Object Oriented Consensus

Yehuda Afek * James Aspnes † Edo Cohen * Danny Vainstein*

Abstract

We suggest a template that reveals the structure of many consensus algorithms as a generic procedure. The template builds on an extension of the well known *adopt-commit* protocol which we call ***vacillate-adopt-commit*** and on an extension of Aspnes' *conciliator* protocol, which we call ***reconciliator***. The consensus algorithm template works in rounds in each of which the *vacillate-adopt-commit* is invoked and if necessary also the *reconciliator* is called upon. The *vacillate-adopt-commit* object observes the processors preferences and suggest a preference output with a measure of confidence (*vacillate, adopt* or *commit*) on the preference. The *reconciliator*, is used when there is no overall clear agreement, to ensure termination. To show that our template correctly captures a natural structure of consensus algorithms we put three well known algorithms within our template: The *Phase King Byzantine* algorithm, the *Ben-Or randomized* algorithm and the *Paxos* algorithm. We analyze and compare our template based on *vacillate-adopt-commit* and *reconciliator* objects to previous work [1, 2], suggesting a decomposition of consensus based on *adopt-commit* and *conciliator* objects. We claim that the three return values of *vacillate-adopt-commit* more accurately describe existing algorithms.

Submitted as a regular paper.
This is a student paper (Danny Vainstein and Edo Cohen are full-time M.Sc. students).

*School of Computer Science, Tel-Aviv University. E-mails: afek@tau.ac.il, danvainstein@tau.ac.il, edocohen@tau.ac.il

†School of Computer Science, Yale University. E-mail: james.aspnes@gmail.com

1 Introduction

The consensus problem introduced by Lamport, Pease and Shostak [3] resides at the heart of many distributed algorithms such as leader election, database transaction handling, resource allocation, ensuring storage replicas are mutually consistent 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 of various types.

Gafni [1] 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 more accurately than the previous decomposition.

The paper is structured as follows. In Section 3, the consensus template is presented. In Section 4, we demonstrate how the decomposition applies to various algorithms illustrating different modes of failure. Section 5 explores 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

Throughout the paper we abbreviate *vacillate-adopt-commit* as *VAC* and Adopt-Commit as *AC*. Moreover, returned values *vacillate, adopt, commit* may be denoted as *V, A, C* respectively when clear from context.

We consider a network consisting of n processors, each with an input value v, an object may be called by any subset of the processors where each object must provide a few guarantees.

A ***consensus*** object guarantees the following: ***(1) validity***, the returned value u must be an input to some processor; ***(2) termination***, the object must return a value within a finite amount of time; and ***(3) agreement***, all processors invoking the object receive the same value u.

An ***adopt-commit*** object weakens *consensus* by returning a value along with one of two options, ***adopt*** or ***commit***. The guarantees offered are: *(1)* ***validity*** and *(2)* ***termination***, as before; ***(3) coherence***, which states that if some processor receives $(commit, v)$, any other returned value (with either *commit* or *adopt*) must have the same value v; ***(4) convergence***, which states that if all processors invoke $AC(v)$ with the same value v, the object must return $(commit, v)$ with the same value v to all processors.

The newly introduced ***vacillate-adopt-commit*** accepts a value v at each processor and returns a value u to that processor tagged with one of three confidence levels, ***vacillate***, ***adopt*** and ***commit*** (e.g., $(commit, v)$). The object holds the following guarantees. ***Validity*** and ***termination*** which are defined as defined in the *adopt-commit* case. We define ***convergence*** such that if the *vacillate-adopt-commit*'s inputs are all the same value, u, then it must return to all processors the value, $(commit, u)$. We further add two more guarantees, *coherence over adopt & commit* and *coherence over vacillate & adopt*. ***Coherence over adopt & commit*** states that 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*). The ***coherence over vacillate & adopt*** guarantee states that 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).

The ***reconciliator*** object guarantees to return the same value to all processors with nonzero probability (that may depend on n, the number of processors). In the *Byzantine* setting, the *reconciliator* guarantees to return the same value to all processors eventually (i.e., if called sufficiently many times). We note that the *reconciliator* weakens Aspnes' *conciliator* [2] definition in that it does not require *validity* (i.e., that the

reconciliator's returned value equals some processor's input). This is due the fact that if all (non-Byzantine) processors' inputs are equal, the *VAC* is defined such that no processor will, anyhow, reach the *reconciliator*.

3 The Generic Form of Consensus

The thesis of 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 literature.

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

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 such 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.

```
1  Consensus(v)
2      m ← 0
3      INIT()
4      while true do
5          m ← m + 1
6          (X, σ) ← VAC(v, m)
7          switch X do
8              case vacillate: v ← Reconciliator(X, σ, m) do
9              case adopt: v ← σ do
10             case commit: v ← σ and decide σ do
11         end
12     end
```

Algorithm 1: Consensus Template

Note that *INIT* is a void function unless stated otherwise. Furthermore, note that the operation, $decide\ \sigma$,

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*** and ***validity*** follow from *VAC*'s *coherence* and *validity* respectively. ***Termination*** follows from the *convergence* property of the *reconciliator* object. □

4 Consensus Decomposition

Here we fit several well known consensus algorithms into the framework by providing implementations of the *vacillate-adopt-commit* and *reconciliator* objects for each algorithm.

4.1 Phase-King Algorithm

Here we show how the Phase-King consensus protocol of Berman, Garay and Perry [4] fits framework. 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 2 and 3 are the *VAC*'s and *reconciliator*'s implementations, Lemmas 2 and 3 prove the implementations correctness, i.e., that they uphold the objects' guarantees.

```
1 Reconciliator(X, σ, m)
2 |  σ_m ← received message from processor m
3 |  return σ_m
```

Algorithm 2: Phase-King's *reconciliator* implementation

Lemma 2. *Algorithm 2 is a correct reconciliator implementation.*

Proof. For round m, such that processor m is non-byzantine, all non-byzantine processors receive and therefore return the same value. Since $t < \frac{n}{3}$ and the algorithm runs for n rounds, such m exists, giving us the desired result. □

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

Proof. The proof is similar to the Phase-King correctness proof given in [4].

Validity, clearly the protocol returns either 0 or 1, therefore it is enough to show that if all inputs are 0 the protocol only returns 0 and likewise if all inputs are 1. This indeed follows by the definition of both exchanges.

Termination, we can assume every processor terminates (and decides on its value) after $m + 1$ rounds, ensuring termination.

Coherence over commit and adopt, assume some processor receives a value of $(commit, x)$. Observe that at the end of Exchange 1 there exists $u \in \{0, 1\}$ such that, for all correct processors, the value of v is either u or 2. Further observe that this also holds after Exchange 2. Assume some other correct processor received a $vacillate$ value. This would imply that at the end of Exchange 2, he would of have $D(x) \leq t$. However, the received $commit$ implies that his $D(x)$ is at least $n - t > 2n/3$, implying that in Exchange 2

```
1  VAC (v,m)
2      send ⟨v⟩ to all        // (* Exchange 1 *)
3      v ← 2
4      for k=0 to 1 do
5          C(k) ← # received k's
6          if C(k) ≥ n − t then
7              v ← k
8          end
9      end
10     send ⟨v⟩ to all        // (* Exchange 2 *)
11     for k=2 downto 0 do
12         D(k) ← # received k's
13         if D(k)> t then
14             v ← k
15         end
16     end
17     if id == m then
18         send ⟨σ⟩ to all
19     end
20     if (v == 2) then
21         return (vacillate, MIN(1, v))
22     else if D(v) < n − t then
23         return (adopt, v)
24     else
25         return (commit, v)
```

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

atleast $\frac{n}{3} > t$ values of x were sent out, resulting in a contradiction. To complete the proof, we assume some other correct processor received (X, y) such that $X \in \{adopt, commit\}$ and show that $x = y$. Indeed, this follows immediately by our second observation.

Coherence over adopt and vacillate, if two processors received the values $(adopt, x)$ and $(adopt, y)$, by our second observation we have $x = y$.

Convergence, follows by what was shown in *validity*.

□

4.2 Ben-Or's Algorithm

In this section we show how Ben-Or's algorithm [5] can be described using our consensus template. Throughout this section the settings are asynchronous, message-passing model and the number of tolerated crash failures, t, is strictly smaller than $n/2$. Algorithms 4 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.

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

Algorithm 4: Ben-Or's *reconciliator* implementation

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

Proof. ***Probabilistic convergence***,
$\mathbb{P}$(all processors finish with the same value after the coin flip) $\geq \frac{2}{2^n} > 0$. □

```
1  VAC (v,m)
2      send ⟨1, v⟩ to all
3      wait to receive n − t ⟨1, ∗⟩ messages
4      if received more than n/2 ⟨1, v⟩ messages then
5          send ⟨2, v, ratify⟩ to all
6      else
7          send ⟨2, ?⟩ to all
8      end
9      wait to receive n − t ⟨2, ∗⟩ messages
10     if received more than t ⟨2, v, ratify⟩ messages then
11         return (commit, v)
12     else if received a ⟨2, v, ratify⟩ message then
13         return (adopt, v)
14     else
15         return (vacillate, v)
16     end
```

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

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 [6].

Validity, termination and ***convergence*** follow easily from the implementation.

Coherence over commit and adopt, assume a process received $(commit, v)$. Therefore, it received more than t $ratify$ messages, meaning a non-faulty processor sent out a $ratify$ message to all other processors - therefore it is enough to show that every 2 $ratify$ messages have the same value, v (since then all processes received atleast one $ratify$ message with the same value, v). Assume towards contradiction that this isn't the case - meaning messages $(2, v, ratify)$ and $(2, u, ratify)$ where $u \neq v$ had been sent. By the first if statement, this means there's a processor that sent out both u and v which is a contradiction.

Coherence over adopt and vacillate, since every 2 $ratify$ messages have the same value, v, coherence follows. □

4.3 Paxos

We consider a synchronous variant of Paxos based on Lamport's original algorithm [7] (which was originally designed for the asynchronous environment). Paxos is designed to handle a very permissive fault tolerant mechanism, a processor is allowed to fail and continue running as long as it preserves its internal state at the cost of weaker guarantees (namely, termination is not guaranteed). We derive a variant of Paxos which is tolerant to $t < \frac{n}{2}$ fail-stop faults. We make the following standard assumptions: a) Each processor is a ***proposer, acceptor*** and ***learner*** (as opposed to Lampert's original algorithm where each entity may be implemented on a separate processor) b) an oracle generating unique integers is accessible for every processor.

In what follows we first describe our variation of the Paxos algorithm (algorithm 6). In order to better decompose the said algorithm using our framework, we first slightly alter our consensus template (algorithm 7). Finally, we describe the decomposition of our variant of Paxos using the slightly altered consensus template (algorithms 8, 9, 10).

Paxos Synchronous Implementation

In algorithm 6 we describe the synchronous fail-stop tolerant Paxos algorithm followed by a proof that the stated algorithm does indeed fulfill validity, agreement and termination.

Paxos can be presented in 3 phases of communication:

- **Proposition** - In this phase each processor proposes itself via a unique integer. Each processor receives proposals from others and 'listens' to the proposal which is accompanied by the maximum integer received (denoted as the *maximal proposal*).
- **Promising** - Each processor promises to 'listen' to the processor which sent the maximal proposal. The promise message contains information of the most recent value and number that were accepted by the promising processor (denoted by $v_{accepted}, k_{accpted}$).
- **Acceptance** - A processor that received a majority of listeners sends an accept command to all its listeners with a value that it has chosen from all the promise messages it has received in the previous phase. The value chosen corresponds to the maximal $k_{accpted}$ received.

A more comprehensive description is given in [7].

Algorithm 6 is a formal description of our variant of the Paxos algorithm.

Lemma 6. *Algorithm 6 satisfies validity, agreement and termination.*

Proof. The proof is given in the appendix. □

Paxos Consensus Template

Since in Paxos the integers chosen by processors have a special role, the consensus framework requires a slight modification with the arguments passed and returned from each procedure. Specifically, the *reconciliator* chooses a new integer and implements a communication phase from which it infers the maximum integer chosen by any other processor and returns the pair of k_{self}, k_{max} (note that there is no prevention that $k_{self} = k_{max}$). Furthermore the *VAC* receives as arguments k_{self} and k_{max} which play an important role as described in algorithm 10. The modified consensus template is given in Algorithm 7.

Paxos is naturally decomposed to fit the new template such that the first promise phase is implemented by the *reconciliator*, and the remaining phases are implemented by the *VAC*. We present Paxos' decomposition using our framework. The implementation of the *INIT* and *reconciliator* objects are given in Algorithms 8 and 9.

The original definition of the *reconciliator* requires that with nonzero probability all returned values by the *reconciliator* are the same. Since Paxos differs from other consensus algorithms by giving special significance to the numbers generated, it is natural to require that with probability greater than 0 all processors return the same value k. This indeed occurs if all processors calling the *reconciliator* do not fail during its operation.

We turn to present the *VAC* implementation and prove its correctness.

Lemma 7. *Algorithm 10 is a correct vacillate-adopt-commit implementation.*

Proof. The proof is given in the appendix. □

```
1  Paxos(v)
2  begin
3      v_accepted ← null
4      k_accepted ← null
5      while true do
6          // (* Exchange 1 *)
7          k_self ← getInt()
8          send ⟨propose, k_self⟩ to all
9          wait to receive n − t ⟨propose, k_i⟩ messages
10         k_max ← n_i s.t. for every two messages received k_i ≥ k_j
11
12         // (* Exchange 2 *)
13         send ⟨promise, k_max, v_accepted, k_accepted⟩ to all
14         wait to receive n − t ⟨promise, k^j_max, v^j_accepted, k^j_accepted⟩ messages
15         if k_self == k_max then
16             if received more than n/2 messages s.t. k_self == k^j_max then
17                 v_propose ← v^i_accepted s.t. for every two messages received k^i_accepted ≥ k^j_accepted
18             else
19                 v_propose ← σ
20             end
21             send ⟨accept!, v_propose, k_self⟩
22         end
23
24         // (* Exchange 3 *)
25         if received ⟨accept!, u, k_u⟩ and k_u == k_max then
26             v_accepted ← u
27             k_accepted ← k_u
28         end
29         send ⟨accepted, v_accepted, k_accepted⟩
30         wait to receive n − t ⟨accepted, r, k_r⟩ messages
31         if received more than n/2 messages s.t. k_r == k_accepted then
32             V ← r
33         end
34     end
35 end
```

Algorithm 6: Paxos synchronous implementation based on [7]

```
1  Consensus(v)
2      k_self, k_max ← INIT(v)
3      while true do
4          (X, σ) ←
             VAC(v, k_self, k_max)
5          switch X do
6              case vacillate do
7                  v ← σ
8                  k_self, k_max ←
                     Reconciliator()
9              case adopt do
10                 v ← σ
11             case commit do
12                 v ← σ
13                 decide σ
14         end
15     end
```

Algorithm 7: Paxos Consensus Template

```
1  INIT()
2      v_accepted ← null
3      k_accepted ← −1
4      return Reconciliator()
```

Algorithm 8: Paxos *Init* implementation

```
1  Reconciliator()
2      k_self ← getInt()
3      send ⟨propose, k_self⟩ to all
4      wait to receive n − t ⟨propose, k_i⟩
         messages
5      k_max ← k_i s.t. for every two
         messages received k_i ≥ k_j
6      return (k_self, k_max)
```

Algorithm 9: Paxos *Reconciliator* implementation

```
1  VAC (v, k_self, k_max)
2      send ⟨promise, k_max, v_accepted, k_accepted⟩ to all
3      wait to receive n − t ⟨promise, k^j_max, v^j_accepted, k^j_accepted⟩ messages
4      if n_self == n_max then
5          if received more than n/2 messages s.t. k_self == k^j_max then
6              v_propose ← v^i_accepted s.t. for every two messages received k^i_accepted ≥ k^j_accepted
7          else
8              v_propose ← v
9          end
10         send ⟨accept!, v_propose, k_self⟩
11         return (adopt, v_propose, k_self)
12     end
13
14     if received ⟨accept!, u, k_u⟩ and k_u == k_max then
15         v_accepted ← u
16         k_accepted ← k_u
17     end
18     send ⟨accepted, v_accepted, k_accepted⟩
19     wait to receive n − t ⟨accepted, r, k_r⟩ messages
20     if received more than n/2 messages s.t. k_r == k_accepted then
21         return (commit, r)
22     else
23         return (vacillate, v)
24     end
```

Algorithm 10: Paxos *vacillate-adopt-commit* implementation

5 *Vacillate-Adopt-Commit* vs *Adopt-Commit*

5.1 Relation Between *Vacillate-Adopt-Commit* and *Adopt-Commit*

Vacillate-adopt-commit can be used to simulate an *adopt-commit* object as seen in Algorithm 11. Clearly all *adopt-commit* properties are preserved due to the stronger properties of *VAC*. The more challenging task is to simulate *VAC* using *adopt-commit* objects. We give such an implementation in Algorithm 12.

```
1 Adopt-Commit(v)
2 |  (X, σ) ← VAC(v)
3 |  if X == commit then
4 |  |  return (commit, σ)
5 |  else
6 |  |  return (adopt, σ)
7 |  end
```

Algorithm 11: *adopt-commit* using *vacillate-adopt-commit*

```
1  VAC(v)
2  |  (X, σ) ← Adopt-Commit(v)
3  |  (Y, φ) ← Adopt-Commit(σ)
4  |  if X == commit then
5  |  |  return (commit, σ)
6  |  else if Y == commit then
7  |  |  return (adopt, φ)
8  |  else
9  |  |  return (vacillate, φ)
10 |  end
```

Algorithm 12: *vacillate-adopt-commit* using *adopt-commit*

We claim that all of *vacillate-adopt-commit*'s properties are preserved using this implementation. Clearly, ***convergence, validity*** and ***termination*** follow due to *adopt-commit*'s properties. The tricky part is ***coherence***. If X is *commit*, then by the *coherence* of *adopt-commit*, all operations over the first *adopt-commit* object return $(commit, \sigma)$ or $(adopt, \sigma)$ and from *convergence* of the second *adopt-commit*, Y is *commit* for all operations. In the case X is *adopt* and Y is *commit*, again from *coherence* of *adopt-commit* we can return $(adopt, \phi)$ as we are promised no other value will be returned by the second *adopt-commit* due to its *coherence* property. If both *adopt-commit* returned *adopt*, the algorithm returns $(adopt, \phi)$. Note that the second *adopt-commit* might return $(commit, \phi)$ to some operations. This is valid behavior, since the *coherence* of *adopt-commit* ensures *coherence over adopt and vacillate*.

5.2 *Adopt-Commit* is Not Enough

The concept of decomposing consensus into separate objects is by no means original and was formally presented in [1]. 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 breaches abstraction responsibilities. The consensus algorithm may be viewed as an iterative process which begins in some arbitrary state and terminates with an agreement upon a *valid* value. Under this interpretation, the framework of repetitive *adopt-commit* followed by *conciliator* fails to distinguish between the phases of the consensus procedure in which part of the participants have terminated and others have not. Due to this limitation and the *coherence* constraint, the *conciliator* must be aware of the global state of the consensus procedure in order to ensure *validity*.

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 [5][1]. 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.

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. 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 reception of *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*. Under our proposed framework, we have shown how well known algorithms fall into the same general abstraction of a simple consensus algorithm. 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] 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.

[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] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[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.

[1] We note that our description follows the presentation of Ben-Or's algorithm given in the survey paper of Aspnes [6]

[5] 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.

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

[7] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

Appendix

A Proofs From Section 4.3

Proof of Lemma 6. ***Validity*** is clear since all values sent origin as some processors initial value v. ***Agreement*** is less trivial and is proved by showing that in any round[2] there may be at most one *Accept!* message sent. Assume towards a contradiction that two such messages were sent, $\langle accept!, u, k_u\rangle$ and $\langle accept!, v, k_v\rangle$. w.l.o.g. $k_v < k_u$ (recall that all integers are unique). Let C_v be the set of processors which sent $\langle promise, k_v, w_{accepted}, k_{accepted}\rangle$ and similarly, C_u is the set of all processors which sent $\langle promise, k_v, v_{accepted}, k_{accepted}\rangle$. Clearly $C_v \cap C_u \neq \emptyset$ as $n-t > \frac{n}{2}$. Let q be a processor s.t. $q \in C_u \cap C_v$, q has sent $\langle promise, k_v, v_{accepted}, k_{accepted}\rangle$ and $\langle promise, k_u, v_{accepted}, k_{accepted}\rangle$ therefore at the same time $k^q_{max} = k_v \neq k_u = k^q_{max}$ which is a contradiction. □

Proof of Lemma 7. ***Termination*** is trivial as in order for each processor to execute successfully *VAC* there must be at least $\frac{n}{2} > n - t$ cooperating processors.

Validity is trivial since no non valid values are transferred.

Coherence by showing that at most one processor sends $\langle accept!, v, k\rangle$ and any $(commit, u)$ will suffice $u = v$ we will guarantee both coherence over *adopt-commit* and over *vacillate-adopt* will hold. Assume that two processors q and q' have issues $\langle accept!, v, k\rangle$ and $\langle accept!, v', k'\rangle$ correspondingly, let C_q and $C_{q'}$ be the sets of processors promising to q and q'. Clearly $C_q \cap C_{q'} \neq \emptyset$, let $\hat{q} \in C_q \cap C_{q'}$. w.l.o.g. $k' > k$, by the algorithm $\hat{q}$ has sent a promise message only to q' with the value k' which is contradiction. It is left to show that if some processor has returned $(adopt, v)$ then for any returned $(commit, u)$ it holds that $u = v$. Assume q returned $(adopt, v)$ and q' returned $(commit, u)$, since q and q' have not failed until returning *adopt* and *commit* they have successfully sent their messages to all neighbors, therefore any valid processor sending $\langle accepted, r, k_r\rangle$ holds $r = v$ and therefore q' cannot receive more than $\frac{n}{2}$ $\langle accepted, u, k_u\rangle$ messages such that $u \neq v$. □

[2]For the sake of analysis all message are accompanied by a counter that is incremented at the beginning of each iteration of the infinite loop, denoted the round by k. This assumption does not affect the algorithm as it already happens implicitly since for each phase to continue at least $n - t$ processors must be at the same implicit round.