Formatted: Right: 0.81"

CSE535: Distributed Systems
Project: ~~Libra ~~DiemBFT v4 Consensus Algorithm
Scott D. Stoller
phase 2: D~~Libra ~~iemBFT v4 consensus algorithm
version: 2021-10-16~~09-05~~
due date: 17~~1~~ Oct 2021

Overview

Implement the ~~Libra (a.k.a. d~~Diem~~)~~BFT v4 consensus algorithm, as described in the paper ~~State Machine Replication in the Libra Blockchain~~DiemBFT v4 - State Machine Replication in the Diem Blockchain. For additional references, see phase 1 assignment.

Field Code Changed

---

1. Pseudo-code

The Libra consensus paper says, in the "Omitted, gory details" section: "In particular, when a message is handled in the pseudo code, we assume that its format and signature has been validated, and that the receivers has synced up with all ancestors and any other data referenced by meta-information in the message." These tasks are also mentioned briefly in Section 4.1: "For brevity, the description henceforth assumes messages have already been filtered for format and validity of cryptographic values, and that any information indirectly referenced by messages (e.g., a QC for a block) is filled by a lower-level communication substrate." Provide pseudo-code for these aspects of the algorithm, by extending the pseudo-code in the paper. [Clarification regarding checks for format and validity of cryptographic values, it's sufficient to show the necessary calls to functions that verify signatures and compute cryptographic hashes; you don't need to show format checks, e.g., that a message or object contains the expected fields or that a value is a bit string of the expected length.] You can modify existing functions, modules, etc., and introduce additional functions. Use similar notation and level of detail as the existing pseudocode.

Another aspect glossed over in the pseudocode in the paper is exactly which client requests to include in each proposal, and how duplicate (retransmitted) requests are handled. Extend the pseudocode to include these aspects. Explain your design in comments embedded in or accompanying the pseudocode.

Extend the pseudocode so that a client can determine when a command it submitted has been committed to the ledger.

Your submission needs to include only modified or added parts of the pseudocode; you do not need to copy unmodified parts of the pseudocode.

---

2. Code

Implement the ~~Libra ~~diemBFT v4 consensus algorithm, including clients that can drive test cases specified in the configuration. your code should correspond closely to the pseudocode in the paper (with your extensions), by using the same or similar names for classes, functions, variables, etc. To further demonstrate this correspondence, each fragment of code should be preceded with a comment containing a copy of the corresponding line of pseudocode, if any.

2.1 Logs. Each process should generate a comprehensive log file describing initial settings (read from configuration), any generated seeds for pseudorandom number generators, the content of every message received, the content of every message sent, and details of every significant internal action (timeouts, committing blocks in the ledger, etc.). every log entry should identify the process that performed the action, preferably using an intuitive name such as “client1” or “validator2”, instead of or in addition to DistAlgo’s built-in process identifiers. every log entry should contain a real-time timestamp. the log file should have a self-describing format, in the sense that each value is labeled with a name indicating its meaning. for example, each component of a message should be labeled to indicate its meaning; do not rely on the reader to know that the first component of the tuple means X, the second component means Y, etc. Log files should have meaningful names, e.g., the same name as the corresponding configuration file or test case.

~~For systems implemented in DistAlgo, the log file should be produced using DistAlgo's built-in support for logging. See the "Logging output" section of the DistAlgo language description (language.pdf) and the logging-related command-line options (search for "log" in the output of "python -m da --help").~~

2.2 Running the system. In simple DistAlgo programs, the top-level main() creates and starts the other processes. Instead, you should add a level of indirection: put that functionality in the run method of a process class (e.g., a Run class), and have the top-level main() create and start instance(s) of that class. This increases flexibility. For example, the top-level main() could easily iterate through a list of scenarios or test cases (each with a different configuration) and create and start an instance of Run to run each of them. To support this, configuration information should be passed to Run.setup. This approach is illustrated in ping.da (based on da/examples/ping.da) and testping.da, attached on Blackboard

2.3 Configuration. Configuration information for a scenario includes number of replicas, number of faulty replicas, number of clients, seeds for pseudorandom number generators (so results are reproducible), timeout-related values (e.g., the value of Δ, if you use $4\times\Delta$ as the round timer formula in Pacemaker), specification of the workload that clients should generate (e.g., send a specified number of requests with specified delays between them), etc. If seeds are omitted from a particular configuration, the system should pseudorandomly generate seeds and write them to a log file.

Configuration should be expressed in DistAlgo (not JSON, etc.) and passed to Run.setup. This is illustrated in testping.da.

2.4 Process Termination. To run test cases one at a time, the test driver needs to know when a test case has finished. For example, clients can tell their parent Run process when they are finished with their workload and about to exit, and the Run process can then tell replicas to exit. Similar logic is used in da/examples/lapaxos.da, where learners tell the main process when they are finished, and the main process tells acceptors and proposers to exit.

2.5 Fault Injection. Your code should be able to simulate message failures~~loss~~ and some simple kinds of misbehavior by faulty replicas, with the configuration specifying details of the desired failure scenario. ~~Details of the required kinds of failures and failure scenarios will be added here soon.~~ Fault-injection code should be encapsulated in a separate class~~es and methods~~ as much as possible, to avoid cluttering the code for the algorithm itself. Accomplish this using the approach in the runtime checking paper (e.g., see Fig. 2 there), by implementing fault injection in a subclass of Replica---named, say, ReplicaFI---that overrides the send method (and potentially other methods) with a variant that incorporates fault injection. The Replica class should define a method setattr(attr,val) that ReplicaFI can call to modify the state of the replica, to simulate a simple kind of Byzantine failure. This is illustrated in fault_injection.da.

A failure configuration has type

```
from collections import namedtuple
FailureConfig = namedtuple('FailureConfig', ['failures', 'seed'], defaults=(None))
```

where seed is the seed for the pseudorandom number generator used to determine outcomes of probabilistic message losses (if seed==None, a seed is generated and written to a log), and failures is a list of namedtuples with type Failure, where

Formatted: Normal, No bullets or numbering

```
Failure = namedtuple('Failure', ['src', 'dest', 'msg_type', 'round', 'prob', 'fail_type', 'val', 'attr'],
defaults=(None,None))
```

Here, src and dest indicate the sender and destination of a message, msg_type has type

Formatted: Normal

```
from enum import Enum
class MsgType(Enum):
    Proposal = 1
    TimeOut = 3
    Vote = 4
    Wildcard = 5   # matches all message types
```

round is a natural number or '_' (a wildcard that matches all values, as in DistAlgo) [note: round='_' was added late, so you don't need to implement it; I added it as a reminder for the future.], prob is a probability (a number from 0 to 1), fail_type has type

Formatted: Normal

```
class FailType(Enum):
    MsgLoss = 1
    Delay = 2
    SetAttr = 3
```

and val is a number. src and dest may be replica numbers (we assume replicas are numbered from 1 to n), 'leader' (indicating the leader of the specified round, which the fault injection code can determine by calling the protocol's get_leader function), or '_' (wildcard); for simplicity, src and dest cannot be 'leader' when round='_'. A failure is potentially triggered when src sends a message of the specified type for the specified round to dest. Each time that occurs, a failure of the specified type occurs with probability prob. FailType.MsgLoss means the message being sent is lost, i.e., not actually sent. FailType.Delay means the process delays (e.g., sleeps by calling time.sleep) for val seconds before sending the message. FailType.SetAttr means that the fault injection code calls setattr(attr,val) to set the replica's attribute attr to value val. setattr should support setting at least the following attributes used in the pseudocode in the paper: highest_vote_round (defined in the Safety module) and current_round (defined in Pacemaker).

These attributes do not need to be attributes of Replica; Replica.setattr can update them wherever they are. An example failure configuration is:

```
failure_config = FailureConfig(failures = [
Failure(src='_',dest='leader',msg_type=MsgType.Vote,round=1,prob=1,fail_type=FailType.Delay, val=0.1),
Failure(src=2,dest='_',msg_type=MsgType.Wildcard,round=3,prob=0.5,fail_type=FailType.MsgLoss),
Failure(src='leader',dest='_',msg_type=MsgType.Vote,round=3,prob=0.5,fail_type=FailType.SetAttr,val=2,at
tr='highest_vote_round')],
seed = 12345678)
```

Formatted: Space After: 0 pt

Formatted: Normal

---

3. User manual

Write a user manual with ~~I~~instructions to build and run your system. the instructions should not rely on an IDE. provide a detailed sequence of commands, a shell script, or something similar. include a specific example of the command(s) to run a selected test case. The user manual should also document the configuration file format (e.g., plain text or JSON) and structure: the name—and meaning, if it's not self-evident—of each field and the valid values for each field (if it's not self-evident).

---

4. Test Report

Describe your main test cases~~ ("test case" here means an execution of Twins, not an individual execution of Libra consensus)~~. For each test case, report the names of the configuration file, ledger files, and log files, number of replicas, number of clients, number of client requests, failure scenario, any other important configuration information, ~~number of rounds executed, ~~and the outcome. In this assignment, it's sufficient to run all processes on a single host.

Test Case Guidelines

Formatted: Font: Bold

some (probably most) test cases should involve multiple clients sending requests concurrently. it's fine to design each client to have at most one pending request at a time, as in PBFT and Raft.

the test cases should involve all of the kinds of failures described in section 2.5 (Fault Injection), in non-trivial patterns and combinations. note that any number of validators may suffer from message loss and delays, while at most f validators may suffer from SetAttribute failures.

some test cases should involve "chains" of failures; loosely, the idea is that a failure occurs, and then another failure occurs before "normal operation" (a round that successfully commits a block) resumes. for example, there should be test cases in which the leaders of multiple consecutive rounds are affected by failures, causing the rounds to end with timeout certificates (TC) instead of quorum certificates (QC).

it's fine for most test cases to use f=1, N=4 (where N = number of replicas), but there should also be some test cases with f=2, N=7.

4.5. Grading Sheet

Formatted: Font: Bold

Formatted: Font: Bold

Fill out the "claimed" column of the phase 2 grading sheet to reflect the status of your implementation; you don't need to fill out claimed points for pseudo-code and documentation. Claim partial credit for items that are partially implemented or have bugs or limitations. For each item for which partial credit is claimed, include a brief explanation in the "notes re: claimed" column of the grading sheet or the "Bugs and limitations" section of the README. To incentivize accuracy of this information, there is a penalty for problems not reflected in the claimed points; this is shown in the Penalties section of the grading sheet.

Include the filled-out grading sheet in your submission in xlsx form, so the TAs can fill the "actual" column.

---

5. README

Write a README with the following sections:

Platform. describe the software platform(s) used in your testing, including DistAlgo version, Python implementation (normally CPython) and version, operating system name and version, and type of host (e.g., laptop, VM on VMWare Workstation Player on laptop, VM on Google Cloud Compute Engine, VM on Amazon Web Services EC2).

Workload generation. describe your design for client workload generation, and mention which file(s) contain the implementation.

Timeouts. discuss your choice of timeout formulas and timeout values for clients and servers (e.g., in function get_round_timer).

Bugs and Limitations. a list of all known bugs in and limitations of your code.

Main files. full pathnames of the files containing the main code for clients and replicas. this will help graders look at the most important code first.

Code size. (1) report the numbers of non-blank non-comment lines of code (LOC) in your system in the following categories: algorithm, other, and total. "algorithm" is for the replica algorithm itself and other functionality interleaved with it (logging, instrumentation, etc.). "other" is for everything that can easily be separated from the algorithm (clients, configuration, test drivers, etc.). (2) report how you obtained the counts (I use CLOC https://github.com/AlDanial/cloc). (3) give a rough estimate of how much of the "algorithm" code is for the algorithm itself, and how much is for other functionality interleaved with it.

Language feature usage (for teams using Python or DistAlgo). report the numbers of list comprehensions, dictionary comprehensions, set comprehensions, aggregations, quantifications, await statements, and receive handlers in your code. the first two are Python features; the others are DistAlgo features.

Formatted: Font: Not Bold

Contributions. a list of each team member's contributions to this submission.

Other comments (optional). anything else you want us to know.

---

6. Submission Instructions

Exactly one team member should submit the assignment as an archive file (zip or tar.gz, not rar) on Blackboard by 11:59pm on the due date. To help the TAs easily find things, the archive file should contain:

- top-level files with the following names (and appropriate extensions, e.g., docx, pdf, or txt): pseudocode, user manual, test report, phase 2 grading sheet.xlsx, README
- top-level folders with the following names: src (for source code), config (for configuration files used in test cases), log (for log files from test cases), ledgers (for ledger files from test cases)

---

7. Grading

Grading is based on (1) clarity, completeness, and accuracy of the pseudocode and all documentation, and (2) functionality and quality of the code, and (3) thoroughness of testing (for the kinds of failures considered in this assignment, as described in item 2.5 above).

Code quality includes:

- Tightness and clarity of correspondence with the pseudocode, and use of comments containing pseudocode to show the correspondence (see section 2).
- Use of appropriate data structures and algorithms
- Modularity. The code should be structured in a reasonable way into classes and methods.
- Meaningful names for classes, methods, variables, etc.
- Appropriate comments in the code. Include useful comments. Don't include "obvious" comments just to have more comments.
- Appropriate use of language features such as inheritance, methods, generic types, enumeration types, for-each loops, and assertions. For example, methods, inheritance, and loops should be used to avoid repeating blocks of identical, or nearly identical, code.
- Consistent code style. You should follow PEP 8 -- Style Guide for Python Code. Check adherence using Pylint or pycodestyle.

---

8. Comments on Some Design Decisions

if your team is considering a choice different than my recommendation for one of these design decisions, it might be OK, but please discuss with me before proceeding.

8.1 Cryptography

I recommend PyNaCl, a Python interface to the libsodium cryptography library, for cryptographic hashes and signatures. it is relatively well-documented and efficient. See https://pypi.python.org/pypi/PyNaCl/ and https://pynacl.readthedocs.io/en/latest/. I recommend SHA256 for cryptographic hashes and Ed25519 for digital signatures. The open-source implementation of ~~Libra consensus~~DiemBFT v4 also uses Ed25519.

Note that DiemBFT v4 ~~Libra consensus algorithm~~ uses ordinary digital signatures, not the threshold digital signatures used in HotStuff.

8.2 Client commands

I recommend that commands sent by clients be strings which are "executed" simply by storing them in ~~a block in~~ the ledger. To facilitate testing and debugging, I suggest that the i'th command issued by client c be str(c)+"-"+str(i) or something similar.

8.3 Client requests

The diemBFT paper does not discuss the pattern of communication between clients and servers. the HotStuff and PBFT papers both state that clients multicast requests to all replicas. you should follow this approach. your extended pseudocode should reflect how the algorithm ensures that a client request is not included in multiple committed blocks; section 1 already mentions this issue in the context in retransmitted requests.

Client requests should be signed, and servers should verify the signature. You can assume the set of clients (like the set of servers) is fixed and known in advance, so all necessary cryptographic keys can be created during initialization and disseminated to all processes during setup.

the papers do not discuss how client requests are identified or how servers detect duplicate requests. The HotStuff arXiv paper mentions this issue briefly in section 4.0: "A client sends a command request to all replicas, and waits for responses from (f +1) of them. For the most part, we omit the client from the discussion, and defer to the standard literature for issues regarding numbering and de-duplication of client requests." Note that de-duplication of client requests is necessary because clients may retransmit requests to which they do not receive a timely response.

Identification and de-duplication of client requests is a generic issue in distributed client-server systems. There is a spectrum of approaches, which vary in cost and strength of guarantee.

a strong approach is to require that each client includes a sequence number in each request, and for the service to ensure that it executes client requests in order and without gaps. to do this, the service keeps track of the sequence number of the most recently executed request for each client. essentially, this information is considered as part of the running state of the replicated object, so the replication algorithm ensures that all replicas agree on it. the cost is reasonable for services with a limited number of clients. for services with large numbers of clients, this may be too expensive.

a weaker and less expensive approach, which I recommend in this project, is for each client to include a unique ID of any kind (a sequence number, timestamp, random number, etc.) in each request, and for replicas to cache the IDs of recently executed requests together with the result (return value) of the request. when a replica receives a retransmitted request, it first checks whether that ID appears in the cache. if so, it simply resends the cached result. otherwise, the replica assumes the request is new. the result cache may be a fixed size, with the oldest entries evicted as needed. this approach is weak because it allows requests to be executed multiple times, if a request is received again after the entry for it in the replica's result cache has been evicted.

8.4 Persistence

~~Libra consensus~~DiemBFT requires persistent storage of the ledger ~~and certain consensus state (see function save_consensus_state)~~. I recommend simply storing ~~this information~~it in text files; I think a database is overkill for this project. The ledger is intrinsically append-only. To ensure persistence, the data should be flushed to disk immediately after appending ~~a block~~commands, by calling flush then fsync.

~~I recommend saving the consensus state by repeatedly appending the new values at the end of a text file and flushing to disk after each append. Although the old values are not needed, this approach is more robust than overwriting the old values, and the history could be helpful in debugging. The amount of "wasted" disk space is negligible, since the consensus state is small.~~

8.5 Ledger

Formatted: Font: Bold

The paper does not give pseudocode for the Ledger module, since internal details of the ledger may vary and are largely orthogonal to the consensus algorithm. The ledger at least needs to store the sequence of executed and committed client commands. If we want the ledger to be verifiable, or if we want to use information in the ledger to "sync up" slow validators (analogous to using information in the log to "sync up" slow replicas in Raft), then additional information (e.g., QCs) needs to be stored in the ledger.

Ledger.speculate(prev_blkid, blkid, txns): creates a state s, and adds it to the tree of pending ledger states. the pending ledger state stored in s is simply txns (i.e., the sequence of operations in block blkid). Ledger.commit appends txns to the persistent ledger state (i.e., the ledger file).

I suggest that the ledger state id (also called commit state id, when the state has been committed) for s be computed as hash(parent_state_id || txns), where parent_state_id is the ledger state id of the state associated with the parent of the current block blkid. This is consistent with the comment in the paper that the ledger state id should cover the history of the ledger. Note: the paper says that Ledger.speculate returns the ledger state id of the new state, but this is unnecessary; the pseudocode uses Ledger.pending_state to obtain the ledger state id.

Ledger.committed_block(block_id) is introduced in diemBFT v4 solely for use in LeaderElection.elect_reputation_leader, which uses this function to look up recently committed blocks only. this function can be implemented without storing block information in the persistent ledger. it suffices to store (in memory) information about the max(window_size, 2 * |validators|) most recently committed blocks. if elect_reputation_leader doesn't find the desired number of "last authors" within that range of blocks, don't worry about it; just use the "last authors" that were found.

Note that each server should maintain its own ledger file. Since you may run multiple servers on the same host, include a server id in the name of each ledger file.

8.6 Timeouts

Formatted: Font: Bold

If you are using DistAlgo, timeouts should be implemented using a DistAlgo await statement with a timeout clause. here is advice on how to do that.

in the pseudocode in the paper, timeouts are implemented in Pacemaker.start_timer, which starts and stops a timer, and Main.start_event_processing, which handles timeouts (i.e., the timer counted down to zero) by calling Pacemaker.local_timeout_round.

you will need to structure the code somewhat differently, because, as I explained in class after my walk-through of the DistAlgo code for lamutex, using await in receive handlers can be tricky. therefore, I do not recommend including an await statement in Pacemaker.start_timer, since it will naturally get called from receive handlers. I suggest including await in a loop in the replica's run() method or a function called from run(). here is some example code to give you the idea; you don't need to do it exactly like this.

```
 class Replica:
    ...
    def run():
        ...
        run_done = False
        while not run_done:
            round_done = False
            timer_duration = Pacemaker.get_round_timer(Pacemaker.current_round)
            await round_done: pass
            timeout timer_duration: Pacemaker.local_timeout_round()
        ...
```

note: this is DistAlgo ideal syntax, not DistAlgo Python syntax.

Formatted: Space After: 0 pt, Line spacing: single

when a receive handler processes a message that causes the replica to advance to the next round, it sets round_done to True; this allows the process to exit from the await statement and therefore has the effect of stopping (cancelling) the timer for the current round.

To determine a reasonable timeout value, I suggest measuring the max time needed for a round to complete in a failure-free execution with several rounds, and defining get_round_timer to return (say) 4x that value, to be safe. the goal is to avoid unexpected timeouts, so timeouts occur only due injected faults, and we always know what to expect as the result of a test case. Client timeout for request retransmission can be determined similarly. timeout periods should be set via the config file, so they are easy to change.

8.7 Message Loss

Formatted: Font: Bold

Some messages do not need to be re-transmitted even if they are dropped by the network. For example, if loss of Vote messages prevents a round from committing a block, round timers will expire, timeout messages will be sent, and the replicas will enter a new round which will send vote messages for a new block containing the client commands that were in the uncommitted block from the previous round. Therefore, there is no need to re-transmit the lost vote messages. This argument does depend on timeout messages getting through, so they should be resent if they are lost. This is easily accomplished by restarting the round timer after sending timeout messages (in local_timeout_round), since the existing code will resend the timeout messages if the round timer expires again.

A related issue is that some correct replicas may get behind, i.e., not know about a recently committed block. This may occur due to message loss (by the network) or misbehavior by faulty replicas. For example, a faulty replica may temporarily behave correctly as a leader and commit a block without sending any messages to some correct replicas. This motivates the comment in section 2.1 of the diemBFT paper, quoted in part in section 1 above, that "The transport takes care .. retrieving any data referenced by delivered messages, in particular, block ancestors. In particular, when a message is handled in the pseudo code, we assume ... that the receivers has synced up with all ancestors and any other data referenced by meta-information in the message." Your extended pseudocode (and your implementation) need to handle this, as stated in section 1.

My first impression is that these two mechanisms---retransmission of timeout messages, and "syncing up" replicas that got behind---are sufficient to deal with message loss between servers. I have not attempted to verify this, so do not take it as a given. Each team should analyze this issue itself and draw its own conclusions about what is needed to handle message loss.

---

9. Demo

Each team will demo its system to a TA during the period Oct 15-25. The demo should present evidence that all claimed points are deserved. Your team should prepare in advance a demo that does this and takes at most 40 minutes. The demo should consist primarily of running a sequence of test cases and showing from the logs, ledger files, and possibly other output that the functionality works correctly. Test cases used in the demo should also appear in your test report. You are welcome to show parts of the code during the demo, but its correctness must be demonstrated through testing to get full credit. Only partial credit will be given for functionality not demonstrated during the scheduled demo timeslot. During the demo, the TA will fill the "phase 2 actual" column in the grading sheet, and record details of bugs and limitations of your system. Partial credit for partially working items will be finalized during a meeting of the instructor and TAs after the demos.

Exactly one member of each team should sign up for exactly one 1-hour timeslot, by left-clicking on the desired timeslot in the appropriate TA's Google appointments calendar (see below), entering the team's name in the Description field, and clicking Save. **Sign up by October 7.**

Formatted: Font: Bold, Font color: Red

These teams should sign up for a demo with Duin in this Google appointment calendar: 3 P's, 3dot67XP, 6 Digits, Algogetters, Algomaniacs, ARPANET, Asynchronous Techies, AZK, Beyond Borders, Brute Force, Chain of Things, Code Alpha, Commuter Bytes, Cuba Libra, DKS.

These teams should sign up for a demo with Murray in this Google appointment calendar: Evil geniuses, Lightspeed, LNB, Loyal Byzantine Generals, Madrox, PreeManiSai, Red, RVP, SAC, Server Side Squad, System Crackers, Team DNA, Team SSA, Team-Cobra, The Posquatters.

Demos will be held on Zoom (links TBA) and recorded. Demos will proceed as follows:

1. Download your submission from Blackboard. you must demo the downloaded version, not a version already present on the computer.

2. Delete all files produced by compilation, e.g., .pyc files.

3. Run your demo, indicating as you go which test case covers each item on the grading sheet. we might ask you to run test cases we supply before or after your prepared demo, and we may ask you to run variants of your test cases.

~~9~~10. Comments on the ~~Libra consensus~~ papers

10~~9~~.1 Network Reliability

The Libra consensus paper says that it assumes “Every message sent at time t must be delivered by time max{t,GST} + Δ.” This may be interpreted to imply that every message sent before GST is eventually delivered. However, that is not actually required. The paper refers to Dwork et al’s paper Consensus in the Presence of Partial Synchrony for the concept of partial synchrony, and the detailed definition of the GST-based model in that paper (see Definition 2.3, especially item (3)) allows loss of messages sent before GST; the assumption of timely delivery applies only to messages sent after GST. You can see that the liveness proofs in Section 6 of the Libra consensus paper do not rely on delivery of messages from “old” rounds. Also, I confirmed with Dahlia Malkhi—an author of the HotStuff paper, and a member of “The LibraBFT Team” that wrote the Libra consensus paper—that the Libra consensus algorithm tolerates message loss before GST. Therefore, your implementation should take into account that message loss before GST is possible. Also, note that the network is not assumed to be FIFO.

10.2 Errata for Libra consensus paper

In Block-tree.process_vote, V should be PendingVotes.

10.3 Errata for DiemBFT v4 paper

Note: If you find typos or other minor errors in the paper, please share them, and I’ll add them to this list.