Object Oriented Consensus

Yehuda Afek
Department of Computer Science, Tel-Aviv University, Israel
afek@tau.ac.il

James Aspnes
Department of Computer Science, Yale University, USA
james.aspnes@gmail.com

Edo Cohen
Department of Computer Science, Tel-Aviv University, Israel
edocohen@tau.ac.il

Danny Vainstein
Department of Computer Science, Tel-Aviv University, Israel
dannyvainstein@gmail.com

Abstract

Our work focuses on the problem of decomposing consensus algorithms into a common framework composed of simple building blocks. We show that earlier decomposition strategies fall short when applied to some well known algorithms and present a new framework in order to tackle the problem. First we use Aspnes' framework [2] composed of *adopt-commit* [5] and *conciliator* [2] objects in order to decompose the well known Phase-King Byzantine algorithm [4]. We then consider two other well-known algorithms and argue that this framework is insufficient in these (and other) cases and offer a new framework. The framework works in rounds where each consists of two steps. The first step involves an object which detects agreement and the second involves an object that aims at achieving consensus. We denote our newly defined objects as ***vacillate-adopt-commit*** and ***reconciliator***. We demonstrate our decomposition on two well known algorithms. Namely, Ben-Or's Randomized algorithm [3] and the Raft algorithm [6].

2012 ACM Subject Classification Theory of computation → Models of computation → Concurrency → Distributed computing models

Keywords and phrases distributed algorithms, wait-free, consensus, message-passing

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The consensus problem introduced by Lamport, Pease and Shostak [7] resides at the heart of many distributed algorithms such as leader election, database transaction handling, resource allocation, ensuring storage replicas are mutually consistent, block chain technology and many more.

In the consensus protocol between n processors, each with an input value, processors agree on a single common output which was the input to one of them. While consensus is trivial in a non-faulty synchronous environment, it is often more difficult in practice as most distributed networks are asynchronous and must be resilient to faults and various miss behaviors.

The philosophy of software engineering asserts that decomposing complex systems into simple building blocks is a good thing. One reason for this being that by analyzing simpler objects we may deduce observations on more complex systems.

One attempt to decompose consensus was provided by Gafni [5]. Gafni proposed *adopt-commit*, an object fulfilling weaker guarantees than consensus, as a building block of consensus. Aspnes further provided a detailed decomposition [2] of consensus into *adopt-commit* and a complementary *conciliator* object which together form a generic framework describing consensus. In this work we extend these decompositions, into *vacillate-adopt-commit* and *reconciliator* objects, which describe the core of many existing consensus algorithms in which it is not clear how to breakdown into the previous frameworks. Our main thesis in the paper is that many known consensus algorithms fall into a similar pattern of a repetitive two-fold process in which the first step evaluates the current status and how close it is to a consensus and a second step which brings closer to a consensus by taking some action. Together with previous work, we provide a more solid interpretation of the mechanisms composing well known consensus algorithms.

Our paper is structured as follows. In Section 3, the consensus template is presented. In section 4 we first tackle the decomposition of the well known phase-king byzantine algorithm [4] and show how it naturally decomposes Aspnes' framework of *adopt-commit* and *conciliator* objects. Thereafter we tackle Ben-Or's randomized algorithm [3] and the raft algorithm [6]. We observe that Aspnes' former framework fails to capture these algorithms naturally and provide our alternate decomposition using *vacillate-adopt-commit* and *reconciliator* objects.

In section 5 we explore the relation between *vacillate-adopt-commit* and *adopt-commit*, showing the latter is slightly weaker. We conclude with final remarks in Section 6.

2 Preliminaries

We consider a network consisting of n processors, $\{p_1, \ldots, p_n\}$, each processor p begins with an input value $p.init$. The goal of ***consensus*** protocol is to agree on a value, u, for all processors while satisfying the following three conditions:

- ***Validity*** - The value u must be the initial value of one of the processors involved in the protocol, i.e., $\exists i,\ s.t.\ p_i.init == u$.
- ***Termination*** - Each processor decides after taking a finite number of steps.
- ***Agreement*** - The value agreed upon, u, is the same for all processors.

In the context of distributed computation it is useful to refer to an implementation of a protocol as an ***object***. We follow these notations in this paper, that is, a processor which is part of a network attempting to achieve consensus invokes an object, $Consensus(v)$, with its initial value $v \leftarrow p.init$ and expects to receive a value u such that the properties above are fulfilled.

An ***adopt-commit*** object may be viewed as a weaker version of consensus. In adopt-commit the returned bit is accompanied by a confidence level. The *agreement* property is waived and two more refined properties are defined instead. Formally, *adopt-commit* receives as an argument a value v and returns a pair $(confidence, u)$ where $confidence \in \{adopt,\ commit\}$ fulfilling the following properties:

- ***Validity, Termination*** - Same as above.
- ***Coherence*** - If some processor receives *(commit,u)* then all other processors receive the same value u (with confidence either *adopt* or *commit*).
- ***Convergence*** - If all processors invoke $adopt - commit(v)$ with the same value v, then all processors receive $(commit, v)$.

The newly introduced object ***vacillate-adopt-commit*** adds an additional confidence level with respect to *adopt-commit*. The object expects a value v from every processor and

returns a value u and one of three confidence levels, ***vacillate***, ***adopt*** or ***commit***. The following guarantees are required of the object:

- ***Validity, Termination, Convergence*** - Same as above.
- ***Coherence over adopt & commit*** - If any processor received *(commit, u)*, then every other processor receives either *(commit, u)* or *(adopt, u)* (this guarantee corresponds to the adopt-commit's coherence).
- ***Coherence over vacillate & adopt*** - If no processor received *commit* and some processor received $(adopt, u)$, then every other processor receives either $(adopt, u)$ or $(vacillate, *)$, where * may be any value (subject to other constraints like validity).

Our main thesis in this paper is that consensus algorithms are essentially a sequence of repetitive rounds of invoking an object which revises the current status of the processors and how close they are to an agreement followed by an action which brings the processors closer to an agreement. The objects *adopt-commit* and *vacillate-adopt-commit* are building blocks suitable to fulfill the role of the agreement detector. In order to provide all the building blocks necessary to compose consensus we must define objects which perform some action that possibly brings closer towards a consensus. In his framework [2], Aspnes defines such an object, ***Conciliator***, which is required to satisfy the following properties:

- ***Validity, Termination*** - Same as above.
- ***Probabilistic Agreement*** - With probability greater than 0, the returned value is the same to all processors.

Another object we define is the ***Reconciliator***. The *reconciliator* is required to satisfy the following properties:

- ***Termination*** - Same as above.
- ***Weak Agreement*** - With probability 1, at some point all invoking processors receive the same value such that this value corresponds to the *adopt* values of the current round.[1]

Despite the name similarities, these objects differ profoundly, the conciliator is essentially a probabilistic consensus while the reconciliator is weaker in the sense that it may be invoked by a subset of the network.

Throughout the paper, we abbreviate the objects *vacillate-adopt-commit* as ***VAC*** and *adopt-commit* as ***AC***. Furthermore, the returned values are denoted by the first letter of the confidence levels, i.e. ***V,A,C***.

3 The Generic Form of Consensus

The idea behind this paper is that many well known consensus algorithms have the same basic structure consisting of two objects, *VAC* that checks whether consensus has been reached or not and *reconciliator* that shakes up the preferences of the processors in case of a stalemate. We do not claim that this structure is necessarily better than using *AC* and *conciliators*, but that it more accurately reflects the existing structure of algorithms in the field.

Informally, the generic consensus algorithms work in rounds. In each round, first the *VAC* is invoked to observe the system state and inform us as to whether consensus has been reached. *VAC* returns to each processor one of three possible outputs: (1) ***(commit, v)***

[1] if no such values were given, the returned value should be one of the inputted values

which indicates that the system has reached an agreement on value v, (2) ***(adopt, v)*** which indicates that it is possible that some processors in the system have agreed on the value v, and (3) ***(vacillate, v)*** indicating that the system is in an indecisive state.

If a processor receives ***(commit, v)***, it is guaranteed that no other processor receives a *vacillate* value and all outputs return with the same value, v. A processor receiving ***(adopt, v)*** is guaranteed that any other processor either received a *vacillate* value or received the same preference that the earlier processor has. Finally, if a processor receives ***(vacillate, v)***, the only guarantee it has is that no other processor received a *commit* value.

We note the key difference between the *VAC* and *AC* objects. An adopt-commit object always returns a new value to be adopted by a process, but this is not consistent with the structure of many consensus protocols in the literature. Adding a third option, i.e., *vacillate*, accounts for situations where the algorithm does not force a process to update its preference. Furthermore *vacillate* gives the receiving processor the information that a consensus has not been reached. This type of information is not available using the *AC* infrastructure, but as we will see is available in existing algorithms in the literature.

The question is how termination of the consensus can be guaranteed if the collection of preferences is balanced and the VAC continually returns vacillate. For that purpose, the *reconciliator* is used to give each vacillating processor a new preference with a guarantee to provide a deciding set of preferences with some probability. That is, whenever a processor receives a *vacillate* value from the *VAC* object the distributed *reconciliator* provides each processor with an alternate preference which guarantees that eventually enough processors will get the same preference leading to *VAC* eventually observing agreement. Pseudocode for the consensus template is given in Algorithm 1.

```
Consensus(v)
    m ← 0
    INIT()
    while true do
        m ← m + 1
        (X, σ) ← VAC(v, m)
        switch X do
            case vacillate:
            v ← Reconciliator(X, σ, m)
            case adopt: v ← σ
            case commit: v ← σ and
            decide σ
        endsw
    end
```

Algorithm 1: Consensus Template

```
Consensus(v)
    m ← 0
    INIT()
    while true do
        m ← m + 1
        (X, σ) ← AC(v, m)
        switch X do
            case adopt:
            v ← Conciliator(X, σ, m)
            case commit: v ← σ and
            decide σ
        endsw
    end
```

Algorithm 2: Consensus Template using AC and Conciliator

Note that *INIT* is a void function unless stated otherwise. Furthermore, note that the operation, *decide* σ, is followed by a halt operation, that is, the processor will decide upon its value and return. The argument m is the phase of the consensus process.

Next we prove that the template indeed achieves consensus, using the VAC and *reconciliator* properties.

▶ **Lemma 1.** *Algorithm 1 is a correct consensus algorithm.*

Proof.
- **Agreement**: If some processor decided on a value v we are guaranteed that it received $(commit, v)$ during that round. Thus by the VAC's coherence over adopt commit guarantee we are ensured that all processors complete the round with the same value. Thus by VAC's convergence all remaining processors will decide on the same value in the next round.
- **Validity**: Follows from the reconcilliator's guarantees and the fact that all inputs to the VAC and reconcilliator objects are clearly valid inputs.
- **Termination**: Follows from the reconcilliator's guarantees.

◀

In addition to algorithm 1, we provide here a concrete algorithm for consensus using the AC and conciliator objects. The algorithm correctness is proved similarly to 3.

4 Consensus Decomposition

We first demonstrate how the phase-king algorithm naturally decomposes using Aspnes' framework [2] (i.e. adopt-commit rather than vacillate-adopt-commit and concilliator rather than reconcilliator). Following that we decompose ben-or's algorithm and the raft algorithm using our framework under the statement that Aspnes' earlier framework is not sufficient.

4.1 Phase-King Algorithm

Here we show how the Phase-King consensus protocol of Berman, Garay and Perry [4] fits the framework given in [2]. Throughout this section we assume a message passing synchronous model and t byzantine processors such that $3t < n$. Note that in contrast to the original consensus template, every algorithm continues to participate in the overall consensus template even after deciding upon a value, as in the original Phase-King algorithm. Algorithms 3 and 4 are the *AC*'s and *conciliator*'s implementations and lemmas 2 and 3 prove the implementations correctness, i.e., ensuring the objects' guarantees.

▶ **Lemma 2.** *Algorithm 3 is a correct adopt-commit implementation.*

Proof. The proof is similar to the Phase-King correctness proof given in [4].
- **Validity**: If all inputs are the same value $v \in \{0, 1\}$ then at the end of the first exchange of the first round all $C(v)$ values are at least $n - t$ for all correct processors (since there are $n - t$ non-byzantine processors) and $C(u) \leq t < n - t$ for $u \neq v$ (since $t < n/3$). Therefore v will remain the chosen value for all non-byzantine processors after exchange 1. The same holds after exchange 2. Thus all correct processors will enter the if statement and receive a value of $(commit, v)$ guaranteeing validity.
- **Convergence**: If all inputs are the same value v then but what we have shown earlier all processors will receive a value of $(commit, v)$ as needed.
- **Termination**: Since we are in the synchronous setting clearly the protocol terminates within a finite amount of rounds.
- **Coherence over commit and adopt**: We first observe that after exchange 1 exists some value $v \in \{0, 1\}$ such that all correct processors' values are either 2 or v (this follows since $t < n/3$ meaning that otherwise we would have some correct processor that broadcasted different values to different processors). Next we observe that in fact this property holds through exchange 2 as well (again, since otherwise we would have a correct processor that broadcasted different values to different processors). Thus, if 2 *commit* messages were received (meaning their values are $v \neq 2$) then their values must coincide.

```
AC(v,m)
    broadcast ⟨v⟩          // (* Exchange 1 *)
    v ← 2
    for k=0 to 1 do
        C(k) ← # received k's
        if C(k) ≥ n − t then
            v ← k
        end
    end
    broadcast ⟨v⟩          // (* Exchange 2 *)
    for k=2 downto 0 do
        D(k) ← # received k's
        if D(k)> t then
            v ← k
        end
    end
    if (v ≠ 2 and D(v) ≥ n − t) then
        return (commit, v)
    else
        return (adopt, v)
    end
```

Algorithm 3: Phase-King's *adopt-commit* implementation

```
Concilliator(X, σ, m)
    if id = m then
        broadcast ⟨MIN(1, v)⟩
    σ_m ← received message from processor m
    return (adopt, σ_m)
```

Algorithm 4: Phase-King's *conciliator* implementation

◀

▶ **Lemma 3.** *Algorithm 4 is a correct conciliator implementation.*

Proof.
- **Validity** Follows since the phase king's inputted value is σ_m.
- **Termination** Clearly guaranteed.
- **Coherence** Only *adopt* is returned therefore coherence holds.
- **Probabilistic Agreement** Since our setting are not probabilistic but rather deterministic we will show eventual agreement, i.e., eventually the concilliator will cause agreement. Consider round m such that processor m is non-byzantine. If during this round all returned values from the adopt-commit object were *adopt* values then following this *concilliator* round all object adopt the same value from the phase-king. Otherwise some processor received at least $n - t$ values that are different than 2 during exchange 2. Since $t < n/3$ this means that $> t$ such values were broadcasted during that exchange. Since as we have seen there is only one value $\neq 2$ that is adopted during exchange 2 and since there are only t byzantine processors, p_m will have adopted that same value and therefore would have broadcasted it through the *concilliator* as needed.

◀

4.2 Ben-Or's Algorithm

In this section we consider Ben-Or's algorithm [3]. Throughout this section the settings are asynchronous, message-passing model and the number of tolerated crash failures, t is strictly

smaller than $n/2$.

After some consideration regarding the decomposition of this algorithm we came to the conclusion that the use of a single adopt-commit followed by a concilliator object is insufficient. This is mainly due to the fact that in Ben-Or's algorithm there are 3 unique types of processor per round - processors that received more than t ratify message, processors that received atleast 1 ratify message but less than t and processors that received no ratify messages. Each type of processor has some guarantee about the state of the network (e.g., if received more than t ratify messages one is ensured the network has achieved consensus). More regarding why adopt-commit is insufficient given in section 5.

Algorithms 6 and 5 are the *VAC*'s and *reconciliator*'s implementations, Lemmas 4 and 5 prove the implementations correctness, i.e., that they uphold the objects' guarantees.

```
VAC(v,m)
    send ⟨1, v⟩ to all
    wait to receive n − t ⟨1, ∗⟩ messages
    if received more than n/2 ⟨1, v⟩
    messages then
        send ⟨2, v, ratify⟩ to all
    else
        send ⟨2, ?⟩ to all
    end
    wait to receive n − t ⟨2, ∗⟩ messages
    if received more than t ⟨2, v, ratify⟩
    messages then
        return (commit, v)
    else if received a ⟨2, v, ratify⟩
    message then
        return (adopt, v)
    else
        return (vacillate, v)
    end
```

Algorithm 5: Ben-Or's *vacillate-adopt-commit* implementation

```
Reconciliator(X, σ, m)
    return CoinFlip()
```

Algorithm 6: Ben-Or's *reconciliator* implementation

▶ **Lemma 4.** *Algorithm 6 is a correct reconciliator implementation.*

Proof. Since any value has a non-zero probability of being outputted, the reconcilliator's guarantee clearly follows. ◀

▶ **Lemma 5.** *Algorithm 5 is a correct vacillate-adopt-commit implementation.*

Proof. The proof is similar to the Ben-Or algorithm correctness proof found in the survey of Aspnes [1].

- **Validity**: Follows since all messages sent and received hold inputted values.
- **Termination**: Since the number of crash failures is less than half, all processors will terminate in a finite amount of time.
- **Convergence**: Assume all processors start out with the same value v. Since the number of crash failures is less than half all live processors will send a ratify message with the same value v. Thus the if statement in line 10 will clearly be satisfied and all processors will receive a value of $(commit, v)$.

- **Coherence over adopt and commit**: If some processor received more than t $(ratify, v)$ messages then by the definition of t at least one live processor broadcast a $(ratify, v)$ message. Thus by line 9 we are guaranteed that every processor received at least one ratify message. Finally the condition in line 4 we are guaranteed that if 2 ratify messages are sent out then they have the same value. Thus all processors received at least one ratify message and they all received the same value.
- **Coherence over vacillate and adopt**: Since the condition in line 4 insures that ratify messages hold the same value, v if some processor received an $(adopt, v)$ message then clearly all other *adopt*-receiving processors received the same value v.

◀

4.3 Raft

In algorithm 7 we use the Raft algorithm [6] to achieve consensus. The raft algorithm is designed for producing a consistent log among distributed systems. Every processor maintains an indexed log of commands which they update continuously. Occasionally the processors apply the commands from their logs to their state machine. The commands applied are always in order and always continue from the last command applied.

Here we describe how the raft algorithm may be used in order to achieve consensus. We use the raft algorithm with a single command (i.e., the logs will consist of a single type of command). The single command used is decide-and-stop-applying-to-state-machine which we denote as $\mathbf{D\&S(v)}$. This command tells the state machine to decide on the value v and stop applying any further commands thereafter (i.e., not to switch its decision).

In the raft algorithm every processor updates an indexed log continuously and occasionally applies the commands given in the log to its state machine. The commands applied are always in order and always continue from the last command applied. Therefore, once a processor decides to update its state machine it will apply the first command in the log, and decide on that given value. This results in the processor deciding upon the first value it sees in its log.

The raft algorithm works as follows; there are 3 states a processor may be in, follower, candidate and leader. Every processor aspires to become a leader and once becoming a leader it tries to have the system decide upon its value. All processors start out as followers and employ a timer. The timer is reset every time the processor receives a message from a fellow processor (with the caveat of terms which will be explained next).

Once a timer runs out the processor converts to candidate and tries to gain enough votes to become leader. Meaning that once becoming a candidate the processor broadcasts *RequestVote* messages and if it achieves a majority of acks it converts to leader.

Once leader, the processor tries to have all other processors append its value to their log by sending them a $\mathbf{D\&S(v)}$ command. Initially the command is sent out as a tentative command. If the leader receives a majority of acks for this append message (denoted by *AppendEntries*) it commits to the command and broadcasts the fact that this command should now be committed to.

Since raft is log based the processors employ a commit-index mechanism in order to achieve the formerly described attributes. An *AppendEntries* message includes a commit index. Meaning that the receiving processor appends the entries it received from the message to its log, however it does not yet apply these commands to its state machine as they may be altered. The processor then looks at the commit index and only then applies all commands up to and including that index in its log. This results in two types of *AppendEntries* messages; the first does not change the commit index of the receiving processor but rather appends

commands the processor's log. The second does aim at appending entries to logs but rather to update the receiving processor's commit index. Since the second type is only sent out once a majority of processors acknowledge the first type, we are ensured that once a command is committed to it will not change (denoted as the **state machine safety** property which will be formally defined later in the section).

The main idea in the algorithm is that in order to achieve consensus we use a 2 step mechanism (not unlike Ben-Or's algorithm). Each processor first tries to gain a majority which would result in leadership. Once leader it then tries to push its a value to decide upon. Only once achieving a majority of acks to that operation, it commits to that log entry and notifies everyone else. We note that both 'wait' operations do not hinder the algorithm's termination since in the background all non-leaders have a randomized timer which has the soul purpose of shaking the protocol out of a stalemate. Once a non-leader's timer runs out it begins the consensus algorithm described in algorithm 7.

The algorithm as described would work just fine in a system without failures, however this is never the case. In real world scenarios processors may fail unexpectedly and messages may be delayed or even lost. In order to maintain log consistency and ensure termination even under these conditions the raft algorithm introduces the notion of terms.

Terms are defined such that leaders are leaders only of a specific (and all lower) term. Once a leader encounters a higher term it immediately reverts to follower and updates its term. Furthermore once a processor converts to candidate and tries to become leader, it increases its term in order to do so. This ensures us that even though some processors may fail ultimately consensus will still be obtained. We note that every processors log consists of indexed entries (indexed continuously from 1, i.e., 1,2,3,...) such that each entry consists of a command and the term in which the command was received.

It may be the case that once a leader is elected it immediately crashes (or it is somehow cut off from a majority of the network). Therefore a different processor will become leader before the earlier one had the chance to alter the processor's logs. This may happen over and over causing a cycle of leaders without any alterations in any of the logs. This would hinder the termination property of our consensus protocol. Therefore the following assumption is made (note that this assumption is made in the original raft paper as well).

We make the assumption that the broadcast time (time it takes to convey a message) is much smaller than randomized timer which is in turn much smaller than the average time between failures of a single machine. This constraint is required in order to maintain a leader which in turn results in consensus termination. We refer to this property as the *timing property*.

We again note the similarity between Ben-Or's algorithm and the algorithm considered here. Both algorithms use a two step mechanism - the first step alerting some processor (the leader in our case) that consensus has been achieved (i.e., all processor's chose the same leader) and the second step conveying that information to all other processors.

Next we turn to formally define the algorithm. Figures 1 and 2 describe the protocol's inner state variables and types of messages used. Note that arrays NextIndex and MatchIndex only apply once a processor is in leader state (and last only for the duration of that term). Furthermore they are reinitialized every time the processor converts to leader. Both these arrays are introduced in order to maintain consistency over the processors' logs. When an AppendEntry message is received the processor may reject it (and return false) if the senders log does not agree with its own (up to a certain degree which will be described next), therefore these 2 arrays help the leader know how far back in its log it has to send to each processor in order to ensure a positive ack.

Algorithms 7, 8 and 9 describe the algorithm of a processor that manages to convey its value (algorithm 7) and the responses to the different messages (algorithms 8 and 9). Note that variables marked with $*$ represent the variables of the processor being described.

We also note the behaviour of processors which receive *AppendEntries* messages (shown in algorithm 9). Once a processor wakes up from a crash it contains an outdated log. Thus once it receives an *AppendEntries* message it may be the case where the processor will have a large chunk of commands missing in its log. In order to prevent such situations the receiving processor has the option of rejecting the message (by returning a false ack). Once a leader receives such an ack it then uses its MatchIndex and NextIndex values in order to go back in its log to the place where the receiving processor crashed. It then retries sending the AppendEntries, however this time the message will include all entries missing in the receiving processor's log (rather than just the last log entry). Therefore, the default leader behaviour would be to only send its last log entry and if that is not enough it continuously retries with an earlier log.

RequestVote[term, candidateId, lastLogIndex, lastLogTerm], where lastLogIndex is the index of the processor's last log and lastLogTerm is the term of that log index.
ack_RequestVote[term, voteGranted], where voteGranted is a boolean variable.
AppendEntries[term, leaderId, prevLogIndex, prevLogTerm, D&S(v), leaderCommit], where prevLogIndex is the index of the log preceding the D&S(v) command and prevLogTerm is its term.
ack_AppendEntries[term, success], where success is a boolean variable.

Figure 1: Raft Consensus Messages

CurrentTerm.
VotedFor - candidateId voted for in current term.
Log[] - indexed list of commands and terms during which they were received.
CommitIndex - log index stating that all commands up and through that index are to be applied to the state machine.
LastApplied - log index of last command applied to state machine.
State - one of follower, candidate or leader.
NextIndex[] (variable applies only while leader) - array of length n (number of processors). Each element is the index of the next log entry to send to that processor. Initialized after election to the leader's last log entry + 1.
MatchIndex[](variable applies only while leader) - array of length n. Each element is the index of highest log entry known to be replicated on that server. Initialized to 0.

Figure 2: Raft Consensus Inner State Variables

We first state a few properties of the Raft algorithm which were stated in [6]. These were also proven in the same paper and therefore, due to space constraints we omit their proofs.

- **Leader Completeness:** if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
- **State Machine Safety:** if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.
- **Log Matching:** if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Next, we turn to prove that the algorithm guarantees consensus.

`Consensus(`v`)`
- state ← candidate
- $v^* \leftarrow \log[lastLogIndex^*]$
- Broadcast RequestVote$[t^*, id^*, lastLogIndex^*, lastLogTerm^*]$
- **Wait** to receive $> n/2$ ack_RequestVote$(t = t^*, granted = true)$
- state ← leader
- $v^* \leftarrow \log[lastLogIndex^*]$
- Broadcast AppendEntries$[t^*, id^*, prevLogIndex^*, prevLogTerm^*, D\&S(v^*), commitIndex^*]$
- **Wait** to receive $> n/2$ ack_AppendEntries$(t = t^*, success = true)$ `// commitIndex is therefore increased, see leader responses`
- Broadcast AppendEntries$[t^*, id^*, prevLogIndex^*, prevLogTerm^*, -, commitIndex^*]$

Algorithm 7: Raft Consensus Protocol

- **if** *received acks_AppendEntries*$(t, false)$ *from* p_i **then**
  - **if** $t > t^*$ **then**
    - state ← candidate, increase term
  - **else**
    - decrement NextIndex[i], resend AppendEntries.
- **if** *received acks_AppendEntries*$(t, true)$ *from* p_i **then**
  - update NextIndex[i] and MatchIndex[i]
  - **if** *exists* N *s.t.* $N > commitIndex^*$, *maj of* $MatchIndex[i] \geq N$, $log[N].term = t^*$ **then**
    - $commitIndex^* \leftarrow N$

Algorithm 8: Raft Consensus Leader Responses

- **if** *received RequestVote[t, id, lastLogIndex, lastLogTerm]* **then**
  - **if** $(t < t*)||(t = t* \&\&$ *votedFor* $\neq$ *null)* **then**
    - return $(t^*, false)$
  - **if** *VotedFor == null && log matches requestor's* **then**
    - return $(t^*, true)$, update t^*
- **if** *received AppendEntries[t, -, prevLogIndex, prevLogTerm, D&S(v), commitIndex]* **then** `// tentatively log D&S(v)`
  - **if** $t < t^*$ **then**
    - return $(t^*, false)$
  - **else if** *log does not match requestor's log at prevLogIndex* **then**
    - return $(t^*, false)$
  - **else**
    - append new entries, delete conflicting ones, if deleted delete all entries that follow as well
    - $commitIndex^* \leftarrow$ min(leaderCommit, index of last new entry)
- **if** *received AppendEntries[t, -, prevLogIndex, prevLogTerm, -, commitIndex]* **then** `// commit to last log entry`
  - **if** $t < t*$ **then**
    - return $(t^*, false)$
  - **else if** *log does not match requestor's log at prevLogIndex* **then**
    - return $(t^*, false)$
  - **else**
    - $commitIndex^* \leftarrow$ min(leaderCommit, index of last new entry)
- **if** $commitIndex^* > lastApplied$ **then**
  - increment lastApplied, apply log[lastApplied] to state machine
- **if** *Timer T runs out* **then**
  - initialize T randomly, increment term and start algorithm 7

Algorithm 9: Raft Consensus Across-State Responses

▶ **Lemma 6.** *Algorithm 7 is a correct consensus protocol.*

Proof. We note that under the definition of $D\&S(V^*)$ a processor will decide upon a value as soon as it increases its commitIndex. Furthermore it will decide upon the value in its first log entry.

We also note that there will never be a majority of processors which failed indefinitely (this assumption has been made in the original raft algorithm as well). We use this assumption in order to guarantee that processors will not remain indefinitely in the *wait* commands in the algorithm. We now prove that the consensus constraints hold.

- **Validity**: Follows from implementation since the only values proposed for consensus are taken from the processor's values.
- **Agreement**: Assume some processor i is the first processor to decide on some value denoted by v_i. Since the leader is always the first to decide upon a value in a given term, we may assume i is a leader and denote its term by t_i. Now, assume some other processor j decided on value v_j during term $t_j \geq t_i$. By the leader completeness property, the entry $D\&S(v_i)$ will appear in t_j's leader's log also in the first entry. Thus for j to increase its commitIndex (and decide on a value) it must have accepted t_j's leader's AppendEntry meaning that by the log matching property their logs must match on atleast the first entry. Therefore j's first entry would be $D\&S(v_i)$ resulting in $v_j = v_i$.
- **Termination**: Leader completeness insures us that if someone commits to a value then eventually all other processors will have that value in their first log index. By the timing property we are insured that eventually all processors increase their commitIndex. This in turn insures us that eventually all processors will decide upon the same value. Furthermore, by our assumption that a majority of live processors will eventually exist

◀

We next turn to show how the consensus protocol can be naturally decomposed using our template. The decomposition works as follows; each term (as described in the consensus protocol) will now refer to a round in our template. This results in the fact that the protocol is unending, however eventually all processors will have committed to some value (as in the original raft protocol).

As in our defined VAC the consensus protocol also results in three types of processors. The first being processors that did not receive a message that a leader was chosen. This matches the vacillate value of VAC in the sense that they have no guarantee regarding the state of the system.

The second type are processors that received an *AppendEntries* message of the first kind, i.e. one that **does not** include a change in the commit index. This matches the adopt value of VAC in that these processors have the guarantee that all other processors which received such a message received it with the same value (this is ensured by the fact that a majority of acks is needed in order to send the message).

The third and final type are processors that received an *AppendEntries* message of the second kind, i.e. one that **does** include a change in the commit index. This matches the commit value of VAC in that these processors are guaranteed that a consensus has been reached (even if not all processors are aware of it), i.e., that all processors receive the same value (being accompanied by either an adopt or commit value). This property is ensured by the *leader completeness* and *state machine safety properties*.

The reconcilliator in our case is aimed at capturing the timer mechanism. In the consensus algorithm the timer mechanism was introduced in order to ensure no stalemate was reached, i.e., to eventually cause convergence. The reconcilliator object was introduced to do just that

and therefore we define it to capture the timer mechanism as closely as possible. Interestingly enough in this case, as opposed to Ben-Or for example, it is not the returned value that causes the wanted behaviour (i.e., prevention of a stalemate) but rather the timing of processors entering the reconcilliator.

In algoritm 10 we define our VAC protocol and in algorithm 11 we define the reconciliator object. We note in addition to the VAC stated we define 2 more changes to the raft consensus protocol. The first is that if a follower recieves an *AppendEntry* message of the first type (i.e., with an appended entry but without an increase in the commit index) and accepts the message, then it also sets its X and v values to *adopt* and the value it sees in its last log entry. The second is that if a follower recieves an *AppendEntry* message of the second type and accepts the message, then it sets X to *commit* and v to the value it sees in its last log entry.

```
VAC(v)
    (X, v*) ← (Vacillate,
    log[lastLogIndex*].value)
    state ← candidate
    Broadcast RequestVote[t*, id*,
    lastLogIndex*, lastLogTerm*]
    Wait to receive > n/2
    ack_RequestVote(t = t*, granted =
    true)
    Freeze timer T
    (X, v*) ← (Adopt,
    log[lastLogIndex*].value)
    state ← leader
    Broadcast AppendEntries[t*, id*,
    prevLogIndex*, prevLogTerm*,
    D&S(v*), commitIndex*]
    Wait to receive > n/2
    ack_AppendEntries(t =
    t*, success = true) /*commitIndex is
    therefore increased, see leader
    responses*/
    (X, v*) ← (Commit,
    log[lastLogIndex*].value)
    Broadcast AppendEntries[t*, id*,
    prevLogIndex*, prevLogTerm*,
    −, commitIndex*]
```

Algorithm 10: VAC Protocol

```
Reconcilliator(v)
    Reset timer and update term
    D&S(v) ← log[lastLogIndex]
    return v
```

Algorithm 11: Reconcilliator Protocol

▶ **Lemma 7.** *Algorithm 10 is a correct VAC protocol.*

Proof. In order to ensure that our guarantees hold we prove them for processors which have not failed during the term. Processors which fail during the term adopt the higher term once waking up anyhow and should therefore be ignored.

Validity: Since all values written into the logs were written using received values validity is insured.

- **Convergence**: We refer the reader to the note following the proof.
- **Termination**: By our assumption that no majority of processors will crash-fail clearly the process will eventually terminate.
- **Coherence over adopt and commit**: As discussed above this is guaranteed by the *leader completeness* and *state machine safety properties*.
- **Coherence over vacillate and adopt**: As discussed above this is ensured by the fact that a majority of acks is needed in order to send an adopt message.

◀

We note that under the raft algorithm infrastructure since consensus is achieved by first electing a leader, convergence does not hold as is. This is indeed plausible due to the fact that the algorithm was made for real world log consistences rather than theoretical consensus. For theoretical purposes one may easily convert the algorithm such that it holds convergence by converting the *wait* steps to *broadcast* steps. I.e., decentralize the messages meaning that instead of electing a leader and having him in charge of logging commands, everyone broadcasts the command they want logged and once someone sees a majority it sends out a commit-to-that-command message. This would result in convergence since if all processors agree on the same value in the first place, all steps would be easily passed.

Interestingly enough, this change results in an algorithm that highly resembles Ben-Or's. The only difference is in the way it handles stalemates, or in other words, the reconciliators implemented are different.

5 *Adopt-Commit* is Not Enough

The concept of decomposing consensus into separate objects is by no means original and was formally presented in [5]. Later work by Aspnes [2] described a framework of *adopt-commit* objects that detect agreement, and *conciliators* that ensure agreement with some probability. We argue that this decomposition fails to capture the inner workings of some well-known algorithms. In these algorithms 3 different types of processors exist throughout the process of achieving coensus; the first are processors which have no guarantee regarding the state of the system. The second are processors that are guaranteed that they are part of a subset of processors that achieved consensus (all other processors are of the first type). The last are processors that are guaranteed that the network has achieved consensus (while not all processors are aware of this fact).

In order to make our argument more concrete, we demonstrate how Ben-Or's consensus algorithm cannot be described by a sequence of *adopt-commit* alternating with *conciliator*, while it is naturally described as a sequence of repetitive *vacillate-adopt-commit* followed by *reconciliator*.

To demonstrate the problem with formulating Ben-Or's consensus protocol using

$$U = A_{-1}; A_0; C_1; A_1; C_2; A_2; \ldots,$$

consider each round of Ben-Or's algorithm [3][2]. Let P be a processor participating in the agreement process. P experiences one of three possible outcomes: (1) not receiving any *ratify* message. (2) receiving up to t *ratify* messages. (3) receiving more than t *ratify* messages.

[2] We note that our description follows the presentation of Ben-Or's algorithm given in the survey paper of Aspnes [1]

These outcomes correspond to vacillate, adopt, and commit, respectively. Option 1 fits a processor which received vacillate as it has no guarantees about other values received by other processors. Option 2 corresponds to *adopt* under the *VAC* framework, since by *coherence*, any processor that received $(adopt, v)$ is guaranteed that every other processor that received either *vacillate* or *commit*, also received the value v. Option 3 corresponds to *commit*, since any processor that received $(commit, v)$ is guaranteed that all other processors received either $(commit, v)$ or $(adopt, v)$.
However, using only *adopt-commit* objects is not enough in order to describe these three options.

It might be tempting to assume that two consecutive *adopt-commit* objects might resolve this entanglement as we have shown that *VAC* may be implemented using two *AC* objects. Note that the concatenation of the *AC* objects (as proposed in [2]) is in a way that is different than our proposed *VAC* implementation since in their case vacillate-receiving processors are not represented.

We argue this is not the case, that is, we claim that the sequence of $U = A_{-1}; A_0^0; A_0^1; C_1; A_1^0; A_1^1; C_2; \ldots$ also fails to describe Ben-Or's consensus protocol. In order to describe option (2) the first *adopt-commit* must return *adopt* while the second returns *commit*. However, the decomposition framework described in [2] requires that upon receiving *commit* the processor immediately decides on the value received, whereas it is possible that in Ben-Or's protocol such a state is reached with value u but a final agreement is achieved with value $u' \neq u$.

6 Conclusions

Motivated by the desire to provide a natural decomposition of consensus into building blocks that describe known algorithms, we defined a more subtle object than *adopt-commit*, the *vacillate-adopt-commit*, which in turn simplifies the role of the *reconciliator* such that in some cases it is only a procedure that flips a coin and does not require machinery to ensure *validity*. Using these building blocks we demonstrate how well known consensus algorithms decompose into a unified template of a repetitive two step process. We hope a better understanding of the consensus object may allow research of complexity bounds of the newly introduced building blocks which in turn may be deduced to consensus.

References

1 James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2–3):165–175, September 2003.

2 James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Computing*, 25(2):179–188, May 2012.

3 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30, 1983. URL: `http://doi.acm.org/10.1145/800221.806707`, `doi:10.1145/800221.806707`.

4 Piotr Berman, Juan A Garay, and Kenneth J Perry. Towards optimal distributed consensus. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 410–415. IEEE.

5 Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 143–152. ACM, 1998.

510 **6** Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm.
511 In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA,*
512 *June 19-20, 2014.*, pages 305–319, 2014. URL: `https://www.usenix.org/conference/`
513 `atc14/technical-sessions/presentation/ongaro`.
514 **7** Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence
515 of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.