SporeDB Whitepaper

Loïck Bonniot (_@lesterpig.com)

Draft Version

Abstract

This Whitepaper introduces a **highly scalable, fast, resilient, decentralized and flexible database engine and its network protocol**, named **SporeDB** by analogy with the Mycology science. Such database system would allow one to build secure voting, membership or assets management, archival, certification or time systems; providing Atomicity, Strong eventual consistency, Isolation and Durability even when a large number of nodes are participating in a malicious or faulty way. SporeDB uses previous academic work about **gossip protocols**, **state-machine replication** and **webs of trust** to achieve its goals.

1 Introduction

1.1 Motivation

Distributed databases are very popular when it comes to service scalability and high availability. Such databases, like Apache Cassandra [1], MongoDB [2] or Redis [3] are able to handle node or network failures, but cannot handle nodes acting in a byzantine way, as introduced by [4].

Solving the Byzantine problem usually involves complex, costly or non-scalable consensus algorithms. We can mention the well-known PBFT [5], the Bitcoin Proof-of-Work [6], the Tendermint protocol [7] or the Stellar CP [8] among many others. Generally, these protocols require some strong coordination between nodes (leadership for example), or are mostly designed for a specific application (crypto-currencies for example). This strong coordination reduces scalability and performance of the global system.

We introduce SporeDB as a way to solve these problems using simple (but powerful) techniques.

1.2 Goals

The main purpose of SporeDB is to share information storage between several nodes that could potentially act as byzantine (faulty or malicious for instance). Every transaction requires a majority or unanimity of nodes endorsements before being executed, while allowing many non-conflicting transactions to be executed in parallel.

Nodes are able to join and leave the network without breaking their consistency, and without slowing down the whole network; like a Mycelium always expands even when some part of it is removed. SporeDB is able to correctly detect network partitions and gracefully recover from them, detect byzantine nodes and report them with strong guarantees, and support up to f Byzantine nodes in a network of $f + 1$ peers. By comparison, current BFT systems usually require at least $2f + 1$ or $3f + 1$ nodes [5].

In order to guarantee better performances than classic BFT consensus algorithms, SporeDB follows a *Strong Eventual Consistency* (SEC [9]) model, adding safety to the liveness property of Eventual Consistency models.

However, for a client dialing with a non-byzantine peer **(our default assumption)**, the SporeDB acts like any regular ACID [10] database, providing Atomicity, Consistency, Isolation and Durability[1].

We suppose that the network is able to eventually deliver a message from a non-malicious node. That being said, a node that is not able to communicate with enough peers is considered byzantine. Thanks to a gossip communication protocol over such network, SporeDB can scale to thousands of nodes while efficiently spotting byzantine nodes.

As an additionnal feature, SporeDB accepts peers that have different *local endorsement policies* without compromising the liveness of the network, while being capable of handling human validation schemes (such as manual acceptation in a membership database, for instance).

Note: this document has been written outside of the academic community, but the author would be really happy to receive some help to reformulate this paper in a more formal way targetting academic publication.

2 Transactions (spores)

The most basic blocks of the SporeDB are the Transactions, named spores. These spores are created by each client to execute one or several operation(s) on the database, and are transmitted between nodes in the network for endorsement and execution. Read-only operations are not propagated to the network: each peer is able to read its own state, because **each peer stores the whole database locally**.

2.1 Data types

The SporeDB is able to store the following data types in a key-value fashion:

- Simple data types
  - Integers of arbitrary length
  - Strings
  - Arbitrary data as bytes
- Composed data types
  - Arrays, with FIFO and LIFO available
  - Sets

2.2 Spore types

A spore's type represents the *operation* that would be executed on the database for a specific key. There may be many types of operations for each of the available data types, and this whitepaper will not be comprehensive in that regard.

For instance:

- On datatype Integer:
- SET
- ADD
- On datatype Array:
- INSERT AT INDEX
- DELETE AT INDEX
- ADD TO SET

[1]We can mix the two notions by calling SporeDB a ASecID database.

2.3 Conflicts between spores

Conflicts will occur when several clients want to update a specific key in a conflicting manner, such as `SET x` and `SET y`. In order to increase performance, many transactions are designed to avoid many conflicts, even when updating the same key. For example, two operations `ADD 5` and `ADD -3` can easily be executed at the same time without creating a conflict. Peers might even execute these transactions in different ways (`+5-3` or `-3+5`) without breaking the SEC property of the database.

This design is heavily inspired from the Redis [3] design. The conflict detection and resolution-when-possible functions are the kernel of SporeDB, and the foundations for the proofs.

However, sometimes it is not possible to avoid conflicts. To gracefully handle these cases, SporeDB maintains a **version number** (can be a simple hash of the value or a universal unique identifier) for each key and sends this number along with the spores. Peers can then compare the version number with their own state before any endorsement or execution. This idea comes from the HyperLedger Fabric new consensus architecture [11].

Finally, each spore has a deadline, detailled later in this document.

To conclude, a spore is structured as this:

- Unique identifier
- Deadline
- Prerequisites (Map between Keys and Latest Version Known)
- List of operations

3 Network (Mycelium)

The SporeDB is decentralized across a large number of nodes, potentially thousands of nodes. This section presents the way nodes discover and trust each others.

3.1 Web of trust

Nodes have to connect to a given number of other nodes (Peers) in order to enter the Mycelium. SporeDB provides a way to authenticate and encrypt communications similar to the PGP Web Of Trust [12]: nodes initially specify a list of peers with an associated trust level (bootstrapping nodes), and share their public keys. Communications are then authenticated using one's private key, and encrypted using trusted peers' public keys.

A trust level specifies how much we trust the other node to correctly authenticate third-party nodes. SporeDB uses standard PGP trust level (ultimate, complete, marginal and unknown), but permits some deeper customization, especially in the marginal level: it is possible to require N marginal-trusted peers to confirm the same data before trusting it. SporeDB call this a N-marginal trust level.

If one node doesn't completely trust any other node, it is advised to use a default (f+1)-marginal trust level, with f maximum number of byzantine nodes. Conflicts are hopefully quickly discovered and signaled. A user of the network would be able to lower the reputation of some nodes if he figures out some strange behaviors when receiving evidences of byzantine actions.

Note: This design, as opposed to central authentication schemes, seems to be weaker. But it obviously offers more reliability in a fully-decentralized network, where no central authority should be trusted.

3.2 Discovery and synchronization

Nodes arbitrarly decide which peers they communicate with. They can choose to only talk to trusted nodes and ignore untrusted incoming messages, *or* prefer nearest geographic nodes, *or* talk with random peers in the network. To empower the network, each node can ask one peer's up-to-date list of available nodes: this way, the global list of peers is quickly exchanged among nodes.

The SporeDB network protocol (Mycelium) is expected to be fast, but it can easily tolerate nodes with unstable connections. There is no "global clock" in the network: each peer chooses its own cadence for spore emission and reception, but slow nodes are more likely to slow down the network if they are critical for spore Endorsement (see consensus).

When a peer enters in a stale state, because it's new in the network or it crashed for a while, it can request a state-transfer from one out of several peers. During such transfers, peers share their global database state with the recovering node. The trust levels are used to analyze the incoming data: it's easy to trust one complete-trusted peer, but it might be more interesting to compare data from several marginal-trusted peers before updating the recovering state. Version numbers are used to optimize the state-transfer and quickly detect byzantine behaviors.

Note: contrary to the blockchain principle, the history of the transactions are not preserved, leading to less cold-storage usage. It allows the network to quickly recover from outages, at the cost of some trust requirement (mycelium web of trust).

For this whole whitepaper, it is expected that nodes are fully discovered, meaning that there is no Mycelium partition due to misconfiguration. To avoid intentional network partitions by byzantine nodes, one peer can ensure the network quality by connecting to at least f+1 peers. **However, temporary network failures causing network partitions are supported and recoverable.**

3.3 Bootstrapping sum-up

When joining a network, the following steps are required for the new node.

1. Specify bootstrapping peers. The user can give their public keys if already known (by another medium), but that is not mandatory ;
2. Fetch nodes list from the peers and merge them. Remember identical public keys signed by enough nodes in order to build a safe `(node, public_key)` dictionnary ;
3. Announce its public key to the network, waiting for some peers to sign it if they are ready to ;
4. Connect to one complete-trusted or several N-marginal trusted peers to obtain a state-transfer of the database ;
5. During the state-transfer, start listening for incoming spores ;
6. As soon as the state-transfer is finished, start the consensus protocol for each received spore.

4 Policies

The different nodes in the Mycelium need a way to exchange information about the allowed operations on a specific set of keys. To achieve consensus easily, nodes hold, exchange and confirm manifests about the way the requests should be processed.

4.1 Global policies

A global policy is a simple manifest shared between nodes, using the Mycelium or any external medium. This manifest contains the **rules** for a specific set of keys (named *Database*), specifying the requirements before any spore execution on these keys. It specifies a list of peers that are allowed to *endorse* a spore by signing it, and a required number of *endorsements* before execution of one spore. It also holds global metadata that should be shared across nodes: standard timeouts, size limits, transaction quotas, confidentiality parameters etc. *Basically, everything that can be useful for the consensus algorithm.*

Each node can fetch the global policies from the peers matching the Web of Trust requirements, and is able to accept or ignore them. Nodes that are sending or accepting spores for a particular database are supposed to observe the associated global policy, but SporeDB supports reluctant and divergent nodes.

Additionally, global policies can be updated via special spores, requiring a specific majorit (usually unanimity). This is useful when adding or removing endorsers for example, but requires a special care to avoid security issues.

4.2 Local policies

An endorsing node is allowed to be reluctant while receiving a specific spore, even if the spore is matching the associated global policy. This is especially useful when building votes systems: a reluctant node can have its very own policy regarding the incoming requests, and is not required to disclose its policy.

The majority requirement defined in a global policy defines the maximum number of reluctant nodes for a spore to be executed. For instance, if a unanimity is required, a single reluctant node will be sufficient to discard the spore (veto).

However, if one spore is endorsed by enough nodes, reluctant nodes have to accept the final consensus and execute the transaction despite their initial wish.

5 Consensus via Gossip

5.1 General design

SporeDB relies heavily on previous academic work on gossip protocols [13][14][15] for their interesting properties; mainly resilience and scalability. Gossip protocols can be used to build fast consensus algorithms [16] thanks to public-key cryptography. We chose a simple continuous gossip protocol inspired by [17]. Nodes relay spores to a randomly or selected set of peers by sending firstly a light metadata, and the whole spore if the peers request it. This simple protocol ensures delivery of a rumor in $O(nloglogn)$ messages complexity with high probability.

In the SporeDB consensus protocol, there is no elected leader responsible for transaction ordering. The protocol is very simple: there is no global synchronization, each peer can start a new consensus and receive feedback on it given the standard assumptions (see the Proofs).

1. The peer creating the spore (*submitter*) sends a `SUBMIT` operation to a set of peers ;
2. Thanks to the Mycelium properties, based on the gossip protocol, we can assume that this `SUBMIT` operation is quickly propagated to a sufficient portion of *endorsers* ;
3. Each endorser checks the spore and determines if it should be accepted or not, based on policies and possible conflicts ;
4. In case of validation, an endorser signs the spore and sends a `ENDORSE` operation in the Mycelium, storing the spore in a "staging" list ;
5. The other nodes wait for enough `ENDORSE` messages for a particular spore (number determined by the global policy associated with the spore), store the spore in their staging list and finally execute it ;
6. If some peers are reluctant and does not want to sign the transaction, they can send `REJECT` in the Mycelium, or in a less efficient way, wait until the spore's deadline.

5.2 Conflict management

- We define as *grace period* the duration given to every non-byzantine node to execute the spore after its deadline. This period is defined in the global policy ;
- We define as *commited* the spores that are executed on a majority of nodes ;

- We define as *dropped* the spores that are either rejected by a sufficient number of nodes, or whose grace period is expired.

Non-byzantine nodes have to respect some rules in order to keep the strong enventual consistency property.

- **Timeout**: A node cannot endorse a spore whose deadline is over according to their clock ;
- **Consistency**: A node cannot endorse a spore which is conflicting with the current state of their database ;
- **Promise**: A node cannot endorse a spore which is conflicting with a non-dropped spore in its "staging" list ;
- **Expiration**: A node cannot execute a dropped spore ;
- **Yield**: A node must execute a commited spore, even if its local policy is not fulfilled by this spore.

The required amount of endorsers is specified in the global policy, and must be at least $\left\lfloor \frac{n+f}{2} \right\rfloor + 1$, with n total number of endorsers and f maximum number of byzantine endorsers. This requirement is explained and proved in the proofs section of this document.

For example, with $n = 10$ endorsers and $f = 3$ maximum number of byzantine endorsers, the global Policies would require a majority between 7 and 10 endorsers, depending on the targetted application.

5.3 Dealing with poison

Some poison appears when one node endorse a spore while ignoring the Conflict Management Rules. In that case, thanks to the Mycelium and under our communication assumptions, it is guaranteed that *at least one node* will receive enough evidences that the concerned node has emitted some poison. Thanks to the cryptographic signatures included in the endorsements, it will be easy to denounce the poisonous node with the collected, poisonous endorsements. Other peers may be able to report the issue and act accordingly (for instance lower the trust in the poisonous node, avoid any communication with it, etc).

Despite the interesting properties provided by the consensus algorithm, a node might being corrupted. This can be the case for nodes acting as faulty during a period of time, and trying to recover from a corrupted state. In such situation, corrupted nodes will have to re-synchronize with their trusted peers, using state-transfer.

6 Proof of Correctness

6.1 Definitions

We re-use the previous definitions present in this paper for the proofs, and add additional elements.

Definition: Set of peers - A node i is able to communicate with every node in the set P_i

Definition: Peer Directed Graph (PDG) - The PDG is the graph $G = (V, A)$ where

$$p \in V \equiv \text{p node in the network}$$

$$\forall x \in V, \forall y \in P_x, (x, y) \in A$$

6.2 Assumptions

We recall the assumptions made across this document in order to validate the correctness property of the SporeDB Consensus Algorithm. We consider a network of n endorsing nodes with at most f byzantine nodes.

1. The PDG is $(f + 1)$-strongly-connected, and each node relays messages to at least $f + 1$ peers ;
2. The communications between two non-byzantine peers are possible in a finite amount of time ;

3. Every non-byzantine node has enough non-byzantine peers in their web of trust to execute safe state-transfers ;
4. Every non-byzantine node has the same initial state ;
5. Every non-byzantine node has a clock synced with at most δ delta ;
6. Messages cannot be spoofed or corrupted, and nodes are computationnaly-bounded to preserve public-key cryptography safety ;
7. If ω is the number of required endorsements to execute a spore, the following property must apply: $\left\lfloor \frac{n+f}{2} \right\rfloor < \omega \leq n$

6.3 Safety

In order to prove the safety of SporeDB, we prove that the database and its consensus provide ASecID transactions for a client dialing with a particular non-byzantine node in the network.

Lemma 1 - *Every message sent in the Mycelium is received by every node in a finite amount of time, called **dissemination time** τ.*

Proof: Let's assume that the Lemma 1 is wrong. The PDG remains strongly connected (1) even if up to f nodes stop relaying messages to their peers (acting as byzantine). A message containing a invalid signature will be ignored, and the emitting peer considered as part of the byzantine nodes thanks to the assumption (6). Therefore, each node in the remaining peer graph is able to send a message to any other node in a finite number of relays. Finally, with (2), there exists a upper bound to the duration between the sending of a message and its reception by every non-byzantine node. This is a contradiction, the Lemma 1 is true. □

Lemma 2 - *Every node eventually owns the same version of the global policy for a specific spore.*

Proof: This is a proof by induction. For $n = 1$, the Lemma 2 is trivially correct. We assume that the Lemma 2 is true for $n = k$. If we add a node in the network, the assumption 3 guarantees that this node will fetch the original global policy thanks to a well-formed web of trust. The communication property is provided by Lemma 1. The Lemma 2 is therefore true for $n = k + 1$. By induction, the Lemma 2 is true $\forall k > 0$ nodes. □

Theorem 1 (Atomicity) - *For a specific spore and a specific non-byzantine node, each operation is executed, or the whole set of operations is rejected.*

Proof: Let's assume that the Theorem 1 is wrong. The only way for a non-byzantine peer to apply partially the operations of a spore is to receive a corrupted spore and enough endorsements for this spore. However, the Lemma 1 guarantees that every message is transmitted across the network without corruption. This is a contradiction. The Theorem 1 is true. □

Theorem 2 (Strong Eventual Consistency) - *Every spore will produce the same set of updates in every non-byzantine node within a long-enough grace period, eventually leading to the same global state in the whole network.*

Proof: Let's assume that the Theorem 2 is wrong and that the grace period of a spore $G(S)$ is long enough compared to the deadline of spore $D(S)$ and τ.

$$G(s) \geqslant D(s) + \tau$$

With this additional assumption, we can guarantee that the non-byzantine endorsements messages are all emitted and received before the end of the grace period, given the Lemma 1, the assumption (5) and the **Timeout** rule. It is then impossible to execute spores after the end of the grace period thanks to the **Expiration** rule. Additionnaly, given the Theorem 1, we can say that either a spore is completely executed or completely ignored on a non-byzantine node.

We also know that every non-byzantine node owns the same global policy (Lemma 2) and we assume that every non-byzantine node is in the same state (4). Hence, if one non-byzantine node executes a spore, others will also execute this spore (**Yielding** rule) **unless** they've applied a conflicting spore in the meantime (**Consistency** rule). Given all these assumptions, Theorem 2 is wrong **if and only if** at least two non-byzantine nodes receive enough conflicting endorsements to execute conflicting spores, leading to a fork in database contents.

However, given the **Promise** rule, only byzantine endorsers can announce two conflicting endorsements. In order to create a fork, the f byzantine endorsers have to send at least two conflicting endorsements of two conflicting spores, trying to fool other endorsers. In the set V of endorsing nodes ($|V| = n$), this is equivalent of creating two sets A and B with $|A \cap B| = f$ and ω required endorsers in both A and B.

$$\omega \leqslant min(|A|, |B|)$$

$$\omega \leqslant |A - B| + |A \cap B| \ \textbf{(majoration)}$$

$$\omega \leqslant \left\lfloor \frac{n-f}{2} \right\rfloor + f \ \textbf{(majoration)}$$

$$\omega \leqslant \left\lfloor \frac{n+f}{2} \right\rfloor \ \textbf{(contradiction with (7))} \ \square$$

Theorem 3 (Isolation) - *Concurrent execution of* ***non-conflicting*** *transactions results in a global state identic to the global state that would be obtained if these transactions were executed serially.*

Proof: This is a proof by induction. For $s = 1$ spore, the Theorem 3 is true, falling under the Theorem 2 case. We assume that the Theorem 3 is true for $s = k$, and we want to execute another non-conflicting spore in parallel. Thanks to the Theorem 1, spores are executed in a atomic fashion; and with the Theorem 2, either the additional spore is executed on every non-byzantine peer, or not executed at all. The Theorem 3 is therefore true for $s = k + 1$. By induction, the Theorem 3 is true $\forall k > 0$ spores. $\square$

Theorem 4 (Durability) - *Once a spore has been commited, it is guaranteed that the operations of this spore have been or will be executed in every non-byzantine node.*

Proof: This property is trivially provable given the Theorem 2 for non-byzantine nodes. For crashed nodes that need to recover, the assumption 3 guarantees that they will be able to recover a clean state using state transfers. $\square$

6.4 Liveness

Theorem 5 - *A spore is either commited or dropped in a finite amount of time.*

Proof: If the spore is executed by a non-byzantine node, the Theorem 2 ensures that it will be executed on every other non-byzantine node. If the spore is being rejected or ignored by a sufficient amount of nodes, then the spore's grace period will finally expire for every node, and the spore will be dropped given the assumption 5. Hence, the liveness is guaranteed as long as $G(s)$ is reasonably high compared to τ, and δ is low across non-byzantine nodes. $\square$

7 Conclusion

We have introduced SporeDB in this paper, a ASecID leader-free decentralized database engine, supporting byzantine nodes. This database engine is designed to be flexible and secure, allowing different nodes to have different policies regarding transaction endorsements. The gossip protocol used guarantees the scalability of the database, possibly leading to networks of thousands of nodes.

However, we need strong mathematical assumptions regarding the network topology and the message delivery. In practice, choosing good values for the deadline $D(s)$ leads to optimal values for the grace period $G(s)$, because the deadline can be determined easily by measuring τ regularly. Failed or satured nodes are gracefully handled, considering them as byzantine nodes; and recovering them using trusted state-transfers.

SporeDB is being implemented in an open-source governance model in order to evaluate and verify the assumptions made in this paper.

Acknowledgements

We thank Quentin Dauchy and anonymous reviewers for their useful comments and suggestions on the different drafts of this paper.

References

[1] A. Lakshman and P. Malik, "Cassandra," *Proceedings of the 28th ACM symposium on Principles of distributed computing - PODC '09*, 2009.

[2] MongoDBInc., "Organization website," Sep-2016. [Online]. Available: https://www.mongodb.com/.

[3] Redislabs, "Organization website," Sep-2016. [Online]. Available: http://redis.io/.

[4] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems ACM Trans. Program. Lang. Syst. TOPLAS*, vol. 4, no. 3, pp. 382–401, Jan 1982.

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst. TOCS ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Jan 2002.

[6] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Nov-2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[7] J. Kwon, "Tendermint: Consensus without Mining," Sep-2016. [Online]. Available: http://tendermint.com/docs/tendermint.pdf.

[8] D. Mazières, "The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus," Feb-2016. [Online]. Available: https://www.stellar.org/papers/stellar-consensus-protocol.pdf.

[9] C. B. Marc Shapiro Nuno Preguiça, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.

[10] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *CSUR ACM Comput. Surv. ACM Computing Surveys*, vol. 15, no. 4, pp. 287–317, Feb 1983.

[11] E. Androulaki, "Hyperledger Next Consensus Architecture Proposal," Sep-2016. [Online]. Available: https://github.com/hyperledger/fabric/blob/master/proposals/r1/Next-Consensus-Architecture-Proposal.md.

[12] A. Abdul-Rahman, "The PGP Trust Model," *EDI Form*, Apr. 1997.

[13] A. Ganesh, A.-M. Kermarrec, and L. Massoulie, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 139–149, 2003.

[14] M. Gurevich and I. Keidar, "Correctness of gossip-based membership under message loss," *Proceedings of the 28th ACM symposium on Principles of distributed computing - PODC '09*, 2009.

[15] A. Allavena, A. Demers, and J. E. Hopcroft, "Correctness of a gossip based membership protocol," *Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed*

computing - PODC '05, 2005.

[16] J. Lim, T. Suh, J. Gil, and H. Yu, “Scalable and leaderless Byzantine consensus in cloud computing environments,” *Information Systems Frontiers Inf Syst Front*, vol. 16, no. 1, pp. 19–34, 2013.

[17] C. Georgiou, S. Gilbert, and D. R. Kowalski, “Meeting the deadline,” *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing - PODC '10*, 2010.