ANALYSIS OF EFFICIENCY AND SECURITY OF EXISTING BFT PAXOS-BASED ALGORITHMS

Illia Melnyk [1,2], Oleksandr Kurbatov [2], Oleg Fomenko [2], Volodymyr Dubinin [2] and Yaroslav Panasenko [2]

[1] National Technical University of Ukraine, "Igor Sikorsky Kyiv Polytechnic Institute"
[2] Distributed Lab, Kyiv, Ukraine

ABSTRACT

Byzantine fault tolerant consensus plays a critical role in maintaining the reliability of distributed systems. This paper surveys and evaluates five Paxos-based algorithms – Byzantine classic Paxos consensus, Castro-Liskov algorithm, Byzantine generalized Paxos consensus, Byzantine vertical Paxos, and Optimistic Byzantine Agreement – comparing their efficiency in terms of process requirements, communication rounds, and message complexity, as well as their resilience against Byzantine behaviors. Through detailed examination of protocol structures and performance trade-offs, we identify the strengths and limitations of each approach under typical and adversarial conditions. Our analysis reveals that the two-phase Byzantine classic Paxos consensus protocol achieves an optimal balance of simplicity, low process overhead, and robust security guarantees, making it a compelling choice for practical Byzantine fault tolerant deployments. We conclude with recommendations for selecting an appropriate consensus algorithm based on system constraints.

KEYWORDS

Byzantine fault tolerance, Paxos-based consensus, Communication complexity, Process requirements, Distributed system security

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. The goal of this paper is to determine the optimal algorithm from the Paxos family for selecting a coordinator in the Threshold Signature Scheme (TSS), providing the Byzantine fault tolerance (BFT) property. Some of these schemes require signers to elect a specific signer as a coordinator to facilitate both the distributed key generation and signing round steps of the protocol. Since signers are distributed and mutually distrustful, TSS coordinator selection is fundamentally a Byzantine setting. Note that not only can individual signers be faulty, but also the coordinator may be faulty.

David C. Wyld et al. (Eds): CRYPIS, CBIoT, CAIML, NLCA, NC, WiMo, ICAIT, ICDIPV, ITCSE – 2025
pp. 23-40, 2025. CS & IT - CSCP 2025 DOI: 10.5121/csit.2025.151702

In general, Paxos is considered a complex and abstract algorithmic concept. It is often mentioned in contrast to the simpler consensus algorithm Raft. However, Raft is a leader-centric scheme that cannot tolerate node or partition failures. Therefore, we chose to study the Paxos family of algorithms, which are considered a reliable classical model of consensus protocols.

Our Contribution and Results. First, we collected the five most promising Paxos BFT algorithms, which are detailed in Table 1 of Section 1.1. Section 2 details the foundations of the original Paxos. In Sections 3-6, we provide a detailed description of each BFT algorithm. We will start with the BPCon and Castro-Liskov algorithms, which are the closest to the original scheme and offer a balance of simplicity and efficiency. Next is Byzantine Generalized Paxos, which allows weakening security conditions under certain circumstances. It is considered a complex modification of Paxos, but maintains efficiency. Vertical Paxos is a more specific algorithm that allows changing consensus modes depending on the situation, but requires more rounds of exchanges. Kursawe‘s Optimistic Byzantine Agreement also uses different modes depending on the conditions that may affect the system. This protocol is very similar to Byzantine Fast Paxos, which we did not include in our work. Although Fast Paxos is considered quite simple, it requires a large number of honest participants compared to others. We summarize our findings in Section 7. Table 6 shows the most extensive comparison of all algorithms by the number of messages.

Table 1. A summary of the algorithms that we compared. By f we denote the number of Byzantine processes.

Algorithm	Number of process	Number of rounds	Features
BPCon	$f+1$	2	Paxos-shaped scheme.
CLA	$3f+1$	2	Modified BPCon with engineering optimizations
BGP	$3f+1$	2	Allows to accept increasing sequence of command; anyone can be a leader; has two ballot options: classic and fast(without leader).
BVP	$f+1$ or $2f+1$	4 or 3	Has two modes: steady state and reconfiguration protocols; uses wedging scheme
OBA	$3f+1$	2 in optimistic case, 4 - otherwise	Has two phase: optimistic (fast agreement, works only in a friendly environment) and pessimistic (if environment is a malicious, should require public-key signatures)

1.1. Related Work

Our study focuses on comparing the BFT Paxos algorithms. Previous works, however, compared Paxos only with other consensus algorithms or focused on a specific protocol.

One of the earliest mentions of BFT properties in the context of Paxos appears in the work of Castro and Liskov[1]. The authors note that their BFT algorithm can be viewed as a modification of Paxos in many aspects. Around the same time, Lamport formally described the Byzantine Paxos and later published the BPCon algorithm [2].

Other works have appeared later [3, 4, 5, 6], typically proposing a scheme where there is a fast consensus option under optimistic conditions and a standard one that includes cases where nodes (processes) may be faulty or byzantine.

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 procedure 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 whom may fail 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 as follows:

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

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

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

Upon receiving 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. Let c be the highest-numbered ballot less than b in which some acceptor voted. 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.

2.1.1. Pcon Scheme

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:

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 values.

The leader sends a 2a message containing the value for which a 1c message was previously sent. Note that it also requires the use of the following properties for choosing a 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 [2] is a consensus algorithm that has $N \geq 3f + 1$ acceptors, f of which are byzantine.

Let's define the set of $byzacceptors$ as the union of the byzantine, and honest acceptors, and $byzquorum$ as a subset of $byzacceptors$ that always contains a quorum of acceptors. If the quorum consists of any q acceptors, then a $byzquorum$ consists of any $q + f$ $byzacceptors$. For liveness, it is assumed that the set of all honest acceptors (which are assumed to never fail) forms a $byzquorum$. This assumption guarantees that even in the worst-case scenario, the system still has at least q real acceptors that can complete the voting phase and ensure the progress of the protocol.

3.1. BPCon's Phases

At first, consider the difference between BPCon and non-byzantine Paxos:

1. There is no explicit $2a$ Phase action. Instead, acceptors cooperate to emulate sending a $2a$ message. The ballot-b leader requests that a Phase 2a action be performed for a value v that was already proposed in one of the $1c$ messages. When an acceptor receives the first such request, it performs a Phase $2av$ action and sends a ballot-b $1c$ message with that value.

2. The algorithm tolerates malicious leaders, meaning that the ballot-b leader may send arbitrary $1a$ and $1c$ messages. The BPCon Phase 1c action allows the ballot-b leader to send any ballot-b $1c$ message at any time. It is essential that acceptors only accept a $1c$ message if it is legal. A nonfaulty leader must send a message that prompts (honest) acceptors to act, to guarantee liveness.

3. In non-byzantine Paxos, a ballot-b $1c$ message can be sent only after collecting a set of $1b$ messages from a quorum and possibly a $1c$ message (follows from the properties). In BPCon, additional information is put in the $1b$ messages 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 is sent in response to a legal $1c$ message.

Also, BPCon should provide the following 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. This message contains either the number of the highest-numbered ballot in which it has voted and the value it voted for in that ballot, or a statement 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.

$(2c)$ 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 liveness requirements of BPCon are the same as non-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 another. However, it is difficult to ensure that a Byzantine leader does not execute a higher-numbered ballot. This requires an engineering solution based on real-time assumptions, such as the one proposed by Castro and Liskov. To guarantee liveness of BPCon under these requirements, the following two conditions must be satisfied:

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

$(1c)$ When a view change occurs, the new-view message replaces $1c$ messages for all 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 other instances – a set of $1c$ messages for all values (satisfying BP3a). Acceptors validate the corresponding $1c$ messages concurrently for all instances.

$(2av)$ This is a backup's prepare message. The primary's pre-prepare message serves both as its $2av$ message and as the implicit request (not modeled in *BPCon*) to perform a Phase 2a action.
$(2b)$ This is the commit message.

4. BGP (Byzantine Generalized Paxos)

Byzantine Generalized Paxos consensus (or BGP) [6] – modification of one of the most recent algorithms of the Paxos family. Generalized Paxos allows weakening the security conditions when different processes may have different views of the agreed sequence.

BGP includes two subprotocols: *View-change* and *Agreement*. View-change ensures that at any given moment, a leader can be elected from one of the proposers by continuously replacing leaders until one that can ensure progress is found. Then, the chosen leader runs the Agreement, which extends the current sequence with a new command or a set of them. It is important to note that there are two ways to choose this extension: using either classic ballots, as in the original Paxos, or fast ballots which leverage a weaker specification of generalized consensus.

Very briefly, here is the model and simplifications that [6] used to build the protocol. It is an asynchronous system in which a set of $n \in \mathbb{N}$ processes communicate by *sending* and *receiving*

messages. Processes may fail if they crash and stop executing the algorithm, or if they do not follow an algorithm, which means they are Byzantine. It is considered that a non-Byzantine process is correct. The processes may have the following roles: *learner*, *proposer* or *acceptor*. Proposers submit input values to be agreed upon by learners, while acceptors assist learners in reaching consensus on a single value. BGP requires the condition $N \geq 3f + 1$, where N is the minimum number of acceptor processes, and f is the maximum number of tolerated Byzantine faults.

In a simplified specification, each learner l maintains a monotonically increasing sequence of commands $learned_l$. To operate on them, a C-struct is usually used, which provides an abstraction of increasing sequences and allows defining various consensus problems.

Two learned sequences of commands are defined as equivalent (~) if one of them can be transformed into the other by rearranging the elements in such a way that the order of non-commutative pairs is preserved. A sequence x is called an equiprefix of another sequence $y (x \sqsubseteq y)$ if the subsequence y containing all elements of x is equivalent (~) to x. Instead of using C-structures, BGP suggests using the matching of 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.
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.

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;

Figure 1. Pseudocode of the proposer p in the Byzantine Generalized Paxos algorithm.

4.1. View-Change

The aim of View-change is to elect a leader among the proposers who performs the agreement protocol. The system moves through sequentially numbered views, where the leader of each view is chosen in a rotational fashion using the equation: $leader(view) = view \bmod N$. To ensure continuous operation, acceptor processes monitor the progress of appending commands. If progress stalls, acceptor multicasts a signed suspicion message about the current leader to all

acceptors. An acceptor that collects $f + 1$ suspicions from other processes can multicast a view-change message consisting of the new view number and the suspicions. Once other acceptors receive and validate a view-change message, they can create and multicast their own.

Finally, an acceptor process must wait for $N - f$ view-change messages before it can start participating in the new view. This involves updating its view number and the corresponding leader process. The acceptor then assembles the $N - f$ view-change messages proving that others are committing to the new view and sends them to the new leader. Once the new leader validates the $N - f$ signatures contained in a single message, it can begin its leadership role in the new view.

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: $proposals = \perp$

Figure 2. Pseudocode of leader ℓ in the Byzantine Generalized Paxos algorithm.

4.2. Agreement Protocol

In BGP, each ballot represents an extension of the learned command sequence in a single consensus instance. Proposers may create proposals, consisting of either a single command or a sequence of commands.

Ballots can be either 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 begins the first phase by sending phase $1a$ messages to all acceptors. These messages contain the ballot number.

3. Similarly to classic Paxos, acceptors send all sequences of commands they voted for in a Phase $1b$ message to the leader. The leader needs to collect $N - f$ proofs from the Phase

1b messages that guarantee that, for any learned sequence s, at least one of the messages is from a correct acceptor that participated in the verification phase of s. In essence, for BGP, $f+1$ identical votes for a value ensure that at least one correct acceptor has voted. Therefore, the leader needs to collect $2f+1$ votes to determine the majority for some value. Note that Byzantine acceptors cannot change the value they are voting for because the proposer signs it. However, they may omit this value from the message $1b$.

4. Once the leader collects a quorum of $N-f$ Phase $1b$ messages, it proceeds to Phase $2a$, during which it prepares a proposal for the acceptors. This proposal sequence must be constructed in the following steps:

a) The leader initiates the proposal sequence with the longest proven among the $N-f$ Phase $1b$ messages. This selection can be made deterministically, since any two such sequences are either equivalent or one extends the other. This holds because, under the conditions described above, there is always at least one correct acceptor who votes for both of any two proven sequences. Importantly, a vote from a correct acceptor is always an extension of previous votes from the same ballot. If there are multiple sequences of equal maximum length, the leader may choose any one of them.

b) The second part of the sequence consists of a concatenation of unproven sequences or commands, arranged in an arbitrary order. Since a learner requires $N-f$ Phase $2b$ messages to learn a sequence, and since the intersection of the quorum and leader's quorum contains at least one correct acceptor, the leader can be confident that if a sequence is unproven in all received Phase $1b$ messages, then it has not yet been learned and may be safely appended to the leader's sequence in any order.

c) The third part includes commands newly proposed to the leader with the intent of being learned at the current ballot. Since these values are being proposed for the first time, they may be appended in any order without restriction.

Fast ballots simplify the procedure and allow any proposer to contact all acceptors to extend the current sequence within only two message delays (if there are no conflicts of parallel proposals). The procedure works as follows:

1. **Proposer to acceptors.** Fast ballot is initiated by the leader, allowing proposers to send proposals directly to the acceptors. Acceptors then receive either a single command or a sequence to append to the history.

2. **Acceptors to acceptors.** Each acceptor appends the received proposals to other accepted proposals in the current ballot. Then it broadcasts the result, which contains a signed tuple of the current ballot and the sequence being voted for. Requiring $N-f$ votes for a sequence of commands ensures that out of two sequences in which the non-commutative commands have different order, only one sequence will receive enough votes. This is true even if all f Byzantine acceptors vote for both sequences, since the remaining $2f+1$ correct acceptors will vote for one sequence.

3. **Acceptors to learners.** Acceptors send to learners Phase $2b$ messages, which contain the current ballot number, the command sequence and the $N-f$ proofs gathered in the verification round. To guarantee that the leader will know what values have been proven and learned, learners must collect $N-f$ Phase $2b$ messages (a guarantee that at least one correct acceptor will relay the last verified sequence to the leader).

4. **Arbitrating an order after a conflict.** In case of proposed non-commutative commands, acceptors may have sequences in different orders that will not be equivalent and will not be learned. Then, the leader runs a classic ballot and collects these unlearned sequences in phase $1b$. The leader will then arbitrate a single serialization of all previously proposed commands and send it to the acceptors. This ensures that the learners will learn the commands in a unique order, thereby preserving consistency.

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;

Figure 3. Pseudocode for acceptor a during the view-change phase of the Byzantine Generalized Paxos algorithm.

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

Upon *receive*$(P1A, ballot, view_l)$ from leader l **do**
  if $view_l == view$ and $bal_a < ballot$ **then**
    SEND $(P1B, ballot, bal_a, proven, val_a, proofs[bal_a])$ to leader;
    $bal_a = ballot$;

Upon *receive*$(FAST, ballot, view_l)$ from leader **do**
  if $view_l == view$ **then**
    $fast\_bal[ballot] = true$;

Upon *receive*$(VERIFY, view_i, ballot_i, val_i, proof)$ from acceptor i **do**
  if $proof_{pub_i} == \langle ballot_i, val_i \rangle$ and $view == view_i$ **then**
    $proofs[ballot_i][val_i][i] = proof$;
    if $(\#(proofs[ballot_i][val_i]) \geq N - f$ or $(\#(proofs[ballot_i][val_i]) > f$ and isUniversallyCommutative$(val_i)))$ and $proofs[ballot_i][val_i][a] \neq \perp$ **then**
      $proven = val_i$;
      SEND $(P2B, ballot_i, val_i, proofs[ballot_i][value_i])$ to learners;

Upon *receive*$(P2A\_CLASSIC, ballot, view, value)$ from leader **do**
  if $view_l == view$ **then**
    PHASE_2B_CLASSIC$(ballot, value)$;

Upon *receive*$(P2A\_FAST, value)$ from proposer **do**
  PHASE_2B_FAST$(value)$;

function PHASE_2B_CLASSIC$(ballot, value)$
  $univ\_commut =$ isUniversallyCommutative(val_a);
  if $ballot \geq bal_a$ and $!fast\_bal[bal_a]$ and $(univ\_commut$ or $proven == \perp$ or $proven ==$ SUBSEQUENCE$(value, 0, \#(proven))$ **then**
    $bal_a = ballot$;
    if $univ\_commut$ **then**
      SEND $(P2B, bal_a, value)$ to learners;
    else
      $val_a = value$;
      $proof = \langle ballot, val_a \rangle_{priv_a}$;
      $proofs[ballot][val_a][a] = proof$;
      SEND $(VERIFY, view, ballot, val_a, proof)$ to acceptors;

function PHASE_2B_FAST$(ballot, value)$
  if $ballot == bal_a$ and $fast\_bal[bal_a]$ **then**
    if isUniversallyCommutative $(value)$ **then**
      SEND $(P2B, bal_a, value)$ to learners;
    else
      $val_a = val_a \bullet value$;
      $proof = \langle ballot, val_a \rangle_{priv_a}$;
      $proofs[ballot][val_a][a] = proof$;
      SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors;

Figure 4. Pseudocode for acceptor a during the agreement phase of the Byzantine Generalized Paxos algorithm.

```
Local variables: learned = ⊥, messages = ⊥

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 == ⟨ballot, value⟩ then
7:             valid_proofs+ = 1;
8:     if valid_proofs ≥ N − f then
9:         messages[ballot][value][a] = proofs;
10:        if #(messages[ballot][value]) ≥ 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 • 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 • c;
23:     return old_seq;
```

Figure 5. Pseudocode for learner ℓ in the Byzantine Generalized Paxos algorithm

5. BYZANTINE VERTICAL PAXOS

Byzantine Vertical Paxos (or BVP) [5] – is a protocol that focuses on two aspects, elasticity (dynamic reconfiguration) and throughput, and is a Byzantine variant of Vertical Paxos. Vertical Paxos has two modes: a simple steady-state protocol and a Paxos-based reconfiguration protocol. The steady-state is often defined as a primary-backup, Chain Replication (CR) or Two-Phase Commit (2PC), allowing the protocol to be optimized for high throughput, while the simple Paxos is only invoked during reconfiguration. This optimization is possible with the assumption that there is an out-of-band synchronous control channel for reconfiguration purposes. In this case, steady-state mode requires $f + 1$ replicas, reconfiguration mode – $2f + 1$ replicas.

5.1. Wedging a Replicated State-Machine

State-Machine Replication (SMR) involves reaching agreement on a sequence of state-machine commands, where consensus on each command is reached independently. In the steady state, a fixed set of replicas and a fixed algorithm drive decisions.

Reconfiguration alters the steady-state mode when the system initiates changing the entire set of replicas or replacing a leader in a leader-based scheme.

The core mechanism employed for reconfiguration is a wedging scheme. A wedging coordinator collects validation from a wedge – a subset of replicas – and retrieves the latest state the wedge stores, including all stored proposals across sequence positions.

The coordinator then initiates a consensus decision for reconfiguration, which is handled by a separate Byzantine consensus component known as the *reconfiguration engine*. Importantly, the reconfiguration consensus decision itself has two components: *next configuration* and *closing state*. The second component, the closing state, plays a crucial role because reconfiguration may begin while some consensus decisions are still in progress. For example, in a $(2f + 1)$-of-$(3f + 1)$ scheme, a wedging coordinator collects information from $2f + 1$, leaving the remaining f unknown. If the coordinator observes that $f + 1$ (of the $2f + 1$) voted for a certain state-machine command, the only safe option is to include that command in the closing state. It is assumed that it is unknown whether the remaining f replicas have or will ever vote.

After the wedging procedure is complete and a consensus on the reconfiguration 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 has 4 rounds and $2f + 1$ replicas for the steady state, and Zyzzyva uses 3 rounds in the optimistic case and requires $3f + 1$ replicas for a steady state. In both cases, reconfiguration requires $3f + 1$ replicas.

5.3. Synchronous-Reconfiguration Model

BVP provides two options in the synchronous model: a 4-round (4 message delays) solution with $f + 1$ replicas, and a 3-round 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:

Table 1. A standart 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

Reconfiguration is executed if at any stage the delay gets too large and, in this case, the model must be synchronous. Reconfiguration is handled as follows: To prevent Byzantine replicas from truncating the history of validated Primary commands, the wedging coordinator must contact all non-faulty replicas. It is assumed that the response of the non-faulty replica might arrive within a certain period of time, given the delay time. After this time, the coordinator reviews the responses. Then, for each command for which a composite message with $f + 1$ signed echoes has been received, the coordinator adopts it in the closing state.

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

Table 2. A standart 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

5.4. Asynchronous Model with a TPM

In this setting, servers are equipped with a Trusted Platform Module. Formally, it is assumed a weak sequential broadcast WScast ensures that:

1. Messages originating from the same sender are delivered in the same order by all correct processes. This ordering is constructed as a per-sender, similar to FIFO broadcast.
2. If the sender is correct, then all processes will eventually receive all messages it sends.

The model implements SMR in a steady state through a 3-round protocol requiring just $f + 1$ replicas, while maintaining linear message complexity.

Table3. A SMR solution with a 3-round protocol

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

The requirement of only $f +$ replicas and the linearity of communication in terms of the number of replicas makes the TMP approach more efficient. In addition, it is possible to use the “proof of elapsed time” to simplify and increase the efficiency of the leader election part inside the reconfiguration service.

6. KURSAWE‘S OPTIMISTIC BYZANTINE AGREEMENT

Optimistic Byzantine Agreement (or OBA) [3] 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 the 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: the protocol tries to reach the agreement by an optimistic phase, believing that the system works in a friendly environment and tests the results; if any inconsistencies are detected, the system invokes the asynchronous, reliable fallback protocol. Note that it is possible that 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.

6.1. Model and Problem Setting

There are n parties $P_1, \ldots, P_n$ up to f, $f < \frac{n}{3}$ of which may be corrupted by an adversary and might behave arbitrarily maliciously. The model uses “time”, however, it 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 a system it "implements" 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 can change this value by either executing **start timer** or **stop timer** commands, or inspect the value of the timer. The **start timer** command works by having the thread send a unique message to itself. The adversary simulates a timeout by delivering this message. Upon delivery, the timer is considered stopped, and if the thread is blocked waiting on a *timeout*, it is then resumed.

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 $\{0,1\}$. Besides normal messages, it may send a message (*ID, DECIDE, final value*). In the last 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 with ID after a decision for it.
There are 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 cause honest parties to disagree.
2. **Validity.** If all honest parties have been activated with the same initial value on a given ID, then all of those parties must decide on that value.
3. **Termination.** Since it is not possible to use the traditional definition of termination, the model constructs computation time in terms of messages sent, and split termination into two conditions: *deadlock freeness* and *efficiency:*

    a. *Deadlock freeness.* It is computationally infeasible for an adversary to induce a state where, for a given ID, all honest-party messages have been delivered, and every party has been activated, while some honest parties remain undecided.
    b. *Efficiency.* For each ID, the communication complexity is probabilistically uniformly bounded.

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

6.2. 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:

1. **Simple Agreement:** In a simple non-Byzantine agreement protocol, each participant broadcasts its preference and waits to receive preferences from all other parties.
2. **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*.

3. **Decide and Hibernate:** A party that decides 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.
4. **Complain and Recover:** A party that receives either inconsistent or insufficient responses broadcasts a complaint. Upon receiving such a complaint, an honest party initiates the pessimistic phase of the protocol.

```
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
```

Figure 10. Pseudocode of the Optimistic-Byzantine Agreement protocol.

7. RESULTS

The key takeaway is that the numbers of processes, rounds, and messages all depend on each other. The choice, therefore, consists of balancing these and considering additional useful features, such as fast ballots or engineering optimizations. So now, let us give a short summary of each protocol:

1. The BPCon is a simple modification for Paxos to make it byzantine. It has a classic two-phase structure, which only differs from the classic Paxos in two steps (1c and 2av (ex. 2a)).
2. The CLA is a refined BPCon that contains optimizations allowing to reduce the number of messages and simplify communication.
3. The BGP is a model that provides two communication steps. At the cost of more complex implementation it provides lower costs for communication. It has one interesting property – it allows to choose not just one value, but a whole sequence of commands. Also, the protocol maintains two modes of ballots: fast and classic. Classic ballots look like an original Paxos ballots when the leader communicates with other processes in a system. In fast ballots, proposers can send their proposal directly to the acceptors without a leader and, if there is no conflict at this stage, they can commit their result. Note that the leader must construct messages for a proposal within the rules.

4. The BVP is a model that has two modes: a simple steady-state protocol and a reconfiguration protocol. The system requires a reconfiguration for changing replicas or the leader, which uses the wedge scheme mechanism. Note that reconfiguration requires a separate Byzantine consensus engine.
5. The OBA is a model that uses two modes. In the optimistic phase, the system is assumed to not be under attack. Then, the system tries to decide on some value and tests the results. If something goes wrong, the system invokes the pessimistic mode, which can solve the consensus problem in any case. This mode requires slightly more communication than the base protocol and usage of public-key cryptography.

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.

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

8. 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 the use of a two-phase algorithm in conjunction with $3f+1$ processes to be the most optimal scheme that can provide robustness properties. Having more rounds is not inherently bad, but at the very least incurs an additional message delay. Namely, BVP's 3 and 4-round modes have delay time up to twice as long. Also, it requires a Trusted Platform module for the asynchronous model. In contrast, FaB's (and PFaB's) $5f+1$ processes make it harder to assemble a byzquorum.

Another important item pertains to the number of messages needed. By using $1c$ step in BPCon, acceptors do not need to communicate about their pre-choices. But BGP requires that N acceptors send N messages. It also needs additional communication with proposers and learners. It might

be a good property when the system needs to propose a lot of values concurrently (BGP supports voting for a set of commands), but suboptimal in the classic case.

Moreover, BPCon main steps are quite similar to PoS BFT algorithms, such as Tendermint [7], Catchain [8], HotStuff [9], GRANDPA [10], RBFT[11] , etc. The general idea of these algorithms is three successive steps of the system: prevote, precommit and commit. In BPCon, acceptors send confirmations for the value from the leader (prevote). The leader sends acceptors the pre-vote that it 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.

However, we believe that BPCon is more of a temporary solution. Further research should be developed into new consensus approaches that comply with modern security requirements, including BFT, and can achieve effective performance evaluations.

REFERENCES

[1] Miguel Castro and Barbara Liskov. “Practical byzantine fault tolerance and proactive recovery”. In: ACM Trans. Comput. Syst. 20.4 (Nov. 2002), pp. 398–461. issn: 0734-2071. doi:10.1145/571637.571640. url: https://doi.org/10.1145/571637.571640.

[2] Leslie Lamport, (2011) “Byzantizing Paxos by refinement”, Lecture Notes in Computer Science, vol. 6950; LNCS; 2011. DOI: 10.1007/978-3-642-24100-0_22.

[3] Klaus Kursawe, (2002) “Optimistic Byzantine agreement”, in Proceedings of the IEEE International Symposium on Reliable Distributed Systems; 2002. DOI: 10.1109/RELDIS.2002.1180196.

[4] Jean Philippe Martin & Lorenzo Alvisi, (2006) “Fast Byzantine consensus”, IEEE Transactions on Dependable and Secure Computing, vol. 3; 2006. DOI: 10.1109/TDSC.2006.35.

[5] Leslie Lamport, Dahlia Malkhi & Lidong Zhou, (2009) “Vertical Paxos and primary-backup replication”, 2009. DOI: 10.1145/1582716.1582783.

[6] Miguel Pires, Srivatsan Ravi & Rodrigo Rodrigues, (2018) “Generalized Paxos Made Byzantine (and Less Complex)”, Algorithms, vol. 11; no. 9; 2018. DOI: 10.3390/a11090141.

[7] Jae Kwon, (2014) TenderMint: Consensus without Mining.

[8] Nikolai Durov, (2020) Catchain Consensus: An Outline; Feb. 19, 2020.

[9] Maofan Yin et al., (2019) “HotStuff”, Association for Computing Machinery; pp. 347–356; Jul. 2019. DOI: 10.1145/3293611.3331591.

[10] Alistair Steward & Eleftherios Kokoris-Kogia, (2020) GRANDPA: A Byzantine Finality Gadget; Jun. 19, 2020.

[11] Pierre Louis Aublin, Sonia Ben Mokhtar & Vivien Quema, (2013) “RBFT: Redundant Byzantine fault tolerance”, Proceedings – International Conference on Distributed Computing Systems; 2013. DOI: 10.1109/ICDCS.2013.53.

AUTHORS

Illia Melnyk has a Master's degree in Applied Mathematics from the National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", NTUU KPI. His research areas include Protocols of Secure Communication, Number theory and Cryptanalysis . Currently working as a teacher's assistant at NTUU KPI and as a researcher at Distributed Lab.

Oleksandr Kurbatov is a PhD Candidate in Karazin Kharkiv National University, Ukraine. His research focuses on Public-Key Infrastructure, Blockchain Technologies, and Anonymous Decentralized Voting Systems. He has presented work at the 2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT), CMiGIN 2019, and the 2023 IEEE 1st Ukrainian Distributed Ledger Technology Forum (UADLTF). His publications include a paper in ISCI 2019: Information Security in Critical Infrastructures. Currently working as the Lead Cryptography Researcher at Distributed Lab.

Oleg Fomenko has successfully completed his B.Sc. degree in Computer Science at Taras Shevchenko National University of Kyiv, Ukraine. His research areas include Private Payments, Zero-Knowledge Virtual Machines, Zero-Knowledge Protocols. Currently working as a researcher at Distributed Lab.

Volodymyr Dubinin holds an M. Sc. in Computer Science and is Co-Founder of Distributed Lab. His research interests span Decentralized Systems, Artificial Intelligence, Blockchain Scalability, and Cryptographic Protocol Design.

Yaroslav Panasenko is the Chief Technology Officer at Distributed Lab. He holds a B.Sc. in Software Engineering from Kharkiv Polytechnic Institute. His research interests span decentralized anonymous banking systems, blockchain infrastructure, biometric systems and AI safety and security.