Delegated Byzantine Fault Tolerance: Technical details, challenges and perspectives

Igor M. Coelho, Vitor N. Coelho, Peter Lin, Erik Zhang

March 14, 2019
Revision: 7e86384

Byzantine tolerant systems have been designed for decades and a recent boom in Blockchain inspired solutions has been boosting studies over this topic. While the literature has been extensively contributing towards more reliability, robustness and resilience of consensus systems, few works addressed large scale industrial cases focused on State Machine Replication with one block finality. This paper, besides being a didactic material, has also the potential of contributing as a scientific reference regarding the use of Byzantine Fault Resistant agent communication in the Blockchain ecosystem.

Contents

1 Delegated Byzantine Fault Tolerance: Technical details, challenges and perspectives

This section is part of the Community Yellow Paper[1] initiative, a community-driven technical specification for Neo blockchain.

Various studies in the literature dealt with partially synchronous and fully asynchronous Byzantine Fault Tolerant systems (Hao et al. 2018; Duan, Reiter, and Zhang 2018; Miller et al. 2016), but few of them were really applied in a live Smart Contract (SC) Scenario with multiple distinct decentralized applications. It is noteworthy that append storage applications pose different challenges when compared to the needs of SC transaction persisting, which involves State Machine Replication (Schneider 1990). In addition, a second important fact to be considered is related to the finality of appending information to the ledger. Final users, merchants, and exchanges want to precisely know if their transaction was definitively processed or if it could still be reverted. Differently than most parts of previous works in the literature, NEO blockchain proposed a Consensus mechanism with **one block finality** in the **first layer** (Hongfei, Da and Zhang, Erik 2015). Besides its notorious advantages for real case applications, this characteristic imposes some additional constraints, vulnerabilities and challenges.

The goal of this technical material is to highlight the main adaptions from the classical Practical Byzantine Fault Tolerance (pBFT) to the Delegated Byzantine Fault Tolerance (dBFT) currently used in the NEO blockchain core library (see Neo Project Github). Furthermore, it describes a novel mathematical model that is able to verify specific consensus behavior by means of a discrete model which can simulate its operation in real cases. While highlighting the positive aspects of the current NEO consensus system, this document also has the goal of pointing out possible faults and future research & development directions. The latter can be achieved by a combination of NEO's requirements and novel ideas in connection with well-known studies from the literature.

The remainder of this document is organized as follows. Section 1.1 provides a brief background on the classical PBFT. Section 1.2 describes the key modification made from the literature for the achievement of NEO's dBFT. Section 1.3 details the current state-of-the-art discussions regarding NEO's dBFT, and presents didactic pseudocodes and flowcharts. Finally, Section 1.9 proposes a novel mathematical programming model based on Linear Integer Programming, which models an optimal adversary that will challenge the network and verify its limitations in worst case scenarios.

1.1 Background on Practical BFT

Practical BFT was first made possible by the work of Miguel Castro and Barbara Liskov (see Figure 1), entitled "Practical Byzantine Fault Tolerance" (Castro and Liskov 1999).

[1] See Community Yellow Paper repository

Figure 1: Turing-Prize winner Barbara Liskov on 2010. Wikipedia CC BY-SA 3.0

Given $n = 3f + 1$ replicas of a State Machine, organized as Primary and Backup nodes, the proposed algorithm guarantees *liveness* and *safety* to the network, if at most f nodes are faulty/Byzantine[2].

- Safety property ensures that all processes will execute as atomic, either executing on all nodes, or reverting as a whole. This is possible due to the deterministic nature of the process (executed on every node), which is also valid for NEO network and blockchain protocols on general.
- Liveness guarantees that network won't be stopped (unless more than f byzantine nodes), by using a mechanism called "change view", which allows Backup nodes to switch Primary node when it seems Byzantine. A timeout mechanism is used, and by doubling delays exponentially at every view, PBFT can prevent attacks from malicious network delays that cannot grow indefinitely. In the current formula, timeout happens following a left-shift operator according to the current view number, for example:
  - Considering 15 second blocks: 15 << 1 is 30s (first change view); 15 << 2 is 60s; 15 << 3 is 120s; 15 << 4 is 240s.
  - Considering 1 second blocks: 1 << 1 is 2s; 1 << 2 is 4s; 1 << 3 is 8s; 1 << 4 is 16s.

The considered network on PBFT assumes that it "may fail to deliver messages, delay them, duplicate them, or deliver them out of order." They also considered public-key cryptography to validate the identity of replicas, which is also the same for NEO dBFT. Since the algorithm does not rely on synchrony for safety, it must rely on it for liveness[3]. The resiliency of $3f + 1$ is optimal for a Byzantine Agreement (Bracha and Toueg 1985), with at most f malicious nodes.

PBFT correctness is guaranteed by having three different phases: pre-prepare, prepare and commit[4].

- On pre-prepare, primary sends a sequence number k together with message m and signed digest d. Backup i accept pre-prepare if signature is correct, k is in valid interval[5], and i has not yet accepted a pre-prepare for same k and same view.

[2] The name Byzantine refers to arbitrary behavior, and was coined by Leslie Lamport and others in the paper "The Byzantine Generals Problem"
[3] This was demonstrated by paper "Impossibility of distributed consensus with one faulty process"
[4] NEO dBFT 2.0 also consists of three phases, with a slight naming change: prepare request, prepare response, and commit
[5] A special technique avoids the exhaustion of sequence number space by faulty primary

- When pre-prepare is accepted, a prepare message is broadcast (including to primary), and a node is considered `prepared` when it receives at least $2f$ prepare messages that match its local pre-prepare, for the same view. So, at this point, for a given view, the non-faulty replicas already agree on total order for requests. As soon as $2f + 1$ non-faulty nodes are `prepared`, the network can be considered as `committed`.
- Every `committed` replica broadcasts a commit message, and as soon as node i has received $2f+1$ commit messages, node i is `committed-local`. It is guaranteed that, eventually, even with the occurrence of change views, a system with `committed-local` nodes will become `committed`.

PBFT considers that clients interact and broadcast messages directly to the primary node, then receiving independent responses from $2f + 1$ nodes in order to move forward (to the next operation). This is a similar situation for NEO blockchain, where information is spread by means of a peer-to-peer network, but in this case, the location of consensus nodes is unknown (in order to prevent direct delay attacks and denial of service). One difference is that, for PBFT, clients submit atomic and independent operations for a unique timestamp, which are processed and published independently. For NEO blockchain, consensus nodes have to group transactions into batches, called blocks, and this process may lead to the existence of thousands of valid blocks for the same height, due to different groupings (different combinations of transactions). So, in order to guarantee block finality (a single and unique block can exist in a given height), we may have to consider situations where the "client" (block proposer) is also faulty, which is not considered on PBFT.

1.2 NEO dBFT core modifications

In summary, we highlight some differences between PBFT and dBFT:

- One block finality to the end-users and seed nodes;
- Use of cryptographic signatures during different phases of the procedures in order to avoid exposure of nodes commitment to the current block;
- Ability of proposing blocks based information sharing of block headers (transactions are shared and storage in an independent syncronization mechanism);
- Avoid double exposure of block signatures by disable change views after commitment phase;
- Regeneration mechanism able to recover failed nodes both, in the local hardware and in the network P2P consensus layer.

1.3 dBFT detailed description

The dBFT consensus mechanism is a state machine, with transitions depending on a round-robin scheme (to define Primary/Backup nodes) and also depending on network messages.

1.3.1 dBFT states

dBFT states are the following:

- Initial : initial machine state
- Primary : depends on block height and view number
- Backup : true if not primary, false otherwise
- ~~RequestSent : true if block header has been proposed, false otherwise~~ (removed on dBFT 2.0 since code tracks all preparation signatures, merged as RequestSentOrReceived)
- ~~RequestReceived : true if block header has been received, false otherwise~~ (removed on dBFT 2.0 since code tracks all preparation signatures, merged as RequestSentOrReceived)
- ~~SignatureSent : true if signature has been sent, false otherwise~~ (removed on dBFT 2.0 because of extra commit phase carrying signatures)
- RequestSentOrReceived : true if a valid signature of Primary has been received, false otherwise (introduced on dBFT 2.0).
- ResponseSent : true if block header confirmation has been sent (introduced on dBFT 2.0: internal state used only for blocking node to triggering consensus OnTransaction event)
- CommitSent : true if block signature has been sent (this state was only introduced on dBFT 2.0 and replaced SignatureSent)
- BlockSent : true if block has been sent, false otherwise
- ViewChanging : true if view change mechanism has been triggered, false otherwise
- IsRecovering : true if a valid recovery payload was received and is being processed (introduced on dBFT 2.0: internal state)

The first dBFT handled these states explicitly as flags (ConsensusState enum). However, dBFT 2.0 can infer this information in a implicit manner, since it has added a track of preparations signatures and state recovery mechanisms.

1.4 Flowchart

Figure 2 presents the State Machine replicated on each consensus node (the term *replica* or *node* or *consensus node* may be considered synonyms for this subsection). The execution flow of a State Machine replica begins on the `Initial` state, for a given block height H on the blockchain. Given T as standard block time (15 seconds); v as current view number (starting from $v = 0$); $exp(j)$ is set to 2^j; `i` as consensus index; R as total number of consensus nodes. This State Machine can be represented as a Timed Automata (Alur and Dill 1994), where C represents the clock variable and operations `(C condition)?` represent timed transitions (`C:=0` resets clock). Dashed lines represent transitions that explicitly depend on a timeout behavior and were included in a different format for clarity. It is also assumed that transitions are processed *in the order* they are presented. For example:

```
(C >= 5)?
A
(C >= 7)?
B
```

This block would first wait until clock C has over 5 seconds, then process A, then check clock to meet 7 seconds, and then process B. This allows a more precise description of the actual dBFT 2.0 implementation.

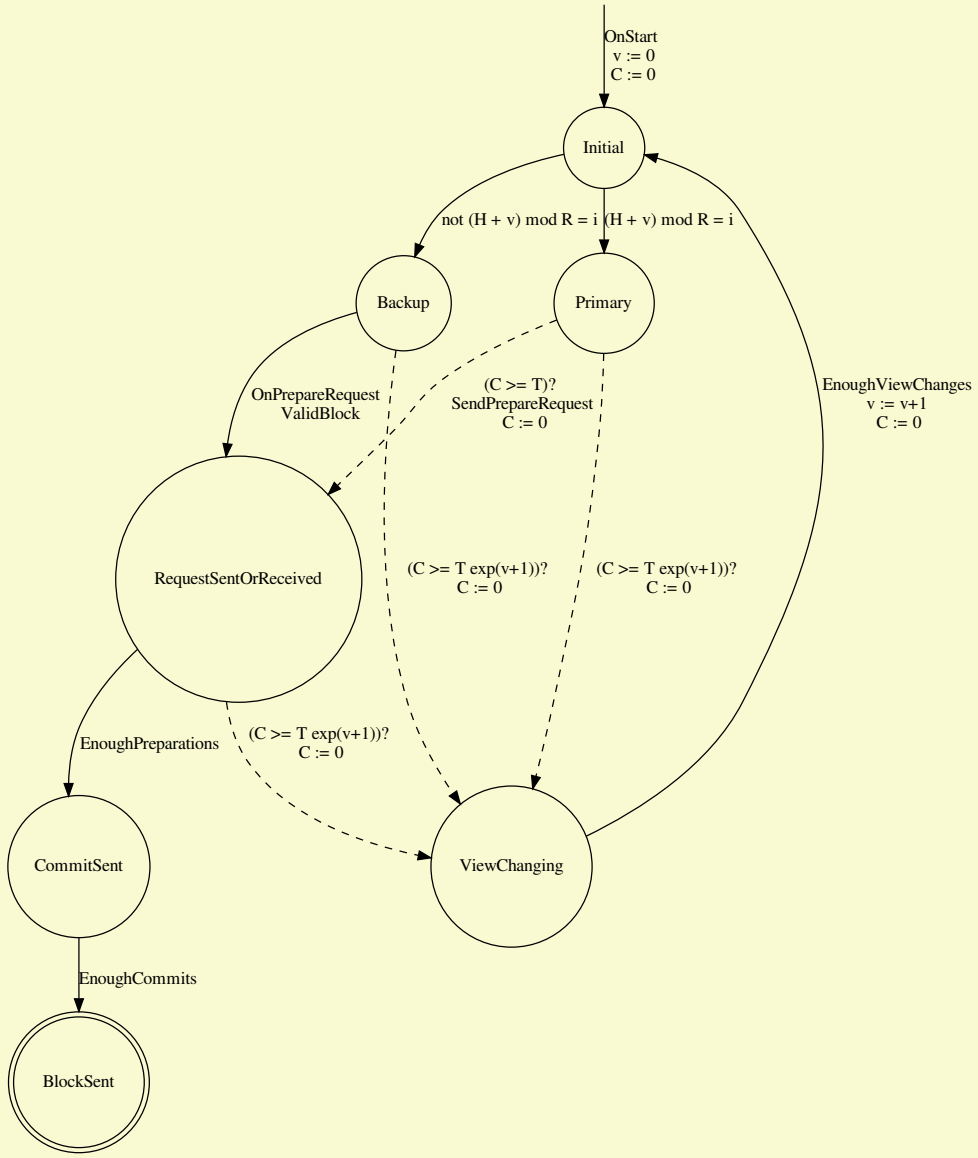

Figure 2: dBFT 2.0 State Machine for specific block height

On Figure 2, consensus node starts on `Initial` state, on view $v = 0$. Given H and v, a round-robin procedure detects if current node i is Primary: $(H + v) \mod R = i$ (it is set to backup otherwise). If node is Primary, it may proceed to `RequestSentOrReceived` after `SendPrepareRequest` action (that selects transactions and creates a new proposed block) after T seconds. If node is Backup, it needs to wait for a `OnPrepareRequest` action. After clocks expire, nodes may enter a `ViewChanging` state, what guarantees *liveness* to the network in case of failed Primary. However,

CommitSet state guarantees that no view change occurs, as the node is already *committed* to that specific block (so it won't provide signature to any other block on that height). Since this could compromise the liveness of the network, a Recovery process was proposed (see Figure 3). `EnoughPreparations`, `EnoughCommits` and `EnoughViewChanges` depend on having enough valid responses that surpass the byzantine level M (thus, respecting maximum number of faulty nodes f). `T` is currently, until version 2.0, calculated as a basin on the time that the node received last block instead of checking the timestamp in which previous header was signed.

1.5 Block finality

Block finality in the Consensus layer level imposes the following condition presented on Equation (1), which defines that there should not exist two different blocks for a given height h, in any time interval t.

$$\forall h \in \{0, 1, \cdots, t\} \Rightarrow b_t^i = b_t^j \tag{1}$$

In summary, the block finality provides that clients do not need to verify the majority of Consensus for SMR. In this sense, seed nodes can just append all blocks that possess the number of authentic signatures defined by the protocol (namely, $M = 2f + 1$). In this sense, as already described for the current NEO dBFT, the minimum number of required signatures is $2f + 1$ as defined in The Byzantine Generals Problems (Lamport, Shostak, and Pease 1982), where $f = \frac{1}{3} \times N$ is the maximum number of Byzantine nodes allowed by the network protocol.

1.6 Multiple block signature exposure

1.6.1 Detected fault on dBFT v1.0

Known Block Hash stuck fork was recently discovered in real operation of NEO blockchain, 2017.

In particular, this happens due to two components of the Blocks that are selected by each node that is a primary:

- Different sets of Transactions;
- Block Nonce.

In particular, the NEO dBFT 1.0 had a simplified implementation of the pBFT without the commit stage.

However, it was detected that under rare situations a given node could receive the desired `M` signatures necessary for persisting a Block, and then suddenly lose connection with other nodes. In this sense, the other nodes could detect a lack of communication (along with other fails between themselves) and generate a new block. Besides breaking block finality 1.5, this problem could halt the consensus node and any client that persists the block that was not adopted by the majority of CN. In addition, in a even more rare situation, x nodes with $ f + 1 < x < M $ could receive a given block while

the other nodes had a different block hash, halting the whole network until a manual decision was reached.

It is noteworthy that even in an Asynchronous Consensus without timeout mechanism this case could lead to problems if the Nonce was not yet defined as well as the transactions to be inserted inside a Block. This real incident motivated several novel insights on the consensus, which covered this "natural" issue due to network as well as added extra security in case of real Byzantine nodes.

1.6.2 Commit phase with change view blocking

Taking into account that the aforementioned fault could happen even with the commit phase, one should verify that nodes could become stuck but not double expose its signature. On the other hand, other attacks could happen if malicious nodes tried to save the signature and perform some specific sets of actions, such as storing information and not sharing it.

In this sense, the possibility that naturally came was:

- Lock view changing (currently implemented since NEO dBFT 2.0) after sending the block header signature. This means that those who are committed with that block will not sign any other proposed Block.

On the other hand, a regeneration strategy sounded compulsory to be implemented since nodes are stuck with their agreement. We defined this as the **indefatigable miners problem**, defined below:

1. The speaker is a Geological Engineering and is searching for a place to dig for Kryptonite;
2. He proposes a geographic location (coordinates to dig);
3. The majority of the team (M) agrees with the coordinates (with their partial signatures) and signs a contract to dig;
4. Time for digging: they will now dig until they really find Kryptonite (no other place will be accepted to be dig until Kryptonite is found). Kryptonite is an infinite divisible crystal, thus, as soon as one finds he will share the kryptonite so that everyone will have a piece for finishing their contract (3);
5. If one of them dies, when it resurrects it will see its previous signed agreement (3) and it will automatically start to dig again (Regeneration strategy). The other minority will suffer the same, they will be fulfilled with hidden messages saying that they should also dig.

This strategy keeps the strength of the the dBFT with the limit of a maximum number of f faulty nodes. In addition, it adds robustness with a survival/regeneration strategy.

1.7 Regeneration

The Recover/Regeneration event is designed for responding to a given failed node that lost part of the history. In addition, it also has a local backup that restores nodes in some cases of hardware failure. This local level of safety (which can be seen as a hardware faulty safety) is essential, reducing the change of specifically designed malicious attacks.

In this sense, if the node had failed and recovered its health, it automatically sends a $change\_view$ to 0, which means that that node is back and wants to hear the history from the others. Thus, it might receive a payload that provides it the ability to check the agreements of the majority and come back to real operation, helping them to sign the current block being processed.

Following these requirements, dBFT 2.0 counted with a set of diverse cases in which a node could recover its previous state, both previously known by the network or by itself. Thus, the recovery is currently encompassing:

- Replay of $ChangeView$ messages;
- Replay of Primary $PrepareRequest$ message;
- Replay of $PrepareResponse$ messages;
- Replay of $Commit$ messages.

The code can possible recover the following cases:

- Restore nodes to higher views;
- Restore nodes to a view with prepare request sent, but not enough preparations to commit;
- Restore nodes to a view with prepare request sent and enough preparations to commit, consequently, reaching `CommitSent` state;
- Share commit signatures to a node that is committed (`CommitSent` flag activated).

Figure 3 summarizes some of the current states led by the recovery mechanisms, which is currently sent by nodes that received a change view request. Recover payloads are sent by a maximum of f nodes that received the $ChangeView$ request. Nodes are currently selected based on the index of payload sender and local current view. It should be noticed that $OnStart$ events trigger a $ChangeView$ at view 0 in order to communicate to other nodes about its initial activity and its willingness to receive any Recover payload. The idea behind this is that a node that is starting late will probably find some advanced state already reached by the network.

Here, the internal state $IsRecovering$, differently than the $ResponseSent$ state, is didactically reproduced for simplifying the possible effects that a Recover message can trigger. In this sense, without loss of generality, arrows that arrive on it can be directly connected with the ones that leave it.

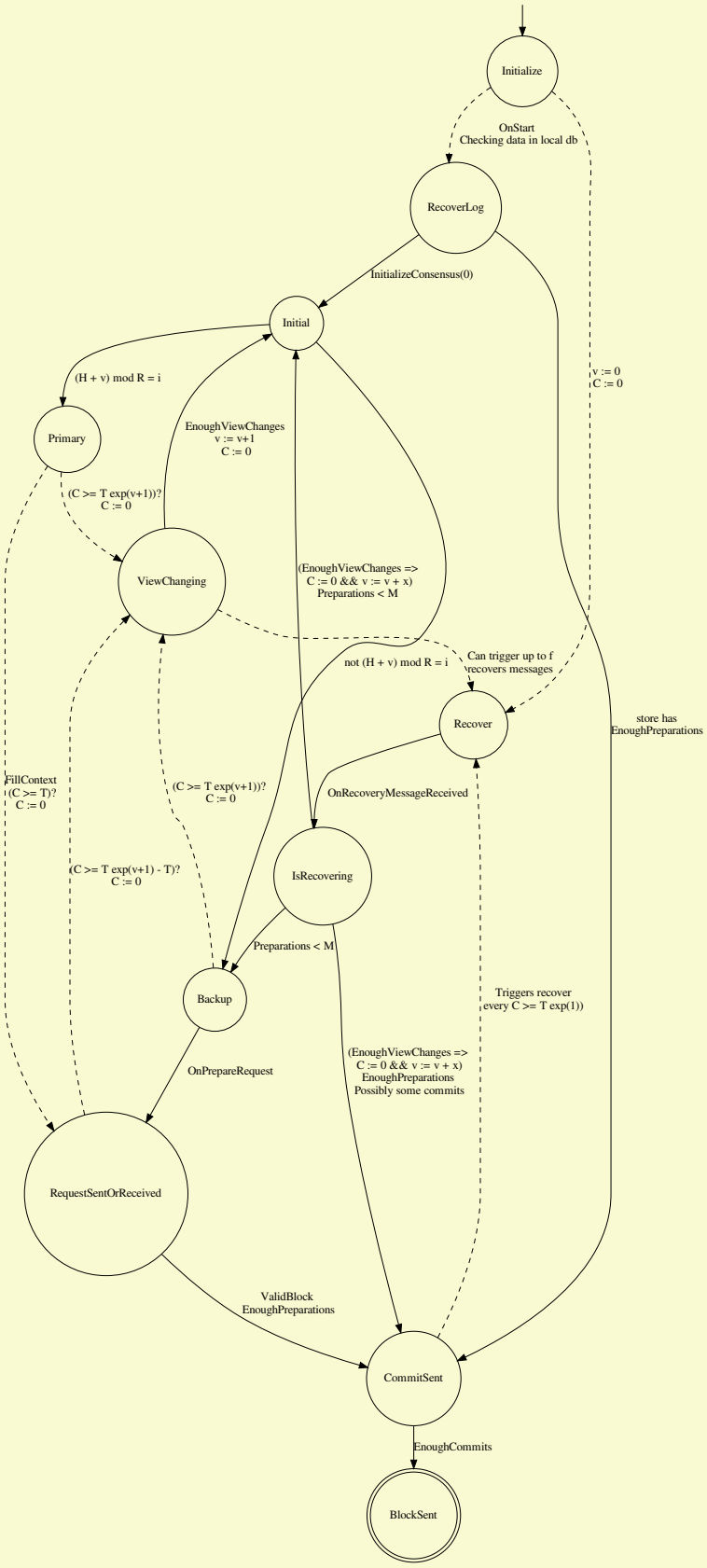

Figure 3: dBFT 2.0 State Machine with recover mechanisms

1.8 Possible faults

1.8.1 Pure network faults

Possible scenarios:

- Up to f nodes are going to delays messages;
- at maximum, f will crash both in terms of hardware fault or software problems.

1.8.2 Mixed malicious Byzantine faults

First of all, Byzantine attacks should be designed in order that nodes will never be able to prove that it was an attack. Otherwise, NEO holder would recriminate such actions and vote in favor of other nodes. Furthermore, nodes that join a given collaborative network possess an identity or stake. If anyone could detect such malicious behavior, then, that node would "automatically" (through the current voting system or an automatic mechanism that could be designed) be removed from the network.

- at maximum, f, nodes will delays messages;
- at maximum, f, nodes will send incorrect information (unlikely as it could reveal malicious behavior);
- at maximum, f, nodes will try to keep correct information for strategic occasions.

1.9 A MILP Model for Failures and Attacks on a BFT Blockchain Protocol

We present a MILP model for failures and attacks on a BFT blockchain protocol, in particular, the design is focused on the specific case of dBFT, without loss of generality for other less specialized cases.

This current model is not fully completed due to the recent updates on dBFT to version 2.0. After being finalized it will include some benchmark results modeled with A Mathematical Programming Language (AMPL), currently under development at https://github.com/NeoResearch/milp_bft_failures_attacks.

1.9.1 Mathematical model

Parameters:

$i \in R$ consensus replica i from set of replicas R. R^{BYZ} is byzantine set. R^{OK} is non-byzantine set. $R = R^{OK} \cup R^{BYZ}$, such that $R^{OK} \cap R^{BYZ} = \emptyset$.

f number of faulty/Byzantine replicas. $f = |R^{BYZ}|$.

N total number of replicas. $N = |R| = |R^{OK}| + |R^{BYZ}| = 3f + 1$.

M safety level. $M = 2f + 1$.

$b \in B$ block b from set of possible proposed blocks B (may be understood as block hash). $B = \{b_0, b_1, b_2, \cdots\}$.

$h \in H$ height h from set of possible heights H (tests may only require two or three heights). $H = \{h_0, h_1, h_2\}$. Multiple heights are considered, such that block generation can be simulated over a bigger horizon (including primary changes).

$v \in V$ view v from set of possible views V (number of views may be limited to the number of consensus nodes N). $V = \{v_0, v_1, \cdots, v_{N-1}\}$

$t \in T$ time unit t from set of discrete time units T. $T = \{t_0, t_1, t_2, \cdots\}$.

Variables:

$primary_{i,h,v}$ binary variable that indicates if Consensus Node i is primary at height h view v.

$initialized^t_{i,h,v}$ binary variable that indicates if replica $i \in R$ is at height h and view v, on time t

$SendPrepReq^t_{i,h,b,v}$ binary variable that indicates if replica $i \in R$ is sending Prepare Request message (to all nodes) at height h and view v, on time t, for proposed block b. ACTION VARIABLE MUST BE SET ONLY ONCE FOR EVERY REPLICA, HEIGHT AND BLOCK. % Nao entendi esse only once, faltou o View na descricao, nao? Caso o view seja outro ela pode ser setada denovo

$SendPrepResp^t_{i,h,b,v}$ binary variable that indicates if replica $i \in R$ is sending Prepare Response message (to all nodes) at height h and view v, on time t, for proposed block b. ACTION VARIABLE MUST BE SET ONLY ONCE FOR EVERY REPLICA, HEIGHT AND BLOCK.

$RecvPrepReq^t_{i,j,h,b,v}$ binary variable that indicates if replica $i \in R$ received a Prepare Request message from replica j at height h and view v, on time t, for proposed block b. ACTION VARIABLE MUST BE SET ONLY ONCE FOR EVERY REPLICA, HEIGHT AND BLOCK.

$RecvPrepResp^t_{i,j,h,b,v}$ binary variable that indicates if replica $i \in R$ received a Prepare Response message from replica j at height h and view v, on time t, for proposed block b. ACTION VARIABLE MUST BE SET ONLY ONCE FOR EVERY REPLICA, HEIGHT AND BLOCK.

$BlockRelay^t_{i,h,b}$ binary variable that indicates if replica i has relayed block b at height h, on time t. ACTION VARIABLE MUST BE SET ONLY ONCE FOR EVERY REPLICA, HEIGHT AND BLOCK.

$RecvBlkPersist^t_{i,j,h,b}$ binary variable that indicates if replica $i \in R$ received a Block Relay message from replica j at height h on time t, for proposed block b. ACTION VARIABLE MUST BE SET ONLY ONCE FOR EVERY REPLICA, HEIGHT AND BLOCK.

$sentPrepReq^t_{i,h,b,v}$ binary variable that indicates if replica $i \in R$ has sent (in past) to all replicas a Prepare Request message at height h and view v, on time t, for proposed block b. Once set to ONE this is carried forever as ONE.

$sentPrepResp^t_{i,h,b,v}$ binary variable that indicates if replica $i \in R$ has sent (in past) to all replicas a Prepare Response message at height h and view v, on time t, for proposed block b. Once set to ONE this is carried forever as ONE.

$recvdPrepReq^t_{i,j,h,b,v}$ binary variable that indicates if replica $i \in R$ has received (in past) from replica j a Prepare Request message at height h and view v, on time t, for proposed block b. Once set to ONE this is carried forever as ONE.

$recvdPrepResp^{t}_{i,j,h,b,v}$ binary variable that indicates if replica $i \in R$ has received (in past) from replica j a Prepare Response message at height h and view v, on time t, for proposed block b. Once set to ONE this is carried forever as ONE.

$sentBlkPersist^{t}_{i,h,b}$ binary variable that indicates if replica $i \in R$ has sent (in past) to all replicas a Block Relay message at height h, on time t, for proposed block b. Once set to ONE this is carried forever as ONE. % Nao se assumi que um byzantine poderia dar dois relays diferentes em views distintos?

$recvdBlkPersist^{t}_{i,j,h,b}$ binary variable that indicates if replica $i \in R$ has received (in past) from replica j a Block Relay message at height h, on time t, for proposed block b. Once set to ONE this is carried forever as ONE.

$blockRelayed_{b}$ binary variable that indicates if block b was relayed (on any time, height or view).

Objective function:

$$maximize \sum_{b \in B} blockRelayed_{b} \tag{2}$$

The adversary can control f replicas, but the other M replicas must follow dBFT algorithm. The adversary can choose any delay for any message (up to maximum simulation time $|T|$). If it wants to shutdown the whole network, no blocks will be ever produced and objective will be zero (minimum possible). So, adversary will try to maximize blocks produced by manipulating delays in a clever way. As described by Equation (2), objective function is bounded to $[0, |B|]$.

Constraints:

Initialization constraints

$$initialized^{t_0}_{i,h_0,v_0} = 1 \qquad \forall i \in R^{OK} \tag{3}$$

$$initialized^{t_0}_{i,h,v} = 0 \qquad \forall i \in R^{OK}, h \in H \backslash \{h_0\}, v \in V \backslash \{v_0\} \tag{4}$$

$$\sum_{v \in V} initialized^{t}_{i,h,v} = 1 \qquad \forall i \in R, t \in T \backslash \{t_0\}, h \in H \tag{5}$$

$$\sum_{h \in H} initialized^{t}_{i,h,v} = 1 \qquad \forall i \in R, t \in T \backslash \{t_0\}, v \in V \tag{6}$$

Time zero constraints:

$$SendPrepReq^{t_0}_{i,h,b,v} = 0 \qquad \forall i \in R, \forall h, b, v \tag{7}$$
$$sentPrReq^{t_0}_{i,h,b,v} = 0 \qquad \forall h, b, i, v \tag{8}$$
$$RecvPrepReq^{t_0}_{i,j,h,b,v} = 0 \qquad \forall i, j \in R, \forall h, b, v \tag{9}$$
$$recvdPrReq^{t_0}_{i,j,h,b,v} = 0 \qquad \forall j, h, b, i, v \tag{10}$$
$$SendPrepResp^{t_0}_{i,h,b,v} = 0 \qquad \forall i \in R, \forall h, b, v \tag{11}$$
$$sentPrResp^{t_0}_{i,h,b,v} = 0 \qquad \forall h, b, i, v \tag{12}$$
$$RecvPrepResp^{t_0}_{i,j,h,b,v} = 0 \qquad \forall i, j \in R, \forall h, b, v \tag{13}$$
$$recvdPrResp^{t_0}_{i,j,h,b,v} = 0 \qquad \forall j, h, b, i, v \tag{14}$$
$$BlockRelay^{t_0}_{i,h,b} = 0 \qquad \forall i \in R, \forall h, b \tag{15}$$
$$sentBlkPersist^{t_0}_{i,h,b} = 0 \qquad \forall i \in R, \forall h, b \tag{16}$$
$$RecvBlkPersist^{t_0}_{i,j,h,b} = 0 \qquad \forall i, j \in R, \forall h, b \tag{17}$$
$$recvdBlkPersist^{t_0}_{i,j,h,b} = 0 \qquad \forall i, j \in R, \forall h, b \tag{18}$$
$$\tag{19}$$

Prepare request constraints:

$$SendPrepReq^{t}_{i,h,b,v} \leq initialized^{t}_{i,h,v} \qquad \forall i, h, b, v, t \tag{20}$$
$$SendPrepReq^{t}_{i,h,b,v} \leq primary_{i,h,v} \qquad \forall i, h, b, v, t \tag{21}$$
$$sentPrReq^{t}_{i,h,b,v} = sentPrReq^{t-1}_{i,h,b,v} + SendPrepReq^{t-1}_{i,h,b,v} \qquad \forall h, b, i, v, t \in T \backslash \{t_0\} \tag{22}$$
$$RecvPrReq^{t}_{i,j,h,b,v} \leq sentPrReq^{t}_{j,h,b,v} \qquad \forall h, b, i \neq j, v, t \tag{23}$$
$$RecvPrReq^{t}_{i,i,h,b,v} = SendPrepReq^{t}_{i,h,b,v} \qquad \forall h, b, i, v, t \tag{24}$$
$$recvdPrReq^{t}_{i,j,h,b,v} = recvdPrReq^{t-1}_{i,j,h,b,v} + RecvPrReq^{t-1}_{i,j,h,b,v} \qquad \forall h, b, i, j, v, t \in T \backslash \{t_0\} \tag{25}$$

Prepare response constraints:

$$SendPrepResp^{t}_{i,h,b,v} \leq initialized^{t}_{i,h,v} \qquad \forall i, h, b, v, t \tag{26}$$
$$SendPrepResp^{t}_{i,h,b,v} \geq \frac{1}{N} \sum_{j \in R} recvdPrReq^{t-1}_{i,j,h,b,v} \qquad \forall i \in R^{OK}, h, b, v, t \tag{27}$$
$$SendPrepResp^{t}_{i,h,b,v} \leq \sum_{j \in R} recvdPrReq^{t-1}_{i,j,h,b,v} \qquad \forall i \in R, h, b, v, t \tag{28}$$
$$sentPrResp^{t}_{i,h,b,v} = sentPrResp^{t-1}_{i,h,b,v} + SendPrepResp^{t-1}_{i,h,b,v} \qquad \forall h, b, i, v, t \in T \backslash \{t_0\} \tag{29}$$
$$RecvPrResp^{t}_{i,j,h,b,v} \leq sentPrResp^{t}_{j,h,b,v} \qquad \forall h, b, i \neq j, v, t \tag{30}$$
$$RecvPrResp^{t}_{i,i,h,b,v} = SendPrepResp^{t}_{i,h,b,v} \qquad \forall h, b, i, v, t \tag{31}$$
$$recvdPrResp^{t}_{i,j,h,b,v} = recvdPrResp^{t-1}_{i,j,h,b,v} + RecvPrResp^{t-1}_{i,j,h,b,v} \qquad \forall h, b, i, j, v, t \in T \backslash \{t_0\} \tag{32}$$

Block persist constraints:

$$sentBlkPersist^{t}_{i,h,b} = sentBlkPersist^{t-1}_{i,h,b} + BlockRelay^{t-1}_{i,h,b} \qquad \forall i \in R, h, b, t \tag{33}$$

$$RecvBlkPersist^{t}_{i,j,h,b} \leq sentBlkPersist^{t}_{j,h,b} \qquad \forall h, b, i \neq j, v, t \tag{34}$$

$$RecvBlkPersist^{t}_{i,i,h,b} = BlockRelay^{t}_{i,h,b} \qquad \forall h, b, i, t \tag{35}$$

$$recvdBlkPersist^{t}_{i,j,h,b} = recvdBlkPersist^{t-1}_{i,j,h,b} + RecvBlkPersist^{t-1}_{i,j,h,b} \qquad \forall h, b, i, j, t \in T \backslash \{t_0\} \tag{36}$$

Block relay constraints:

$$\sum_{t \in T} BlockRelay^{t}_{i,h,b} \leq 1 \qquad \forall i \in R, \forall h, b \tag{37}$$

$$blockRelayed_b \geq \frac{1}{N|H|} \sum_{t \in T} \sum_{i \in R} \sum_{h \in H} BlockRelay^{t}_{i,h,b} \qquad \forall b \in B \tag{38}$$

$$BlockRelay^{t}_{i,h,b} \leq \frac{1}{M} \sum_{j \in R} recvdPrResp^{t-1}_{i,j,h,b,v} + \sum_{j \in R} recvdBlkPersist^{t}_{i,j,h,b} \qquad \forall i \in R, h, b, v, t \tag{39}$$

1.9.2 Example

Fixed values presented in bold.

$initialized^{t}_{i,h,v}$, for $i \in R^{OK}, h = 0, v = 0$:

i=0	**1**	1	1	1	1	...
t	0	1	2	3	4	...

$primary_{i,h,v}$, $h = 0$:

i=0	**1**	**0**	**0**	...
i=1	**0**	**1**	**0**	...
i=2	**0**	**0**	**1**	...
v	0	1	2	...

$primary_{i,h,v}$, $h = 1$:

i=0	**0**	**1**	**0**	...
i=1	**0**	**0**	**1**	...
i=2	**0**	**0**	**0**	...
v	0	1	2	...

$SendPrepReq^{t}_{i,h,b,v}$, for $i = 0, h = 0, b = 0, v = 0$:

SendPrepReq(i=0)	**0**	0	1	0	0	0	0	0	...
t	0	1	2	3	4	5	6	7	...

$sentPrepReq^{t}_{i,h,b,v}$, i=0, $h, b, v = 0$:

(i=0)	**0**	0	0	1	1	1	1	1	...
t	0	1	2	3	4	5	6	7	...

$recvdPrepReq^{t}_{i,j,h,b,v}$, for i=0,j=0, $h, b, v = 0$:

-	**0**	0	0	1	1	1	1	1	...
t	0	1	2	3	4	5	6	7	...

$recvdPrepReq^{t}_{i,j,h,b,v}$, i=0,j=1, $h, b, v = 0$:

-	**0**	0	0	1	1	1	1	1	...
t	0	1	2	3	4	5	6	7	...

1.10 Acknowledgements

The key ideas and development behind dBFT 2.0 were mainly guided by Erik Zhang, in open GitHub discussions, and also valuable contributions from community members who took their precious time to discuss BFT ideas and to propose improvements. We are also grateful to developers that managed to turn this into reality, by constantly solving challenging problems. For this, a special thanks to those who helped this become reality: jsolman, shargon, longfei, tog, edge, and many others.[6]

Alur, Rajeev, and David Dill. 1994. "A Theory of Timed Automata." *Theoretical Computer Science* 126: 183–235. https://www.cis.upenn.edu/~alur/TCS94.pdf.

Bracha, Gabriel, and Sam Toueg. 1985. "Asynchronous Consensus and Broadcast Protocols." *J. ACM* 32 (4): 824–40. https://doi.org/10.1145/4221.214134.

Castro, Miguel, and Barbara Liskov. 1999. "Practical Byzantine Fault Tolerance." In *OSDI*, 99:173–86.

Duan, Sisi, Michael K. Reiter, and Haibin Zhang. 2018. "BEAT: Asynchronous Bft Made Practical." In *Proceedings of the 2018 Acm Sigsac Conference on Computer and Communications Security*, 2028–41. CCS '18. New York, NY, USA: ACM. https://doi.org/10.1145/3243734.3243812.

Hao, X., L. Yu, L. Zhiqiang, L. Zhen, and G. Dawu. 2018. "Dynamic Practical Byzantine Fault Tolerance." In *2018 Ieee Conference on Communications and Network Security (Cns)*, 1–8. https://doi.org/10.1109/CNS.2018.8433150.

Hongfei, Da and Zhang, Erik. 2015. "NEO: A Distributed Network for the Smart Economy." https://github.com/neo-project/docs/blob/master/en-us/whitepaper.md.

Lamport, Leslie, Robert Shostak, and Marshall Pease. 1982. "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4 (3): 382–401.

Miller, Andrew, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. "The Honey Badger of Bft Protocols." In *Proceedings of the 2016 Acm Sigsac Conference on Computer and Communications Security*, 31–42. ACM.

Schneider, Fred B. 1990. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." *ACM Comput. Surv.* 22 (4): 299–319. https://doi.org/10.1145/98163.98167.

[6] sorry if we forget important names here, this is an open document and we will fix this as soon as possible :)