Consensus Refined

Ognjen Marić, Christoph Sprenger, and David Basin
Institute of Information Security, Department of Computer Science
ETH Zurich, Switzerland

***Abstract*—Algorithms for solving the consensus problem are fundamental to distributed computing. Despite their brevity, their ability to operate in concurrent, asynchronous, and failure-prone environments comes at the cost of complex and subtle behaviors. Accordingly, understanding how they work and proving their correctness is a non-trivial endeavor where abstraction is immensely helpful. Moreover, research on consensus has yielded a large number of algorithms, many of which appear to share common algorithmic ideas. A natural question is whether and how these similarities can be distilled and described in a precise, unified way. In this work, we combine stepwise refinement and lockstep models to provide an abstract and unified view of a sizeable family of consensus algorithms. Our models provide insights into the design choices underlying the different algorithms, and classify them based on those choices. All our results are formalized and verified in the theorem prover Isabelle/HOL, yielding precision and strong correctness guarantees.**

I. Introduction

Distributed consensus is a fundamental problem in distributed computing: a fixed set of processes must agree on a single value from a set of proposed ones. Algorithms that solve this problem provide building blocks for many higher-level tasks, such as distributed leases, group membership, atomic broadcast (also known as total-order broadcast or multi-consensus), and so forth. These in turn provide building blocks for yet higher-level tasks like system replication. In this paper, however, our focus is on consensus algorithms proper, rather than their applications (such as Multi-Paxos [23] or Zab [21]). Namely, we consider consensus algorithms for the asynchronous message-passing setting with benign link and process failures.

Although the setting we consider explicitly excludes malicious behavior, the interplay of concurrency, asynchrony, and failures can still drive the execution of any proposed consensus algorithm in many different ways. This makes the consensus problem not only difficult, but even impossible to solve deterministically [15]. Partial synchrony [14] imposes just enough constraints on the asynchrony to admit solutions, but still retains the main features of the fully asynchronous model. Hence the number of possible executions of an algorithm in this setting is still immense, making the understanding of both the algorithms and their correctness non-trivial. This understanding can be greatly aided by appropriate *abstractions* that simplify the algorithms or the setting, making the development of such abstractions an appealing research topic.

As examples of work on algorithm simplification, [5], [6], [9], [26], [32] all provide more abstract descriptions of Lamport's seminal Paxos algorithm [22], [23]. Another line of work [12], [16] provides an abstraction of the asynchronous (or partially synchronous) setting for the class of algorithms operating in communication-closed rounds. For this class, the asynchronous setting is replaced by what is essentially a synchronous model weakened by message loss (dual to strengthening the asynchronous model by failure detectors [33]). As the resulting models provide the illusion that all the processes operate in lockstep, we refer to them as *lockstep models*.

More than 30 years of research on consensus has also yielded a large collection of consensus algorithms. Many of them appear to share similar underlying algorithmic ideas, although their presentation, structure, and details differ. A natural question is whether their similarities can be distilled and captured in a uniform and generic way, and this has led to another substantial body of work [17]–[19], [29], [30], [34], [35]. In the same vein, one may ask whether the algorithms can be classified by some natural criteria.

Approach Taken

Given the situation outlined above, we see a clear need for (i) abstraction and simplification and (ii) unification and classification of consensus algorithms, in order to understand their essence and relationships. Additionally, as the setting they operate in is complex, it is necessary that both (i) and (ii) are addressed in a precise and correct manner. We thus add to our wish list (iii) precision and correctness guarantees.

We address these issues by combining three elements. First, we describe consensus algorithms using *stepwise refinement*. In this method, we derive an algorithm through a sequence of models. The initial models in the sequence can describe the algorithm in arbitrarily abstract terms. In our abstractions, we describe the system using non-local steps that depend on the states of multiple processes, removing the need for communication. These abstractions allow us to focus on the main algorithmic ideas, without getting bogged down in details, thereby providing simplicity. We then gradually introduce details in successive, more concrete models that refine the abstract ones. In order to be implementable in a distributed setting, the final models must use strictly local steps, and communicate only by message passing. The link between the abstract and concrete models is precisely described and proved using *refinement relations*. Furthermore, the same abstract model can be implemented by different algorithms. This results in a *refinement tree* of models, where branching corresponds to different implementations as illustrated in Figure 1. This tree captures the relationships between the different consensus algorithms, found at its leaves, providing a natural classification of the algorithms. The use of refinement thus addresses the points (i) and (ii) raised above.

Second, since the distributed algorithms we derive operate in communication-closed rounds, we employ lockstep models to describe these algorithms in a synchronous fashion. This not only simplifies our models of concrete algorithms, it also

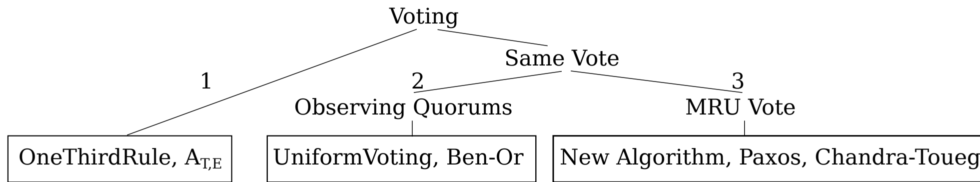

Figure 1. The consensus family tree. Boxes contain models of concrete algorithms.

further increases the abstraction level in our initial models. It allows our non-local steps to not only depend on, but also change the state of multiple processes. Hence, this choice further helps with point (i). Yet our results translate to the asynchronous setting of the real world, thanks to the preservation result established in [11].

Finally, we have formalized [28] all the models we present in the theorem prover Isabelle/HOL [31], using the Isabelle formalization of the Heard-Of model by Debrat and Merz [13]. We used Isabelle to prove the desired properties of our models and establish refinement relations between them without assuming any fixed bounds on either the number of processes or rounds. This provides us with strong guarantees about the precision and correctness of our results, addressing point (iii).

Related Work

The technique of stepwise refinement is well-known in the distributed systems community, and has already been successfully applied to consensus algorithms [8], [25], [26]. Lampson's refinement-based descriptions [26] of the Paxos algorithm were even used as blueprints for the consensus portion of Zab [21]. Somewhat surprisingly, the application of refinement has been limited to variants of single algorithms, and there appears to be no work that derives entire families of different algorithms, as we do here.[1] Moreover, there is also no prior work that combines refinement with lockstep models.

The existing work on unifying consensus algorithms [17]–[19], [29], [30], [34], [35] provides generic algorithms that can be instantiated with different parameters and primitives. However, with the exception of [18], these generic algorithms do not abstract communication away, and thus are significantly more detailed and complicated than our abstract models. Furthermore, their scope is limited. They each cover at most one of our three classes of algorithms (i.e., the leaf nodes in Figure 1) with the exception of [35] and [34], which, when limited to benign failures, each cover algorithms from two classes. Another limitation of these generic algorithms is that they have limited power to explain the relationships between the different algorithms. The only classification of algorithms is offered in [34]. However, it is more technical and less focused on design choices than ours. Finally, none of these works have been fully formalized, and they contain numerous parts with missing proofs or just proof sketches.

Contributions

We see the contributions of our work as follows. First, in contrast to previous work, our combined use of stepwise refinement, lockstep models, and formal modeling and reasoning addresses all three of the above desiderata. The refinement approach provides a natural framework for the abstraction, unification, and classification of a variety of algorithms, the lockstep model additionally increases the abstraction level, and the formality of our approach provides the desired precision and correctness guarantees.

Second, our abstract models provide insight into different classes of consensus algorithms by laying bare the underlying mechanisms in a clear, simple, and concise way. These models consist of a single non-deterministic event modeling a round of the algorithm. The enabling predicates of this event precisely capture the safety conditions needed to preserve agreement.

Third, the branching points in our refinement tree directly reflect the design choices behind the different types of algorithms. We classify the algorithms (Figure 1) along three main branches: (1) algorithms that allow multiple values per round (OneThirdRule [12] and $A_{T,E}$ [4] are representatives), (2) algorithms that allow only a single value per round and depend on waiting and observations (Ben-Or's algorithm [3] and UniformVoting [12]) and (3) algorithms that allow only a single value and require no additional information (Paxos [22] and the $\Diamond\mathcal{S}$-based algorithm of Chandra and Toueg [10]). For the benign setting, this development includes more algorithms and algorithm types than any other one presented in the literature, testifying to the flexibility of our approach.

Finally, while we focus on existing algorithms, we also derive a new one. Our development provided insights that allowed us to affirmatively answer a question raised in [12], asking whether there exists a leaderless consensus algorithm that requires no waiting to provide safety, while tolerating up to $\frac{N}{2}$ process failures.

Outline: In Section II, we provide background on our modeling languages, refinement, and assumptions about failures and network behavior. We review the consensus problem in Section III. Sections IV–VIII, which constitute the main part of our paper, follow the path traced out by Figure 1; each section covers a single abstract model, together with a sample concrete algorithm, where applicable. We conclude in Section IX, where we also discuss the limitations of our approach and describe future work.

II. System Specifications and Refinement

We introduce generic event-based system specifications and a notion of refinement akin to [1], [2]. We then show how we specialize them to model distributed systems, and how we model failures and network behavior.

A. Event-based Systems

All of our specifications have a semantics in terms of unlabeled state transition systems $T = (S, S^0, \rightarrow)$, henceforth

[1]Note that van Renesse et al. [36] derive families of replication algorithms at a higher abstraction layer.

simply called *systems*, where S is a (possibly infinite) set of states, S^0 is a (non-empty) subset of initial states, and $\rightarrow \subseteq S \times S$ is a transition relation. We write $s \rightarrow s'$ for transitions $(s, s') \in \rightarrow$. For convenience, we specify system states by a record containing the system's variables and its transitions by a set of parameterized *events*. An event is specified by a *guard* and an *action*. The guard is a predicate defining when the event is enabled in a state. The action describes a relation between the source and the target state, specified as a state update. Here is a prototypical event with a vector of parameters $\overline{a}$, a guard G, and an action that simultaneously updates the state variables $\overline{x}$ using the update functions $\overline{g}$, one for each variable in $\overline{x}$:

Event evt($\overline{a}$):
Guard
    $G(\overline{x}, \overline{a})$
Action
    $\overline{x} := \overline{g}(\overline{x}, \overline{a})$

An event $evt(\overline{a})$ has a straightforward relational semantics, denoted by $\rightarrow_{evt(\overline{a})}$. The system's transition relation $\rightarrow$ is the union over all relations induced by the events.

The semantics of a system T is given by its set of *traces*. A trace is any finite sequence of states from S. We sometimes view traces as partial functions $tr : \mathbb{N} \rightharpoonup S$, whose domain $dom(tr)$ is an initial segment of $\mathbb{N}$. The traces of T, written $traces(T)$, are those obtained by starting from an initial state and taking a finite number of steps based on the enabled events.

B. Properties and Refinement

A *property* is a set of traces. For example, the agreement property specifies the traces in which no two processes decide on different values. A system T *satisfies* a given property ϕ if all its traces are included in the property, i.e., $traces(T) \subseteq \phi$.

Due to their trace semantics, we can also view systems as properties and relate two systems by relating their sets of traces. If $traces(T_2) \subseteq traces(T_1)$, we say that T_2 *refines* T_1 or conversely, that T_1 *abstracts* T_2. We call T_2 the concrete system and T_1 the abstract system. So far, we assumed that both the system and the property (or the abstract system) use the same set of states. We can relax this assumption by providing a relation between two different sets of states. A system T with state set S satisfies a property ϕ over state set S' under the relation $R \subseteq S' \times S$ if each of its traces is an image of some trace in the property under R, i.e., $traces(T) \subseteq R(\phi)$, where $R(\phi) = \{\tau \mid \exists \sigma \in \phi.\, dom(\sigma) = dom(\tau) \land \forall i \in dom(\sigma).\, (\sigma(i), \tau(i)) \in R\}$. The meaning of property satisfaction (or refinement) now also depends on the relation R. It is easy to see that refinement is transitive: if T_2 refines T_1 under R_1, and T_3 refines T_2 under R_2, then T_3 refines T_1 under $R_2 \circ R_1$. Furthermore, if T_1 satisfies some property ϕ, then so do T_2 and T_3 under the suitable relations. This allows us to carry out *stepwise* refinement, producing a sequence (or a tree) of models. The concrete systems then immediately satisfy all the properties of the systems they refine, under suitable relations.

To prove that T_2 refines T_1 under a relation $R \subseteq S_1 \times S_2$, we employ the standard technique of *forward simulation*. This obliges us to prove two things. First, every initial state $t^0 \in S_2^0$ of the concrete system has a related abstract state $s^0 \in S_1^0$ such that $(s^0, t^0) \in R$. Second, for every step of the concrete system T_2, that is, any of its events $evt_2(\overline{a})$, the abstract system can take a related step, i.e., for all $s \in S_1$ and $t, t' \in S_2$ such that $(s, t) \in R$ and $t \rightarrow_{evt_2(\overline{a})} t'$, there exists a state $s' \in S_1$ such that $s \rightarrow_1 s'$ and $(s', t') \in R$. Often, the concrete event will refine a particular abstract one, and the second proof obligation will decompose into two parts: (1) *guard strengthening*, i.e., the concrete guard implies the abstract one and (2) *action refinement*, i.e., the updated states are also related by R.

Liveness properties are often conditioned on *fairness* assumptions, that is, they are not required to hold on all traces but only on those satisfying the fairness assumptions. Such conditional properties are not preserved by the notion of refinement introduced. However, we still need to handle them in our development. For example, termination of consensus, meaning that every process eventually decides on a value, often relies on the fairness assumption that each process, when repeatedly offered the possibility to make a decision, eventually does so. For simplicity, we avoid extending the framework and prove termination directly on the concrete models of our refinement tree (Figure 1).

C. Distributed System Models

The system specifications and refinements described so far are standard and not specific to distributed systems. We now specialize them to the distributed setting. In the rest of the paper, we assume a fixed set Π of N processes, and adopt the convention that p and q range over Π, and r over $\mathbb{N}$. In this section we also assume a set of messages M.

In all our models, the set of states S is the product of local state sets S_p for each process p, and the computation is structured into rounds where all processes make their transitions simultaneously, proceeding to the next round. Hence, these models are lockstep. Since we derive algorithms by stepwise refinement, we take the liberty of working with two types of lockstep models with different communication mechanisms, which give rise to two abstraction levels and associated views:

1. *global view*: lockstep models with direct access to all processes' states. These models clearly exhibit the central ideas underlying the algorithms and simplify reasoning about them, but implementing such models in a distributed fashion requires further refinement.
2. *local view*: lockstep models with message passing communication, including the possibility of message loss. Note that this possibility means that they do *not* correspond to synchronous distributed system models with known upper bounds on message delays. These models can directly serve as a basis for a distributed implementation.

We specify the first type of models directly as event-based specifications introduced in Section II-A. Guards can refer to the state variables of any process. Likewise, a state update may affect any process. This type of systems is used in all non-leaf models of the tree in Figure 1.

For the local view, we adopt the Heard-Of (HO) model [12], which we use to represent the concrete algorithms, i.e., the (boxed) leaf models in Figure 1. In this model, in every round, each process sends a message to every other process,

Process	$HO^r_{p_i}$	Messages received: μ^p_r
p_1	$\{p_1, p_2, p_3\}$	$\{(p_1, m_1), (p_2, m_2), (p_3, m_3)\}$
p_2	$\{p_1, p_2\}$	$\{(p_1, m_1), (p_2, m_2)\}$
p_3	$\{p_1, p_3\}$	$\{(p_1, m_1), (p_3, m_3)\}$

Figure 2. An example of filtering by HO sets within a round, for $N = 3$. For simplicity, we assume that the process broadcast messages in this round, i.e., $\mathtt{send}^r_{p_i}(s_{p_i}, \cdot) = m_i$

receives messages from a specified set of processes, and then performs a local computation step. Hence, the behavior of process p in round r is specified by a function $\mathtt{send}^r_p$, a set of processes HO^r_p, called a *heard-of set*, and a function $\mathtt{next}^r_p$, which we now explain in turn.

The function $\mathtt{send}^r_p : S_p \times \Pi \to M$ determines the messages sent by process p to the other processes. For uniformity, we assume that p sends a message to every other process. If nothing needs to be sent, p sends some predefined dummy message from M. The messages received by process p in round r are described by a partial function $\mu^r_p : \Pi \rightharpoonup M$ defined as follows: $\mu^r_p(q) = \mathtt{send}^r_q(s_q, p)$ if $q \in HO^r_p$ and is undefined otherwise (s_q is the projection of the global system state s to process q's local state). This means that p receives only the messages from the processes in the heard-of set HO^r_p, while the other messages are lost. Figure 2 gives an example. This filtering of messages captures various kinds of failures, including link failures and timeouts. Importantly, it also captures process failures, removing the need for an explicit notion of such failures in the HO model. The function $\mathtt{next}^r_p : S_p \times (\Pi \rightharpoonup M) \to 2^{S_p}$ takes process p's state s_p and the messages μ^r_p that p receives in round r and returns a set of states $\mathtt{next}^r_p(s_p, \mu^r_p)$. The successor state is chosen non-deterministically from this set. This is performed simultaneously for all processes and yields a new global system state s' for the next round. This determines the transition system semantics of heard-of models. Since each transition includes an instantaneous exchange of messages, this semantics does not require an explicit representation of the network. This greatly simplifies reasoning about these models compared to an asynchronous model.

Reality, of course, does not proceed in lockstep. Hence, there is a second, asynchronous semantics of the HO model [11]. Here, each process has its own view of the current round number. All messages carry the sender's round number and are explicitly transmitted over a network. Each process only accepts messages carrying its round number. Hence, rounds are *communication-closed*. A process receives only the messages from processes in its HO set. Once it has received all such messages, it can take a $\mathtt{next}^r_p$ transition to move on to the next round. Each process does this independently. This asynchronous semantics thus closely corresponds to the real world, where the sequence of HO sets is, however, generated dynamically, depending on when the processes decide to move on to the next round.

The theorem of [11] tells us that a certain class of so-called local properties, when proved under the lockstep semantics also hold in the asynchronous one. We exploit this result to simplify our correctness proofs, since consensus can be specified as a set of such local properties.

D. Assumptions on Failures and the Network

The result in [15] rules out a deterministic solution to the consensus problem in a completely asynchronous setting when even just a single process fails. However, there are solutions provided we are willing to either give up determinacy [3] or assume partial synchrony, where asynchronous behavior is interspersed with "good" periods of predictable behavior [14]. Moreover, we also need a bound on the number f of processes that can fail. This bound measures the *fault tolerance* of a distributed algorithm and is usually expressed as a fraction of the total number of processes N.

In the HO model, the assumptions on network behavior and fault tolerance are reflected in *communication predicates*. These predicates play a role similar to failure detectors and their properties (e.g. eventual perfect accuracy) in the more traditional asynchronous models [16]. A communication predicate $P : (\Pi \times \mathbb{N} \to 2^\Pi) \to bool$ is a predicate on heard-of sets, viewed as functions $HO : \Pi \times \mathbb{N} \to 2^\Pi$. For example, we will often use predicates $\exists r.\, P_{unif}(r)$ and $\forall r.\, P_{maj}(r)$, where:

$$P_{unif}(r) \triangleq \forall p, q.\ HO^r_p = HO^r_q, \qquad (P_{unif})$$

$$P_{maj}(r) \triangleq \forall p.\ \left|HO^r_p\right| > \frac{N}{2}. \qquad (P_{maj})$$

The first predicate ensures the existence of a round in which every process sees the same messages; the second one ensures that every process sees at least $\frac{N}{2}$ messages in each round.

Given assumptions on the network and failures, an algorithm's implementation must guarantee both that all processes eventually advance their rounds and that any sequence of HO sets generated in this way satisfies the appropriate predicate. For example, the predicate $\exists r.\, P_{unif}(r)$ can be implemented (e.g., using timeouts [20]) under the partial synchrony assumption of a global stabilization time, after which no failures occur and process speeds and message delays respect known bounds. The predicate $\forall r.\, P_{maj}(r)$ can be implemented by waiting on messages and using retransmission, assuming fair-lossy links and $f < \frac{N}{2}$. In fact, since the HO model has no notion of explicit process failure, our assumptions on f (and thus an algorithm's fault tolerance) will only be visible implicitly, through such communication predicates.

III. Consensus Properties

A system *solves* the consensus problem if it guarantees:

Uniform agreement No two processes ever decide on two different values.

Termination Every process eventually decides on a value.

Non-triviality Any value decided upon has been proposed by some process.

Stability Once a process has made a decision, it never reverts to an undecided state.

Since non-triviality and stability are usually straightforward, we do not discuss them further in this paper. The difficult part is achieving both agreement and termination. Typically, the termination requirement is limited only to non-failed processes. As there is no notion of process failure in the HO model, this is not necessary for our models.

Ideally, we would like to show that our abstract models already guarantee both termination and agreement, and conclude

from the refinement proof that the implementations inherit these guarantees. As discussed in Section II-B, this works for unconditional properties like agreement, but not for termination, which usually requires fairness assumptions. While we will consider termination conditions informally for all of our models, we take the easy way out and prove termination individually for each concrete algorithm formulated in the HO model. Fortunately, assuming a suitable communication predicate such as P_{unif}, this is fairly simple.

IV. Voting, Quorums, and Defection

All the consensus algorithms we consider share a few basic ideas. These ideas are captured by our most abstract model, which we call Voting and describe in this section. To motivate these ideas, let us first consider some other, more obvious candidate solutions to the consensus problem, and see what their shortcomings are.

The first candidate is to have all processes mutually exchange their proposals, and pick the result deterministically, for example, by taking the smallest proposal. Unfortunately, in the presence of even a single failure, this scheme can violate agreement. Any failure could cause two processes to end up with different sets of proposals, as the example from Figure 2 shows, and thus pick different values.

Another obvious candidate is to have one distinguished process, the *leader*, collect the proposals, pick one, and announce its decision to the others. Two-phase commit protocols are based on this idea. This guarantees agreement, but the leader is a single point of failure for termination. If it fails, there is no way of proceeding; we do not know if it decided anything, and whether it announced its decision to the other processes. Trying again, with a different leader, could violate agreement.

We thus need to revert to a decentralized approach. All the algorithms we consider achieve this by *voting*, based on simple counting. Each process picks a value to vote for, and announces the vote to all other processes. Processes then count the votes: if a process sees that some value received an absolute majority (more than $\frac{N}{2}$) of the votes, it decides on that value. Clearly, two different values cannot both get a majority of votes, ensuring agreement. We take a slightly more abstract view, which will be useful later, and require a value to receive votes from a *quorum* of processes instead of a majority. A set of processes is a quorum if it is a member of a *quorum system* $QS \subseteq 2^{\Pi}$, where, to ensure agreement, we require:

$$\forall Q, Q' \in QS.\ Q \cap Q' \neq \emptyset. \qquad \text{(Q1)}$$

In contrast to the leader-based approach, voting has no single point of failure. However, while it is possible to terminate and reach a decision even when any non-quorum of processes fails, there are no guarantees. For example, if all the processes vote for different values, no value will receive a quorum of votes. Furthermore, even if a value does receive a quorum of votes, message loss can prevent processes from learning this. To address these problems, we iterate voting in multiple *rounds*, and allow processes to *switch* their votes between the rounds. Switching allows us to eventually form a quorum of votes for the same value within a round. Voting is iterated until such a quorum is formed, and until all the processes become aware of the quorum and decide; to simplify, we assume that it is iterated forever.

A. Formalizing Voting

We now have the basic ingredients of our most abstract model. Its system state is represented by the record:

record `v_state` =
  `next_round` : $\mathbb{N}$
  `votes` : $\mathbb{N} \to (\Pi \rightharpoonup V)$
  `decisions` : $\Pi \rightharpoonup V$

where V is the set of possible proposed values. The fields' names suggest their purpose:

- `next_round` is the next round to be run. It is a natural number, initially 0.
- `votes` is a (curried) function that, given a round number and a process, tells us which vote, if any, the process cast in that round. In other words, `votes` is the system's *voting history*. Initially, no votes are cast.
- `decisions` records the current decision, if any, of the given process. Initially, no decisions are made.

We overload the notation for curried functions in the usual way, writing $g(p, q)$ for $g(p)(q)$. We also treat partial functions $g : A \rightharpoonup B$ as total and write $g(x) = \bot$ if $x \notin dom(g)$, where $\bot$ is a distinguished value that is not in any of the sets we use. In particular, $\bot \notin V$. Moreover, we write $g[S]$ to denote the image of a set under g and we define the range of g by $ran(g) = g[A]$. Note that $\bot \in ran(g)$ unless $dom(g) = A$.

With this, we formalize the voting principle for decisions in a single round, where $r_decisions$ and r_votes are partial functions of type $\Pi \rightharpoonup V$:

$$\texttt{d_guard}(r_decisions, r_votes) \triangleq \forall p. \forall v \in V.$$
$$r_decisions(p) = v \Longrightarrow \exists Q \in QS.\ r_votes[Q] = \{v\}\,.$$

A process can decide on any value v that receives a quorum of votes. We always allow the processes not to decide, even if such a v exists, to anticipate the possibility of message loss in the implementations.

The voting principle ensures agreement within a single round, but the rounds are iterated. This, together with vote switching, gives us some hope of achieving termination, but we must now also ensure agreement across the rounds. The basic property we must establish is that if a value receives a quorum of votes in some round, then no other value ever receives a quorum of votes in any other round. Formally:

$$\forall r, r'.\ \forall v, v' \in V.\ \forall Q, Q' \in QS.$$
$$\texttt{votes}(r)[Q] = \{v\} \wedge \texttt{votes}(r')[Q'] = \{v'\} \Longrightarrow v = v'.$$

This formulation of the property, however, leaves open how to implement it. We thus replace it by a slightly stronger and more operational property: forbidding *defection*. That is, once a quorum for a value is formed, no process from that quorum may ever vote for any other value. To anticipate the unreliability of the distributed setting, we always allow a process not to vote, modeled as a vote for $\bot$. We formalize this as the following predicate, where $v_hist : \mathbb{N} \to (\Pi \rightharpoonup V)$ is a voting history:

$$\texttt{no_defection}(v_hist, r_votes, r) \triangleq$$
$$\forall r' < r.\ \forall v \in V.\ \forall Q \in QS.$$
$$v_hist(r')[Q] = \{v\} \Longrightarrow r_votes[Q] \subseteq \{\bot, v\}\,.$$

We can now clearly see the tension between agreement and termination present in all voting-based consensus algorithms. To achieve termination, processes may need to switch their votes between the rounds; but in doing so, they must not defect, if agreement is to be preserved.

We now have all the ingredients for the sole event of this model: a round of voting. Its parameters are the current round r and the round votes and decisions, both of type $\Pi \rightharpoonup V$.

Event v_round($r, r_votes, r_decisions$):
Guard
r = `next_round`
`no_defection`(`votes`, r_votes, r)
`d_guard`($r_decisions, r_votes$)
Action
`next_round` := $r + 1$
`votes` := `votes`($r := r_votes$)
`decisions` := `decisions` $\triangleright$ $r_decisions$

Here, $g \triangleright h$ denotes the update of the partial function g with the partial function h. The event's guards and actions directly formalize the previous discussion. There must be no defection in the round votes, and the decisions are made by the voting principle. The next state is obtained by increasing the round, and updating the voting history and decisions.

B. Formalizing Agreement

We now formalize the agreement property of consensus. A trace τ over states of type `v_state` satisfies agreement if:

$$\forall i, j \in dom(\tau).\ \forall p, q.\ \forall v, w \in V.$$
$$\tau(i).\texttt{decisions}(p) = v \wedge \tau(j).\texttt{decisions}(q) = w \Longrightarrow$$
$$v = w.$$

Here s.`decisions` denotes the value of the field `decisions` in the state s.

Proving agreement for the Voting model is straightforward. The `d_guard` predicate, combined with the quorum property (Q1), ensures that agreement is preserved within a round. The `no_defection` guard ensures it across the different rounds.

The agreement property will be inherited by all the subsequent models, since we will prove that they refine Voting. As discussed in Section III, this is not the case for termination, and we impose no termination conditions on the Voting model. We do, however, informally discuss termination next.

C. Towards an Implementation

The Voting model uses a global view of the system. To make the model implementable, we must reconstruct this view using a combination of strictly local process actions and communication. More concretely, the processes need to exchange their votes and voting histories using messages. However, due to failures, the view that a process can reconstruct this way might be only partial, reflected in the filtering by HO sets (Figure 2). As an intermediate informal step towards an implementation, we now consider what happens if we try to perform the steps globally, but based on a partial view.

Consider the scenario shown in Figure 3 where, after one round of voting, the votes of processes p_1–p_4 are visible to us,

Process	p_1	p_2	p_3	p_4	p_5
Vote	0	0	1	1	?

Figure 3. A possible partial view of histories after 1 round of voting

but the vote of process p_5 is not. The example demonstrates a *vote split*, with p_1–p_2 voting 0, and p_3–p_4 voting 1. Define quorums to be simple majorities. As neither 0 nor 1 receives a quorum of visible votes, we cannot make a decision based on these votes. We could make one in the next round by changing some of the votes. However, we cannot distinguish between the following three possibilities:

1) The process p_5 voted 0, forming a quorum of three votes for 0. Although this quorum is not visible to us, it does exist, and we must not change the votes of the processes voting for 0 if we are to preserve the no defection property. Hence we should change the votes of the processes voting for 1 (to 0).
2) This case is the same as the last, but swapping 0 and 1.
3) The process p_5 did not vote at all, or it voted for some value other than 0 or 1; we may freely change the votes of the other processes.

The partial information we receive is thus ambiguous and prevents us from achieving termination while preserving safety. We have two options: either (1) remove the ambiguity so that we can change the votes of some of the processes or (2) prevent the situation from occurring in the first place. These options represent the next design choice (after using voting), and correspond to the two branches from the root in Figure 1.

V. Fast Consensus: Enlarging Quorums

The failed example from Figure 3 is suggestive. Both the votes for 0 and the votes for 1 could be extended to a quorum of three votes by including the vote of the fifth process. But there is an easy way to prevent the confusion: require four votes to reach a decision instead of three. In other words, we change the definition of a quorum to mean all sets of size four or larger, instead of simple majorities. It is not hard to see that, for any split of the four visible votes, this enables us to determine at least one vote that we can safely change.

This solution works for the concrete problem above, but how do we generalize it? First, note that this solution does not work when our partial view includes only three or fewer processes. Thus, we assume a lower bound on the number of visible processes. More precisely, we assume a set of *guaranteed visible sets* of processes. The assumption is that eventually at least one such set will be visible. In the implementations, the guaranteed visible sets will be realized by guaranteed HO sets, where the guarantee comes from the communication predicates (such as $\exists r.\ P_{maj}(r)$) and relies on network and failure assumptions (Section II-D). Once such a set is visible, we should be able to change the votes and make progress.

What was blocking this progress in Figure 3 was a partitioning of the visible set $S = \{p_1, \dots p_4\}$ into two sets of processes $S_0 = \{p_1, p_2\}$ and $S_1 = \{p_3, p_4\}$, respectively voting for 0 and 1, such that:

$$S_0 \cup \overline{S} \in QS \wedge S_1 \cup \overline{S} \in QS,$$

where $\overline{S}$ is S's complement and QS is the quorum system. Set theory gives us:

$$\begin{aligned} & S_0 \cup \overline{S} \in QS \wedge S_1 \cup \overline{S} \in QS \\ \Longrightarrow\ & \exists Q_0, Q_1 \in QS.\ Q_0 \cap Q_1 \subseteq \overline{S} \\ \Longleftrightarrow\ & \exists Q_0, Q_1 \in QS.\ Q_0 \cap Q_1 \cap S = \emptyset. \end{aligned}$$

This leads to an obvious strengthening of (Q1), requiring that for all quorums Q and Q', and all guaranteed visible sets S:

$$Q \cap Q' \cap S \neq \emptyset. \tag{Q2}$$

This ensures that, in case of a vote split, only one subset of votes from a guaranteed visible set can be extended to a quorum. Thus, we can switch all the other ones. To ensure that we can also decide based on any guaranteed visible set, we also stipulate that for every such set S, there exists a quorum Q such that:

$$Q \subseteq S. \tag{Q3}$$

Interestingly, conditions (Q2) and (Q3) define a dissemination quorum if we interpret $\overline{S}$ as a set of Byzantine processes [27]. As these conditions strengthen (Q1), the Voting model will still guarantee agreement under them. They also allow us to achieve termination, but the termination guarantees might not be inherited by refinement, and hence we do not formalize this discussion in an abstract model. We will explain how it is reflected in the concrete algorithms later. Moreover, all these algorithms employ an optimization to the Voting model that avoids exchanging the entire voting histories. We formalize and describe this optimization next.

A. Optimizing Voting

This optimization is based on two observations. First, a process can clearly never defect by repeating its last non-$\perp$ vote. Second, when changing its vote, it is enough to check for defection against the last non-$\perp$ votes of the other processes, rather than checking against their entire voting histories. That is, if a process is not defecting with respect to the last votes of the other processes, it will not defect with respect to the votes in any previous round. To see why, assume that in round r a quorum Q of processes all voted for a value v. By the no defection property, in rounds between r and the current round, no process p in Q can change its vote to one different from v. Therefore, p's last non-$\perp$ vote must remain v. So if Q was a quorum of processes voting for v in round r, after this round, all members of Q will always retain v as their last vote.

The state of the system is thus changed to record just the last non-$\perp$ vote of each process instead of the entire voting history. We still use $\perp$ to denote that a process never voted.

record `opt_v_state =`
  `next_round` : $\mathbb{N}$
  `last_vote` : $\Pi \rightharpoonup V$
  `decisions` : $\Pi \rightharpoonup V$

The guard for checking defection now looks at just the last votes $lvs : \Pi \rightharpoonup V$ instead of the entire voting history:

$$\texttt{opt_no_defection}(lvs, r_votes) \triangleq \forall v \in V.\ \forall Q \in QS.\ lvs[Q] = \{v\} \Longrightarrow r_votes[Q] \subseteq \{\perp, v\}\,.$$

The voting round uses this modified defection guard and replaces the update of `votes`(r) with the update of `last_vote` to `last_vote` $\triangleright\ r_votes$, but otherwise remains the same.

1: **Initially:** `last_vote`$_p$ is p's proposed value
2: `decision`$_p$ is $\perp$
3: send$_p^r$:
4: send `last_vote`$_p$ to all
5:
6: next$_p^r$:
7: **if** received some vote $w > \frac{2N}{3}$ times **then**
8: `decision`$_p$:= w
9: **if** $\left|HO_p^r\right| > \frac{2N}{3}$ **then**
10: `last_vote`$_p$:= smallest most often received vote

Figure 4. The HO model of OneThirdRule

B. Implementations: Fast Consensus

The optimized model abstracts several variants of consensus algorithms found in the literature. We prove that it is refined by OneThirdRule from [12] and its generalization $A_{T,E}$ from [4] (assuming no Byzantine processes). Moreover, it also describes the algorithms used in the first round of the protocol from [7] and in the fast rounds of Fast Paxos [24]. As an example, Figure 4 shows the HO model of the OneThirdRule algorithm, presented in pseudocode for simplicity.

In OneThirdRule, quorums are sets of more than $\frac{2N}{3}$ processes. Hence, the decision rule in lines 7–8 ensures the `d_guard` from the (optimized) Voting model, where the refinement relation relates the state variables of each process p to the p-values of the fields with same names in the abstract model. Given the new quorum size, defining guaranteed visible sets to also be of size greater than $\frac{2N}{3}$ ensures conditions (Q2) and (Q3). As OneThirdRule is a concrete, fully distributed algorithm, we replace guaranteed visible sets by guaranteed HO sets, reflected in the communication predicate required for termination:

$$\exists r.\, P_{unif}(r) \wedge \exists r' > r.\, \forall r'' \in \{r, r'\}.\, \forall p.\ \left|HO_p^{r''}\right| > \frac{2N}{3}$$

The most interesting part of the algorithm is lines 9–10, which guarantees no defection and at the same time directs the votes such that they eventually converge to a common value. By (Q2) and the HO set condition in line 9, we know that only one received value could have been voted for by a quorum; the greater than $\frac{2N}{3}$ requirement on quorums and HO sets ensures that it is the one that received the most votes. If there is a tie in the number of votes, no value could have received a quorum of votes, and processes may switch their votes freely. Either way, no process will defect by choosing a value that received the most votes. Choosing the smallest such value provides the required vote convergence. The communication predicate ensures the existence of a round in which all processes adopt the same vote, and of a later round in which the processes receive enough votes to decide and terminate.

In OneThirdRule, a round of voting requires one round of communication. This also applies to the other algorithms of this type, earning them the name Fast Consensus. If all the processes start with the same value v, the algorithm can terminate within a single failure-free round. Otherwise, the algorithm still terminates within two rounds that satisfy the above communication predicate. The speed comes at a price though. From the communication predicate, we see that

OneThirdRule requires the HO sets to contain more than $\frac{2N}{3}$ processes. Hence $f < \frac{N}{3}$, where f is, as before, the number of tolerated failures. It is not difficult to see that this is optimal, given conditions (Q2) and (Q3). It is, however, possible to implement the Voting model without the additional requirements on quorum sets; this will only require $f < \frac{N}{2}$. We will show how to do this in the next section. The price paid is that the algorithms become more complicated and require multiple communication steps to perform one round of voting.

VI. Same Vote

Fast Consensus resolved the situation from Figure 3 by disambiguating the vote split. In this section, we take the other approach, corresponding to the other branch from Figure 1: we prevent the split from ever happening, thus eliminating the problematic example completely. For this, all the votes cast within a round must be the same. We allow the possibility that some processes do not cast a vote. Formally, we will replace the Voting round event $\texttt{v_round}(r, r_votes, r_decisions)$ by a Same Vote round $\texttt{sv_round}(r, S, v, r_decisions)$, where the processes in S vote for a value $v \in V$, and the others vote $\bot$.

This requires *vote agreement*: all processes must agree on the value of v. But this seems like a paradoxical, circular way to solve consensus: having all processes agree on a single value is exactly what consensus is about! There is, however, a subtle difference between vote agreement and consensus. Because we allow processes to vote $\bot$, unlike for consensus, it is not necessary that every process gets an (non-$\bot$) output from vote agreement. Thus, vote agreement does not share the termination requirement of consensus. However, to make progress, we cannot drop this requirement completely, but we instead relax it: we require that enough processes get an output in some round. The relaxed termination requirement is now collective. Each voting round contains one instance of vote agreement; it is not necessary that all instances terminate, but at least one must do so. Moreover, the outcomes of the different vote agreement instances are independent; we do not require that they match.

The consequence of the laxer termination requirements is that some of the ideas that we described at the start of Section IV, and which failed to solve consensus because of their weak termination properties, can now be recycled to successfully solve the vote agreement problem. The two ideas are non-iterated voting, which we will henceforth refer to as *simple* voting, and the leader-based approach. Before we put either of them to use in the implementations, we must ensure (and this is the tricky part of the algorithms) that any agreed upon vote preserves the no defection property of the Voting model.

A. Formalizing Same Vote

The system state remains the same as in the Voting model. As before, we require that there is no defection in the votes. Since each process will now vote for either v or $\bot$, voting for v must not cause any process to defect; we say that v must be *safe*. If there previously existed a quorum for a value w, we must have $v = w$. Otherwise, the processes that previously voted for w could defect by voting for v. Formally:

$$\texttt{safe}(v_hist, r, v) \triangleq \forall r' < r. \forall w \in V. \forall Q \in QS. \; v_hist(r')[Q] = \{w\} \Longrightarrow v = w.$$

Round \ Process	p_1	p_2	p_3	p_4	p_5
0	0	0	$\bot$	?	?
1	$\bot$	$\bot$	1	?	?
2	$\bot$	$\bot$	$\bot$	?	?

Figure 5. Same Voting: a possible partial view of histories after three voting rounds.

In a Same Vote round r, the processes in some set S receive an output v from vote agreement and vote for v, while the others vote for $\bot$. If $S = \emptyset$ then v is unused and unconstrained, otherwise it must be safe. Formally:

Event $\text{sv_round}(r, S, v, r_decisions)$:
Guard
$r = \texttt{next_round}$
$S \neq \emptyset \Longrightarrow \texttt{safe}(\texttt{votes}, r, v)$
$\texttt{d_guard}(r_decisions, [S \mapsto v])$
Action
$\texttt{next_round} := r + 1$
$\texttt{votes} := \texttt{votes}(r := [S \mapsto v])$
$\texttt{decisions} := \texttt{decisions} \triangleright r_decisions$

Here $[S \mapsto v]$ maps all processes from S to v, and the others to $\bot$. The refinement relation between Voting and Same Vote is just the identity. The refinement proof hinges on the fact that $\texttt{safe}$ implies $\texttt{no_defection}$ with $r_votes = [S \mapsto v]$.

B. Towards an Implementation

As a step towards an implementation of the Same Vote model in a distributed setting, we again look at some possible scenarios with only partial information. As we wish to improve the fault tolerance to $f < \frac{N}{2}$ process failures (over $\frac{N}{3}$ for Fast Consensus), we restrict our view to just over $\frac{N}{2}$ processes.

It was the combination of partial information and vote splits that prevented us from changing the votes without causing defection in the example of Figure 3. That particular situation is now eliminated, as vote agreement ensures that such vote splits within a single round can no longer occur. However, vote agreement prevents neither vote splits across multiple rounds, nor hiding of quorums by a partial view. Consider the example in Figure 5: it is not obvious which values are safe for round 3. A priori, it may be that 0 received a quorum of votes in round 0 (if process p_4 or p_5 voted for 0), or that 1 received a quorum in round 1 (if p_4 and p_5 both voted for 1), resembling the ambiguity present in the Voting model and Figure 3. However, the situation can be resolved, and the next two sections describe two ways to do so. They correspond to the two branches from the Same Vote model in Figure 1.

VII. Observing Quorums

Figure 5 demonstrates the difficulty of detecting vote quorums and finding safe values based on a partial view of the voting history. The main idea behind the solution introduced in this section is that each process maintains a vote *candidate* value $v \in V$ that is safe to vote for by construction. Maintaining the candidates' safety requires each process to detect when a quorum of votes is formed for some value.

For this, each process must observe the votes of the other processes. We now describe this scheme in more detail.

Initially, all values are safe. Thus, processes can initialize their candidates to arbitrary values; in particular, they can use their proposed values. Furthermore, all values will remain safe until the first time a quorum is formed for some value, at which point all processes must update their candidates to this value. To ensure that this happens, we require each process p to try to update its candidate in every round, based on the votes it observes in the round. More precisely, consider an arbitrary process p and a round r. Due to the Same Vote principle, there is some value $v \in V$ such that each vote cast in r is either for v or for $\perp$. We say that process p's *observation* in round r is v if p receives a vote for v from at least one process in r and $\perp$ if it receives only votes for $\perp$. If p observes v (i.e., not $\perp$) then it updates its candidate to v.

Assume now that r is in fact the first round in which a quorum of votes (for v) is formed. If p observes v, it will update its candidate to v, and safety will be guaranteed. However, if p observes $\perp$, it fails to update the candidate, which may violate safety. To avoid this possibility, we require that p *waits* to receive votes from some quorum Q of processes before it makes its observation and moves on to the next round. By (Q1), Q intersects with the set of processes voting for v, which ensures that p will receive at least one vote for v, and thus update its candidate to v. Thus, after round r, the candidates of all processes will become v. Since we assume that the votes are always selected from the set of candidates, v is the only value that can be voted for, and hence observed after this point. Further updates based on observations will thus not change the candidates and therefore preserve safety.

As an example, interpret Figure 5 as if it were showing the observations that the processes make in each round, instead of the votes they cast. The candidates after round 2 are:

$$[p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 1, p_4 \mapsto ?, p_5 \mapsto ?],$$

that is, processes p_1 and p_2's candidate is 0 and p_3's candidate is 1, while p_4 and p_5's candidates are unknown. We immediately see that both 0 and 1 are safe for round 3, as they are among the candidates. Moreover, we can even conclude that *all* values are safe. Otherwise, the set of candidates would be a singleton, containing only the unique value that has received a vote quorum.

A. Formalizing Observing Quorums

First, we extend the state record `v_state` with following field to record the processes' candidates:

$$\texttt{cand} : \Pi \rightarrow V.$$

The safety of a new vote v is now determined based on the candidates. With $cs : \Pi \rightarrow V$ we define this as follows:

$$\texttt{cand_safe}(cs, v) \triangleq v \in ran(cs).$$

We represent the observations made in each round by a partial function $obs : \Pi \rightharpoonup V$. According to the discussion above, obs is of the form $[OS \mapsto v]$, where v is the round vote, and OS is the set of processes observing v. If v receives a quorum of votes, we require $OS = \Pi$. We can however generalize this by allowing processes to observe not only votes but also each other's candidate values, i.e., we only require $ran(obs) \subseteq ran(\texttt{cand})$. From the previous discussion we know that all old candidate values remain safe if v does not receive a quorum of votes. Otherwise, we still require $obs = [\Pi \mapsto v]$. This adoption of others' candidates will prove useful for termination. We formalize these considerations in the following round event.

Event obsv_round($r, S, v, r_decisions, obs$):
Guard
$r = \texttt{next_round}$
$S \neq \emptyset \Longrightarrow \texttt{cand_safe}(\texttt{cand}, v)$
$ran(obs) \subseteq ran(\texttt{cand})$
$S \in QS \Longrightarrow obs = [\Pi \mapsto v]$
$\texttt{d_guard}(r_decisions, [S \mapsto v])$
Action
$\texttt{next_round} := r + 1$
$\texttt{cand} := \texttt{cand} \triangleright obs$
$\texttt{decisions} := \texttt{decisions} \triangleright r_decisions$

The guard $S \in QS \Longrightarrow obs = [\Pi \mapsto v]$ ensures that quorums of votes are reflected in all processes' observations. Since no guard consults the voting history and only the current round's votes are needed to make decisions, there is, in contrast to the Same Vote model, no need to record votes. We therefore drop the field `votes` from the state.

The refinement relation between the Observing Quorums and Same Vote models relates the fields `votes` in Same Vote and `cand` in Observing Quorums by requiring that

$$\texttt{votes}(r)[Q] = \{v\} \Longrightarrow \texttt{cand} = [\Pi \mapsto v]$$

holds for all values $v \in V$, quorums $Q \in QS$, and rounds r preceding the current one. The common fields `next_round` and `decisions` are related by the identity. Based on this relation, we can prove that $\texttt{cand_safe}(\texttt{cand}, v)$ implies $\texttt{safe}(\texttt{votes}, r, v)$.

B. Implementing Observing Quorums

The previous model captures several algorithms from the literature. We prove that it is refined by Ben-Or's algorithm [3] and UniformVoting [12]. It also captures the generic algorithm from [17], although we do not formally prove this. The model, however, only tells us how to pick safe values in each round. In the implementation, the processes must use a vote agreement scheme to agree on one such value. We have already mentioned two candidate schemes: the leader-based scheme and simple voting. Either can be used here. As an example, we show the UniformVoting algorithm (Figure 6), which uses simple voting.

In the algorithm, a round of voting requires two sub-rounds of communication. Vote agreement takes place in the first sub-round, while casting and observing votes take place in the second sub-round. The inputs to vote agreement are the candidates of each process (line 6); picking any one of them will satisfy the `cand_safe` guard. The output, recorded in the variable $\texttt{agreed_vote}_p$, is generated by simple voting, and corresponds to the parameter v of the abstract model's event `obsv_round`. The voting principle of simple voting is encoded in the combination of the check in line 10 and the assumed communication predicate $\forall r. P_{maj}(r)$, with P_{maj} as defined in (P_{maj}). The same predicate is used in the second sub-round to

```
1:  Initially: cand_p is p's proposed value
2:             other fields are ⊥
3:
4:  Sub-Round r = 2φ:       // vote agreement
5:  send_p^r:
6:     send cand_p to all
7:
8:  next_p^r:
9:     cand_p := smallest value received
10:    if all the values received equal v then
11:        agreed_vote_p := v
12:    else
13:        agreed_vote_p := ⊥
14: Sub-Round r = 2φ + 1:  // casting and observing votes
15: send_p^r:
16:    send (cand_p, agreed_vote_p) to all
17:
18: next_p^r:
19:    if at least one (_, v) with v ≠ ⊥ received then
20:        cand_p := v
21:    else
22:        cand_p := smallest w from (w, ⊥) received
23:    if all received equal (_, v) for v ≠ ⊥ then
24:        decision_p := v
```

Figure 6. The HO model of UniformVoting

ensure the guards $S \in QS \Longrightarrow obs = [\Pi \mapsto v]$ (lines 19–20) and $\texttt{d_guard}$ (lines 23–24).

Processes update their candidates either to the round vote (line 20) or to a candidate of some other process (lines 9 and 22). This corresponds to making a non-$\perp$ observation and satisfies the guard $ran(obs) \subseteq ran(\texttt{cand})$. Moreover, the adoption of other processes' candidates helps the convergence to a common vote candidate, necessary for termination of vote agreement. This termination is guaranteed by the additional communication predicate $\exists r.\, P_{unif}(r)$. In the round satisfying $P_{unif}(r)$, all processes will adopt the same candidate, and thus agree on a vote (line 11) and decide (line 24).

The refinement relation relates, for each p, the value $\texttt{cand}_p$ to the value of $\texttt{cand}(p)$ in the abstract model and the value of $\texttt{decision}_p$ to $\texttt{decisions}(p)$. The refinement proof follows the above remarks.

The algorithms in this section tolerate $f < \frac{N}{2}$ process failures and thus exhibit better fault tolerance than the fast consensus algorithms from Section V-B. Depending on the scheme used for vote agreement, they can also terminate within two fault-free communication rounds, as shown in [17]. However, the use of waiting (e.g., visible in the communication predicate of UniformVoting) requires a more complicated communication layer, since retransmission is necessary, and it hides the fact that additional messages are required for acknowledgments. The same level of fault tolerance can be achieved without waiting, as the algorithms in the next section show.

VIII. Most Recently Used (MRU) Vote

In the algorithms of the previous section, processes maintain a safe vote candidate at every point in time. In this section, we show how to generate such candidates only when they are needed, based on just partial views of voting histories and without resorting to waiting.

Going back to the definition of a Same Voting round, we observe that any state of Same Voting satisfies the invariant:

$$\texttt{votes}(r,p) = v \Longrightarrow \texttt{safe}(\texttt{votes}, r, v),$$

for any $v \in V$. Moreover, such a v is the only value receiving votes in round r. Hence, no other value can receive a quorum of votes in r, and v is also safe in round $r+1$. Formally:

$$\texttt{votes}(r,p) = v \Longrightarrow \texttt{safe}(\texttt{votes}, r+1, v).$$

Returning to the example from Figure 5, by the above we conclude that the value 1 is safe in round 2. Moreover, we see a quorum of $\perp$ votes in round 2. By the intersection property (Q1), no value whatsoever could have received a quorum of votes in that round; value 1 is therefore still safe in round 3, and can be chosen as the next vote. In this way we have in fact generated, on the fly, the same candidate that a hypothetical process running the Observing Quorums scheme would do, if its observations were based on the votes shown in Figure 5.

It is straightforward to generalize this solution. If we see the voting history of a quorum Q after multiple rounds, the *most recently used* (MRU) vote will still be safe. This value is unique, since all values cast within the same round are the same. If nobody in Q ever voted, we define the MRU vote to be $\perp$. In this case, by (Q1) no value ever received a quorum of votes and all values are safe. Formally, given a voting history $\texttt{v_hist}$ and a quorum Q, we define a function $\texttt{the_mru_vote}$ as above, and we say that Q is an MRU guard for v if:

$$\begin{aligned}&\texttt{mru_guard}(v_hist, Q, v) \triangleq \\ &\quad Q \in QS \wedge \texttt{the_mru_vote}(v_hist, Q) \in \{\perp, v\}.\end{aligned}$$

Following the discussion above, we prove that:

$$\texttt{mru_guard}(\texttt{votes}, Q, v) \Longrightarrow \texttt{safe}(\texttt{votes}, \texttt{next_round}, v),$$

for any $Q \in QS$ and $v \in V$. Replacing $\texttt{safe}$ with $\texttt{mru_guard}$ in the event $\texttt{sv_round}$ thus yields a correct refinement of Same Voting. As the MRU scheme works even with just partial information, we are now ready to move to a distributed implementation.

A. Optimizing MRU Vote

Like Fast Consensus, the MRU scheme can also be optimized to avoid transmitting the entire voting histories of all the processes. The histories were only used in the $\texttt{mru_guard}$, to determine the MRU vote of a quorum of processes. This can obviously also be done by just looking at the MRU vote of each individual process in the quorum, together with its associated round number. The optimized state of the system is thus:

```
record opt_v_state =
    next_round : ℕ
    mru_vote : Π ⇀ (ℕ × V)
    decisions : Π ⇀ V
```

The guard is changed in the obvious fashion. It now takes a parameter $mrus : \Pi \rightharpoonup (\mathbb{N} \times V)$:

$$\begin{aligned}&\texttt{opt_mru_guard}(mrus, Q, v) \triangleq \\ &\quad Q \in QS \;\wedge \texttt{opt_mru_vote}(mrus[Q]) \in \{\perp, v\}.\end{aligned}$$

For brevity we skip the definition of `opt_mru_vote`. The voting round is changed as expected, and the refinement proof is straightforward.

Event opt_mru_round($r, S, v, Q, r_decisions$):
Guard
r = `next_round`
$S \neq \emptyset \Longrightarrow$ `opt_mru_guard`(`mru_vote`$, Q, v$)
`d_guard`($r_decisions, [S \mapsto v]$)
Action
`next_round` := $r + 1$
`mru_vote` := `mru_vote` $\triangleright [S \mapsto (r, v)]$
`decisions` := `decisions` $\triangleright\ r_decisions$

This gives us a method of picking candidates. The processes must also agree on exactly one such candidate, and, as in the previous section, we must choose a vote agreement scheme. The Paxos [22] and Chandra-Toueg [10] algorithms opt for a leader-based scheme. We have derived both of these algorithms in our formal development, but we do not discuss them here, and instead present a new algorithm that we devised.

B. New Algorithm

In [12], Charron-Bost and Schiper posed the question whether there exists a leaderless consensus algorithm tolerating $f < \frac{N}{2}$ failures, whose safety does not depend on waiting (and, more generally, without invariant on the HO sets). Leaderless algorithms are useful in settings with no stable leader and can decrease latency [24], while the drawbacks of waiting were already mentioned in Section VII-B.

The classification we propose in this paper provides us with sufficient guidance to find such an algorithm. The requirement to tolerate $f < \frac{N}{2}$ failures disqualifies the Fast Consensus algorithms (which handle only less than $\frac{N}{3}$ failures). We thus turn to the Same Vote mechanism. Since the Observing Quorums model requires waiting in order to provide safety, we are left with the MRU model as the only candidate in the hierarchy. The only remaining question is whether we need a leader to implement vote agreement. But we know already that the answer is no: we can implement it using the simple voting scheme.

The pseudocode of the HO model of the algorithm is shown in Figure 7. One round of voting requires three sub-rounds of communication. The first sub-round equips each process with a safe value. Based on these values, vote agreement by simple voting takes place in the second sub-round, and voting proper takes place in the third one.

In the first sub-round, processes exchange their MRU votes, along with `prop` values, explained shortly. Based on these MRU votes, each process p tries to determine a safe candidate value for the common round vote (variable $\texttt{cand}_p$). If p receives messages from a quorum of other processes, it looks at the output of the `opt_mru_vote` function (lines 12–16). This output is used as the safe candidate (line 14) unless it is $\bot$, in which case any value is safe and the candidate is set to the smallest `prop` value received (line 16). Either way, the resulting $\texttt{cand}_p$ satisfies the `opt_mru_guard`. If p does not receive enough messages, it sets $\texttt{cand}_p$ to $\bot$ (line 18). In the second sub-round, processes use their `cand` values as inputs to vote agreement (line 21). This uses simple voting, with the

1: **Initially:** $\texttt{prop}_p$ is p's proposed value, other fields are $\bot$
2:
3: **Sub-Round** $r = 3\phi$: *// finding safe vote candidates*
4: send_p^r:
5: send ($\texttt{mru_vote}_p$, $\texttt{prop}_p$) to all
6:
7: next_p^r:
8: **if** $HO_p^r \neq \emptyset$ **then**
9: $\texttt{prop}_p$:= smallest w from $(_, w)$ received.
10: **if** $|HO_p^r| > \frac{N}{2}$ **then**
11: **let** $mrus$ = set of all tsv's from $(tsv, _)$ received
12: **let** mru = `opt_mru_vote`($mrus$)
13: **if** $mru \neq \bot$ **then**
14: $\texttt{cand}_p$:= mru
15: **else**
16: $\texttt{cand}_p$:= $\texttt{prop}_p$
17: **else**
18: $\texttt{cand}_p$:= $\bot$
19: **Sub-Round** $r = 3\phi + 1$: *// vote agreement*
20: send_p^r:
21: send $\texttt{cand}_p$ to all
22:
23: next_p^r:
24: **if** received some $v \neq \bot$ more than $\frac{N}{2}$ times **then**
25: $\texttt{mru_vote}_p$:= (ϕ, v)
26: $\texttt{agreed_vote}_p$:= v
27: **else**
28: $\texttt{agreed_vote}_p$:= $\bot$
29: **Sub-Round** $r = 3\phi + 2$: *// voting proper*
30: send_p^r:
31: send $\texttt{agreed_vote}_p$ to all
32:
33: next_p^r:
34: **if** received some $v \neq \bot$ more than $\frac{N}{2}$ times **then**
35: $\texttt{decision}_p$:= v

Figure 7. The HO model of the New Algorithm

voting principle reflected in the check in line 24, and requires no leader. The output of vote agreement corresponds to the parameter v of the abstract `opt_mru_round` event. If a process accepts v as its round vote, it updates its MRU vote, reflecting the update of the `mru_vote` field in the abstract event. The last sub-round then deals with ordinary voting, with the decision rule in lines 34–35 implementing the `d_guard`.

The refinement relation equates the $\texttt{decision}_\texttt{p}$ and $\texttt{mru_vote}_\texttt{p}$ variables of each process to `decision(p)` and `mru_vote(p)` in the abstract model, and the proof follows the discussion above. The only part of the algorithm that we must still explain is termination, which relies on the communication predicate:

$$\exists \phi.\ P_{unif}(3\phi) \wedge \forall i \in \{0, 1, 2\}\ .\ P_{maj}(3\phi + i).$$

As in UniformVoting, the processes try to help convergence to a common vote candidate by taking the smallest `prop` value seen so far. The above predicate then ensures that all processes hold the same (non-$\bot$) `cand` after sub-round 3ϕ. In sub-rounds $3\phi + 1$ and $3\phi + 2$, the processes then adopt this `cand` first as the round vote, and then also as the decision.

IX. CONCLUSION

We have presented a unified description of a number of consensus algorithms found in the literature: OneThirdRule, $A_{T,E}$, Paxos, Chandra-Toueg, Ben-Or, UniformVoting, and the generic algorithm of [17]. By using refinement, we could (1) describe the main algorithmic ideas behind them in simple terms and (2) create a taxonomy of the algorithms based on these ideas. We hope that we have also shed light on why the algorithms are constructed the way they are. Finally, the insights gained from the taxonomy helped us develop a new algorithm, which is leaderless, tolerates $f < \frac{N}{2}$ failures, and does not employ waiting to guarantee safety. This answers a question posed in [12] asking whether such an algorithm exists.

The consensus problem has been thoroughly studied. Nevertheless, we believe that our work provides both a useful synthesis of existing knowledge about the algorithms we cover and a novel way of understanding and relating them. In particular, the Voting model and the no defection property provide a simple basis for describing the different algorithms, a basis we have not seen in the literature before.

Our work, of course, has its own limitations. The assumption of communication-closedness puts some practical algorithms, such as Disk Paxos, outside of our scope. Furthermore, algorithms such as Fast Paxos [24] essentially combine several algorithms and as such they do not cleanly fit into our hierarchy. Finally, our abstract models capture only the safety guarantees of the target algorithms. It is unclear whether and how this could be extended to termination in a simple fashion.

As future work, we plan to extend our development to cover Byzantine failures. We are confident that this is possible, as refinement has been used to adapt consensus algorithms to the Byzantine setting before [25], [26]. We would also like to extend the scope of our work to tasks that build upon consensus, such as atomic broadcast. Stepwise refinement has been used in this context already [36], but it may be possible to soundly apply some form of lockstep abstraction to these tasks as well. Finally, while the lockstep abstraction is extremely useful for reasoning, the resulting models are somewhat farther away from implementations than the more standard asynchronous models with failure detectors, despite the result in [11]. It would be interesting to see how to best bridge this gap and develop formally verified implementations of protocols specified in the HO and similar models.

REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[3] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *PODC*, pages 27–30, 1983.

[4] M. Biely, J. Widder, B. Charron-Bost, A. Gaillard, M. Hutle, and A. Schiper. Tolerating corrupted communication. In *PODC*, pages 244–253, 2007.

[5] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.

[6] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Reconstructing Paxos. *SIGACT News*, 34(2):42–57, 2003.

[7] F. Brasileiro, F. Greve, A. Mostefaoui, and M. Raynal. Consensus in one communication step. In *PaCT*, pages 42–50, 2001.

[8] J. W. Bryans. Developing a consensus algorithm using stepwise refinement. In *ICFEM*, pages 553–568, 2011.

[9] C. Cachin. Yet another visit to Paxos. Technical Report RZ 3754, IBM Research, 2009. Revised April 7, 2011.

[10] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[11] M. Chaouch-Saad, B. Charron-Bost, and S. Merz. A reduction theorem for the verification of round-based distributed algorithms. In *Reachability Problems*, pages 93–106. 2009.

[12] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[13] H. Debrat and S. Merz. Verifying fault-tolerant distributed algorithms in the heard-of model. *Archive of Formal Proofs*, 2012.

[14] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[16] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.

[17] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Trans. Computers*, 53(4):453–466, 2004.

[18] R. Guerraoui and M. Raynal. The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67, 2007.

[19] M. Hurfin, A. Mostéfaoui, and M. Raynal. A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors. *IEEE Trans. Computers*, 51(4):395–408, 2002.

[20] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *DSN*, pages 92–101, 2007.

[21] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, pages 245–256, 2011.

[22] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[23] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[24] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[25] L. Lamport. Byzantizing Paxos by refinement. In *Distributed Computing*, pages 211–224. Springer, 2011.

[26] B. Lampson. The ABCD's of Paxos. In *PODC*, 2001.

[27] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[28] O. Marić and C. Sprenger. Consensus refined. *Archive of Formal Proofs*, 2015. http://afp.sf.net/entries/Consensus_Refined.shtml.

[29] A. Mostéfaoui, S. Rajsbaum, and M. Raynal. A versatile and modular consensus protocol. In *DSN*, pages 364–373, 2002.

[30] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *DISC*, pages 49–63, 1999.

[31] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002.

[32] R. D. Prisco, B. W. Lampson, and N. A. Lynch. Revisiting the Paxos algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

[33] M. Raynal and J. Stainer. Synchrony weakened by message adversaries vs asynchrony restricted by failure detectors. In *PODC*, pages 166–175, 2013.

[34] O. Rütti, Z. Milosevic, and A. Schiper. Generic construction of consensus algorithms for benign and byzantine faults. In *DSN*, pages 343–352, 2010.

[35] Y. J. Song, R. van Renesse, F. B. Schneider, and D. Dolev. The building blocks of consensus. In *ICDCN*, pages 54–72, 2008.

[36] R. van Renesse, N. Schiper, and F. B. Schneider. Vive la différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2014.