BFT Paxos comparison

Illia Melnyk, Oleksandr Kurbatov, Oleg Fomenko

June 2024

1 Introduction

The consensus problem can be described in terms of the actions taken by three classes of agents: *proposers*, who propose values, *acceptors*, who are responsible for choosing a single proposed value, and *learners*, who must receive the chosen value. This model looks simple, but in practice, there are a lot of challenges if we need to provide safety and liveness properties here. Paxos is one of the standard consensus-reaching protocols that has grown into a whole family of protocols for different systems. The paper aims to analyze and compare Paxos-based algorithms that provide BFT property.

2 Preliminaries

Here are some terms for general understanding:

Safety — a property that must satisfy two stages: agreement, that no two correct processes decide on different values and validity, that a correct process decides on a valid value.

Liveness — a property that provide termination, that every correct process eventually decides on a value.

Fault tolerance — is an ability of a model to continue operating without interruption while some processes fail. In the context of Paxos, fault tolerance guarantees that the algorithm can tolerate the failure of a minority of nodes (acceptors) and still reach consensus.

2.1 Classic Paxos

Paxos proposes the protocol of conducting numbered ballots, each orchestrated by a leader (multiple ballots with different leaders concurrently are possible). Let N be the number of acceptors, f of which can be failing and $N > f$. Let a *quorum* be any $N - f$ acceptors. For safety, any two quorums must have a non-empty intersection, which is true if $N > 2f$.

The general scheme of the Paxos phases is shown as follows:

$1a$ The ballot-b leader sends a $1a$ message to all acceptors.

$1b$ Each an acceptor responds to the leader‘s ballot-b $1a$ message with a $1b$ message containing the number of the highest-numbered ballot in which it has voted and the value it voted, or saying that it has cast no votes.

$2a$ Using the $1b$ messages sent by a quorum of acceptors, the leader chooses a value v that is safe at b and sends a $2a$ message containing v to the acceptors.

$2b$ Upon receipt of the leader‘s ballot-b $2a$ message, an acceptor votes for v in ballot b by sending a $2b$ message.

The algorithm maintains the following properties:

P1. An acceptor can vote for a value v in ballot b only if v is safe at b.

P2. Different acceptors cannot vote for different values on the same ballot.

P3a. If no acceptor in the quorum has voted in a ballot numbered less than b, then all values are safe at b.

P3b. If some acceptor in the quorum has voted, let c be the highest-numbered ballot less than b in which such a vote was cast. The value voted for in ballot c is safe at b.

Properties $P3a$ and $P3b$ are necessary to ensure that the leader determines a safe value from the ballot-b $1b$ messages.

In addition, we describe a more general scheme of the classic Paxos, called PCon. It is almost identical to the above scheme, except that it splits phase $2a$ into two sub-phases, shifting the leader's selection of the sent values to phase $1c$, where it reports which multiple values are safe, and modifies phase $2a$ accordingly:

$1c$ Using the $1b$ messages from a quorum of acceptors, the leader chooses a set of values that are safe at b and sends a $1c$ message for each of those value.

$2a$ The leader sends a $2a$ message for some value for which it has sent a $1c$ message.

Note that it also requires the use of the following properties for choosing safe value:

P3a. If no acceptor in the quorum has voted in a ballot numbered less than b, then all values are safe at b.

P3c. If a ballot-c message with value v has been sent, for some $c < b$, and (i) no acceptor in the quorum has voted in any ballot greater than c and less than b, (ii) and any acceptor in the quorum that has voted in ballot c voted for v in that ballot, then v is safe at b.

3 BPCon (Byzantine classic Paxos consensus)

BPCon [Lam11] — is a consensus algorithm that has $N \geq 3f + 1$ acceptors, f of which are byzantine.

We define the set of *byzacceptors* as the union of the byzantine and honest acceptors. We define a *byzquorum* as a set of *byzacceptors* that is guaranteed to contain a quorum of acceptors. If the quorum consists of any q acceptors, then a *byzquorum* consists of any $q + f$ *byzacceptors*. For liveness, we need the assumption that the set of all honest acceptors (which we assume never fail) form a *byzquorum*.

3.1 BPCon‘s Phases

At first, we consider the difference between BPCon and not Byzantine Paxos:

1. There is no explicit $2a$ message or Phase 2a action. Instead, the acceptors cooperate to emulate sending a $2a$ message. The ballot-b leader requests that a Phase 2a action be performed for a value v for which it has already sent a $1c$ message. On receiving the first such request, an acceptor executes a Phase $2av$ action, sending a ballot-b $1c$ message with that value.

2. Because the algorithm must tolerate malicious leaders, the ballot-b leader send any $1a$ and $1c$ messages it wants. The BPCon Phase 1c action allows the ballot-b leader to send any ballot-b $1c$ message at any time. Essential that acceptors will ignore a $1c$ message unless it is legal. To ensure liveness, a nonfaulty leader must send a message that (honest) acceptors act upon.

3. In not Byzantine Paxos, the sending of a ballot-b $1c$ message is enabled by properties, which requires the receipt of a set of $1b$ messages from a quorum and possibly of a $1c$ message. In BPCon, into the $1b$ messages put additional information to enable the deduction that $1c$ message was sent. An acceptor includes in its $1b$ messages the set of all $2av$ messages that it has sent – except that for each value v, it includes (and remembers) only the $2av$ message with the highest numbered ballot that it sent for v. Each of those $2av$ messages was sent in response to a legal $1c$ message.

Also, BPCon should provide next conditions:

1. BP3a. Each message in S asserts that its sender has not voted.

2. BP3c. For some $c < b$ and some value v:

   (a) each message in S asserts that (i) its sender has not voted in any ballot greater than c and (ii) if it voted in c then that vote was for v

   (b) there are $f + 1$ $1b$ messages (not necessarily in S) from byzacceptors saying that they sent a $2av$ message with value v in ballot c.

A little thought shows we can weaken condition (b) of BP3c to assert:

($\tilde{b}$) there are $f + 1$ $1b$ messages from byzacceptors saying that they sent a $2av$ message with value v in a ballot $\geq c$.

After the above statements and details, let us describe the general scheme of the action phases:

($1a$) The ballot-b leader sends a $1a$ message to the acceptors

($1b$) An acceptor responds to the leader's ballot-b $1a$ message with a $1b$ message containing the number of the highest-numbered ballot in which it has voted and the value it voted for in that ballot, or saying that it has cast no votes.

($1c$) Using the $1b$ messages from a byzquorum of acceptors, the leader chooses a set of values that are safe at b and sends a $1c$ message for each of those values.

($2av$) An acceptor received the necessary $1c$ action and send a ballot-b $2av$ message if it has not already done so.

($2b$) Upon receipt of the $2av$ messages, an acceptor votes for v in ballot b by sending a $2b$ message (whereas value v should be from a quorum of acceptors and no two acceptors can execute Phase $2b$ actions for different values)

3.2 BPCon's liveness

The requirements for liveness of BPCon are the same as not Byzantine Paxos: a nonfaulty leader executes a ballot b, no leader begins a higher-numbered ballot, and the leader and nonfaulty acceptors can communicate with one other. However, it is difficult to ensure that a Byzantine leader does not execute a higher-numbered ballot. Doing this seems to require an engineering solution based on real-time assumptions. One such solution is presented by Castro and Liskov. And assuming these requirements, liveness of BPCon requires satisfying the following two conditions:

BL1. The leader can find $1b$ messages satisfying *BP3a* or *BP3c*

BL2. All honest acceptors will know that those messages have been sent

These two conditions imply that that the leader will send a legal $1c$ message, a byzquorum BQ of honest (nonfaulty) acceptors will receive that $1c$ message and send $2av$ messages, all the acceptors in BQ will receive those $2av$ messages and send $2b$ messages. Learners, upon receiving those $2b$ messages will learn that the value has been chosen.

3.3 Castro-Liskov Algorithm

The Castro-Liskov algorithm (or CLA) is a refined BPCon that contains engineering optimizations for dealing with the sequence of instances – in particular, for garbage collecting old instances and for transferring state to repaired processes.

In the CLA, byzacceptors are called *replicas*. The ballot-b leader is the replica called the primary, other byzacceptors being called *backups*. The replicas also serve as learners.

General scheme of the action phases:

($1a$) There is no explicit $1a$ message; its sending is emulated cooperatively by replicas when they decide to begin a view change.

($1b$) This is the *view-change* message

(1c) During a view change, the new-view message acts like 1c messages for all the consensus instances. For an instance in which the primary instructs the replicas to choose a specific value, it is a 1c message with that value. For all other instances, it is a set of 1c messages for all values. (Condition BP3a holds in those other instances.) The acceptors check the validity of these 1c messages simultaneously for all instances.

(2av) This is a backup's prepare message. The pre-prepare message of the primary serves as its 2av message and as the message (not modeled in *BPCon*) that requests a Phase 2a action.

(2b) This is the commit message.

4 BGP (Byzantine Generalized Paxos)

Byzantine Generalized Paxos consensus (or BGP) [PRR18] – algorithm, that leverages a weaker performance guarantee to decrease the requirements regarding the minimum number of processes. In detail, BGP only provides a two communication step latency when proposed sequences are commutative with any other concurrently proposed sequence.

Suppose there is an asynchronous system in which a set of $n \in \mathbb{N}$ processes communicate by *sending* and *receiving* messages. Each process may fail in two different ways: may stop executing algorithm by crashing or may stop following the algorithm assigned to it, in which case it considered *Byzantine*. We define that a non-Byzantine process is *correct*. Let consider the *authenticated* Byzantine model: every process can produce cryptographic digital signatures. Also assumes that authenticated perfect links, where a message that it sent by nonfaulty sender is eventually received and message cannot be forged. A process may be a *learner*, *proposer* or *acceptor*. Protocol requires a minimum number of acceptor processes (N), which is a function of the maximum number of tolerated Byzantine faults (f), namely $N \geq 3f + 1$. Let assume that acceptor processes have identifiers in the set $\{0, ..., N - 1\}$.

In simplified specification of Generalized Paxos, each learner l maintains a monotonically increasing sequence of commands $learned_l$. *C-structs* (sets of values, that are formed from "null" element, which call $\perp$ by the operation of appending command) provide this increasing sequence abstraction and allow the definition of different consensus problems.

Let's define that two learned sequences of commands to be equivalent ($\sim$), if one can be transformed into the other by permuting the elements in a way such that the order of non-commutative pairs is preserved. A sequence x is defined to be an eq-prefix of another sequence $y (x \sqsubseteq y)$, if the subsequence of y that contains all the elements in x is equivalent ($\sim$) to x. To simplify the original specification, instead of using c-structs, BGP specialize to agreeing on equivalent sequences of commands:

1. **Nontriviality.** If all proposers are correct, $learned_l$ can only contain proposed commands.
2. **Stability.** If $learned_l = s$ then, at all later times, $s \sqsubseteq learned_l$, for any sequence s and correct learner l.
3. **Consistency.** At any time and for any two correct learners l_i and l_j , $learned_{l_i}$ and $learned_{l_j}$ can subsequently be extended to equivalent sequences.
4. **Liveness.** For any proposal s from a correct proposer, and correct learner l, eventually $learned_l$ contains s.

Algorithm 1 Byzantine Generalized Paxos – Proposer p

Local variables: $ballot\_type = \perp$

```
1: Upon receive(BALLOT, type) do
2:     ballot_type = type;
3:
4: Upon command_request(c) do
5:     if ballot_type == fast_ballot then
6:         SEND(P2A_FAST, c) to acceptors;
7:     else
8:         SEND(PROPOSE, c) to leader;
```

Let's split the protocol into two parts – *View-change* and *Agreement protocol*.

4.1 View-change

The goal of the view-change subprotocol is to elect a distinguished acceptor process, called the leader, that carries through the agreement protocol, i.e., enables proposed commands to eventually be learned by all the learners (similar to the corresponding part of existing BFT state machine replication protocols).

In this subprotocol, the system moves through sequentially numbered views, and the leader for each view is chosen in a rotating fashion using equation: leader (view) = *view mod* N. The protocol works continuously by having acceptor processes monitor whether progress is being made on adding commands to the current sequence, and, if not, they multicast a signed suspicion message for the current view to all acceptors suspecting the current leader. Then, if enough suspicions are collected, processes can move to the subsequent view. To prevent Byzantine processes, acceptor processes will multicast a view-change message indicating their commitment to starting a new view only after hearing that $f + 1$ processes suspect the leader to be faulty. This message contains the new view number, the $f+1$ signed suspicions, and is signed by the acceptor that sends it. This way, if a process receives a view-change message without previously receiving $f + 1$ suspicions, it can also multicast a view-change message, after verifying that the suspicions are correctly signed by $f + 1$ distinct processes. This guarantees that if one correct process receives the $f + 1$ suspicions and multicasts the view-change message, then all correct processes, upon receiving this message will be able to validate the proof of $f + 1$ suspicions and also multicast the view-change message.

Algorithm 2 Byzantine Generalized Paxos – Leader 1

Local variables: $ballot_l = 0$, $maxTried_l = \perp$, $proposals = \perp$, $accepted = \perp$, $notAccepted = \perp$, $view = 0$

1: **Upon** *receive(LEADER, $view_a$, proofs)*
2: from acceptor a **do**
3: $valid\_proofs = 0$;
4: **for** p **in** acceptors **do**
5: $view\_proof = proofs[p]$;
6: **if** $view\_proof_{pub_p} == \langle view\_change, view_a \rangle$ **then**
7: $valid\_proofs+ = 1$;
8: **if** $valid\_proofs > f$ **then**
9: $view = view_a$;
10:
11: **Upon** $trigger\_next\_ballot(type)$ **do**
12: $ballot_l+ = 1$;
13: SEND*(BALLOT, type)* to proposers;
14: **if** $type == fast$ **then**
15: SEND$(FAST, ballot_l, view)$ to acceptors;
16: **else**
17: SEND$(P1A, ballot_l, view)$ to acceptors;
18:
19: **Upon** $receive(PROPOSE, prop)$ from proposer p_i **do**
20: **if** isUniversallyCommutative(*prop*) **then**
21: SEND$(P2A_CLASSIC, ballot_l, view, prop)$;
22: **else**
23: $proposals = proposals \bullet prop$;
24:
25: **Upon** $receive(P1B, ballot, bal_a, proven, val_a, proofs)$
26: from acceptor a **do**
27: **if** $ballot \neq ballot_l$ **then**
28: **return**;
29:
30: $valid\_proofs = 0$;
31: **for** i **in** $acceptors$ **do**
32: $proof = proofs[proven][i]$;
33: **if** $proof_{pub_i} == \langle bal_a, proven \rangle$ **then**
34: $valid\_proofs+ = 1$;
35:
36: **if** $valid\_proofs > N - f$ **then**
37: $accepted[ballot_l][a] = proven$;
38: $notAccepted[ballot_l] = notAccepted[ballot_l] \bullet$
39: $(val_a \backslash proven)$
40:
41: **if** $\#(accepted[ballot_l] \geq N - f)$ **then**
42: PHASE_2A();
43:
44: **function** PHASE_2A
45: $maxTried =$ LARGEST_SEQ $(accepted[ballot_l])$;
46: $previousProposals =$REMOVE_DUPLICATES(
47: $notAccepted[ballot_l])$;
48: $maxTried = maxTried \bullet previosProposals \bullet proposals$;
49: SEND$(P2A_CLASSIC, ballot_l, view, maxTried_l)$ to acceptors;
50: $proposals = \perp$

Finally, an acceptor process must wait for $N - f$ view-change messages to start participating in the new view, i.e., update its view number and the corresponding leader process. At this point, the acceptor also assembles the $N - f$ view-change messages proving that others are committing to the new view, and sends them to the new leader. This allows the new leader to start its leadership role in the new view once it validates the $N - f$ signatures contained in a single message.

4.2 Agreement protocol

In BGP, instead of being a separate instance of consensus, ballots correspond to an extension to the sequence of learned commands of a single ongoing consensus instance. Proposers can try to extend the current sequence by either single commands or sequences of commands. Term *proposal* is define either the command or sequence of commands that was proposed.

Ballots can either be classic or fast. Classic ballots work as follows:

1. The leader continuously collects proposals by assembling all commands that are received from the proposers since the previous ballot in a sequence;

2. When the next ballot is triggered, the leader starts the first phase by sending phase 1a messages to all acceptors containing just the ballot number;

3. Similarly to classic Paxos, acceptors reply with a Phase 1b message to the leader, which reports all sequences of commands they voted for. Leader need to collect $N - f$ proofs from Phase $1b$ messages. By waiting for $N - f$ of such messages, the leader is guaranteed that, for any learned sequence s, at least one of the messages will be from a correct acceptor that, due to the quorum intersection property, participated in the verification phase of s. It is important for the leader, because, for BGP, $f + 1$ identical votes for some value only attest that at least one correct process voted for that value. It's impossible to determine if some value was chosen by a majority unless the leader witnesses $2f + 1$ identical votes. Note that, since each command is signed by the proposed (this signature and its check is not explicit in the pseudocode), a Byzantine acceptor can't relay made-up commands. However, it can omit commands from its Phase $1b$ message, which is why it's necessary for the leader to be sure that at least one correct acceptor in its quorum took part in the verification quorum of any learned sequence.

4. After gathering a quorum of $N - f$ Phase $1b$ messages, the leader initiates Phase $2a$ where it assembles a proposal and sends it to the acceptors. This proposal sequence must be carefully constructed by next steps:

   (a) The first part of the sequence will be the largest sequence of the $N - f$ prove sequences sent in the Phase $1b$ messages. The leader can pick such a value deterministically because, for any two proven sequences, they are either equivalent or one can be extended to the other. The leader is sure of this because for the quorums of any two proven sequences there is at least one correct acceptor that voted in both and votes from correct acceptors are always extensions of previous votes from the same ballot. If there are multiple sequences with the maximum size then they are equivalent (by same reasoning applied previously) and any can be picked.

   (b) The second part of the sequence is simply the concatenation of unproven sequences or commands in an arbitrary order. Since $N-f$ Phase $2b$ messages are required for a learner to learn a sequence and the intersection between the leader's quorum and the quorum gathered by a learner for any sequence contains at least one correct acceptor, the leader can be sure that if a sequence of commands is unproven in all of the gathered Phase $1b$ messages, then that sequence wasn't learned and can be safely appended to the leader's sequence in any order.

   (c) The third part consists simply of commands sent by proposers to the leader with the intent of being learned at the current ballot. These values can be appended in any order and without any restriction since they're being proposed for the first time.

Fast ballots, in turn, allow any proposer to attempt to contact all acceptors in order to extend the current sequence within only two message delays (in case there are no conflicts between concurrent proposals). Let's explain each of the protocol's steps for fast ballots in greater detail:

1. **Proposer to acceptors.** To initiate a fast ballot, the leader informs both proposers and acceptors that the proposals may be sent directly to the acceptors. Unlike classic ballots, in a fast ballot, proposals can be sent to the acceptors in the form of either a single command or a sequence to be appended to the command history. These proposals are sent directly from the proposers to the acceptors.

2. **Acceptors to acceptors.** Acceptors append the proposals they receive to the proposals they have previously accepted in the current ballot and broadcast the result to the other acceptors. This broadcast contains a signed tuple of the current ballot and the sequence being voted for and intuitively corresponds to a verification phase where acceptors gather proof that a sequence gathered enough support to be committed. Recall that a sequence is equivalent to another if it can be transformed into the second one by reordering its elements without changing the order of any pair of non-commutative commands. (In the pseudocode, proofs for equivalent sequences are being treated as belonging to the same index of the proofs variable, to simplify the presentation.) By

requiring $N - f$ votes for a sequence of commands, we ensure that, given two sequences where non-commutative commands are differently ordered, only one sequence will receive enough votes even if f Byzantine acceptors vote for both sequences. Outside the set of (up to) f Byzantine acceptors, the remaining $2f + 1$ correct acceptors will only vote for a single sequence, which means there are only enough correct processes to commit one of them. Note that the fact that proposals are sent as extensions to previous sequences is critical to the safety of the protocol. In particular, since the votes from acceptors can be reordered by the network before being delivered to the learners, if these values were single commands it would be impossible to guarantee that non-commutative commands would be learned in a total order.

3. **Acceptors to learners.** Phase 2*b* messages, which are sent from acceptors to learners, contain the current ballot number, the command sequence and the $N - f$ proofs gathered in the verification round. Due to the possibility of learners learning sequences without the leader being aware of it, if we allowed the learners to learn after witnessing $N - f$ proofs for just one acceptor then that would raise the possibility of that acceptor not being present in the quorum of Phase 1*b* messages. Therefore, the leader wouldn't be aware that some value was proven and learned. The only way to guarantee that at least one correct acceptor will relay the latest proven sequence to the leader is by forcing the learner to require $N - f$ Phase 2*b* messages since only then will one correct acceptor be in the intersection of the two quorums.

4. **Arbitrating an order after a conflict.** When, in a fast ballot, non-commutative commands are concurrently proposed, these commands may be incorporated into the sequences of various acceptors in different orders, and therefore the sequences sent by the acceptors in Phase 2b messages will not be equivalent and will not be learned. In this case, the leader subsequently runs a classic ballot and gathers these unlearned sequences in Phase 1b. Then, the leader will arbitrate a single serialization for every previously proposed command, which it will then send to the acceptors. In this case, the learners will learn them in a total order, thus preserving consistency.

Algorithm 3 Byzantine Generalized Paxos – Acceptor a (view-change)

Local variables: $suspicions = \perp$, $new\_view = \perp$, $leader = \perp$, $view = 0$, $bal_a = 0$, $val_a = \perp$, $fast\_bal = \perp$, $checkpoint = \perp$

Upon *suspect_leader* **do**
  if $suspicions[p] \neq true$ **then**
    $suspicions[p] = true$;
    $proof = \langle suspicion, view \rangle_{priv_a}$;
    SEND$(SUSPICION, view, proof)$);

Upon $receive(SUSPICION, view_i, proof)$ from acceptor i **do**
  if $view_i \neq view$ **then**
    return;
  if $proof_{pub_i} == \langle suspicion, view \rangle$ **then**
    $suspicions[i] = proof$;
  if $\#(suspicions) > f$ and $new\_view[view + 1][p] == \perp$ **then**
    $change\_proof = \langle view\_change, view + 1 \rangle_{priv_a}$;
    $new\_view[view + 1][p] = change\_proof$;
    SEND*(VIEW_CHANGE, view+1, suspicions, change_proof)*;

Upon $receive(VIEW\_CHANGE, new\_view_i, suspicions, change\_proof_i)$ from acceptor i **do**
  if $new\_view_i \leq view$ **then**
    return;

  $valid\_proofs = 0$;
  for p **in** $acceptors$ **do**
    $proof = suspicions[p]$;
    $last\_view = new\_view_i - 1$;
    if $proof_{pub_p} == \langle suspicion, last\_view \rangle$ **then**
      $valid\_proofs += 1$;

  if $valid\_proofs \leq f$ **then**
    return;

  $new\_view[new\_view_i][i] = change\_proof_i$;
  if $new\_view[view_i][a] == \perp$ **then**
    $change\_proof = \langle view\_change, new\_view_i \rangle_{priv_a}$;
    $new\_view[view_i][a] = change\_proof$;
    SEND*(VIEW_CHANGE, view$_i$, suspicions, change_proof)*;

  if $\#(new\_view[new\_view_i]) \geq N - f$ **then**
    $view = view_i$;
    $leader = view \bmod N$;
    $suspicions = \perp$;
    SEND $(LEADER, view, new\_view[view_i])$ to leader;

Algorithm 4 Byzantine Generalized Paxos – Acceptor a (agreement)

Local variables: $leader = \perp$, $view = 0$, $bal_a = 0$, $val_a = \perp$, $fast\_bal = \perp$, $proven = \perp$

1: **Upon** $receive(P1A, ballot, view_l)$ from leader l **do**
2: **if** $view_l == view$ and $bal_a < ballot$ **then**
3: SEND $(P1B, ballot, bal_a, proven, val_a, proofs[bal_a])$
4: to leader;
5: $bal_a = ballot$;
6:
7: **Upon** $receive(FAST, ballot, view_l)$ from leader **do**
8: **if** $view_l == view$ **then**
9: $fast\_bal[ballot] = true$;
10:
11: **Upon** $receive(VERIFY,\ view_i,\ ballot_i,\ val_i,\ proof)$
12: from acceptor i **do**
13: **if** $proof_{pub_i} == \langle ballot_i, val_i \rangle$ and
14: $view == view_i$ **then**
15: $proofs[ballot_i][val_i][i] = proof$;
16: **if** $(\#(proofs[ballot_i][val_i]) \geq N - f$ or
17: $(\#(proofs[ballot_i][val_i]) > f$ and
18: isUniversallyCommutative$(val_i)))$
19: and $proofs[ballot_i][val_i][a] \neq \perp$ **then**
20: $proven = val_i$;
21: SEND $(P2B, ballot_i, val_i, proofs[ballot_i][value_i])$
22: to learners;
23:
24: **Upon** $receive(P2A\_CLASSIC, ballot, view, value)$
25: from leader **do**
26: **if** $view_l == view$ **then**
27: PHASE_2B_CLASSIC$(ballot, value)$;
28:
29: **Upon** $receive(P2A\_FAST, value)$ from proposer **do**
30: PHASE_2B_FAST$(value)$;
31:
32: **function** PHASE_2B_CLASSIC$(ballot, value)$
33: $univ\_commut$ = isUniversallyCommutative(val_a);
34: **if** $ballot \geq bal_a$ and $!fast\_bal[bal_a]$ and $(univ\_commut$ or $proven == \perp$ or $proven ==$ SUBSEQUENCE$(value, 0, \#(proven))$ **then**
35: $bal_a = ballot$;
36: **if** $univ\_commut$ **then**
37: SEND $(P2B, bal_a, value)$ to learners;
38: **else**
39: $val_a = value$;
40: $proof = \langle ballot, val_a \rangle_{priv_a}$;
41: $proofs[ballot][val_a][a] = proof$;
42: SEND $(VERIFY, view, ballot, val_a, proof)$ to acceptors;
43: **function** PHASE_2B_FAST$(ballot, value)$
44: **if** $ballot == bal_a$ and $fast\_bal[bal_a]$ **then**
45: **if** isUniversallyCommutative $(value)$ **then**
46: SEND $(P2B, bal_a, value)$ to learners;
47: **else**
48: $val_a = val_a \bullet value$;
49: $proof = \langle ballot, val_a \rangle_{priv_a}$;
50: $proofs[ballot][val_a][a] = proof$;
51: SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors;

Algorithm 5 Byzantine Generalized Paxos – Learner l

Local variables: $learned = \perp$, $messages = \perp$

1: **Upon** $receive(P2B, ballot, value, proofs)$ from acceptor a
2: **do**
3: $valid\_proofs = 0$;
4: **for** i **in** $acceptors$ **do**
5: $proof = proofs[i]$;
6: **if** $proof_{pub_i} == \langle ballot, value \rangle$ **then**
7: $valid\_proofs+ = 1$;
8: **if** $valid\_proofs \geq N - f$ **then**
9: $messages[ballot][value][a] = proofs$;
10: **if** $\#(messages[ballot][value]) \geq N - f$ **then**
11: $learned$ = MERGE_SEQUENCES$(learned, value)$;
12:
13: **Upon** $receive(P2B, ballot, value)$ from acceptor a **do**
14: **if** isUniversallyCommutative$(value)$ **then**
15: $messages[ballot][value][a] = true$;
16: **if** $\#(messages[ballot][value]) > f$ **then**
17: $learned = learned \bullet value$;
18:
19: **function** MERGE_SEQUENCES(old_seq, new_seq)
20: **for** c **in** new_seq **do**
21: **if** !CONTAINS(old_seq, c) **then**
22: $old\_seq = old\_seq \bullet c$;
23: **return** old_seq;

5 Byzantine Vertical Paxos

Byzantine Vertical Paxos (or BVP) [LMZ09] – is a protocol that focus on two aspects, elasticity (dynamic reconfiguration) and throughput, and be Byzantine variant of Vertical Paxos. As we know Vertical Paxos separates a solution into two modes: a simple steady state protocol and a reconfiguration protocol. Often, the steady state protocol is just primary-backup or its generalization to multiple nodes via Chain Replication (CR) or Two Phase Commit (2PC). In BVP the steady state mode can be simple and highly optimized despite the Byzantine settings.

Additionally, in many realistic settings the network is synchronous. In this case, our steady-state solution requires only $f + 1$ replicas, and $2f + 1$ replicas for reconfiguration. In fact, BVP does not really require the system to be synchronous in steady state, only to have some out-of-band synchronous control channel for reconfiguration purposes. Yet other settings may have secure hardware devices such as TPMs.

5.1 Wedging a Replicated State-Machine

State-Machine Replication (SMR) is the task of forming agreement on a sequence of state-machine *commands*. Consensus on each command is formed independently of other commands. In steady state,

there is a fixed set of replicas and a fixed algorithm driving decisions, one after another.

A *reconfiguration* mechanism changes the steady-state mode of the system. Reconfiguration may be employed to change the entire set of replica or, for example, it can be employed to replace a leader in a leader-based scheme.

The core mechanism employed for reconfiguration is a wedging scheme. A wedging coordinator obtains validation from a wedge, which is a subset of the replicas. At the same time, the coordinator obtains the latest state the wedge stores (including all the proposals it stores in every sequence position).

Then the coordinator drives a reconfiguration consensus decision. This consensus decision is implemented by a separate Byzantine consensus engine called a *reconfiguration engine*. Importantly, the reconfiguration consensus decision itself has two components: *next configuration* and *closing state*. The second component is important for us, because, when wedging starts, some consensus decisions may be only partially completed. This is inevitable in a distributed system, and consequently, there is going to be uncertainty about the status of ongoing decisions. For examle, in a $(2f + 1)$-of-$(3f + 1)$ scheme, a wedging coordinator collects information from $2f + 1$, leaving the remaining f unknown. If it hears that $f + 1$ (of the $2f + 1$) voted for a certain state-machine command, say command number 1, the only safe course for the coordinator is to adopt the command into the closing state. Note that it is unknown whether the remaining f replicas ever vote for this command, past or future.

After the wedging procedure is complete and reconfiguration a consensus decision is reached, the SMR implementation switches to a conceptually new system (although the configuration change itself may be minimal, e.g., a leader change).

5.2 Asynchronous Model

In this model, PBFT gives 4 rounds and $2f + 1$ for steady state and Zyzzyva uses 3 rounds in the optimistic case but requires $3f + 1$ replicas for the steady state. Using $3f + 1$ for reconfiguration is also standard in both cases.

5.3 Synchronous-Reconfiguration Model

In the synchronous model, BVP provide two options, a 4-round (4 message delays) solution with $f + 1$ replicas, and a 3-round (3 message delays) solution with $2f + 1$ replicas.

4-**Round:** The first option is a steady-state mode with one (trivial) quorum of $f + 1$ replicas. Can run a standard 4-round protocol:

Round 1	Client sends to Primary
Round 2	Primary signs and sends to all $f + 1$
Round 3	All $f + 1$ send signed-echoes of the Primary‘s message to each other
Round 4	Each of $f + 1$ sends a composite message containing all signed-echoes to Client
Client proceeds when all $f + 1$ composites arrive	
Closing state	Every composite containing $f + 1$ signed echoes

Table 1: A standart 4-round protocol

Note that we do not require a synchronous model for this interaction, only for reconfiguration. Namely, if at any stage the delay gets too large we can run a reconfiguration. Each round icurs one message delay, for a total of 4. The third round is an all-to-all exchange, and the fourth has linear size messages, each incurring quadratic single-message complexity.

Reconfiguration is handled as follows. The wedging coordinator must contact all surviving, nonfaulty replicas in order to prevent Byzantine replicas from truncating the history of validated Primary commands. The coordinator relies on a known bound on communication delays and waits for it to expire. This guarantees that any surviving, nonfaulty replica responds to the coordinator. Every command for which the wedging coordinator receives a composite message with $f + 1$ signed echoes is adopted in the closing state.

On a practical note, it is possible to engineer systems to switch to synchronous mode when needed, e.g., by employing during these periods certain networking infrastructure or a special authority which has network priority. Alternatively, we could simply assume that a system has synchrony. As for the

consensus reconfiguration engine itself, it is well known that this can be done using $2f + 1$ replicas in the synchronous model.

3-**Round:** The second option is a steady-state mode with one (trivial) quorum of $2f + 1$ replica, a Zyzzyva-like 3-round protocol:

Round 1	Client sends to Primary
Round 2	Primary signs and sends to all $2f + 1$
Round 3	All $2f + 1$ send signed-echoes of the Primary's message to Client
Client proceeds when all $f + 1$ composites arrive	
Closing state	Every command which has $f + 1$ signed echoes

Table 2: A standart Zyzzyva-like 3-round protocol

5.4 Asynchronous Model with a TPM

In this setting servers are equipped with Trusted Platform Module. Formally let's assume a weak sequential broadcast — WScast that ensures that

(a) messages from a given sender are delivered by correct processes in the same order; this is an ordering per-sender, similar to FIFO broadcast

(b) if the sender is correct then eventually all processes will receive all its messages.

This model provide a SMR solution with a steady state 3-round protocol, using $f + 1$ replicas, and incurring a linear message complexity:

Round 1	Client sends to Primary
Round 2	Primary WCast to all $f + 1$
Round 3	All $2 + 1$ WCast echoes of the Primary's message to Client
Client proceeds when $f + 1$ echoes arrive	
Closing state	Every command which has $f + 1$ signed echoes

Table 3: A SMR solution with a 3-round protocol

Note that there are several performance advantages of using a TPM. In steady state, we need only $f + 1$ replicas, and at the same time, communication is linear in number of replicas (unlike the quadratic number of messages in the standard models). Another advantage of using secure hardware, which is left outside the scope of discussion here, is the possibility to leverage "proof of elapsed time" to simplify and improve the efficiency of the leader election part inside the reconfiguration service.

6 Fast Byzantine Paxos

Fast Byzantine Paxos (or FaB) [MA06] – is a protocol that requires only two communication steps to reach consensus in the common case. It reduced latency comes at a price: FaB Paxos requires $a \geq 5f + 1$ acceptors to tolerate f Byzantine acceptors, and also $p \geq 3f + 1$ proposers, $l \geq 3f + 1$ learners. As in Paxos, each process can play one or more of these three roles.

6.1 FaB Paxos in the Common Case

Let's first describe how FaB Paxos works when there is a unique correct leader and all correct acceptors agree on its identity.

This model can satisfy next safety (CS1-CS3) and liveness (CL1-CL2):

CS1 Only a value that has been proposed may be chosen.

CS2 Only a single value may be chosen.

CS3 Only a chosen value may be learned by a correct learner.

CL1 Some proposed value is eventually chosen.

CL2 Once a value is chosen, correct learners eventually learn it.

variable	initial	comment
Globals		
p,a,l		Number of proposers, acceptors, learners
f		Number of Byzantine failure tolerated
Proposer variables		
Satisfied	$\emptyset$	Set of proposers that claim to be satisfied
Learned	$\emptyset$	Set of learners that claim to have learned
Acceptor variables		
accepted	$(\perp, \perp)$	Value accepted and the corresponding proposal number
Learner variables		
learner.accepted[j]	$(\perp, \perp)$	Value and matching proposal number acceptor j says it accepted
learner.learn[j]	$(\perp, \perp)$	Value and matching proposal number learner j says it learned
learner.learned	$(\perp, \perp)$	Value learned and the corresponding proposal number

Table 4: Variables for the FaB pseudocode

Algorithm 6 Byzantine Fast Paxos – Interface for leader election protocol

```
1: int leader-election.getRegency()
2: // return the number of the current regent (leader is regent % p)
3: // if no correct node suspect it then the regency continues.
4:
5: int leader-election.getLeader() : return getRegency() %p;
6:
7: void leader-election.suspect(int regency)
8: // indicates suspicion of the leader for "regency".
9: // if a quorum of correct nodes suspect the same regency r,
10: // then a new regency will start
11:
12: void leader-election.consider(proof)
13: // consider outside evidence that a new leader was elected
```

In the common case FaB is very simple and terminates in two steps. Table 4 shows the variables that use and Algorithm 7 shows the protocol's pseudocode. The *pnumber* variable (proposal number) indicates which process is the leader; in the common case, its value wiil not change. The code starts executing in the `onStart` methods. In the first step the leader proposes its value to all acceptors (line 3). In the second step, the acceptors accept this value (line 23) and forward it to the learners (line 24). Learners learn a value v when they observe that $\lceil (a+3f+1)/2 \rceil$ acceptors have accepted the value (line 28). In the common case, the timeout at line 13 will never trigger.

The leader election interface is given in Algorithm 10 FaB avoids digital signatures in the common case because they are computationally expensive. Adding signatures would reduce neither the number of communication steps nor the number of servers since FaB is already optimal in these two measures.

6.2 Fair Links and Retransmissions

Now, we assumed asynchrony. While synchrony is a reasonable assumption in the common case, protocol must also be able to handle periods of asynchrony. Note that now consensus may take more than two communication steps to terminate, e.g. when all messages sent by the leader in the first round are dropped.

End-to-end retransmission policy is based on the following pattern: the caller sends its request repeatedly, and the callee sends a single response every time it receives a request. When the caller is satisfied by the reply, it stops retransmitting. For protocol it looks: other processes must be able to determine whether the leader is making progress, and therefore the leader must make sure that they, too, receive the reply. To that end, learners report not only to the leader but also to the other proposers (Algorithm 7 line 29). When proposers receive enough acknowledgments, they are "satisfied" and notify the leader (line 9). The leader only stops resending when it receives $\lceil (p+f+1)/2 \rceil$ such notifications (line 4). If proposers do not hear from $\lceil (l+f+1)/2 \rceil$ learners after some time-out, they start suspecting the leader (line 14). If $\lceil (p+f+1)/2 \rceil$ proposers suspect the leader, then a new leader is elected. The retransmission policy therefore ensures that in periods of synchrony the leader will retransmit until it is guaranteed that no leader election will be triggered. Note that the proposers do not wait until they hear from all learners before becoming satisfied (since some learners may have crashed). It is possible therefore that the leader stops retransmitting before all learners have learned the value. To ensure that eventually all correct learners do learn the value, lines 31–43 of the protocol require all correct learners still in the dark to pull the value from their peers.

Algorithm 7 Byzantine Fast Paxos (excluding recovery)

```
1: function LEADER.ONSTART():
2:     //proposing (PC is null unless recovering)
3:     SEND(PROPOSE, value, pnumber, PC) to all acceptors
4:     until |Satisfied| ≥ ⌈(p + f + 1)/2⌉;
5:
6: function PROPOSER.ONLEARNED(): from learner l
7:     Learned := Learned union {l};
8:     if |Learned| ≥ ⌈(l + f + 1)/2⌉ then
9:         SEND(SATISFIED) to all proposers;
10:
11: function PROPOSER.ONSTART():
12:     wait for timeout
13:     if |learned| < ⌈(l + f + 1)/2⌉ then
14:         leader-election.suspect(leader-election.getRegency());
15:
16: function PROPOSER.ONSATISFIED(): from proposer x
17:     Satisfied := Satisfied ∪ {x};
18:
19: function ACCEPTOR.ONPROPOSE(value, pnumber, progcert):
20: from leader
21:     if not already accepted then
22:         accepted := (value, pnumber); // accepting
23:     SEND (ACCEPTED, accepted) to all learners
24:
25: function LEARNER.ONACCEPTED(value, pnumber): from ac-
    ceptor ac
26:     accepted[ac] := (value, pnumber);
27:     if there are ⌈(a+3f+1)/2⌉ acceptor x that accepted[x] ==
    (value, pnumber) then
28:         learned := (value, pnumber); // learning
29:         SEND (LEARNED) to all proposers;
30:
31: function LEARNER.ONSTART(()):
32:     wait for timeout
33:     Upon not learned do
34:         SEND(PULL) to all learners;
35:
36: function LEARNER.ONPULL(): from learner ln
37:     if this process learned some pair (value, pnumber) then
38:         SEND (LEARNED, value, pnumber) to ln
39:
40: function LEARNER.ONLEARNED(value, pnumber): from learner
    ln
41:     Learn[ln] := (value, pnumber);
42:     if (there are f + 1 learners x such that learn[x] ==
    (value, pnumber)) then
43:         learned := (value, pnumber) // learning
```

6.3 Recovery protocol

Recovery occurs when the leader election protocol elects a new leader. Although can reuse existing leader election protocols as-is, it is useful to go through the properties of leader election. The output of leader election is a regency number r. This number never decreases, and we say that proposer $r \ mod \ p$ is the leader. Each node in the system has an instance of a leader-election object, and different instances may initially indicate different regents. Nodes indicate which other nodes they suspect of being faulty; that is the input to the leader election protocol. If no more than f nodes are Byzantine and at least $2f+1$ nodes participate in leader election, then leader election guarantees that if no correct node suspects the current regent, then eventually:

(i) all leader-election objects will return the same regency number;

(ii) that number will not change.

Leader election also guarantees that if a quorum of correct nodes ($\lceil (p+f+1)/2 \rceil$ nodes out p) suspect regent r, then the regency number at all correct nodes will eventually be different from r. Finally, leader election also generates a $proof_r$ when it elects some regent r. If $proof_r$ from a correct node is given to a leader-election object o, then o will elect regency $r', r \leq r'$.

Back to the leader election interface Algorithm 10, `getRegency()` returns the current regency number, and `getLeader()` converts it to a proposer number. Nodes indicate their suspicion by calling `suspect(`r`)`. When leader-election elects a new leader, it notifies the node through the `onElected(`$regency, proof_r$`)` callback (not shown). If necessary, $proof_r$ can then be given to other leader-election objects through the `consider(`$proof_r$`)` method.

When proposers suspect the current leader of being faulty, they trigger an election for a new leader who then invokes the recovery protocol. There are two scenarios that require special care:

1. Some value v may have already been chosen: the new leader must then propose the same v to maintain CS2.

2. A previous malicious leader may have a *poisonous write*, i.e. a write that prevents learners from reading any value — for example, a malicious leader could propose a different value to each acceptor. If the new leader is correct, consensus in a synchronous execution should nonetheless terminate.

If system maintained the requirement that acceptors must only accept the first value they receive, the new leader would be unable to recover from poisonous write. So, acceptors can change their mind and accept multiple values, but system must take precautions to ensure that CS2 still holds.

Algorithm 8 Byzantine Fast Paxos – recovery

```
1: function LEADER.ONELECTED(newnumber, proof):
2:     // this fuction is called when leader-election picks a new regency
3:     // proof is a piece of data that will sway leader-election at the
4:     // other nodes.
5:     pnumber := newnumber // no smaller than the previous pnumber
6:     if not leader for pnumber then
7:         return;
8:     SEND (QUERY, pnumber, proof) to all acceptos
9:     until GET (REP, signed(value, pnumber)) from a − f acceptors
10:    PC := the union of these replies
11:    if PC vouches for (v', pnumber) then
12:        value := v';
13:    OnStart();
14:
15: function ACCEPTORS.ONQUERY(pn, proof): from leader
16:    leader-election.consider(proof)
17:    if leader-election.getRegency() ! = pn then
18:        return; // ignore bad requests
19:    SEND(REP, signed(value, pn)) to leader-election.getLeader()
20:
21: function ACCEPTOR.ONPROPOSE(value, pnumber, progcert): from proposer
22:    if pnumber ≠ leader-election.getRegency() then
23:        return; //only listen to current leader
24:    if accepted(v, pn) and pn == pnumber then
25:        return; //only once per prop. number
26:    if accepted(v, pn) and v ≠ value and progcert does not vouch for (value, pnumber) then
27:        return; //only change with progress certificate
28:    accepted := (value, pnumber) // accepting
29:    SEND (ACCEPTED, accepted) to all learners
```

6.4 Parametrized FaB Paxos

Parameterized FaB Paxos is an algorithm, that spans the whole design space between minimal number of processes (but no guarantee of two-step executions) and two-step protocols (that require more processes). This trade-off is expressed through the new parameter $t, (0 \leq t \leq f)$. Parameterized FaB Paxos requires $3f + 2t + 1$ processes, is safe despite up to f. Byzantine failures, and all its executions are two-step in the common case despite up to t Byzantine failures. So, FaB Paxos is just a special case of Parameterized FaB Paxos, with $t = f$.

Several choices of t and f may be available for a given number of machines. For example, if seven machines are available, an administrator can choose between tolerating two Byzantine failures and slowing down after the first failure ($f = 2, t = 0$) or tolerating only one Byzantine failure but maintaining two-step operation despite the failure ($f = 1, t = 1$).

The Parameterized FaB Paxos code does not include any mention of the parameter t: if there are more than t failures, the two-step feature of Parameterized FaB Paxos may never be triggered because there are not enough correct nodes to send the required number of messages.

But, it contains three modification. First, acceptors modification so that, after receiving a proposal, they sign it (including the proposal number) and forward it to each other so each of them can collect a commit proof. A *commit proof* for values v at proposal number pn consist of $\lceil (a + f + 1)/2 \rceil$ statements from different acceptors that accepted value v for proposal number pn (function valid, Algorithm 9). The purpose of commit proofs is to give evidence as to which value was chosen. If there is a commit proof for value v at proposal pn, then no other value can possibly have been chosen for proposal pn. In Algorithm 9 commit proofs includes in the progress certificates so that newly elected leaders have all the necessary information when deciding which value to propose. The commit proofs are also forwarded to learners to guarantee liveness when more than t acceptors fail. Second, learners modification so that they learn a value if enough acceptors have a commit proof for the same value and proposal number. And last, redefine "chosen" and "progress certificate" to take commit proofs into account. Value v is *chosen for* proposal number pn if $\lceil (a+f+1)/2 \rceil$ correct acceptors have accepted v in proposal pn or if $\lceil (a+f+1)/2 \rceil$ acceptors have (or had) a commit proof for v when they know v has been chosen. The protocol ensures that only that only a single value may be chosen. *Progress certificates* still consist of $a - f$ entries, but each entry now contains an additional element: either a commit proof or a signed statement saying that the corresponding acceptor has no commit proof. A progress certificate *vouches* for value v' at proposal number pn if all entries have proposal number pn, there is no value $d \neq v'$ contained $\lceil (a - f + 1)/2 \rceil$ times in the progress certificate, and the progress certificate does not contain a commit proof for any value $d \neq v'$ (function "vouches-for", Algorithm 9). The purpose of progress certificates is, as before, to allow learners to convince acceptors to change their accepted value.

Algorithm 9 Byzantine Parametrized FaB Paxos

```
1: function LEADER.ONSTART():
2:     //proposing (PC is null unless recovering)
3:     SEND(PROPOSE, value, number, PC) to all acceptors
4:     until |Satisfied| ≥ ⌈(p + f + 1)/2⌉;
5:
6: function LEADER.ONELECTED(newnumber, proof):
7:     pnumber := newnumber // no smaller than the
8: previous pnumber
9:     if not leader for pnumber then
10:        return;
11:    SEND (QUERY, pnumber, proof) to all acceptos
12:    until GET (REP, ⟨value, pnumber, commit_proof, j⟩)
13: from a − f acceptors
14:    PC := the union of these replies
15:    if ∃v′ s.t. vouches-for(PC, v′, pnumber) then
16:        value := v′;
17:    OnStart();
18:
19: function PROPOSER.ONLEARNED(): from learner l
20:    Learned := Learned union {l};
21:    if |Learned| ≥ ⌈(l + f + 1)/2⌉ then
22:        SEND(SATISFIED) to all proposers;
23:
24: function PROPOSER.ONSTART():
25:    wait for timeout
26:    if |learned| < ⌈(l + f + 1)/2⌉ then
27:        suspect the leader;
28:
29: function PROPOSER.ONSATISFIED(): from proposer x
30:    Satisfied := Satisfied ∪ {x};
31:
32: function ACCEPTOR.ONPROPOSE(value, pnumber, progcert):
33: from leader
34:    if pnumber ≠ leader-election.getRegency() then
35:        return;
36:    if accepted(v, pn) and ((pnumber ≤ pn) or
37: (v ≠ value)) and not
38: vouches-for(progcert, value, pnumber) then
39:        return;
40:    accepted := (value, number); // accepting
41:    SEND (ACCEPTED, accepted) to all learners
42:    //i is the number of this acceptor
43:    SEND ⟨ACCEPTED, value, pnumber, i⟩_i
44: to all acceptors
45:
46: function ACCEPTOR.ONACCEPTED(value, pnumber, j):
47:    if pnumber > tentative_commit_proof[j].pnumber then
48:        tentative_commit_proof[j] :=
49: ⟨ACCEPTED, value_j, pnumber, j⟩_j;
50:    if valid(tentative_commit_proof, value,
51: leader-election.getRegency()) then
52:        commit_proof := tentative_commit_proof;
53:        SEND(COMMITPROOF, commit_proof) to
54: all learners;
55:
56: function ACCEPTORS.ONQUERY(pn, proof): from proposer
57:    leader-election.consider(proof)
58:    if leader-election.getRegency() ! = pn then
59:        return; // ignore bad requests
60:    leader := leader-election.getLeader();
61:    SEND(REP,⟨ accepted.value, pn, commit_proof, i ⟩_i) to
leader
62:
63: function LEARNER.ONACCEPTED(value, pnumber): from ac-
ceptor ac
64:    accepted[ac] := (value, pnumber);
65:    if there are ⌈(a + 3f + 1)/2⌉ acceptor x such that
accepted[x] == (value, pnumber) then
66:        learn(value, pnumber); // learning
67:
68: function LEARNER.ONCOMMITPROOF(commit_proof): from ac-
ceptor ac
69:    cp[ac] := commit_proof;
70:    (value, pnumber) := accepted[ac]
71:    if there are ⌈(a + f + 1)/2⌉ acceptor x such that
valid(cp[x], value, pnumber) then
72:        learn(value, pnumber); // learning
73:
74: function LEARNER.LEARN(value, pnumber):
75:    learned := (value, pnumber);
76:    SEND (LEARNED) to all proposer
77:
78: function LEARNER.ONSTART(()):
79:    wait for timeout
80:    While(not learned) SEND(PULL) to all learners;
81:
82: function LEARNER.ONPULL(): from learner ln
83:    if this process learned some pair (value, pnumber) then
84:        SEND (LEARNED, value, pnumber) to ln
85:
86: function LEARNER.ONLEARNED(value, pnumber): from learner
ln
87:    Learn[ln] := (value, pnumber);
88:    if (there are f + 1 learners x such that learn[x] ==
(value, pnumber)) then
89:        learned := (value, pnumber);
90:
91: function VALID(commit_proof, value, pnumber):
92:    c := commit_proof;
93:    if there are ⌈(a + f + 1)/2⌉ distinct values of x such that
(c[x].value) == value and c[x].pnumber == pnumber then
94:        return true;
95:    else
96:        return false;
97:
98: function VOUCHES-FOR(PC, value, pnumber):
99:    if there exist ⌈(a − f + 1)/2⌉ x such that all
PC[x].value == d and d ≠ value then
100:       return false;
101:   if there exist x, d ≠ value such that
valid(PC[x].commit_proof, d, pnumber) then
102:       return false;
103:    return true;
```

7 Kursawe's Optimistic Byzantine Agreement

Optimistic Byzantine Agreement (or OBA) [Kur02] – it is a protocol, which is very close to Byzantine Fast Paxos. The main idea of OBA is to use different parts of it depending on the conditions that may affect the system. I.e., it uses *optimistic* protocol (part) when system is not under attack and the environment is not behaving in a hostile manner. But, if the environment turns out to be less friendly, the performance is slightly lower compared to non-optimistic protocols, but security remains unaffected. In practice, it works as follows: protocol tries to reach agreement by optimistic phase, believing that the system works in a friendly environment and tests the results; if any inconsistencies are detected, system invokes the asynchronous, reliable fallback protocol. Note that, it is possible while some parties decide in the optimistic part of the protocol and another in the fallback; in this case the protocol guarantees that agreement holds anyway.

7.1 Model and Problem Setting

Let's describe the model in detail: there are n parties $P_1, ..., P_n$ up to f, $f < \frac{n}{3}$ of which may be corrupted by an adversary and might behave arbitrarily maliciously. Model uses "time", however, is very well implementable in a fully asynchronous system, unlike failure detectors. Out timeout corresponds to a failure detector that only needs to satisfy completeness, but not accuracy; in system it "implements" like the timeout mechanism. To this end, each protocol instance has a special state variable *timer*, which can take on the values *stopped* or *running*. Initially, the timer is *stopped*. A thread may change this value by executing **start timer** or **stop timer** commands, or inspect the value of the timer. The **start timer** command is implemented by sending a unique message to oneself, and the adversary delivers a timeout signal by delivering this messages. When this happens, the timer is stopped, and if that thread is waiting on a *timeout*, the thread is activated. In formal model, it effectively makes no assumptions about the accuracy of the clock, except that the clock runs forward. In a real-world implementation, where timeout mechanism use real clock, there is some hope, however, that some timing properties hold most of the time. Essentially, it states that if a protocol instance waits for a (sub)protocol to either terminate or timeout, it will usually terminate before timeout

Byzantine Agreement. A party is activated for a particular instance ID of the Byzantine agreement protocol by receiving a message (*ID, PROPOSE, initial value*), where *initial value* is $\{0, 1\}$. Beside normal messages, it may send a message(*ID, DECIDE, final value*). In this case, P_i *decides final value* for ID. P_i may make a decision for a given ID at most once. However, the adversary may continue to deliver messages involving ID after P_i has made a decision for ID. Exist three basic properties that an agreement protocol must satisfy:

1. **Agreement.** Any two honest parties that decide on a value for a particular ID must decide on the same value. More precisely, it is computationally *infeasible* for an adversary to make two honest parties decide on different values.

2. **Validity.** If all honest parties have been activated on a given ID with the same initial value, then all honest parties that decide must decide on this value.

3. **Termination.** Since it is not possible to use the traditional definition of termination, model constructs computation time in terms of messages sent, and split termination into two conditions: *deadlock freeness* and *efficiency:*

   (a) *Deadlock freeness.* It is infeasible for the adversary to create a situation where for some ID there are some honest parties who are not decided, yet all parties have been activated on ID, and all messages relating to this ID generated by honest parties have been delivered.

   (b) *Efficiency.* For every ID, the communication complexity for ID is probabilistically uniformly bounded.

For the pessimistic phase model uses efficient randomized protocol.

Returning to the description of the protocol phases, logically that the optimistic phase does not need to terminate with a decision. In the optimistic case an agreement is reached in optional time. Furthermore, no expensive computation (for example, due to public key cryptography) is needed. In pessimistic case, when optimistic pre-protocol invokes the fallback protocol, if few failures occur, the fallback protocol is invoked with all honest parties having the same start value. This causes the protocol to terminate quite fast.

7.2 The protocol

Each party P_i, $1 \leq i \leq n$, gets an input value v_i and a corresponding transaction identifier ID. The protocol outputs some decision value p or invokes the fallback protocol BA. While a party decides p in the optimistic protocol and still invokes the fallback protocol, the decision of BA is ignored. To ensure validity, if any honest party decides p in the optimistic part of the protocol, then the decision in BA can only be p.

Optimistic-BA works in four phases:

* **Simple Agreement:** A simple non-Byzantine agreement is invoked by every party broadcasting its preference and waiting for all other parties to answer.

* **Commit and Check for Decision:** Every party commits to the value it perceives as the output of the simple agreement protocol. If it receives n identical commitments from all parties, then it can *decide*.

* **Decide and Hibernate:** A party that decided cannot completely terminate the protocol, as it is possible that some other party could not reach a decision. Instead, decided parties hibernate; they remember their decision for this transaction, but do nothing unless a complaint is received.

* **Complain and Recover:** A party that gets either inconsistent or not enough answers broadcasts a complaint. On receiving such a complaint. an honest party starts the "pessimistic part" of the protocol, i.e., enters the fallback protocol BA.

Algorithm 10 Optimistic-Byzantine Agreement

```
1: //Vote
2: Upon receiving a message(in, propose, v_i) do
3:     start timer
4:     send to all parties the message (init − vote, v_i)
5: //Commit
6: Upon receiving a timeout signal from the timer or n init-votes do
7:     if no commit message has been sent by P_i then
8:         v_i^commit is a simple majority, if n votes where collected of the init votes
9:     else
10:        v_i^commit = v_i
11:    send to all parties the message (commit, v_i^commit);
12:    start timer
13:
14: //CheckforDecision
15: Upon receiving n commitments do
16:    stop timer
17:    output(out, decide, p)
18:
19: //FallBacktoPessimisticProtocol
20: Upon receiving a timeout signal from the timer do
21:    if a commit message has been sent then
22:        send to all parties the message (pessimism)
23: Upon receiving a pessimism message do
24:    if P_i did not broadcast a pessimism-message yet then
25:        send to all parties the message (pessimism)
26:    invoke BA on transaction ID with input value v_i^commit
```

Given a protocol BA for Byzantine agreement as a fallback protocol, if the adversary faithfully delivers the timeout signals, the optimistic protocol solves Byzantine agreement for $n > 3f$, subject to the restrictions of BA.

8 Results

By analysing the algorithms, the main summary is that we can reduce the number of processes but, for correct work, we need to do more rounds to provide the BFT property. This statement explains, that we should to balance and, if model allows that, use modifications for optimization. So now, let's explain a short idea of each protocols:

1. The BPCon is a model how we can simple modify Paxos to Byzantine Paxos. It has a classic two-phase structures which does not differ from the classic Paxos except two steps (1c and 2av (ex. 2a)).

2. The CLA is a refined BPCon that contains engineering optimizations, has a simplified form.

3. The BGP is a model that provides a two communication steps. Have a little bit difficult implementation, but provides less costs on communication. It has one interesting property – it allows to choose an increasing sequence of commands, not only one value. Also protocol maintain two mode of ballot: fast and classic. Classic ballots look like an original Paxos ballot, when the leader communicates with other processes in a system. Fast ballots mean that proposers can send their proposal directly to the acceptors without a leader and, if there is no conflict at this stage, they can fix their result. Note, that leader must construct messages for propose within the rules.

4. The BVP is a model that has two modes: a simple steady-state protocol and reconfiguration protocol. The system requires a reconfiguration for changing replicas or leader which use wedge scheme mechanism. Note that for reconfiguration consensus decision it requires separate Byzantine consensus engine.

5. The FaB is a model that requires only two communication steps at a price — $5f + 1$ acceptors to tolerate f Byzantine acceptors. Similar to other algorithms, it has two modes: common case and recovery. Recovery is used when the leader election protocol elects a new leader and support system for new rounds of voting.

6. The PFaB is a model that includes a new parameter t, which is a trade-off between the min number of processes and the two-step protocol. But mentioned implementation does not use t, because if it has more than t failures, the two-step feature of PFaB may never be triggered. But it contains three modifications which describing how the system will work when acceptors sign proposal, forward it to each other and then start collecting a commit proof.

7. The OBA is a model that uses two modes: optimistic phase, when the system is not under attack. In this case, the system tries to decide some value and tests the results. If something goes wrong, the system invokes the pessimistic mode, that can solve the consensus problem in any case. But, pessimistic mode require expensive computation, like a public-key signature, and logically more than 2 rounds of communication (in assumption – 4).

Algorithm	***Number of process***	***Number of rounds***	***Features***
BPCon	$3f+1$	2	Use Paxos-shaped scheme.
CLA	$3f+1$	2	Modyficated BPCon with engineering optimizations
BGP	$3f+1$	2	Allow to accept increasing sequence of command; anyone can be a leader; have two ballot options: classic and fast(without leader).
BVP	$f+1 or 2f+1$	4 or 3	Have two modes: steady state and reconfiguration protocols; use wedjing scheme.
FaB	$5f+1$	2	Requires the highest number of process; have two cases of ballots: common and recovery(when leader election); not use digital signature in common case, but rely when electing a new leader.
PFaB	$3f+2t+1$	2	Include new parameter t, that trade-off between min number of process and two-steps protocol.
OBA	$3f+1$	2 in optimistic case, 4 – otherwise	Have two phase: optimistic (fast agreement, work only in friendly environment) and pessimistic (if environment is malicious, should require public-key signatures)

Table 5: Results — Algorithm comparison

Algorithm	Leader -> Acceptors	Leader -> Proposers	Acceptors -> Acceptors	Acceptors -> Leader	Acceptors -> Learners	Proposers -> Proposers	Learners -> Proposers	Learners -> Learners	Proposers-> Leader	Acceptor-> Proposers	Total number
BPCon, $N \geq 3f+1$	$2N$	-	-	$3N$	-	-	-	-	-	-	$5N$
CLA, $N \geq 3f+1$	$2N$	-	-	$2N$	-	-	-	-	-	-	$4N$
BGP, $N \geq 3f+1$	$(2+p)N$	$(1+p)P$	N^2	N	$2NL$	-	-	-	P	-	$(3+p)N+N^2$ $+2NL+(2+p)P$
BVP, $N \geq f+1$ or $2f+1$	N	-	N^2	-	-	-	-	-	1	N (only in 4-rounds)	4-rounds: N^2+2N 3-rounds: N^2+N
FaB, $N \geq 5f+1$, $P, L \geq 3f+1$	N	-	-	-	NL	P	LP	$2L^2$	-	-	L^2+P+ $LP+N(1+L)$
PFaB, $N \geq 3f+2t+1$	N	-	N^2	-	$2NL$	P	LP	$2L^2$	-	-	N^2+2L^2+P+ $LP+N(2L+1)$
OBA, $N \geq 3f+1$	N	-	-	N	-	-	-	-	-	-	$2N$

Table 6: Number of messages in a successful scheme, where N – number of acceptors, P – number of proposers, L – number of learners, p – number of proposals.

9 Conclusions

Comparing the algorithms with each other, as well as considering the future prospects of implementation, we prefer the BPCon algorithm. The rationale for this choice is as follows: canonical implementation with two phases and optimal number of processes, a simplified communication between system processes, proximity to well-known BFT algorithms.

We consider that usage two-phase algorithm in conjunction with $3f+1$ processes is the most optimal scheme, that can provide essential properties of robustness. Of course, this does not mean that algorithm with more than 2 rounds are bad, but in our case, BVP has worse implementation than BPCon. In addition to that, FaB (and PFaB) also have worse implementation by requiring $5f+1$ processes, that will slow down the realization.

Another important item pertains to number of messages that system requires from all processes. In BPCon by using $1c$ step, acceptors do not need to send message to other acceptors about their pre-choice. But BGP require that N acceptors send N messages. Also this algorithm uses additional communication with proposers and learners. It might be good property while system needs to propose a lot of values concurrently (BGP supports vote for a set of commands), but not optimal in the classic case.

Moreover that BPCon is quite similar to PoS's BFT algorithms in main steps, for example Tendermint [Kwo14], Catchain [Dur20], HotStuff [Yin+19], GRANDPA [SK20], RBFT [AMQ13], etc. The general idea of these algorithms that system has three successive steps: prevote, precommit and commit. In detail about BPCon, acceptors send that they confirm the value from leader(prevote). The leader sends acceptors pre-vote that he received (precommit). And after that, if acceptors do not need to launch the

$2av$ phase, they vote for values (commit). In Tendermint and GRANDPA nodes receive messages from each other at the prevote step, then do the same at the precommit step and, if we consider the optimistic situation, they send and receive a commit as a result of consensus. RBFT, Catchain and HotStuff look similar, but they involve more explicit communication with the leader.

References

[Kur02] Klaus Kursawe. "Optimistic Byzantine agreement". In: 2002. DOI: 10.1109/RELDIS.2002.1180196.

[MA06] Jean Philippe Martin and Lorenzo Alvisi. "Fast Byzantine consensus". In: vol. 3. 2006. DOI: 10.1109/TDSC.2006.35.

[LMZ09] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. "Vertical paxos and primary-backup replication". In: 2009. DOI: 10.1145/1582716.1582783.

[Lam11] Leslie Lamport. "Byzantizing Paxos by refinement". In: vol. 6950 LNCS. 2011. DOI: 10.1007/978-3-642-24100-0_22.

[AMQ13] Pierre Louis Aublin, Sonia Ben Mokhtar, and Vivien Quema. "RBFT: Redundant byzantine fault tolerance". In: *Proceedings - International Conference on Distributed Computing Systems*. 2013. DOI: 10.1109/ICDCS.2013.53.

[Kwo14] Jae Kwon. *TenderMint : Consensus without Mining*. 2014.

[PRR18] Miguel Pires, Srivatsan Ravi, and Rodrigo Rodrigues. "Generalized Paxos Made Byzantine (and Less Complex)". In: *Algorithms* 11 (9 2018). ISSN: 19994893. DOI: 10.3390/a11090141.

[Yin+19] Maofan Yin et al. "HotStuff". In: Association for Computing Machinery (ACM), July 2019, pp. 347–356. DOI: 10.1145/3293611.3331591.

[Dur20] Nikolai Durov. *Catchain Consensus: An Outline*. Feb. 19, 2020.

[SK20] Alistair Steward and Eleftherios Kokoris-Kogia. *GRANDPA: A Byzantine Finality Gadget*. June 19, 2020.