Nova Univerisy of Lisbon
NOVALINCS Technical Reports

TR-201501NOVALINCS01
FCT/UNL, Caparica, Portugal

Lower Bound and Correctness Proofs for Consensus in the Visigoth Model

Daniel Porto[†], *João Leitão*[†], *Cheng Li*[‡], *Allen Clement*[‡], *Aniket Kate*[♮],
Flavio Junqueira[♯] *and Rodrigo Rodrigues*[†]
[†]*NOVA Univ. Lisbon / NOVA LINCS,* [‡]*MPI-SWS,*[♮]*MMCI/Saarland University,* [♯]*Microsoft Research*

1 Visigoth model

The Visigoth model makes the following set of assumptions.

- The system consists of a set of n processes.
- Processes communicate by sending and receiving messages over bidirectional links.
- The network is asynchronous (subject to the restrictions below), and thus messages can be lost, duplicated, or arbitrarily delayed. We also assume that links are pairwise-authenticated, meaning that if process i receives a message m in the incoming link from process j, then process j sent message m to i beforehand.
- Each process executes a sequence of steps (actions) triggered upon either: receiving a message, an internal timer expiring, receiving an external input, or a condition in the state becoming true.
- Processes may fail by crashing permanently, or by suffering a commission fault, otherwise they are non-faulty. Commission-faulty processes may send any number of arbitrary faulty messages throughout the entire execution (subject to the remaining model constraints). We say that a process is "correct" if it does not suffer a commission fault throughout its execution.
- There are at most u faulty processes (crash+commission).
- In the definition of the actions of a process for implementing a given protocol, some of these actions can be marked by the protocol writer as "message collection steps". These must be internal steps resulting from a condition over the state becoming true, and the condition that triggers these actions must be that the process collected in its internal state a sufficiently large threshold of messages from distinct processes that meet a certain condition. (Both the threshold and the condition are protocol-specific.) **Example.** For example, a message collection step can be triggered by gathering a majority (or a quorum) of messages of the same type and for the same sequence number from distinct processes.
- At most o processes may suffer commission faults that are correlated. We define that commission faults are correlated if they lead to sending incorrect messages that can be used to trigger the same message collection step. More precisely, assuming that S is a state variable maintained by a process in the protocol, if the condition that triggers a message collection step is of the form:
  $|\{m \in S \text{ from distinct } p_i : C(m) = \textit{true}\}| > t$

then there are up to o commission faults that can lead to incorrect messages m such that $C(m) = \textit{true}$.
Example. In the previous example of a message collection step, this means that the majority (or quorum) of messages (of the same type and for the same sequence number) that were collected cannot contain more than o messages that were generated processes that did not follow the correct protocol due to a commission fault.

- We define that process i is slow with respect to j if one or more messages from i to j or j to i take longer than T time units to be transmitted. (By not taking longer than T to be transmitted, we mean that the time from the execution of the action that sent a message to the corresponding message receive action does not exceed T.)
- For any process, there is a maximum number s of other processes that are slow with respect to it. (While we assume that the set of s processes that are slow with respect to a given process remains fixed throughout the entire execution, in practice it is possible to relax this assumption by only requiring that it remains fixed for a "long enough" period that allows for the execution of a given protocol instance.)
- Processes and clocks need not be synchronized. Nevertheless, we assume that a clock drift is bounded such that timers can be used to safely determine if a certain amount of time (related to T) has passed.

2 Lower bound

To obtain our lower bound result, we consider the problem of solving consensus in a distributed system.

2.1 Problem Statement

In this context, for a given instance c, consensus has the following four properties:

Termination: Every correct process eventually decides some value.

Validity: If all correct processes propose the same value v, and a correct process decides v' then $v' = v$.

Integrity: No correct process decides twice.

Agreement: No two correct processes decide differently.

2.2 Proof

Theorem 1. *Given a VFT system of n processes, with $n < u + o + min(u, s) + 1$, there is no solution for consensus.*

Proof. (by contradiction)
We consider two cases, namely when $u > s$ and $u \leq s$.
Case I: Here, we assume that $u > s$.
Suppose there exists an algorithm that solves consensus with $n = u + o + s$. We can partition the set of processes into four sets, A, B, C and D: A and B with s processes each, C with $u - s$ processes, and D with o processes. We can construct the following three runs of the algorithm and in the last one we show that it violates the agreement property.
Run 1: Processes in B and C crash at the beginning of the run (time t_0); all other process are correct. All input proposals are v.
- By termination and validity, all processes in A and D eventually decide v at a time t_3.

Run 2: Processes in A and C crash at the beginning of the run (time t_0); all other process are correct. All input proposals are v'.

- By termination and validity, processes in B and D eventually decide v' at a time t_4.

Run 3: Similar to Run 1, except that: i) processes in B propose v' instead of crashing; ii) instead of proposing v, processes in D suffer a commission fault which makes them send different proposals to processes in A and B: they act as in Run 1 towards processes in A, and they act as in Run 2 towards processes in B; and iii) processes in A are perceived as slow by processes in B, while processes in A are perceived as slow by processes in B until max(t_3, t_4), resulting in messages between A and B only being delivered to processes in the other group after that time.

Notice that there is no process in the third run that perceives more than s processes as being slow, thus satisfying the assumptions the Visigoth model.

- Processes in A eventually decide v at a time t_3, since it is indistinguishable from Run 1.
- Processes in B eventually decide v' at a time t_4, since it is indistinguishable from Run 2; this violates the *agreement* property.

Case II: Here, we assume that $u \leq s$.

Suppose there exists an algorithm that solves consensus with $n = u + o + u$. We can partition the set of processes into three sets, A, B, and C: A and B with u processes each, and C with o processes. We can construct the following three runs of the algorithm and in the last one we show that it violates the agreement property.

Run 1: Processes in A crash at the beginning of the run (time t_0); all other process are correct. All input proposals are v.

- By termination and validity, all processes in B and C eventually decide v at a time t_3.

Run 2: Processes in B crash at the beginning of the run (time t_0); all other process are correct. All input proposals are v'.

- By termination and validity, processes in A and C eventually decide v' at a time t_4.

Run 3: Similar to Run 1, except that: i) processes in A propose v' instead of crashing; ii) instead of proposing v, processes in C suffer a commission fault which makes them send different proposals to processes in A and B: they act as in Run 1 towards processes in A, and they act as in Run 2 towards processes in B; and iii) processes in A are perceived as slow by processes in B, while processes in B are perceived as slow by processes in A until max(t_3, t_4), resulting in messages between A and B only being delivered to processes in the other group after that time.

Notice that there is no process in the third run that perceives more than s processes as being slow as in this case $s > u$, thus satisfying the assumptions the Visigoth model.

- Processes in B eventually decide v at a time t_3, since it is indistinguishable from Run 1.
- Processes in A eventually decide v' at a time t_4, since it is indistinguishable from Run 2; this violates the *agreement* property.

□

In the rest of the report we will focus only on the case $u > s$, and show that consensus is indeed solvable in the Visigoth model with $u+o+s+1$ replicas. This not only has the advantage of significantly simplifying the notation and the proofs (since the $u \leq s$ case needs to be handled separately), but also focuses on the interesting case, where the Visigoth model is advantageous. Furthermore, the $u \leq s$ case is very similar and leads to the same solutions as in the well-studied asynchronous setting, since one cannot in general improve the algorithms by inferring how many processes have crashed after a timeout

In addition, we will focus on a particular primitive of consensus called *abortable epoch consensus*. We do so because, in the main paper that describes the model, we have presented an adaptation of a state machine replication library for the Byzantine model named BFT-SMaRt, which at its core relies on a concrete implementation of this particular consensus primitive. Furthermore, once state machine replication is solved using this primitive [1], it is straightforward to perform a reduction to solving consensus. Therefore the following algorithms (and proofs) show not only the tight upper bound for solving consensus in a VFT system, but also show that the correctness of the implementation presented in the main document is correct.

3 Upper bound: Preliminaries

Now we shift our focus to the upper bound result for the case where $s < u$ and therefore a VFT system has $n > u + o + s$ processes. First, we state some important definitions, which are closely related to the primitives that we will introduce in the following sections.

We start by providing the definition of a *Quorum*, which is a central concept to both our primitives and our proofs. We leverage the notation introduced by Cachin et al. [1] in this definition.

Definition 1. *A* quorum *is a vector M of size n indexed by process number p, where for any process p, $M[p] = \text{UNDEFINED} \vee m$. Here, m is a message such that any entry in M different from* UNDEFINED *was generated and sent by process p; i.e.,*

$$M : p \in P \mapsto \{\mathit{UNDEFINED}\} \cup \mathcal{M} \tag{1}$$

where $\mathcal{M}$ is the set of valid messages a process p may send.

We denote the number of entries in a quorum M that have a value different from UNDEFINED by $\#(M)$, referred to as the size of quorum M. Furthermore, we will use the following terminology when referring to quorums. If $M[p] \neq \text{UNDEFINED}$, we say that p has participated in quorum M, conversely, if $M[p] = \text{UNDEFINED}$ we say that p was excluded from quorum M.

A quorum system [3] is a non-empty set of subsets of the set of processes (where each subset is a quorum), which defines intersection properties that are of interest to different algorithms that make use of those systems. In our context, we will make use of *dissemination quorum systems* [3], which ensure that any two quorums intersect in at least one correct process.

A quorum M of size n can be parameterized by a predicate condition $C(\cdot)$, which is an efficiently computable Boolean function on a quorum:

$$C : M \in [p \in P \mapsto \{\text{UNDEFINED}\} \cup \mathcal{M}] \mapsto \{\text{true, false}\} \tag{2}$$

We introduce a functions CP on a quorum M and a predicate C that outputs the largest subset quorum ($M' \subseteq M$) such that $C(M')$ is true. We call such an output quorum M' as a *conditioned* quorum. Note that the predicate is expected to be specified at the time of the primitive initialization and cannot be changed afterward. Moreover, every correct process must specify the same predicate condition.

Additional assumptions. We assume that processes have access to a public key infrastructure (PKI) that enable them to sign the contents of messages, and verify the signatures by others.

In all algorithms presented in this document, we assume that messages exchanged among processes are tagged with the unique identifier of the protocol instance with which they are associated. Furthermore, we assume that processes simply ignore any received message that is tagged with the identifier of a protocol instance different from the one that is running. For clarity of exposition we omit these identifiers from the pseudo-code of our algorithms.

Pseudo-code Notation. We first would like to provide some quick notes concerning the notation used in the pseudo-code throughout the document. The interfaces used by the primitives are denoted by events tagged with the identifier of the primitive (e.g. Q-GP), followed by the name of the input or the output, and then arguments. The transmission and reception of messages among processes is captured through the use of special events named $Send$ and $Deliver$, in which the first argument is the identifier of the destination or sender process respectively, and the third argument is the transmitted message.

4 Quorum Gathering Primitive (Q-GP)

4.1 Specification

The quorum gathering primitive (Q-GP) has the following specification.

Definition 2 (Quorum Gathering Primitive)**.** *A Quorum Gathering Primitive (Q-GP) is parameterized by a gatherer process g, a predicate C and a timeout value T_{QGP}. This primitive takes as input at each process p a value m and outputs, at most once, a quorum M at the gatherer, guaranteeing the following properties:*

- *Liveness: if the gatherer is correct and a quorum M_c can be formed by input values from all correct processes such that $C(M_c)$ holds, then the primitive eventually returns a quorum M such that $C(M)$ holds.*
- *Integrity: if the gatherer is correct and delivers M such that some $M[p] \neq$ UNDEFINED and p is correct then p has value $M[p]$ as an input.*
- *Intersection: If all correct, non-crashed processes start to execute the Q-GP protocol within a maximum δ time window such that $T + \delta < T_{QGP}$, then the quorums output by this primitive form a dissemination quorum system, i.e., if two correct processes p and p' gather quorums M and M', then M and M' intersect in at least one correct replica.*

4.2 Q-GP Algorithm

Figure 1 presents our protocol for the Quorum Gathering Primitive (Q-GP) under the Visigoth model. As discussed previously, the protocol is parameterized with a gatherer process g, a condition C, and a timeout value T_{QGP}. Here, the gatherer process g is the only one that initializes the primitive, and this process does not send explicit requests for input values to other processes. Instead, all processes are expected to trigger an input event on this instance, passing as arguments a value m, and a protocol-specific proof, to justify the value m.[1] Additionally, some protocols might not require a proof to validate the quorum M returned by the primitive, and in this case the proof argument on the input event can be vacuously verified to true. The primitive outputs (only at the gatherer) a conditioned quorum of values and a vector of proofs. The vector of proofs includes all proofs that were sent by all processes.

As mentioned, for the intersection property to be met, T_{QGP} must be greater than $T + \delta$, where δ is an upper bound on the time between the moment when the gatherer instantiated Q-GP and the moment when any non-faulty process instantiated Q-GP. Evidently, the concrete value for T_{QGP} depends on various aspects, namely the potential delays that might be introduced by the steps of a concrete protocol that precede the moment when a process can input its value and therefore start this protocol. We will revisit this issue in the context of our consensus algorithm in the next sections.

[1]In our usage of this primitive, this proof in some cases is a signature of m by process p that produces it, and in other cases it is a collection of messages proving that some previous condition in the protocol was met, for instance that a given number of messages were received by that process to justify its current local state in the execution, enabling p to issue m as input.

```
upon event ⟨ Q-GP, $Init$ ⟩ do // only $g$ executes this
    messages ← [UNDEFINED]$^n$; proofs ← $[\perp]^n$;
    verification ← $FALSE$;
    previousMessages ← $\perp$; previousProofs ← $\perp$;
    myInput ← $\perp$; myProof ← $\perp$;
    gathered ← $FALSE$;
    quorumSize ← $n - s$;
    trigger start timer ⟨ Q-GP-Timer, $T_{QGP}$ ⟩;

upon event ⟨ Q-GP, $Input, m$ ⟩ do
    myInput ← $m$;
    myProof ← Proof($self, self||Input||m$);
    trigger ⟨ $Send$, $g$, [QSEND, myInput, myProof] ⟩;

upon event ⟨ $Deliver$, $p$, [QSEND, $m$, proof ] ⟩ do
    if validateProof $\langle p, p||INPUT||m, proof\rangle$ then
        messages[$p$] ← $m$; proofs[$p$] ← proof;

upon timer ⟨ Q-GP-Timer ⟩ do
    quorumSize ← $n - u$;

upon $\#(CP$(messages,$C$)) $\geq$ quorumSize $\wedge$ gathered $= FALSE$ then
    if verification $\neq TRUE$ $\wedge \#(CP$(messages,$C$)) $< n - s$ then
        previousMessages ← messages; previousProofs ← proofs;
        messages ← [UNDEFINED]$^n$; proofs ← $[\perp]^n$;
        quorumSize ← $n - s$;
        verification ← $TRUE$;
        forall $q \in P$ do
            trigger ⟨ $Send$, $q$, [QREQ,] ⟩;
        trigger start timer ⟨ Q-GP-Timer, $2 \cdot T$ ⟩;
    else
        gathered ← $TRUE$
        trigger stop timer⟨ Q-GP-Timer, $\perp$ ⟩;
        trigger ⟨ Q-GP, [Q-GP-Delivery, $CP$(previousMessages $\cup$ messages,C), previousProofs $\cup$ proofs] ⟩;

upon event ⟨ $Deliver$, $p$, [QREQ] ⟩ do
    trigger ⟨ $Send$, $p$, [QSEND, myInput, myProof] ⟩;

//This is an optimization
upon verification = True $\wedge$ ($\#(CP$(messages,C)) $\geq$ quorumSize $\vee$ $\#(CP$(previousMessages $\cup$ messages),C) $\geq n - s$) then
    trigger ⟨ Q-GP, [Q-GP-Delivery, $CP$(previousMessages $\cup$ messages,C), previousProofs $\cup$ proofs] ⟩;
```

Figure 1: The Q-GP Protocol

The implementation of the primitive works as follows. When the primitive is initialized by process g the timer configured with the value described above is also initialized. At this point the gatherer attempts to collect $(n - s)$ values from other processes. However, and due to the timing assumptions of the Visigoth model, if this timer expires then the target quorum size is reduced to $n - u$. Thus, the target quorum size can be achieved in two situations: (1) either the timer has not yet expired, and the target quorum size is still $n - s$, in which case the gatherer immediately returns from the primitive through a Q-GP-Delivery upcall, which delivers both the gathered quorum, and the corresponding proofs; or (2) the timer had expired and the current quorum of values gathered has a size smaller than $n - s$.

In the second case, we start a *verification phase*, which is denoted by a local boolean variable *verification*. The gatherer drives the verification round by explicitly requesting all process to resend their input values (through the $QREQ$ message). It ensures that no crashed processes are included in the returned quorum, since this could lead the gatherer to make incorrect assumptions concerning the number of crashed processes, which could lead to a violation of the intersection property among concurrent quorums gathered through this primitive.

An Illustrative Example for the Verification Phase. We illustrate how this problem could arise through the following example. Consider a Visigoth System composed of $n = 4$ processes $\{a, b, c, d\}$ with $u = 2$, $o = 0$, and $s = 1$. Consider that processes a and b initiate two distinct instances of Q-GP. They do not communicate with each other, as a perceives b to be slow and symmetrically b perceives a to be slow. Assume that process c participates in the Q-GP instance of process a and then crashes (before being able to participate in the instance initiated by process b), while process d is able to participate in the Q-GP instance of process b and then crashes (without participating in the instance of process a). Also process a is able to participate in its own instance of Q-GP, while similarly process b is able to participate in its own instance.

While the scenario described above is allowed by the concrete configuration of this Visigoth system, in this case neither a nor b are able to gather $n - s = 3$ values in their quorums, therefore the timer in each process will expire. At this point both processes will be able to infer that at least one process has crashed (as more than s values were not received before the expiration of the timer), and will therefore readjust their target quorum size to $n-u = 2$ which should represent a dissemination quorum, and therefore proceed with their gathered quorums. This happens because both a and b are unable to detect that the process that crashed (from their perspective) may have participated in a concurrent instance of the Q-GP. Additionally, they are unable verify if some process that participated in their quorum has crashed meanwhile, and is therefore unable to propagate the information of its own quorum to any other concurrent instance. To circumvent this, the extra verification round is incorporated to allow both a and b to validate their current quorum so that this situation is ruled out. For example, the case described above, both processes a and b will have to wait for the participation of each other in their respective Q-GP instances.

In this case, the set of values and proofs collected previously is stored in local variables, the target quorum size is reset to $n - s$, the $QREQ$ message is transmitted to all processes and the timer is also reset. However, this time the timer value is set to $2T$ to accommodate the necessary transmission time of the $QREQ$ messages. From this point on, the gatherer restarts the process of collecting values and proofs, until one of two situations occurs. Either a quorum of the target size (either $n - s$ or $n - u$ depending on whether the second timer has expired or not) is collected and its union with the previously gathered set is returned at this time through the Q-GP-Delivery upcall, or, if the combination of values gathered in the previous execution and the current execution has a size larger or equal to $n - s$, then the combination of both these sets is returned through the same upcall.

4.3 Proof for the Q-GP Protocol under Visigoth model

We start by proving a helper lemma stating that whenever a gatherer g assemble, during any of the this of the protocol, a quorum of size $n - z$ where $s < z \leq u$, this implies that at least $z - s$ processes have failed before the moment when the quorum was assembled.

In the following Lemma, we define the *first phase quorum* as the quorum of messages present in the "messages" variable when the action that sets the "verification" variable to True starts. We also refer to the moment when that action is triggered as the *transition* time.

Lemma 3. *In the Q-GP protocol (Figure 1), assume that a quorum M_c can be formed by input values from all correct processes such that $C(M_c)$ holds. If a correct gatherer g collects a first phase (resp. final) conditioned quorum M of size $n - z$ with $s < z \leq u$, then at least $z - s$ processes have failed before the transition time (resp. the end of the execution of the primitive).*

Proof. By the protocol construction and the properties of the Visigoth model, out of the $z > s$ processes that are not in M, at most s processes are slow processes that can take longer than the timeout to participate in the quorum. This implies that at least $z - s$ processes have crashed before being able to participate; in particular, they have crashed at the latest right before the transition time or the end of the execution of the primitive. □

We now prove the liveness, integrity and intersection properties of the Q-GP protocol.

Theorem 2. *Liveness: if the gatherer is correct and a quorum M_c can be formed by input values from all correct processes such that $C(M_c)$ holds, then the Q-GP protocol instance eventually returns a quorum M such that $C(M)$ holds.*

Proof. By the protocol in Figure 1, a conditioned quorum M composed of $n - u$ values is enough to ensure the termination of the algorithm. By the assumptions of the model, there are at most u total failures. Given that the remaining processes are correct, and input correct values, then eventually the gatherer will be able to collect $n - u$ values from different processes twice (this is required due to the need of executing the verification round), which is enough to return that quorum. □

Theorem 3. *Integrity: In the Visigoth Quorum Gathering Primitive, if the gatherer is correct and delivers M such that some $M[p] \neq$ UNDEFINED and p is correct then p has value $M[p]$ as an input.*

Proof. By the protocol in Figure 1, all entries in a quorum M are initially set with a value of UNDEFINED, and if the gatherer is correct, an entry $M[p]$ can only have its value modified to some value m through the reception of a message $QSEND$ issued by p containing an input value of m. As the model prescribes authenticated point-to-point links, if p is correct, then for g to receive a value m from p implies that p has used that value as its input. □

Theorem 4. *Intersection: If all correct, non-crashed processes start to execute the Q-GP protocol within a maximum δ time window such that $T + \delta < T_{QGP}$, then the output quorums form a dissemination quorum system, i.e., if two correct processes p and p' gather quorums M and M', then M and M' intersect in at least one correct replica.*

Proof. By the protocol in Figure 1, the quorums that are returned can either have a size $Q1 = n - s$ or $n - u \leq Q2 < n - s$. The latter is returned only after a timeout of T_{QGP} and the execution of a verification round. We have to show that $i)$ any two quorums of dimension $Q1$ intersect in at least one correct process; $ii)$ any two quorums of dimension $Q2$ intersect; and finally, $iii)$ that a quorum of dimension $Q1$ intersects with another quorum of dimension $Q2$.

We start by the first case: any two quorums of dimension $Q1$ intersect. A quorum of dimension $Q1$ has $n - s$ processes, so it enough to show that $n - s + n - s > n + o$ to ensure the intersection on one correct process. Assuming the most unfavorable case of $n = u + o + s + 1$, this can be written as $2u + 2o + 2 > u + 2o + s + 1$, which simplifies to $u + 1 > s$. As we are considering the case where $u > s$, this condition holds.

Next we consider the second case where the two quorums have size $Q2$. The protocol that gathered each quorum ran two consecutive rounds, and therefore it must be that case that the first phase of one of the quorums must finish before the *verification phase* of the other quorum begins. Assume, without loss of generality, that $Q1$'s first phase finishes before $Q2$'s *verification phase* starts. Call the quorum gathered at the first phase of $Q1$ "$q1$", and generically write their sizes as $q1 = n - u + a$, and $Q2 = n - u + b$.

At the instant when $q1$ ends, by Lemma 3, it is certain that at least $u - a - s$ processes have crashed (all non-responsive ones except s slow processes). This implies that the system size at that instant is $n' =$

$n-u+a+s$. Given this system size, the intersection in one correct process is guaranteed if $q1+Q2+n' > o$. This gives:

$$\begin{aligned} q1+Q2-n' &> o \\ n-u+a+n-u+b-n+u-a-s &> o \\ u+o+s+1-u+a-u+b+u-a-s &> o \\ 1+b &> 0 \end{aligned}$$

Finally, we prove the third case using an argument similar. If $Q1 = n-s$ and $Q2 = n-u+a$, then by Lemma 3 at least $u-a-s$ processes have crashed during or before the execution that gathered Q2, and therefore the number of processes left are $n' = n-u+a+s$, and the intersection requires

$$\begin{aligned} Q1+Q2-n' &> o \\ n-s+n-u+a-n+u-a-s &> o \\ u+o+s+1-s-u+a+u-a-s &> o \\ u-s+1 &> 0, \end{aligned}$$

which is true since we are considering the case when $u > s$.

□

5 Conditional Collect Primitive

The purpose of the *conditional collect* primitive, is to collect information in the system, in the form of values from other processes, *in a consistent way*. The abstraction is initialized by every process (through an Init event) and then a distinguished process called *leader* collects a quorum M of values using the Q-GP introduced previously. When this quorum is gathered by the leader, this event is disseminated to all processes, and that quorum (after local validation) is returned through an event $\langle COLLECTED$,M,$proof\rangle$ at every process. In this quorum, $M[p]$ is either equal to UNDEFINED or corresponds to the input message of process p. The proof that is output together with the vector M is a corresponding vector of signatures over each non-undefined element of M, where each signature was produced by the process that had that message as input. A conditional collect primitive must collect the same vector of messages at every correct process such that this vector satisfies some (static) condition C, which, similarly to the one employed in Q-GP, is materialized by a boolean operator that can be employed both on individual values or a quorum of values. Correct processes must all input messages that satisfy the condition, otherwise this goal cannot be achieved.

5.1 CCP Specification

The Conditional Collect Primitive is parameterized by a distinguished leader process l, a timeout value T_{CC}, and a condition C.

CCP takes as input, at each process, a value m. It outputs, at each process, a quorum M of values, and a "proof" consisting of either $n-s$ signatures over the values in M, or a smaller number z of signatures ($n-u \leq z < n-s$) in which case it must also return at least $o+1$ signed values vouching for the non-reachability of processes not in M.

The specification of the primitive has the following correctness properties (closely following the properties from the original BFT version of the primitive [1]).

Definition 4 (Conditional Collect Primitive)**.**

Consistency *If the leader is correct then every correct process collects the same M, and this M forms a dissemination quorum system with all quorums returned by any other instance of the primitive in the execution.*

Integrity *If some correct process collects M with $M[p] \neq$ UNDEFINED for some process p and p is correct, then p has input message $M[p]$.*

Termination *If all correct processes input compliant messages and the leader is correct, then every correct process eventually collects some M such that $C(M) =$ True.*

5.2 Signed Conditional Collect Algorithm

Figure 2 present a protocol for CCP in the Visigoth model. This algorithm is based on the Signed Conditional Collect algorithm presented in [1], originally proposed for the Byzantine model, but adapts it substantially due to the need to validate if the leader has used the appropriate quorum size.

The algorithm presented above is governed by a leader l and relies on the previously presented Q-GP primitive. We note that in the pseudo-code we denote the operation of concatenation through the use of the $||$ operator. The algorithm structure is relatively simple: the leader l initializes a Q-GP instance, and all processes input to that Q-GP instance a value m, which is passed in the initialization of the current conditional collect instance. (This value m is intuitively encoding a view of the system according that process, which is being aggregated with that of other processes in a quorum and gathered by the conditional collect algorithm.) When the Q-GP primitive returns at the leader, the leader l first signs the gathered quorum and corresponding proofs (i.e., the Messages vector) and sends the resulting quorum, the corresponding collected proofs, and its signature to all processes through a COLLECTED message.

Upon receiving this message each process verifies the signature of the leader over the set of values, and then for each value in the quorum that is different from UNDEFINED, it verifies the signature of the process that generated the value. If all these checks pass, the process verifies if the quorum disseminated by the leader has a size of at least $n - s$ (i.e., at least $n - s$ values different from UNDEFINED). In this case the process delivers the set of values and the signature of the leader through the Collected event (while also setting the boolean local variable *collected* to true, ensuring that no other set will be delivered). This happens because, as discussed previously, any quorum with a size of at least $n - s$ will form a quorum dissemination system with all other quorums gathered by a Q-GP primitive.

However, if the quorum disseminated by the leader has a size of at least $n - u$ but smaller than $n - s$, processes initiate a verification phase. The goal of this phase is to ensure that an incorrect leader (i.e., one that suffers a commission fault) does not wrongly provide a quorum that is too small, thus not ensuring the intersection property with all other quorums in one correct replica. To this end, any process that receives such a quorum from the leader does the following: i) it stores all information carried by the COLLECTED message issued by the leader on a local tuple named preCollected; ii) it instantiates a Q-GP instance; iii) it inputs its value and proof on the Q-GP instance that is initialized by each process for the verification. (In other words, there is an all-to-all broadcast formed by various Q-GP instances.) Processes that were not contacted by the leader and receive a message from one of these Q-GP instances broadcast their signed value so that they attempt to forward their input to the leader.

At this point it is possible to perform an early detection of a commission faulty leader (and consequently stop the execution of the primitive) in case the leader provably had no reasons to propose a quorum smaller than $n - s$. This is done by verifying that a quorum gathered by each process has a size that differs from the proposed quorum in more than s (which would be disallowed by the model). In such a case, it implies that the leader is not correct, and hence the process will not proceed with the execution of the CC primitive.

Otherwise, then the leader instantiates a new Q-GP instance (Q-GP-CHECK) and all processes input in this instance the quorum gathered by them (and proofs) for the verification phase.

```
upon event ⟨ $CC, Init, m$ ⟩ do
    collected ← FALSE; preCollected ← $\perp$;
    value ← $\perp$; proof ← $\perp$;

upon event ⟨ $CC, Input, m$ ⟩ do
    if self = leader do
        trigger ⟨ $Q-GP, Init$ ⟩; //only leader $l$ executes this
    value ← $m$; proof ← sign(self, self$|m$);
    trigger ⟨ $Q-GP, Input, m, proof$ ⟩;

upon event ⟨ $Q-GP$, [Q-GP-Delivery, Messages, Proofs] ⟩ do
    $\sigma$ ← sign(self, self||Messages||Proofs);
    forall $q \in \Pi$ do
        trigger ⟨ Send, $q$, [COLLECTED, Messages, Proofs, $\sigma$, $\perp$] ⟩;

upon event ⟨ Deliver, $p$, [COLLECTED, M, S, $\sigma$, Verification] ⟩ do
    if collected = FALSE $\wedge$ verifysign($l$, $l$||Messages||Proofs, $\sigma$) $\wedge$
        (forall $p \in \Pi$ such that M[$p$] $\neq$ UNDEFINED, it holds
            verifysig($p$, vcc||$p$||$INPUT$||M[$p$], S[$p$]) then
        if #(M) $\geq n-s$ then
            collected ← TRUE; trigger ⟨ CC, [CC-Collected, M, {S,$\sigma$}] ⟩;
        else
            if collected = FALSE $\wedge$ Verification = $\perp$ $\wedge$ preCollected = $\perp$ then
                forall $q \in \Pi$ do
                    trigger ⟨ Send, $q$, [CC-VERIFY, {M, S, $\sigma$}] ⟩;
            else if Verification $\neq \perp \wedge \exists$ at least $o+1$ quorums $Q$ in
                Verification such that || $\#(M)$ - $\#(Q)$ || $\leq s$ then
                collected ← TRUE; trigger ⟨ CC, [CC-Collected, M, {S,$\sigma$}] ⟩;

upon event ⟨ Deliver, $p$, [CC-VERIFY, proposeCollect] ⟩ do
    if preCollected = $\perp$ $\wedge$ CHECK(proposeCollect)then
        preCollected ← proposeCollect; trigger ⟨ $Q-GP[self]$, $Init$ ⟩;
        forall $q \in \Pi$ do
            trigger ⟨ Send, $q$, [CC-VERIFY, proposeCollect] ⟩;
            trigger ⟨ $Q-GP[q]$, $Input$, value, proof ⟩;

upon event ⟨ $Q-GP[self]$, [Q-GP-Delivery, Messages, Proofs] ⟩ do
    if self = leader do
        trigger ⟨ $Q-GP-CHECK, Init$ ⟩; //only leader $l$ executes this
    if | #(preCollected) - #(M)| $\leq s$ then
        trigger ⟨ $Q-GP-CHECK, Input, Messages, Proofs$ ⟩;

    upon event ⟨ $Q-GP-CHECK$, [Q-GP-Delivery, Messages, Proofs] ⟩ do
        forall $p \in Pi$ such that Messages[$p$] $\neq$ UNDEFINED
            $M$ ← Messages[$p$]; $S$ ← Proofs[$p$];
            forall $p \in \Pi$ such that M[$p$] $\neq$ UNDEFINED, where it holds
                verifysig($p$, $M[p]$, $S[p]$) do
                if preCollected[1][p] = UNDEFINED then
                    preCollected[1][$p$] ← $M[p]$;
                    preCollected[2][$p$] ← $S[p]$;
        $\sigma$ ← sign(self, self||preCollected[2]||preCollected[2]);
        forall $q \in \Pi$ do
            trigger ⟨ Send, $q$, [COLLECTED, $M$: preCollected[1], $P$: preCollected[2], $\sigma$, Proofs] ⟩;
```

Figure 2: Pseudo code for Visigoth Signed Conditional Collect.

When this instance returns at the leader, it resends the COLLECTED message to all processes, but before includes in its previous quorum any message that it previously missed that was forwarded to it during the verification phase. Also, a set of at least $o + 1$ confirmations gathered during the verification phase is disseminated in this COLLECTED message. Upon receiving this second COLLECTED message each process will do the following. Either the proposed quorum has a size of at least $n - s$ and the instance of CC returns that quorum (and corresponding signatures, which form a proof), or the quorum has a size smaller that $n - s$, in which case the process verifies if there are at least $o + 1$ processes vouching that the size of that quorum is valid, i.e., vouching for a current system membership that is at least a subset of the view of the system in the quorum collected by the leader. In the latter case, that quorum, alongside the $o + 1$ validation messages, constitute a proof.

Timer configuration. As discussed before, the Quorum Gathering Primitive (Q-GP), which is used by this protocol, requires the configuration of a timer, which is the parameter T_{QGP}. This primitive is central to the design to the Conditional Collect Primitive, and, as expected, the use of Q-GP in different steps of the CC algorithm has to consider different values of the timer.

In particular, the CC algorithm relies on Q-GP on 3 different steps of the algorithm:

The initial Q-GP used by the leader: In this case the timeout should be configured to have a time equal to the parameter T_{CC}. In other words, we delegate to the client algorithms that make use of this Conditional Collect the computation of the expected maximum delay between two processes triggering the Init event of CC, and this is passed directly to the first step of the protocol, which is an invocation if QGP.

The Q-GP instances used by each process to verify small quorums proposed by the leader: In this case each process should configure the visigoth timer to have a value of $2T$. This happens because some processes might not receive the COLLECTED message of the leader in a timely fashion, and hence will only start the execution phase upon receiving the first CC-Verify message.

The Q-GP-CHECK instance used by the leader to gather verifications of other processes for small quorums: should be configured with a timeout of $3T$, where $2T$ serves to accommodate potential delay in the previous execution step, and T allows all processes to timely communicate with the leader.

5.3 Proof for Signed Conditional Collect Algorithm under the Visigoth model

We now prove the properties of the previously presented implementation of the Conditional Collect primitive under the visigoth model. These proofs highly rely on the properties of the Quorum Gathering Primitive previously introduced.

Theorem 5. *Consistency: if the leader is correct, all correct processes input compliant messages, and all correct, non-crashed processes start to execute the protocol within a maximum δ time window such that $T + \delta < T_{CC}$ then every correct process collects the same M, and this M forms a quorum dissemination system with all quorums returned by any other instance of the primitive in the execution.*

Proof. With a correct leader, this property follows directly from the liveness (Theorem 2) and Intersection (Theorem 4) properties of the Quorum Gathering Primitive and by construction of the conditional collect algorithm introduced earlier. However, we need to consider the intersection with any quorum returned by another instance of the primitive, including those with a faulty leader. If that is the case, then this follows from Lemma 5, which we present next. □

Theorem 6. *Integrity: If some correct process collects M with $M[p] \neq$ UNDEFINED for some process p and p is correct, then p has input message $M[p]$.*

Proof. This property follows from the Integrity property (Theorem 3) of the Quorum Gathering Primitive and by construction of the conditional collect algorithm introduced earlier. □

Theorem 7. *Termination: If all correct processes input compliant messages and the leader is correct, then every correct process eventually collects some M such that $C(M) =$ True.*

Proof. This property follows directly from the liveness property (Theorem 2) of the Quorum Gathering Primitive and by construction of the conditional collect algorithm introduced earlier. □

The properties above all require the leader to be correct. However it is also relevant to show that even if the leader is incorrect, any quorum M returned by a process will still form a quorum dissemination system with any other quorum returned by any other CC instance. To this end we introduce the following additional lemma.

Lemma 5. *Extended Consistency: If all correct processes input compliant messages and all correct, non-crashed processes start to execute the protocol within a maximum δ time window such that $T + \delta < T_{CC}$ and if a correct process returns from a CC instance, then the returned quorum M forms a dissemination quorum system with all quorums returned by any other quorum M returned by another CC instance (or any quorum gathering primitives).*

Proof. There are two scenarios that have to be considered.

The first case is when the leader is correct, and in this case by theorem 5 then the quorum M returned by a CC instance by any correct process forms a dissemination quorum system with other CC quorums and with all other quorums returned by quorum gathering primitives.

The second case is when the leader is not correct, the leader might suffer a commission fault and disseminate a quorum with a size below the appropriate size given the number of crashed nodes at the time, which would violate the intersection property of the quorum gathering primitives. However, if the leader tries to disseminate a quorum below the appropriate size, then every correct process will execute a verification step where all participants in the returned quorum must validate the composition of the quorum, i.e., that no other reachable processes exist. Since this verification step only succeeds when $o + 1$ processes validate that the set of reachable node is at least a subset of the quorum that is returned, and at least one of those processes is correct, then the view of the system that is expressed in the quorum matches the view of the system of a correct process, which suffices to ensure the required intersection, for the same reasons that are given in the case of a correct leader.

□

6 Epoch Consensus Primitive

We now show how we leverage the previously introduced primitives to adapt an existing algorithm that implements the abortable epoch-consensus primitive in a Byzantine system. We refer the interested reader to [1] for a complete description of the original primitive and its operation.

This abortable epoch-consensus operates by having processes evolving through a set of increasing consensus epochs, labelled with a monotonically increasing timestamp ts, where a deterministically selected leader l attempts to achieve consensus among all processes on a given value v. In each consensus epoch ts, the leader is responsible for verifying if some value v was already agreed on a previous instance $ts' < ts$, and then propose that value v if it exists, or some other value v' proposed by a process. Each instance terminates by having processes outputting an ep-decide event, or aborting in which case a new leader is selected and a new consensus epoch instance starts. When a process aborts it outputs an Abort Event, which ensures that no decision will be output in that instance.

This primitive has the following key properties (we borrow the properties provided by the authors in [1] with trivial adaptations for the Visigoth model):

6.1 ECP Specification

Definition 6 (Epoch consensus primitive)**.**

Lock-in: If a correct process has ep-*decided v in an epoch consensus with a timestamp $ts' < ts$, then, no correct process* ep-*decides a value different from v.*

Agreement: No two correct processes ep-decide differently.

Validity: If a correct process ep-decides v, then v was proposed by some leader l' of some epoch consensus with timestamp ts' such that $ts' \leq ts$ and leader l'.

Termination: if the leader l is correct, has proposed a value, and no correct process aborts the epoch consensus primitive, then every correct process eventually decides some value.

Abort Behavior: If a correct process aborts an epoch consensus instance, then the abort eventually completes and that process does no ep-decides on that instance; moreover, a correct process completes an abort only if the epoch consensus has been aborted by some correct process.

Integrity: Every correct process ep-decides at most once.

6.2 ECP Algorithm

```
upon event ⟨ $VEP$, $Init$, epochstate ⟩ do
    ($valts, vwriteset$) ← epochstate;
    decided ← False;
    trigger ⟨ CC, Init, ⟩;

//Read Phase
upon event ⟨ $VEP$, $Propose, v$ ⟩ do $//onlyleader$
    if $val = \perp$ then
        $val \leftarrow v$;
    forall $q \in \Pi$ do
        trigger ⟨ send, $q$, [PROPOSAL, $v$] ⟩;

upon event ⟨ receive, $l$, [PROPOSAL, $v$] ⟩ do
    if $val = \perp$ then
        $val \leftarrow v$;
    trigger ⟨ CC, Input, [Read] ⟩;

upon event ⟨ $CC$, [$Collected$, states, proof] ⟩ do
    // states[p] = [STATE, ts, v, ws] or states[p] = UNDEFINED
    tmpval ← ⊥;
    if exists ts ≥ 0, $v \neq \perp$ from $S$ such that binds($ts$, $v$, $states$) then
        tmpval ← v;
    else if exists $v \neq \perp$ such that unbound(states) ∧ states[$l$] = [STATE, . , $v$ , . ] then
        tmpval ← v;
    if tmpval ≠ ⊥ then
        if exists $ts$ such that ($ts$, $tmpval$) ∈ $writeset$ then
            $writeset \leftarrow writeset \setminus \{(ts, tmpval)\}$;
        $writeset \leftarrow writeset \cup \{(ets, tmpval)\}$;
        trigger ⟨ Q-GP[$w, self$], Init ⟩;
        forall $q \in \Pi$ do
            trigger ⟨ Q-GP[$w, q$], Input, [Write, $tmpval$], [states, proof] ⟩;

//Write-Phase
upon event ⟨ Q-GP[$w, self$], [Q-GP-Delivery, WriteValues , Proofs] ⟩ do
    if ∄ {[state,$\sigma$], [state',$\sigma'$] ∈ Proofs} such that state≠state' ∧ verifysign($l$, $l$||state, $\sigma$) ∧ verifysign($l$. $l$||state', $\sigma'$) then
        select $v$ such that ∃ $q$, WriteValues[$q$] = [Write, $v$];
        $(valts, val) \leftarrow (ets, v)$;
        trigger ⟨ Q-GP[$a, self$], Init ⟩;
        forall $q \in \Pi$ do
            trigger ⟨ Q-GP[$a, q$], Input, [Accept, $val$], ⊥ ⟩;
    else
        trigger ⟨ VEP, Abort ⟩;

upon event ⟨ Q-GP[$a, self$], [Q-GP-Delivery, AcceptValues , Proofs] ⟩ do
    select $v$ such that ∃ $q$, AcceptValues[$q$] = [Accept, $v$];
    if decided = False then
        decided ← True;
        trigger ⟨ VEP, [EP-Decide, $v$] ⟩;

upon event ⟨ VEP, Abort ⟩ do
    trigger ⟨ VEP, [ Aborted, ($valts, val, writeset$)] ⟩;
    halt;
```

Algorithm 1: Pseudo code for Visigoth Read/Write Epoch Consensus.

Algorithm 1 shows the pseudo-code for a Read/Write Epoch Consensus algorithm which is an implementation of the abortable epoch-consensus for the visigoth model. This algorithm is an adaptation of the algorithm provided by the authors of [1] for the Byzantine Abortable Epoch Consensus primitive. In relation to the original algorithm devised for the byzantine model and introduced in [1], we have mostly adapted the algorithm by mapping all-to-all communication steps through the use of n Q-GP primitives, one instantiated by each process individually. Additionally we also replace the use of the original Conditional Collect primitive implementation proposed by the authors in [1] by our own implementation of the Conditional Collect primitive for the visigoth model. This algorithm shows how processes run an instance of consensus for a particular epoch ets with leader l.

The read phase is executed when a new epoch ets is initiated among the processes. Each epoch ets has a deterministically assigned leader l. External entities can propose a value v to the leader through a Propose event. Note that the start of a new epoch (denoted by an Init event in the algorithm) assumes that correct processes receive the state from the previous epoch, which enables any process starting that epoch to be aware of all relevant decisions made in previous epochs. This local state includes the timestamp for the epoch where the last value was ep-decided ($valts$), which is the epoch identifier of the last epoch where an EP-DECIDED event was delivered; the last decided value (val), which is the value decided in the previous referred epoch, and the write set ($writeset$), which is a quorum of messages gathered in the previous epoch, which encodes the progress of the protocol in that instance. The algorithm is essentially organized in two consecutive phases, a read phase and a write phase, which on their own form the basic structure of an epoch, while epochs succeed after each other, with the execution of a view change protocol to coordinate processes in moving to a different epoch.

Upon receiving a propose event, the leader l will start the read phase, whose goal is to identify any previous value that is already locked-in (meaning that the value might have already been ep-decided by a correct process in a previous epoch). This is achieved by having all processes execute an instance of the Visigoth Signed Conditional Collect Primitive (CC) described previously. In CC all processes input their current state, namely their write set quorum from the previous epoch (gathered by the leader through an instance of Q-GP and then disseminated through authenticated reliable points-to-point links to all processes) and therefore, when the leader is correct, all processes output the same set of states.

All processes then inspect the set of states returned by the CC primitive, and locally decide if there exists some value that was already locked-in in a previous epoch ; if no such value exists, then processes will adopt the value proposed by the leader of the current epoch. In the end of this step, all processes initialize an instance of Q-GP to gather Write statements which are composed of the currently adopted value. Importantly, besides inputing the value as described before, each process also inputs a proof composed by the vector of states and the proof (composed by the vector of proofs for the states collected by the leader, and the leader signature over both the quorum of states and the vector of proofs) that was returned by the instance of the CC primitive.

When the instance of Q-GP gathering Write statements at each process returns, the processes then first verify the proof output by the Q-GP instance. Recall that this proof is a set of state vectors collected by each process that attempted to input a statement on the Q-GP instance which contain the output of the CC instance used in the read phase, as well as the signature of the leader for that output, and a signature of the process itself for both these contents and the value supported by that process on the statement that it inputed on the Q-GP instance. This enables each process to detect a specific behavior of a commission-faulty leader, namely if there are two different state vectors in the proof, for which the signature of the leader l holds. If the leader is considered to be faulty then the process aborts the current instance of consensus by triggering an Abort event (which will trigger a epoch change afterwards).

Otherwise, when there is only a single value in the set of statements collected by the Q-GP instance at each process and the leader is not suspected. In this case, each process will first update its local state to reflect that this value has been written, and then each process will initialize another instance of Q-GP to gather Accept statements from other processes, and, similarly to the previous step, each process will input an Accept statement with this value on the corresponding instance of Q-GP of each other process in the system. When these instances return, a single value will be contained in the set of Accept statements collected by the primitive as processes select in this phase the value that had the highest support in the previous protocol step, leading processes to simply ep-decide that value.

Note that the use of Q-GP instances to gather Write and Accept statements is similar to the typical all-to-all communication pattern employed by PBFT [2] and similar protocols, in which each process sends a message to all other processes (which is replaced by having each process input a statement on the Q-GP instance of other processes), and also each process gathers a set of messages before moving forward in the protocol (which is replaced by having each process wait for the Q-GP instance to return).

6.3 Proof for ECP Algorithm under visigoth model

We now prove that the proposed algorithm for Abortable Epoch Consensus presented above is correct. We do this by focusing on each of the properties of this abstraction and also by leveraging the proofs of the original Byzantine Abortable Epoch Read/Write Consensus algorithm presented by the authors of [1] and discussed previously in this document.

Theorem 8. *Lock-in: If a correct process ep-decided v in an epoch consensus instance with timestamp ts, then no correct process ep-decides a value different from v in any epoch consensus instance with timestamp $ts' \geq ts$.*

Proof. For a process p to decide some value v in an epoch ts, that process will have to have obtained a valid dissemination quorum of *Accept* statements for v through a Quorum Gathering Primitive (Q-GP). The same is true for any correct process to decide a value v' at any epoch consensus instance with a timestamp greater or equal to ts. By the property of intersection of dissemination quorums, there is at least one correct process c that provided an Accept statement for v as an input on that quorum gathered by p on instance ts, and similarly that same process c will also participate in any quorum of *Accept* statements gathered by any other process p' at a later instance $ts' \geq ts$ which implies that any valued decided based on the quorum gathered by p' will have to reflect that v was already decided, and hence any correct process p' will decide value $v' = v$. □

Theorem 9. *Validity: If a correct process ep-decides v, then v was proposed*[2] *by some leader l' of some epoch consensus with timestamp ts' such that $ts' \leq ts$ and leader l.*

Proof. If some correct process ep-decided v on some consensus instance in epoch ts it means that it was able to gather a valid quorum of *Accept* statements for v through an instance of the Quorum Gathering Primitive (Q-GP). According to the intersection property of the Q-GP, at least one correct process has provided an *Accept* statement for v. For this to happen that correct process was also able to previously gather a Write statement quorum proposing value v. For this to happen only two things are possible, either v was proposed by the leader of the epoch consensus instance with timestamp ts and correct processes were able to gather write quorums proposing v on that instance, or v had been proposed by the leader of some previous epoch consensus instance with timestamp $ts' < ts$, where correct processes also gathered write quorums proposing v. In the latter case, by the intersection property on quorum dissemination systems, v will be present on any read quorum gathered by the leader of any epoch consensus instance with $ts > ts'$ which would lead that leader to propose v.

□

[2]Since it is not precisely defined what are the actions of a (commission) faulty process that are considered a proposal, we need to clarify that if a commission faulty leader behaves as a correct leader to a large enough subset of processes to carry through a proposal, then we consider that the faulty leader has proposed a value.

Theorem 10. *Termination: if the leader l is correct, has proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually decides some value.*

Proof. If the leader is correct, and proposes a value and if no correct process aborts the consensus instance, then by the termination property of the Conditional Collect Primitive (CCP), all correct processes will obtain the same set of states as an output of the CCP. Then by the termination property of the Quorum Gathering Primitive (Q-GP), all correct processes will gather a valid quorum of Write statements for some value v, and then by the same termination property all correct process will gather a valid quorum of Accept statements for v. Additionally by the intersection property of Q-GP, the information gathered by every correct process at each step of the algorithm will contain contributions from at least a correct process, which will ensure that processes can make progress in the protocol. Since no process ep-aborts, then all correct processes will ep-decide v. □

Theorem 11. *Abort Behavior: If a correct process aborts an epoch consensus instance, then the abort eventually completes and that process does no ep-decides on that instance; moreover, a correct process completes an abort only if the epoch consensus has been aborted by some correct process.*

Proof. By construction, the algorithm returns an Aborted event containing the local state of the process immediately, and only if it was aborted. □

Theorem 12. *Integrity: Every correct process ep-decides at most once.*

Proof. By construction, a process can only decide as long as its local variable *decided* is set to false, and this variable is set to true just before triggering the ep-decide event. □

Theorem 13. *Agreement: No two correct processes ep-decide differently.*

Proof. There are two cases to be considered here, the first when the leader is correct, and the second when the leader is faulty. For the first case in which the leader is correct, by the property of Consistency of the Conditional Collect Primitive (CCP) all correct processes will output the same set of states, implying that all correct processes will input compliant Write statements with a value v on all Quorum Gathering Primitive (Q-GP) instances used to gather Write statements. By the property of intersection of Q-GP, all returned quorums will intersect in a correct process, implying that all correct processes will also input compliant Accept statements for the same value v in all the Q-GP instances used to gather Accept messages, meaning that all correct processes will eventually decide the same value v. If some process decides v and then the instance is aborted, by the property of lock-in of the Epoch Read/Write consensus, then no other process will decide some value v' such that $v' \neq v$ at some future time.

In the second case in which the leader is faulty, the worst case scenario happens when the leader suffers a commission fault during the execution of CCP resulting in the output of different vectors of states at different processes. Notice however that processes input the vector of states that they outputted as part of the execution of CCP alongside the leader signature of this vector of states in the write QGP instances, as a proof for their statement (i.e, the value they they to agree on). By the intersection property of quorum dissemination systems, any write quorum will allow correct processes to verify that the leader is faulty and issued divergent state vectors to different processes. This implies that correct processes will not continue the execution of the algorithm and hence will not ep-decide. □

References

[1] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition).* Spinger, 2011.

[2] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[3] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 569–578, New York, NY, USA, 1997. ACM.