Project Report

State-of-the-art and trends on PBFT-inspired and asynchronous consensus for Blockchain: dBFT 3.0

Igor M. Coelho[1,2], Vitor N. Coelho[3], Rodolfo P. Araujo[1], Wang Y. Qiang[4], Brett Rhodes[5]

1 Graduate Program in Computational Sciences (PPG-CComp), Universidade do Estado do Rio de Janeiro, Pavilhão João Lyra Filho, 6º andar - Bloco B, Rua São Francisco Xavier, 524 - Maracanã, Rio de Janeiro - RJ, 20.550-013, Brazil
2 Institute of Computer Science, Universidade Federal Fluminense, Av. Gal. Milton Tavares de Souza, São Domingos, Niterói - RJ, 24210-310, Brazil
3 OptBlocks, Avenida João Pinheiro, 274 Sala 201 - Lourdes, 30130-186, Belo Horizonte - MG, Brazil
4 Research & Development Deparment, NEO Global Development, Shanghai, China
5 NEO News Today, Leeds, United Kingdom
* Correspondence: igor.machado@gmail.com, vncoelho@gmail.com

Received: date; Accepted: date; Published: date

Abstract: Report by NeoResearch: theoretical basis of BFT consensus and perspectives for dBFT 2.0 and dBFT 3.0.
(Pre-release v2020-01-19)

Keywords: dBFT; PBFT; Byzantine Fault Tolerance; NEO Blockchain; Distributed Computing; Timed Automata; Consensus; LibBFT

Key Contribution: Highlights state-of-the-art consensus protocols applied on Blockchain.

---

1. Introduction

This report elaborates on existing Consensus technologies for Blockchain, and indicates promising directions for next generation consensus for NEO Blockchain. It is intended to not dig deeper on any concept (in a non-technical fashion). Yet, it intends to go deep enough to demonstrate the existing challenges, and proposals on how to deal with them. Some basic concepts of NEO Consensus system, the Delegated Byzantine Fault Tolerance (dBFT), is also presented.

This report is organized in four sections, besides this introduction. Section 2 presents background on Byzantine Fault Tolerance technologies and its fundamental concepts. In Section 3, it is presented a brief history of dBFT algorithm, with details of first dBFT. Section 4 describes its evolution to dBFT 2.0, including achievements and existing limitations. Finally, Section 5 closes the report, with indications on promising extensions of dBFT towards a dBFT 3.0.

2. Background

2.1. Twenty years of practical byzantine fault tolerance

Over twenty years ago, the groundbreaking work on Practical Byzantine Fault Tolerance (PBFT) by Miguel Castro and Barbara Liskov [1] have finally demonstrated the practical feasibility of dealing with systematic byzantine[1] faults on a consensus system, even in face of a very strong adversary. The

[1] The term Byzantine was coined by Leslie Lamport [2], on the Byzantine Generals Problem, which simply represents a faulty machine that shows arbitrary behavior, thus invalidating or even attacking the protocol itself.

resiliency of such system was proven to be optimal, resisting up to f faulty/byzantine nodes, from a total of $N = 3f + 1$ replicated state machines [3]. The reason why $3f + 1$ nodes are required is also straightforward due to the hostile nature of the considered network: messages can be delayed, delivered out of order, in duplication, or even lost. Since f nodes can be faulty, you need to decide after communicating with the other nodes. But even when you receive f responses from them, you still cannot know if those are the faulty ones or not! Maybe the others were just delayed, so you need extra $f + 1$ confirmations to be certain of the decision.

2.2. Network Properties

The PBFT work presented the following properties of the underlying network:

(P.1) fail to deliver messages
(P.2) delay messages
(P.3) duplicate messages
(P.4) message delivery out of order

We introduce another network constraint, which is fundamental to a decentralized public Blockchain ecosystem:

(P.5) the network communication is performed by collaborative nodes, that can join and leave at will
(P.6) requests are managed via a *memory pool* mechanism, that aggregates requests not yet executed

Because of its decentralized nature, a Peer-to-Peer (P2P) design is usually adopted for Blockchain networks, satisfying (P.5). Although it may not be beneficial in terms of efficiency, it favors resiliency.

According to (P.6), *mempool* is naturally *non-deterministic*, since there's no guarantee that two nodes will have the same pending requests.

2.3. Message Delivery

Most works consider an *asynchronous scenario*, where messages are delayed, but will eventually arrive. This way, previous works argue that *guaranteed delivery of messages* can be achieved by replaying messages, as a way to circumvent (P.1), a strategy that we will call $\overline{(P.1)}$.

On practice, there is no central authority or absolute control over the P2P network (including its software), so it is very hard to guarantee that certain properties hold, such as $\overline{(P.1)}$. Software bugs and upgrades on different clients (at different times) may generate unexpected behavior, leading to eventual message loss and temporary disruption of P2P channel. These disruptive events may also be caused by external decisions related to offchain governance and incentives for nodes in the network.

This way, a well designed and healthy decentralized network can guarantee *at least* that messages are *almost always* delivered, unless of extremely rare events (where it can eventually halt), which we call $\widetilde{(P.1)}$. In our perspective, this modeling approaches more closely a public decentralized Blockchain ecosystem. This path is also in line with decisions of public Blockchain projects, such as NEO [4], that do not expose public IPs of consensus nodes, nor connect them directly (thus depending on a healthy network to be able to communicate).

So, in this work, we will explore alternative models to deal with (P.1), namely $\widetilde{(P.1)}$ and $\overline{(P.1)}$, which may not be a good approach to devise correctness proofs, yet it is useful to argue for pros and cons of existing consensus models.

2.4. Adversarial Model

We consider a very strong adversary (similar to [1]), which is capable to:

(A.1) coordinating faulty nodes, with arbitrary failures
(A.2) delaying communication (as seen in Section 2.2)

(A.3) completely block the network and drop messages, in extremely rare situations[2]
(A.4) delaying any correct node, up to a polynomial timelimit[3]
(A.5) explore issues of *byzantine fault tolerance privacy*, as leaked information from non-faulty replicas can be explored by byzantine agents

We also consider the following limitations on the adversary:

(L.1) it is compute bound, such that it cannot subvert the given public-key cryptography or generate hash collisions
(L.2) it cannot fake message authentication, due to (L.1)
(L.3) correct nodes have independent implementations and infrastructure, such that no general failure can be explored

The (A.4) is a weak synchrony assumption, thus allowing a non-violation of the celebrated FLP result [5], that forbids the existence of a purely asynchronous consensus (from a deterministic perspective). On the other hand, there exists recent works on literature that propose efficient purely asynchronous consensus [6], claiming that its "non-deterministic nature" during decision-making allows circumventing the FLP result. In this case, condition (A.4) is not required to hold, which we will call $\overline{(A.4)}$. This will require an *asynchronous network*, so (A.3) is not expected to hold anymore, which we name $\overline{(A.3)}$.

2.5. PBFT Discussion

We have presented basic concepts that support the development of the most recent (and efficient) consensus systems. Most of these concepts date back to PBFT [1], a consensus proven to have optimal resilience while handling Byzantine Faults. PBFT handles *safety* in an asynchronous manner, being correct according to linearizability [7], but it depends on a weak synchrony (A.4) to guarantee *liveness*. PBFT has three phases in its algorithm: `pre-prepare`, `prepare` and `commit`.

In short, the PBFT intends to find a *global order* for a given *set of operations* (more details will follow later). To accomplish that, it divides its nodes into two types: one *primary* and a set of *backups*. The primary performs the ordering (giving an unique *sequence number* to each operation), while the others verify it, being capable to fully replace the primary when it appears to have failed (by means of a protocol called *view change*). There are many interesting extensions of PBFT, and some of them will be cited in next section. We focus a little bit now on the main story character: the dBFT.

3. A Brief History of dBFT

The Delegated Byzantine Fault Tolerance, known as dBFT, was proposed in 2014 during early design and development of NEO Blockchain [4]. In fact, it can be seen as one of the core components of NEO (possibly the most fundamental one), as it allows a set of N independent nodes (called Consensus Nodes) to cooperate in a global decentralized network, with optimal resilience against byzantine faults: up to $\lfloor \frac{N-1}{3} \rfloor$ nodes.

Besides the groundbreaking contributions of PBFT, the dBFT includes ideas also discussed in other works from the literature. One example is a round-robin selection of primary nodes, also found on Spinning BFT [8], that drastically reduces the attack surface on the consensus, reducing damage when the primary node is the faulty one.

The dBFT goes much beyond an ordinary component that provides global ordering, since its nodes are also representatives (or *delegates*) of the entire NEO Blockchain network, elected by NEO token owners. Consensus Nodes enforce *onchain governance*, while dBFT ensures all theses decisions

---

[2] This condition is due to previous decentralized peer-to-peer discussions, according to $\widetilde{(P.1)}$. Note that it is assumed that it can only happen rarely, otherwise the adversary would simply halt all communications during all times.
[3] We consider the same delay conditions as [1], where delays can never grow exponentially

(including transaction computation) are viewed consistently by all nodes in the network, as a unique Global State.

In this sense, an unique feature of dBFT (and NEO Blockchain since its conception) is its capability to provide Single Block Finality (or One Block Finality – 1BFT), such that every decision is final once approved/signed by $2f + 1$ consensus nodes (or simply $N - f$). This was devised as one of the core goals of the platform and enforced since early design phases of the project, however it's not a trivial task to put on (even counting on existing PBFT background, it was needed to move one step further).

3.1. dBFT 1.0

The first version of dBFT just included two phases: `prepare-request` and `prepare-response` (corresponding to PBFT's `pre-prepare` and `prepare`). This was a necessary simplification, due to high communication costs and natural efficiency issues faced by a first prototype. Removing a third phase allowed consensus to perform much faster, specially in a peer-to-peer network with large delays. On practice, it demonstrated how really strong is the adversary, according to (A.2).

We start demonstrating three "good cases" for dBFT 1.0: (a) no replica is faulty (Figure 1); (b) one replica is faulty, but not primary (Figure 2) (c) primary is faulty, requiring a change view (Figure 3).

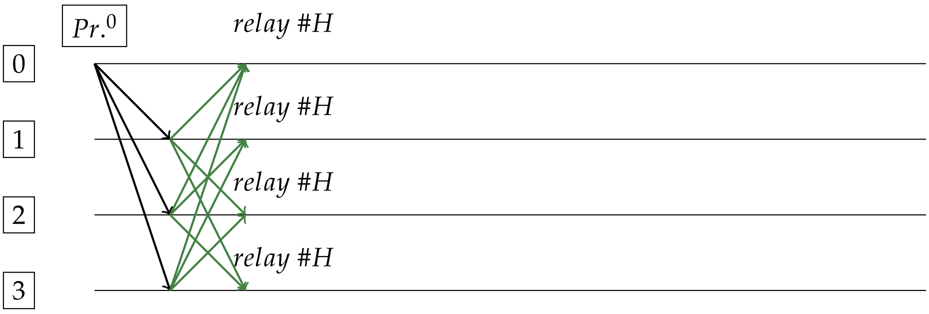

Figure 1. Perfect case. Request sent and received by Primary (replica 0). Everyone relays block $\#H$.

Figure 2. Good case (faulty non-primary replica 2). Request sent and received by Primary (replica 0). All except 2 relays block $\#H$.

3.2. Forks and sporks

One side effect of this decision (not having a third phase) was that, due to network delays, even when no byzantine node was present among Consensus Nodes, multiple valid blocks could be eventually forged (for the same Blockchain height). During community discussions, this phenomena was once called a *spork*, highlighting its difference from a classic *chain fork* (that happens naturally on blockchains without One Block Finality, such as Bitcoin [9]). But since a *spork* does not allow multiple (endless) continuations after each block, a single orphaned (but valid) block was created, and progress could continue on the other block. Although not explicitly forking the network, it could potentially generate issues, as nodes that accepted the orphaned block would never give up on it (due to 1BF

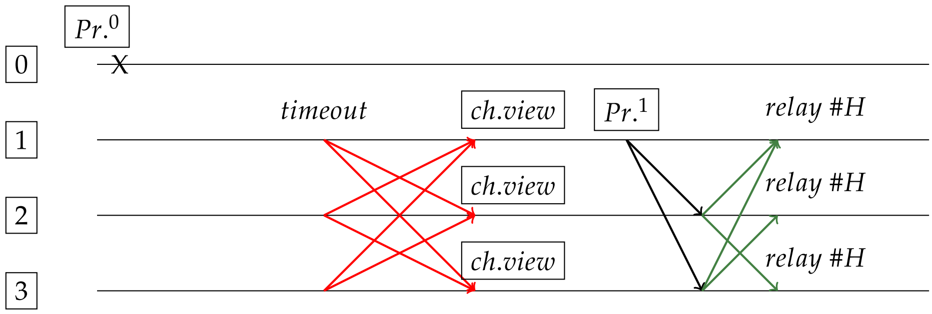

Figure 3. Primary 0 is dead and no proposal is made. Replicas will change view after timeout (red lines) due to not receiving each other's responses in time. Replica 1 becomes new primary ($Pr.^1$) and successfully generates block $\#H$.

policy). These affected nodes would then require a manual software restart, parsing the whole chain again, in order to finally attach the correct block on top of its chain.

The reason behind this phenomena was simple: some primary node could eventually propose a valid block, receiving support from enough nodes (at least $2f + 1$), but these same nodes could only receive messages from each other after a long time (due to (A.2)). This would cause a *view change*, since the given timeout for that view has expired (on every replica), so another backup would become primary and propose another valid block for that same height, receiving again enough signatures from its colleagues (at least $2f + 1$). This situation is presented in Figures 4-5.

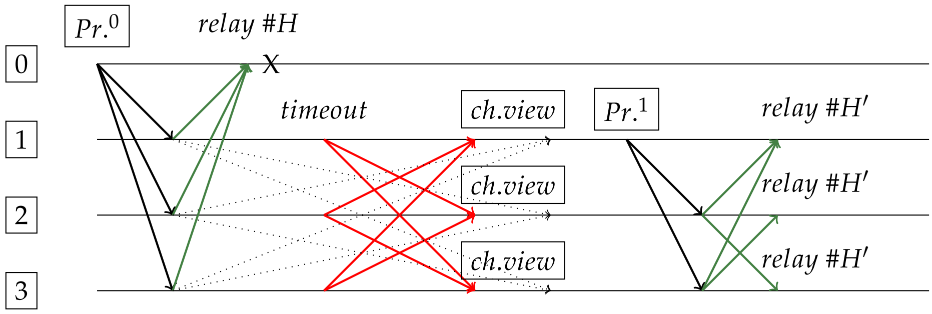

Figure 4. Request sent and received by replica 0 ($Pr.^0$ means primary of view 0). Others won't follow it, since they will change view (red lines) due to not receiving each other's responses (green lines) in time. Replica 1 becomes new primary ($Pr.^1$) and generates "spork" (blocks $\#H$ and $\#H'$ are in same height but with different contents). The "X" indicates a complete node crash. Ignored messages are dotted lines (received after a change view has already occurred).

We still lacked an important explanation. The adversary is very strong as we know, but the problem seemed to happen only from time to time... sometimes it took nearly a month to happen again. The peer-to-peer network had large delays, but since NEO blocks are generated every 15 seconds, and first view change only happened at twice this value, we would require a delay of at least 30 seconds on the network for a spork to happen! Yet, it kept happening from time to time.

3.3. Sporks explained

The dBFT doesn't have an unified notion of "time", so every node has its independent timer. Each of them times out when they believe primary has failed (with exactly the same timeout value T), but note that this may happen in completely different moments. The reason is that each node depends on the *persist time* of the previous block to start its next timer, since it is local information (and trusted regardless of primary node, differently from the block timestamp itself).

So, the only possibility for a spork to happen naturally is that the primary have already been delayed (by perhaps a delayed reception of previous block), so it starts to work and finishes its valid

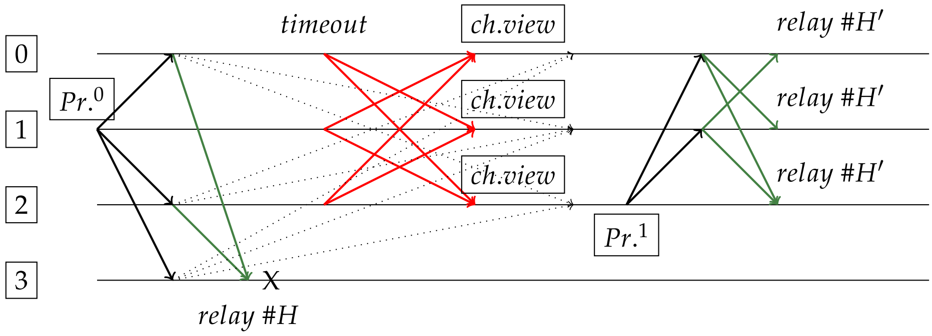

Figure 5. Request sent by replica 0 and $2f$ `prepare-response` to replica 3. Primary and others won't follow it, since they will change view due to not receiving each other's responses. Replica 1 becomes new primary and generates "spork" (blocks $\#H$ and $\#H'$, in same height but with different contents). The "X" indicates a complete node crash. Ignored messages are dotted lines.

block just in time for all the others to expire their timers. And just in that precise moment, none of them happen to receive any message from each other nor the delayed primary, so *puf*: we got a spork.

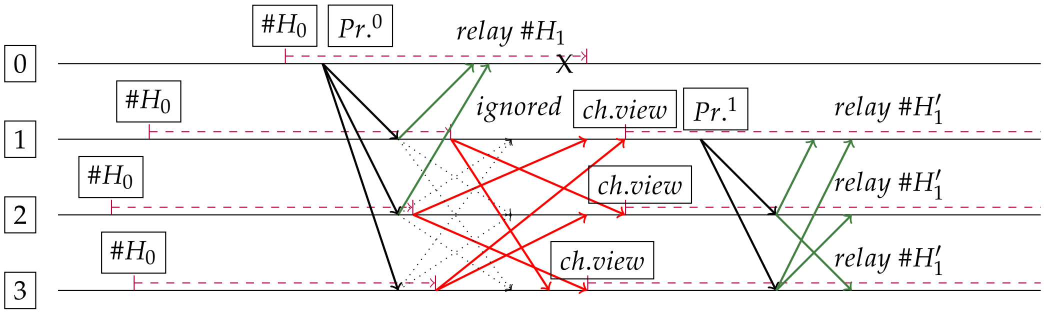

Figure 6. More "realistic" case analogous to Figure 4, when Primary proposal is naturally delayed due to a delay on previous Block persistence (event $\#H_0$). Timeouts are set to $2^v \cdot T$ time units (in purple), where v is view number. Note that most prepare response messages are *ignored* after Change View has started, even with a regular and very small delay. Since Primary was already delayed in relation to others, time to respond was actually minimal before a protocol change occurred on replicas 1, 2 and 3.

This time difference is still a challenging issue on dBFT 2.0, which will be further discussed during dBFT 3.0 proposal. So how dBFT deals with delays on block proposals (avoiding large differences on block timestamps)? In short: the current time difference between nodes is allowed up to several minutes (10-20 min, to be precise), so we still need another mechanism that further reduces this difference (up to second/millisecond scale). Clock drifting is a real problem on machines, but nodes can collaborate with each other guarantee that a much smaller time difference is tolerated (see proposed solutions on dBFT 3.0 section).

3.4. *towards a solution to sporks*

Solution to sporks was evident: dBFT needed a third phase. It was clear that block signatures could not be exposed to network before the replica was confident that other nodes would follow it, otherwise multiple valid blocks could be generated at each height (as seen on Figures 4-6). This issue is related to adversarial condition (A.5), where byzantine agents can aggregate signatures $(\#H)_{\sigma_i}$ from $2f+1$ replicas (sent on prepare response), allowing the creation of a valid block $\#H$ even if all $2f+1$ replicas i finally decided to relay $\#H'$ after a change view.

An interesting characteristic of this issue, from a practical/applied point-of-view, is that it only happened eventually (few times per month), even on a blockchain with systematic block generation (average 15 seconds). Considering that all replicas are non-faulty, only a very "unlucky" combination of delays could lead to such scenario, taking several days to happen again. From a theoretical perspective, the issue is easy to reproduce it but very hard to fix, as we will see on next section.

4. dBFT 2.0

The development of dBFT 2.0 had specifically targeted a solution to the "sporks". This was consistent to a scenario where code quality has been drastically improved[4] (up to 100x), due to several code optimizations and the adoption of the Akka framework (removing several locks from Consensus/Blockchain system). In this new scenario, the adoption of a third phase on the consensus was no longer deemed prohibitive, in order to keep achieving slightly over the 15-seconds average block-time on practice.

5. Promising dBFT Extensions: towards dBFT 3.0

A next generation of dBFT 2.0 may help NEO to achieve better efficiency on consensus, and also to allow broader community participation on the process. In this design phase, we believe two directions are worth exploring: (a) a weak-synchronous extension of current dBFT 2.0 (b) a purely asynchronous version inspired by other recent works in literature.

For for both directions, some common design principles were proposed.

5.1. Design principles

Proposal of dBFT 3.0 intends to improve current consensus in the following manner:

- (X.1) in-time consensus: exact T seconds with block finality (on average)
- (X.2) allowing multiple "simultaneous" block proposals (in same view)
- (X.3) remove the "special status" of a single Primary node: by having (X.2), a single faulty replica cannot delay block generation
- (X.4) community participation: by having (X.2), one can afford letting "unvoted" and "highly staked" *guest nodes* to participate on consensus, eventually contributing and also earning fees
- (X.5) performance enforcing of replicas: precise timestamp verification and (temporary) blacklist mechanisms for misbehavior/faults

Some interesting extensions can also be included, such as:

- (X.6) community interaction messages from nodes via p2p
- (X.7) better monetization for nodes: novel division of System and Network fees

We believe that (X.6) can incentive the early report of failures (or even scheduled ones) via p2p, and instantly passed to users via integration with social networks, wallets and traditional messaging systems. This can also be used (although in a limited way) to provide new monetization means for consensus nodes, by exploring its visibility and "status", to merchandise for its affiliate projects via p2p. This strategy is also part of (X.7), that consists of allowing a basic and sustainable funding for nodes via: (a) minimum fixed amount of GAS (partially by System Fees or GAS inflation); (b) variable funding via fair participation on consensus process, where next "primaries" should use a given strategy to determine which nodes are worth more or less fees based on previous participation.

5.2. dBFT3-WS2: double proposals on dBFT

A known weakness of PBFT, dBFT and others primary-based consensus (spinning, etc) is that consistent primary failures will always trigger a change views (losing time every round). In the spirit of dBFT 3.0 with multiple proposals, one way to extend single block proposal limitation (single primary) is to simply extend it to *two primaries*. We were hopefully able to design such system in a safe manner,

---

[4] This value is an average from the ones unofficially reported by several core developers, during this project phase in 2018.

by introducing an extra `Pre-Commit Phase`. We call this consensus: *dBFT3 with weak-synchrony and two primaries* (dBFT3-WS2). An interesting property of dBFT3-WS2 is that, although it may require four phases to achieve consensus, for the **great majority**[5] of cases, it could shortcut directly to `Commit Phase`, leading to three phase consensus. This happens when network is synchronous and first primary is not faulty, or even when network is synchronous and second primary is alive (when first is dead).

The general strategy is quite similar to dBFT2, which we present step-by-step for dBFT3-WS2:

(W.1) Replicas receive past block H_0 with timestamp $t(H_0)$

(W.2) Primary 0 of next block on view 0, i.e. $Pr.^{0,0}$, set next timeout to $t(H_0) + T$, while Primary 1 of same view 0, i.e. $Pr.^{1,0}$, sets its to $t(H_0) + 4\frac{T}{3}$ (extra T/D seconds, e.g. for a parameter $D = 3$)

(W.3) If Primary 0 is well, it will release a good proposal H_1^0 on time

(W.4) If network delays are not huge (good case), non-faulty replicas will receive proposal and respond to it. In this case, Primary 1 do not need to send proposal H_1^1, but there may be good reasons to do it anyway, as discussed further on.

(W.5) This still follows dBFT2 pattern of `PrepareRequest` and `PrepareResponse` phases. Now, we aggregate $2f$ responses and the proposal into a *pre-commit*. The interesting point of `Pre-Commit` phase is that it can be skipped completely if two situations arise:

   (a) if *pre-commit* includes $2f + 1$ responses (including proposal) to Primary 1, replica commits to 1
   (b) if *pre-commit* includes $f + 1$ responses (including proposal) to Primary 0, replica commits to 0 (obviously, the same happens if $2f + 1$ responses to 0)

(W.6) This strategy allows Primary 0 to have priority over 1, thus resolving consensus much quickly, and directly triggering a `Commit` message (with signature release, that cannot be undone)

(W.7) one can note that *pre-commit* may not have either 0 or 1, and in this case (a hard one), replica *pre-commit* is *undecided*

(W.8) if replica is *undecided*, it will wait for other pre-commits, and by aggregating information from them it may finally commit.

(W.9) if no commit is issued, and timer expires, replicas will issue a Change View (same as dBFT2), but unlike dBFT2, replicas will also include all available information on ViewChange message (a last effort to force a commit before view changes).

(W.10) on next round, some interesting strategies are possible: trying to stick to previous strong proposals (with next primary re-proposing them), thus allowing newer responses to them and resolve undecided situation; or even starting from zero, if no good proposal seems to exist.

5.2.1. Byzantine attacks on dBFT3-WS2

Byzantine nodes can be faulty up to f, and by systematically not replying to messages may cause undecided situations to arise, requiring a change view (only faulty node is a non-primary, otherwise there's no undecided situations). This happens to likely be the worst actions a Byzantine actor can do on dBFT3-WS2 (shutdown its own replica), as discussed in next paragraphs.

Malformed proposals or responses are not smart attacks, since they are directly detected and will trigger a temporary *blacklist* action on that replica. They cannot fake other identities as well, due to public-key cryptography. A Byzantine primary can still propose a "valid" block, but including certain transactions that are not found on mempool (or doesn't even exist), according to (P.6). However, this Byzantine action still cannot force nodes to commit to a bad proposal, as every non-faulty response to that proposal would require a non-faulty replica to actually have all the transactions (guaranteeing that block is valid afterall).

On the other hand, a tricky thing that a Byzantine node can try to do is to respond *simultaneously* to multiple proposals (also using network asynchrony to its favor), in the hope of committing other

[5] A big set of experiments if performed specially to detect such cases: https://github.com/NeoResearch/dbft3-spork-detect

nodes to follow different proposals (thus producing a spork). There are two ways of circumventing this malicious attack, according to the global strategy of network. One thing is certain: any detected double proposal will issue a *blacklist* on that node, with all side effects it will have according to an offchain governance. Yet, it could cause temporary problems on the network, so one must choose between being more *cautious* or more *speculative* according to responses of replicas. The interesting thing of this double approach, is that this applies not just to consensus nodes, but to every node on network, allowing one to take more risky actions to allow faster convergence, while others decide to move a little bit slowly (few milliseconds extra) but with certain guarantees. Here are the options:

1. To detect double proposals efficiently (and blacklist faster), one can aggregate $2f + 1$ *pre-commits* before issuing/following a commit. This leaves no space for Byzantine nodes to pass undetected[6]
2. To be more speculative, one may follow *pre-commits* fast strategies (thus allowing to skip one phase in most cases), and issuing a blacklist on Byzantine node only after it is detected (perhaps after issuing its own Commit, that may cause a spork)

The question is: what are the interests of a Consensus Node issuing a double response, in trying to spork when values are at stake? This question doesn't need even to be fully resolved as a $\{0, 1\}$ matter. One node may be more speculative according to the responses of some *more trusted* replica, while being more conservative according to other (perhaps towards an unknown guest node). Many *flavors* are possible here (both for consensus and network nodes), according to overall network policy and values at stake. We believe that this opens even more doors for community discussions on the topic, allowing different solutions to different implementations of Consensus Node for dBFT3-WS2.

5.2.2. Some initial experiments

In order to validate the idea of dBFT3-WS2, we have performed some initial experiments[7]. The intention is to discover which scenarios are capable of generating sporks, if any. We believe this is a faster approach to validate the idea, before devising possibly complex proofs. This approach is also supported by the fact that, indeed, some scenarios may lead to lack of decision on first phase, two of these curious cases are presented next. We present in a condensed table representation, and then an illustrative example. Notation $0^{(i)}$ means that agreement to proposal 0 was sent by replica i, what implies that $0^{(0)}$ and $1^{(1)}$ are proposals (others are responses).

R_0	$0^{(0)}$	$0^{(2)}$	$0^{(3)}$	$PreCommit(0) \rightarrow Commit(0)$
R_1	$1^{(1)}$	$0^{(0)}$	$0^{(3)}$	$PreCommit(0) \rightarrow Commit(0)$
R_2	$0^{(0)}$	$0^{(2)}$	$0^{(3)}$	$PreCommit(0) \rightarrow Commit(0)$
R_3	$0^{(0)}$	$0^{(3)}$	$0^{(2)}$	$PreCommit(0) \rightarrow Commit(0)$

Table 1. dBFT3-WS2: Good Case for Primary 0. No faulty node. All nodes received $f + 1 = 2$ confirmations for 0, meaning that at least one non-faulty agrees with replica 0, after $2f + 1 = 3$ responses/proposals. System commits globally at 0.

5.3. dBFT3-WSf: general $f + 1$ proposals

Note that dBFT3-WS2 already achieves $f + 1$ proposals for $f = 1$ in a $N = 4$ consensus (double proposal). Thus, at least it does not look impossible to design on practice, for $f = 2$ and beyond. More challenges will emerge though:

- extra complications on commit rules

---

6 In first experiments this appeared to be true in all cases tested, yet it's in early phases to give a definitive proof on the matter. Feeling up to now indicates that it will be nearly impossible for a Byzantine node to pass unnoticed from double responses, thus certainly being blacklisted, and the spork prevented.

7 Source code available at: https://github.com/NeoResearch/dbft3-spork-detect

R_0	$0^{(0)}$	-	-	-
R_1	$1^{(1)}$	$1^{(2)}$	$1^{(3)}$	$PreCommit(1) \to Commit(1)$
R_2	$1^{(1)}$	$1^{(2)}$	$1^{(3)}$	$PreCommit(1) \to Commit(1)$
R_3	$1^{(1)}$	$1^{(3)}$	$1^{(2)}$	$PreCommit(1) \to Commit(1)$

Table 2. dBFT3-WS2: Good Case for Primary 1. Primary 1 is faulty (or too slow). Replicas agree with primary 1 after $2f + 1 = 3$ responses/proposals for it. System commits globally at 1.

R_0	$0^{(0)}$	$1^{(1)}$	$1^{(3)}$	$PreCommit(?)$
R_1	$1^{(1)}$	$0^{(0)}$	$1^{(3)}$	$PreCommit(?)$
R_2	$0^{(0)}$	$0^{(2)}$	$1^{(1)}$	$PreCommit(0) \to Commit(0)$
R_3	$1^{(1)}$	$1^{(3)}$	$0^{(0)}$	$PreCommit(?)$

Table 3. dBFT3-WS2: Strange Case 0. For subset $\{R_0, R_1, R_3\}$, no agreement currently exists with $f + 1$ zeros (priority) or $2f + 1$ ones. No faulty node, but requires last message $0^{(2)}$ to arrive (on *PreCommit*/*Commit* from R_2), or it will change view. Replica 2 is already committed to zero, if any other replica receives its precommit/commit, it will also commit to zero. Change views could allow next primary re-propose 0 or 1. System is theoretically committed to zero already, so one should prefer 0 over 1 in next view. After change view, system would reinforce commitment to zero, thus System will $Commit(0)$. But we must be careful to not *force* a commit on one, making a *too weak* replica 0 (by voting between two ones and a zero, for example) could break system (some $Commit(0)$ and others $Commit(1)$).

R_0	$0^{(0)}$	$1^{(1)}$	$1^{(2)}$	$PreCommit(?)$
R_1	$1^{(1)}$	$0^{(0)}$	$1^{(2)}$	$PreCommit(?)$
R_2	$1^{(1)}$	$1^{(2)}$	$0^{(0)}$	$PreCommit(?)$
R_3	$1^{(1)}$	$1^{(3)}$	$0^{(0)}$	$PreCommit(?)$

Table 4. dBFT3-WS2: Strange Case 1. For any $(2f + 1 = 3)$-subset, no agreement currently exists with $f + 1$ zeros (priority) or $2f + 1$ ones. No faulty node, but requires last message $1^{(3)}$ to arrive (to $Commit(1)$), that should not be accepted by any other node, so it will change view. Change views would make next primary may re-propose 0 or 1, now re-accepting messages. But we must be careful to not *force* a commit on zero, making a *too strong* replica 0 (only valid proposal is enough without any response) could break system (some $Commit(0)$ and others $Commit(1)$ due to missing message).

R_0	$0^{(0)}$	$1^{(1)}$	$1^{(3)}$	$PreCommit(?)$
R_1	$1^{(1)}$	$0^{(0)}$	$1^{(3)}$	$PreCommit(?)$
R_2	-	-	-	-
R_3	$1^{(1)}$	$1^{(3)}$	$0^{(0)}$	$PreCommit(?)$

Table 5. dBFT3-WS2: Faulty Case R_2. For subset $\{R_0, R_1, R_3\}$, no agreement currently exists with $f + 1$ zeros (priority) or $2f + 1$ ones. Replica 2 is faulty, but other replicas cannot know if it will recover to resolve commit, so system will change view. Current system is Undecided. Change views would make next primary re-propose 0 or 1 (but 1 is much less risky). But we must be careful to not *force* a commit on zero, making a *too strong* replica 0 (only valid proposal is enough without any response) could break system (some $Commit(0)$ and others $Commit(1)$ due to missing message).

- extra edge cases to monitor
- extra messages on network

Thus, we argue if this is indeed necessary, as blacklisting mechanisms of dBFT3-WS2 already allow close monitoring of nodes, specially when consistently faulty. Typically, if multiple nodes are constantly faulty, some offchain governance (such as voting) may typically "solve this problem", while the only issue of having multiple failed nodes is extra change views (as on dBFT2).

5.4. Feature comparison

A short comparison of features in each dBFT version, as well as its fork-avoidance mechanisms.

Table 6. Comparison table for existing dBFT variants

dBFT comparison				
Name	**Phases**	**Issues Faced**	**Worst Case**	**Detailed Explanation**
dBFT	2 phases	Signature leaks happen at `prepare request` phase, allowing multiple valid (signed) blocks at same height (*spork*)	$f+1$ sporks	A non-faulty primary node is tricked by delays, relaying a first block $\#H_1$. After that, $2f+1$ replicas Change View, and a faulty primary assumes. It waits until near expiration of timeout, and "unlucky" delays trick other nodes again. This can be repeated f times (by changing f views), and sporking $f+1$ replicas.
dBFT 2.0	3 phases	No signature leakage due to unique commits, but commits may be different at each view. Recovery mechanism is necessary to put node in correct state after crashing. Possible locking on different commit/views need to be avoided using synchrony conditions.	0 sporks	Some non-faulty replica can achieve a valid commit state (with $2f+1$ responses), thus exposing its signature when others decide to change view (afterwards). Replicas can enter in a locked state situation, where successive change views commits them differently, one by one. A synchrony condition is used $f'+c' \geq f$, where c' is known committed replicas (in lower views), and f' is expected failed nodes (no communication for some time). This is used to prevent change views when a node knows consensus will be impossible in upper views.

Table 7. Comparison table for proposed dBFT variants

dBFT comparison				
Name	**Phases**	**Issues Faced**	**Worst Case**	**Detailed Explanation**
dBFT3-A	async	Based on Honey Badger [6]. Depends on guaranteed message delivery on p2p. Faulty nodes may propose empty blocks, and duplicate transaction may be proposed (wasting space). Non-existing transactions may be proposed. Bigger complexity on underlying cryptographic and algorithmic techniques.	0 sporks	Consensus is fully asynchronous, thus will depend on message delivery to eventually happen. Hash collision on Bloom Filters may prevent this from happening, also any issue with offchain governance and interests/issues of nodes in P2P network. Since every node proposes part of block, this may always be empty, or duplicated due to hash collisions. There's no guarantee that transaction exists, so block may be released while not actually valid. Cryptographic techniques are more complex, and possibly weaker than existing ones.
dBFT3-WS2 dBFT3-WSf	3*-4 phases * very likely to be so on practice (in double primary mode) 4 phases are likely necessary for general $f+1$ proposals	Multiple proposals in first view may lead to multiple valid commits. If limited by $f+1$ proposals in first view, $f+1$ commits may exist. Impossible conflicts may require a new consensus after change view, but only 12% of time with constantly faulty f (and extremely asynchronous network).	0 sporks (f resolved conflicts may exist in first view, +1 on every view)	Timestamp is usually controlled by taking *median* from replicas (safe within $2f+1$), allowing achievement of precise timing T for block timestamps. After multiple change views, this is ignored, as network may have been failed for longer periods of time (recovery/fallback mode). Up to f nodes can make proposals, including *guest nodes* from community (highly staked). Timing between proposals should be respected in honest protocol, in T/f intervals ($T = 15$ implies 7.5 seconds difference to resolve). *guest nodes* cannot change view, just help in first phase to resolve (being rewarded if so). Change View is fallback, with $+1$ node every round.

Author Contributions: The group of authors contain researchers with a variety of backgrounds, from applied Computers Scientists to specialists in Optimization, Cryptography, Game Theory and Business. The team has been formed by interaction with the open-source project NEO, which connected these members into different events that they participated together in USA, China and Brazil.

Funding: This research has been done in partnership with NeoResearch, a worldwide open-source community, with suport of NEO Foundation. Igor M. Coelho was partially supported by the CNPq.

Acknowledgments: The authors are also grateful to contributors that shared their vision on open discussions on GitHub.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BFT	Byzantine Fault Tolerance
dBFT	Delegated Byzantine Fault Tolerance
1BF	One-Block-Finality
FLP	Fischer, Lynch and Paterson
P2P	Peer-to-Peer
PBFT	Practical Byzantine Fault Tolerance

References

1. Castro, M.; Liskov, B. Practical Byzantine fault tolerance. OSDI, 1999, Vol. 99, pp. 173–186.
2. Lamport, L.; Shostak, R.; Pease, M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **1982**, *4*, 382–401.
3. Bracha, G.; Toueg, S. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* **1985**, *32*, 824–840.
4. Hongfei, D.; Zhang, E. NEO White Paper. https://docs.neo.org/docs/en-us/basic/whitepaper.html. Acessed: 2019-09-30.
5. Fischer, M.J.; Lynch, N.A.; Paterson, M.S. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* **1985**, *32*, 374–382. doi:10.1145/3149.214121.
6. Miller, A.; Xia, Y.; Croman, K.; Shi, E.; Song, D. The honey badger of BFT protocols. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016, pp. 31–42.
7. Herlihy, M.P.; Wing, J.M. Axioms for concurrent objects. Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, 1987, pp. 13–26.
8. Veronese, G.S.; Correia, M.; Bessani, A.N.; Lung, L.C. Spin one's wheels? Byzantine fault tolerance with a spinning primary. 2009 28th IEEE International Symposium on Reliable Distributed Systems. IEEE, 2009, pp. 135–144.
9. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report, Open-source, 2008.

A. Appendix: highlights

(H.1) presenting foundations of classic BFT consensus systems, clearly stating its assumptions and adversarial conditions
(H.2) explore both worlds of weakly-synchronous and fully asynchronous consensus
(H.3) argue that *guaranteed message delivery* is not trivial to ensure on a decentralized peer-to-peer network (see Subsection A.1 below). This a dependency of pure asynchronous consensus.
(H.4) by having a weak-synchrony-based consensus, it may be naturally resilient to few message losses (as many messages are already ignored if received after timeouts)
(H.5) dBFT nodes does not trust the time of others, only themselves. This incurs in very large difference in block timestamp limits (up to several minutes). In these cases, only primary is trusted.
(H.6) This strategy was good to start, but for fine-tunning strategies, it is possible to know *at least* the time of a single *non-faulty* node (even if f nodes collude with wrong/absurd times, median will still fall within interval of good nodes, considering $2f + 1$). This allow safely reducing timestamp difference limit up to seconds, and perhaps, milliseconds.
(H.7) Good information is not passed to other nodes during Change Views (dBFT and dBFT 2.0). This reduces network costs, but may incur on unexpected (and unnecessary) change views. This also drastically increases fork case possibilities (during design).
(H.8) With a strong efficiency increase on dBFT, it was possible to add an extra phase: commit phase. Now, we believe it's time to add extra information, such as received `prepares` during Change View, since it guarantees mathematically that all nodes will be fully aware of current situation, thus reducing fork risks to zero. This also allows simultaneous proposals (multiple primaries).
(H.9) This strategy should be added together with strategies that definitely indicate failure on nodes, allowing an automatic (temporary) blacklisting
(H.10) purely asynchronous consensus such as HoneyBadgerBFT look promising, but when analysed we demonstrated that they are unfeasible for practice on Neo (details on report). The worst problem is inability to deal with message losses and to guarantee that proposed block transactions actually exist. It may be good for private networks, not decentralized public ones.
(H.11) we propose dBFT 3.0 as a version of dBFT 2.0, including precision timers (using medians), and multiple simultaneous block proposals (with blacklisting and extra information passing). It will allow block times to reach **exact** 15 seconds (or less). It will also allows much more nodes to participate, including nodes from community (not voted ones, but with high NEO staking). We also propose novel interaction mechanisms for consensus with community, that may help raising funds for maintenance, and finally, a fair model for network and system fees.

A.1. Ensuring Message Delivery on Peer-to-Peer: Challenges

In order to relay messages and avoid duplication, it is usually employed techniques such as Bloom Filters and probabilistic tables. Although this strongly favor efficiency and reduces overall memory usage, there's a risk of completely denying genuine messages, due to conflicts of previously generated message hashes. The probability of this event may be very small, yet not negligible on practice if the consensus mechanism depends that all messages are eventually delivered.

One way to reduce the probability of such failures, is to have separated filters for Priority (consensus) and Non-priority messages. Since a majority of messages is of non-priority (transactions, blocks, etc), they may overload the probabilistic tables with information that may cause non-deterministic conflicts on priority messages. Having less info on priority tables (and perhaps a larger size), it is further reduced such conflict probability that yields false positives on message relay.

Dealing with adversaries that may intentionally drop messages, thus not relaying them, is even harder. To guarantee that priority messages arrive, one must enforce redistribution for *every new node* that connects to it, keeping a history of messages that may help current consensus to evolve, and dropping past messages that are not necessary anymore. Nodes should also disconnect and reconnect to others periodically, as stable bonds could lead to situations where a strong adversary is capable of controlling important nodes and routes inside peer-to-peer.