the application's state. However, the application layer plays a role in accepting or refusing peer-set changes during the block commit phase. For example, the application could refuse all InternalTransactions (thereby preventing the peer-set from ever changing), or accept only up to N participants, or finally, it could base the decision on a predefined whitelist; anything goes, as long as the rule is deterministic (all nodes make the same decision).

```
type InternalTransactionBody struct {
  Type TransactionType
  Peer peers.Peer
}

type InternalTransaction struct {
  Body      InternalTransactionBody
  Signature string
}
```

The ProxyInterface, between Babble and the application-layer, is thus slightly extended to account for InternalTransactions. Here is a example of a CommitHandler that systematically accepts all InternalTransactions:

```
func (a *State) CommitHandler(block hashgraph.Block) (proxy.CommitResponse, error) {
  a.logger.WithField("block", block).Debug("CommitBlock")

  err := a.commit(block)
  if err != nil {
    return proxy.CommitResponse{}, err
  }

  receipts := []hashgraph.InternalTransactionReceipt{}
  for _, it := range block.InternalTransactions() {
    r := it.AsAccepted()
    receipts = append(receipts, r)
  }

  response := proxy.CommitResponse{
    StateHash:                  a.stateHash,
    InternalTransactionReceipts: receipts,
  }

  return response, nil
}
```

10.3 PeerSet

Until now, the peer-set has been a static list of peers; we now associate each Hashgraph round with a potentially different peer-set. We maintain a sorted table of round-to-peer-set associations, such that all rounds between two

consequent entries of this table are associated with the left-most peer-set. For example, if the table of peer-sets is [{0, PS1}, {5, PS2}, {12, PS3}], then rounds 0 to 4 will be associated to peer-set PS1, rounds 5 to 11 will be associated to PS2, and rounds 12 and above will be associated to PS3 (until the peer-set changes again).

We will see in the next section how to account for different peer-sets in the core consensus methods, but since they are allowed to change from one round to another, peer-sets must also be accounted for in the Frame and Block data-structures. Indeed, when verifying a Block, one must know which peer-set to count signatures against. Therefore, we have extended the Frame and Block objects to contain a PeerSetHash, that uniquely identifies the peer-set of the corresponding round-received. In the future, we will need to include a proof of peer-set change inside the Blocks, so that clients may follow and verify the evolution of the peer-set; ie, something that captures the following information:

```
PS0 + InternalTransaction0 + PS0-signatures(InternalTransaction0) => PS1
PS1 + InternalTransaction1 + PS1-signatures(InternalTransaction1) => PS2
...
PSN + InternalTransactionN + PSN-signatures(InternalTransactionN) => PSN+1
```

10.4 Algorithm Updates

There is no better documentation than the code itself, but here is a high level overview of what has changed. This section assumes familiarity with Babble, and Hashgraph.

10.4.1 StronglySee

Informally, StronglySee is the function that determines whether there is a path in the Hashgraph connecting two Events such that the path includes Events from a strong majority of participants. This obviously begs the question: "strong majority of which set of participants?". So we extended the StronglySee method with a PeerSet parameter.

10.4.2 Round

An Event's round is determined by taking the max of its parents rounds, and adding 1 if, and only if, the Event can strongly-see a super-majority of Witnesses from that round (max of the parents). So, in this call to StronglySee, we pass the peer-set corresponding to the max parent round, and the super-majority is counted based on the max parent round peer-set.

10.4.3 Witness

An Event is a witness if, and only if, it is a creator's first Event in its round AND its creator belongs to the round's peer-set.

10.4.4 Fame

With Dynamic Membership, different peer-sets may be involved in deciding the fame of a single witness. Although, Babble's implementation of the Hashgraph algorithm is slightly different, here are the changes that Dynamic Membership introduce in the algorithm as described in the original Hashgraph whitepaper:

```
for each event x in order from earlier rounds to later
  x . famous ← UNDECIDED
  for each event y in order from earlier rounds to later
      if x . witness and y . witness and y . round>x . round
```

(continues on next page)

(continued from previous page)

```
        d ← y . round - x . round
        s ← the set of witness events in round y . round -1 that y can strongly see
**      [based on y.round-1 peer-set]
        v ← majority vote in s ( is TRUE for a tie )
        t ← number of events in s with a vote of v

        if d = 1 // first round of the election
            y . vote ← can y see x ?
        else
**          [n ←  number of peers in y.round peer-set]
            if d mod c > 0 // this is a normal round
                if t > 2* n /3 // if supermajority, then decide
                    x . famous ← v
                    y . vote ← v
                    break out of the y loop
                else // else, just vote
                    y . vote ← v
            else // this is a coin round
                if t > 2* n /3 // if supermajority, then vote
                    y . vote ← v
                else // else flip a coin
                    y . vote ← middle bit of y . signature
```

10.4.5 R+6

When an InternalTransaction is committed, when should we start counting the new peer-set in order to guarantee that all correct nodes will do the same thing? The answer is R+6 where R is the round-received of the Event containing the InternalTransaction.

We need only determine the lower-bound because the goal is obviously to change the peer-set as soon as possible.

The solution is basically contained in Lemmas 5.15 and 5.17 of the original hashgraph whitepaper:

> Lemma 5.15. If hashgraphs A and B are consistent, and A decides a Byzantine agreement election with result v in round r and B has not decided prior to r, then B will decide v in round r + 2 or before.
>
> Lemma 5.17. For any round number r, for any hashgraph that has at least one event in round r+3, there will be at least one witness in round r that will be decided to be famous by the consensus algorithm, and this decision will be made by every witness in round r + 3, or earlier.

If one hashgraph decides RoundReceived = R, then a strong majority of round R witnesses are decided, and by Lemma 5.17 they are necessarily decided in round R+3 or earlier. Hence, by Lemma 5.15, any other consistent hashgraph will have decided by round R + 5 or earlier. It is then safe to set the new peer-set for round R + 6.