dBFT 3.0 - Specification

Key features

dBFT 3.0 is a proposed extension for dBFT 2.0, which has the following properties as its core:

- Double Speakers during the `Prepare-Request` phase;
- Additional `Pre-Commit` phase;
- A speed-up mechanism for skiping `Pre-Commit` conditions in some specific scenarios;
- A timing strategy that guides `Priority Primary` and `Fallback Primary` to send `Prepare-Request` at different timeouts;
- An improved Round-Robin strategy that makes the `Fallback Primary` to vary as an uniform permutation;
- A compability mechanism that turns dBFT 3.0 into dBFT 2.0 if `View` is increased.

Additional optional directions for safety and robustness are also provided, which are defined as dBFT 2.0+ and dBFT 3.0+.

Guidelines for implementation

In this document, we present the following:

- A detailed description of all phases of dBFT 3.0;
- A theoretical perspective on the TPS improvement by following the devised strategy of dBFT 3.0;
- dBFT 3.0 sporks safety verification done throughout a Mixed-Integer Programming Model (MILP) created based on dBFT 1.0 and dBFT 2.0.

Description

Firstly, we define the concepts of *Priority* and *Fallback* Primaries [^1]:

- **Priority Primary**, namely `Primary-P1`, can be seen as the same `Primary` of dBFT 2.0, with the same timeout strategy.

- **Fallback Primary**, namely `Primary-P2`, follows the same logic with a timeout slighly shifted from `Primary-P1`.

[^1]: The term *primary* and *speaker candidate* are considered to be the same. *Primary* is opposed to *Backup*, and *Priority* is opposed to *Fallback*.

Prepare-Requests

- `Prepare-Request` (on `View` 0) for the next block is sent when:
  - `Primary-P1` timeout at `T` and relays `Prepare-Request-P1` (interval `t(h0)+T`);
  - `Primary-P2` timeout at `4/3*T` and relays `Prepare-Request-P2` (interval `t(h0)+4/3*T`), as should be noticed with a small shift, `1/3*T`, expected to be delayed from `Primary-P1`.

Prepare-Response

- `Prepare-Response` payloads are sent in the following conditions:
  - `Prepare-Response-P1` if node receives `Prepare-Request-P1`;
  - `Prepare-Response-P2` if node receives `Prepare-Request-P2`.

Note that if you are the `Primary-P1` you automatically receive `Prepare-Response-P1` if you send `Prepare-Request-P1`, which is analoguous for `Primary-P2`. This same pattern occurs on dBFT 2.0 because the `Prepare-Request` carries `Preparation` hash. Thus, we name `Preparations` as a set of hashes that comes from `Prepare-Request` (from Primary) and `Prepare-Response` (from Backups). On the same reasoning, when a node receives a `Prepare-Request` it also receives the respective `Prepare-Response`, from `P1` or `P2`.

An honest node should not send `Prepare-Response` to both `P1` and `P2`, which is called a **Type-II Attack**. On the other hand, we define **Type-I Attack** as a situation where node dies (intentionally or non-intentionally).

Pre-commit

- Nodes send a `Pre-Commit` payload when entering the following conditions:
  - `Pre-Commit-P1` is sent when a node has `f+1` *preparations* to `Primary-P1`, among `2f+1` *preparation* payloads already received;
  - `Pre-Commit-P2` if node receives `2f+1` *preparations* for `Primary-P2`.

An honest node should not send `Pre-Commit` to both `P1` and `P2` (**Type-II Attack**).

Commit

- Nodes can enter on `Commit` following two conditions:
  - `Commit-P1` if node receives `2f+1` `Pre-Commit-P1`;
  - `Commit-P2` if node receives `2f+1` `Pre-Commit-P2`.

An honest node should not send `Commit` to both `P1` and `P2` (**Type-II Attack**).

Speed mechanism (fast path)

- Specifically, `Commit-P1` can can emmitted without waiting for a `Pre-Commit-P1` agreement (a minimum number of `f+1` `Pre-Commit-P1` among, at least, `f+1 Pre-Commit`);
    - Any node can directly send `Commit-P1` if it has received `2f+1` *Preparations* from `Primary-P1`.

dBFT 3.0 compatibility conditions

In order to simplify the design and avoid additional unnecessary changes, the following properties are hold on dBFT 3.0:

- If `View 0` leads to a scenario where nodes clock expires (`2^1*T` seconds until timeout), and the system converges to `View 1`, there will be no `Fallback Primary` and the algorithm flow will keep the same as dBFT 2.0;
- As aforementioned, nodes are forbidden to reply with both `Prepare-Response-P1` and `Prepare-Response-P2` at `View 0` (**Attack Type II**);
- As aforementioned, nodes are forbidden to reply with both `Prepare-Commit-P1` and `Prepare-Commit-P2` at `View 0` (**Attack Type II**);
- As aforementioned, nodes are forbidden to reply with both `Commit-P1` and `Commit-P2` at `View 0` (**Attack Type II**);
- If committed on `View 0` we still hold the same synchrony condition of dBFT 2.0:
    - Avoid locking messages if `(c' + f' > f)`, where `c'` are the number of known nodes committed and `f'` are the known nodes failed. Currently, this condition can be seen here https://github.com/neo-project/neo/blob/e91abdcd476a6cf112d87a6998d9177679011ea0/src/neo/Consensus/ConsensusContext.cs#L66-L72.

Throughput and Round-Robin Strategies

The Consensus mechanism is one of the fundamental pieces of a blockchain, since it dictates the pace where blocks are produced, in other words, the *rhythm* of the blockchain system. Block production depends on a balanced timing between peer-to-peer communication capabilities, and also transaction execution/persistance performance.

Throughput Assumptions

At this point, we need some assumptions in order to dig into block production capabilities of dBFT 3.0. We consider:

- Block capability is limited to `B_max` transactions
- Block time target is `T` seconds
- Mempool capability is *at least* `Memcap = B_max * T` transactions
- Transactions are uniformly distributed within `T` time interval
- Maximum Transactions-Per-Second (TPS) is calculated as `Memcap / T = B_max`

Specific throughput values strongly depend on implementation, underlying hardware, so as computational limits regarding transaction verification and block persistance. These will not limit TPS, as long as the processing time `t_exec` seconds for each transaction is *at most* `T / B_max`.

Some examples for `T=10` and `T=15`:

T	B_max	Memcap	Max TPS	t_exec
15	2000	30000	2000	0.0075
15	10000	150000	10000	0.0015
10	2000	20000	2000	0.005
10	10000	100000	10000	0.001

Round-Robin Strategies for Type-I Attacks

On both dBFT 1.0 and 2.0, the speaker `i` is selected by a round-robin formula: `i = (H+v) % (3f+1)`. Due to faulty nodes (Type-I Attack), throughput is affected by a factor of *at most* `f / (3f+1) ~ 33.33%` (for a very large `f`). We call the *first view probability* `1-f/(3f+1)`, the ratio of transactions being produced in first view, without any delay:

f	f/(3f+1)	1-f/(3f+1)
1	25%	75%
2	28%	72%
3	30%	70%
...	...	...
100	33%	66%

Round-Robin strategies for double speakers

In hand of double speakers, we have the chance of making Type-I attacks to cause the least damage as possible. We must first ensure that nodes `i` and `j` are distinct, and equally balanced (among a full round-robin cycle).

Strategy 1 is a direct extension from dBFT 2.0 round-robin: `i = (H+v)%(3f+1)` and `j = (H+1+v)%(3f+1)`, in other words, `j=(i+1)%(3f+1)`. For `f=1`, this means that *fallback TPS ratio* `1 - (f-1)/(3f+1)` will equal 100% (perfect first view TPS). For `f=2`, ratio is 85%, although better than single speaker model, still a degradation from `f=1` case.

For this reason, we discuss a **Strategy 2**, consisting of a much longer (quadratic) round-robin cycle that prevents sequential failures. This is less simple, but highly effective: `i = H % (3f+1)` and `j = ((H+1) % 3f) < i ? ((H+1) % 3f) : ((H+1) % 3f)+1`, in other words, *selecting first in* `N` *and second from remaining* `N-1` (avoiding collisions). The number of combinations (`Cycle Size`), for **Strategy 2**, is the formula of a simple permutation `N!/(N-f)!`.

Considering scenarios with `f=2`, performance drops only to 97% with **Strategy 2**. Performance degradation is quite slower in this case, with `1-1/(3f*(3f+1))`, and although it keeps growing for larger value of `f`, on practice, this approximates throughput to the perfect scenarios.

Max TPS	TPS with Strategy 1	TPS with Strategy 2
2000	1700	1940

Max TPS	TPS with Strategy 1	TPS with Strategy 2
10000	8500	9700

Round-Robin strategies: examples

Strategy 1:

- `N` = 3f+1 = 4, `Faulty node` = (n0), `Cycle Size` = 4

Round	H	i	j	ok
1	100	0	1	1
2	101	1	2	1
3	102	2	3	1
4	103	3	0	1

- `N` = 3f+1 = 7, `Faulty nodes` = (n0,n1), `Cycle Size` = 7

Round	H	i	j	ok
1	98	0	1	0
2	99	1	2	1
3	100	2	3	1
4	101	3	4	1
5	102	4	5	1
6	103	5	6	1
7	104	6	0	1

Strategy 2:

- `N` = 3f+1 = 4, `Faulty node` = (n0), `Cycle Size` = 4*3 = 12

Round	H	i	j	ok
1	100	0	3	1
2	101	1	0	1
3	102	2	1	1
4	103	3	2	1
5	104	0	1	1
6	105	1	2	1
7	106	2	3	1

Round	H	i	j	ok
8	107	3	0	1
9	108	0	2	1
10	109	1	3	1
11	110	2	0	1
12	111	3	1	1

- `N` = 3f+1 = 7, `Faulty nodes` = (n0,n1), `Cycle Size` = 7*6 = 42

Round	H	i	j	ok
1	98	0	4	1
2	99	1	5	1
3	100	2	6	1
4	101	3	0	1
5	102	4	1	1
6	103	5	2	1
7	104	6	3	1
8	105	0	5	1
...	...	...	...	...
22	119	0	1	0
...	...	...	...	...
36	133	0	3	1
37	134	1	4	1
38	135	2	5	1
39	136	3	6	1
40	137	4	0	1
41	138	5	1	1
42	139	6	2	1

Safety verification of dBFT 3.0

In order to safety verify dBFT 3.0 we assumed a mathematical formulation that has been previously verified for dBFT 1.0 and dBFT 2.0. It should be noticed that the same assumptions have been generalized for dBFT 3.0 and *it does not ensure* that errors and failures do not exist. On the other hand, it provides a strong mathematical background that demonstrates some assumptions using state-of-the-art concepts of optimization.

The creation of the Mixed-Integer Linear Programming (MILP) model

Some of the model's properties should be described before going deep into its use:

- This model discretize consensus decision in time, allowing consensus to perform any task with an arbitrary limited delay.
- In addition, the model goes even further than currently NEO network scenario because it considers true byzantine nodes, which are not reality of NEO current set of Validators, because what currently occurs are just system/delays failures, which are different than a true byzantine node operating without constraints:
    - In this sense, a set of nodes `f` is allowed to do a variety of behaviors.

Experimental results for dBFT 1.0

The model for dBFT 1.0 can be found in https://github.com/NeoResearch/milp_bft_failures_attacks/tree/master/dbft1.0, implemented following the classical AMPL Language.

- In summary, the resolution of this model proofs that sporks can be obtained;
- No liveness checks are done since the failure of sporks was considered to be enough for avoiding the use of dBFT 1.0.

For simplicity, no deep discussions on dBFT 1.0 are presented here.

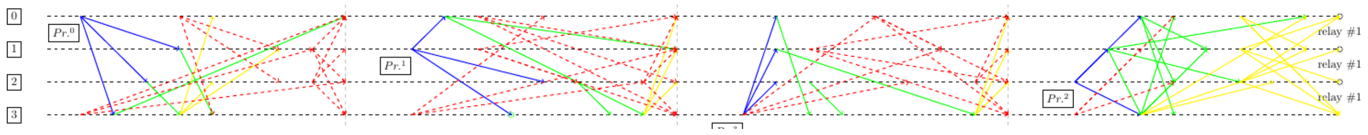
Experimental results for dBFT 2.0

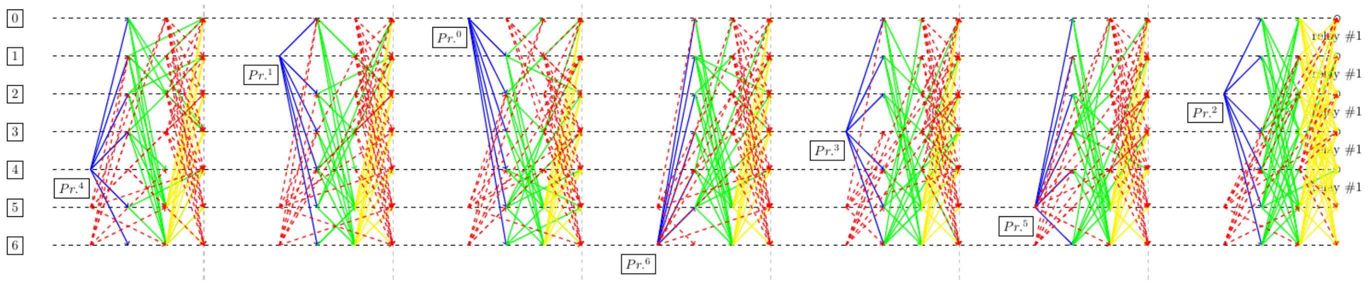
The model for dBFT 2.0 can be found in https://github.com/NeoResearch/milp_bft_failures_attacks/tree/master/dbft2.0. We suggest a carefull attention on the most recent model implemented with the Python-Mip library, file `dbft2.0_Byz_Liveness.py`.

Why is it important to describe dBFT 2.0 results in detail? As defined, in case of a `Change View` (after `View 0`), dBFT 3.0 behavior does not include `Primary-P2`, thus, behaving like dBFT 2.0.

According to the obtained experimental results. The following properties hold:

1. No sporks were obtained for: `N=4`, `N=7` and `N=10`;





- These figures show how maximizing number of blocks, number of messages and change views can not force the generation of an additional block. It is infeasible to obtain more than a single block (`PrepareRequest` depicted with blues lines, `PrepareResponse` in green, `Commit` in Yellow and `ChangeView` in Red).

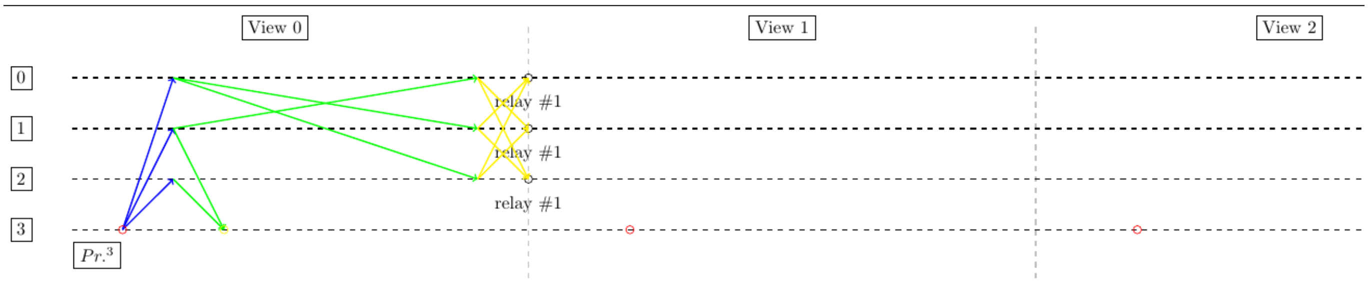
2. Liveness holds if we ensure that `Commits` and `Preparations` are going to arrive. However, this is not true in practice (but we didatically play with these constraints);
    - If we force the reception of blocks we need to use the following constraints:

```
for (i, j, v) in product(R_OK, R_OK, V):
  if i != j:
      m += (
          xsum(RecvPrepReq[t, i, j, v] for t in T - {1})
          >= xsum(SendPrepReq[t, j, v] for t in T - {1}),
          "prepReqReceivedNonByz(%s,%s,%s)" % (i, j, v),
      )
      m += (
          xsum(RecvPrepResp[t, i, j, v] for t in T - {1})
          >= xsum(SendPrepRes[t, j, v] for t in T - {1}),
          "prepRespReceivedNonByz(%s,%s,%s)" % (i, j, v),
      )
      m += (
          xsum(RecvCommit[t, i, j, v] for t in T - {1})
          >= xsum(SendCommit[t, j, v] for t in T - {1}),
          "commitReceivedNonByz(%s,%s,%s)" % (i, j, v),
      )
  # In particular, when only CV is forced, and number of change views is minimized, commits are relayed and lost.
  # On the other hand, enabling it and commits together, model can only find N rounds as minimum
      m += (
          xsum(RecvCV[t, i, j, v] for t in T - {1})
          >= xsum(SendCV[t, j, v] for t in T - {1}),
          "cvReceivedNonByz(%s,%s,%s)" % (i, j, v),
      )
```

- On the other hand, liveness holds when we penalize `View-Changes` exponentially.
    - By considering an exponential penalization on `View-Changes`, the mathematical model could always find an optimum solution with, at maximum, 1 block.
        - In the following example totalBlockRelayed has a penalization of `99` and each view is penalized with `10^v`, which implies in a total penalization of (`10`, `110`, `11110` and

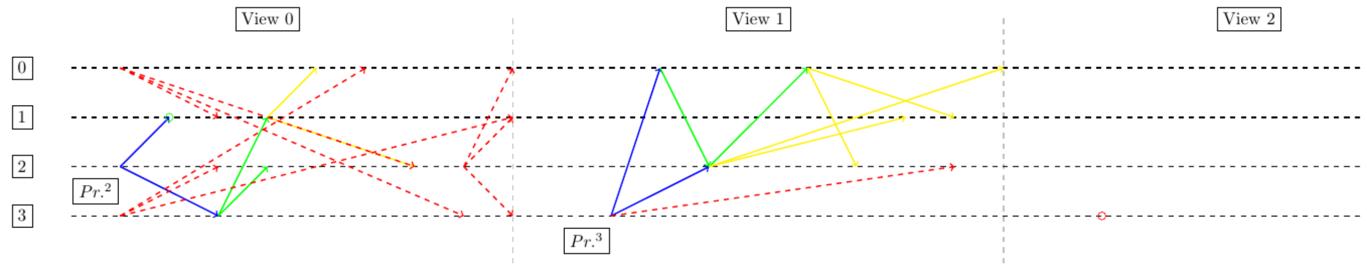
11110), which makes the model to produce a block instead of delaying further only by ensuring commits to arrive.

```
 # Exponentially penalize extra view (similar to what time does to an
attacker that tries to delay messages)
 m.objective = minimize(totalBlockRelayed*99 + xsum(Primary[i, v]*10**v
for (i, v) in product(R, V)));
```



- By changing the blocks weight to 101 we would enable a liveness lock on the system.

```
  m.objective = minimize(totalBlockRelayed*101 + xsum(Primary[i, v]*10**v
for (i, v) in product(R, V)));
```



This image is a typical case where the syncrony condition is needed, where node 1 commits at `View 0` and the other 3 nodes moves to `View 1`, however, the byzantine `node 4` looks to be dead and no progress is made.

In summary, the mathematical model enables us to analyse the model throughout a modification on mathematical formulas, empowering developers with automatic tools for testing features while basic safety checks needs to hold. Following this reasoning, the proposal for dBFT 3.0 needs to hold, at least, for the same set of conditions that ensure robustness, sporks proof-bullets and liveness of dBFT 2.0.

On the other hand, proofs throughout this mathematical model are not error free and there is no warranty that all possibilities have been covered.

Experimental results for dBFT 3.0

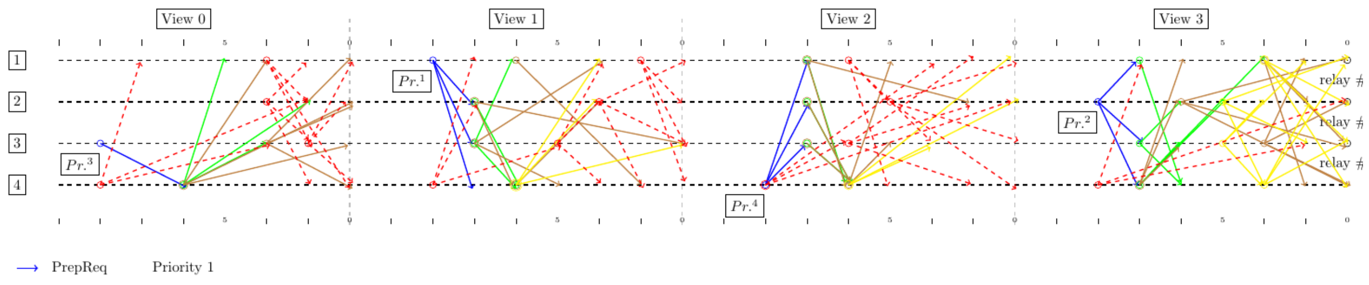
The model for dBFT 3.0 can be found in https://github.com/NeoResearch/milp_bft_failures_attacks/tree/master/dbft3.0. A python implementation has been verified against a classical AMPL. Both of them are available at the repository, but the `dbft3.0_2P.py` is the one with up-to-date features.

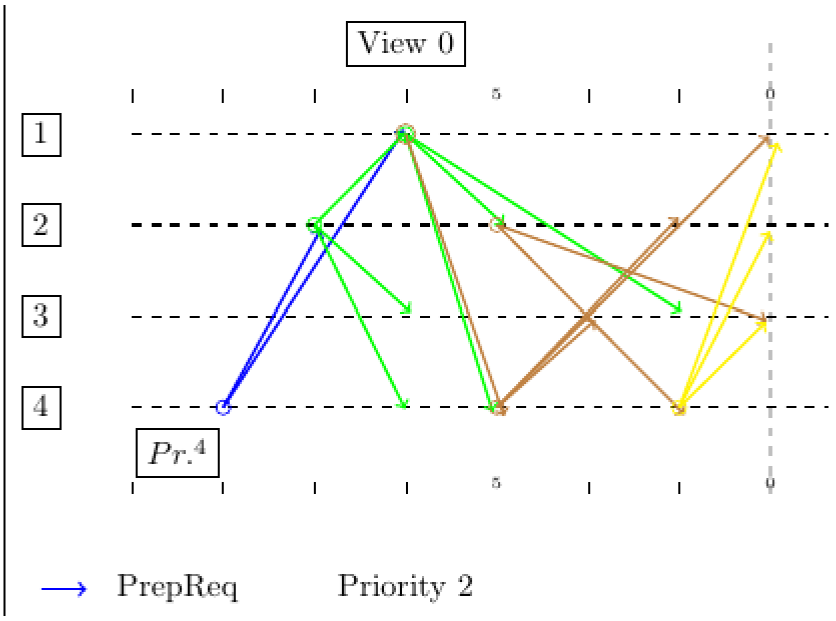
The following properties hold:

- No sporks were obtained for: `N=4` and `N=7`.
- The proposed speed-up mechanism was able to completely remove the additional phase when enabled;
- As previously emphasized, liveness holds if we ensure that `Commits`, `Preparations` arrives.
- Liveness holds when we penalize `View-Changes` exponentially.
    - By considering a exponential penalization on `View-Changes`, the mathematical model could always find an optimum solution with, at maximum, a single block.

The next figures shows a test with the following direction, a quase optimum solution is presented (due to the difficulty in maximizing number of messages - current solution took 30min optimized with a branch&cut algorithm):

```
# Maximize number of blocks, number of messages and views
python3 dbft3.0_2P.py --maximization --w1=1000 --w2=100 --w3=1 --speedup --
circle_all_send --rand_pos
```

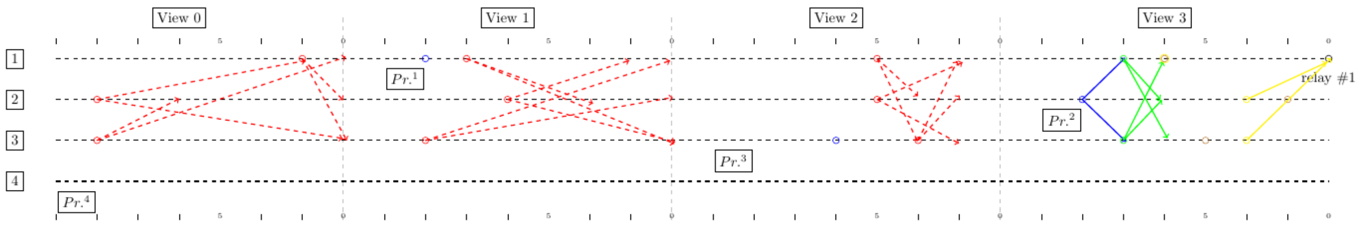




Next, we show a condition where messages are minimized but blocks are still maximized (as should be noticed, `Primary-P2` produces no messages in this optimization direction).

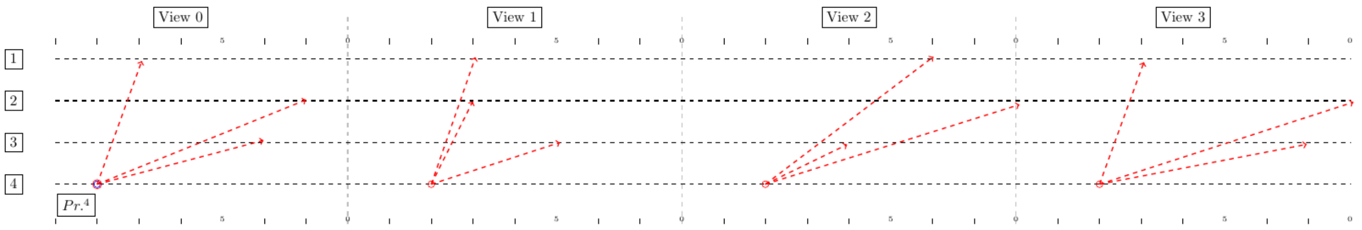
```
# Maximize number of blocks and views and minimizes messages
python3 dbft3.0_2P.py --maximization --w1=1000 --w2=100 --w3=-1 --speedup -
```

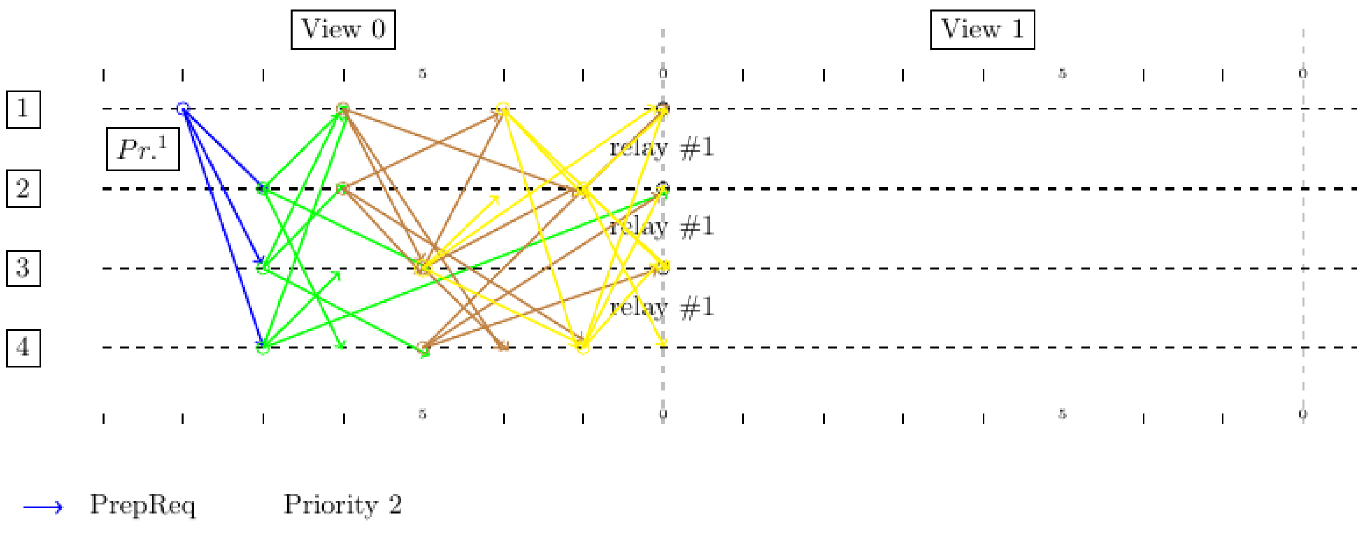
```
-circle_all_send --rand_pos
```



Following, we try to reduce the number of `views`, maximizing messages (an interesting perspective is that speed-up is not triggered for fallback primary, thus, block is produced there in order to maximize messages):

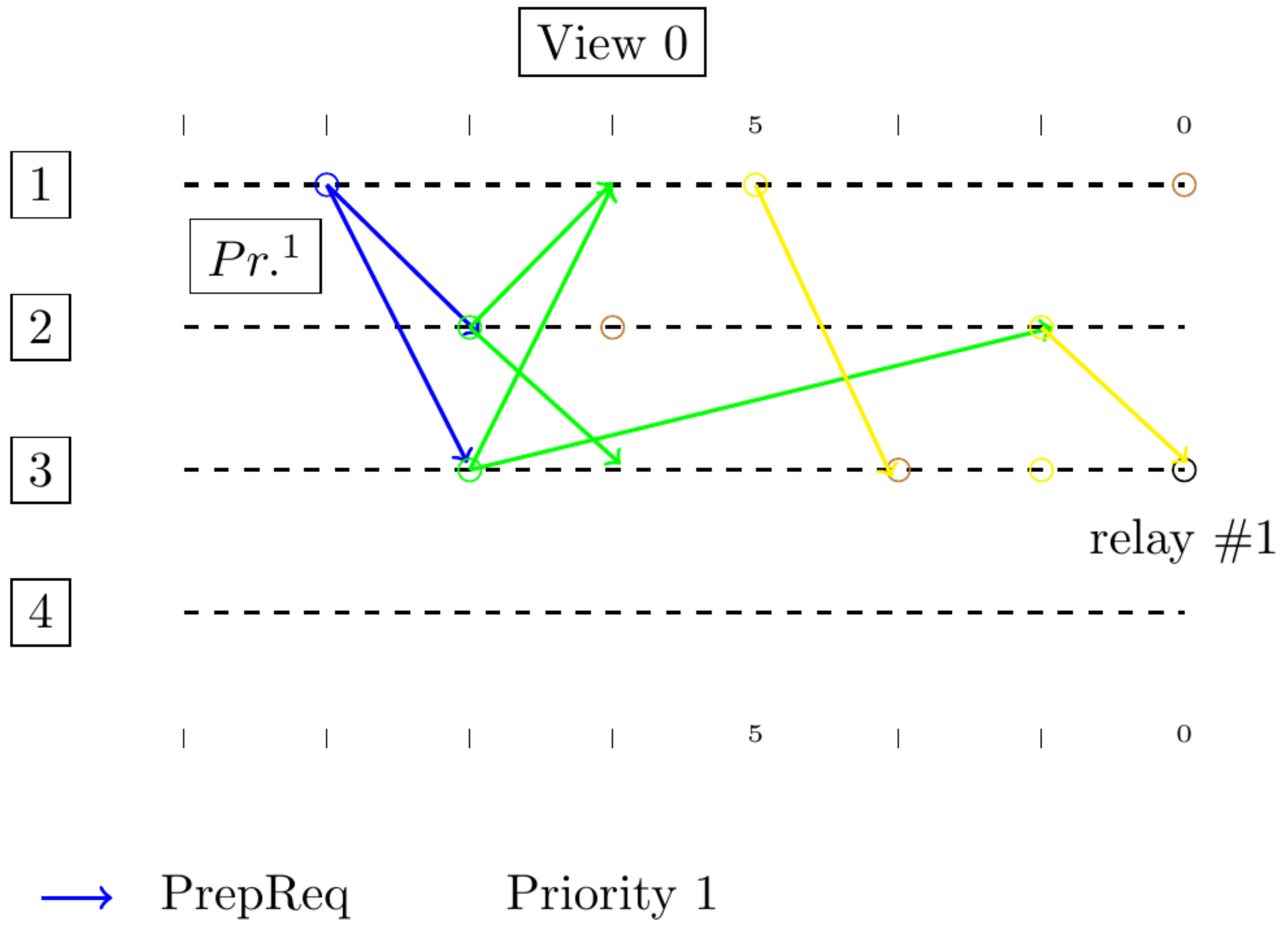
```
# Maximize number of blocks and messages and minimizes number of views
python3 dbft3.0_2P.py --maximization --w1=1000 --w2=-100 --w3=1 --speedup -
-circle_all_send --rand_pos
```





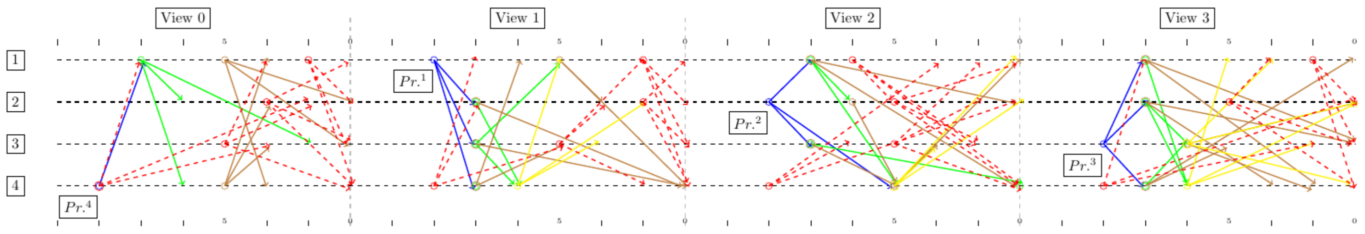
In the next case, the most efficient case is presented (fallback primary has no messages):

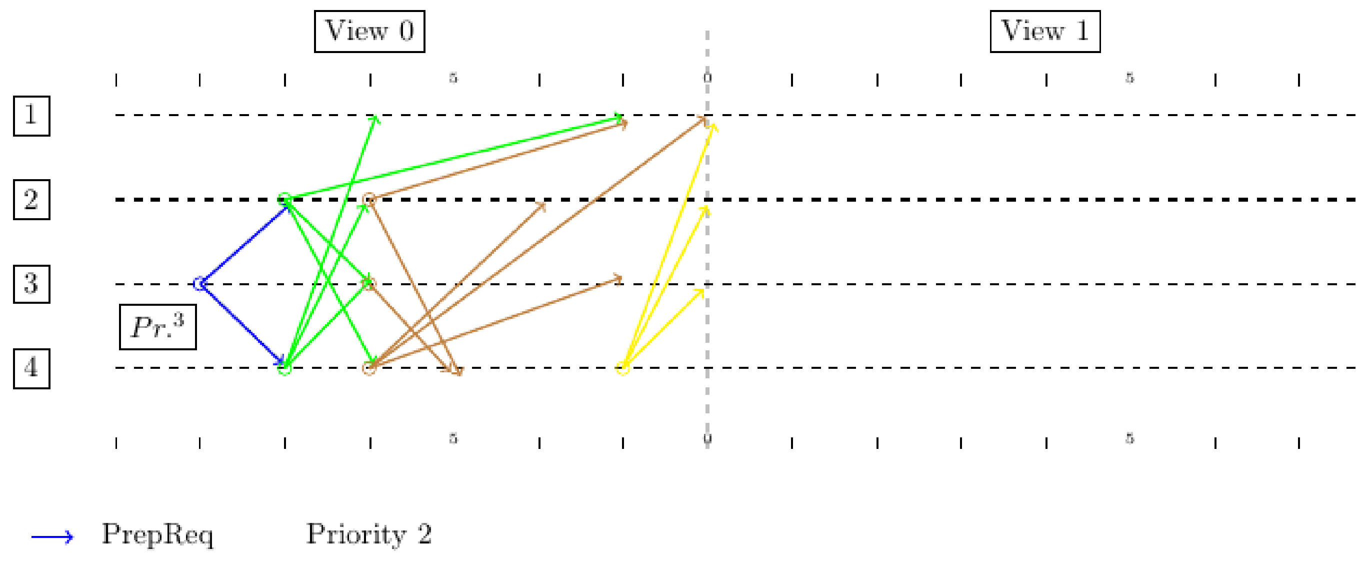
```
# Maximize number of blocks and minimize messages and views
python3 dbft3.0_2P.py --maximization --w1=1000 --w2=-100 --w3=-1 --speedup
--circle_all_send --rand_pos
```



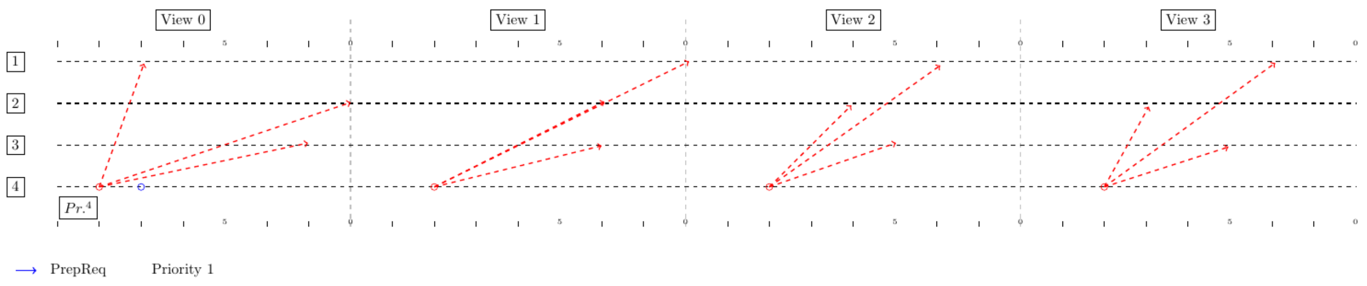
Next, we test liveness by trying to minimize number of blocks, but maximize views and messages (no blocks are produced, but honest nodes respect **Type-II Attack** and messages are maximized with assistance of fallback primary).

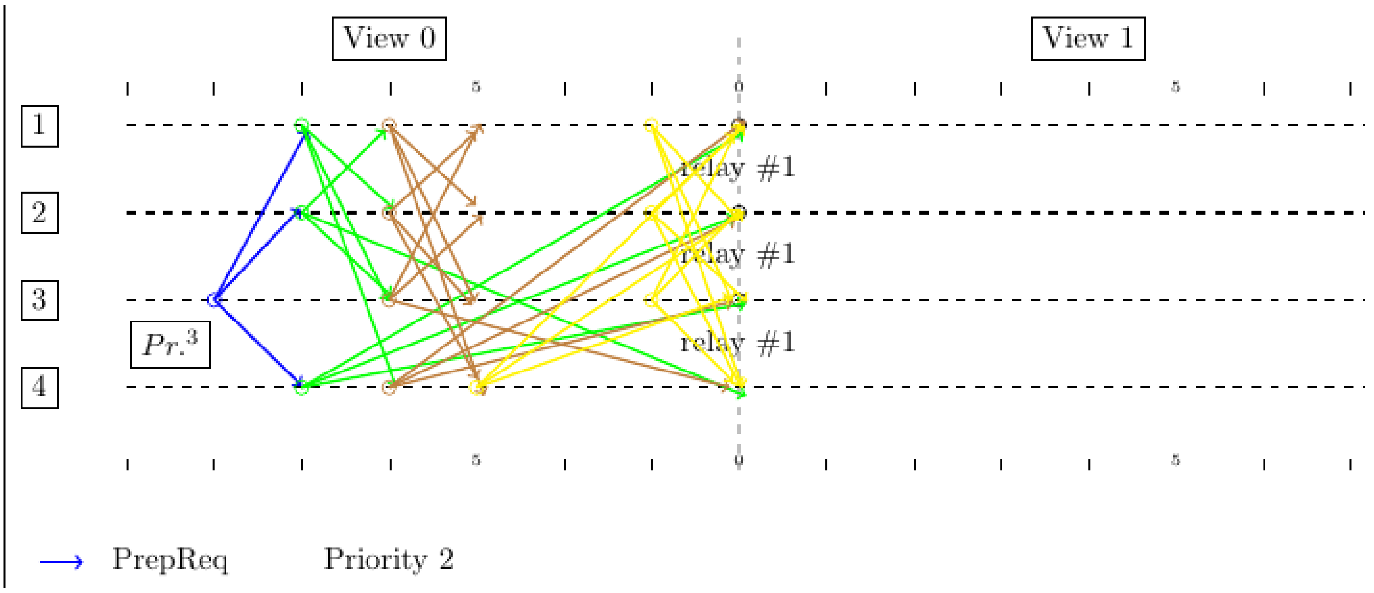
```
# Maximize number of blocks and minimize messages and views
python3 dbft3.0_2P.py --minimization --w1=1000 --w2=-100 --w3=-1 --speedup
--circle_all_send --rand_pos
```





Finally, by forcing messages to be received within simulation time of each `view`, a block will be produced in the first round because (for `N=4`) or as soon as an honest node becomes Primary (for `N>4`).





An analogous situation is obtained when we ponder number of views exponentially. Then, blocks are produced instead of delaying the network.

Additional considerations and features (optionals)

During our research we identified that consensus dBFT 2.0 and, consequently, dBFT 3.0 could receive some other changes such as described in the next sections.

dBFT 2.0+

A version of dBFT 2.0 that ensures that all proofs are sent during `change-view` phase.

- Replicas would inform which txs were verified with sucess (if `PrepareRequest` was received);
- If `f+1` supports the same transaction, it is guaranteed to exist (and be valid). If `2f+1` support that transaction, it is fundamental that the next primary should include it since, at least, `f+1` honest nodes are going to realize the aforementioned guarantee;
- If a sufficient amount of transactions are present (within a threshold), next primary could propose the same block, resolving all possible issues with nodes committed in the past.

This additional information sharing can surely play a crucial role in improving the `syncrony condiction` of the current dBFT 2.0.

For a more detailed description please check Section 4.3, "Aggregating Extra Information on dBFT 2.0: The dBFT 2.0+", of the paper "*Challenges of PBFT-Inspired Consensus for Blockchain and Enhancements over Neo dBFT*" available at https://www.mdpi.com/1999-5903/12/8/129.

dBFT 3.0+

The same characteristic of dBFT 2.0+ plus the following additional feature:

- It is impossible to conduct a **Type-II Attack** when `Pre-Commit` carry `Preparations`. That leaves no space for byzantine nodes to pass undetected (in empirical experiments this appeared to be true for all tested cases. In hand of some empirical experiments following done in https://github.com/NeoResearch/dbft3-spork-detect, it has been suggested that it will be nearly impossible for a byzantine node to pass unnoticed from double responses, thus certainly being blacklisted and, consequently, the spork prevented.);
- To be more speculative, one may follow `Pre-Commits` fast strategies (thus allowing the skipping of one phase in most cases), and issuing a blacklist on byzantine node only after it is detected (perhaps after issuing its own commit, which may cause a spork).

Working Window and Reliable Timing

Up to this point, we assumed that timing mechanisms of dBFT 2.0 and 3.0 are the same, following timers on a phase protocol for eventual protocol changes (view changes). However, we feel that it is also possible to enhance current timing strategy, in order to have more reliable timing and even a perfect timing approach to achieve exact `T` seconds, even when the node clocks are slightly incorrect.

Current approach on timing is based on the timestamp for *block persistance*, which is a reliable manner to verify times (since its local to the node), but very limited regarding network delays. Since the timers are based on that timing, let's say `t_local(H_0)`, and set to `t_local(H_0)+T`, blocks are generated after `t_local(H_0)+T+delta` seconds, where `delta` includes processing and communication times. It's very unlikely that the whole consensus system could generate blocks systematically at `T` seconds, since that would require no wide errors on `t_local(H_0)` and `delta=0` (no delays or processing times). We consider this approach to be *correct*, in the sense that attackers cannot easily manipulate network times and prevent consensus, but attackers can effectively change block timestamps up to few minutes.

We discuss the additional possibility of considering *block timestamps* during decision making, namely `t_header(H_0)`, instead of *block persistence time*. This means that the proposed timestamp must be validated by peers during consensus, within some safe and reliable time interval. Working window can be drastically increased, by proposing a block with a future timestamp `t_header(H_1)`, according to previous time `t_header(H_0)`. Time would be bound by the *median* of all signed commit payloads from previous phase plus a small deviation factor `lambda`. This means that only honest nodes could interfere on timing calculation, and as long as time does not vary up to this `lambda`, all block proposals would be naturally accepted. If a proposal is over `lambda+median` of known timing information, the replica can request a change view due to failed primary.

Proposed time protocol would be naturally disabled after view 0, in standard dBFT 2.0 mode. The reason is that, after some catastrophic event, time could be completely disrupted, requiring a "fresh start" from the consensus node pool. The same happens when a node is faulty and returns to the consensus pool, lacking commit payloads to include in proposal. In this situation, a change view would need to occur (or a valid proposal from the fallback primary).

NeoResearch team

September 20th, 2020