Babble Documentation

Release 0

Mosaic Networks

Dec 19, 2018

Contents

CHAPTER 1

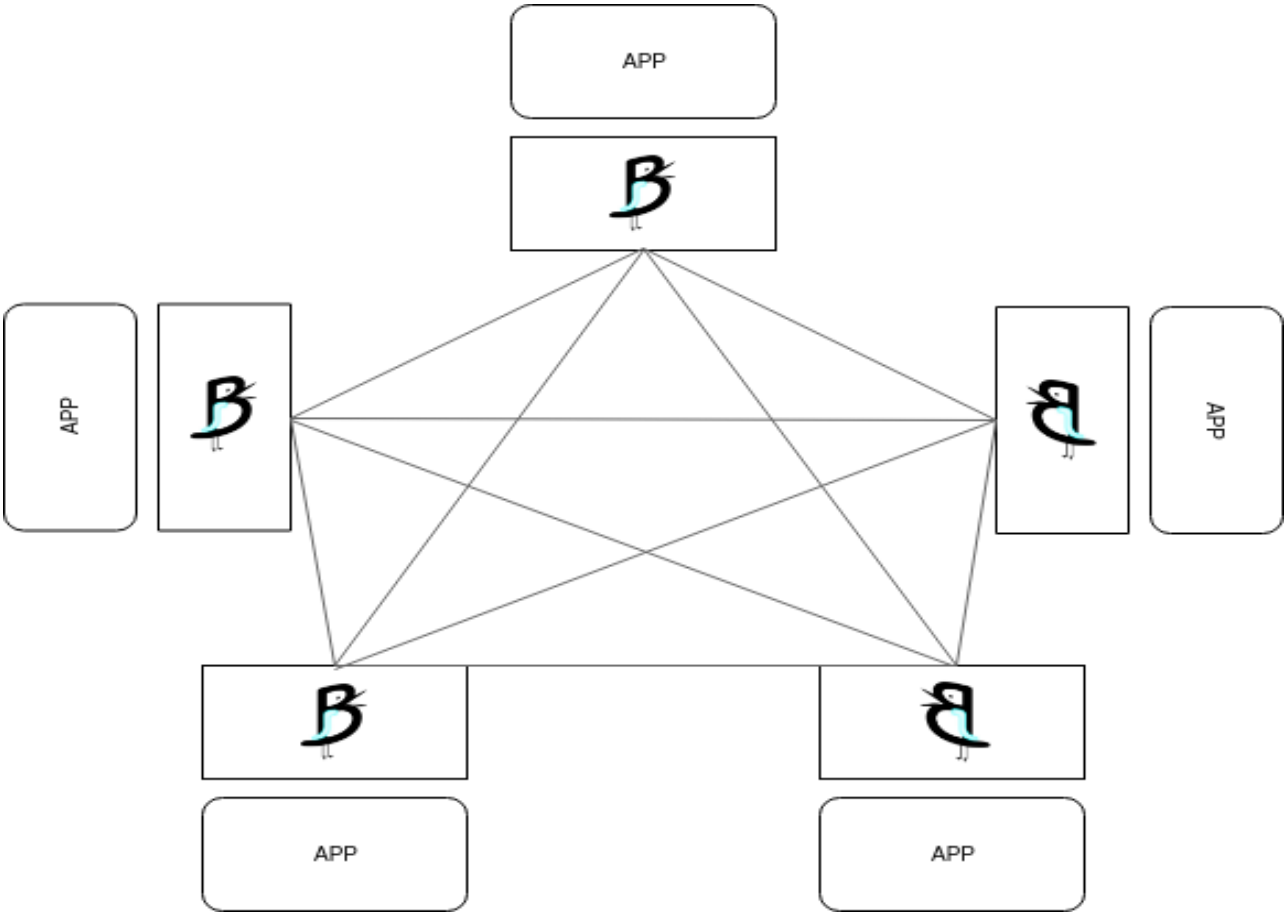
Babble

1.1 Introduction

1.1.1 What is Babble?

Babble is an open-source software component intended for developers who want to build peer-to-peer (p2p) applications, mobile or other, without having to implement their own p2p networking layer from scratch. Under the hood, it enables many computers to behave as one; a technique known as state machine replication.

Babble is designed to easily plug into applications written in any programming language. Developers can focus on building the application logic and simply integrate with Babble to handle the replication aspect. Basically, Babble will connect to other Babble nodes and guarantee that everyone processes the same commands in the same order. To do this, it uses p2p networking and a Byzantine Fault Tolerant (BFT) consensus algorithm.



Babble is:

- **Asynchronous:** Participants have the freedom to process commands at different times.
- **Leaderless:** No participant plays a 'special' role.
- **Byzantine Fault-Tolerant:** Supports one third of faulty nodes, including malicious behavior.
- **Final:** Babble's output can be used immediately, no need for block confirmations, etc.

1.2 Install

1.2.1 From Source

Clone the repository in the appropriate GOPATH subdirectory:

```
$ mkdir -p $GOPATH/src/github.com/mosaicnetworks/
$ cd $GOPATH/src/github.com/mosaicnetworks
[...]/mosaicnetworks$ git clone https://github.com/mosaicnetworks/babble.git
```

The easiest way to build binaries is to do so in a hermetic Docker container. Use this simple command:

```
[...]/babble$ make dist
```

This will launch the build in a Docker container and write all the artifacts in the build/ folder.

```
[...]/babble$ tree build
build/
├── dist
│   ├── babble_0.1.0_darwin_386.zip
│   ├── babble_0.1.0_darwin_amd64.zip
│   ├── babble_0.1.0_freebsd_386.zip
│   ├── babble_0.1.0_freebsd_arm.zip
│   ├── babble_0.1.0_linux_386.zip
│   ├── babble_0.1.0_linux_amd64.zip
│   ├── babble_0.1.0_linux_arm.zip
│   ├── babble_0.1.0_SHA256SUMS
│   ├── babble_0.1.0_windows_386.zip
│   └── babble_0.1.0_windows_amd64.zip
└── pkg
    ├── darwin_386
    │   └── babble
    ├── darwin_amd64
    │   └── babble
    ├── freebsd_386
    │   └── babble
    ├── freebsd_arm
    │   └── babble
    ├── linux_386
    │   └── babble
    ├── linux_amd64
    │   └── babble
    ├── linux_arm
    │   └── babble
    ├── windows_386
    │   └── babble.exe
    └── windows_amd64
        └── babble.exe
```

1.2.2 Go Devs

Babble is written in Golang. Hence, the first step is to install **Go version 1.9 or above** which is both the programming language and a CLI tool for managing Go code. Go is very opinionated and will require you to define a workspace where all your go code will reside.

1.2.3 Dependencies

Babble uses Glide to manage dependencies. For Ubuntu users:

```
[...]/babble$ curl https://glide.sh/get | sh
[...]/babble$ glide install
```

This will download all dependencies and put them in the **vendor** folder.

1.2.4 Testing

Babble has extensive unit-testing. Use the Go tool to run tests:

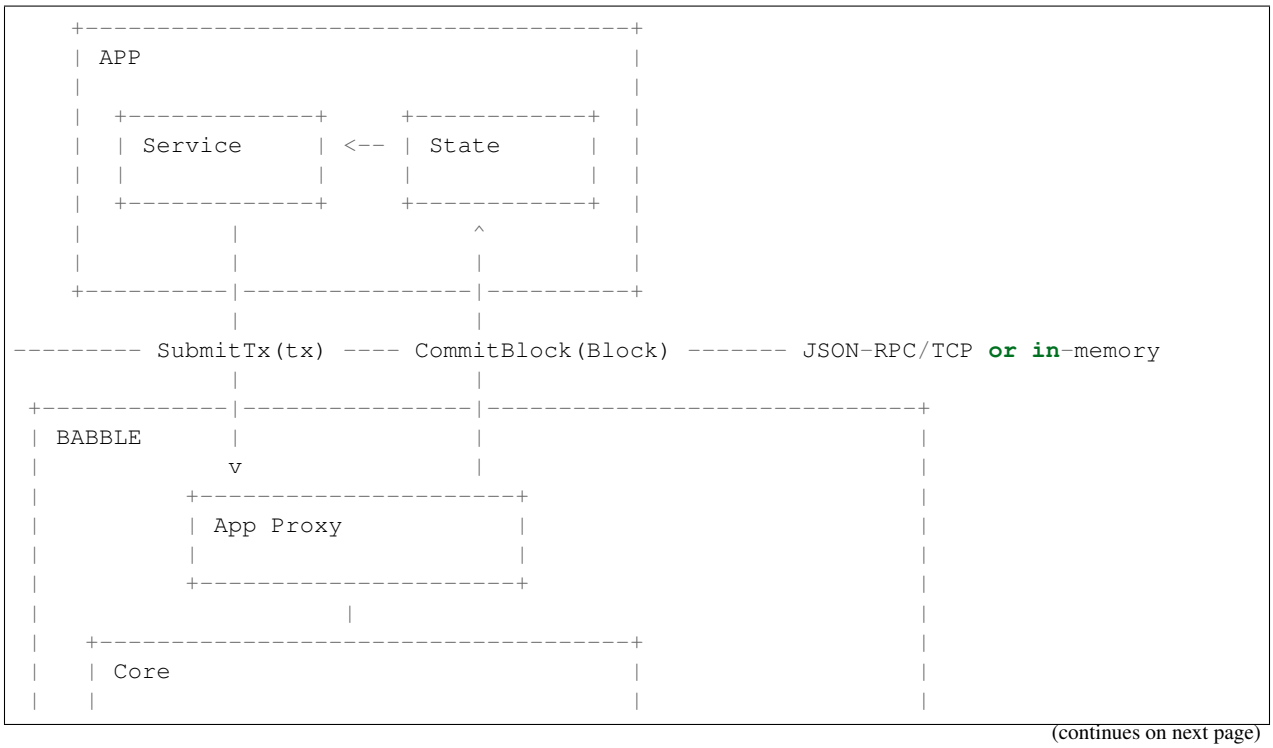
```
[...]/babble$ make test
```

If everything goes well, it should output something along these lines:

```
?       github.com/mosaicnetworks/babble/src/babble     [no test files]
ok      github.com/mosaicnetworks/babble/src/common     0.015s
ok      github.com/mosaicnetworks/babble/src/crypto     0.122s
ok      github.com/mosaicnetworks/babble/src/hashgraph  10.270s
?       github.com/mosaicnetworks/babble/src/mobile     [no test files]
ok      github.com/mosaicnetworks/babble/src/net        0.012s
ok      github.com/mosaicnetworks/babble/src/node       19.171s
ok      github.com/mosaicnetworks/babble/src/peers      0.038s
?       github.com/mosaicnetworks/babble/src/proxy      [no test files]
ok      github.com/mosaicnetworks/babble/src/proxy/dummy        0.013s
ok      github.com/mosaicnetworks/babble/src/proxy/inmem        0.037s
ok      github.com/mosaicnetworks/babble/src/proxy/socket       0.009s
?       github.com/mosaicnetworks/babble/src/proxy/socket/app   [no test files]
?       github.com/mosaicnetworks/babble/src/proxy/socket/babble        [no test␣
↪files]
?       github.com/mosaicnetworks/babble/src/service    [no test files]
?       github.com/mosaicnetworks/babble/src/version    [no test files]
?       github.com/mosaicnetworks/babble/cmd/babble     [no test files]
?       github.com/mosaicnetworks/babble/cmd/babble/commands    [no test files]
?       github.com/mosaicnetworks/babble/cmd/dummy      [no test files]
?       github.com/mosaicnetworks/babble/cmd/dummy/commands     [no test files]
```

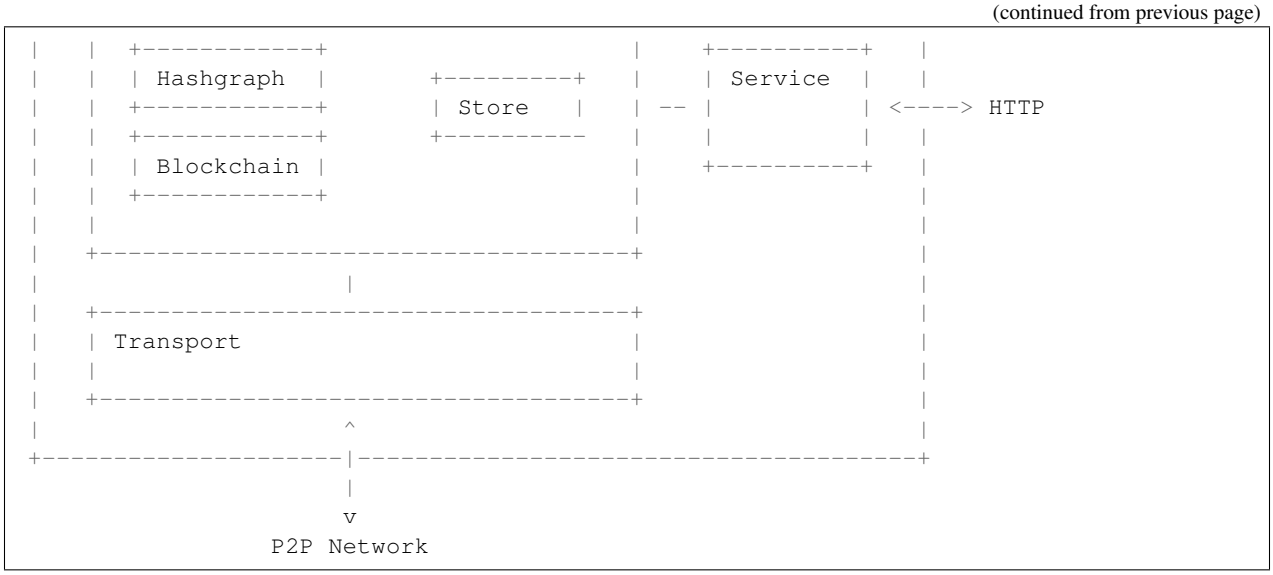
1.3 Design

1.3.1 Overview



(continues on next page)

(continued from previous page)



Almost any software application can be modeled in terms of a *service* and a *state*. The *service* is responsible for processing commands (ex. user input), while the *state* is responsible for manipulating and storing the data (eg. database). Usually, when commands require updating the data, the *service* will invoke the *state* directly. In a distributed application, however, commands (referred to as *transactions* in this context), must be broadcasted to all replicas and put in a common order before being applied to the *state*. This is what ensures that all replicas process the same commands in the same order. Hence, the *service* no longer communicates directly with the *state* (except for read-only requests), but forwards commands to a *transaction ordering system* which takes care of broadcasting and ordering the transactions across all replicas before feeding them back to the application's *state*.

Babble is an ordering system that plugs into any application thanks to a very simple interface. It uses a consensus algorithm, to replicate and order the transactions, and a blockchain to represent the resulting list. A blockchain is a linear data structure composed of batches of transactions, hashed and signed together, easily allowing to verify any transaction. So, instead of applying commands directly to the *state*, Babble applications must forward the commands to Babble and let them be processed asynchronously by the consensus system before receiving them back, in blocks, ready to be applied to the *state*.

1.3.2 Consensus and Blockchain

At the core of Babble is an algorithm insuring that all participants process the same transactions in the same order. We have chosen to implement a system suitable for the most adversarial conditions - with powerful attackers. This is known as Byzantine Fault Tolerance (BFT) and has been a field of research for over 30 years. Our system is based on our interpretation of a paper published by Leemon Baird in 2016 http://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf which describes a method for extracting a consensus order of events from a data structure representing the history of gossip between a set of participants. Instead of only gossiping transactions and exchanging votes directly, members gossip about gossip itself; something that can be represented in a Directed Acyclic Graph (DAG) - the hashgraph. The algorithm for extracting a consensus order from the hashgraph is proven to work and attains the theoretical limits of Byzantine fault-tolerance in terms of the number and power of the malicious members it can cope with. The messaging routines required to gossip and create the hashgraph are very simple compared to other BFT algorithms. The hashgraph itself, however, is not ideal for representing the ordered list of transactions because it is a two-dimensional object which doesn't explicitly express a linear order of the items it contains. Hence, we developed a method to map the hashgraph onto a blockchain.

A blockchain is a data structure where transactions are packaged in hash-linked blocks. Each block is identified by a cryptographic hash and contains a hash of the previous block, so that blocks can form a chain from the first block ever to the last formed block. In this way, all the block - and transactions - are connected via a linear linked list structure.

In our system, blocks contain a collection of signatures of their own hash from the participants. A block with valid signatures from at least one third of validators can be considered valid because - by hypothesis - at least one of those signatures is from an honest member.

Projecting the hashgraph onto a blockchain makes it much easier for third parties to verify the consensus order. It makes it possible to build light-clients and to integrate Hashgraph based systems with other blockchains. For more detail about the projection method, please refer to *From Hashgraph to Blockchain*

1.3.3 Proxy

Babble communicates with the App through an *AppProxy* interface, which has two implementations:

- `SocketProxy`: A SocketProxy connects to an App via TCP sockets. It enables the application to run in a separate process or machine, and to be written in any programming language.
- `InmemProxy`: An InmemProxy uses native callback handlers to integrate Babble as a regular Go dependency.

The `AppProxy` interface exposes three methods for Babble to call the App:

- `CommitBlock(Block) ([]byte, error)`: Commits a block to the application and returns the resulting state hash.
- `GetSnapshot(int) ([]byte, error)`: Gets the application snapshot corresponding to a particular block index.
- `Restore([]byte) error`: Restores the App state from a snapshot.

Reciprocally, `AppProxy` relays transactions from the App to Babble via a native Go channel - `SubmitCh` - which ties into the application differently depending on the type of proxy (Socket or Inmem).

Babble asynchronously processes transactions and eventually feeds them back to the App, in consensus order and bundled into blocks, with a **CommitBlock** call. Transactions are just raw bytes and Babble does not need to know what they represent. Therefore, encoding and decoding transactions is done by the App.

See the *API* section for more details about the Proxy API.

1.3.4 Transport

Babble nodes communicate with other Babble nodes in a fully connected Peer To Peer network. Nodes gossip by repeatedly choosing another node at random and telling eachother what they know about the hashgraph. The gossip protocol is extremely simple and serves the dual purpose of gossiping about transactions and about the gossip itself (the hashgraph). The hashgraph contains enough information to compute a consensus ordering of transactions.

The communication mechanism is a custom RPC protocol over TCP connections. It implements a Pull-Push gossip system. At the moment, there are two types of RPC commands: **Sync** and **EagerSync**. When node **A** wants to sync with node **B**, it sends a **SyncRequest** to **B** containing a description of what it knows about the hashgraph. **B** computes what it knows that **A** doesn't know and returns a **SyncResponse** with the corresponding events in topological order. Upon receiving the **SyncResponse**, **A** updates its hashgraph accordingly and calculates the consensus order. Then, **A** sends an **EagerSyncRequest** to **B** with the Events that it knows and **B** doesn't. Upon receiving the **EagerSyncRequest**, **B** updates its hashgraph and runs the consensus methods.

The list of peers must be predefined and known to all peers. At the moment, it is not possible to dynamically modify the list of peers while the network is running but this is not a limitation of the Hashgraph algorithm, just an implementation prioritization.

1.3.5 Core

The core of Babble is the component that maintains and computes the hashgraph. The consensus algorithm, invented by Leemon Baird, is best described in the white-paper and its accompanying document.

The hashgraph itself is a data structure that contains all the information about the history of the gossip and thereby grows and grows in size as gossip spreads. There are various strategies to keep the size of the hashgraph limited. In our implementation, the **Hashgraph** object has a dependency on a **Store** object which contains the actual data and is abstracted behind an interface.

There are currently two implementations of the **Store** interface. The `InmemStore` uses a set of in-memory LRU caches which can be extended to persist stale items to disk and the size of the LRU caches is configurable. The `BadgerStore` is a wrapper around this cache that also persists objects to a key-value store on disk. The database produced by the `BadgerStore` can be reused to bootstrap a node back to a specific state.

1.3.6 Service

The Service exposes an HTTP API to query information about the state of the node as well as the underlying hashgraph and blockchain. At the moment, it services two queries:

[GET] /stats:

Returns a map with information about the Babble node.

```
$curl -s http://[ip]:80/stats | jq
{
    "consensus_events": "145",
    "consensus_transactions": "100",
    "events_per_second": "0.00",
    "id": "1",
    "last_block_index": "4",
    "last_consensus_round": "14",
    "num_peers": "3",
    "round_events": "18",
    "rounds_per_second": "0.00",
    "state": "Babbling",
    "sync_rate": "1.00",
    "transaction_pool": "0",
    "undetermined_events": "22"
}
```

[GET] /block/{block_index}:

Returns the Block with the specified index, as stored by the Babble node.

```
$curl -s http://[ip]:80/block/0 | jq
{
  "Body": {
    "Index": 0,
    "RoundReceived": 7,
    "StateHash": "ib8wpBS/W18OT07R+HFxBVYjS/lwPPRtuAV/rcrpQ9w=",
    "FrameHash": "T7EVNhAfbIx3jGyu5fXnyYs+eAihWCxFdu+8UDYOzfA=",
    "Transactions": [
      "Tm9kZTEgVHgx",
      "Tm9kZTEgVHgy",
      "Tm9kZTEgVHgz",
      "Tm9kZTEgVHg0",
```

(continues on next page)

(continued from previous page)

```
      "Tm9kZTEgVHg1",
      "Tm9kZTEgVHg2",
      "Tm9kZTEgVHg3",
      "Tm9kZTEgVHg4",
      "Tm9kZTEgVHg5",
      "Tm9kZTEgVHgxMA==",
      "Tm9kZTEgVHgxOA==",
      "Tm9kZTEgVHgxMQ==",
      "Tm9kZTEgVHgxMg==",
      "Tm9kZTEgVHgxMw==",
      "Tm9kZTEgVHgxNA==",
      "Tm9kZTEgVHgxNQ==",
      "Tm9kZTEgVHgxNg==",
      "Tm9kZTEgVHgxNw=="
    ]
  },
  "Signatures": {

↪"0x0442633367F4F3C3B00533956CF5231600EB5622765A064C0BFCC547611293F3353BE2404D01FBF66184DB486C92F50
↪":
↪"2a2wij946jjhb0nnqcqspk5m3irnw6pyqevsgl833urt453nwq|50npyfnd9c2whz8gqe3jv4ya1qu2if3s25qofuah8565au
↪",

↪"0x04C1795E3C6C66CA3DF09C89FAC9FD5AC1BFF7C8BFE7D1DEF7CEC1A3BD9162F37CE841EE5ACE29B65486DD8EA976D5D
↪":
↪"636m75hq7vmz66vgscosrvhv3ultq1ndh477h3hx8oa38qkxkm|611yf6veodg7kwedt99kuuftjzturj8sowu2c1b65e323u
↪",

↪"0x04C8754230AF8F4A3491E16B8508E7D4C6944E496C95E0F6CF2B21ABBDD7BF9768E3F63B63166CE20FF8B7AF8B29C57
↪":
↪"39u9n7nk31nsyjsnrclcvtgo2emx3hu8gpsvfdzy497bbwaoam|69sl3o2rvy9fqant3ui86pffqcdb6tofhp1padlc011oyu
↪",

↪"0x04F753E04757A4D6ABC5741AC80D5CC98D5CE8F68C15104D73C447835D51A7840805614A221FD72C069C3D54E92FC8D
↪":
↪"1ajuve68asea9ydczz7j1vbi4p1rs4svzbyjwkxc0dswppmw7j|353mq56tycr44mmzzr5j5zs3mjwz74g5eladozhbwojfkk
↪"
  }
}
```

1.4 API

As mentioned in the *Design* section, Babble communicates with the App through an `AppProxy` interface, which exposes three methods for Babble to call the App. Here we explain how to implement this API.

Note: The Snapshot and Restore methods of the API are still work in progress. They are necessary for the *FastSync* protocol which is not completely ready yet. It is safe to just implement stubs for these methods.

1.4.1 Inmem

The `InmemProxy` uses native callback handlers to enable Babble to call methods on the App directly. Applications need only implement the `ProxyHandler` interface and pass that to an `InmemProxy`.

Here is a quick example of how to use Babble as an in-memory engine (in the same process as your handler):

```
package main

import (
	"github.com/mosaicnetworks/babble/src/babble"
	"github.com/mosaicnetworks/babble/src/crypto"
	"github.com/mosaicnetworks/babble/src/hashgraph"
	"github.com/mosaicnetworks/babble/src/proxy/inmem"
)

// Implements proxy.ProxyHandler interface
type Handler struct {
	stateHash []byte
}

// Called when a new block is committed by Babble. This particular example
// just computes the stateHash incrementaly with incoming blocks.
func (h *Handler) CommitHandler(block hashgraph.Block) (stateHash []byte, err error) {
	hash := h.stateHash

	for _, tx := range block.Transactions() {
		hash = crypto.SimpleHashFromTwoHashes(hash, crypto.SHA256(tx))
	}

	h.stateHash = hash

	return h.stateHash, nil
}

// Called when syncing with the network
func (h *Handler) SnapshotHandler(blockIndex int) (snapshot []byte, err error) {
	return []byte{}, nil
}

// Called when syncing with the network
func (h *Handler) RestoreHandler(snapshot []byte) (stateHash []byte, err error) {
	return []byte{}, nil
}

func NewHandler() *Handler {
	return &Handler{}
}

func main() {

	config := babble.NewDefaultConfig()

	// To use babble as an internal engine we use InmemProxy.
	proxy := inmem.NewInmemProxy(NewHandler(), config.Logger)

	config.Proxy = proxy

	// Create the engine with the provided config
	engine := babble.NewBabble(config)

	// Initialize the engine
	if err := engine.Init(); err != nil {
		panic(err)
```

(continues on next page)

(continued from previous page)

```
	}

	// Submit a transaction directly through the Proxy
	go func() { proxy.SubmitTx([]byte("some content")) }()

	// This is a blocking call
	engine.Run()
}
```

1.4.2 Socket

The `SocketProxy` is simply a TCP server that accepts *SubmitTx* requests, and calls remote methods on the App through a JSON-RPC interface. The App is therefore expected to implement its own component to send out SubmitTx requests through TCP, and receive JSON-RPC messages from the remote Babble node.

The advantage of using a TCP interface is that it provides the freedom to implement the application in any programming language. The specification of the JSON-RPC interface is provided below, but here is an example of how to use our Go implementation, `SocketBabbleProxy`, to connect to a remote Babble node.

Assuming there is a Babble node running with its proxy listening on `127.0.0.1:1338` and configured to speak to an App at `127.0.0.1:1339` (these are the default values):

```
package main

import (
	"time"

	"github.com/mosaicnetworks/babble/src/crypto"
	"github.com/mosaicnetworks/babble/src/hashgraph"
	"github.com/mosaicnetworks/babble/src/proxy/socket/babble"
)

// Implements proxy.ProxyHandler interface
type Handler struct {
	stateHash []byte
}

// Called when a new block is comming. This particular example just computes
// the stateHash incrementaly with incoming blocks
func (h *Handler) CommitHandler(block hashgraph.Block) (stateHash []byte, err error) {
	hash := h.stateHash

	for _, tx := range block.Transactions() {
		hash = crypto.SimpleHashFromTwoHashes(hash, crypto.SHA256(tx))
	}

	h.stateHash = hash

	return h.stateHash, nil
}

// Called when syncing with the network
func (h *Handler) SnapshotHandler(blockIndex int) (snapshot []byte, err error) {
	return []byte{}, nil
}
```

(continues on next page)

(continued from previous page)

```
// Called when syncing with the network
func (h *Handler) RestoreHandler(snapshot []byte) (stateHash []byte, err error) {
      return []byte{}, nil
}

func NewHandler() *Handler {
      return &Handler{}
}

func main() {
      // Connect to the babble proxy at :1338 and listen on :1339.
      // The Handler ties back to the application state.
      proxy, err := babble.NewSocketBabbleProxy("127.0.0.1:1338", "127.0.0.1:1339",␣
↪NewHandler(), 1*time.Second, nil)

      // Verify that it can listen
      if err != nil {
            panic(err)
      }

      // Verify that it can connect and submit a transaction
      if err := proxy.SubmitTx([]byte("some content")); err != nil {
            panic(err)
      }

      // Wait indefinitly
      for {
            time.Sleep(time.Second)
      }
}
```

Example SubmitTx request (from App to Babble):

```
request: {"method":"Babble.SubmitTx","params":["Y2xpZW50IDE6IGhlbGxv"],"id":0}
response: {"id":0,"result":true,"error":null}
```

Note that the Proxy API is **not** over HTTP; It is raw JSON over TCP. Here is an example of how to make a SubmitTx request manually:

```
printf "{\"method\":\"Babble.SubmitTx\",\"params\":[\"Y2xpZW50IDE6IGhlbGxv\"],\"id\
↪":0}" | nc -v  172.77.5.1 1338
```

Example CommitBlock request (from Babble to App):

```
request:
      {
          "method": "State.CommitBlock",
          "params": [
              {
              "Body": {
                  "Index": 0,
                  "RoundReceived": 7,
                  "StateHash": null,
                  "FrameHash": "gdwRCdwxoyLUyzzRK6N31rlJFBJu5By/vDk5gSQHJHQ=",
                  "Transactions": [
```

(continues on next page)

(continued from previous page)

```
                    "Tm9kZTEgVHg5",
                    "Tm9kZTEgVHgx",
                    "Tm9kZTEgVHgy",
                    "Tm9kZTEgVHgz",
                    "Tm9kZTEgVHg0",
                    "Tm9kZTEgVHg1",
                    "Tm9kZTEgVHg2",
                    "Tm9kZTEgVHg3",
                    "Tm9kZTEgVHg4",
                    "Tm9kZTEgVHgxMA=="
                    ]
                },
                "Signatures": {}
                }
            ],
            "id": 0
        }

response: {"id":0,"result":{"Hash":"6SKQataObI6oSY5n6mvf1swZR3T4Tek+C8yJmGijF00="},
↪"error":null}
```

The content of the request's "params" is the JSON representation of a Block with a RoundReceived of 7 and 10 transactions. The transactions themselves are base64 string encodings.

The response's Hash value is the base64 representation of the application's State-hash resulting from processing the block's transaction sequentially.

1.5 Usage

In this section we will guide you through deploying an application on top of Babble. Babble comes with the Dummy application which is used in this demonstration. It is a simple chat application where participants write messages on a channel and Babble guarantees that everyone sees the same messages in the same order.

1.5.1 Docker

We have provided a series of scripts to bootstrap a demo. Let us first use the easy method to view the demo and then we will take a closer look at what is happening behind the scenes.

Make sure you have Docker installed.

The demo will pull Docker images from our official public Docker registry

```
[...]/babble$ cd demo
[...]/babble/demo$ make
```

Once the testnet is started, a script is automatically launched to monitor consensus figures:

```
consensus_events:180 consensus_transactions:40 events_per_second:0.00 id:1 last_block_
↪index:3 last_consensus_round:17 num_peers:3 round_events:7 rounds_per_second:0.00␣
↪state:Babbling sync_rate:1.00 transaction_pool:0 undetermined_events:18
consensus_events:180 consensus_transactions:40 events_per_second:0.00 id:3 last_block_
↪index:3 last_consensus_round:17 num_peers:3 round_events:7 rounds_per_second:0.00␣
↪state:Babbling sync_rate:1.00 transaction_pool:0 undetermined_events:20
```

(continues on next page)

(continued from previous page)

```
consensus_events:180 consensus_transactions:40 events_per_second:0.00 id:2 last_block_
↪index:3 last_consensus_round:17 num_peers:3 round_events:7 rounds_per_second:0.00␣
↪state:Babbling sync_rate:1.00 transaction_pool:0 undetermined_events:21
consensus_events:180 consensus_transactions:40 events_per_second:0.00 id:0 last_block_
↪index:3 last_consensus_round:17 num_peers:3 round_events:7 rounds_per_second:0.00␣
↪state:Babbling sync_rate:1.00 transaction_pool:0 undetermined_events:20
```

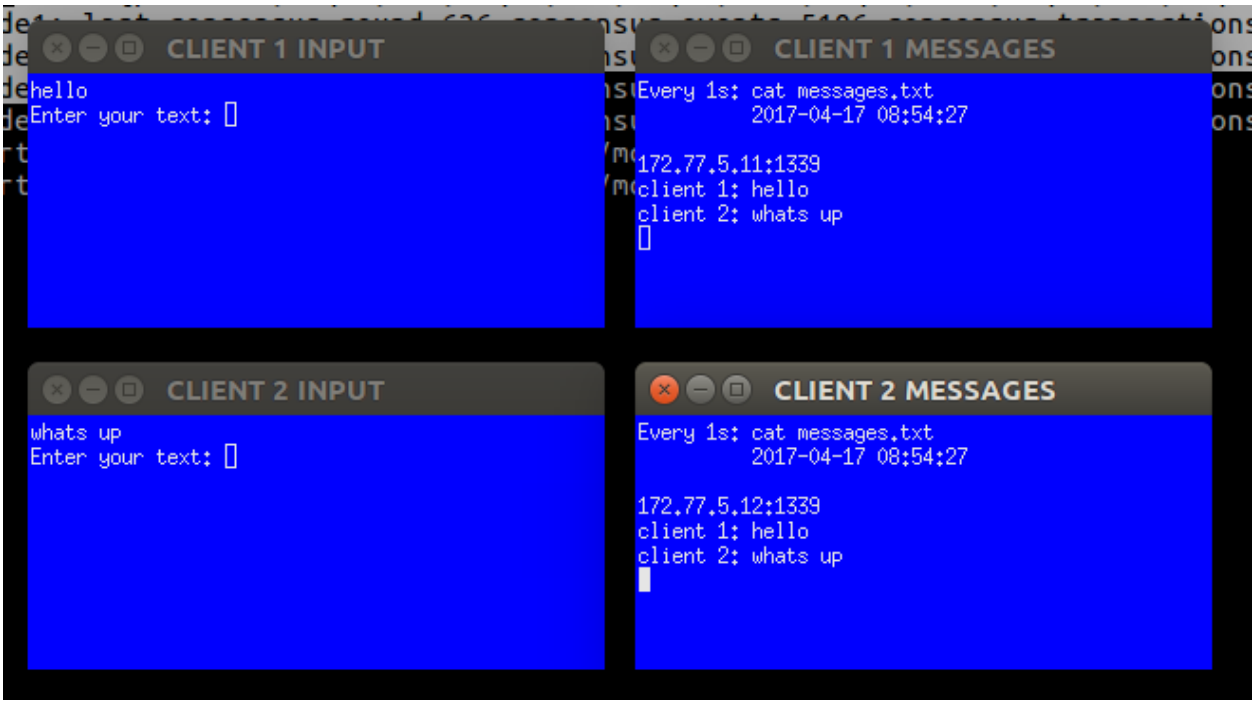
Running `docker ps -a` will show you that 9 docker containers have been launched:

```
[...]/babble/demo$ docker ps -a
CONTAINER ID        IMAGE                      COMMAND                  CREATED          ␣
↪     STATUS              PORTS                  NAMES
ba80ef275f22        mosaicnetworks/watcher     "/watch.sh"              48 seconds ago ␣
↪     Up 7 seconds                               watcher
4620ed62a67d        mosaicnetworks/dummy       "dummy '--name=client"   49 seconds ago ␣
↪     Up 48 seconds       1339/tcp               client4
847ea77bd7fc        mosaicnetworks/babble      "babble run --cache_s"   50 seconds ago ␣
↪     Up 49 seconds       80/tcp, 1337-1338/tcp  node4
11df03bf9690        mosaicnetworks/dummy       "dummy '--name=client"   51 seconds ago ␣
↪     Up 50 seconds       1339/tcp               client3
00af002747ca        mosaicnetworks/babble      "babble run --cache_s"   52 seconds ago ␣
↪     Up 50 seconds       80/tcp, 1337-1338/tcp  node3
b2011d3d65bb        mosaicnetworks/dummy       "dummy '--name=client"   53 seconds ago ␣
↪     Up 51 seconds       1339/tcp               client2
e953b50bc1db        mosaicnetworks/babble      "babble run --cache_s"   53 seconds ago ␣
↪     Up 52 seconds       80/tcp, 1337-1338/tcp  node2
0c9dd65de193        mosaicnetworks/dummy       "dummy '--name=client"   54 seconds ago ␣
↪     Up 53 seconds       1339/tcp               client1
d1f4e5008d4d        mosaicnetworks/babble      "babble run --cache_s"   55 seconds ago ␣
↪     Up 54 seconds       80/tcp, 1337-1338/tcp  node1
```

Indeed, each node is comprised of an App and a Babble node (cf Design section). The `watcher` container monitors consensus figures.

Run the `demo` script to play with the `Dummy App` which is a simple chat application powered by the Babble consensus platform:

```
[...]/babble/demo$ make demo
```



Finally, stop the testnet:

```
[...]/babble/demo$ make stop
```

1.5.2 Manual Setup

The above scripts hide a lot of the complications involved in setting up a Babble network. They generate the configuration files automatically, copy them to the right places and launch the nodes in Docker containers. We recommend looking at these scripts closely to understand how the demo works. Here, we will attempt to explain the individual steps that take place behind the scenes.

Configuration

Babble reads configuration from the directory specified by the `datadir` flag which defaults to `~/.babble` on linux/osx. This directory must contain two files:

- `peers.json` : Lists all the participants in the network.
- `priv_key.pem`: Contains the private key of the validator runnning the node.

Every participant has a cryptographic key-pair that is used to encrypt, sign and verify messages. The private key is secret but the public key is used by other nodes to verify messages signed with the private key. The encryption scheme used by Babble is ECDSA with the P256 curve.

To run a Babble network, it is necessary to predefine who the participants are going to be. Each participant will generate a key-pair and decide which network address it is going to be using for the Babble protocol. Someone, or some process, then needs to aggregate the public keys and network addresses of all participants into a single file - the peers.json file. Every participant uses a copy of the same peers.json file. Babble will read that file to identify the participants in the network, communicate with them and verify their cryptographic signatures.

To generate key-pairs in a format usable by Babble, we have created the `keygen` command. It is left to the user to derive a scheme to produce the configuration files but the docker demo scripts are a good place to start.

So let us say I want to participate in a Babble network. I am going to start by running `babble keygen` to create a key-pair:

```
babble keygen
Your private key has been saved to: /home/martin/.babble/priv_key.pem
Your public key has been saved to: /home/martin/.babble/key.pub
```

The private key looks something like this:

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIJ3orqofiSXu07mD+f46gZFK3EKSTqhXsbLVmA/aLmyqoAoGCCqGSM49
AwEHoUQDQgAEXgNNc8hJdWrntlFcpg2WpakRsTpNi0W8DgsC7bRQCd9szAdO6298
Z5V0D5k2ZO3ulw+KcXyJNE+EN/QSvfDRfA==
-----END EC PRIVATE KEY-----
```

and the corresponding public key looks like this:

```
0x045E034D73C849756AE7B6515CA60D96A5A911B13A4D8B45BC0E0B02EDB45009DF6CCC074EEB6F7C6795740F993664EDEE
```

DO NOT REUSE THESE KEYS

Next, I am going to copy the public key (key.pub) and communicate it to whoever is responsible for producing the peers.json file. At the same time, I will tell them that I am going to be listening on 172.77.5.2:1337.

Suppose three other people do the same thing. The resulting peers.json file could look something like this:

```
[
    {
            "NetAddr":"172.77.5.1:1337",
            "PubKeyHex":
↪"0x0471AEE3CAE4E8442D37C9F5481FB32C4531511988652DF923B79ED4ED992021183D31E0F6FBFE96D89B6D03D725029
↪"
    },
    {
            "NetAddr":"172.77.5.2:1337",
            "PubKeyHex":
↪"0x045E034D73C849756AE7B6515CA60D96A5A911B13A4D8B45BC0E0B02EDB45009DF6CCC074EEB6F7C6795740F993664E
↪"
    },
    {
            "NetAddr":"172.77.5.3:1337",
            "PubKeyHex":
↪"0x047CCCD40D90B331C64CE27911D3A31AF7DC16C1EA6D570FDC2120920663E0A678D7B5D0C19B6A77FEA829F8198F4F48
↪"
    },
    {
            "NetAddr":"172.77.5.4:1337",
            "PubKeyHex":
↪"0x0406CB5043E7337700E3B154993C872B1C61A84B1A739528C4A10135A3D64939C094B4A999BD21C3D5E9E9ECF15B2024
↪"
    }
]
```

Now everyone is going to take a copy of this peers.json file and put it in a folder together with the priv_key.pem file they generated in the previous step. That is the folder that they need to specify as the datadir when they run Babble.

1.5.3 Babble Executable

Let us take a look at the help provided by the Babble CLI:

```
Run node

Usage:
  babble run [flags]

Flags:
      --cache-size int           Number of items in LRU caches (default 500)
  -c, --client-connect string    IP:Port to connect to client (default "127.0.0.1:1339
↪")
      --datadir string           Top-level directory for configuration and data␣
↪(default "/home/martin/.babble")
      --heartbeat duration       Time between gossips (default 1s)
  -h, --help                     help for run
  -l, --listen string            Listen IP:Port for babble node (default ":1337")
      --log string               debug, info, warn, error, fatal, panic
      --max-pool int             Connection pool size max (default 2)
  -p, --proxy-listen string      Listen IP:Port for babble proxy (default "127.0.0.
↪1:1338")
  -s, --service-listen string    Listen IP:Port for HTTP service
      --standalone               Do not create a proxy
      --store                    Use badgerDB instead of in-mem DB
      --sync-limit int           Max number of events for sync (default 100)
  -t, --timeout duration         TCP Timeout (default 1s)
```

So we have just seen what the `datadir` flag does. The `listen` flag corresponds to the NetAddr in the peers.json file; that is the endpoint that Babble uses to communicate with other Babble nodes.

As we explained in the architecture section, each Babble node works in conjunction with an application for which it orders transactions. When Babble and the application are connected by a TCP interface, we specify two other endpoints:

- `proxy-listen` : where Babble listens for transactions from the App
- `client-connect` : where the App listens for transactions from Babble

We can also specify where Babble exposes its HTTP API providing information on the Hashgraph and Blockchain data store. This is controlled by the optional `service-listen` flag.

Finally, we can choose to run Babble with a database backend or only with an in-memory cache. With the `store` flag set, Babble will look for a database file in `datadir`/babdger_db. If the file exists, the node will load the database and bootstrap itself to a state consistent with the database and it will be able to proceed with the consensus algorithm from there. If the file does not exist yet, it will be created and the node will start from a clean state.

Here is how the Docker demo starts Babble nodes together wth the Dummy application:

```
for i in $(seq 1 $N)
do
    docker run -d --name=client$i --net=babblenet --ip=172.77.5.$(($N+$i)) -it␣
↪mosaicnetworks/dummy:0.4.0 \
    --name="client $i" \
    --client-listen="172.77.5.$(($N+$i)):1339" \
    --proxy-connect="172.77.5.$i:1338" \
    --discard \
    --log="debug"
done
```

(continues on next page)

(continued from previous page)

```
for i in $(seq 1 $N)
do
    docker create --name=node$i --net=babblenet --ip=172.77.5.$i mosaicnetworks/
↪babble:0.4.0 run \
    --cache-size=50000 \
    --timeout=200ms \
    --heartbeat=10ms \
    --listen="172.77.5.$i:1337" \
    --proxy-listen="172.77.5.$i:1338" \
    --client-connect="172.77.5.$(($N+$i)):1339" \
    --service-listen="172.77.5.$i:80" \
    --sync-limit=1000 \
    --store \
    --log="debug"

    docker cp $MPWD/conf/node$i node$i:/.babble
    docker start node$i
done
```

1.5.4 Stats, blocks and Logs

Once a node is up and running, we can call the `stats` endpoint exposed by the HTTP service:

```
curl -s http://172.77.5.1:80/stats
```

or request to see a specific block:

```
curl -s http://172.77.5.1:80/block/1
```

Or we can look at the logs produced by Babble:

```
docker logs node1
```

1.6 Babble Consensus

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.

1.6.1 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?

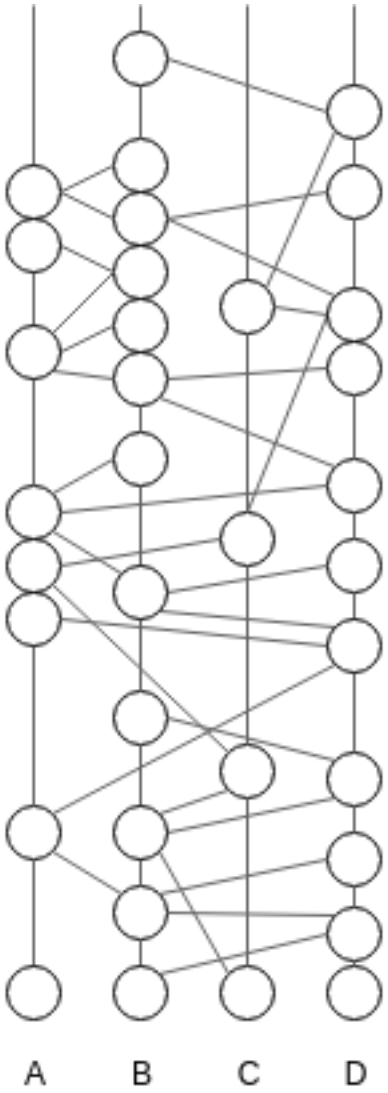
One general, having made the decision to attack, could send a messenger to the other general. Upon receiving that message, the second general knows that the first general wants to attack, but he doesn't know that the first general knows that he received the message. So he sends an acknowledgment. Upon 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.

1.6.2 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.



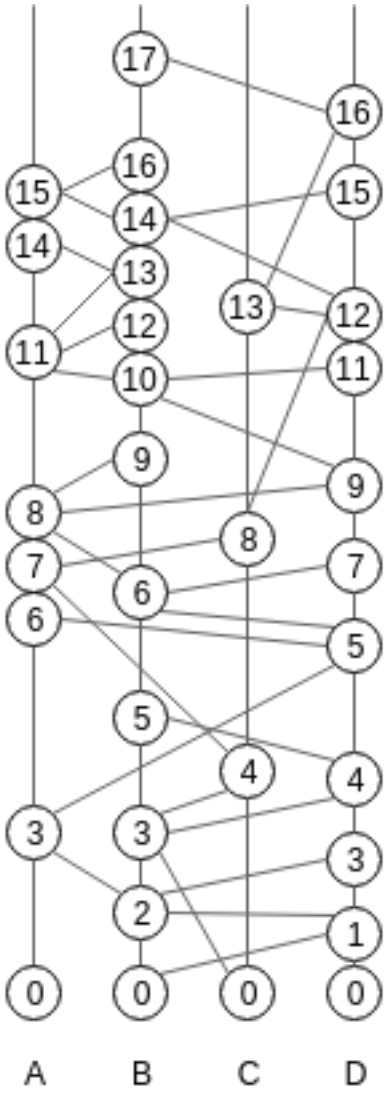
The communication graph is a very rich data structure from which we can extract all sorts of information about the history of gossip, and also derive a consistent ordering of the events, even in the presence of faulty participants. But let's take it step by step.

1.6.3 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.

1.6.4 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.

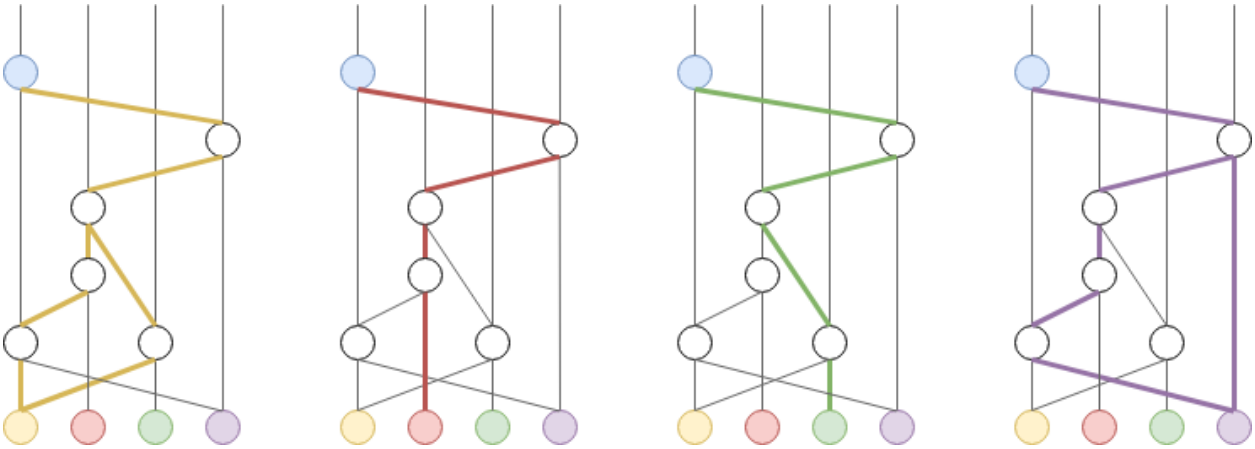
Usually, the solutions vary around who gets to propose the value - aka the leader - and how this leader is elected or changed.

1.6.5 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.

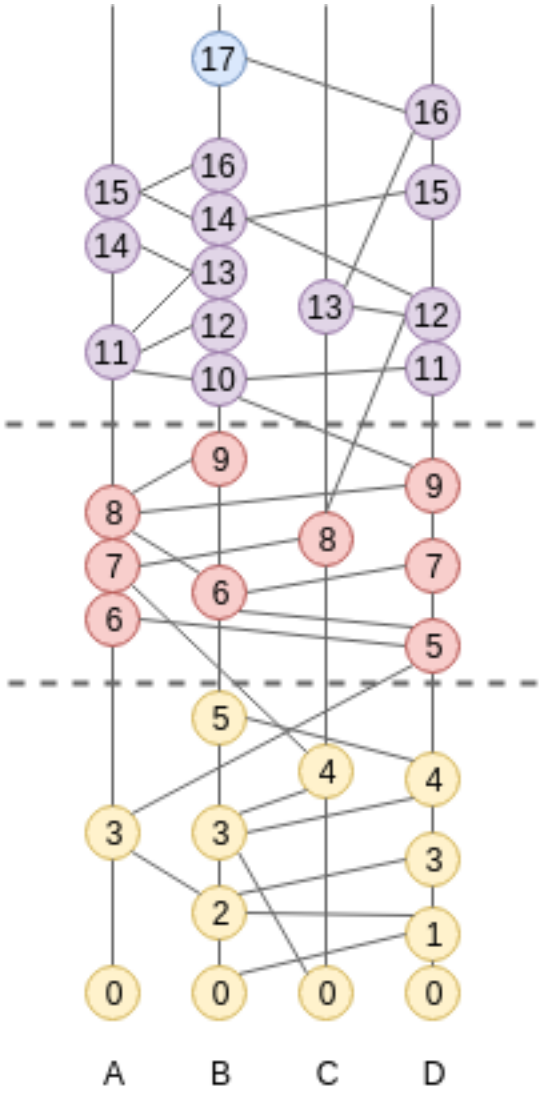
First, the Hashgraph defines a concept of *Strongly Seeing*:

"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.

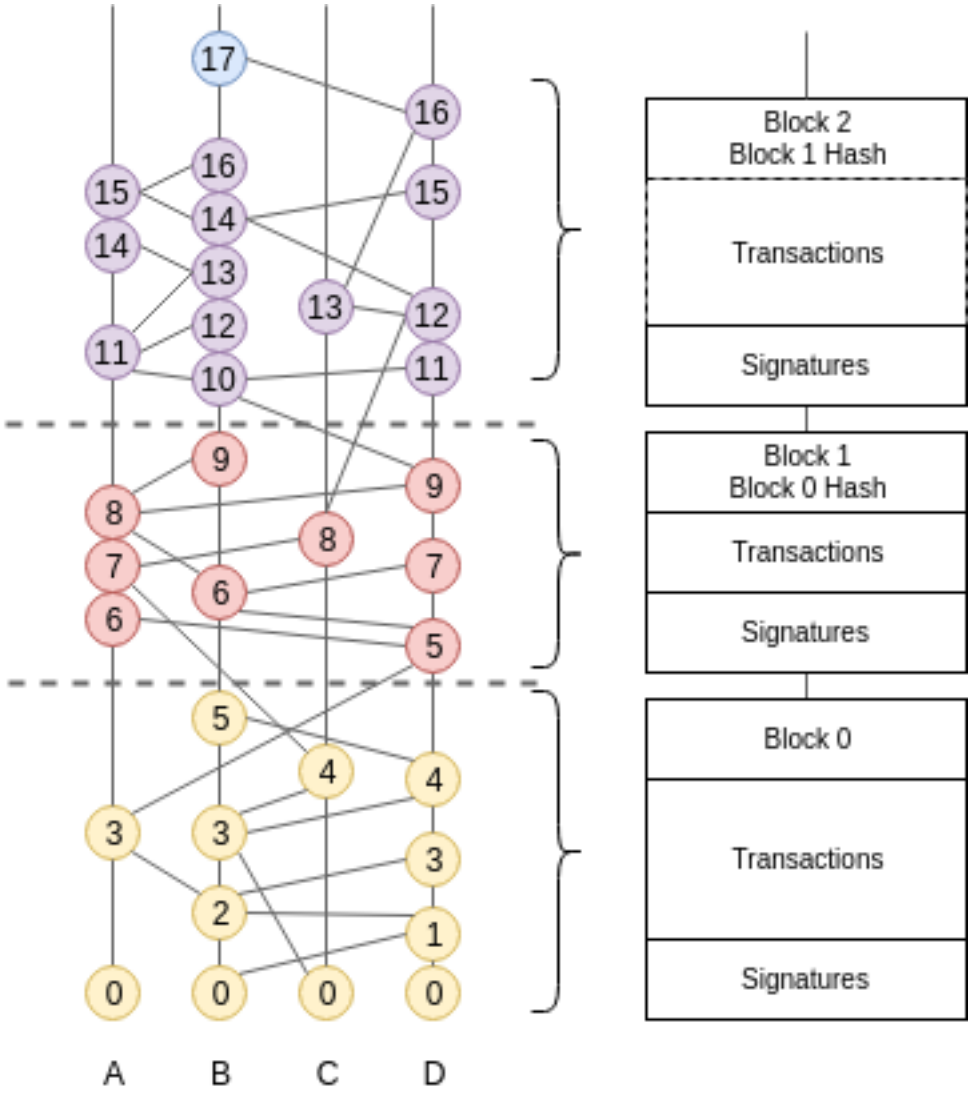
Also, we do not need a leader to propose a value. Instead, participants compute virtual cuts in the hashgraph, called rounds, which allow processing events in batches. This is also a distributed algorithm where all members end up with the same rounds. Roughly speaking, starting at round 0, when we reach 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.

1.6.6 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.

1.7 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 previous block, a hash of the resulting application state, 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.

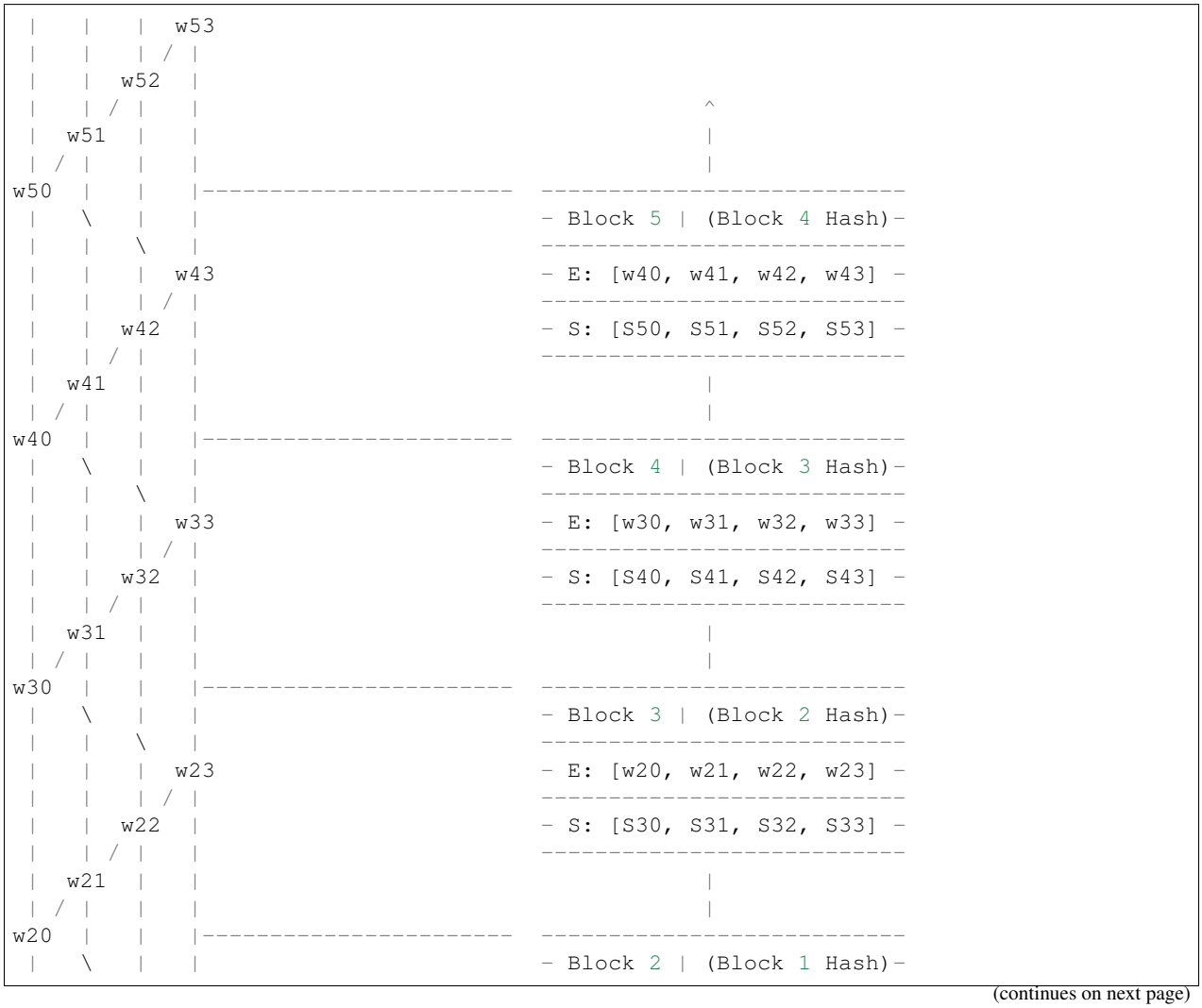
1.7.1 Motivation

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, easily allowing to verify 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.

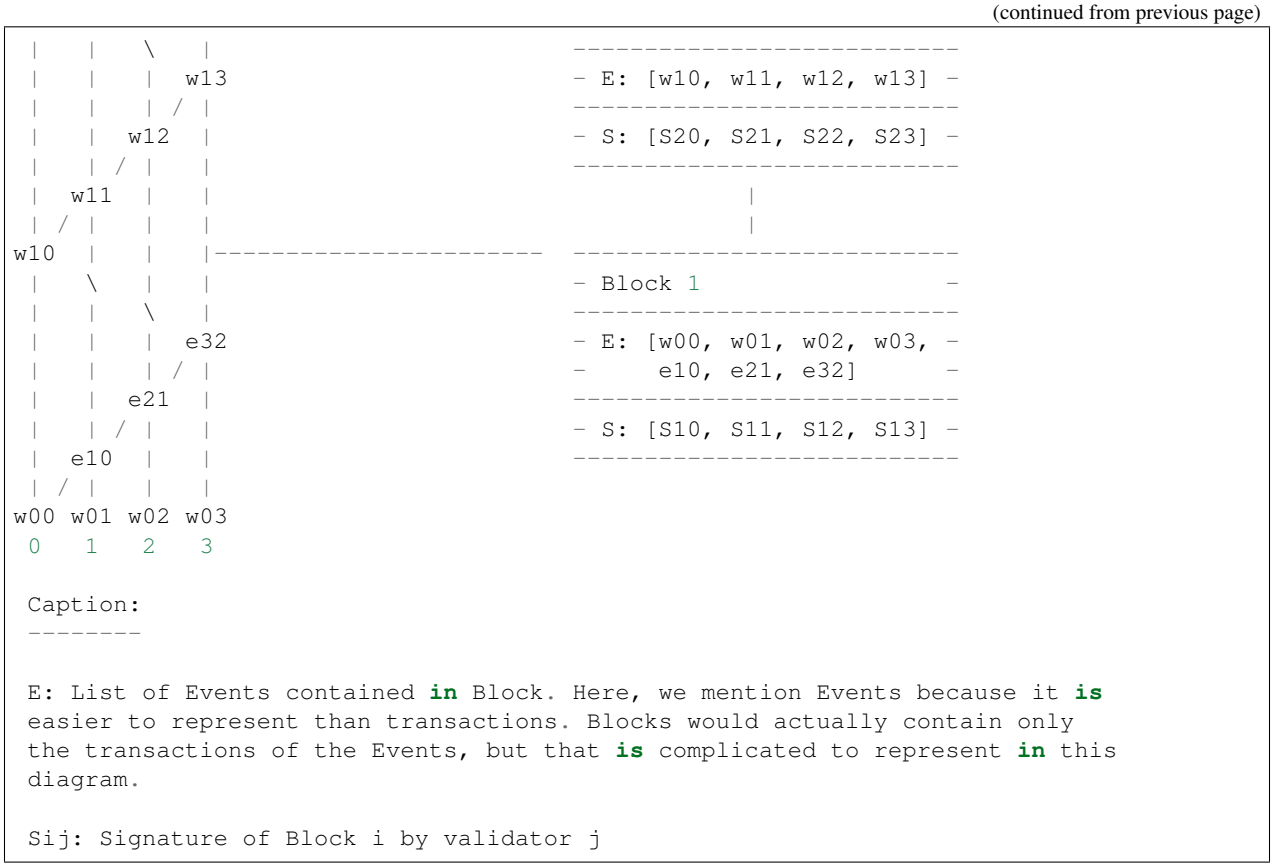
The "hashgraph vs blockchain" debate is a red herring. Blockchain is just a data structure; the engine is the underlying consensus algorithm. The projection method exposes an easy-to-work-with blockchain powered by the efficient Hashgraph consensus algorithm.

1.7.2 Implementation



(continues on next page)

(continued from previous page)



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.

1.7.3 Block Structure

```
Block: {
    Header:{
        Index:          int,
        RoundReceived: int,
        PrevBlockHash: []byte,
        BodyHash:       []byte,
        StateHash:      []byte,
    }
    Body:{
        Transactions: [][]byte
    }
    Signatures: map[string][]byte
}
```

Blocks contain a Header and a Body. Signatures are based on the Header only; which is enough to verify the entire block because it contains a digital fingerprint of the Body. Since Headers also contain a hash of the previous block, each block signature adds further validation to previous blocks. The Header's *RoundReceived* corresponds to the *RoundReceived* of the hashgraph Events who's transactions are included in the block; it serves the purpose 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 block Body also contains a hash of the application's state resulting from applying the block's transactions sequentially. Counting signatures from one third of validators provides a proof that all honest nodes have not only applied the same transactions in the same order, but also computed the same state.

1.7.4 Enhancements

Dynamic Validator Set

The system described above assumes that the set of validators is fixed; block signatures are always checked against the same list of public keys. In Hashgraph, it is possible to make the set of validators change dynamically. The projection would have to be extended such that block Headers would also contain a Merkle root of the current validator set, thereby providing a simple method of verifying that a signer belongs to the set of validators corresponding to the block it signed.

Inter-Blockchain Communication

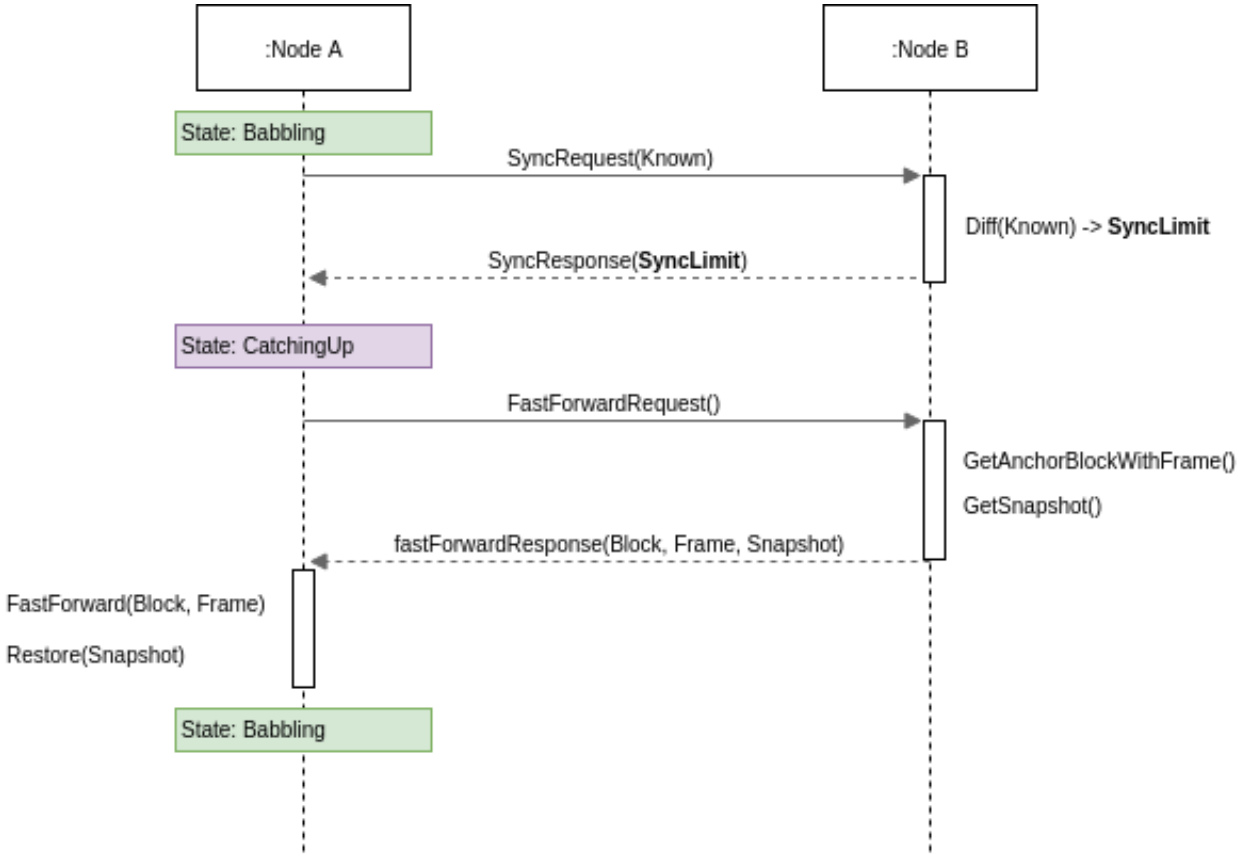
Inter-Blockchain Communication (IBC) is about verifying on one chain that a transaction happened on another chain; one blockchain acts as a light-client to another blockchain. It is much simpler to build a light-client for a blockchain than for a hashgraph. In an effort to enable interoperability between 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.

1.8 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.

1.8.1 Overview



The Babble node is implemented as a state machine where the possible states are: **Babbling**, **CatchingUp**, and **Shutdown**. A node is normally in the **Babbling** state where it performs the regular Hashgraph gossip routines, but a **sync_limit** response from a peer will trigger the node to enter the **CatchingUp** state, where it will attempt to fast-forward to a recent snapshot. A **sync_limit** response indicates that the number of Events that the node needs to download exceeds the **sync_limit** configuration value.

In the **CatchingUp** state, a node repeatedly chooses another node at random (although the above diagram uses the same peer that returned the **sync_limit** response) 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*.

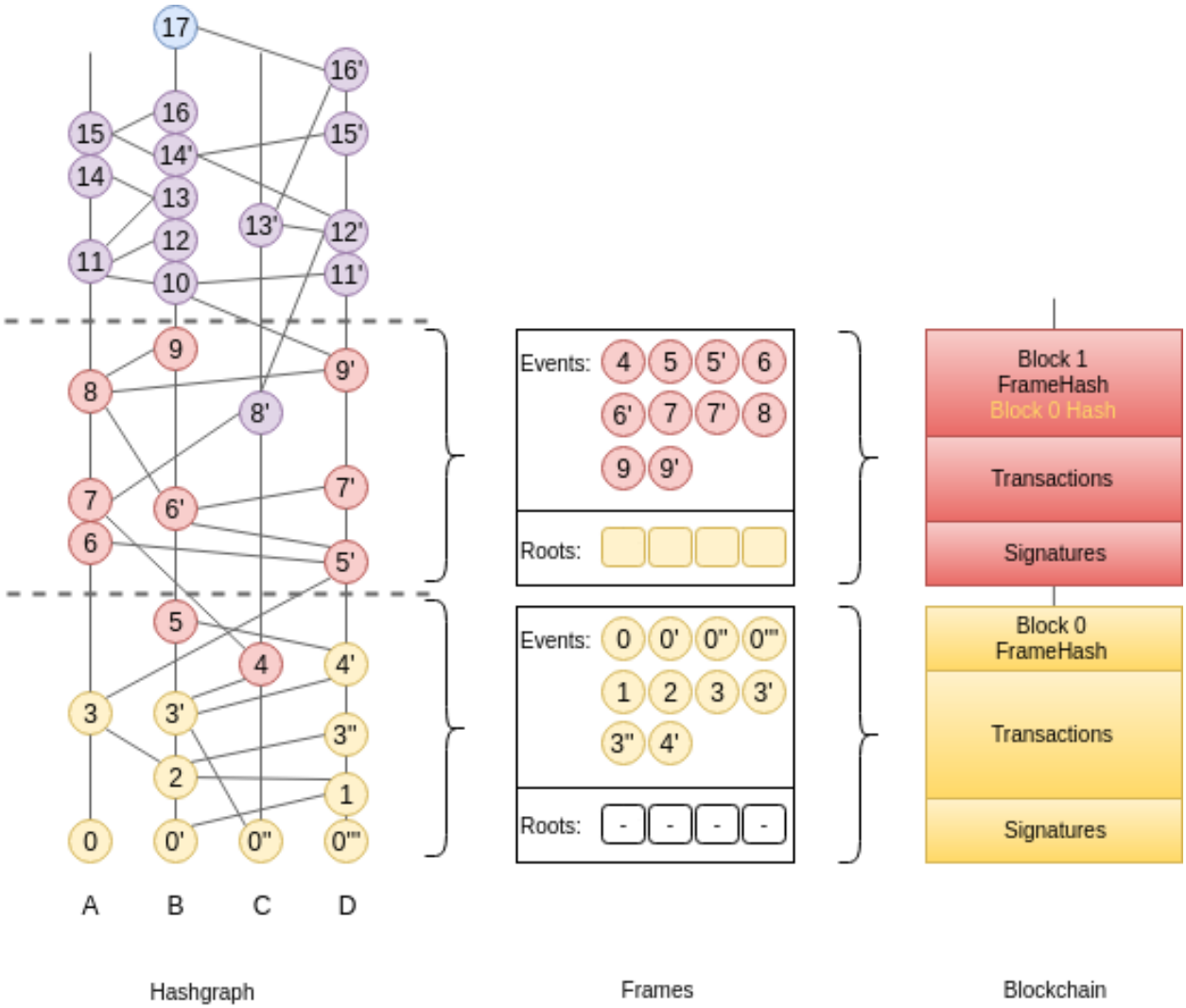
Upon receiving a FastForwardRequest, a node must respond with the last consensus snapshot, as well as the corresponding Hashgraph section (the Frame) and Block. With this information, and having verified the Block signatures against the other items as well as the known validator set, the requesting node 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.

1.8.2 Frames

Frames are self-contained sections of the Hashgraph. They are composed of Roots and regular Hashgraph Events, where Roots are the base on top of which Events can be inserted. 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
      Roots  []Root   //[participant ID] => Root
      Events []Event  //Events with RoundReceived = Round
}
```

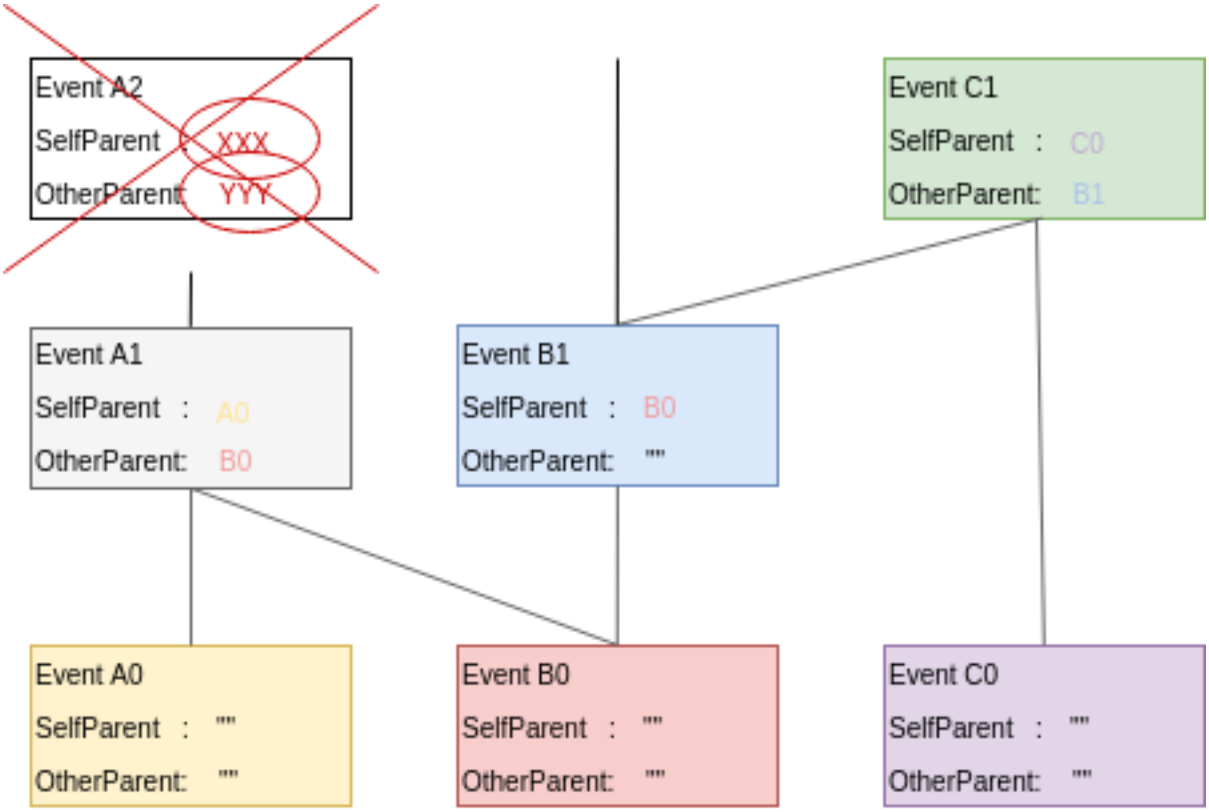
A Frame corresponds to a Hashgraph consensus round. Indeed, the consensus algorithm commits Events in batches, which we map onto Frames, and finally onto a Blockchain. This is an evolution of the previously defined *blockchain mapping*. Block headers now contain a Frame hash. As we will see later, this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



1.8.3 Roots

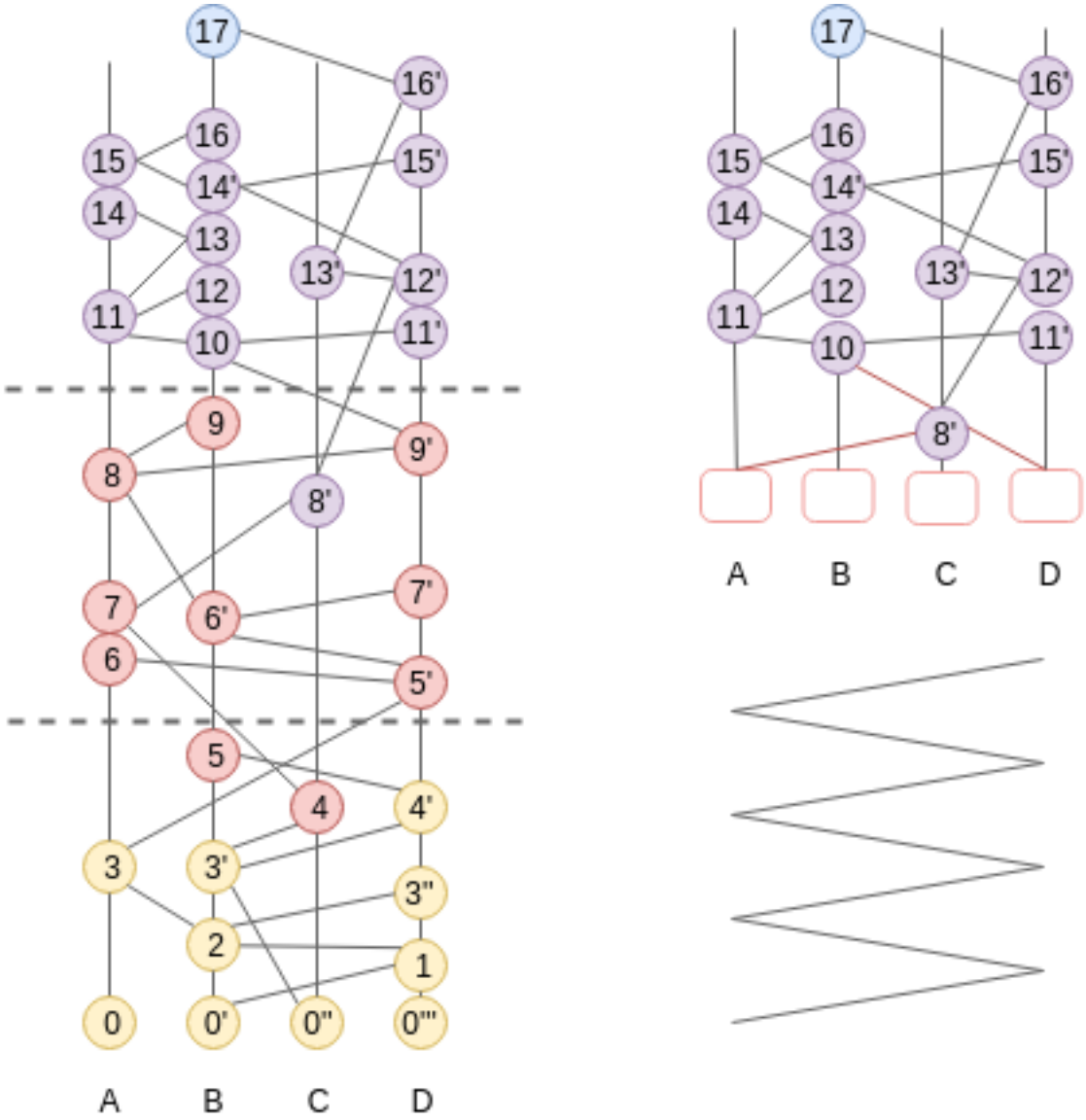
Frames also contain Roots. To get an understanding for why this is necessary, we must consider the initial state of a Hashgraph, i.e., the base on top of which the first Events are inserted.

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.



So what about the first Event? Until now, we simply implemented a special case, whereby the first Event for any participant, could be inserted without checking its references. In fact the above picture shows that Events A0, B0, and C0, have empty references, and yet they are part of the Hashgraph. This special case is fine as long as we do not expect to initialize Hashgraphs from a 'non-zero' state.

We introduced the concept of Roots to remove the special case and handle more general situations. They make it possible to initialize a Hashgraph from a section of an existing Hashgraph, and discard everything below it.



A Root is a data structure containing condensed information about the ancestors of the first Events to be added to the Hashgraph. Each participant has a Root, containing a *SelfParent* - the direct ancestor of the first Event for the corresponding participant - and *Others* - a map of Event hashes to the corresponding Other-Parents. These parents are instances of the **RootEvent** object, which is a minimal version of the Hashgraph Event. RootEvents contain information about the Index, Round, and LamportTimestamp of the corresponding Events. The Root itself contains a NextRound field, which helps in calculating the Round of its direct descendant.

```
type Root struct {
  NextRound  int
  SelfParent RootEvent
  Others     map[string]RootEvent
}

type RootEvent struct {
  Hash             string
  CreatorID        int
  Index            int
```

(continues on next page)

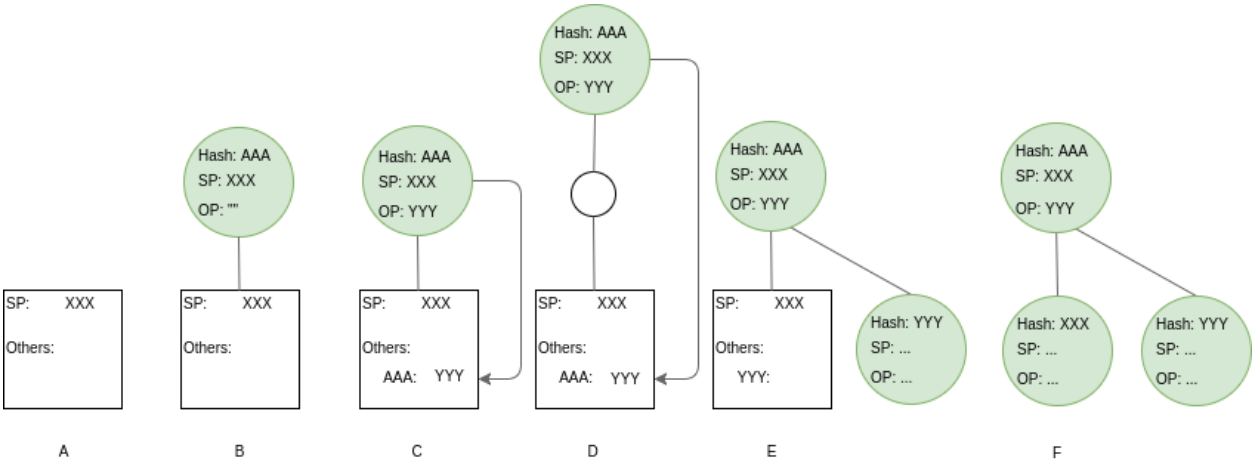
(continued from previous page)

```
  LamportTimestamp int
  Round            int
}
```

1.8.4 Algorithm Updates

The new rule for inserting an Event in the Hashgraph prescribes that an Event should only be inserted if its parents belong to the Hashgraph or are referenced in one of the Roots. The algorithms for computing an Event's Round and LamportTimestamp have also changed slightly.

There are six different scenarios to consider when computing an Event's Round; each corresponding to a different relationship between the Event and its creator's Root.



Scenario	Description	Round	LamportTimestamp
A	The Event is a Root itself	Root.SelfParent.Round	Root.SelfParent.LamportTimestamp
B	The Event is directly attached to the Root, and its OtherParent is empty	Root.NextRound	Root.SelfParent.LamportTimestamp + 1
C	The Event is directly attached to the Root, and its OtherParent is referenced in Root.Others	Root.NextRound	Max(Root.SelfParent.LamportTimestamp, Root.Others[AAA].LamportTimestamp) +1
D	The Event is not directly attached to the Root, but its OtherParent is referenced in Root.Others	Max(Event.SelfParent.Round, Root.Others[AAA].Round) + RoundInc()	Max(Event.SelfParent.LamportTimestamp, Root.Others[AAA].LamportTimestamp) +1
E	The Event is directly attached to the Root, and its OtherParent is a normal Event	Max(Root.SelfParent.Round, Event.OtherParent.Round) + RoundInc()	Max(Root.SelfParent.LamportTimestamp, Event.OtherParent.LamportTimestamp) +1
F	Both parents are regular Events (or OtherParent is empty)	Max(Event.SelfParent.Round, Event.OtherParent.Round) + RoundInc()	Max(Event.SelfParent.LamportTimestamp, Event.OtherParent.LamportTimestamp) +1

Here RoundInc() is the function that computes whether and Event's Round should be incremented over its ParentRound. It checks if the Event can StronglySee a super-majority of witnesses from ParentRound, as described in the original whitepaper.

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. Frames could be made to include more information so as to avoid this type of problem with greater probability, but such and approach could eventually undermine the usefulness of Frames as light-weight data points. As we shall see later, a potential solution to such and edge-case would be to adopt a "let it crash" philosophy and rely on an other level to handle the burden.

1.8.5 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.

1.8.6 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) ([]byte, 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.

1.8.7 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.