Quorum Tree Abstractions of Consensus Protocols

Berk Cirisci[1][0000−0003−4261−090X], Constantin Enea[2][0000−0003−2727−8865], and Suha Orhun Mutluergil[3][0000−0002−0734−7969]

[1] IRIF, Université Paris Cité
cirisci@irif.fr
[2] LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris
cenea@lix.polytechnique.fr
[3] Sabanci University
suha.mutluergil@sabanciuniv.edu

Abstract. Distributed algorithms solving agreement problems like consensus or state machine replication are essential components of modern fault-tolerant distributed services. They are also notoriously hard to understand and reason about. Their complexity stems from the different assumptions on the environment they operate with, i.e., process or network link failures, Byzantine failures etc. In this paper, we propose a novel abstract representation of the dynamics of such protocols which focuses on quorums of responses (votes) to a request (proposal) that form during a run of the protocol. We show that focusing on such quorums, a run of a protocol can be viewed as working over a tree structure where different branches represent different possible outcomes of the protocol, the goal being to stabilize on the choice of a fixed branch. This abstraction resembles the description of recent protocols used in Blockchain infrastructures, e.g., the protocol supporting Bitcoin or Hotstuff. We show that this abstraction supports reasoning about the safety of various algorithms, e.g., Paxos, PBFT, Raft, and HotStuff, in a uniform way. In general, it provides a novel induction based argument for proving that such protocols are safe.

1 Introduction

Consensus or state-machine replication protocols are essential ingredients for maintaining strong consistency in modern fault-tolerant distributed systems. Such protocols must execute in the presence of concurrent and asynchronous message exchanges as well as benign (message loss, process crash) or Byzantine failures (message corruption). Developing practical implementations or reasoning about their correctness is notoriously difficult. Standard examples include the classic Paxos [21] or PBFT [5] protocols, or the more recent HotStuff [37] protocol used in Blockchain infrastructures.

In this paper, we propose a new abstraction for representing the executions of such protocols that can be used in particular, to reason about their safety, i.e., ensuring *Agreement* (e.g., all correct processes decide on a single value) and *Validity* (e.g., the decided value has been proposed by some node participating in the protocol). Usually, protocol executions are composed of a number of communication-closed rounds [11], and each round consists of several phases in which a process broadcasts a request and expects to collect responses from a quorum of processes before advancing to the next phase. The abstraction is defined as a *sequential* object called *Quorum Tree (QTree)* which maintains a tree structure where each node corresponds to a different round in an execution. The operations of QTree, to add or change the status of a node, model quorums of responses that have been received in certain phases of a round.

For instance, a round in single-decree Paxos consists of two phases: a *prepare* phase where a pre-determined leader broadcasts a request for joining that round and expects a quorum of responses from the other processes before advancing to a *vote* phase where it broadcasts a value to agree upon and expects a quorum of responses (votes) in order to declare that value as decided in that round. Rounds are initiated by their respective leaders and can run concurrently. The idea behind QTree is to represent a Paxos execution using a rooted tree where each node different from the root corresponds to a round where the leader has received *a quorum of responses in the prepare phase*. The parent-child relation models the data flow from one round to a later round: responses to join requests contain values voted for in previous rounds (if any) and one of them will be included by the leader in the vote phase request. The round in which that value was voted defines the parent. Then, each node has one out of three possible statuses: `ADDED` if the vote phase can still be successful (the leader can collect a quorum of votes) but this did not happen yet, `GHOST` if the vote phase can not be successful (e.g., a majority of processes advanced to the next round without voting), and `COMMITTED` if the leader has received a quorum of responses in the vote phase. This is a tree structure because before reaching a quorum in the vote phase of a round, other rounds can start and their respective leaders can send other vote requests (with possibly different values). The specific construction of requests and responses in Paxos ensures that all the `COMMITTED` nodes in this tree belong to a *single* branch, which entails the agreement property (this will become clearer when presenting the precise definition of QTree in Section 2).

The QTree abstraction is applicable to a wide range of protocols beyond the single-decree Paxos sketched above. It applies to state-machine replication protocols like Raft [36] and HotStuff [37] where the tree structure represents logs of commands (inputted by clients) stored at different processes and organized according to common prefixes (each node corresponds to a single command) and multi-decree consensus protocols like multi-Paxos [21] and its variants [16, 26, 23, 18], or PBFT [5] where different consensus instances (for different indices in a sequence of commands) are modeled using different QTree instances.

We show that all these protocols are *refinements* of QTree in the sense that their executions can be mapped to sequences of operations on a QTree state, which are about agreeing on a branch of the tree called the *trunk*. These operations are defined as invocations of two methods *add* and *commit* for adding a new leaf to the tree (during which some other nodes may turn to GHOST) and changing the status of a node from ADDED to COMMITTED, respectively. Any sequence of invocations to these methods ensures that all the COMMITTED nodes lie on the same branch of the tree (the trunk). In relation to protocol executions, *add* and *commit* invocations that concern the same node correspond to receiving a quorum of responses in two specific phases of a round, which vary from one protocol to another.

The mapping between protocol executions and QTree executions is defined as in proofs of linearizability for concurrent objects with *fixed* linearization points. Analogous to linearizability, where the goal is to show that an object method takes effect instantaneously at a point in time called linearization point, we show that it is possible to mark certain steps of a given protocol as linearization points of *add* or *commit* operations[4], such that the sequence of *add* and *commit* invocations defined by the order between linearization points along a protocol execution is a correct QTree execution. We introduce a declarative characterization of correct QTree executions that simplifies the proof of the latter (see Section 3).

The QTree abstraction offers a novel view on the dynamics of classic consensus or state-machine replication protocols like Paxos, Raft, and PBFT, which relates to the description of recent Blockchain protocols like HotStuff and Bitcoin [27], i.e., agreeing on a branch in a tree. It provides a formal framework to reason *uniformly* about single-decree consensus protocols and state-machine replication protocols like Raft and HotStuff. For single-decree protocols (or compositions thereof), the parent-child relation between QTree nodes corresponds to the data-flow between a quorum of responses to a leader and the request he sends in the next phase while for Raft and HotStuff, it corresponds to an order set by a leader between different commands.

Our work relies on a hypothesis that correctness proofs based on establishing a refinement towards an *operational* specification such as QTree, which can be understood as a sequence of steps, are much more intuitive and "explainable" compared to classic proofs based on inductive invariants. An inductive invariant has to describe all intermediate states produced by all possible orders of receiving messages and a precise formalization is quite complex. As an indication, the Paxos invariant used in recent work [29] (see formulas (4) to (12) in Section 5.2) is a conjunction of eight quantified first-order formulas which are hard to reason about and not re-usable in the context of a different protocol.

We believe that operational specifications are also helpful in taming complexity while designing new protocols or implementations theoreof, or in gaining confidence about their correctness without going through ad-hoc and brittle proof arguments. For instance, our proofs are very clear about the phases of a round in which quorums need to intersect, which provides flexibility and opti-

[4] These linearization points are *fixed* in the sense that they correspond to specific instructions in the code of the protocol, and they do not depend on the future of an execution. For an expert reader, this actually corresponds to a proof of strong linearizability [15].

mization opportunities for deciding on quorum sizes in each phase. Depending on environment assumptions, quorum sizes can be optimized while preserving correctness. Compared to previous operational specifications for reasoning about consensus protocols, e.g., [3, 12], QTree is designed to be less abstract so that the refinement proof, establishing the relationship between a given protocol and QTree, is less complex (see Section 8 for details).

2 Quorum Tree

We describe the QTree sequential object which operates on a tree and has two methods *add* and *commit* for adding a new node and modifying an attribute of a node (committing a node), respectively. When used as an abstraction of consensus protocols, invocations of these two methods correspond to certain quorums that are reached during a round of the protocol.

2.1 Overview

QTree is a sequential rooted-tree, a possible state being depicted in Figure 1. The nodes with black dashed margins are not members of the tree and they are discussed later. Each node in the tree contains a round number, a value, and a status field set to `ADDED`, `GHOST`, or `COMMITTED`. The round number acts as an identifier of a node since there can not exist two nodes with the same round number. The *Root* node is part of the initial state and its status is `COMMITTED`. A QTree state consists of a trunk, alive branches, and dead branches; a branch is a chain of nodes connected by the parent relation. Alive branches are extensible with new `ADDED` nodes but dead branches are not. The trunk is a particular branch of the tree that starts from the root. It contains all the `COMMITTED` nodes and it ends with a `COMMITTED` node. It may also contain `ADDED` or `GHOST` nodes. For example, in Figure 1, the trunk consists of *Root* and n_3. All alive branches are connected to the last `COMMITTED` node of the trunk (alive branches can include `ADDED` or `GHOST` nodes). For instance, in Figure 1, the subtree rooted at n_3 contains a single alive branch whose leaf node is n_5. Dead branches can contain only `GHOST` nodes. In Figure 1, the tree contains a single dead branch containing the node n_1.

Nodes can be added to the tree as leaves. The status of a newly added node is either `ADDED` or `GHOST`. The status `ADDED` may turn to `GHOST` or `COMMITTED`. The `GHOST` status is "final" meaning that it can never turn into `COMMITTED` afterwards. However, `GHOST` nodes can be part of alive branches, and they can help in growing the tree.

QTree has two methods *add* and *commit*:

– *add* generates a new leaf with a round number r value v and parent p identified by the round number r_p given as an input. Its status is set to `ADDED` or `GHOST` provided that some conditions hold. If the status of the new node is set as `ADDED`, then it either extends (has a path to the end of) an existing alive branch or creates a new alive branch from the trunk. The new node

may also "invalidate" some other nodes by changing their status from `ADDED` to `GHOST`.
- *commit* extends the trunk by turning the status of a node from `ADDED` to `COMMITTED`. This extension of the trunk may prevent some branches to be extended in the future (some alive branches may become dead), i.e., future invocations of *add* that extend those branches will add only `GHOST` nodes.

Each node models the evolution of a round in a consensus protocol and the value attribute represents the value proposed by the leader of that round. The round and value attributes of a node are immutable and cannot be changed later. We assume that round numbers are strictly positive except for *Root* whose round number is 0.

QTree applies uniformly to a range of consensus or state-machine replication protocols. We start by describing a variation that applies to single-decree consensus protocols, where a number of processes aim to agree on a single value. Multi-decree consensus protocols that are used to solve state-machine replication can be simulated using a number of instances of QTree, one for each decree (the instances are independent one from another). Then, state-machine replication protocols like HotStuff that rely directly on a tree structure to order commands can be simulated by the QTree for single-decree consensus modulo a small change that we discuss later.

2.2 Definition of the Single-Decree Version

Algorithm 1 lists a description of QTree in pseudo-code. The following set of predicates are used as conditions inside methods:

1. $\mathit{link}(n) \equiv n.\text{parent} \in \text{Nodes} \land n.\text{parent.round} < n.\text{round}$
2. $\mathit{newRound}(n) \equiv \forall n' \in \text{Nodes}.\ n'.\text{round} \neq n.\text{round}$
3. $\mathit{maxCommitted}(n) \equiv n.\text{status} = \texttt{COMMITTED} \land$
   $(\forall n' \in \text{Nodes}.\ n'.\text{status} = \texttt{COMMITTED} \implies n'.\text{round} < n.\text{round})$
4. $\mathit{extendsTrunk}(n) \equiv \exists n' \in \text{Nodes}.\ \mathit{maxCommitted}(n') \land$
   $(n \text{ extends } n' \lor n.\text{round} < n'.\text{round})$
5. $\mathit{valid}(n) \equiv \mathit{link}(n) \land \mathit{newRound}(n) \land \mathit{extendsTrunk}(n)$
6. $\mathit{valueConstraint}(n) \equiv n.\text{parent} \neq \mathit{Root} \implies n.\text{value} = n.\text{parent.value}$

The *add* method (lines 5-17) generates a new node n with round, value, and parent set according to the method's inputs. Then, it adds n to the tree by linking it to the selected parent if n satisfies the following *validity* conditions:

- n's parent belongs to the tree and its round number is smaller than r (predicate *link* at (1)),
- the tree does not contain a node with round number r (predicate *newRound* at (2)),
- if r is bigger than the round number of the last node of the trunk, then n must extend the trunk (predicate *extendsTrunk* at (4)),
- n's value must be the same as its parent's value unless the parent is the *Root* (predicate *valueConstraint* at (6)).

Algorithm 1: THE QTREE OBJECT

```
1  Initialize:
       /* ⊥ denotes non-initialized values */
2      $Root$.round = 0; $Root$.status = COMMITTED;
3      $Root$.value = ⊥; $Root$.parent = $Root$;
4      Nodes = {$Root$};
5  Method add ($r$, $v$, $r_p$)
6      Pre: $r > 0$
7      $n$ = new Node(round = $r$, status = ⊥,
         value = $v$, parent = $p : p.round = r_p$);
8      if $valid(n) \wedge valueConstraint(n)$
9          Nodes = Nodes ∪ {$n$};
10         $n$.status = ADDED;
11         if $\exists n' \in Nodes.\ n'.round > n.round$
12             $n$.status = GHOST;
13         forall $n' \in Nodes.\ n'.round < n.round$
14             if $n$ is conflicting with $n'$
15                 $n'$.status ← GHOST;
16         return OK
17     return FAIL
18 Method commit ($r$)
19     if $\exists\ n \in Nodes.\ n.round = r \wedge$
         $n.status$ = ADDED
20         $n$.status ← COMMITTED;
21         return OK
22     return FAIL
```

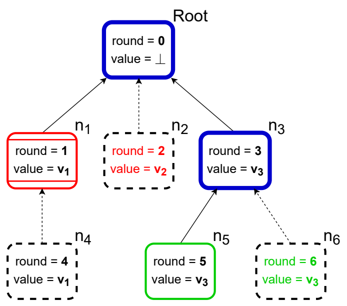

Fig. 1: A state of QTree. We represent ADDED nodes with green solid margins, GHOST nodes with red double-line margins, and COMMITTED nodes with blue thick margins. The nodes with black dashed margins are not part of the state, they are fictitious nodes used to explain the method for adding new nodes.

The *valid* predicate at (5) is the conjunction of the first three constraints.

For example, let us consider an invocation of *add* in a state of QTree that contains the non-dashed nodes in Figure 1. If the invocation generates n_2, n_4, or n_6 (receiving as input the corresponding attributes), then n_2 and n_6 do satisfy all these constraints and can be added to the tree. The node n_4 fails the *extendsTrunk* predicate because it is not extending the last node of the trunk (n_3) and its round number is higher.

If a node n satisfies the conditions above, the *add* method turns its status to either ADDED or GHOST. If there is another node in the tree with a higher round number, n's status becomes GHOST. Otherwise, it becomes ADDED. As a continuation of the example above, the status of n_2 is set to GHOST because the tree contains node n_3 with a higher round number and the status of n_6 is set to ADDED.

Moreover, the addition of n can "invalidate" some other nodes, turn their status to GHOST. This is based on a notion of *conflicting* nodes. We say that two nodes are conflicting if they are on different branches, i.e., there is no path from one node to the other. An *add* invocation that adds a node n changes the

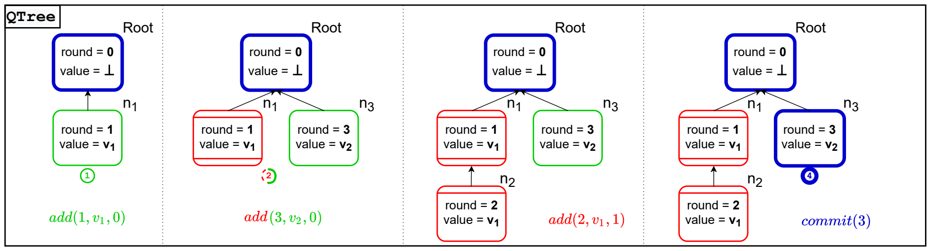

Fig. 2: Explaining the behavior of *add* and *commit* methods. Colors are interpreted as in Fig 1.

status of all the nodes n' in the tree that conflict with n and have a lower round number than n, to GHOST. For example, Figure 2 pictures a sequence of QTree states in an execution, to be read from left to right. The first state represents the result of executing $add(1, v_1, 0)$ on the initial state of QTree, adding node n_1. Executing $add(3, v_2, 0)$ on this first state creates another node n_3 and sets its status to ADDED. This invocation will also turn the status of n_1 to GHOST since its round number is less than the round number of n_3 and they are on different branches. Afterwards, by executing $add(2, v_1, 1)$, a node n_2 is added to the tree with status GHOST since there is a node n_3 on a different branch which has a higher round number.

The method *add* returns OK when the created node is effectively added to the tree (it satisfies the conditions described above) and $FAIL$, otherwise.

Lastly, the *commit* method takes a round number r as input and turns the status of the node containing r to COMMITTED if it was ADDED. If successful, it returns OK and $FAIL$, otherwise. As a continuation of the example above, the right part of Figure 2 pictures a state obtained by executing $commit(3)$ on the state to the left. This sets the status of n_3 to COMMITTED as n_3 was previously ADDED. Note that the conditions in *add* ensure that the tree can not contain two nodes with the same round number.

Safety Properties. We show that the QTree object in Algorithm 1 can be used to reason about the safety of single-decree consensus protocols, in the sense that it satisfies a notion of *Validity* (processes agree on one of the proposed values) and *Agreement* (processes decide on a single value). More precisely, we show that every state that is reachable by executing a sequence of invocations of *add* and *commit* (in Algorithm 1), called simply *reachable state*, satisfies the following:

– *Validity*: every node different from *Root* contains the same value as a child of *Root*, and
– *Agreement*: every two COMMITTED nodes different from *Root* contain the same value.

Proposition 1 (Validity). *Every node in a reachable state that is different from Root contains the same value as a child of Root.*

Proof. A node n is added to the tree only if the predicate *valueConstraint* holds, which implies that it is either a child of *Root* or it has the same value as its parent which is a descendant of *Root*. Also, since the value attribute of a node is immutable, any `COMMITTED` node contains the same value that it had when it was created by an *add* invocation.

Therefore, the fact that a consensus protocol refining QTree satisfies validity, i.e., processes decide on a value proposed by a client of the protocol, reduces to proving that the phases of a round simulated by *add* invocations that add children of *Root* use values proposed by a client. This is ensured using additional mechanisms, i.e., a client broadcasts its value to all participants in the protocol, so that each participant can check the validity of a value proposed by a leader.

Next, we focus on *Agreement*, and show that `COMMITTED` nodes belong to a single branch of the tree.

Proposition 2. *Let n_1 and n_2 be two `COMMITTED` nodes in a reachable state. Then, n_1 and n_2 are not conflicting.*

Proof. Assume towards contradiction that QTree reaches a state where two `COMMITTED` nodes n_1 and n_2 are conflicting. Let $r_1 = n_1$.round and $r_2 = n_2$.round. Without loss of generality, we assume that $r_1 < r_2$. Such a state is reachable if $add(r_1, \_, \_)$ and $add(r_2, \_, \_)$ resulted in adding the nodes n_1 and n_2 and set their status to `ADDED` (we use _ to denote arbitrary values), and subsequently, $commit(r_1)$ and $commit(r_2)$ switched the status of both n_1 and n_2 to `COMMITTED`. If $add(r_1, \_, \_)$ were to execute before $add(r_2, \_, \_)$, then $add(r_2, \_, \_)$ would have changed the status of n_1 to `GHOST` because it is conflicting with n_2. Otherwise, if $add(r_2, \_, \_)$ were to execute before $add(r_1, \_, \_)$, then the latter would have set the status of n_1 to `GHOST` since the tree contains n_2 that has a higher round number. In both cases, executing $commit(r_1)$ can never turn the status of n_1 to `COMMITTED`.

Proposition 2 allows to conclude that any two `COMMITTED` nodes (different from *Root*) contain the same value. Indeed, a node can become `COMMITTED` only if it was `ADDED`, which implies that is has the same value as its parent (the predicate *valueConstraint* holds), and by transitivity, as any of its ancestors, except for *Root*.

Proposition 3 (Agreement). *Let n_1 and n_2 be two `COMMITTED` nodes in a reachable state, which are different from Root. Then, $n_1.value = n_2.value$.*

2.3 State Machine Replication Versions

The single-decree version described above can be extended easily to a *multi-decree* context. As multi-decree consensus protocols, used in state machine replication, can be seen as a composition of multiple instances of single-decree consensus protocols, a multi-decree version of QTree is obtained by composing multiple instances of the single-decree version. Each of these instances manipulates a tree as described above without interference from other instances. The validity and agreement properties above apply separately to each instance.

The single-decree version can also be extended for state machine replication protocols like HotStuff and Raft where the commands (values) are a-priori structured as a tree, i.e., each command given as input is associated to a predetermined parent in this tree. Then, the goal of such a protocol is to agree on a sequence in which to execute these commands, i.e., a branch in this tree. Simply removing the *valueConstraint* condition in the *add* method (underlined in Algorithm 1) enables QTree to simulate such protocols. A node's value need not be the same as its parent's value to be valid for *add*. Proposition 2 that implies the agreement property of such protocols still holds (Proposition 3 does not hold when the *valueConstraint* condition is removed; this property is specific to single-decree consensus). Since the value field remains immutable, the validity property of such protocols reduces to ensuring that the values generated during phases simulated by *add* correspond to commands issued by the client (Proposition 1 is also specific to single-decree consensus and it does not hold). As before, this requires additional mechanisms, i.e., a client broadcasting a command to all the participants in the protocol, whose correctness can be established quite easily.

3 Consensus Protocols Refining QTree

In the following, we show that a number of consensus protocols are refinements of QTree in the sense that their executions can be mimicked with *add* and *commit* invocations. This is similar to a linearizable concurrent object being mimicked with invocations of a sequential specification. The refinement relation allows to conclude that the *Validity* and *Agreement* properties of QTree imply similar properties for any of its refinements.

The definition of the refinement relation relies on a formalization of protocols and QTree as *labeled transition systems*. For a given protocol, a state is a tuple of process local states and a set of messages in transit, and a transition corresponds to an indivisible step of a process (receiving a set of messages, performing a local computation step, or sending a message). For QTree, a state is a tree of nodes as described above and a step corresponds to an invocation to *add* or *commit*. An execution is a sequence of transitions from the initial state.

Refinement corresponds to a mapping between protocol executions and QTree executions. This mapping is defined as in proofs of linearizability for concurrent objects with *fixed linearization points*, where the goal is to show that each concurrent object method appears to take effect instantaneously at a point in time that corresponds to executing a fixed statement in its code. Therefore, certain steps of a given protocol are considered as linearization points of *add* and *commit* QTree invocations (returning OK), and one needs to prove that the sequence of invocations defined by the order of linearization points in a protocol execution is a correct execution of QTree.

Formally, a labeled transition system (LTS) is a tuple $L = (\mathcal{Q}, q_0, \mathcal{T}, \mathcal{A}_L)$ where $\mathcal{Q}$ is a set of states, q_0 is the unique initial state, $\mathcal{A}_L$ is a set of actions (transition labels) and $\mathcal{T}$ is a set of transitions (q, a, q') such that $q, q' \in \mathcal{Q}$ and $a \in \mathcal{A}_L$. An *execution* E from q_0 is a finite sequence of alternating states and actions such that $E = q_0, a_0, q_1, a_1, \ldots, q_n$ with $(q_i, a_i, q_{i+1}) \in \mathcal{T}$ for each

$0 \leq i \leq n-1$. A trace t is the sequence of actions projected from some execution E. $T(L)$ denotes the set of traces of L.

The standard notion of refinement between LTSs states that an LTS L is a refinement of another LTS L' when $T(L) \subseteq T(L')$. In this paper, we consider a slight variation of this definition of refinement that applies to LTSs that do *not* share the same set of actions, representing for instance, some concrete protocol and QTree, respectively. This notion of refinement is parametrized by a mapping Γ between actions of L and L', respectively. We say that L *Γ-refines* L' when $\Gamma(T(L)) \subseteq T(L')$. Here, a mapping $\Gamma : \mathcal{A}_L \rightarrow \mathcal{A}_{L'}$ is extended to sequences and sets of sequences as expected, e.g., $\Gamma(a_1 \dots a_n) = \Gamma(a_1) \dots \Gamma(a_n)$. With this extension, the preservation of safety specifications from an LTS to a refinement of it requires certain constraints on the mapping Γ that will be discussed in Section 4.2.

In the context of proving that a concrete protocol refines QTree, the goal is to define a mapping Γ between actions of the protocol and QTree *add*/*commit* invocations such that Γ applied to protocol executions results in correct QTree executions. In the following, we provide a characterization of correct QTree executions that simplifies such refinement proofs.

3.1 Characterizing QTree Invocation Sequences

An *invocation label* $add(r, v, r_p) \Rightarrow RET$ or $commit(r) \Rightarrow RET$ combines a QTree method name with input values and a return value $RET \in \{OK, FAIL\}$. An invocation label is called *successful* when the return value is OK. A sequence σ of invocation labels is called *correct* when there exist QTree states $q_0, \dots, q_{|\sigma|}$, such that q_0 is the QTree initial state and for each $i \in [1, |\sigma|]$, executing σ_i starting from q_{i-1} leads to q_i.

Theorem 1. *A sequence σ of successful invocation labels is correct if and only if the following hold (we use _ to denote arbitrary values):*

1. *for every r, σ contains at most one invocation label $add(r, \_, \_)$ and at most one invocation label $commit(r)$*
2. *every $commit(r)$ is preceded by an $add(r, \_, \_)$*
3. *if $r_p > 0$, every $add(r, v, r_p)$ is preceded by $add(r_p, v', \_)$ where $0 < r_p < r$ (a) and $v = v'$*
4. *if σ contains $add(r, \_, \_)$ and $add(r', \_, r'')$ with $r'' < r < r'$, then σ does* not *contain $commit(r)$*

Properties 1–3 are straightforward consequences of the *add* and *commit* definitions. Indeed, it is impossible to add two nodes with the same round number r, which implies that there can not be two successful $add(r, \_, \_)$ invocations, the status of a node can be flipped to `COMMITTED` exactly once, which implies that there can not be two successful $commit(r)$ invocations, and a $commit(r)$ is successful only if a node with round number r already exists, hence Property 2 must hold. Moreover, a node's parent defined by the input r_p must already exist in the tree, which implies that Property 3 must also hold. Property 4 is more involved and relies on the fact that a node n with round number r can be `COMMITTED` only if there exist no other conflicting node n' with a bigger round number r' (the parent of n' having a round smaller than r implies that n and n' are conflicting).

Proof. ($\Rightarrow$): Assume that σ is correct. We show that it satisfies the above properties:

– Property 1: The $\mathit{newRound}(n)$ predicate used at line 8 in Algorithm 1 ensures that it is impossible to add two nodes with the same round number r, and therefore σ can not contain two successful $\mathit{add}(r, \_, \_) \Rightarrow \mathit{OK}$ invocations. The conditions at line 19 ensure that $\mathit{commit}(r) \Rightarrow \mathit{OK}$ can flip the status of a node only once, and therefore only one such successful invocation can occur in σ.
– Property 2: The conditions at line 19 in Algorithm 1 imply that the state in which $\mathit{commit}(r) \Rightarrow \mathit{OK}$ is executed contains a node with round number r. This node could have only added by a previous $\mathit{add}(r, \_, \_) \Rightarrow \mathit{OK}$ invocation.
– Property 3: The $\mathit{link}(n)$ predicate used at line 8 in Algorithm 1 ensures that the state in which $\mathit{add}(r, v, r_p) \Rightarrow \mathit{OK}$ is executed contains a node with round number r_p. This node could have only added by a previous $\mathit{add}(r_p, v', \_) \Rightarrow \mathit{OK}$ invocation, for some v'.
  • Property 3a: It is a direct consequence of the $\mathit{valueConstraint}(n)$ predicate used at line 8 in Algorithm 1.
– Property 4: Let n and n' be the nodes of the QTree state q reached after executing σ, which have been added by $\mathit{add}(r, \_, \_) \Rightarrow \mathit{OK}$ and $\mathit{add}(r', \_, r'') \Rightarrow \mathit{OK}$, respectively. We have that n'.round $>$ n.round $>$ n.parent.round. Since the round numbers decrease when going from one node towards *Root* in a reachable QTree state, it must be the case that n and n' are conflicting. By Lemma 1, we get that n.status is `GHOST`. Since the `GHOST` status can not be turned to `COMMITTED` and vice-versa, it follows that σ can not contain $\mathit{commit}(r) \Rightarrow \mathit{OK}$.

($\Leftarrow$): We prove that every sequence σ that satisfies properties 1–4 is correct. We proceed by induction on the size of σ. The base step is trivial. For the induction step, let σ be a sequence of size $k+1$. If σ satisfies properties 1-4, then the prefix σ' containing the first k labels of σ satisfies properties 1-4 as well. By the induction hypothesis, σ' is correct. We show that the last invocation of σ, denoted by σ_{k+1} can be executed in the QTree state $q_{|\sigma'|}$ reached after executing σ'. We start with a lemma stating an inductive invariant for reachable QTree states:

Lemma 1. *For every node n in any state q reached after executing a correct sequence σ of successful invocations, $n.status$ is `COMMITTED` if n is Root or σ contains a $commit(r)$ invocation. Else, $n.status$ is `GHOST` if q contains a node n' with $n'.round > n.round$ and n' is conflicting with n, and it is `ADDED`, otherwise.*

Proof. We proceed by induction on the size of σ. The base step is trivial. For the induction step, let σ be a sequence of size $m+1$. Let q_m be the state reached after executing the prefix of size m of σ, and let σ_{m+1} be the last invocation label of σ. We show that the property holds for any possible σ_{m+1} that takes the QTree state q_m to some other state q_{m+1}:

- $\sigma_{m+1} = add(r, v, r_p) \Rightarrow OK$, for some r, v, r_p: Let n be the new node added by this invocation. The status of n can be `ADDED` or `GHOST`. If q_m contains a node n' with n'.round $> r$ (since round numbers are decreasing going towards the *Root* and n is a new leaf node, any existing node with a higher round number such as n' is also conflicting with n), then the status of n becomes `GHOST` by the predicate at line 11 in Algorithm 1 (otherwise, it remains `ADDED`). This implies that n's status satisfies the statement in the lemma. This invocation may also turn the status of some set of nodes N from `ADDED` to `GHOST` by the statement at line 13 in Algorithm 1. The nodes in N have a lower round number than r and conflicting with n. Therefore, the statement of the lemma is satisfied for the nodes in N.
- $\sigma_{m+1} = commit(r) \Rightarrow OK$, for some r: For $commit(r)$ to be successful the conditions at line 19 in Algorithm 1 must be satisfied. If it is satisfied, only the status of node n is changed from `ADDED` to `COMMITTED`. Note that *Root* exists by definition and its status is `COMMITTED`. Since the statuses of the rest of the nodes stay the same, the statement of the lemma holds. □

There are two cases to consider depending on whether σ_{k+1} is an *add* or *commit* invocation label:

- $add(r, v, r_p)$: This invocation label is successful if and only if the predicates $valid(n)$ and $valueConstraint(n)$ at line 8 in Algorithm 1 are satisfied after generating a new node n with the given inputs in the state $q_{|\sigma'|}$:
  - $newRound(n)$: Due to Property 1, $r \neq n'$.round for any other node $n' \in q_{|\sigma'|}$ and the predicate is satisfied.
  - $link(n)$: To satisfy this predicate, there must exist a node in $q_{|\sigma'|}$ with round r_p where $r_p < r$. By Property 3, if σ contains $add(r, \_, r_p) \Rightarrow OK$ with $r_p \neq 0$, then $add(r_p, \_, \_) \Rightarrow OK$ also exists in σ. Hence, there exists a node p with round r_p in $q_{|\sigma'|}$, and the predicate is satisfied. If $r_p = 0$, then $q_{|\sigma'|}$ contains the *Root* node (with round 0) which ensures that the predicate is satisfied.
  - $extendsTrunk(n)$: This predicate states that n extends the node n' which has the highest round number among the nodes with `COMMITTED` status, if n.round $> n'$.round. Assume by contradiction that this is not the case, i.e., n.round $> n'$.round but n and n' are conflicting. Let n_1 be the lowest common ancestor of n and n' (the first common node on the paths from n and n' to the *Root*). Since the round numbers decrease when going from one node towards *Root*, we have that n_1.round $< n'$.round. If we consider the nodes on the path from n to n_1, since n.round $> n'$.round, there must exist a node n_2 such that n_2.round $> n'$.round but n_2.parent.round $< n'$.round. The node n_2 in $q_{|\sigma'|}$ corresponds to the

invocation label $add(n_2.\text{round}, \_, n_2.\text{parent.round})$ in σ'. Moreover, the `COMMITTED` status of n' implies the existence of $commit(n'.\text{round})$ in σ' as stated in Lemma 1. However, it is impossible that σ' contains both these invocation labels if Property 4 holds.
  - $valueConstraint(n)$: It is implied trivially as Property 3a holds.
- $commit(r)$: It is successful if and only if the conditions at line 19 in Algorithm 1 are satisfied. Then by Property 1 and 2, there exist $add(r, \_, \_)$ in σ' but not $commit(r)$. As $add(r, \_, \_)$ is successful, there already exist a node n in $q_{|\sigma'|}$ where its round is r but its status can be either `ADDED` or `GHOST`. Towards a contradiction, assume that $n.\text{status} = \texttt{GHOST}$ in $q_{|\sigma'|}$. This means that there exists a node n' conflicting with n such that $n'.\text{round} > n.\text{round}$ as stated in Lemma 1. Let n_1 be the least common ancestor of n and n'. Since round numbers are decreasing going towards the $Root$, $n_1.\text{round} < n.\text{round}$. If we consider nodes on the path from n' to n_1, there exists a node n_2 such that $n_2.\text{round} > n.\text{round}$ and $n_2.\text{parent.round} < n.\text{round}$. That's why, there is an invocation label $add(n_2.round, \_, n_2.parent.round)$ in σ'. However, σ cannot contain both of these invocation labels together according to Property 4. □

4 Linearization Points

We describe an instrumentation of consensus protocols with linearization points of successful QTree invocations, and illustrate it using Paxos as a running example. Section 5 and Section 6 will discuss other protocols like HotStuff, Raft, PBFT, and multi-Paxos. This instrumentation defines the mapping Γ between actions of a protocol and QTree, respectively, such that the protocol is a Γ-refinement of QTree. We also discuss the properties of this instrumentation which imply that establishing Γ-refinement is an effective proof for the safety of the protocol.

The identification of linearization points relies on the fact that protocol executions pass through a number of rounds, and each round goes through several phases (rounds can run asynchronously – processes need not be in the same round at the same time). The protocol imposes a total order over the phases inside a round and among distinct rounds. Processes executing the protocol can only move forward following the total order on phases/rounds. Going from one phase to the next phase in the same round is possible if a quorum of processes send a particular type of message. The refinement proofs require identifying two quorums for each round where a value is first proposed to be agreed upon and then decided. They correspond to linearization points of successful $add(r, \_, \_)$ and $commit(r)$, respectively. The linearization point of $add(r, v, r_p) \Rightarrow OK$ occurs when intuitively, the value v is proposed as a value to agree upon in round r. For the protocols we consider, v is determined by a designated leader after receiving a set of messages from a quorum of processes. For single-decree consensus, members of the quorum send the latest round number and value they adopted (voted) in the past and the leader picks a value corresponding to the maximum round number r_p. If no one in the quorum has adopted any value yet, then the leader is free to propose any value received from a client, and r_p equals a default value 0. For state-machine replication protocols like HotStuff or Raft, the round r_p is defined in a different manner – see Section 5 (and the full version of this work [9]). The linearization point of $commit(r) \Rightarrow OK$ occurs when a quorum of nodes adopt (vote for) a value v proposed at round r.

By Theorem 1, proving that the order between linearization points along a protocol execution defines a correct QTree execution reduces to showing Properties 1–4. In general, Properties 1–3 are quite straightforward to establish and follow from the control-flow of a process. Property 3a is specific to single-decree consensus protocols or compositions thereof, e.g., (multi-)Paxos and PBFT. It will not hold for Raft or Hotstuff. Property 4 is related to the fact that any two quorums of processes intersect in a correct process.

Above, we have considered the case of a protocol that is a refinement of a single instance of QTree. State machine replication protocols that are composed of multiple independent consensus instances, e.g., PBFT (see Section 6), are refinements of a set of QTree instances (identified using a sequence number) and every linearization point needs to be associated with a certain QTree instance.

4.1 Linearization Points for Paxos

For concreteness, we exemplify the instrumentation with linearization points on the single-decree Paxos protocol. We start with a brief description of this protocol that focuses on details relevant to this instrumentation.

Paxos proceeds in rounds and each round has a unique leader. Since the set of processes running the protocol is fixed and known by every process, the leader of each round can be determined by an a-priorly fixed deterministic procedure (e.g., the leader is defined as $r \textit{ mod } N$ where r is the round number and N the number of processes). For each round, the leader acts as a proposer of a value to agree upon.

A round contains two phases. In the first phase, the leader broadcasts a `START` message to all the processes to start the round, executing the **`START`** action below, and processes acknowledge with a `JOIN` message if some conditions are met, executing the **`JOIN`** action:

- **`START` Action:** The leader p of round $r > 0$ (the proposer) broadcasts a `START`(r) message to all processes.
- **`JOIN` Action:** When a process p' receives a `START`(r) message, if p' has not sent a `JOIN` or `VOTE` message (explained below) for a higher round in the past[5], it replies by sending a `JOIN`(r) message to the proposer. This message includes the maximum round number ($maxVotedRound$) for which p' has sent a `VOTE` message in the past and the value ($maxVotedValue$) proposed in that round. If it has not voted yet, these fields are 0 and $\perp$.

[5] Each process has a local variable $maxJoinedRound$ that stores the maximal round it has joined or voted for in the past and checks whether $maxJoinedRound < r$

If the leader receives `JOIN` messages from a quorum of processes, i.e., at least $f+1$ processes from a total number of $2f+1$, the second phase starts. The leader broadcasts a `PROPOSE` message with a value, executing the **`PROPOSE`** action below. Processes may acknowledge with a `VOTE` message if some conditions are met, executing a **`VOTE`** action. If the leader receives `VOTE` messages from a quorum of processes, then the proposed value becomes decided (and sent to the client) by executing a **`DECIDE`** action:

- **`PROPOSE` Action:** When the proposer p receives `JOIN`(r) messages from a quorum of $(f+1)$ processes, it selects the one with the highest vote round number and proposes its value by broadcasting a `PROPOSE`(r) message (which includes that value). If there is no such highest round (all vote rounds are 0), then the proposer selects the proposed value randomly simulating a value given by the client (whose modeling we omit for simplicity).
- **`VOTE` Action:** When a process p' receives a `PROPOSE`(r) message, if p' has not sent a `JOIN` or `VOTE` message for a higher round in the past, it replies by sending a `VOTE`(r) message to the proposer with round number r.
- **`DECIDE` Action:** When the proposer p receives `VOTE`(r) messages from a quorum of processes, it updates a local variable called *decidedVal* to be the value it has proposed in this round r. This assignment means that the value is decided and sent to the client.

Linearization points in Paxos. We instrument Paxos with linearization points as follows:

– the linearization point of $add(r, v, r') \Rightarrow OK$ occurs when the proposer broadcasts the `PROPOSE`(r) message containing value v after receiving a quorum of `JOIN`(r) messages (during the **`PROPOSE`** action in round r). The round r' is extracted from the `JOIN`(r) message selected by the proposer.
– the linearization point of $commit(r) \Rightarrow OK$ occurs when the leader of round r updates *decidedVal* after receiving a quorum of `VOTE`(r) messages (during the **`DECIDE`** Action).

We illustrate the definition of linearization points for Paxos in relation to QTree executions in the full version [9].

Theorem 2. *Paxos refines QTree.*

Proof. We show that the sequence of successful *add* and *commit* invocations defined by linearization points along a Paxos execution satisfies the properties in Theorem 1 and therefore, it represents a correct QTree execution:

– **Property 1:** Each round has a unique leader and the leader follows the rules of the protocol (no Byzantine failures), thereby, making a single proposal. Therefore, the linearization point of an $add(r, \_, \_) \Rightarrow OK$ will occur at most once for a round r. Since a single value can be proposed in a round, and all processes follow the rules of the protocol, they can only vote for that single value. Thus, at most one linearization point of $commit(r) \Rightarrow OK$ can occur for a round r.

- **Property 2:** This holds trivially as all the processes follow the rules of the protocol and they need to receive a PROPOSE(r) message (which can occur only after the linearization point of an $\mathit{add}(r, \_, \_) \Rightarrow OK$) from the leader of round r to send a VOTE(r) message.
- **Property 3:** By the definition of the **PROPOSE** action, the proposer selects a highest vote round number r' from a quorum of JOIN(r) messages that it receives, before broadcasting a PROPOSE(r) message. If such a highest vote round number $r' > 0$ exists, then there must be a VOTE(r') message which is a reply to a PROPOSE(r') message. Thus, if the linearization point of $\mathit{add}(r, \_, r') \Rightarrow OK$ occurs where $r' \neq 0$, then it is preceded by $\mathit{add}(r', \_, \_)$. Also, by the definition of **JOIN**, a process can not send a JOIN(r) message after a VOTE(r') message if $r \not> r'$.
  - **Property 3a:** By the definition of **PROPOSE**, the proposer selects the JOIN message with the highest vote round number and proposes its value. Thus, if the linearization points of both $\mathit{add}(r, v, r') \Rightarrow OK$ and $\mathit{add}(r', v', \_) \Rightarrow OK$ occur, then $v = v'$.
- **Property 4:** Assume by contradiction that the linearization point of *commit* $(r) \Rightarrow OK$ occurs along with the linearization points of $\mathit{add}(r, \_, \_) \Rightarrow OK$ and $\mathit{add}(r', \_, r'') \Rightarrow OK$, for some $r'' < r < r'$. The linearization point of $\mathit{commit}(r)$ occurs because of a quorum of VOTE(r) messages sent by a set of processes P_1, and $\mathit{add}(r', \_, r'')$ occurs because of a quorum of JOIN(r') messages sent by a set of processes P_2. Since P_1 and P_2 must have a non-empty intersection, by the definition of **JOIN**, it must be the case that $r'' \geq r$, which contradicts the hypothesis.

The proof of Property 4 relies exclusively on the quorum of processes in the first phase of a round intersecting the quorum of processes in the second phase of a round. It is not needed that quorums in first, resp., second, phases of different rounds intersect. This observation is at the basis of an optimization that applies to non-Byzantine protocols like Flexible Paxos [18] or Raft (see the full version [9]).

4.2 Inferring Safety

The main idea behind these linearization points is that successful *add* and *commit* invocations correspond to some process doing a step that witnesses for the receipt a quorum of messages sent in a certain phase of a round. Intuitively, linearization points of successful *add* invocation occur when some process in some round is certain that a quorum of processes received or will receive the same proposal (same value, parent etc.) for the same round and acts accordingly (sends a message). Such proposal on a value v in a round r is denoted by the linearization point of successful $\mathit{add}(r, v, r')$ for some r'. On the other hand, the linearization point of a successful $\mathit{commit}(r)$ invocation occurs when a process decides on a value in round r (e.g., after receiving a quorum of votes). Formally, if we denote the actions of a protocol that correspond to linearization points of successful $\mathit{add}(r, v, r')$ and $\mathit{commit}(r)$ invocations using a_a and a_c, respectively, then $\Gamma(a_a) = \mathit{add}(r, v, r') \Rightarrow OK$ and $\Gamma(a_c) = \mathit{commit}(r) \Rightarrow OK$.

When the protocol is such a Γ-refinement of QTree, then, it satisfies agreement and validity. If a decision on a value v in a round r of a protocol is the linearization point of a successful $commit(r)$, then by Theorem 1, the corresponding QTree state contains a node n with n.round $= r$, n.value $= v$, and n.status $=$ `COMMITTED`. For single-decree consensus, Proposition 3 ensures that all rounds decide on the same value. For state machine replication protocols like Raft and HotStuff, where the goal is to agree on a sequence of commands, Proposition 2 ensures that all the decided values lie on the same branch of the tree which ensures that all processes agree on the same sequence of commands.

For validity, when $valueConstraint(n)$ is considered, successful $add(r, v, 0)$ invocations represent proposals of client values. Theorem 1 ensures that these invocations correspond to nodes n that are immediate children of *Root* and for any such node n, n.value $= v$. Therefore, by Proposition 1, we can conclude that only client values can be decided. When $valueConstraint(n)$ is not considered, the fact that the value of each node is obtained from a client is ensured using additional mechanisms that are straightforward, e.g., a client broadcasting a command to all the participants in the protocol.

5 HotStuff Refines QTree

We present an instrumentation of HotStuff with linearization points of successful *add* and *commit* invocations. We use HotStuff as an example of a state machine replication protocol where processes agree over a sequence of commands to execute, and any new command proposed by a leader to the other processes comes with a well-identified immediate predecessor in this sequence. Agreement over a command entails agreement over all its predecessors in the sequence. This is different from protocols such as multi-Paxos or PBFT, discussed in the next section, where commands are associated to indices in the sequence and they can be agreed upon in any order. Instrumentation of Raft is presented in the full version [9] and behaves in a similar manner.

In HotStuff, f out of a total of $N = 3f + 1$ processes might be Byzantine in the sense that they might show arbitrary behavior and send corrupt or spurious messages. However, they are limited by cryptographic protocols. HotStuff requires that messages are signed using public-key cryptography, which implies that Byzantine processes cannot imitate messages of correct (non-faulty) processes. Additionally, after receiving a quorum of messages, leaders must include certificates in their own messages to prove that a quorum has been reached. These certificates are constructed using threshold signature schemes and correct processes will not accept any message from the leader if it is not certified. Because of Byzantine processes, HotStuff requires quorums of size of $2f + 1$ which ensures that the intersection of any two quorums contains at least one correct process.

Each process stores a tree of commands. When a node in this tree (representing some command) is decided, all the ancestors of this node in the tree (nodes on the same branch) are also decided. For a node to become decided, a leader must receive a quorum of messages in 3 consecutive phases after the proposal. After each quorum is established, the leader broadcasts a different certificate to state which quorum has been achieved and the processes update different local variables accordingly, with the same node (if the certificate is valid). These local variables are *preNode*, *votedNode* and *decidedNode* in the order of quorums.

To start a new round, processes send their *preNode*'s to the leader of the next round in `ROUND-CHANGE(r)` messages and increment their round number. After getting a quorum of messages and selecting the *preNode* with the highest round, the leader broadcasts a `PROPOSE(r)` message with a new node (value is taken from the client) whose parent is the selected *preNode*. When the message is received by a process, it first checks if the new node extends the selected *preNode*. Then it accepts the new node if the node extends its own *votedNode* (it is a descendant of *votedNode* in the tree) or it has a higher round number than the round number of its *votedNode*, and sends[6] a `JOIN(r)` message with the same content. In the second (resp., third) phase, if a quorum of `JOIN(r)` (resp., `PRECOMMIT_VOTE(r)`) messages is received by the leader, it broadcasts a `PRE-COMMIT(r)` (resp., `COMMIT(r)`) message, and processes update their *preNode* (resp., *votedNode*) with the new node, sending a `PRECOMMIT_VOTE(r)` (resp., `COMMIT_VOTE(r)`) message. In the fourth phase, when the leader receives a quorum of `COMMIT_VOTE(r)`, it broadcasts a `DECIDE(r)` message and processes update their *decidedNode* accordingly. See the full version [9] for more details.

For HotStuff, the linearization points of *add* and *commit* occur with the broadcasts of `PRECOMMIT`(r) and `DECIDE`(r) messages, respectively, that are *valid* , i.e., (1) they contain certificates for quorums of `JOIN(r)` or `COM-MIT_VOTE(r)` messages, respectively, which respect the threshold signature scheme, and (2) they contain the same node as in those messages. More precisely,

- the linearization point of $add(r, v, r') \Rightarrow OK$ occurs the *first* time when a *valid* `PRECOMMIT`(r) message containing node v is sent. r' is the round of the node which is the parent of v and it is contained in a previous `PROPOSE(r)` message (r' can be 0 in which case parent of v is a distinguished root node that exists in the initial state).
- the linearization point of $commit(r) \Rightarrow OK$ occurs the *first* time when a *valid* `DECIDE`(r) message is sent.

Note that a Byzantine leader can send multiple *valid* `PRECOMMIT`(r) messages that include certificates for different quorums of `JOIN(r)` messages. A linearization point occurs when the first such message is sent. Even if processes reply to another valid `PRECOMMIT`(r) message sent later, this later `PRECOMMIT`(r) message contains the same *preNode* value, and their reply will have the same content. The same holds for `DECIDE`(r) messages. This remark along with the restriction

[6] For all received messages, a correct process also checks if the round number of the node sent by the leader is equal to the current round number of its own, and can send only one message for each phase in each round.

to valid messages and the fact that any two quorums intersect in at least one correct process implies that the sequence of successful *add* and *commit* invocations defined by these linearization points satisfies the properties in Theorem 1 and therefore,

Theorem 3. *HotStuff refines QTree.*

A detailed proof of the theorem above is given in the full version [9].

6 PBFT Refines QTree

The protocols discussed above are refinements of a *single* instance of QTree. State-machine replication protocols based Multi-decree consensus like Multi-Paxos or PBFT can be seen as compositions of a number of single-decree consensus instances that run concurrently, one for each index in a sequence of commands to agree upon, and they are refinements of a set of independent QTree instances. We describe the instrumentation of PBFT and delegate multi-Paxos (and variants) to the full version [9].

PBFT is a multi-decree consensus protocol in which processes aim to agree on a sequence of values. As in HotStuff, f out of a total number of $3f + 1$ processes might be Byzantine and quorums are of size at least $2f + 1$. To ensure authentication, messages are signed using public-key cryptography. Messages sent after receiving a quorum of messages in a previous phase include that set of messages as a certificate.

A new round r starts with the leader receiving a quorum of ROUND-CHANGE(r) messages (like in HotStuff). Each such message from a process p includes the VOTE message with the highest round (similarly to the **JOIN** action of Paxos) that p sent in the past, for each sequence number that is not yet agreed by a quorum. For an arbitrary set of sequence numbers sn, the leader selects the VOTE message with the highest round and broadcasts a PROPOSE(r,sn) message that includes the same value as in the VOTE message or a value received from a client if there is no such highest round. As mentioned above, this message also includes the VOTE messages that the leader received as a certificate for the selection. When a process receives a PROPOSE(r,sn) message, if r equals its current round, the process did not already acknowledge a PROPOSE(r,sn) message, and the value proposed in this message is selected correctly w.r.t. the certificate, then it broadcasts a JOIN(r,sn) message with the same content (this is sent to all processes not just the leader). If a quorum of JOIN(r,sn) messages is received by a process, then it broadcasts a VOTE(r,sn) message with the same content. If a process receives a quorum of VOTE(r,sn) messages, then the value in this message is decided for sn. When a process sends its highest round number VOTE messages to the leader of the next round (in ROUND-CHANGE messages), it also includes the quorum of JOIN messages that it received before sending the VOTE, as a certificate.

PBFT is a refinement of a set of independent QTree instances, one instance for each sequence number. The linearization points will refer to a specific instance identified using a sequence number, e.g., $sn.add(r, v, r')$ denotes an $add(r, v, r')$ invocation on the QTree instance sn. Therefore,

- the linearization point of $sn.add(r, v, r') \Rightarrow OK$ occurs the *first* time when a process p sends a `VOTE`(r, sn) message, assuming that p is *honest*, i.e., it already received a quorum of `JOIN`(r, sn) messages with the same content. v is the value of the `VOTE`(r', sn) message that is included in the `PROPOSE`(r,sn) message (it is possible that $r' = 0$ and v is selected randomly).
- the linearization point of $sn.commit(r) \Rightarrow OK$ occurs the *first* time when a process p decides a value for sn, assuming that p is *honest*, i.e., it already received a quorum of `JOIN`(r, sn), resp., `VOTE`(r, sn), messages with the same content.

A protocol refines a set of QTree instances identified using sequence numbers when it satisfies Properties 1-4 in Theorem 1 for each sequence number, e.g., Property 1 becomes for every sn and every r, a protocol execution contains a linearization point for at most one invocation $sn.add(r, \_, \_) \Rightarrow OK$ and at most one invocation $sn.commit(r) \Rightarrow OK$. A detailed proof of the following theorem is given in the full version [9].

Theorem 4. *PBFT refines a composition of independent QTree instances.*

7 Discussion

Protocols considered in this work can be grouped under three classes: single-decree consensus (Paxos), multi-decree consensus (PBFT, Multi-Paxos) and state machine replication (Raft, HotStuff)[7]. We show that they all refine QTree: a single instance for Paxos and HotStuff, and a set of independent instances (one for each sequence number in a command log) for PBFT, Multi-Paxos, and Raft. The more creative parts of the refinement proofs are the identification of *add* and *commit* linearization points and establishing Property 4 in Theorem 1 which follows from the intersection of quorums achieved in different phases of a round. The other 3 properties in Theorem 1 which guarantee that the linearization points are correct are established in a rather straightforward manner, based on the control-flow of a process participating to the protocol.

The linearization points of successful *add* and *commit* invocations correspond to some process doing a step that witnesses for the receipt a quorum of messages sent in a certain phase of a round, e.g., the leader broadcasting a `PROPOSE`(r) message in Paxos entails that a quorum of `JOIN`(r) messages have been sent in the first phase and received. Protocols vary in the total number of phases in a round, and the phases for which quorums of sent messages should be received in order to have a linearization point of *add* or *commit*. A summary is presented in Table 1. The * on the total number of phases means that the first phase is skipped in rounds where the leader is stable. For Multi-Paxos and Raft, if the first phase

[7] This is a slight abuse of terminology since multi-decree consensus protocols are typically used to implement state machine replication.

is skipped, then the linearization point of an *add* is determined by a quorum of received messages sent in the next phase (and coincides with the linearization point of a *commit*). We use "1/2" to denote this fact. In PBFT and HotStuff, due to Byzantine processes, quorums of messages sent in two consecutive phases need to be received in order to ensure that the processes are going to vote on the same valid proposal. The 3rd phase in HotStuff is used to ensure progress and can be omitted when reasoning only about safety.

Table 1: Summary of linearization point definitions. For each protocol, we give the total number of phases in a round and the number of the phase for which a quorum of sent messages should be received in order to have a linearization point of *add* or *commit*.

Class	Protocol	#Phases	*add* Quorum Pha.	*commit* Quorum Pha.
Single-Decree Cons.	Paxos	2	1	2
Multi-Decree Cons.	Multi-Paxos	2*	1/2	2
	PBFT	3*	2	3
State Machine Repl.	Raft	2*	1/2	2
	HotStuff	4	2	4

8 Conclusion and Related Work

We have proposed a new methodology for proving safety of consensus or state-machine replication protocols, which relies on a novel abstraction of their dynamics. This abstraction is defined as a sequential QTree object whose state represents a global view of a protocol execution. The operations of QTree construct a tree structure and model agreement on values or a sequence of state-machine commands as agreement on a fixed branch in the tree. Our methodology applies uniformly to a range of protocols like (multi-)Paxos, HotStuff, Raft, and PBFT. We believe that this abstraction helps in improving the understanding of such protocols and writing correct implementations or optimizations thereof.

As a limitation, it is not clear whether QTree applies to protocols such as Texel [31] which do not admit a decomposition in rounds. As future work, we might explore the use of QTree in reasoning about liveness. This would require some fairness condition on infinite sequences of add/commit invocations, and a suitable notion of refinement which ensures that infinite sequences of protocol steps cannot be mapped to infinite sequences of stuttering QTree steps.

The problem of proving the correctness of such protocols has been studied in previous work. We give an overview of the existing approaches that starts with safety proof methods based on refinement, which are closer to our approach.

Refinement based safety proofs. Verdi [35] is a framework for implementing and verifying distributed systems that contains formalizations of various network semantics and failure models. Verdi provides *system transformers* useful for refining high-level specifications to concrete implementations. As a case study, it includes a fully-mechanized correctness proof of Raft [36]. This proof consists of 45000 lines of proof code (manual annotations) in the Coq language for a 5000 lines RAFT implementation, showing the difficulty of reasoning on consensus protocols and the manual effort required. Iron Fleet [17] uses TLA [22] style transition-system specifications and refine them to low-level implementations described in the Dafny programming language [25]. Boichat et al. [3] defines a class of specifications for consensus protocols, which are more abstract than QTree and can make correctness proofs harder. Proving Paxos in their case is reduced to a linearizability proof towards an abstract specification, which is quite complex because the linearization points are *not fixed*, they depend on the future of an execution. As a possibly superficial quantitative measure, their Paxos proof reduces to 7 lemmas that are formalized by Garcia-Perez et al. [12, 13] in 12 pages (see Appendix B and C in [13]), much more than our QTree proof. Our refinement proof is also similar to a linearizability proof, but the linearization points in our case are *fixed* (do not depend on the future of an execution) which brings more simplicity. In principle, the specifications in [3] could apply to more protocols, but we are not aware of such a case. The inductive sequentialization proof rule [20] is used for a fully mechanized correctness proof of a realistic Paxos implementation. This implementation is proved to be a refinement of a *sequential* program which is quite close to the original implementation, much less abstract than QTree, and relies on commutativity arguments implied by the communication-closed round structure [11]. A similar idea is explored in [14], but in a more restricted context.

Inductive invariant based safety proofs. Ivy [30] is an SMT-based safety verification tool that can be used for verifying inductive invariants about global states of a distributed protocol. In order to stay in a decidable fragment of first-order logic, both the modeling and the specification language of IVY are restricted. A simple model of Paxos obeying these restrictions is proven correct in [29].

Beyond safety. The TLA+ infrastructure [22] of Lamport has been used to verify both safety and liveness (termination) of several variations of Paxos, e.g., Fast Paxos [23] or Multi-Paxos [6]. Bravo et al. [4] introduce a generic synchronization mechanism for round changes, called the view synchronizer, which guarantees liveness for various Byzantine consensus protocols including our cases studies HotStuff and PBFT. This work includes full correctness proofs for single-decree versions of HotStuff and PBFT and a two-phase version of HotStuff. PSync [10] provides a partially synchronous semantics for distributed protocols assuming communication-closed rounds in the Heard-Of model [8]. PSync is used to prove both safety and liveness of a Paxos-like consensus protocol called *lastVoting*.

Relating different consensus protocols. Lamport defines a series of refinements of Paxos that leads to a Byzantine fault tolerant version, which is refined by PBFT [24]. Our proof that Paxos refines QTree can be easily extended to this Byzantine fault tolerant version in the same manner as we did for PBFT.

Wang et al. [34] shows that a variation of RAFT is a refinement of Paxos, which enables porting some Paxos optimizations to RAFT. Renesse et al. [32] compare Paxos, Viewstamped Replication [28] and ZAB [19]. They define a rooted tree of specifications represented in TLA style whose leaves are concrete protocols. Each node in this tree is refined by its children. Common ancestors of concrete protocols show similarities whereas conflicting specifications show the differences. Similarly, [33] shows that Paxos, Chandra-Toueg [7] and Ben-Or [2] consensus algorithms share common building blocks. Aublin et al. [1] propose an abstract data type for specifying existing and possible future consensus protocols. Unlike our QTree, core components of this data type are not implemented and intentionally left abstract so that it can adapt to different network and process failure models.

References

1. Aublin, P., Guerraoui, R., Knezevic, N., Quéma, V., Vukolic, M.: The next 700 BFT protocols. ACM Trans. Comput. Syst. **32**(4), 12:1–12:45 (2015). https://doi.org/10.1145/2658994, https://doi.org/10.1145/2658994
2. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: Probert, R.L., Lynch, N.A., Santoro, N. (eds.) Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983. pp. 27–30. ACM (1983). https://doi.org/10.1145/800221.806707, https://doi.org/10.1145/800221.806707
3. Boichat, R., Dutta, P., Frølund, S., Guerraoui, R.: Deconstructing paxos. SIGACT News **34**(1), 47–67 (2003). https://doi.org/10.1145/637437.637447, https://doi.org/10.1145/637437.637447
4. Bravo, M., Chockler, G.V., Gotsman, A.: Making byzantine consensus live. In: Attiya, H. (ed.) 34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference. LIPIcs, vol. 179, pp. 23:1–23:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.DISC.2020.23, https://doi.org/10.4230/LIPIcs.DISC.2020.23
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Seltzer, M.I., Leach, P.J. (eds.) Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999. pp. 173–186. USENIX Association (1999), https://dl.acm.org/citation.cfm?id=296824
6. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 119–136 (2016). https://doi.org/10.1007/978-3-319-48989-6_8, https://doi.org/10.1007/978-3-319-48989-6_8
7. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996). https://doi.org/10.1145/226643.226647, https://doi.org/10.1145/226643.226647

8. Charron-Bost, B., Merz, S.: Formal verification of a consensus algorithm in the heard-of model. Int. J. Softw. Informatics **3**(2-3), 273–303 (2009), http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=273&flag=1
9. Cirisci, B., Enea, C., Mutluergil, S.O.: Quorum tree abstractions of consensus protocols (2023). https://doi.org/10.48550/ARXIV.2301.09946, https://arxiv.org/abs/2301.09946
10. Dragoi, C., Henzinger, T.A., Zufferey, D.: Psync: a partially synchronous language for fault-tolerant distributed algorithms. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 400–415. ACM (2016). https://doi.org/10.1145/2837614.2837650, https://doi.org/10.1145/2837614.2837650
11. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. **2**(3), 155–173 (1982). https://doi.org/10.1016/0167-6423(83)90013-8, https://doi.org/10.1016/0167-6423(83)90013-8
12. García-Pérez, Á., Gotsman, A., Meshman, Y., Sergey, I.: Paxos consensus, deconstructed and abstracted. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801, pp. 912–939. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_32, https://doi.org/10.1007/978-3-319-89884-1_32
13. García-Pérez, Á., Gotsman, A., Meshman, Y., Sergey, I.: Paxos consensus, deconstructed and abstracted (extended version). CoRR **abs/1802.05969** (2018), http://arxiv.org/abs/1802.05969
14. von Gleissenthall, K., Kici, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: synchronous verification of asynchronous distributed programs. Proc. ACM Program. Lang. **3**(POPL), 59:1–59:30 (2019). https://doi.org/10.1145/3290372, https://doi.org/10.1145/3290372
15. Golab, W.M., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: Fortnow, L., Vadhan, S.P. (eds.) Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011. pp. 373–382. ACM (2011). https://doi.org/10.1145/1993636.1993687, https://doi.org/10.1145/1993636.1993687
16. Gray, J., Lamport, L.: Consensus on transaction commit. CoRR **cs.DC/0408036** (2004), http://arxiv.org/abs/cs.DC/0408036
17. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015. pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428, https://doi.org/10.1145/2815400.2815428
18. Howard, H., Malkhi, D., Spiegelman, A.: Flexible paxos: Quorum intersection revisited. CoRR **abs/1608.06696** (2016), http://arxiv.org/abs/1608.06696
19. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011. pp. 245–256.

IEEE Compute Society (2011). https://doi.org/10.1109/DSN.2011.5958223, https://doi.org/10.1109/DSN.2011.5958223

20. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 227–242. ACM (2020). https://doi.org/10.1145/3385412.3385980, https://doi.org/10.1145/3385412.3385980
21. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229, https://doi.org/10.1145/279227.279229
22. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002), http://research.microsoft.com/users/lamport/tla/book.html
23. Lamport, L.: Fast paxos. Distributed Comput. **19**(2), 79–103 (2006). https://doi.org/10.1007/s00446-006-0005-x, https://doi.org/10.1007/s00446-006-0005-x
24. Lamport, L.: Byzantizing paxos by refinement. In: Peleg, D. (ed.) Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6950, pp. 211–224. Springer (2011). https://doi.org/10.1007/978-3-642-24100-0_22, https://doi.org/10.1007/978-3-642-24100-0_22
25. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20, https://doi.org/10.1007/978-3-642-17511-4_20
26. Malkhi, D., Lamport, L., Zhou, L.: Stoppable paxos. Tech. Rep. MSR-TR-2008-192 (April 2008)
27. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2008), https: //bitcoin.org/bitcoin.pdf
28. Oki, B.M., Liskov, B.: Viewstamped replication: A general primary copy. In: Dolev, D. (ed.) Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988. pp. 8–17. ACM (1988). https://doi.org/10.1145/62546.62549, https://doi.org/10.1145/62546.62549
29. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA), 108:1–108:31 (2017). https://doi.org/10.1145/3140568, https://doi.org/10.1145/3140568
30. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 614–630. ACM (2016). https://doi.org/10.1145/2908080.2908118, https://doi.org/10.1145/2908080.2908118
31. van Renesse, R.: Asynchronous consensus without rounds. CoRR **abs/1908.10716** (2019), http://arxiv.org/abs/1908.10716
32. van Renesse, R., Schiper, N., Schneider, F.B.: Vive la différence: Paxos vs. viewstamped replication vs. zab. IEEE Trans. Dependable Secur.

Comput. **12**(4), 472–484 (2015). https://doi.org/10.1109/TDSC.2014.2355848, https://doi.org/10.1109/TDSC.2014.2355848

33. Song, Y.J., van Renesse, R., Schneider, F.B., Dolev, D.: The building blocks of consensus. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) Distributed Computing and Networking, 9th International Conference, ICDCN 2008, Kolkata, India, January 5-8, 2008. Lecture Notes in Computer Science, vol. 4904, pp. 54–72. Springer (2008). https://doi.org/10.1007/978-3-540-77444-0_5, https://doi.org/10.1007/978-3-540-77444-0_5
34. Wang, Z., Zhao, C., Mu, S., Chen, H., Li, J.: On the parallels between paxos and raft, and how to port optimizations. In: Robinson, P., Ellen, F. (eds.) Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019. pp. 445–454. ACM (2019). https://doi.org/10.1145/3293611.3331595, https://doi.org/10.1145/3293611.3331595
35. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 357–368. ACM (2015). https://doi.org/10.1145/2737924.2737958, https://doi.org/10.1145/2737924.2737958
36. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: Avigad, J., Chlipala, A. (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016. pp. 154–165. ACM (2016). https://doi.org/10.1145/2854065.2854081, https://doi.org/10.1145/2854065.2854081
37. Yin, M., Malkhi, D., Reiter, M.K., Golan-Gueta, G., Abraham, I.: Hotstuff: BFT consensus with linearity and responsiveness. In: Robinson, P., Ellen, F. (eds.) Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019. pp. 347–356. ACM (2019). https://doi.org/10.1145/3293611.3331591, https://doi.org/10.1145/3293611.3331591