1. Introduction

This document provides a simplified proof of the consensus algorithm SABR-Paxos.[1] SABR-Paxos provides rapid and scalable consensus in a decentralized (i.e., leaderless) and asynchronous environment, assuming the presence of a persistent advanced threat.

As the number of processes running a proposal in SABR-Paxos is potentially infinite, the mathematical proof of SABR-Paxos relies on set theory. Consequently, the authors presume that the reader is familiar with set theory.

2. Preliminaries

2.1. SABR-Paxos parameters

An application using SABR-Paxos must decide on several parameters:

Name	**Definition**
Proposal	A value (i, j) that is nominated as true
Process	A run N of the SABR-Paxos protocol

[1] The proofs in this paper are written in pseudocode and monospaced unicode, instead of TLA+, to make the paper as widely accessible as possible. However, any reader familiar with TLA+ could easily convert the proofs in this document to TLA+.

Consenus	Agreement on the proposal
Node	An instance connected to SABR-Paxos
Timeout $\triangle$	the upper bound on the communication delay between any two correct machines

3. Definitions

The Paxos Protocol (classic Paxos) relies on two definitions:

- **DEF1**: Proposal v is accepted iff it is accepted by at least one N.
- **DEF2:** Proposal v is chosen iff it is accepted by a strict majority of N.

In Section 4, we prove a modified version of the first of these definitions as DEF3A. The authors acknowledge that the proof of Paxos has been achieved many times before, but the proof of DEF1 is important for understanding SABR-Paxos.

DEF1 and DEF2 imply an inherent problem with Paxos that SABR-Paxos solves. The above definitions do not apply a solution if a strict majority of processors cannot agree on a proposal (i.e., dueling leaders); essentially, the FLP impossibility. Multi-Paxos solves this problem by assigning a stable leader/distinguished proposer, but this introduces centralization and places a large (theoretically infinite) demand for process instances on the leader.

Egalitarian-Paxos (EPaxos) removed the requirement for a stable leader by attaching ordering constraints to an instance. However, EPaxos 1) required a consistent message delivery order for conflicting messages and 2) failed to track the last proposal. EPaxos is also susceptible to a Finney Attack as the initial proposer is the leader.

SABR-Paxos removes the constraint of a consistent message delivery order by introducing tolerance of Byzantine faults in combination with network asynchrony, as long as a majority of replicas are correct and communicate synchronously. SABR-Paxos assigns a random acceptor as the leader for each N, but does not require that the leader is consistent beyond each N (see Section 6.2).

Consequently, SABR-Paxos achieves decentralized (i.e., leaderless) consensus as long as no more than $\frac{n-1}{2}$ nodes are hostile,[2] producing the following definitions:

- **DEF3A:** Proposal v with timestamp T from N_1 is accepted iff it does not contradict at least one other proposal (m, u) with timestamp T.

$\wedge$

- **DEF3B:** Proposal v with timestamp T from N_1 is chosen iff it is accepted by a strict majority of N in operation during timeout $\triangle$.

Similar to EPaxos, SABR-Paxos has a fast path for consensus when there is no competing proposal, although SABR-Paxos adds a last proposal identification derived from timestamp T to

[2] The bound of $\frac{n-1}{2}$ is the general consensus limit; no algorithm can mathematically produce consensus outside of this bound.

ensure security. However, SABR-Paxos also assumes a persistent advanced threat to participants, deviating from Egalitarian Paxos in the slow path where it is assumed that an adversary is controlling $\frac{n-1}{2}$ nodes *and* the message delivery schedule across the entire network.

4. The SABR-Paxos Proof

4.1 Proof for DEF3A:

For any X arbitrary set of value x

$\wedge$ *Any finite set of acceptors N:*

Next $\equiv \text{set} = \{\} \wedge \exists\ \vee \in \text{arbitary\_set}: \text{set}' = \{\ \vee\ \}$

Theorems:

$\forall i: |\text{set\_i}| \leq 1$

$\wedge\ \forall \text{i, j}: i \leq j \wedge (\text{set\_i} \neq V \Rightarrow \text{set\_i} = \text{set\_j})$

The first theorem states that the set of i is less than or equal to 1. This allows empty sets, but precludes a set greater than 1, meaning that the desired set may contain, at most, one value. Establishing this parameter allows us to perform calculations using the second theorem.

For the second theorem, if i and j are far from each other in state space, then we immediately apply *NEXT*. if i1 = j1 - 1, *NEXT* does not hold for i, j, concluding the proof.

For the next theorems, SABR-Paxos must achieve consensus in a byzantine-fault environment:

4.2 Proof for DEF3B:

For any X arbitrary set of value x

∧ *Any finite set of acceptors N*

∧ *Last proposal P*

∧ *Message m*

∧ *Acceptor a*

∧ *Set msg of all m sent until timeout* △*:*

```
Round1(ballot) ≡
    m(message, ballot, T)
    ∧ max_ballot' = max_ballot
    ∧ P' = P
    ∧ T ≮ timeNow
    ∧ T ≤ △
Round2a(ballot) ≡
    ∃ m ∈ msg:
        Round1 ∧ max_ballot(Round1) < Round1(ballot)
        ∧ max_ballot' = ⋋ a1 ∈ acceptors: if a = a1
                                then m(ballot) - 1
                                else max_ballot (Round1)
        ∧ P' = P
        ∧ m2(message, ballot, T, P)
```

```
Round2b(ballot b, vote) $\equiv$

    $\exists$ m $\in$ msg:

        Round2a $\wedge$ m(ballot).= b

        $\wedge$ $\exists$ quorum $\in$ all_ quorums:

        let

        quorum_msg $\equiv$ { m $\in$ msg: m2 $\wedge$ m(ballot).= b $\wedge$ m(acc) $\in$ quorum

        quorum_votes $\equiv$ { m $\in$ quorum_msgs: m(vote) $\neq$ null }

        in

        $\forall$ a $\in$ quorum: $\exists$ m $\in$ quorum_msgs: m(acc) = a

        $\wedge$ (

           quorum_votes = {}

           $\vee$ $\exists$ m $\in$ quorum_votes: m(vote) = v

           $\wedge$ $\forall$ m(Round1) $\in$ quorum_votes: m(Round1(vote))  < =  m(vote)

           $\wedge$ m2(ballot, v)

        )

        $\wedge$ max_ballot' = max_ballot

        $\wedge$ P' = P

        $\wedge$ T $\nless$ timeNow

        $\wedge$ T $\leq$ $\vartriangle$

Round2c(ballot) $\equiv$

    $\exists$ m $\in$ msg:

        Round1 $\wedge$ max_ballot(Round1) < Round1(ballot)
```

```
∧ max_ballot’ = ⋋ a1 ∈ acceptors: if a = a1 then m(ballot) else max_ballot(a1)
        ∧ P’ = ⋋ a1 ∈ acceptors: if a = a1
                        then {m(ballot), m(v)}
                        else P(a1)
        ∧ m3(ballot, v, a, T)
```

Round1 represents an opening proposal, including a ballot number, timestamp, and last proposal identifier (which was missing in EPaxos).

Round2a represents a random acceptor receiving Round1. If the acceptor can vote in the ballot (i.e., it has not already sent its last proposal for the ballot), it replies with P. If the acceptor has already moved beyond the ballot round, Round1 is ignored.

In Round2a, the first node receiving the opening proposal is selected to be the leader. If a proposal fails to achieve consensus, the leader selection for the round is irrelevant; wlog, the proof of leader selection validity is contained in DEF3B.

Round2b ensures that 2a is not already started to prevent the same process for different values. This round requires that the ballot leader (the proposer of Round1) be stable throughout the N. All Round 2b messages are collected from the quorum of nodes. Quorum members increase max_ballot before sending m2, preventing voting in earlier ballots. The value of v is safe if the whole quorum has not voted yet or if it is the latest vote of a quorum member. As long as v is safe, message 2 is broadcast.

Round 2c looks similar to Round 2a, with the important distinction that Round 2c is the acceptance round. The authors highlighted the difference between Round 2a and Round 2c with red text to clarify the difference. Round 2c confirms that P is valid and, if so, accepts the votes for v.

5. Limitions

SABR-Paxos cannot reach valid consensus if more than $\frac{n-1}{2}$ are hostile during timeout $\triangle$.

6. Attacks and Mitigations

The classic problem with decentralized confirmation that Satoshi Nakomoto solved with bitcoin is the double-spend problem. For example, while the classic example is that Bob buys something in transaction Y with funds A and then transfers funds A into a separate account controlled by Bob before Y is confirmed, Y will not be confirmed and Bob has effectively bought something for free. While often considered in terms of currency, double-spends are a problem for any record that attempts to determine truth. For example, a person could have multiple identities, or multiple encryption pairs, held on a ledger. In centralized ledgers (like SQL servers) the double spend problem is prevented by a controlling authority determining what is valid or invalid. In decentralized ledgers, the majority of participants decide.

SABR-Paxos reaches consensus quickly, resolving some double-spend concerns, but their are two attacks possible on any decentralized ledger that are also applicable to SABR-Paxos: Race Attacks and Finney Attacks.

6.1 Race Attack

A race attack occurs when two communications on a ledger, one valid and the other invalid, are sent at almost the same time, with the possibility that an invalid communication may race to confirmation before a valid confirmation. From the proofs above, each confirmation is ordered by timestamp and confirmed asynchronously, negating the possibility of a Race Attack ***as long as a user, as a business rule, does not accept an unconfirmed communication***.

6.2 Finney Attack

A Finney Attack occurs when the proposer of a communication controls the consensus approver. EPaxos is uniquely vulnerable to this attack as the proposer is also the leader. However, SABR-PAXOS mitigates this attack by assigning the leader to a random acceptor. As long as the pool of participants is sufficiently large to prevent control of a proposer and acceptor by the same entity, a Finney Attack is not possible.

7. IPR

8. Acknowledgements

Thanks to Leslie Lamport for inventing the Paxos protocol, upon whose work much of the SABR-Paxos protocol is based. Thanks to Iulian Moraru, David G. Andersen, and Michael Kaminsky for inventing Egalitarian Paxos. Thanks to Pierre Sutra for demonstrating the flaw with Egalitarian Paxos. Thanks to Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quema, Marko Vukolic for inventing XPaxos, an XFT SMR protocol. Thanks to Pasindu Tennage, Cristina Basescu, Eleftherios Kokoris Kogias, Ewa Syta, Philipp Jovanovic, and Bryan Ford for inventing Baxos and demonstrating the utility of a Random Exponential Backoff to distributed consensus.